

# Programación Funcional en Scala

## – Tema 4 – 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 YesNo

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

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

## 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): F[A] => F[B]
```

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

## 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 Opcion: gestión de errores**
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

## 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.

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

## 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.



- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo**
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

## Caso Práctico 3

Existen ciertas funciones que son no deterministas. Un ejemplo de este tipo de función podría ser *sqrt* (raíz cuadrada). ¿Qué debería devolver el siguiente programa? *add(2, sqrt(9))* La respuesta más directa podría ser 5. No obstante, no debemos olvidar que *sqrt(9)* devuelve en realidad  $(+3, -3)$ . Por tanto, el resultado del programa debería ser  $(+5, -1)$  La mónada Lista nos permite lidiar con funciones no deterministas de una manera muy elegante.

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común**
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

# Una mónada para gobernarlos a todos

## Mónada como abstracción común

Una mónada es la abstracción que generaliza las funciones *unit* y *flatMap*. Nos permite representar computaciones definidas como secuencias de pasos. Se aplican en contextos muy diversos, por lo que es muy recomendable analizar las diversas instancias existentes para adquirir una mejor intuición (*State*, *Reader*, *Free*...)

### Listing 8: Ejercicio: Implementa estas funciones

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

# Una mónada para gobernarlos a todos

## Funciones monádicas

Al igual que ocurría con *Functor*, existen una serie de funciones que podemos implementar una única vez y estarán disponibles para cualquier tipo de dato que se una a la typeclass *Monad*.

### Listing 9: Ejercicio: Implementa estas funciones genéricas

```
trait Monad[M[_]] {  
  def map2[A, B, C](m: M[A], mb: M[B])(f: (A, B) => C): M[C] =  
    ???  
  
  def sequence[A](lma: List[M[A]]): M[List[A]] = ???  
  def product[A, B](ma: M[A], mb: M[B]): M[(A, B)] = ???  
}
```

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways

# Scala, ¿con pilas incluidas?

## ¿Qué ofrece y qué no ofrece Scala de forma nativa?

- Scala no despliega *typeclasses*, aunque como ya hemos visto, nos permite simularlas.
- Scala no da entidad al concepto *Functor*, ni al concepto *Monad*, pero nos ofrece una implementación para funciones tales como *map* o *flatMap*.
- Scala nos dulcifica la sintaxis de una mónada mediante una *for-comprehension*, aplicable a nuestros propios tipos.
- Existen librerías como *scalaz* que dan entidad a abstracciones funcionales y facilitan el trabajo en este lenguaje con el paradigma funcional.

- 1 Funtores
- 2 Efectos y Mónadas
- 3 La mónada Opcion: gestión de errores
- 4 La mónada Writer: imprimiendo trazas en la consola
- 5 La mónada Lista: lidiando con el no-determinismo
- 6 Mónada como abstracción común
- 7 Scala, ¿con pilas incluidas?
- 8 Takeaways



# 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 y varios ejemplos, con objetivos muy dispares entre sí.

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

El próximo día veremos cómo implementar nuestros propios “efectos”, haciendo fuerte empeño en separar los aspectos de descripción e interpretación.