Programación Funcional en Scala

– Tema 4 (Parte I) –
Funtores y Mónadas

Jesús López González
jesus.lopez@hablapps.com

Programación Funcional en Scala
Habla Computing

Cursos ETSII-URJC 2015

Warming Up

Listing 1: Warming Up con typeclass YesNo

```
trait YesNo[A] {
  def yesNo(value: A): Boolean
}
```

Índice

Motivación (1/2)

En clases anteriores, ya habíamos dado nuestra propia implementación del método *map* para los tipos *Opcion* y *Lista*.

Listing 2: Implementación de map para 'Opcion' y 'Lista'

```
def map[B](f: A => B): Opcion[B] = this match {
  case Ninguno => Ninguno
  case Algun(a) => Algun(f(a))
}

def map[B](f: A => B): Lista[B] = this match {
  case Nada => Nada
  case Cons(x, xs) => Cons(f(x), xs.map(f))
}
```

Motivación (2/2)

Existen ciertos métodos que pueden implementarse en términos de map, por ejemplo *distributeLista* y *distributeOpcion*. Basándote en la signatura de tales funciones, ¿cuál crees que es su misión?

Listing 3: Ejercicio: Implementa distributeList y distributeOption

```
def distributeLista[A, B](lst: Lista[(A, B)]): (Lista[A],
    Lista[B]) =
???

def distributeOpcion[A, B](opc: Opcion[(A, B)]): (Opcion[A],
    Opcion[B])
= ???
```

Funtores

¿Qué es un funtor?

Un funtor no es más que la abstracción que nos permite generalizar la función map. De forma muy informal, podemos entender que un funtor recoge el comportamiento de "cosas" que se pueden mapear. La typeclass *Functor* recibe un constructor de tipos *F[_]* como parámetro tipo (cuyo kind es * -> *) Por tanto, podríamos instanciar dicha typeclass para tipos tales como *Opcion*, *Lista*, *Tuple1*...

Listing 4: La typeclass Functor

```
trait Functor[F[_]] {
  def map[A, B](value: F[A])(f: A => B): F[B]
}
```

Funciones que trabajan con funtores

Existe un gran rango de funciones que trabajan sobre funtores, por ejemplo *distribute*. Cualquier tipo que se una a la typeclass *Functor* tendrá acceso a ellas.

Listing 5: Implementa las siguientes funciones genéricas

```
def distribute[F[_]: Functor, A, B](fab: F[(A, B)]): (F[A], F[B])

def replace[F[_]: Functor, A, B](fa: F[A], b: B): F[B]

def strengthR[F[_]: Functor, A, B](fa: F[A], b: B): F[(A, B)]
```

Leyes de los Funtores

Unirse a un funtor no consiste exclusivamente en dar una implementación a la typeclass. Existen ciertas leyes que deben cumplirse para asegurar la propiedad *structure-preserving* del funtor. Es responsibilidad del programador el asegurar que se cumplen. De no hacerse, podrían obtenerse resultados inesperados al ejecutar funciones genéricas.

Una segunda intuición: levantando funciones

Hasta ahora hemos estado tratando a los funtores como "cosas" que se pueden mapear. Existe otro punto de vista, más acorde con el punto de vista de la teoría de las categorías. Para obtener dicha intuición, resulta conveniente invertir el orden de las listas de parámetros:

Listing 6: Ejercicio: ¿Qué intuición ofrece cada signatura?

```
// def map[A, B](value: F[A])(f: A => B): F[B]
def map[A, B](f: A => B): F[A] => F[B]
```

Índice

Efectos en FP

"Functional programmers often informally call type contructors like Par , Option , List , Parser , Gen , and so on effects . This useage is distinct from the term side effect , which implies some violation of referential transparency. These types are called 'effects' because they augment ordinary values with 'extra' capabilities (Par adds the ability to define parallel computation, Option adds the possibility of failure, and so on)."
(Functional Programming in Scala)

¿Qué es una Mónada?

Es una abstracción difícil de entender, ya que a diferencia del concepto "objeto", una mónada es algo que no existe en nuestra vida cotidiana. Su misión es la de abstraer ciertos comportamientos (generalmente asociados a efectos) y pensamos que la mejor forma de entender su significado es mediante ejemplos.

Listing 7: Monad es una nueva typeclass

```
trait Monad[M[_]] {
  def unit[A](value: A): M[A]
  def flatMap[A, B](m: M[A])(f: A => M[B]): M[B]
}
```

Índice

La mónada Opcion: gestión de errores

Caso Práctico 1

Se pretende ofrecer el problema de la gestión de errores como motivación hacia la mónada *Option*. La gestión de errores en la programación funcional genera mucho boilerplate y código verdaderamente tedioso. Con las abstracciones oportunas, se consigue un código mucho más legible y conciso.

Índice

La mónada Writer: imprimiendo trazas en la consola

Caso Práctico 2

Como ya vimos anteriormente, pintar en la consola mediante *println* es un efecto de lado que rompe la pureza. Por tanto, debe mitigarse o llevarse a las capas más externas de nuestro programa. Lo hacemos devolviendo el String pertinente para pintarlo más adelante. Esta práctica genera bastante boilerplate, que se reduce mediante los combinadores adecuados.

Takeaways

Hemos descubierto los funtores, estructura que nos permite abstraer el método *map* para instanciar objetos que se pueden mapear.

Hemos visto la estructura de una mónada, aunque todavía desconocemos todo su potencial. No obstante, hemos visto cómo nos pueden ayudar para gestionar errores y describir un sistema de Logging.

Empezamos a ver pinceladas de separación entre "descripción" e "interpretación" (ej: *runWriter*)

El próximo día veremos la mónada Lista, que nos permite introducir no determinismo, y la mónada Estado, que nos enseñará que paradigmas funcional e imperativos no están tan reñidos como podemos pensar.