

# 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

## Listing 1: Warming Up con typeclass Show

```
trait Show[A] {  
  def show(value: A): String  
}
```

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Option: gestión de errores
- 4 La mónada Either: gestión de errores con información
- 5 La mónada List: lidiando con el no-determinismo

## 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])  
= ???
```

## ¿Qué es un functor?

Un functor no es más que la abstracción que nos permite generalizar la función `map`. De forma muy informal, podemos entender que un functor recoge el comportamiento de “cosas” que se pueden mapear. La typeclass *Functor* recibe un constructor de tipos  $F[_]$  como parámetro tipo (cuyo kind es  $* \rightarrow *$ ) 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 responsabilidad 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)(value: F[A]): F[B]
```

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Option: gestión de errores
- 4 La mónada Either: gestión de errores con información
- 5 La mónada List: lidiando con el no-determinismo

## Efectos en FP

*“Functional programmers often informally call type constructors like `Par` , `Option` , `List` , `Parser` , `Gen` , and so on effects . This usage 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]  
}
```

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Option: gestión de errores
- 4 La mónada Either: gestión de errores con información
- 5 La mónada List: lidiando con el no-determinismo

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Option: gestión de errores
- 4 La mónada Either: gestión de errores con información
- 5 La mónada List: lidiando con el no-determinismo

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Option: gestión de errores
- 4 La mónada Either: gestión de errores con información
- 5 La mónada List: lidiando con el no-determinismo