Clases Functor y Monad

Ruben Astudillo

11 de mayo del 2017

Motivacion

El mundo real es mucho mas complicado que lo que se puede hacer con simple composicion de funciones puras.

Idea: ¿Que tanto podemos simular con funciones?

Errores

```haskell
Either a b = Left a | Right b
  deriving (Show)

-- instance Functor (Either a) where ...
-- instance Applicative (Either a) where ...
-- instance Monad (Either a) where ...
```

Errores

```haskell
Either a b = Left a | Right b
  deriving (Show)

-- instance Functor (Either a) where ...
-- instance Applicative (Either a) where ...
-- instance Monad (Either a) where ...
```

Eso no pueden ser funciones! Las funciones en haskell son **puras** y ademas estas utilizando la do-notation. Trampa!

Errores

```haskell
Either a b = Left a | Right b
  deriving (Show)

-- instance Functor (Either a) where ...
-- instance Applicative (Either a) where ...
-- instance Monad (Either a) where ...
```

Eso no pueden ser funciones! Las funciones en haskell son *puras* y ademas estas utilizando la do-notation. Trampa! (respuesta, mira los tipos de *edadEgreso*).

Estado

¿Que *es* el estado? son usualmente variables que estan en el stack que puedo acceder en este scope de subrutinas.

Si haskell no tiene estado implicito, ¿podemos hacerlo explicito?

Estado

```haskell
newtype State s a = State ( s -> (a , s) )
-- No Show instance, how do you show a function?

-- instance Functor (State s) where ...
-- instance Applicative (State s) where ...
-- instance Monad (State s) where ...

runState :: State s a -> s -> (a , s)
runState (State f) estadoInicial = f estadoInicial

put :: s -> State s ()
put newState = State $ \s -> ( () , newState )

get :: State s s
get = State $ \s -> (s , s)
```

Estado

Eso luce como AlgoI/C/D...

Pero es puro haskell por debajo, es decir *puras funciones puras*

Metodo

Todo esto, las funciones que no parecen funciones, que pueden igual componerse de alguna manera, que tienen la do-notation disponible es porque estos *datatypes* implementan las clase *Monad*.

Tipos con Kind de orden superior

Los tipos con orden superior son una forma de decir "funciones de tipos a tipos". Ejemplos se pueden ver en ghci

```
:kind Either -- Either :: * -> * -> *
:kind Maybe
:kind IO
:kind (->)
:kind []
```

`Either` no tiene terminos como tipo, pero `Either a b` si tiene terminos donde `a` y `b` son tipos.

Clases regulares de orden normal

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

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

Primera clase de orden superior: Functor

Functor nos da una forma de acceder y modificar los contenidos de un contexto. Donde un contexto usualmente representa algun efecto computacional como podria ser indeterminismo, IO, estado implicito, exceptions.

Primera clase de orden superior: Functor

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

Primera clase de orden superior: Functor

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

ejemplo util

```
fmap :: (a -> b) -> [a] -> [b]
```

¿Luce como alguna funcion familiar?

Primera clase de orden superior: Functor

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b

-- laws
fmap id = id
fmap f . fmap g = fmap (f . g)
```

Las leyes nos ayudaran a implementar instancias de nuestros types.

Instancia Functor para Maybe

Dado que `Maybe a` esta definido por

```
Maybe a = Just a | Nothing
  deriving (Show)
```

queremos implementar una funcion

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

Instancia Functor para Maybe

```
fmap f (Just a) = Just (f a)
fmap f Nothing  = Nothing
```

cumple la leyes!

Instancia Functor para Maybe

Esta instancia, no cumple las leyes

```
fmap f (Just a) = Just (f a)
fmap f Nothing  = Nothing
```

¿Cual falla?

Instancia de Functor para (State s)

Queremos implementar

```
fmap :: (a -> b) -> State s a -> State s b
```

Aqui el *Functor* es *State s*, ie el contexto.

Instancia de Functor para (State s)

Queremos implementar

```
fmap :: (a -> b) -> State s a -> State s b
```

Aqui el *Functor* es *State s*, ie el contexto.

idea util para implementar

```
id = (\a -> id a)
```

podemos trabajar con los argumentos de una funcion como si los tuvieramos en la mano.

Instancia de Functor para (State s)

```
fmap f (State state) =
     State $ \s -> let (a, s') = state s
                   in  (f a, s' )
```

mostrarlas leyes es mas tedioso, pero mecanico. Idea se basa es que nunca modificamos nosotros el estado `s`.

Monad

Aveces no queremos simplemente modificar el valor dentro del contexto. Aveces queremos que – dependiendo del valor dentro del contexto – modificar el contexto mismo.

Monad

```
class Monad (m :: * -> *) where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

`return` nos da una forma de inyectar valor a un contexto.

`(>>=)`: dado un contexto con valores `a` y una funcion que dependiendo del valor, me genera un nuevo contexto y valores `b` me genera un nuevo contexto "combinado" de valores `b`.

¿En que sentido combinado?. En el sentido de que las instancias respetan las siguientes leyes.

```
return a >>= k  =  k a
m >>= return  =  m
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Monad

Dos formas de implementar estas instancias

1. Podemos reutilizar el trabajo ya hecho con Functor `f :: a -> m b m :: m a fmap f m :: m (m b)` y buscar una forma de "eclosionar" `m (m b)` hacia `m b`. Despues de todo, si son dos instancias de un contexto, podria expresarlo en un simple nivel.

Monad

2. A lo puro y duro, básicamente tomando todos los argumentos y viendo cuales son las únicas formas posibles de combinarlos.

Utilizaremos la primera alternativa para `State s` y la segunda para `Maybe`

Monad

Queremos una funcion que “eclosione” `State s (State s a)` en `State s a`. A esta funcion le llamaremos

```
join :: State s (State s a) -> State s a
```

Monad State

Queremos una funcion que "eclosione" `State s (State s a)` en `State s a`. A esta funcion le llamaremos

```
join :: State s (State s a) -> State s a
join outstate =
     State $ \s0 ->
        let (innerstate, s1) = runState outstate s0
        in  runState innerstate s1
```

Monad Maybe

Veamos los tipos de los argumentos involucrados

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

return a = Just a

Nothing >>= f = Nothing
(Just a) >>= f = f a
```

¿Como funciona IO?

Notemos que `IO` esta definido asi en `System.IO`

```
{-
'IO' is a monad, so 'IO' actions can be combined
using either the do-notation or the '>>' and '>>='
operations from the 'Monad' class.
-}
newtype IO a =
    IO (State# RealWorld -> (# State# RealWorld, a #))
```

Esto es un `State RealWorld`. Por lo tanto puede implementar la do-notation

Conclusion

La clase `Monad` nos da una forma de definir algebraicamente efectos computaciones. Entre ellos se encuentra IO, estado, errores, paralelismo, concurrencia, indeterminismo, backtracking, continuaciones, entorno lectura, logging. Es una buena herramienta a conocer para definir denotaciones de lo que puede significar efectos en algún lenguaje objetivo y para definir pequeñas DSL.