Type Class

AKA trait, protocol, interface, ...

Brian “601” Russell - 2014 Nov 06, Thu

Environmental Cartoons by Joel Pett

http://www.climateactionreserve.org/wp-content/uploads/2012/08/climatesummit.jpg

Type classes began as a way for Philip Wadler to embarrass the ML community (**`eqtype`**), but turned out to be a great idea anyway.

Trigger Warning: Haskell has an implicit *type* fetish:

- **`class`** is really **type class**
- **`data`** is really **data type**
- **`instance`** is really **kind of a type class instance type**

As they all just make more kinds of types.

Eq

Egalitarianism

Haskell

```haskell
class  Eq a  where
  (==) :: a -> a -> Bool -- should be only ONE colon!
  (/=) :: a -> a -> Bool

data K = K

instance  Eq K  where -- means:  K is Eq compliant, and here is how
  (==) _ _ = False
  (/=) _ _ = False    -- worse then breaking the 2nd Monad law

let o = K

( o == o, o /= o ) -- (False, False)
```

Scala

```scala
class K {
  override def equals (that: Any): Boolean = false } // DTTAH

val o = new K()

println(( o == o, o != o, o eq o )) // (false,true,true)
```

Swift

```swift
class K : Equatable { }

func == (lhs: K, rhs: K) -> Bool { return false }
func != (lhs: K, rhs: K) -> Bool { return false }

let o = K()

println(( o == o, o != o, o === o )) // (false, false, true)
```

** bold italic is library code*

Ordering

	Less is More

Haskell	`data Ordering = LT | EQ | GT`
	
	`class  Eq a => Ord a  where`
	`  compare :: a -> a -> Ordering`
	
	`instance  Ord K  where compare _ _ = LT`
	
	`let k = o`
	
	`( o < k, k < o ) -- (True,True)`

Scala	`object K extends Ordering[K] {`
	`  override def compare (x: K, y: K): Int = -601 } // 1.6 bits used`
	
	`val k = o`
	
	`println(( o < k, k < o )) // (true,true)`

Swift	`extension K : Comparable { }`
	
	`func < (lhs: K, rhs: K) -> Bool { return true }`
	
	`let k = o`
	
	`println(( o < k, k < o )) // (true,true)`

These K won't sort very easily (although it wouldn't matter), but any `Ord a =>` type that implements `compare` properly (e.g. transitivity), will bring order to chaos.

Motivation

I want to use the same function to fix things:

```
fix :: Int -> String
fix i = show (i + 600)

fix :: Float -> String -- ghc: Duplicate type signatures for ‘fix’
fix f = show (100 * f) -- ghc: Multiple declarations of ‘fix’
```

But with a *type* **class**:

```
class  Fixer a  where fix :: a -> String

instance  Fixer Int     where fix i = show (i + 600)
instance  Fixer Float   where fix f = show (100 * f)
instance  Fixer Char    where fix c = c : ['0', '1']
instance  Fixer String  where fix s = s

let i =    1 :: Int
let f = 6.01 :: Float

(fix i, fix f, fix '6', fix "601")

-- ("601","601.0","601","601")
```

Monad

To easily comprehend the mysteries of the Monad, just read the original paper on the topic: **La Monadologie**, Leibniz (1714).

> *"Further, there is no way of explaining how a Monad can be altered in quality or internally changed by any other created thing; since it is impossible to change the place of anything in it or to conceive in it any internal motion which could be produced, directed, increased or diminished therein, although all this is possible in the case of compounds, in which there are changes among the parts. The Monads have no windows, through which anything could come in or go out. Accidents cannot separate themselves from substances nor go about outside of them, as the 'sensible species' of the Scholastics used to do. Thus neither substance nor accident can come into a Monad from outside."*

And tragically, Monad papers haven't gotten any better in 300 years.

<table>
<tr><th></th><th>The Monadology</th></tr>
<tr><td>Haskell</td><td><pre>class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- bind (aka flatMap)

instance Monad [] where
  m >>= f = foldr ((++) . f) [] m

instance Monad Maybe where
  (Just x) >>= f = f x
  Nothing >>= _ = Nothing

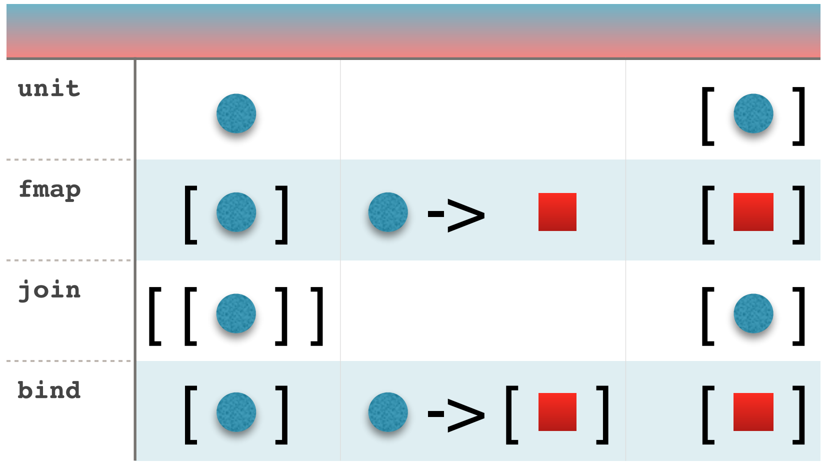
do x <- [-1..1]; y <- [4..6]; [(x * y)]

-- [-4,-5,-6,0,0,0,4,5,6]</pre></td></tr>
<tr><td>Scala</td><td><pre>trait FilterMonadic[+A] extends Any {
  def flatMap[B, That](f: A => GenTraversableOnce[B]): That }

println(for (x <- -1 to 1; y <- 4 to 6) yield x * y)

// Vector(-4, -5, -6, 0, 0, 0, 4, 5, 6)</pre></td></tr>
<tr><td>Swift</td><td><pre>// DIY :-(</pre></td></tr>
</table>

Shapes



Easy to see Monad Laws

	monad composition	≡
Left identity	return >=> *f*	*f*
Right identity	*f* >=> return	*f*
Associativity	(*f* >=> *g*) >=> *h*	*f* >=> (*g* >=> *h*)

The Kleisli composition operator

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

f >=> g = \x -> f x >>= g -- built with bind inside

(m >=> n) x = do { y <- m x; n y }
```