Contenido

Functores

La clase Functor de Haskell se define de esta manera

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

Cualquier tipo T puede ser instancia de Functor si puede satisfacer fmap[1]. De hecho, hemos conocido dos tipos que cumplen esta propiedad:

```
map :: (a -> b) -> [a] -> [b]
```
las listas

```
mmap :: (a -> b) -> Maybe a -> Maybe b
```
los Maybe

¿Qué kind tiene Functor?

```
:i Functor
-- constructor class with arity * -> *
```

Por supuesto, el mismo que Maybe y que las listas. Necesita un tipo para poder construir a su vez otro. Los functores representan cosas que pueden transformarse/mapearse (por eso definen el fmap como único requisito de su interfaz).

Y si nos fijamos en el Prelude:

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

o sea, la definición de fmap no es otra cosa que nuestra definición de mmap del módulo 9:

```
mmap f Nothing  = Nothing
```

[1] De hecho, todo functor debe satisfacer:
Ley de la Identidad: fmap id === id (mapear la función identidad no debe tener ningún efecto)
Ley distributiva de la composición: fmap f . fmap g === fmap (f . g)

```
mmap f (Just x) = Just (f x)
```

Y tenemos la definición de fmap para listas:

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

Por eso podemos usar el fmap tanto para listas como para maybes:

```
> fmap (^4) [3,2]
[81,16]

> fmap (* 8) $ Just 2
Just 16
```

¿Qué ganamos conociendo al functor?
Antes teníamos dos funciones: map y mmap, ahora usamos la misma función fmap y usamos polimórficamente dos tipos distintos.

Para pensar: ¿Podemos definir al tipo String como un Functor?
No, ¿cuál es el kind de String y cuál el de Functor?
El kind de String es *, el de Functor * -> * …
Definir en nuestro .hs
instance Functor Int where (...)

nos lleva a la misma explicación por parte de Haskell:

```
ERROR "D:\Program Files\Hugs98\lib\PDP_clase7.hs":160 - Kind
error in class constraint
*** constructor    : Int
*** kind           : *
*** does not match : * -> *
```

¿De qué tipo es fmap (*2)?

```
fmap (*2) :: (Num a, Functor f) => f a -> f a
```

Para más información véase:

- http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass
- http://stackoverflow.com/questions/2030863/in-functional-programming-what-is-a-functor
- http://www.seas.upenn.edu/~cis194/static/2010-10-25-slides.pdf

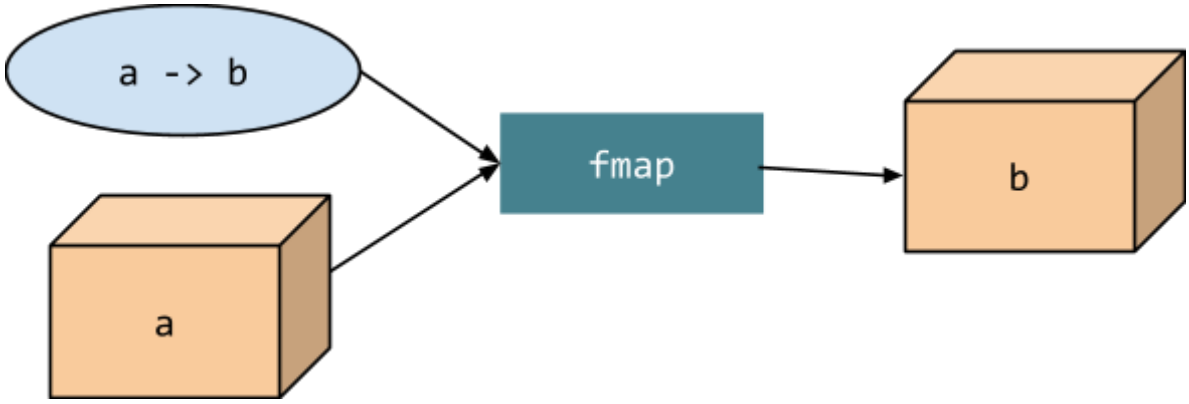
La caja: una metáfora útil

Podríamos pensar al Functor f donde f es una caja. Entonces revisando la definición de

fmap:

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

esto puede leerse como “recibimos una función que va de a en b” y “una caja de a” y termina devolviendo “una caja de b”.



IO: Entrada y Salida[2]

El siguiente bien pudo haber sido el primer ejemplo del curso de Haskell:

```
>putStrLn "Hola mundo"
Hola mundo
```

Claro, pero esta instrucción que pareciera tan ingenua, esconde conceptos que no son tan triviales de explicar. Para comenzar, ¿cuál es el dominio e imagen de putStrLn?

```
putStrLn :: String -> IO ()
```

Espera un String y devuelve… un IO (), esto se lee como una
“acción de Entrada/Salida” => IO
“ de ningún tipo asociado” => () –equivalente al void de Java o C-

Al ejecutarse una acción de I/O esto producirá efectos colaterales… ¿Cómo?
Pensemos en una función que permita que un usuario ingrese un conjunto de caracteres por el teclado: getLine. Si getLine devuelve un String, entonces no puede ser una función “en el sentido matemático”: podría devolver “hola” y “mundo” en diferentes momentos.

[2] El mejor material de referencia para el lector es el siguiente artículo: http://learnyouahaskell.com/input-and-output. También puede verse http://book.realworldhaskell.org/read/io.html

Por ese motivo, getLine no devuelve un String, sino un IO de tipo String. Volviendo al ejemplo de las cajas, getLine devuelve una caja que sabemos que adentro tendrá un string.

Separando una solución en partes con y sin efecto

Ejercicio: pedir al usuario que ingrese su nombre para luego sacar sus iniciales.

```
iniciales :: String -> String
iniciales = map head . words

mainIniciales = do
   putStrLn "Por favor ingrese su nombre completo: "
   nombreCompleto <- getLine
   putStrLn $ (++) "Sus iniciales son "  $ iniciales
nombreCompleto
```

Evaluamos el ejemplo:

```
> mainIniciales
Por favor ingrese su nombre completo:
Fernando Esteban Dodino
Sus iniciales son FED
```

Fíjense que Haskell permite aislar funciones *puras* (sin efecto colateral) e *impuras* (con efecto colateral):

- mainIniciales necesita tener efecto colateral dado que el input varía cada vez que le pido al usuario que ingrese su nombre
- pero la función iniciales no necesita tener efecto colateral: trabaja cómodamente con Strings y puede asegurar que siempre que ingrese “Quique Wolff” me devolverá “QW”

Esto ocurre muchas veces en la industria: hay partes de un sistema donde queremos tener efecto colateral y partes en las que no, es interesante notar que el contrato mismo de la función mainIniciales nos está indicando que va a tener efecto colateral:

```
> :t mainIniciales
mainIniciales :: IO ()
```

<= acción de Input/Output (tiene efecto)

Algo que no es posible lograr del mismo modo en objetos, incluso en lenguajes con chequeo estático, porque el paradigma de objetos deja abierta la posibilidad de que

cualquier método tenga efecto colateral.

do, <-

Vamos a aclarar un poco detalles de la implementación de mainIniciales:

```
mainIniciales = do
   putStrLn "Por favor ingrese su nombre completo: "
   nombreCompleto <- getLine
   putStrLn $ (++) "Sus iniciales son "  $ iniciales
nombreCompleto
```

1) La sentencia do permite agrupar un conjunto de acciones (en este caso, de I/O). La indentación no es casual, es importante para Haskell.
2) Cada una de estas acciones puede retornar o no algo. En particular, nos interesa saber qué nos devuelve getLine.
3) Dijimos anteriormente que getLine devuelve un IO String, o sea una caja con un string adentro. ¿Qué ocurre en la línea nombreCompleto <- getLine? bueno, no estamos asignando el resultado de getLine, dado que no existe ese concepto dentro del paradigma. Lo que sí hay es un “binding”, que consiste en sacar de la caja lo que devolvió el getLine. Como getLine es un IO String, nombreCompleto contendrá un String (lo que está adentro de la caja). Entonces sí puedo enviarle un String a la función iniciales (no puedo mandarle un IO String, porque iniciales espera un String para devolver otro String).

<- vs. let

Es posible utilizar bindings de dos maneras:

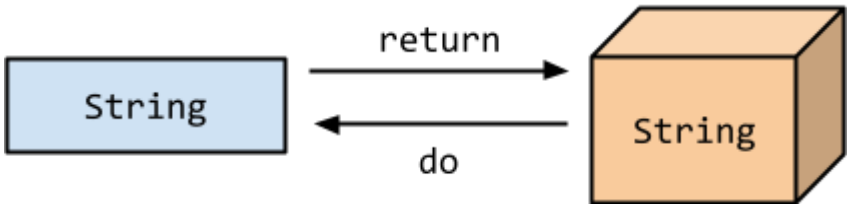
- variable <- expresión. Si la expresión devuelve un tipo S a, la variable es de tipo a y se bindea con el contenido de la caja
- let variable = expresión. La expresión devuelve un tipo a, la variable obviamente es de tipo a y se bindea con el resultado de la expresión.

En el anterior ejemplo, podríamos obtener las iniciales del nombre completo en un paso previo:

```
mainIniciales = do
   putStrLn "Por favor ingrese su nombre completo: "
   nombreCompleto <- getLine
   let inicialNombre = iniciales nombreCompleto
   putStrLn $ (++) "Sus iniciales son " inicialNombre
```

return

De la misma manera que <- permite sacar el contenido de lo que está en una caja, la sintaxis return permite hacer lo contrario:



La palabra return suele llevar a confusiones para quienes venimos de lenguajes imperativos: no corta el flujo de ejecución del programa, sino que permite tomar un valor de tipo a y llevarlo a algo de tipo S a.

En particular,

```
>:t return "hola"
```

es del tipo S [Char] o S String (en Haskell, S es una mónada, pero por el momento vamos a prescindir de ingresar en este mundo)

Un ejemplo rápido que muestra la diferencia entre el return de C/Java y el de Haskell:

```
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let fullName = first++" "++last
                putStrLn ("Pleased to meet you, "++fullName++"!")
                return fullName

nameReturn' = do putStr "What is your first name? "
                 first <- getLine
                 putStr "And your last name? "
                 last <- getLine
                 let fullName = first++" "++last
                 putStrLn ("Pleased to meet you,
"++fullName++"!")
                 return fullName
                 putStrLn "I am not finished yet!"
```

Evaluamos nameReturn:

```
Main> nameReturn
What is your first name? Fernando
And your last name? Dodino
```

```
Pleased to meet you, Fernando Dodino!
```

Y ahora nameReturn':

```
Main> nameReturn'
What is your first name? Fernando
And your last name? Dodino
Pleased to meet you, Fernando Dodino!
I am not finished yet!
```

1) La línea "I am not finished yet!" se imprime, eso comprueba que return no es una instrucción que retorna el flujo a la función original. En el caso de la función nameReturn', la línea return fullName no tiene efecto, dado que no se bindea contra ninguna variable.

2) Si le preguntamos a Haskell, ¡los tipos de nameReturn y nameReturn' son diferentes!

```
nameReturn :: IO [Char]    vs. nameReturn' :: IO ()
```

Esto tiene que ver con la última línea de la secuencia de acciones: en el primer ejemplo es un return, entonces si fullName es un String, return fullName devuelve un IO String. En cambio en nameReturn', la última acción es un putStrLn que es un IO ()…

IO como functor

IO, al igual que Maybe y las listas, es un fuctor. Definimos entonces fmap:

Recordemos que fmap tiene este tipo:

```
fmap :: Functor a => (b -> c) -> a b -> a c
```

Entonces, necesitamos pasarle una función a una acción IO, lo que devuelve es una acción de IO.

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

Uso de fmap

Volviendo al ejemplo de las iniciales:

```
mainIniciales = do
   putStrLn "Por favor ingrese su nombre completo: "
```

```
   nombreCompleto <- getLine
   let inicialNombre = iniciales nombreCompleto
   putStrLn $ (++) "Sus iniciales son " inicialNombre
```

Podemos reescribirlo utilizando fmap:

```
mainIniciales' = do
   putStrLn "Por favor ingrese su nombre completo: "
   inicialNombre <- fmap iniciales getLine
   putStrLn $ (++) "Sus iniciales son " inicialNombre
```

Claro, necesitamos utilizarlo en el contexto de una acción de IO, dado que fmap devuelve una acción IO. Entonces lo bindeamos contra inicialNombre para sacarlo de la caja y transformarlo a un String. El resto es historia conocida.

Queda para el módulo 11:

- Intro a mónadas.
- Ejemplo de bind