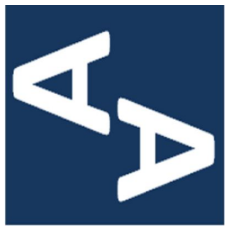
Programación funcional en Scala

2. Más allá de las HOFs

2.1 Type classes

2.2 Funtores

2.3 Mónadas



Habla Computing
info@hablapps.com
@hablapps

Objetivos

- Entender el papel de las *type classes* dentro del esquema de mecanismos de **modularidad**

- Saber implementar instancias de *type classes* para
  - Tipos regulares
  - **Constructores de tipos**

- Saber explicar lo que es un **monoide**, un **funtor** y una **mónada**.
  - *Estructuras de datos funtoriales y monádicas*

Más allá de las HOFs

- **Type classes**
- Funtores
- Mónadas

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES

¿Qué son las *type classes*?

- Las *type classes* son clases de tipos
  - Permiten caracterizar tipos en base a una serie de operaciones
  - Son un mecanismo de modularidad

- No confundir con las *clases en OO*, que son clases de objetos

Type classes
(I) Programas monolíticos

```scala
// (I) Versión monolítica

// Ejemplo 1
def sum(l: List[Int]): Int =
  l match {
    case Nil => 0
    case x :: r => x + sum(r)
  }

// Ejemplo 2
def concat(l: List[String]): String =
  l match {
    case Nil => ""
    case x :: r => x + concat(r)
  }
```

Type classes
(II) Patrón recurrente

```scala
// (II) Patrón recurrente
// Sin type classes: abstraemos los valores y funciones
def collapse[A](l: List[A])(
  zero: A, add: (A,A) => A): A =
  l.fold(zero)(add)

// NOTA: `foldRight` y `fold`(catamorfismo) son exactamente
iguales para listas. Pero NO lo son para otras estructura
```

Type classes
(II) Patrón recurrente

```scala
// (II) Patrón recurrente
// Con type classes
trait Monoide[T]{           // Los Monoides tienen leyes:
  def add(t1: T, t2: T): T // - Asociatividad
  val zero: T               // - Elemento neutro
}

def collapse[A](l: List[A])(monoid: Monoide[A]): A =
  l.fold(monoid.zero)(monoid.add)
```

Type classes
(III) Versiones modularizadas

```scala
// (III) Versión modularizada

// Ejemplo 1
val intMonoid: Monoide[Int] = new Monoide[Int]{
  val zero: Int = 0
  def add(i1: Int, i2: Int): Int = i1 + i2
}
def sum(l: List[Int]): Int = collapse(l)(intMonoid)

// Ejemplo 2
object strMonoid extends Monoide[String]{
  val zero: String = ""
  def add(s1: String, s2: String): String = s1 + s2
}
def concat(l: List[String]): String =
  collapse(l)(strMonoid)
```

¿Qué son las *type classes*?

- Las *type classes* son tipos genéricos que proporcionan una funcionalidad adicional al tipo que parametrizan (métodos, valores, ...)
- Estos métodos, valores, etc., tienen que estar implementado conforme a unas **leyes**
  - **E.g.:** Monoides (asociatividad e identidad)
- Permiten agrupar la funcionalidad y valores comunes de tipos

Ventajas de las type classes:
Corrección, Expresividad y Generalidad

- Generalidad
  - Una *type class* debe poder aplicarse a muchos tipos
- Expresividad
  - Cuanto más potentes las funcs. primitivas de la *type class*, más operaciones derivadas podré definir
- Corrección
  - Al agrupar las impls del `trait` es más fácil vigilar la corrección de sus propiedades

```
Ejercicio: EXPRESIVIDAD y GENERALIDAD... ¿son
complementarias o se contradicen entre sí?
```

Type classes
En Scala: *implicits* + *context bounds*

- Objetivo: enseñar el código idiomático en Scala para *type classes*
  - Aún tengo que pasar la instancia de **`Monoide`** a la función ¿Puedo evitarlo?

```
// (II) Patrón recurrente
def collapse[A](l: List[A])(monoid: Monoide[A]): A =
  l.foldLeft(monoid.zero)(monoid.add)

// (III) Versión modularizada
def sum(l: List[Int]): Int = collapse(l)(intMonoid)
def concat(l:List[String]): String = collapse(l)(strMonoid)
```

Type classes con *implicits*

```scala
// (II) Patrón recurrente (implícitos)
def collapse[A](l:List[A])(implicit monoid: Monoide[A]): A =
  l.foldLeft(monoid.zero)(monoid.add)

// (III) Versión modularizada
implicit val intMonoid: Monoide[Int] = new Monoide[Int]{
  val zero = 0
  def add(i1: Int, i2: Int): Int = i1 + i2
}
implicit object stringMonoid extends Monoide[String]{
  val zero: String = ""
  def add(s1: String, s2: String): String = s1 + s2
}

def sumaInt(l: List[Int]): Int = collapse(l)
def concat(l: List[String]): String = collapse(l)
```

Type classes con *context bounds*

```scala
// (II) Patrón recurrente (context bounds, con implicitly)
def collapse[A: Monoide](l:List[A]): A = {
  val monoid = implicitly[Monoide[A]]
  l.foldLeft(monoid.zero)(monoid.add)
}

// (II) Patrón recurrente (context bounds, sin implicitly)
object Monoide {
  def apply[A](implicit ev: Monoide[A]): Monoide[A] = ev
}
def collapse[A: Monoide](l:List[A]): A =
  l.foldLeft(Monoide[A].zero)(Monoide[A].add)

// (III) Versión modularizada
def sumaInt(l: List[Int]): Int = collapse(l)
def concat(l: List[String]): String = collapse(l)
```

Type classes - *context bound*

- No toda *type class* usa *context bound*
  - Recordar primer ejemplo **`Monoide`** al principio
- Pero si hay *context bound*, hay *type class*

```
def f[T: TypeClass](t: T)
// T:TypeClass → “T es una instancia de TypeClass”
// t:T         → “t es una instancia de T”
```

- **`[T: TypeClass]`** expresa que (dentro de la función) el tipo **`T`** pertenece a **`TypeClass`**
  - Métodos de **`TypeClass`** disponibles sobre **`T`**

Polimorfismo *ad-hoc*

- El código no es “solo” paramétrico en **T**, *i.e.* las funciones reciben info extra (*ad-hoc*) sobre **T**

- Una *type class* puede dar información extra para convertir, serializar, combinar... valores de tipo **T**
  - `Show[T]`, `Equals[T]`, `Order[T]`, `...`

Type classes
¿Y por qué no herencia?

- Se debe poder cambiar (extender) `class T`
  - Imposible si es `final`
  - Clases de la librería estándar (`Int`, `String`, ...) no son extensibles

- No es posible acceder a info ‘estática’ de la clase (mirar ejemplo siguiente *slide*)

Type classes
¿Y por qué no herencia?

```
trait Monoide[T]{
  def add(t1: T, t2: T): T
  def zero: T
}

class C extends Monoide[C]{
  def zero: C = ???   // Impl de zero para C
  def add(c2:C) = ??? // Impl de add para C
  /// ... Resto implementación C
}

object Test {
  def collapse[A <: Monoide[A]](l:List[A]) =
    l.foldLeft(???)(_ add _) // Imposible acceder a zero!
}
```

Ejercicios type classes

Ejercicio: Implementar una *type class* para cálculos estadísticos dentro del `object TypeClassParaEstadisticas` en **tema2-typeclasses/EjerciciosClase_TypeClasses.scala**

En ese mismo object encontrarás implementaciones alternativas de la misma funcionalidad. Puedes estudiarlas para compararlas con la implementación mediante *type classes*.

Más allá de las HOFs

- Type classes
- Monoides
- **Funtores**
- Mónadas

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES

Genericidad *higher-kind*

```scala
// Ejemplo 1 (map for List)
def map[A,B](l: List[A])(f: A=>B): List[B] =
  l match {
    case Nil => Nil
    case head :: tail =>
      (f(head): B) :: map(tail)(f)
  }

// Ejemplo 2 (map for Option)
def map[A,B](o: Option[A])(f: A=>B): Option[B] =
  o.fold(None:Option[B])(
    (a: A) => Some(f(a)):Option[B]
  )

// map for Tree, Set, Try...
```

Genericidad *higher-kind*
(I) Monolítica

```scala
// (I) Versión monolítica

// Ejemplo 3 (duplicar elementos List)
def duplicate[A](l: List[A]): List[(A,A)] =
  map(l)((a: A) => (a,a))

// Ejemplo 4 (duplicar elementos Option)
def duplicate[A](o: Option[A]): Option[(A,A)] =
  map(o)((a: A) => (a,a))

// duplicate for Tree, Set, Try...
```

Genericidad *higher-kind*
(II) Patrón recurrente

```scala
// (II) Patrón recurrente

// Necesitamos abstraer ‘map’ sobre List[_], Option[_],
// Set[_]... estos tipos se llaman “constructores de tipos”

// Abstracción sobre el constructor de tipos en una
// type class (antes abstraíamos sobre tipos)
trait MapFunction[F[_]]{
  def apply[A,B](fa: F[A])(f: A=>B): F[B]
}

// Generalización de ‘duplicate’ a partir de MapFunction
def duplicateF[F[_],A](fa:F[A])(map:MapFunction[F]): F[(A,
A)] =
  map[A,(A,A)](fa)((a: A) => (a,a))
```

Genericidad *higher-kind*
(III) Modularizado

```scala
// (III) Versión modularizada
// Creamos instancia de type class MapFunction para Option
val optionMapFunction = new MapFunction[Option]{
  def apply[A,B](oa: Option[A])(f: A=>B): Option[B] =
    map(fa)(f)
}
// Ya podemos modularizar a partir de duplicateF
def modularDuplicate[A](oa: Option[A]): Option[(A,A)] =
  duplicateF(oa)(optionMapFunction)
```

Genericidad *higher-kind*
(III) Modularizado

```
// Antes de modularizar



map(List)
map(Option)
map(Tree)
map(Set)
…

duplicate(List)
duplicate(Option)
duplicate(Tree)
duplicate(Set)
…
```

```
// Modularizado

trait MapFunction(F[_])

MapFunction(List)
MapFunction(Option)
MapFunction(Tree)
MapFunction(Set)
…


// One for all!
duplicate(F[_])(MapFunction[F])
```

Genericidad *higher-kinded* en Scala

- Hemos visto ejemplos de constructores de tipos *(tipos que permiten construir tipos a partir de otros tipos)*
  - Es decir, `List[_]`, `Map[_,_]`, `Option[_]`…
- Scala permite abstraer los *constructores de tipos*

```
class ClassUsingHigherKinded[M[_]] { … }

val v1 = new ClassUsingHigherKinded[List]()
val v2 = new ClassUsingHigherKinded[Option]()
```

¿Qué es un funtor?

- Funtor es una *type class* que se define sobre *constructores de tipos* (`F[_]`)

- Transforma el contenido del tipo `F[_]`, *manteniendo la forma de la estructura*

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

```
Ejercicio: ¿Puedo implementar
def filter[A](f:A=>Boolean): F[A] solo con map?
```

Leyes de los funtores

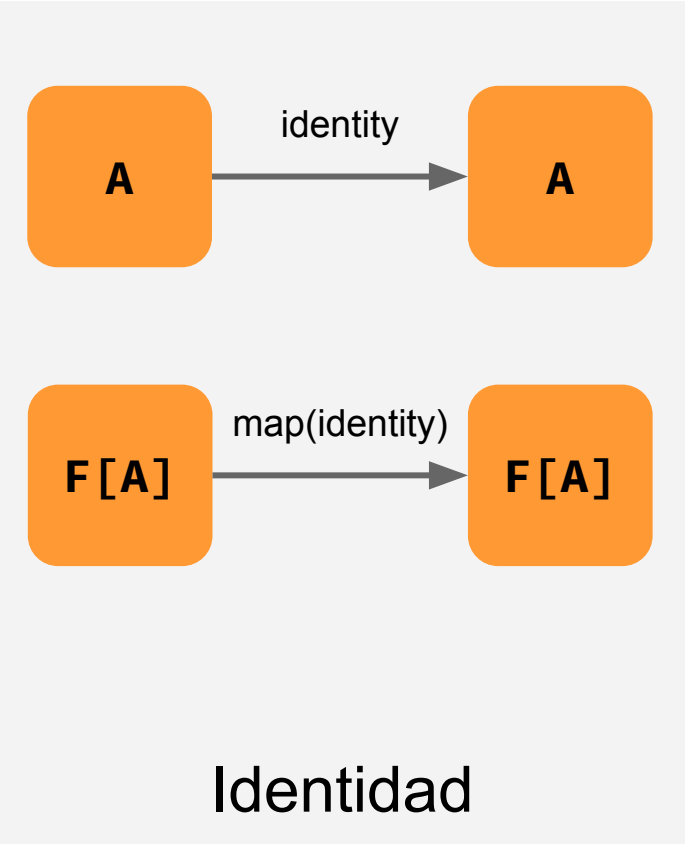
- Identidad: si no hay cambio en el contenido, no hay cambio en absoluto

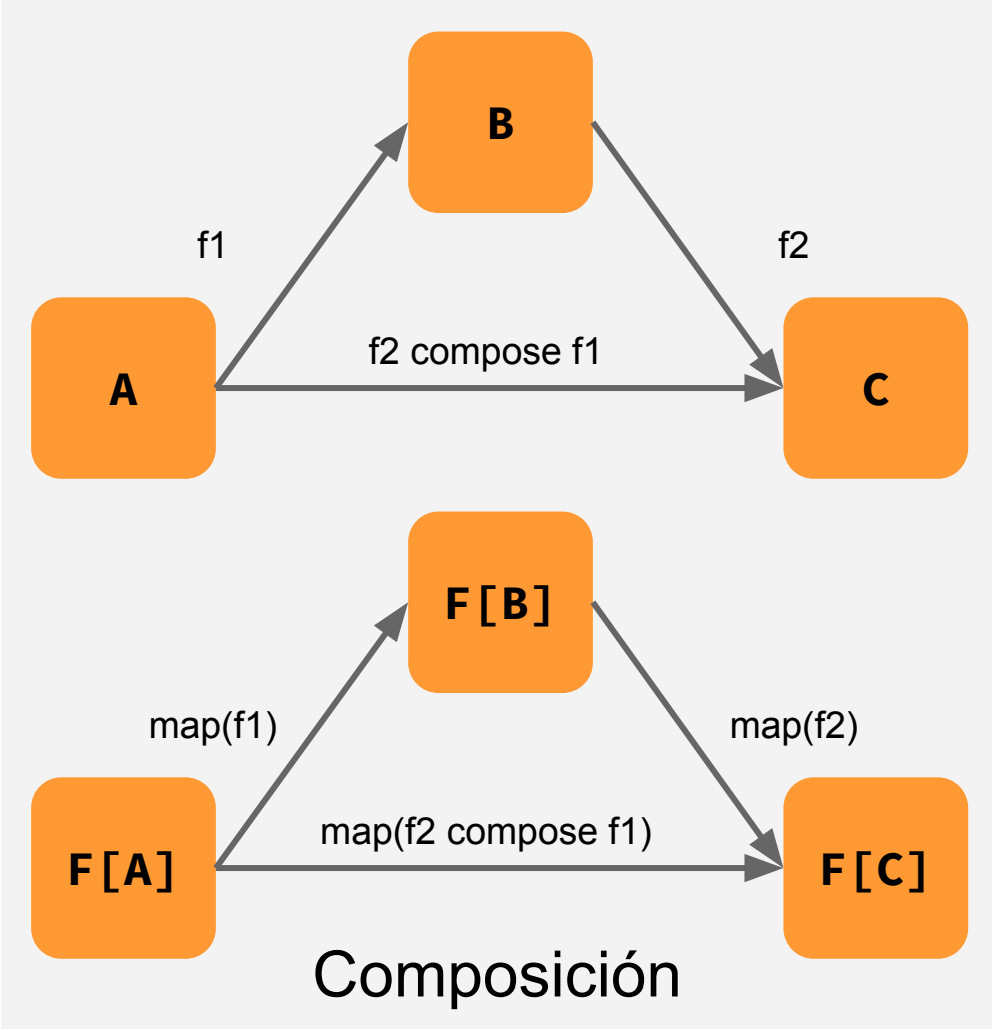
```
def identity[A](fa: F[A]) = map(fa)(x => x) === fa
```

- Composición: preserva la composición de funciones

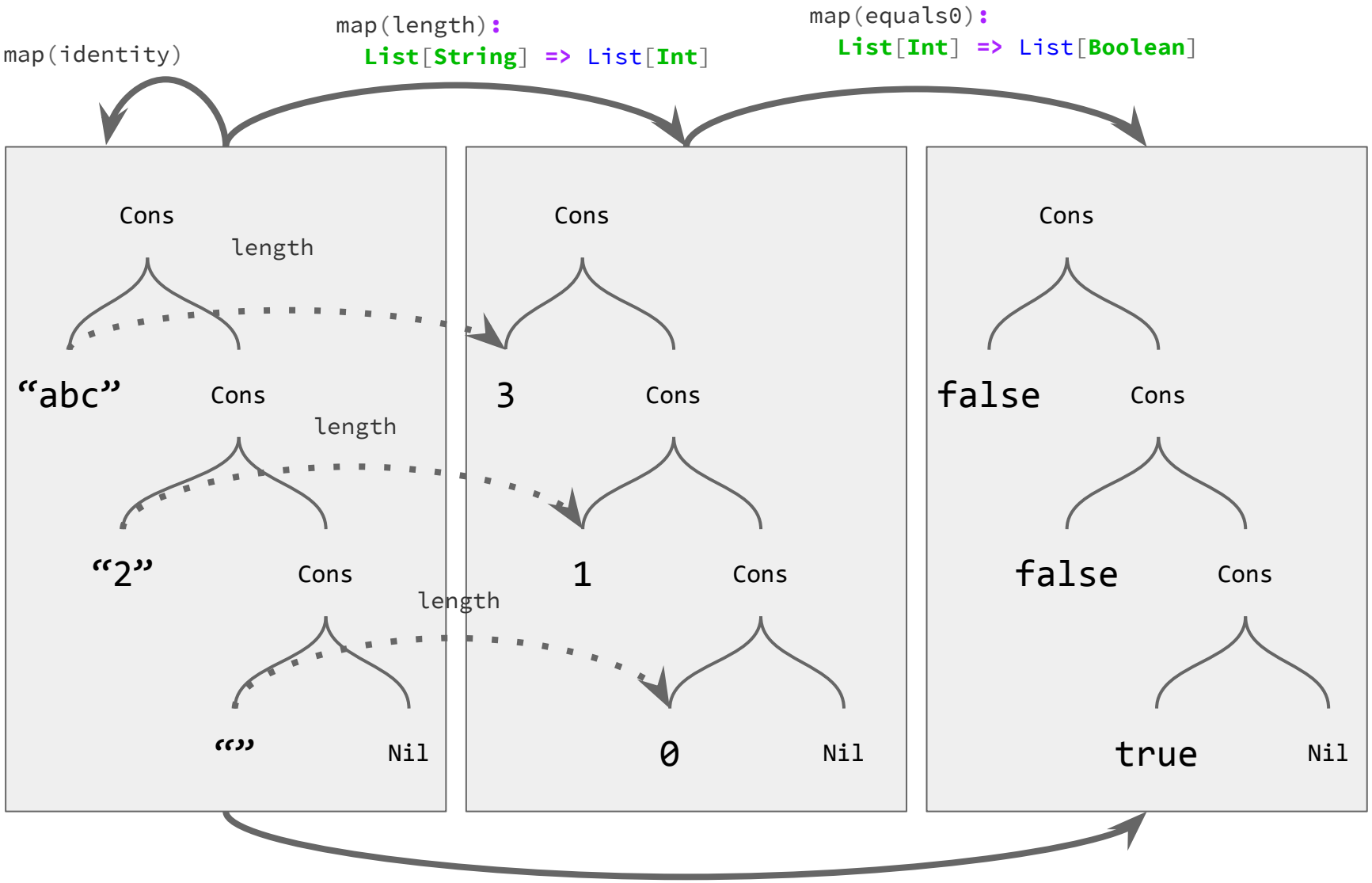
```
def composite[A, B, C](fa: F[A], f1: A=>B, f2: B=>C)(...) =
  map(map(fa)(f1))(f2) === map(fa)(f2 compose f1))
```

Leyes de los funtores





Ejemplo con List



Ejercicios (FUNTORES)

Ejercicio: Implementar una *type class* para cálculos estadísticos dentro del `object EjerciciosFuntores` en **tema2-typeclasses/EjerciciosClase_Funtores.scala**

En ese mismo object encontrarás implementaciones alternativas de la misma funcionalidad. Puedes estudiarlas para compararlas con la implementación mediante *type classes*.

Más allá de las HOFs

- Type classes
- Monoides
- Funtores
- **Mónadas**

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES

¿Qué es una Mónada?

- Mónada es una *type class* que se define sobre *constructores de tipos* (`M[_]`)
- Toda mónada es un funtor...
  - tiene `def map[A, B](fa: M[A])(f: A => B): M[B]`
- ... con funcionalidad adicional

```
trait Monada[M[_]] extends Funtor[M] {
  def flatten[A](mma: M[M[A]]): M[A]
  def pure[A](value: A): M[A]
}
```

¿Qué es una estructura de datos monádica?

- Una estructura que puede ser aplanada

```
val l1: List[List[Int]] = List(List(1, 2), List(3, 4))
val l2:      List[Int]  = flatten(l1)
// l2 = List(1, 2, 3, 4)
```

- Una estructura que se puede crear a partir de un valor dado

```
val l: List[Int] = pure(8)
// l = List(8)
```

Leyes de las Mónadas

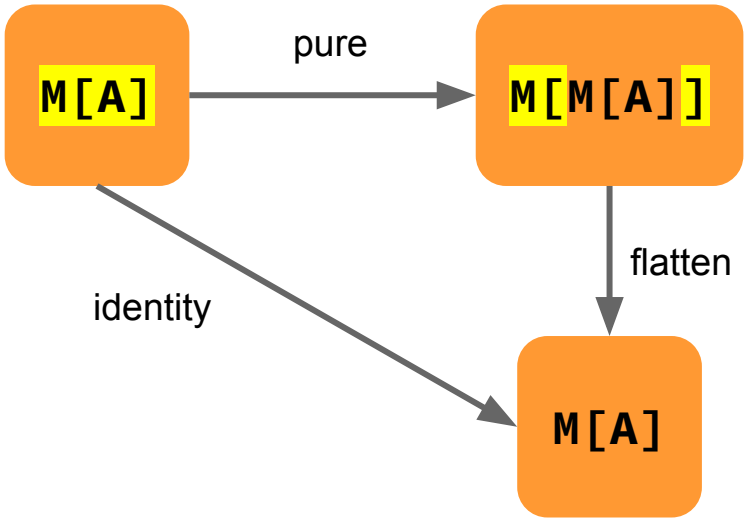
- Identidad: añadir estructura sin añadir información no modifica el contenido

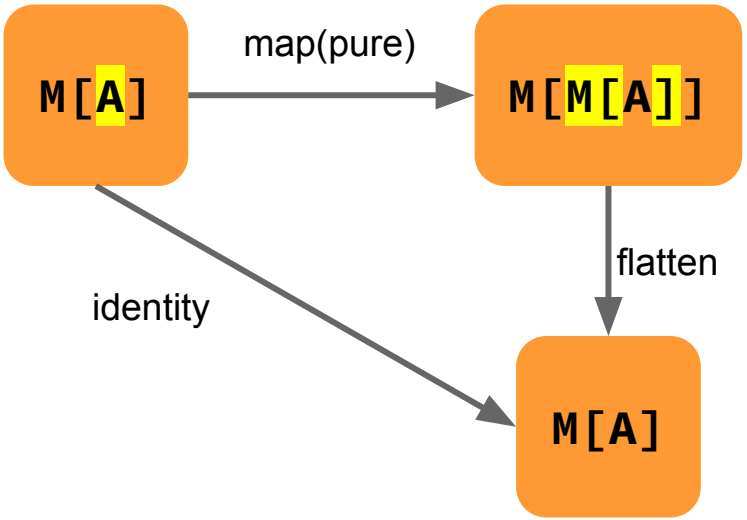
```
def identity[A](ma: M[A])   =   flatten(pure(ma)) === ma
```

- Asociatividad: no importa el orden de ‘aplanamiento’

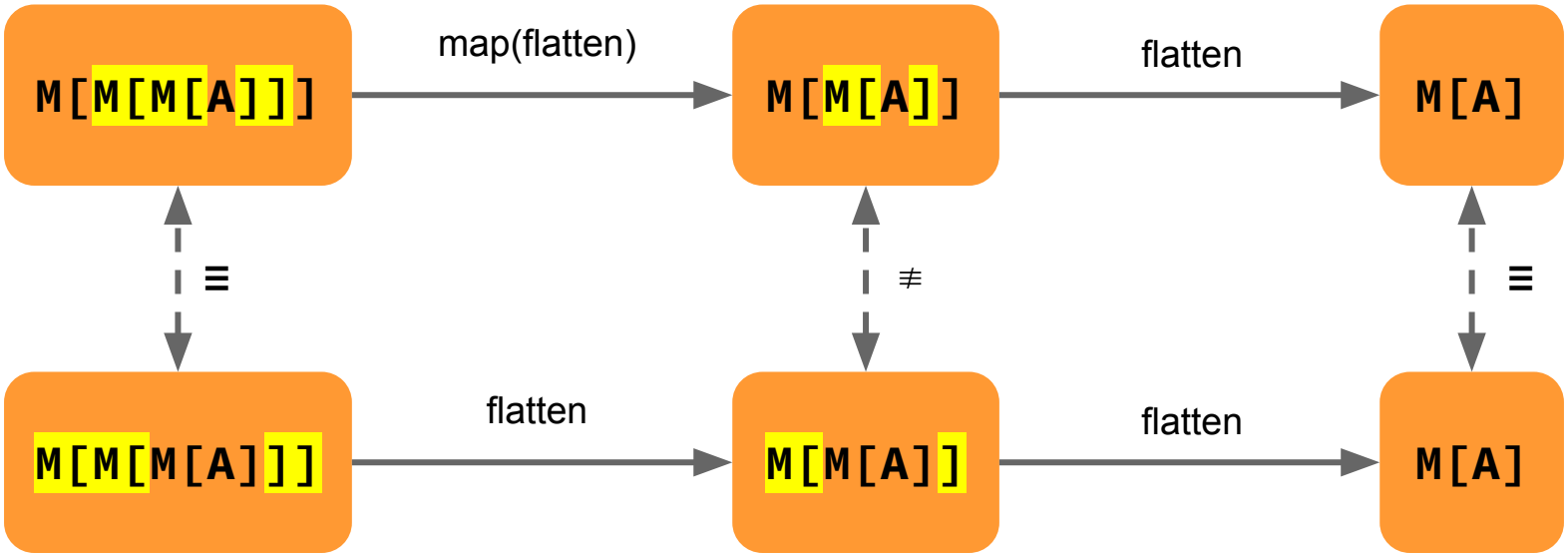
```
def associative[A,B,C](mmma:M[M[M[A]]], f1:A=>B, f2:B=>C) =
  flatten( map(mmma)(flatten) ) === flatten(flatten(mmma))
```

Leyes de las Mónadas *(identidad)*





Leyes de las Mónadas *(asociatividad)*



Cálculo de posibilidades

- Podemos utilizar listas para representar posibilidades

```
val dado: List[Int] = List(1, 2, 3, 4, 5, 6)
```

- ¿Cómo combinamos posibilidades?

```
val dosDados: List[List[Int]] = dado map (_ => dado)
// List(List(1, 2, 3, 4, 5, 6), List(1, 2, ...), ...)
```

Cálculo de posibilidades

- Problema1: Estamos perdiendo la información del primer dado. Hay que propagarla.

```
val dosDados: List[List[Int]] =
  dado map (d1 => dado map (_ + d1))
// List(List(2, 3, 4, 5, 6, 7), List(3, 4, ...), ...)
```

- Problema2: Estamos anidando listas. Aplanamos...

```
val dosDadosFlatten: List[Int] = dosDados.flatten
// List(2, 3, 4, 5, 6, 7, 3, 4, ...)
```

¡*Map* + *flatten* = *flatMap*!

- Combinar un map y un flatten es un patrón muy recurrente

```
trait Monada[M[_]] extends Funtor[M] {
  def flatten[A](mma: M[M[A]]): M[A]
  def pure[A](value: A): M[A]

  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B] = {
    val mmb: M[M[B]] = map(ma)(f)
    flatten(mmb)
  }
}
```

Otra manera de ver las mónadas

- Hay diferentes formas de representar una mónada
  - *map* + *flatten* + *pure*
  - *flatMap* + *pure*

```
trait Monada[M[_]] extends Funtor[M] {
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  def pure[A](value: A): M[A]

  def map[A, B](ma: M[A])(f: A => B): M[B] =
    flatMap(ma)(f andThen pure)
  def flatten[A](mma: M[M[A]]): M[A] =
    flatMap(mma)(identity)
}
```

Cálculo de posibilidades (*flatMap*)

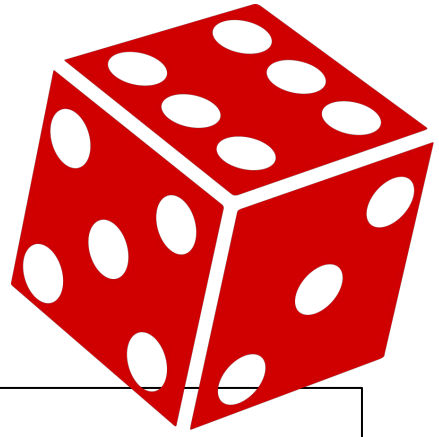
- Sin *flatMap*

```
val dosDados: List[Int] =
  (dado map (d1 => dado map (_ + d1))).flatten
// List(2, 3, 4, 5, 6, 7, 3, 4, ...)
```

- Con *flatMap*

```
val dosDados: List[Int] =
  dado flatMap (d1 => dado map (_ + d1))
// List(2, 3, 4, 5, 6, 7, 3, 4, ...)
```

Cálculo de posibilidades (*azúcar sintáctico*)



- Sin *azúcar*

```
val dosDados: List[Int] =
  dado flatMap (d1 => dado map (d2 => d1 + d2))
// List(2, 3, 4, 5, 6, 7, 3, 4, ...)
```

- Con *azúcar*

```
val dosDados: List[Int] =
  for {
    d1 <- dado
    d2 <- dado
  } yield d1+d2
// List(2, 3, 4, 5, 6, 7, 3, 4, ...)
```

for-comprehension en Scala

- Scala traduce `for {...} yield` en secuencias de `flatMap`, `filter` y `map`

```
val res: List[Int] = for {
  i <- (1 to 10) if(i % 2) != 0 // filter + flatMap
  j <- List(100,200) // map
} yield i + j

(1 to 10)
  .filter(_%2!=0)
  .flatMap(i=>List(100,200)
    .map(j => i+j))
```

Cálculo de posibilidades (*ejercicio*)

```scala
def calcular[A](posibilidades: List[A])
  (cond: A => Boolean): Double = ???
```

Ejercicios: en el `object EjerciciosClase` en **tema2-typeclasses/EjerciciosClase_Funtores.scala**

- Implementar un calculador de probabilidades
- Utilizarlo para calcular algunas condiciones

Resumen

Monoide

Permite 'reducir' estructuras del tipo `T`

```
trait Monoide[T] {
  def add(t1: T, t2: T): T
  val zero: T
}
```

```
// ASOCIATIVIDAD
def associative[T](t1: T, t2: T, t3: T) =
  add(t1,add(t2,t3)) === add(add(t1,t2), t3)

// ELEMENTO NEUTRO
def neutro[T](t:T) = add(t,zero) === add(zero,t) === t
```

Funtor

Transformaciones sobre *type const.* `F[_]` que cambian contenido pero no estructura

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

```
// IDENTIDAD
def identity[A](fa: F[A]) = map(fa)(x => x) === fa

// COMPOSICIÓN
def composite[A,B,C](fa:F[A],f1:A=>B,f2:B=>C) =
  map(map(fa)(f1))(f2) === map(fa)(f2 compose f1))
```

Mónada

Transformaciones sobre *type const.* `M[_]` para modificar la estructura + Funtor

```
trait Monada[M[_]] extends Funtor[M]{
  def flatten[A](mma: M[M[A]]): M[A]
  def pure[A](value: A): M[A]
  // flatMap ⇔ map + flatten
}
```

```
// IDENTIDAD
def identity[A](ma:M[A]) =
  flatten(pure(ma)) === ma &&
  flatten(map(ma)(pure)) === ma

// ASOCIATIVIDAD
def associative[A,B,C](mmma:M[M[M[A]]],f1:A=>B,f2:B=>C) =
  flatten(map(mmma)(flatten)) === flatten(flatten(mmma))
```

Conclusiones

- Las *type classes* nos permiten **clasificar** tipos, constructores de tipos, etc., que representan estructuras de datos en función de las operaciones que soportan
- El éxito de una *type class* lo da el número de operaciones que se pueden definir mediante su interfaz, y por su grado de aplicabilidad potencial
- Los funtores representan *cierto tipo de estructuras de datos*: se puede modificar su contenido sin modificar su forma
- Las mónadas representan *cierto tipo de estructuras de datos*: las que se pueden aplanar
- Scala tiene mecanismos para trabajar con type classes en general (*implicits*, *context bound*, *higher-kind generics*, ...) y con mónadas en particular (*for-comprehension*)

Ejercicios para casa

- **`Ej.1 tema2-typeclasses/homework/EjercicioTypeClasses.scala`**
  - Impl. *type class* para cálculos estádísticos
  - Comparar con una solución típica de OO (patrón *Adapter*)

- **`Ej.2 tema2-typeclasses/homework/EjercicioFuntores.scala`**
  - Impl. **`map`** para clase **`Tree`** e **`Intento`** dentro de la instanci del **`Funtor`** correspondiente
  - Impl. **`suma1`** para sumar 1 a los miembros de **`F[G[Int]]`** siendo **`F`** y **`G`** dos funtores

- **`Ej.3 tema2-typeclasses/homework/EjercicioMonadas.scala`**
  - Generar el espectro de posibilidades de lanzar monedas a *cara o cruz*
  - Utilizar la mónada *Option* como método para lidiar con errores