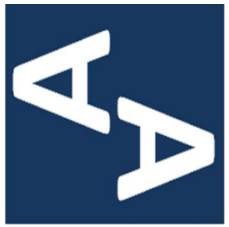
Programación funcional en Scala

1. ¿Qué es la programación funcional?



Habla Computing
info@hablapps.com
@hablapps

Objetivos

- Saber qué es la programación funcional, las **funciones** y los **tipos algebraicos de datos**
- Ser capaz de explicar sus ventajas, y el papel que juega en ellas la **modularidad**
- Entender cómo las **funciones**, el **polimorfismo paramétrico** y las **funciones de orden superior** (**catamorfismos** en particular) ayudan a crear código más modular

¿Qué es la programación funcional?

- **Funciones y ADTs**
- ¿Por qué PF? ¡Modularidad!
- La escalera de modularidad

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES

¿Qué es la programación funcional?

- Un *paradigma* o estilo de programación: patrones de diseño que determinan una forma de estructurar programas
  - funcional
  - orientado a objetos
  - imperativo
  - lógico
  - actores
  - ensamblador
  - ...

Hitos en la Programación Funcional

- 1930s- *Lambda calculus* (Church)
- 1958- *LISP* (McCarthy)
- 1970s- *ML* (Milner), HOPE
- 1987- *HASKELL*
- **1989- Monads in Computer Science (MOGGI)**
- 1990- Monads in Haskell (Wadler)
- **2004- *Scala* (Odersky)**
- 2005- *F#* (Microsoft)
- 2007- *Clojure* (Hickey)
- 2008- Scalaz (Tony Morris, Jason Zaugg, Runar B., ...)
- 2013- Shapeless (Miles Sabin)
- 2014- *Java8*, *Swift* (Apple)

Definición

La programación funcional es programar con ***funciones puras***: módulos de software altamente componibles, que reciben valores de entrada, devuelven valores de salida, y no hacen nada más.

Funciones como métodos

```scala
def length(s: String): Int = s.length

def add(i: Int, j: Int): Int = i + j

def times(i: Int, j: Int): Int = i * j

def odd(i: Int): Boolean = i % 2 == 1

def even(i: Int): Boolean = !odd(i)

def five: Int = 5
```

Funciones recursivas

```
// Suponemos que n va a ser >= 0 por simplicidad
def factorial(n: Int): Int = {
  if (n > 1)
    n * factorial(n-1)
  else
    1
}
```

Funciones *“tail recursive”*

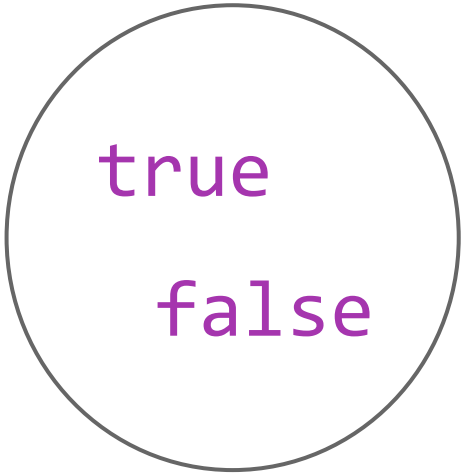
```scala
// Suponemos que n va a ser >= 0 por simplicidad
def factorial(n: Int): Int = {
  @scala.annotation.tailrec
  def go(acc: Int, _n: Int): Int = {
    if (_n > 1)
      go(_n * acc, _n - 1)
    else
      acc
  }
  go(1, n)
}
```

Tipos y valores en funciones

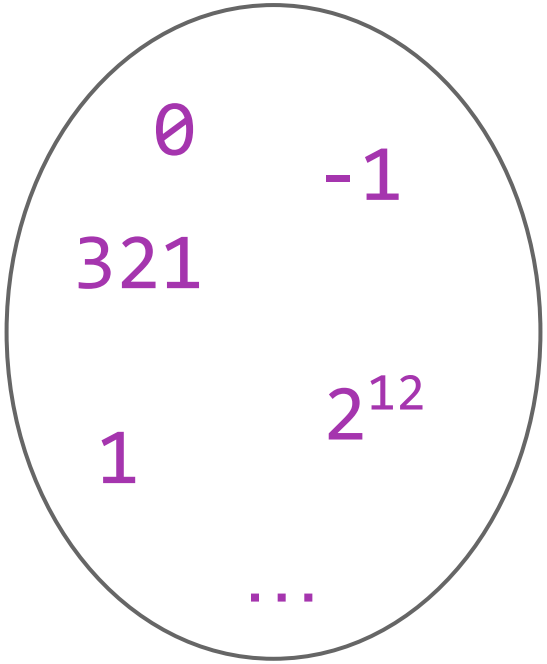
- Las funciones transforman valores en valores (dominio → codominio)
- Estos valores pertenecen a un tipo
- Los tipos son *conjuntos de valores*
- Ejemplos de valores: `3`, `4`, `true`, `"hola mundo"`, ...
- Ejemplos de tipos: `Int`, `Boolean`, `String`

Tipos y valores en funciones

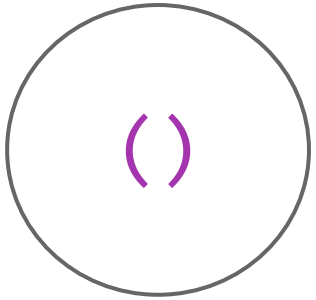
|Boolean| = 2



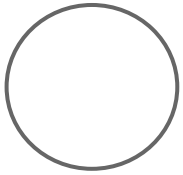
|Int| = 2^{32}



|Unit| = 1



|Nothing| = 0



¡¡*Cualquier* tipo no es más que un conjunto de valores!!

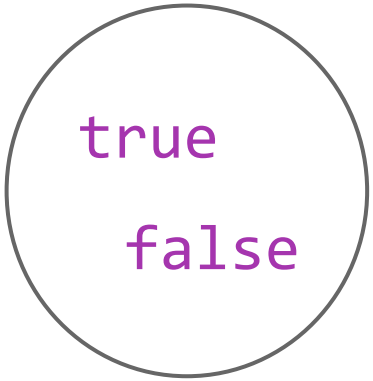
Algebraic Data Types (ADTs)

- Combinamos tipos para conseguir tipos más complejos (tipos derivados)
- Realizamos estas combinaciones mediante operaciones algebraicas (+, ×)
- ¡Por eso se llaman tipos algebraicos!

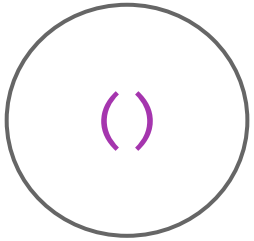
Productos

- Por ejemplo, podemos multiplicarlos...

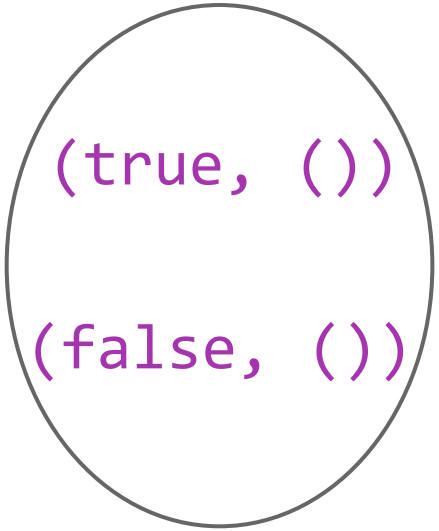
|Boolean| = 2



|Unit| = 1



|Boolean × Unit| = 2×1



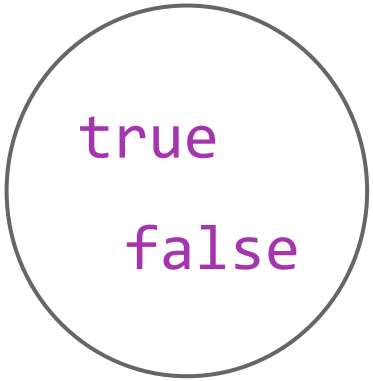
Ejemplos de productos

```scala
// Int x Boolean
type T1 = Tuple2[Int, Boolean]
val t1: T1 = (3, true)

// Unit x Int
type T2 = (Unit, Int)
val t2: T2 = ((), 4)

// String x Int
case class T3(name: String, age: Int)
val t3: T3 = T3("John Doe", 26)
```

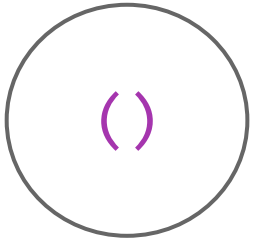
Sumas

- O también podemos sumarlos...

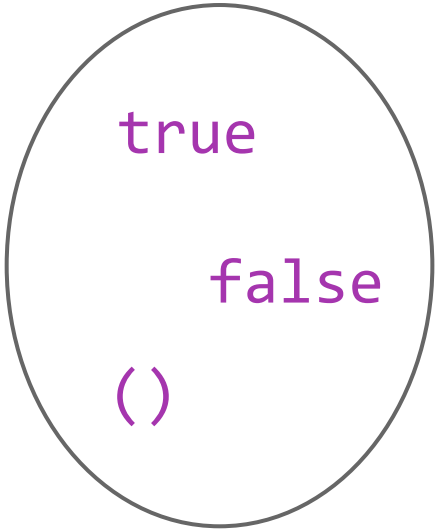
|Boolean| = 2



|Unit| = 1



|Boolean + Unit| = 2+1



Ejemplos de sumas

```scala
// Int + Boolean
type T1 = Either[Int, Boolean]
val t1: T1 = Left(4)

// Unit + Int
type T2 = Either[Unit, Int]
val t2: T2 = Right(18)

// Nothing + Boolean
type T3 = Either[Nothing, Boolean]
val t3: T3 = Right(true)
```

Algebraic data types

Tipos primitivos	`Int`
	`Boolean`
	`Unit`
	`Nothing`
	`...`
		$\|Int\| = 2^{32}$
		$\|Boolean\| = 2$
		$\|Unit\| = 1$
		$\|Nothing\| = 0$
		`...`

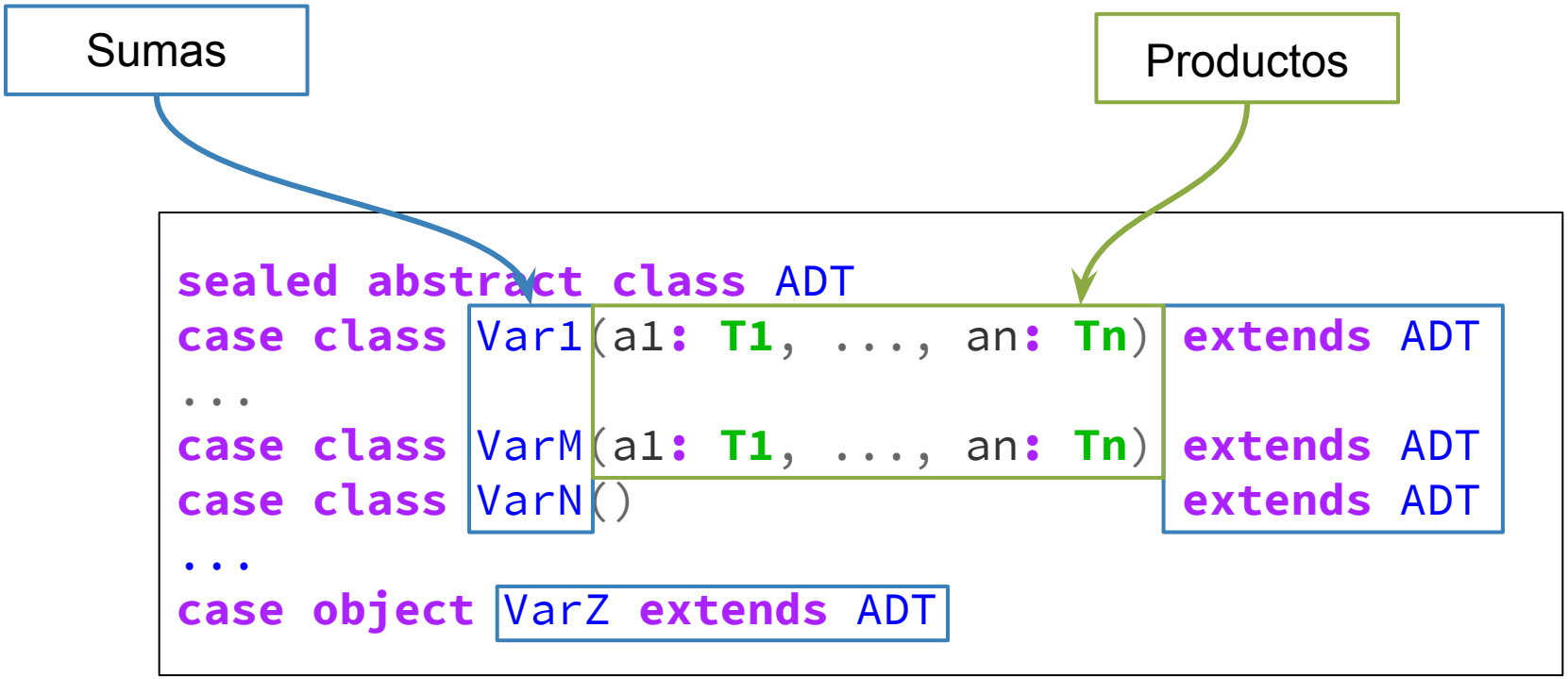
Tipos suma (coproductos)	`Int + Boolean`
	`Unit + Int`
	`Nothing + Boolean`
	`...`
		$\|Int\|+2$
		$1+\|Int\|$
		$0+2$
		`...`

Tipos producto	`Unit × Boolean`
	`Unit × Int`
	`Nothing × Boolean`
	`...`
		1×2
		$1\times\|Int\|$
		0×2

ADTs

Los ADTs son tipos compuestos!

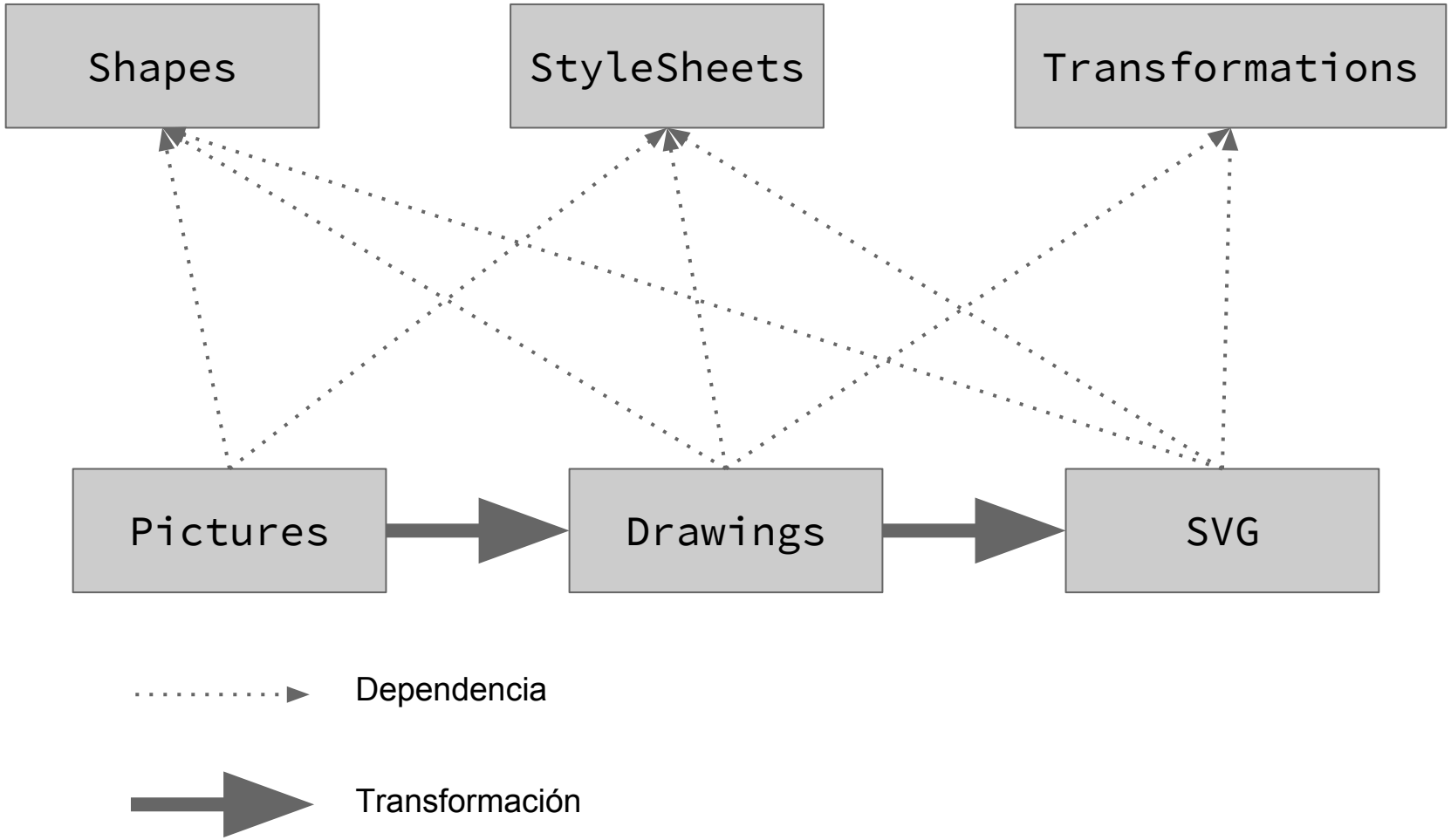
ADTs en Scala (design pattern)



Un caso real: Diagrams

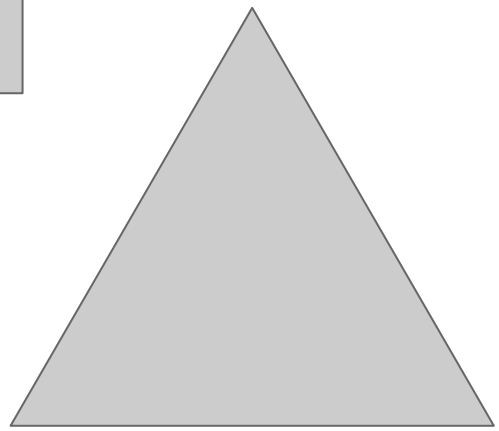
- Es un DSL para crear gráficos escrito en Haskell
- Aplica técnicas de programación funcional
  - Declaratividad
  - Modularidad
  - Lenguajes
  - Type classes
- Trabajaremos con un subconjunto implementado en Scala

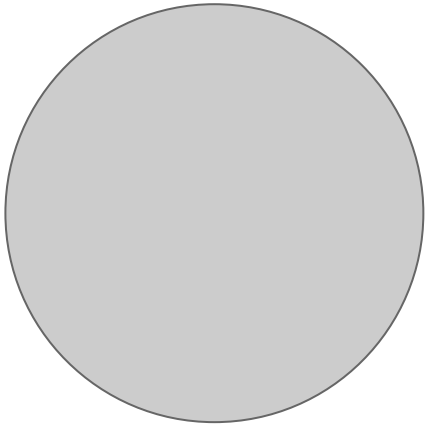
Diagrams


Shapes
StyleSheets
Transformations
Pictures
Drawings
SVG
Dependencia
Transformación

Shapes

```
sealed trait Shape
case class Rectangle(width: Double, height: Double) extends Shape
case class Circle(radius: Double) extends Shape
case class Triangle(width: Double) extends Shape
```



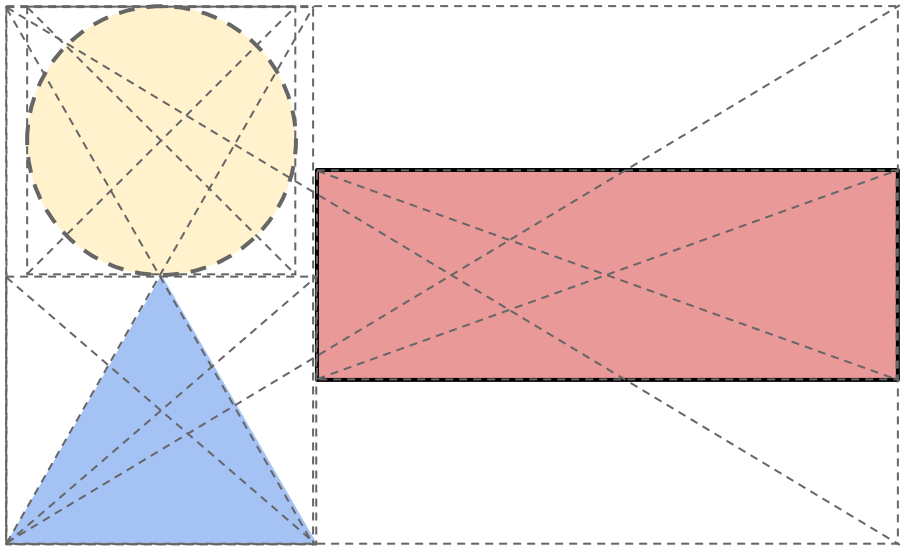


StyleSheet

```
type StyleSheet = List[Styling]

sealed trait Styling
case class FillColor(c: Color) extends Styling
case class StrokeColor(c: Color) extends Styling
case class StrokeWidth(w: Double) extends Styling

sealed trait Color
case object Red extends Color
case object Blue extends Color
// ...
case object Alpha extends Color
```

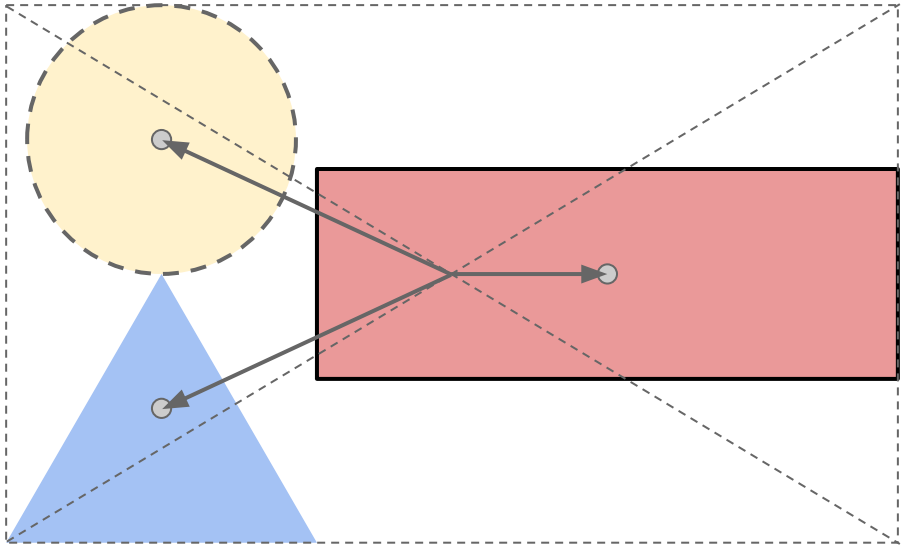
Picture

```
sealed trait Picture
case class Place(style: StyleSheet, shape: Shape) extends Picture
case class Above(top: Picture, bottom: Picture) extends Picture
case class Beside(left: Picture, right: Picture) extends Picture
```



Drawing

```
type Drawing = List[(Transform, StyleSheet, Shape)]
```



¿Qué es la programación funcional?

- Funciones y ADTs
- **¿Por qué PF? ¡Modularidad!**
- La escalera de modularidad

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

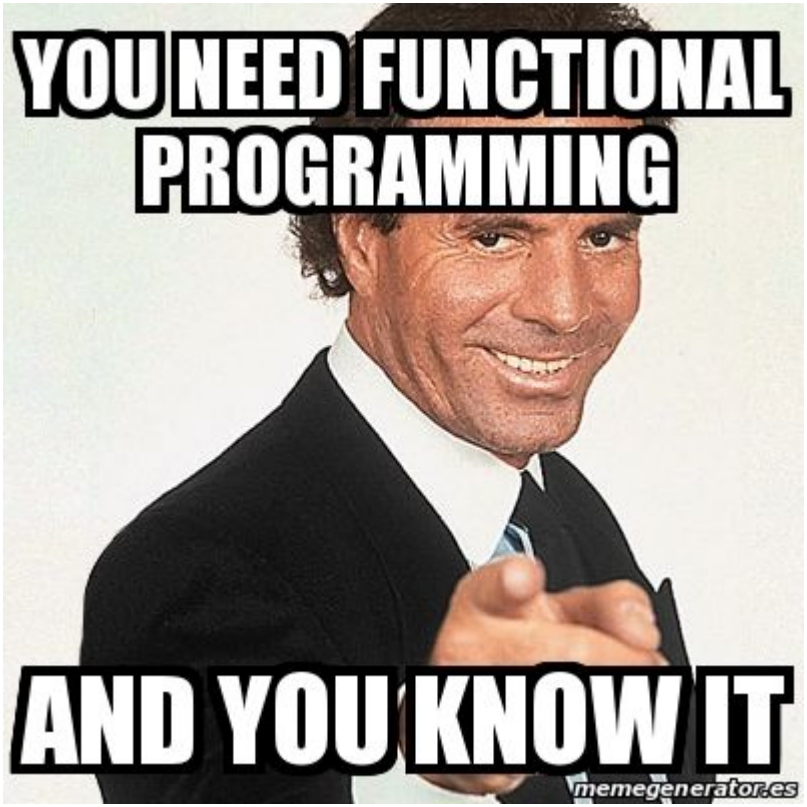
POLIMORFISMO PARAMÉTRICO

FUNCIONES

¿Por qué debería utilizar siquiera PF?

Si quieres que tus programas sean fácilmente...

- Entendibles
- Testeables
- Mantenibles
- Optimizables
- Reusables
- Componibles
- ...



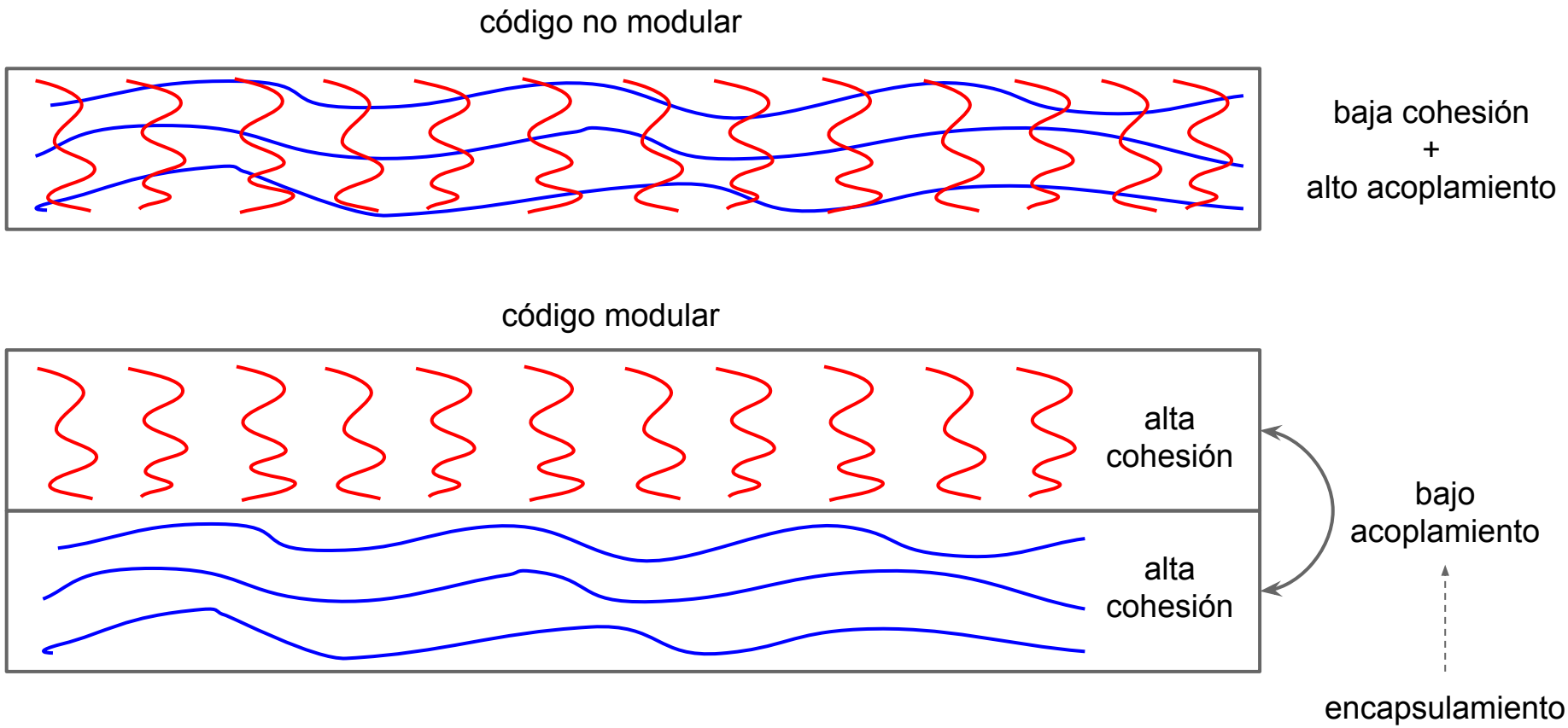
¿Cómo es capaz la PF de ofrecer esas garantías no funcionales?

- Mediante técnicas de modularidad
  - Funciones
  - Polimorfismo paramétrico
  - Funciones de orden superior
  - Type classes (polimorfismo Ad Hoc)
  - Datatype generics
  - Evaluación perezosa
  - Lenguajes
  - ...

¿Qué es la modularidad?

- Código monolítico
  - Distintos conceptos entrelazados en el mismo módulo
  - Difícil de entender, probar, reusar, mantener, etc.
- Código modular
  - Cada concepto se mantiene en diferentes módulos, y se combinan entre sí para obtener la misma funcionalidad pero con mejores garantías no funcionales
  - Fácil de entender, testear, reutilizar, etc.

¿Qué es la modularidad?
¡Alta cohesión y bajo acoplamiento!

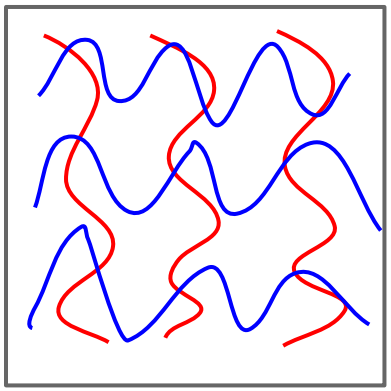


¿Qué son las técnicas de modularidad por abstracción?

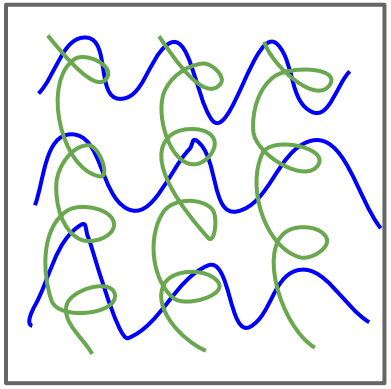
- Todas promueven la reutilización mediante
  - *Identificación de patrones* recurrentes
  - *Abstracción* de la parte específica
- En esencia, se diferencian en el tipo de entidades que abstraen y en los patrones comunes que identifican

¿Qué son las técnicas de modularidad por abstracción?

p1

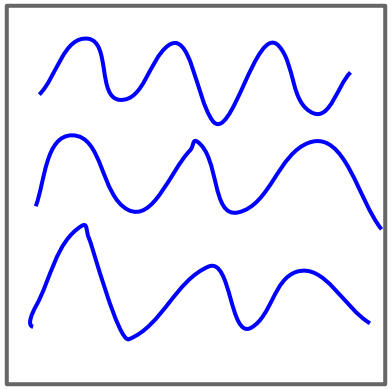


p2



módulo abstracto

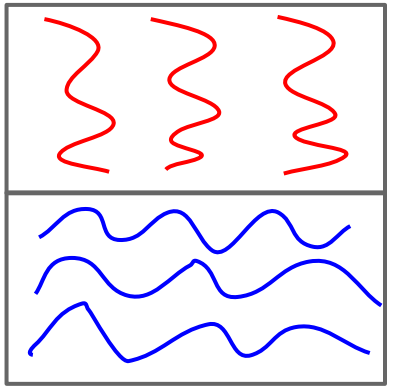
- función
- función genérica
- HOF
- type class
- lenguaje
- …



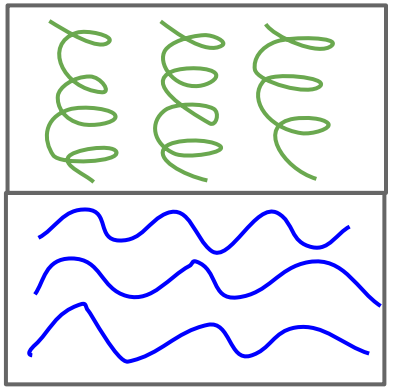
p1 ~= p1 modular

p2 ~= p2 modular

p1 modular



p2 modular



¿Qué es la programación funcional?

- Funciones y ADTs
- ¿Por qué PF? ¡Modularidad!
- **La escalera de modularidad**

....

LENGUAJES

TYPE CLASSES

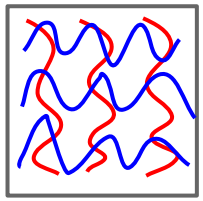
FUNCIONES DE ORDEN SUPERIOR

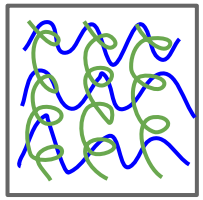
POLIMORFISMO PARAMÉTRICO

FUNCIONES

Funciones con parámetros como mecanismo de modularidad

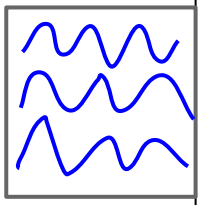
```
// (I) Monolithic programs
val url: String = config.get("URL") match {
  case Some(u) => u
  case None => "default.url"
}

val port: String = config.get("PORT") match {
  case Some(p) => p
  case None => "8080"
}
```

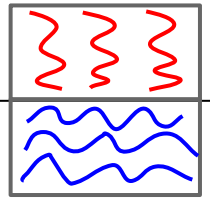


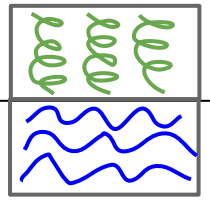


Funciones con parámetros como mecanismo de modularidad

```
// (II) Abstraction
def getConfigProperty(name: String, default: String) =
  config.get(name) match {
    case Some(p) => p
    case None => default
  }

// (III) Modularised
val url: String = getConfigProperty("URL", "default.url")
val port: String = getConfigProperty("PORT", "8080")
```







¿Qué es la programación funcional?

- Funciones y ADTs
- ¿Por qué PF? ¡Modularidad!
- **La escalera de modularidad**

....

LENGUAJES

TYPE CLASSES

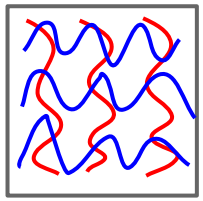
FUNCIONES DE ORDEN SUPERIOR

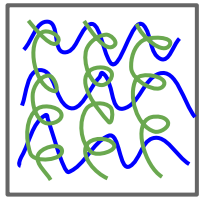
POLIMORFISMO PARAMÉTRICO

FUNCIONES

Polimorfismo paramétrico como mecanismo de modularidad (tipos)

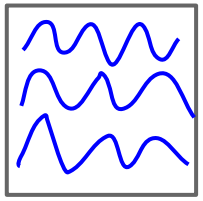
```scala
// (I) Monolithic types
sealed trait ListaString
case class NilString() extends ListaString
case class ConsString(
  elemento: String,
  resto: ListaString) extends ListaString

sealed trait ListaBoolean
case class NilBoolean() extends ListaBoolean
case class ConsBoolean(
  elemento: Boolean,
  resto: ListaBoolean) extends ListaBoolean
```

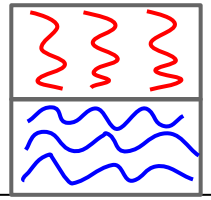


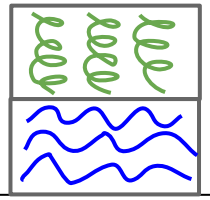


Polimorfismo paramétrico como mecanismo de modularidad (tipos)

```scala
// (II) Abstraction
sealed trait Lista[T]
case class Nil[T]()    extends Lista[T]
case class Cons[T](
  elemento: T      ,
  resto: Lista[T]   ) extends Lista[T]

// (III) Modularised
type ListaString = Lista[String]
type ListaBoolean = Lista[Boolean]
```

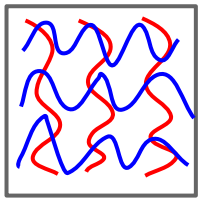


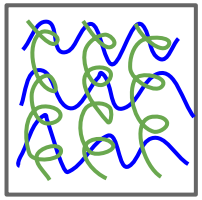




Polimorfismo paramétrico como mecanismo de modularidad (funcs.)

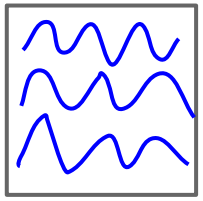
```scala
// (I) Monolithic functions
def duplicateInt(l: List[Int]): List[Int] =
  l match {
    case Nil => Nil
    case head :: tail =>
      head :: head :: duplicateInt(tail)
  }

def duplicateString(l: List[String]): List[String] =
  l match {
    case Nil => Nil
    case head :: tail =>
      head :: head :: duplicateString(tail)
  }
```

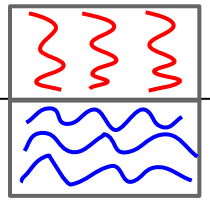


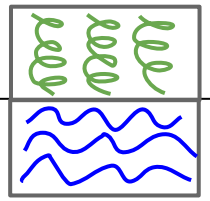


Polimorfismo paramétrico como mecanismo de modularidad (funcs.)

```scala
// (II) Abstraction
def duplicate[A](l: List[A]): List[A] =
  l match {
    case Nil => Nil
    case head :: tail =>
      head :: head :: duplicate(tail)
  }

// (III) Modularised
def duplicateInt(l: List[Int]): List[Int] =
  duplicate[Int](l)
def duplicateString(l: List[String]): List[String] =
  duplicate[String](l)
```







¿Qué es la programación funcional?

- Funciones y ADTs
- ¿Por qué PF? ¡Modularidad!
- **La escalera de modularidad**

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES

¿Qué son las funciones de orden superior?

Son funciones que reciben como argumentos o devuelven como resultados otras funciones

Funciones como objetos en Scala

FUNCTIONS AS OBJECTS

```scala
val f1: Function1[String, Int] =
  new Function1[String, Int] {
    def apply(s: String): Int = s.length
  }

val g1: String => Int = s => s.length
```

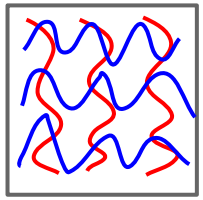
```scala
// eta expansion η
val f1: String => Int = m1 _
```

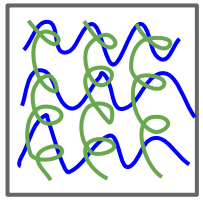
FUNCTIONS AS METHODS

```scala
def m1(s: String): Int = s.length
```

Funciones de orden superior

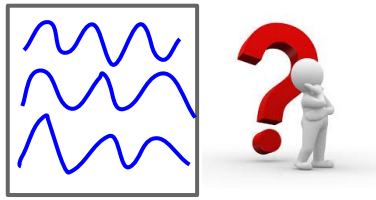
```
type Drawing = List[(Transform, StyleSheet, Shape)]

// (I) Monolithic
def filterRed(d: Drawing): Drawing = d match {
  case Nil => Nil
  case head :: tail =>
    if (head._2.contains(FillColor(Red))) head :: filterRed(tail)
    else filterRed(tail)
}

def filterCircle(d: Drawing): Drawing = d match {
  case Nil => Nil
  case head :: tail =>
    if (head._3 == Circle) head :: filterCircle(tail)
    else filterCircle(tail)
}
```





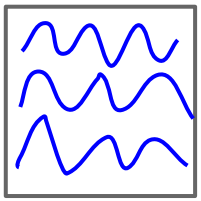
Funciones de orden superior

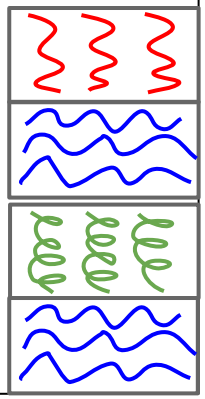
```scala
// (II) Abstraction
def filter(d: Drawing)
    (cond: (Transform, SytleSheet, Shape) => Boolean): Drawing =
  d match {
    case Nil => Nil
    case head :: tail =>
      if (cond(head)) head :: filter(tail)
      else filter(tail)
  }

// (III) Modularised
def filterRed(d: Drawing): Drawing =
  filter(d)(_._2.contains(FillColor(Red)))
def filterCircle(d: Drawing): Drawing =
  filter(d)(_._3 == Circle)
```



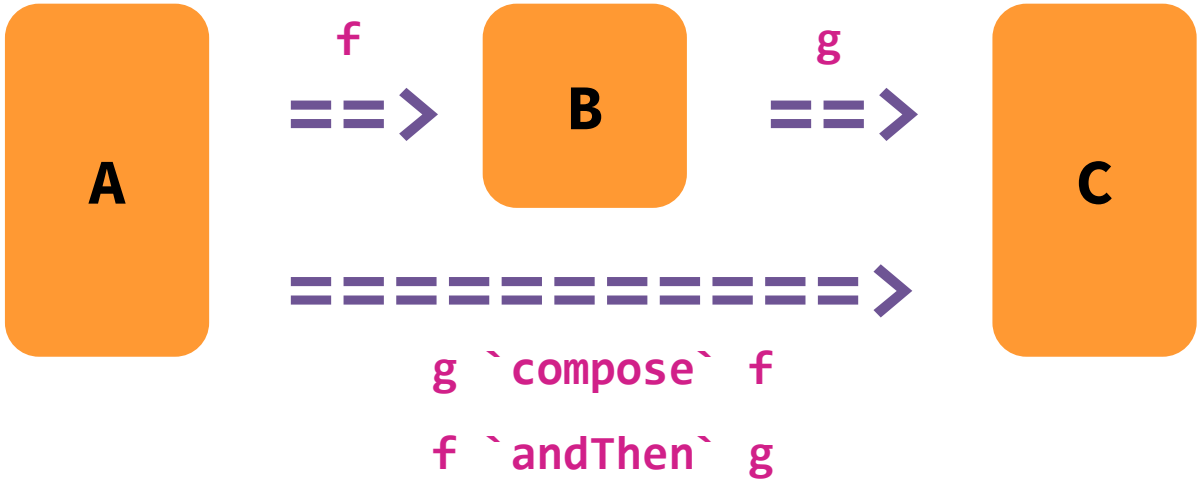
Funciones de orden superior

```scala
// (II) Abstraction
def filter[A](l: List[A])
    (cond: A => Boolean): List[A] =
  l match {
    case Nil => Nil
    case head :: tail =>
      if (cond(head)) head :: filter(tail)(cond)
      else filter(tail)(cond)
  }

// (III) Modularised
def filterRed(d: Drawing): Drawing =
  filter(d)(_._2.contains(FillColor(Red)))
def filterCircle(d: Drawing): Drawing =
  filter(d)(_._3 == Circle)
```





Composición de funciones



Ejercicio: Implementar

```
def compose[A,B,C](g: B => C, f: A => B): A => C
def andThen[A,B,C](f: A => B, g: B => C): A => C
```

dentro del `object ComposicionFunciones` en **tema1-hofs/EjerciciosClase.scala**

Hall of Fame HOFs (e.g.: List)

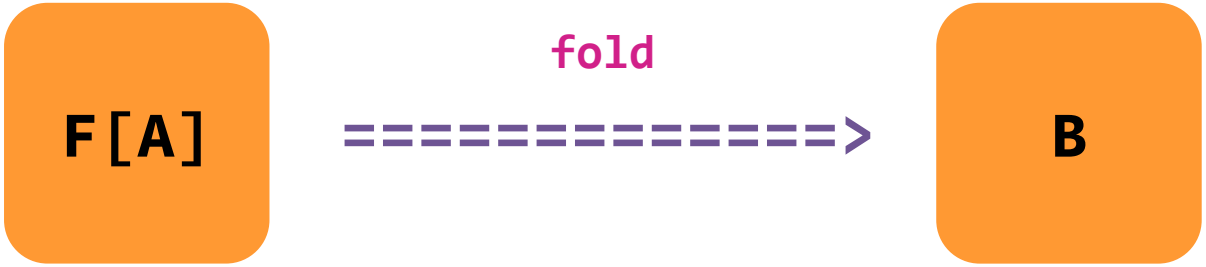
```
foldRight[B]    (l: List[A])(z: B)(op: (A, B) => B) : B
foldLeft[B]     (l: List[A])(z: B)(op: (B, A) => B) : B
reduce[A1 >: A](l: List[A])(op: (A1, A1) => A1)     : A1
map[B]          (l: List[A])(f: (A) => B)           : List[B]
filter          (l: List[A])(p: (A) => Boolean)     : List[A]
flatMap[B]      (l: List[A])(f: (A) => List[B])     : List[B]
// ...
```

Hall of Fame HOFs (e.g.: Tree)

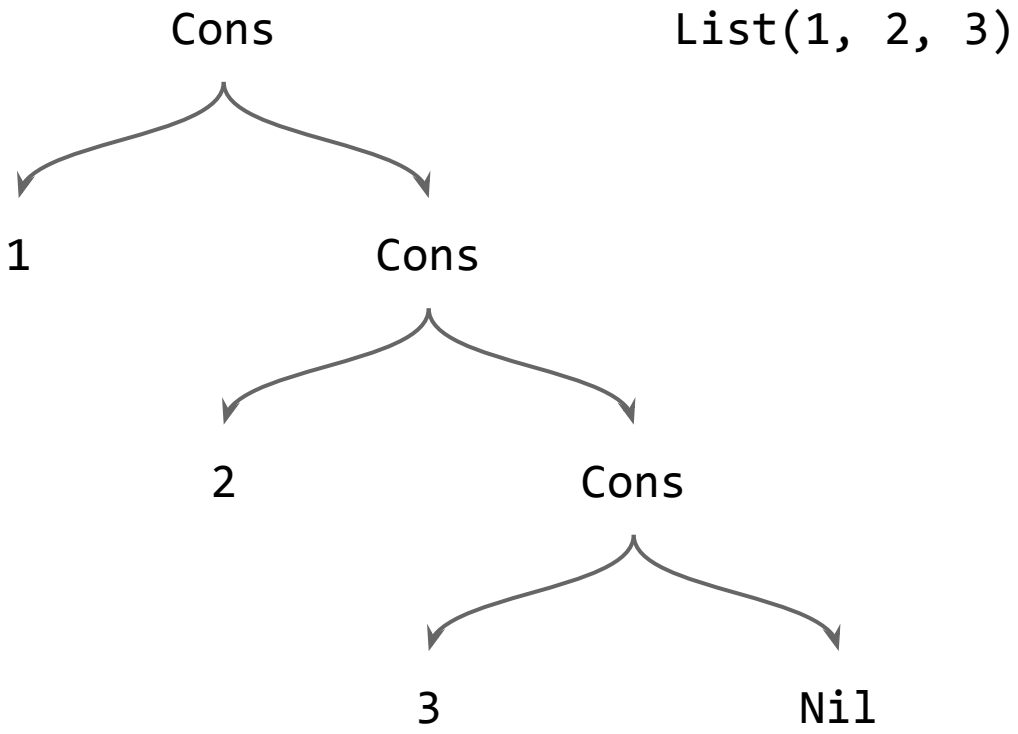
```
foldRight[A, B]   (t: Tree[A])(z: B)(op: (A, B) => B)     : B
foldLeft[A, B]    (t: Tree[A])(z: B)(op: (B, A) => B)     : B
fold[A, B]        (t: Tree[A])(z: B)(op: (B, A, B) => B) : B
reduce[A, A1 >: A](t: Tree[A])(op: (A1, A1) => A1)        : A1
map[A, B]         (t: Tree[A])(f: A => B)                 : Tree[B]
flatMap[A, B]     (t: Tree[A])(f: A => Tree[B])           : Tree[B]
filter[A]         (t: Tree[A])(p: A => Boolean)           : Tree[A]
// ...
```

Fold (catamorfismo)

- Se puede ver como una forma de *interpretar/consumir/plegar* un ADT
- Se puede definir para cualquier ADT. La definición del fold está determinada por la estructura del ADT.

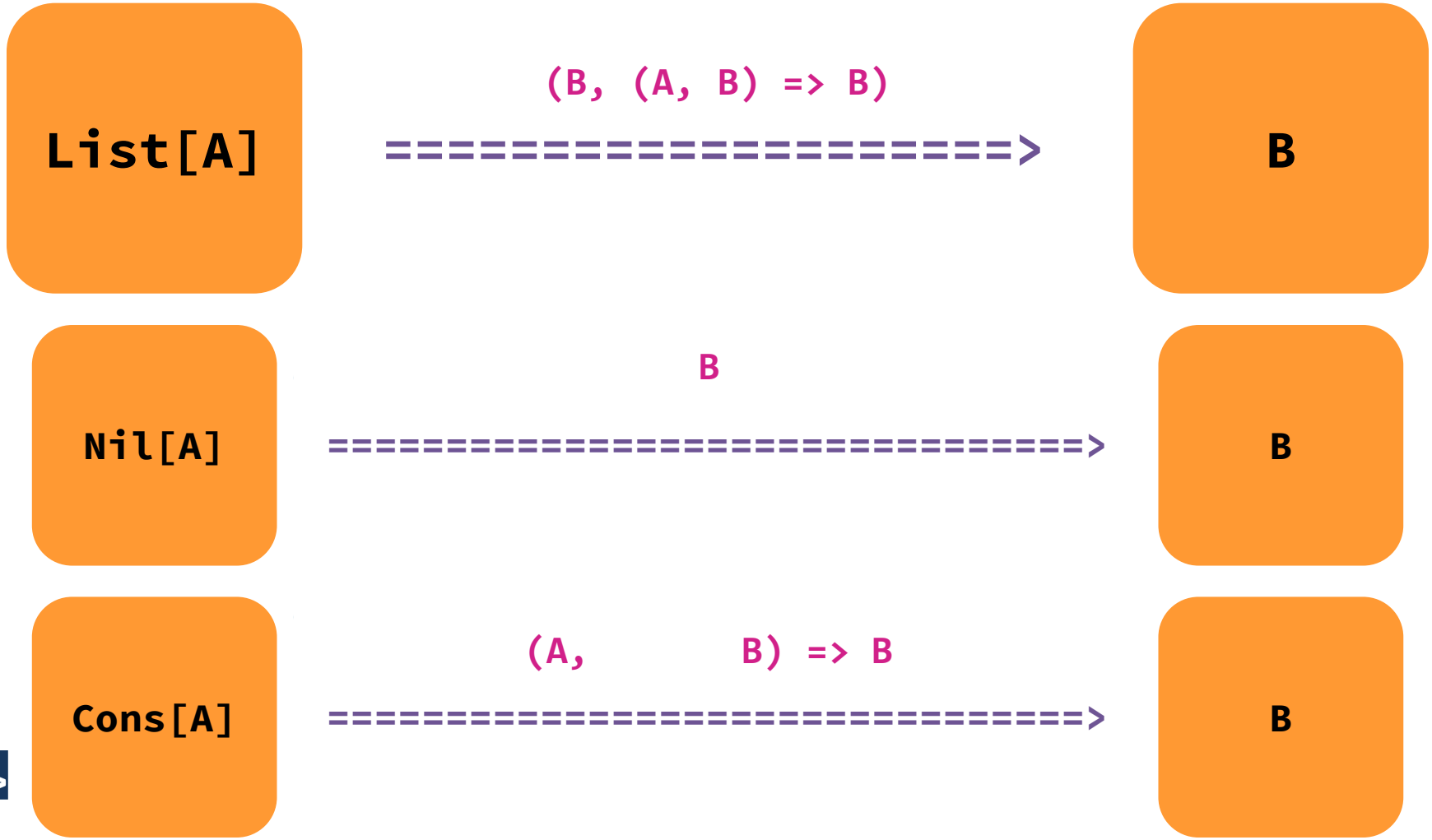


Fold en List

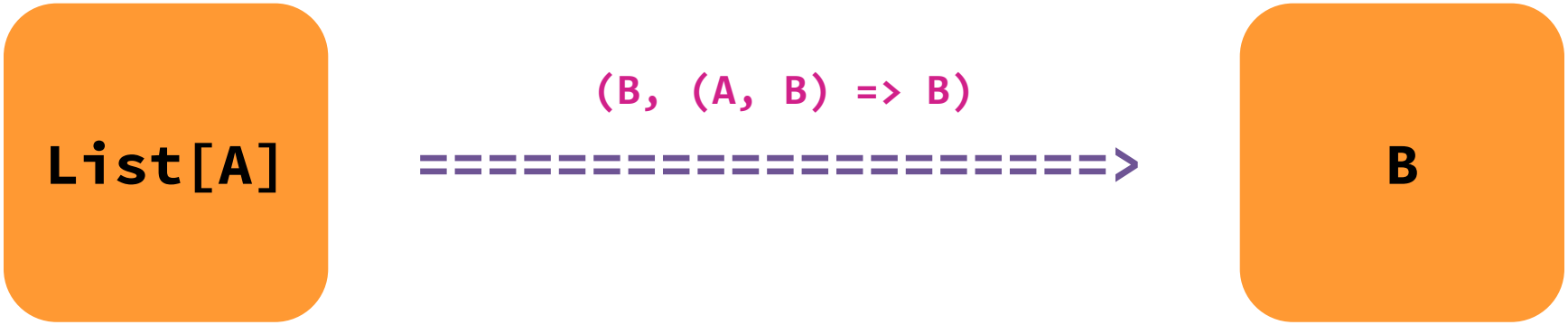



```
sealed abstract class List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case class Nil[A]() extends List[A]
```

Fold en List

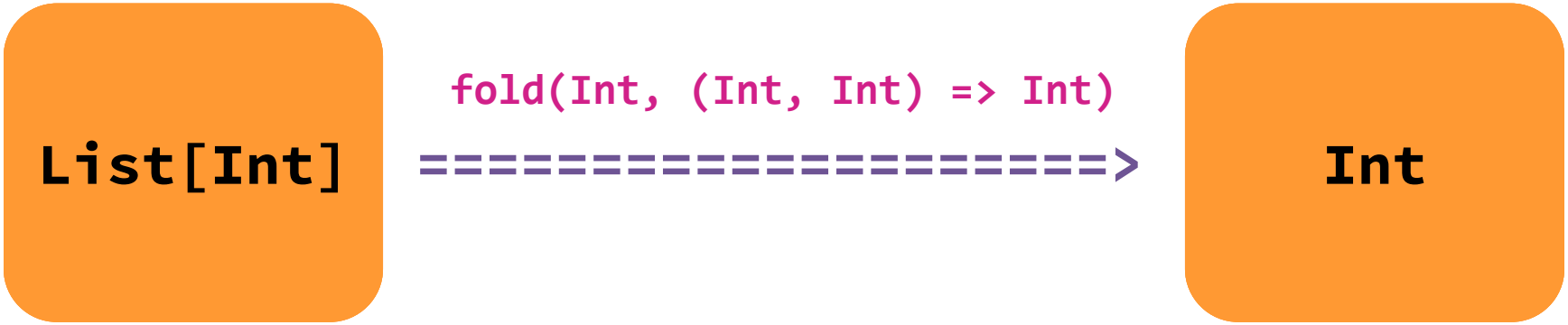


Fold en List



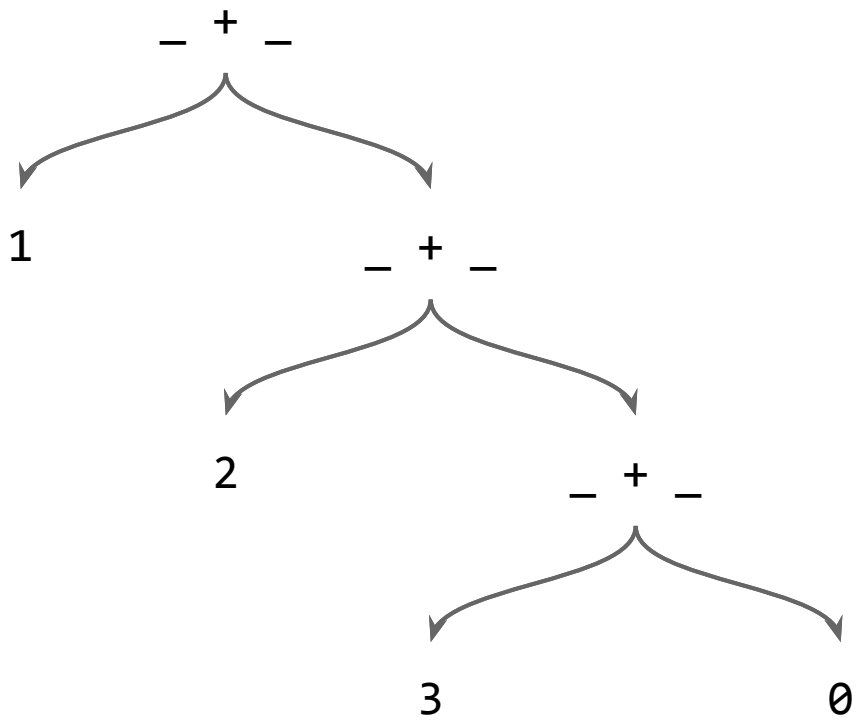

```
def fold[A, B](l: List[A])
    (nil: B, cons: (A, B) => B): B =
  l match {
    case Cons(h, t) => cons(h, fold(t)(nil, cons))
    case Nil() => nil
  }
```

Ejemplo: Suma



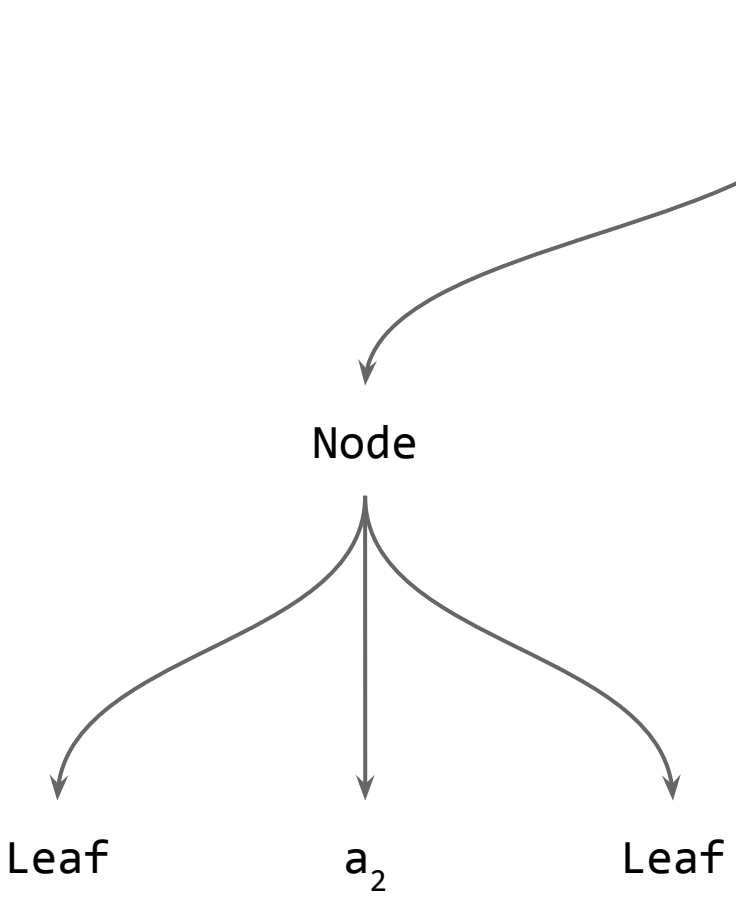

```
nil = 0

cons = (i1: Int, i2: Int) => i1 + i2
```

Ejemplo: Suma




```
fold(0, _ + _)
```

Fold en Tree




```
sealed abstract class Tree[A]
case class Node[A](
  l: Tree[A], a: A, r: Tree[A])
  extends Tree[A]
case class Leaf[A]() extends Tree[A]
```

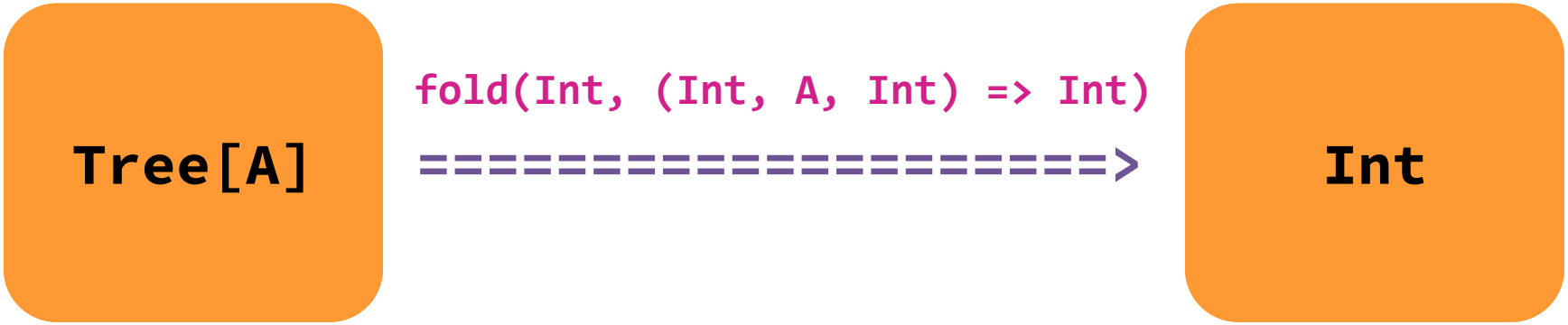
Fold en Drawing (SVG)

```
val d: Drawing = ???

d.foldRight("") {
  case ((t, ss, Rectangle(width, height)), acc) =>
    val Pos(x, y) = transformPos(t)(Pos(-width/2, -height/2))
    s"""<rect x="$x" y="$y" width="$width" height="$height" ${styleToSVG(ss)}
/>""" + acc
  case ((t, ss, Circle(radius)), acc) =>
    val Pos(x, y) = transformPos(t)(Pos(0, 0))
    s"""<circle r="$radius" cx="$x" cy="$y" ${styleToSVG(ss)} />""" + acc
  case ((t, ss, Triangle(width)), acc) =>
    val height = math.sqrt(3) * width / 2
    val Pos(x, y) = transformPos(t)(Pos(-width/2, -height/2))
    s"""<polygon points="$x,$y ${x+(width/2)},${y+height} ${x+width},$y"
${styleToSVG(ss)} />""" + acc
}
```

Fold en Drawing (SVG)

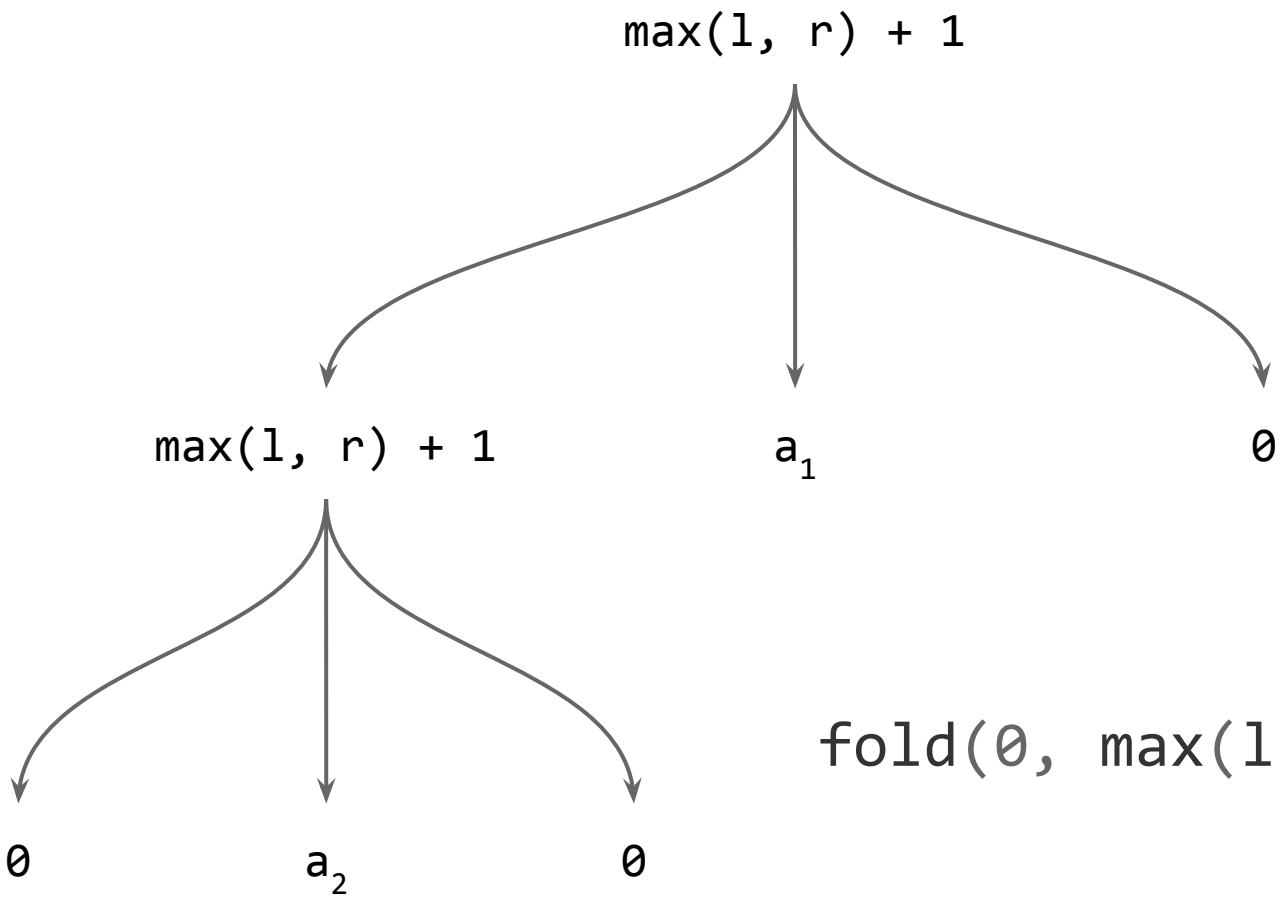
```
<svg width="30.0" height="16.660254037844386" xmlns="http://www.w3.
org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
  <g transform="translate(15.0, 8.330127018922193)">
    <g transform="scale(1, -1)">
      <circle r="4.0" cx="-10.0" cy="4.330127018922193"  />
      <polygon points="-15.0,-8.330127018922193 -10.0,0.33012701892219276 -5.0,
-8.330127018922193" fill="#00f"  />
      <rect x="-5.0" y="-3.5" width="20.0" height="7.0" fill="#f00"  />
    </g>
  </g>
</svg>
```

Ejemplo: Altura




```
leaf = 0

node = (l: Int, a: A, r: Int) => max(l, r) + 1
```

Ejemplo: Altura




```
fold(0, max(l, r) + 1)
```

Ejercicio

Ejercicio: Implementar **filter** en función de **fold** para instancias de la clase **List** dentro del **object** FilterEnFuncionDeFoldParaListas en **tema1-hofs/EjerciciosClase.scala**

Conclusiones

- La PF aporta **modularidad** mediante distintos mecanismos: funciones, polimorfismo paramétrico, HOFs, etc.
- Los **ADTs** son tipos definidos por el programador mediante *sumas* y *productos*
  - Las sumas representan los distintos tipos de *constructores*
- Los ADTs no son ***clases*** orientadas a objetos
  - No son extensibles
  - No encapsulan comportamiento

Conclusiones

- Las **funciones de orden superior** son uno de los mecanismos básicos de modularidad en PF
  - *Concisión*
  - *Reusabilidad*
  - *Composicionalidad*
  - *etc.*
- Los ADTs se construyen y posteriormente se *consumen* o *interpretan*
  - Asociado a cada *constructor* encontraremos un *destructor* o consumidor
  - Los **catamorfismos** son una clase de intérpretes de ADTs muy común

¿Quiéres saber más? (papers)

- *Why Functional Programming Matters*

  http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html

  "*In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity.*"

- *How Functional Programming Mattered*

  http://nsr.oxfordjournals.org/content/early/2015/07/13/nsr.nwv042

  "*In this paper, we review the impact of functional programming, focusing on how it has changed the way we may construct programs, the way we may verify programs, and fundamentally the way we may think about programs.*"

- Datatype-Generic Programming (section 2)

  http://www.cs.ox.ac.uk/jeremy.gibbons/publications/dgp.pdf

  "*In this section, we review a number of interpretations of 'genericity' in terms of the kind of parametrization they support*"

¿Quiéres saber más? (recursos)

- *The Algebra of Algebraic Data Types*

  http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/

- *Your data structures are made of maths!*

  http://www.slideshare.net/kenbot/your-data-structures-are-made-of-maths

Ejercicios para casa

- **`Ejercicio 1`** **`tema1-hofs/homework/Ejercicio1.scala`**
  - Implementa funciones de orden superior **`filter`** y **`map`** sobre el tipo **`Option`** de la librería estándar utilizando *pattern matching*
  - Crea el **`fold`** (catamorfismo) y modulariza las funciones anteriores

- **`Ejercicio 2`** **`tema1-hofs/homework/Ejercicio2.scala`**
  - Define un ADT para árboles binarios con valores en los nodos
  - Implementa funciones para este ADT utilizando *pattern matching*
  - Crea el **`fold`** (catamorfismo) para este ADT y modulariza las funciones anteriores

- **`Ejercicio 3`** **`tema1-hofs/homework/Ejercicio3.scala`**
  - Igual que el Ejercicio 2, pero con un ADT que representa expresiones aritméticas con números enteros