Universidad de Huelva

Práctica 7

Funtores, aplicativos y mónadas

Índice

- Un ***funtor*** es un tipo de datos que se puede mapear, es decir, que envuelve valores sobre los que se pueden aplicar funciones.
- El módulo *Prelude* contiene la clase ***Functor*** que expresa esta propiedad.
- Los siguientes tipos desarrollan la clase Functor:
  - Listas
  - Maybe
  - Either a
  - IO
  - (->) r

- La clase *Functor* contiene una única función llamada *fmap*

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

- La función *fmap* toma como primer argumento una función que transforma un valor de tipo *a* en un valor de tipo *b*, como segundo argumento un funtor *f* que envuelve un tipo de dato *a* y genera un funtor que envuelve un tipo de dato *b*.
- Se dice que *fmap* mapea una función sobre un funtor.

- Las listas son funtores

```
instance Functor []  where
    fmap f [] = []
    fmap f x:xs = (f x) : fmap f xs
```

- El comportamiento de *fmap* consiste en aplicar la función sobre los elementos de la lista. En realidad ya tenemos la función *map* que hace exactamente lo mismo.

```
instance Functor []  where
    fmap = map
```

- El tipo *Maybe* también es un funtor.
- *Maybe* permite expresar una cantidad que puede tener un valor o no. Este tipo se suele usar, por ejemplo, para envolver resultados que pueden contener errores.

```
instance Functor Maybe  where
    fmap f  Nothing = Nothing
    fmap f (Just x) =  Just (f x)
```

- La función de mapeo consiste en aplicar la función al valor indicado en *Just* o mantener *Nothing*.

- El tipo *Either* es un funtor aplicado al segundo argumento.
- *Either* permite expresar una cantidad que puede tener dos posibles valores incluso de tipo diferente. Este tipo se suele usar para expresar cantidades que pueden haber dado error. El primer tipo (indicado por el constructor *Left*) suele indicar el error mientras que el segundo (*Right*) contiene un valor correcto.

```
instance Functor Either a where
    fmap f  (Left x) = Left x
    fmap f  (Right x) =  Right (f x)
```

- El tipo *IO* es un funtor.
- Como hemos visto, *IO* expresa un valor cuya evaluación conlleva una acción de entrada/salida.
- Para mapear una función sobre una acción *IO* se realiza la acción para obtener el valor y, a continuación, se aplica la función para transformar el resultado.

```
instance Functor IO where
    fmap f  action  =  do
        result <- action
        return (f result)
```

- Los árboles también son funtores.
- Un árbol almacena valores de un cierto tipo en las hojas mientras que los nodos intermedios contienen ramificaciones (dos si se trata de un árbol binario). Por ejemplo,

```
data Tree a = Leaf   a   |   Node (Tree a)  (Tree a)
```

- El mapeo consiste en recorrer el arbol aplicando la función a las hojas

```
instance Functor Tree where
      fmap f  (Leaf  x) = Leaf ( f x )
      fmap f  (Node  l  r)  =  Node (fmap f  l) (fmap f r)
```

- El tipo “función sobre r” es un funtor.
- Las funciones las definimos como (*r* -> *a*) en notación infija, pero podemos describirlas en notación prefija como ((->) *r a*).
- En un lenguaje funcional, como Haskell, las funciones se pueden tratar como cualquier otro tipo de dato, luego (->) es un constructor de tipo.

- En este caso, la función *fmap* daría lugar al siguiente tipo:

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

- O, lo que es lo mismo

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

- El mapeo entonces convierte una función (r -> a) en una función (r -> b) por medio de una función (a -> b). Esto no es mas que la composición de funciones.

```
instance Functor (->) r where
    fmap =  ( . )
```

- Para que un tipo de dato *f* pueda considerarse como un funtor correcto debe cumplir dos propiedades o leyes:
- La primera ley es que si mapeamos la identidad no debemos producir cambios

```
fmap  id  = id
```

- Por ejemplo,

```
> fmap id (Just 3)
Just 3
> fmap id [1..5]
[1,2,3,4,5]
> fmap id []
[]
> fmap id Nothing
Nothing
```

- La segunda ley dice que si mapeamos el resultado de una composición de dos funciones sobre un funtor debe devolver lo mismo que si mapeamos una de estas funciones sobre el funtor inicial y luego mapeamos la otra función.

```
fmap (f . g) = fmap f . fmap g
```

- O, dicho de otra forma

```
fmap (f . g) F = fmap f (fmap g F)
```

- Estas dos leyes son propiedades matemáticas que deben cumplir los funtores, pero corresponde al programador verificar que son ciertas para un tipo antes de describirlo como instancia de Functor.
- Ejemplo de un tipo que no cumple estas leyes:

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
instance Functor CMaybe where
  fmap f CNothing = CNothing
  fmap f (CJust counter x) = CJust (counter+1) (f x)
```

- Se trata de una versión de *Maybe* que cuenta el número de mapeos realizado. No cumple la primera ley.

- La función *fmap* :: (*a* -> *b*) -> *f a* -> *f b* podemos interpretarla como una función que toma una función (*a->b*) y un funtor (*f a*) y genera un funtor (*f b*).
- Podemos interpretarla también como una función que toma una función (*a->b*) y genera una función (*f a* -> *f b*). Esto se conoce como "mover una función".
- Por ejemplo, la expresión *fmap (*2)* genera una función que toma un funtor sobre números y devuelve otro funtor sobre números.

- Hasta ahora hemos visto funtores que envuelven valores simples. Sin embargo también podemos usarlos para envolver funciones.
- Por ejemplo, si mapeamos la función (*) sobre el funtor (*Just 3*) obtenemos (*Just ((*) 3)*). Es decir, un funtor que envuelve a la función que multiplica por 3.

```
Prelude> let a = fmap (*) (Just 3)
a :: Num a => Maybe (a -> a)
Prelude> let b = fmap (*) [1,2,3,4]
b :: Num a => [a -> a]
```

- Supongamos que tenemos un funtor que envuelve a una función o a una lista de funciones. ¿Cómo podemos aplicar esas funciones a un dato concreto?
- Es decir, si tenemos (*f (a->b)*) y tenemos un valor *a* ¿Cómo conseguimos aplicar a *a* la función envuelta en el funtor?
- Como tenemos la función envuelta en el funtor, podemos utilizar *fmap* para acceder al contenido del funtor.

- Con *fmap* necesito utilizar una función que tome la función (*a->b*) y le aplique el valor *a*.

```
Prelude> let a = fmap (*) (Just 3)
a :: Num a => Maybe (a -> a)
Prelude> fmap (\f -> f 5) a
Just 15


Prelude> let b = fmap (*) [1,2,3,4]
b :: Num a => [a -> a]
Prelude> fmap (\f -> f 9) b
[9,18,27,36]
```

- ¿Y si el valor de *a* también está envuelto en un funtor?
- Por ejemplo, si queremos aplicar (*Just (*3)*) a (*Just 5*) para obtener (*Just 15*).
- Utilizando *fmap* por sí solo no es posible.
- Podríamos utilizar patrones sobre *Maybe*, pero estaríamos dando una solución para *Maybe*, no una solución genérica sobre funtores.
- La solución es definir una tipo de dato abstracto que amplíe el concepto de funtor para añadir esta funcionalidad.

- La clase *Applicative* está definida en el módulo *Control.Applicative* y define dos métodos que no tienen implementacion por defecto.

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

- La clase *Applicative* se refiere solo a tipos que sean instancias de la clase *Functor*.

- La función *pure :: a -> f a* toma un valor y genera un funtor aplicativo que envuelve a ese valor.
- Esto se puede interpretar como que la función toma un valor y le añade un contexto mínimo (contexto puro) para envolver ese valor.
- Por ejemplo,

```
Prelude> import Control.Applicative
Prelude Control.Applicative> pure 3 :: Maybe Int
Just 3
```

- (Nota: el tipo de dato *Maybe* es un funtor aplicativo)

- La función <*> :: *f (a -> b) -> f a -> f b* es muy parecida a la función *fmap* de los funtores, pero en este caso el primer argumento es una función que ya está envuelta en el funtor.
- La función <*> está definida como un operador infijo asociativo a la izquierda. Esto permite encadenar operaciones.
- El tipo de dato *Maybe* es una instancia de *Applicative*:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

- Ejemplos de uso de *Maybe* en estilo aplicativo:

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude > pure (+3) <*> Just 10
Just 13
Prelude > pure (+) <*> Just 3 <*> Just 5
Just 8
Prelude > pure (+) <*> Just 3 <*> Nothing
Nothing
Prelude > pure (+) <*> Nothing <*> Just 5
Nothing
```

- El operador <*> permite aplicar funciones envueltas en un funtor mientras que la función *fmap* permite aplicar funciones que no están envueltas en un funtor.
- Para utilizar ambas formas en un estilo aplicativo se define el operador <$> como una versión infija de la función *fmap*.

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
  f <$> x = fmap f x
```

- De esta forma podemos escribir

```
Prelude> (+) <$> Just 3 <*> Just 5
Just 8
```

- Las listas también son funtores aplicativos:

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

- La función pure se limita a crear una lista de un elemento.
- La función <*> crea una lista con todas las combinaciones posibles entre los elementos de la izquierda y de la derecha.

```
Prelude> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

- El uso de listas como datos aplicativos permite modelar valores indeterministas.
- Si un cómputo puede generar distintos valores, el resultado se puede expresar como la lista de valores posibles.
- Un error se expresaría como una lista vacía.
- El contexto aplicativo permite operar con este tipo de datos indeterminados

```
Prelude> (+) <$> [1,2,3] <*> [10,20,30]
[11,21,31,12,22,32,13,23,33]
```

- Las acciones de Entrada/Salida son funtores aplicativos:

```
instance Applicative IO where
  pure = return
  a <*> b = do
    f <- a
    x <- b
    return (f x)
```

- Con esta definición estas dos declaraciones son equivalentes

```
myAction :: IO String
myAction = do
  a <- getLine
  b <- getLine
  return (a ++ b)
```

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

- La definición de las listas como funtores aplicativos genera todas las combinaciones posibles.
- Si queremos que el comportamiento sea aplicar la primera función sobre el primer argumento, la segunda función sobre el segundo argumento y así sucesivamente hay que definir un nuevo tipo.
- Con este objetivo se define el tipo *ZipList* en el módulo *Control.Applicative*

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

- Leyes de los funtores aplicativos:
  - pure f <*> x = fmap f x
  - pure id <*> v = v
  - pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
  - pure f <*> pure x = pure (f x)
  - u <*> pure y = pure ($ y) <*> u

- El concepto de funtor nos define un dato con un contexto sobre el que podemos mapear una función manteniendo el contexto.
- La clase de tipos *Applicative* nos permite utilizar funciones normales con esos contextos de forma que los contextos se mantengan.
- De esta forma, los valores *Maybe a* representan cómputos que pueden fallar, [*a*] representan cómputos que tienen varios resultados (cómputos no deterministas), *IO a* representan valores que tienen efectos secundarios, etc.

- Las mónadas son una extensión natural de los funtores aplicativos y tratan de resolver lo siguiente: si tenemos un valor en un cierto contexto, *m a*, ¿cómo podemos aplicarle una función que toma un valor normal *a* y devuelve un valor en un contexto? Es decir, ¿cómo podemos aplicarle una función del tipo *a -> m b*?
- Para ello crearemos una nueva clase con una función que resuelva es cuestión. La función se suele denominar "lazo" y se escribe en notación infija como (>>=).

- La definición de la clase *Monad* es la siguiente:

```
class Monad m where
   return :: a -> m a

   (>>=) :: m a -> (a -> m b) -> m b

   (>>) :: m a -> m b -> m b
   x >> y = x >>= \_ -> y

   fail :: String -> m a
   fail msg = error msg
```

- Para crear una instancia de la clase *Monad* hay que definir las funciones *return* y (>>=).
- La función *return* toma un valor y lo envuelve en el contexto que describe la mónada. Es lo mismo que realiza la función *pure* respecto a los funtores aplicativos.
- La función (>>=) toma un valor monádico (es decir, un valor en un cierto contexto) y lo pasa a una función que toma un valor normal pero devuelve otro valor monádico.

- El tipo *Maybe* es una mónada:

```
instance Monad Maybe where
   return x = Just x
   Nothing >>= f = Nothing
   Just x >>= f = f x
   fail _ = Nothing
```

- Por ejemplo

```
Prelude> Just 9 >>= \x -> return (x*10)
Just 90
Prelude> Nothing >>= \x -> return (x*10)
Nothing
```

- El sentido de la función (>>=) es que permite encadenar funciones manteniendo el contexto.

```
Prelude> let doble x = Just (2*x)
doble :: Num a => a -> Maybe a
Prelude> let triple x = Just (3*x)
triple :: Num a => a -> Maybe a
Prelude> Just 5 >>= doble >>= triple
Just 30
Prelude> Nothing >>= doble >>= triple
Nothing
```

- La notación *do*.

- Leyes de las mónadas:
  - (return x >>= f) es exactamente lo mismo que (f x)
  - (m >>= return) es igual que (m).
  - ((m >>= f) >>= g) es igual a (m >>= (\ x -> f x >>= g))

- Ver capítulos 11, 12 y 13 del manual "*Aprende Haskell por el bien de todos*".

http://aprendehaskell.es/content/Funtores.html