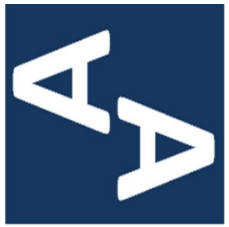
Programación funcional en Scala

4. El ecosistema funcional de Scala

4.1 Scalaz

4.2 Scalacheck



Habla Computing
info@hablapps.com
@hablapps

Objetivos

- ¿Es necesario implementarse los monoides, las mónadas, etc.?
- ¿Qué ofrece el ecosistema de Scala a los programadores funcionales?
- ¿Qué nos ofrece **Scalaz**?
- ¿Qué nos ofrece **Scalacheck**?

El ecosistema funcional de Scala

- **Sección 0 - El ecosistema funcional de Scala**
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

Typelevel.org

- Shapeless
  - Type-level, generic programming
- Cats
  - Abstracciones de Programación Funcional
- Macros y plugins del compilador
  - Simulacrum, kind-projector, ...
- Scalacheck
  - Property-based testing
- Spire
  - Librería numérica
- ....

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- **Sección 1 - Scalaz intro**
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

¿Qué es Scalaz?

- Una librería de *type classes*
- Define una gran cantidad de ***type classes***
  - De carácter funcional y general
- Define nuevos **tipos de datos**
  - Para suplir tipos de datos *"erróneos"* en *stdlib*
  - Nuevos tipos de datos no existentes en *stdlib*
- Proporciona instancias de dichas *type classes* tanto a los tipos de la librería estándar de **Scala**, como a los propios de **Scalaz**
  - Por lo tanto, se puede considerar también como una extensión de funcionalidad de la librería estándar de **Scala**

Type classes (v7.2.0)

Align
Applicative
ApplicativePlus
Apply
Arrow
Associative
Bifoldable
Bifunctor
Bind
BindRec
Bitraverse
Catchable
Category
Choice
Cobind
Comonad
Compose
Contravariant
Cozip
Divide
Divisible
Enum
Equal
Foldable
Foldable1
Functor
InvariantFunctor
IsEmpty
Monad
MonadError
MonadListen
MonadPlus
MonadTell
MonadTrans
Monoid
Nondeterminism
Optional
Order
Plus
PlusEmpty
ProChoice
Profunctor
Representable
Corepresentable
Semigroup
Show
Split
Strong
Traverse
Traverse1
Unzip
Zip

(type classes in **bold** face are included in the *typeclassopedia*)
https://wiki.haskell.org/Typeclassopedia

Data types (propios)

Either
Free
FreeT
Kleisli
Id
NonEmptyList
Maybe
Reducer
State
Tag
These
Tree
Validation
Writer

Data types (estándar)

AnyVal
Either
Function
Future
Iterable
List
Map
Option
PartialFunction
Set
SortedMap
Stream
String
Try
Tuple
TypeConstraint
Vector

Instancias librería estándar

Instancias propias Scalaz

\ **Type class** **Data type**	Monoid	Equal	Functor	Show
AnyVal	✔			
Either	✔	✔	✔	✔
Function	✔			
Future	✔			
Iterable		✔		
List	✔	✔	✔	✔
Map	✔	✔		
Option	✔	✔	✔	✔
PartialFunction				
...				

Instancias librería estándar

Instancias propias Scalaz

\ Type class Data type	Monoid	Equal	Functor	Show
Either	✔	✔		✔
Kleisli	✔		✔	
Id		✔		✔
Maybe	✔	✔		✔
Reducer				
State			✔	
These				
Tree		✔		
Validation	✔	✔		✔
Writer	✔	✔	✔	✔

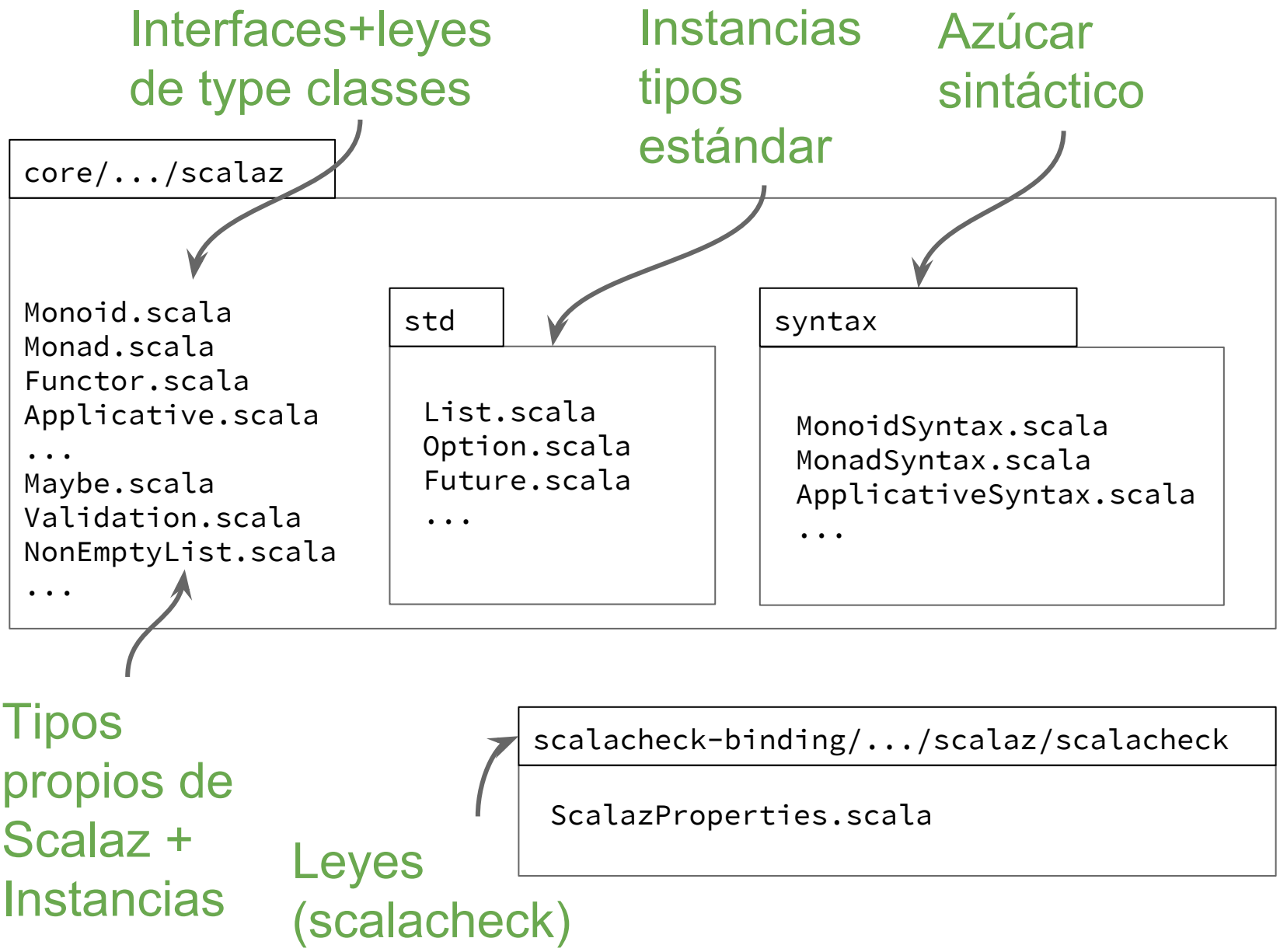
El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- **Sección 2 - Patrón de diseño de type classes**
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

Diseño de *type class* & implementación

Interfaz	Operaciones primitivas que deben implementarse para los tipos de la type class
Leyes	Propiedades que deben satisfacer dichas operaciones
Operaciones	Operaciones derivadas definidas en términos de las operaciones primitivas u otras operaciones derivadas; cuantas más operaciones derivadas tenga una type class, más **expresiva** será
Instancias	Instanciaciones de esta clase para diferentes tipos; cuantas más instancias se pueden dar de una type class, más **general** será
Sintaxis	Azúcar sintáctico para facilitar el uso de la type class

¿Dónde puedo encontrar … ?



Interfaz	Leyes	Instancias	Ops	Sintaxis

```scala
package scalaz

trait Semigroup[F]  {
  def append(f1: F, f2: => F): F
  ...
}
```

core/.../scalaz/Semigroup.scala

```scala
package scalaz

trait Monoid[F] extends Semigroup[F] {
  def zero: F
  ...
}
```

core/.../scalaz/Monoid.scala

```scala
package scalaz

trait Monoid[F] extends Semigroup[F] {
  ...
  trait MonoidLaw extends SemigroupLaw {
    def leftIdentity(a: F)(implicit F: Equal[F]) =
      F.equal(a, append(zero, a))
    def rightIdentity(a: F)(implicit F: Equal[F]) =
      F.equal(a, append(a, zero))
  }
  def monoidLaw = new MonoidLaw {}

}
```

core/.../scalaz/Monoid.scala

```scala
package scalaz.std

trait ListInstances extends ListInstances0 {
  implicit def listMonoid[A]: Monoid[List[A]] =
    new Monoid[List[A]] {
      def append(f1: List[A], f2: => List[A]) = f1 ::: f2
      def zero: List[A] = Nil
    }
  ...
}

object list extends ListInstances ...
```

core/.../scalaz/std/List.scala

```scala
package scalaz

trait Monoid[F] extends Semigroup[F] { self =>
  // derived functions
  def multiply(value: F, n: Int): F = ???

  def isMZero(a: F)(implicit eq: Equal[F]): Boolean = ???

  final def ifEmpty[B](a: F)(t: => B)(f: => B)(implicit
      eq: Equal[F]): B = ???
  ...
}
```

core/.../scalaz/Monoid.scala

Interfaz	Leyes	Instancias	Ops	Sintaxis

```
package scalaz.syntax

final class MonoidOps[F] private[syntax](val self: F)
    (implicit val F: Monoid[F]) extends Ops[F] {

  final def multiply(n: Int): F = F.multiply(self, n)

  final def ifEmpty[A](tv: =>A)(fv: =>A)(implicit
      e:Equal[F]): A =
    F.ifEmpty(self)(tv)(fv)

  final def isMZero(implicit e: Equal[F]): Boolean =
    F.isMZero(self)
  ...
}
```

core/.../scalaz/syntax/MonoidSyntax.scala

Interfaz	Leyes	Instancias	Ops	Sintaxis

```scala
package scalaz.syntax

trait ToMonoidOps extends ToSemigroupOps {
  implicit def ToMonoidOps[F](v: F)(implicit
      F0: Monoid[F]) =
    new MonoidOps[F](v)

  def mzero[F](implicit F: Monoid[F]): F = F.zero
  ...
}
```

core/.../scalaz/syntax/MonoidSyntax.scala

Interfaz	Leyes	Instancias	Ops	Sintaxis

```
package scalaz

package object syntax extends Syntaxes
```

core/.../scalaz/syntax/package.scala

```
package scalaz.syntax

trait Syntaxes {
  object monoid extends ToMonoidOps
  ...
}
```

core/.../scalaz/syntax/Syntaxes.scala

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - **Ejercicio 1 - Uso de monoides**
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

Ejercicio 1

Ejercicio 1: Implementa instancias de monoides para ciertos tipos de datos (String, option, …), y funciones sobre monoids

tema4-libs/ejercicio1/monoid.scala

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- **Sección 3 - Scalacheck intro**
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

Escribir estos tests es tedioso

```
"countTrues" should "return 0 if no list element is true" in {
  countTrue(List()) should be (0)
  countTrue(List(false)) should be (0)
  countTrue(List(false, false)) should be (0)
  countTrue(List(false, false, false)) should be (0)
}

it should "return 1 if there is exactly one true value" in {
  countTrue(List(true)) should be (1)
  ....
```

```
[info] SimpleUnitTesting:
[info] countTrues
[info] - should return 0 if no list element is true
[info] - should return 1 if there is exactly one true value
[info] - should return 2 if there are exactly two true values
```

Property-based testing al rescate

- Basado en la declaración de *propiedades* y la generación automática de *valores de entrada*

```
property("Equal to filtered size") =
  forAll{ l: List[Boolean] =>
    l.filter(identity).size == countTrue(l)
  }
```

```
[info] + countTrue.Equal to modularised version: OK, passed 100 tests.
[info] + countTrue.Equal to filtered size: OK, passed 100 tests.
[info] + countTrue.Equal to recursive spec: OK, passed 100 tests.
```

PBT & Programación Funcional

- Encaja perfectamente con **funciones puras**
  - No tenemos control sobre los valores de entrada de funciones impuras, y los resultados se escapan de su signatura
- Encaja perfectamente con **type classes**
  - Las leyes son las propiedades que se deben satisfacer
- Encaja perfectamente con el **diseño algebraico**, en general
  - Escribir propiedades nos fuerza a pensar sobre la especificación de nuestros programas (precondiciones inclusive)

¿Cómo se definen propiedades?

Propiedades proposicionales	
Prop(*bool*)	Si es *true*, es cierta; si es *false*, es falsa
A && B	Tanto A como B son ciertas
A || B	Al menos una entre A o B es cierta
A ==> B	Si A es cierta, entonces B es cierta
Propiedades cuantificativas (requieren generadores de valores de tipo T)	
forAll{ i: T => A(i)}	La propiedad A(*i*) es cierta para todo *i*
exists{ i: T => A(i)}	La propiedad A(i) es cierta al menos para un *i*

¿Cómo se crean los generadores de valores?

`fail[A]`	`Gen[A]`
`const[A]`	`A => Gen[A]`
`choose[A: Choose]`	`A => A => Gen[A]`
`oneOf[A]`	`Seq[A] => Gen[A]`
`frequency[A]`	`Seq[(Int,A)] => Gen[A]`
`pick[A]`	`Int => Iterable[A] => Gen[Seq[A]]`
`someOf[A]`	`Iterable[A] => Gen[Seq[A]]`

… y a partir de otros generadores también

suchThat[A]	:	Gen[A] => (A=>Boolean)	=> Gen[A]
map[A,B]	:	Gen[A] => (A=>B)	=> Gen[B]
flatMap[A,B]	:	Gen[A] => (A=>Gen[B])	=> Gen[B]
oneOf[A]	:	Seq[Gen[A]]	=> Gen[A]
frequency[A]	:	Seq[(Int,Gen[A])]	=> Gen[A]
sequence[Col:*B*,A]	:	Traversable[Gen[A]]	=> Gen[Col]
listOf[A]	:	Gen[A]	=> Gen[List[A]]
listOfN[A]	:	N => Gen[A]	=> Gen[List[A]]
nonEmptyListOf[A]	:	Gen[A]	=> Gen[List[A]]
mapOf[A,B]	:	Gen[(A,B)]	=> Gen[Map[(A,B)]]

Definiendo generadores

```scala
// Simple generators
val intGen: Gen[Int] = arbitrary[Int]
val stringGen: Gen[String] = arbitrary[String]

// Products
case class User(name: String, age: Int)
val userGen: Gen[User] =
  for {
    name <- stringGen
    age <- intGen
  } yield User(name, age)
```

Definiendo generadores

```
// Simple generators
val intGen: Gen[Int] = arbitrary[Int]
val stringGen: Gen[String] = arbitrary[String]

// Sums
val someIntGen: Gen[Option[Int]] =
  intGen map Option.apply
val noneIntGen: Gen[Option[Int]] =
  const(Option.empty[Int])
val optionIntGen: Gen[Option[Int]] =
  oneOf(someIntGen, noneIntGen)
```

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- **Sección 4 - Leyes de los monoides con Scalacheck**
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

```
package scalaz.scalacheck

object ScalazProperties {
  object monoid {
    ...
    def leftIdentity[A](implicit
        A: Monoid[A],
        eqa: Equal[A],
        arb: Arbitrary[A]) =
      forAll{ a: A => A.monoidLaw.leftIdentity(a) }

    def rightIdentity[A](...) = ...
  }
}
```

scalacheck-binding/.../ScalazProperties.scala

```scala
package scalaz.scalacheck

object ScalazProperties {
  object monoid {
    ...
    def laws[A](implicit
        A: Monoid[A],
        eqa: Equal[A],
        arb: Arbitrary[A]) = new Properties("monoid") {
      include(semigroup.laws[A])
      property("left identity") = leftIdentity[A]
      property("right identity") = rightIdentity[A]
    }
  }
}
```

scalacheck-binding/.../ScalazProperties.scala

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - **Ejercicio 2 - Probar una instancia de monoide**
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

Ejercicio 2

Ejercicio 2: Prueba que las instancias de monoides que dimos en el ***ejercicio 1*** cumplen con las leyes de los monoides

tema4-libs/ejercicio2/monoid-test.scala

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- **Sección 5 - Mónadas en Scalaz**
  - Ejercicio 3 - Uso de mónadas
- Sección 6 - Conclusión del curso

Interfaz	Leyes	Instancias	Ops	Sintaxis

```scala
trait Applicative[F[_]] extends Apply[F] {
  def point[A](a: => A): F[A]
}
```

core/.../scalaz/Applicative.scala

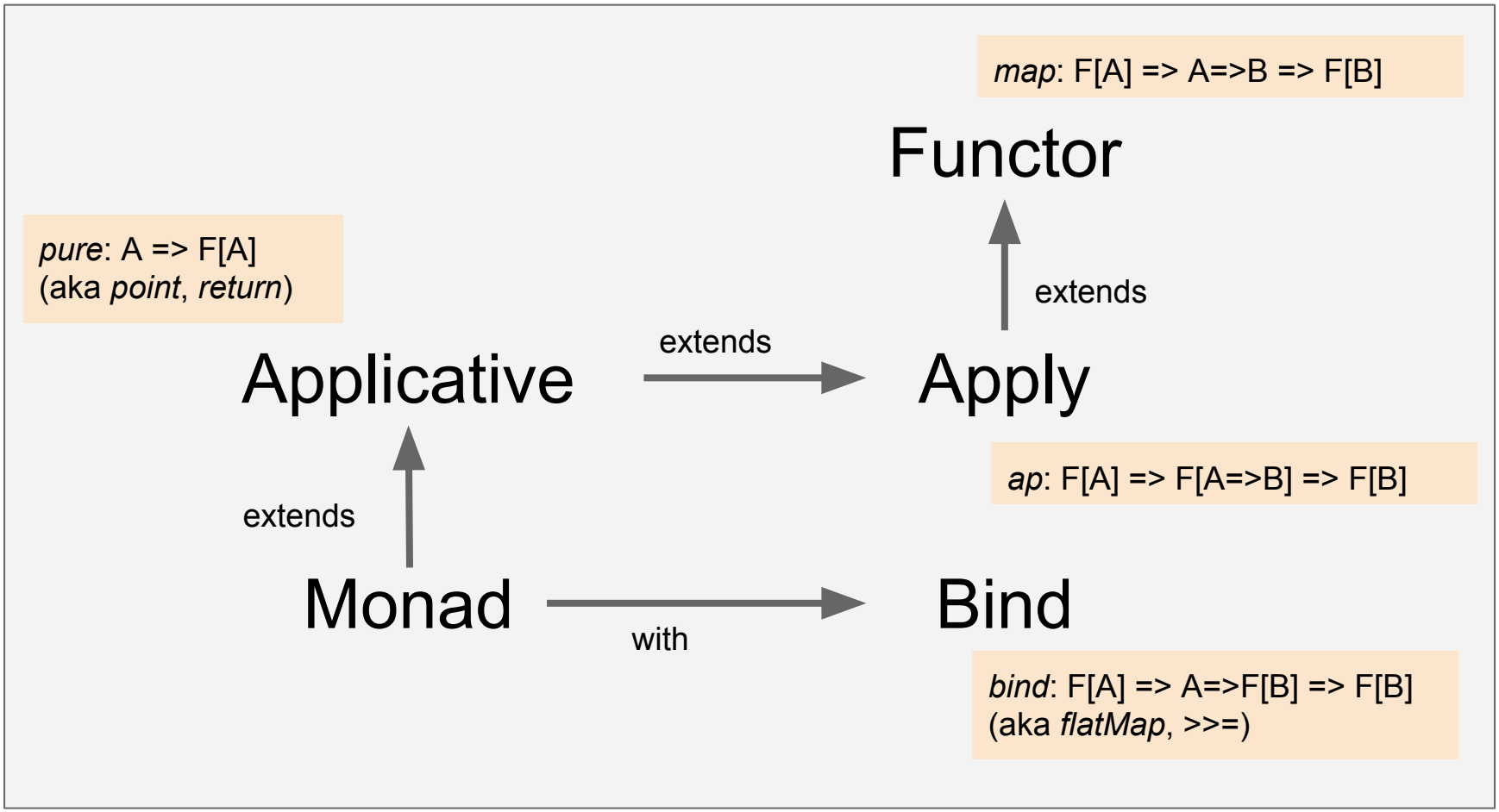
```scala
trait Bind[F[_]] extends Apply[F] {
  def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

core/.../scalaz/Bind.scala

```scala
trait Monad[F[_]] extends Applicative[F] with Bind[F]
```

core/.../scalaz/Monad.scala

La jerarquía de la type class Monad



Interfaz	Leyes	Instancias	Ops	Sintaxis

```scala
final class BindOps[F[_],A](val self: F[A])(
  implicit val F: Bind[F]) extends Ops[F[A]] {

  def >>=[B](f: A => F[B]) = F.bind(self)(f)

  def >>[B](b: => F[B]): F[B] = F.bind(self)(_ => b)

  def >>![B](f: A => F[B]): F[A] =
    F.bind(self)(a => F.map(f(a))(_ => a))
  ...
}
```

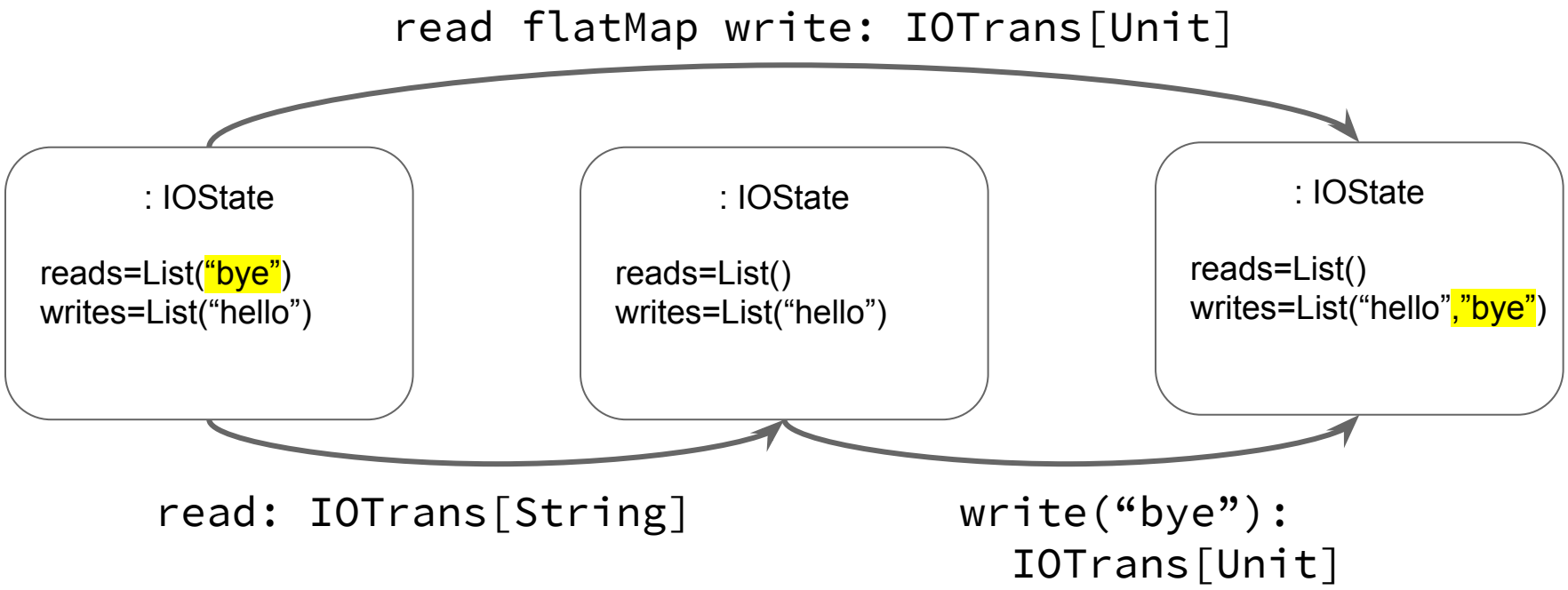
Scalaz disjunctions
Una alternativa a `Either`

```scala
import scalaz.\/
import scalaz.syntax.either._

type Error = String
def factorial(n: Int): \/[Error, Int] = {
  @scala.annotation.tailrec
  def go(_n: Int, acc: Int): Int =
    if (_n > 1) go(_n-1, acc*_n)
    else acc

  if (n >= 0) go(n, 1).right
  else "No se puede calcular el factorial ...".left
}
```

Scalaz disjunctions
Una alternativa a `Either`

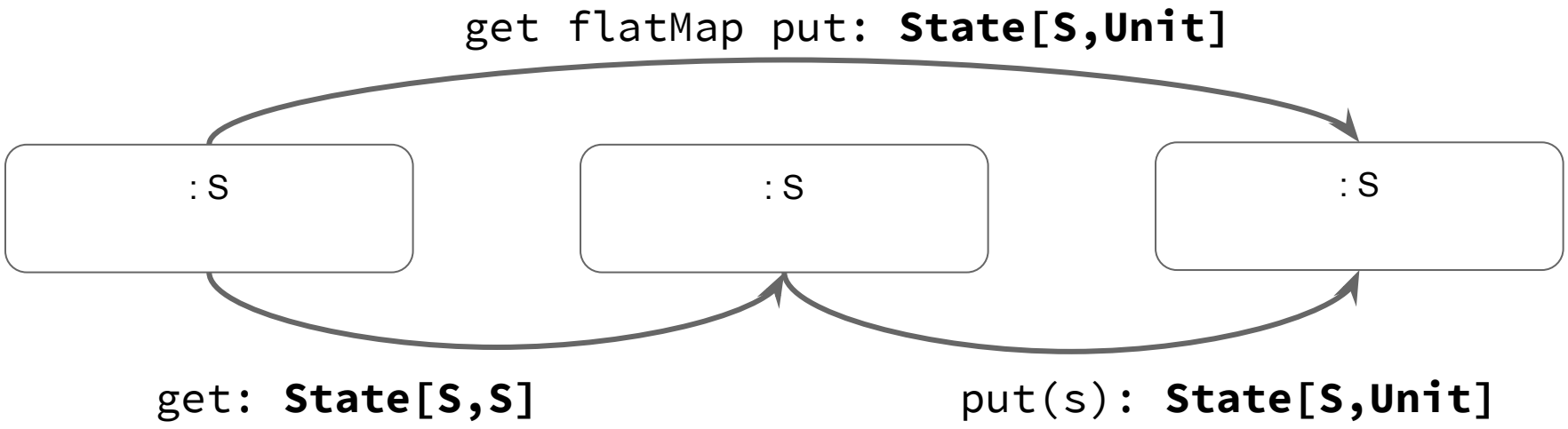
```scala
import scala.util.Try
import scalaz.\/
import scalaz.syntax.either._
import scalaz.syntax.monad._
import scalaz.syntax.std.`try`._

def toInt(s: String): \/[Error, Int] =
  Try(s.toInt)
    .toDisjunction
    .leftMap(_ => s"$s no es un entero válido")

def strToFactorial(s: String): \/[Error, Int] =
  toInt(s) >>= factorial
```

Computaciones con estado
Prueba de programas IO




```
type IOTransformation[T] = IOState => (IOState, T)
case class IOState(reads: List[String], writes: ...)
object PureIO extends IO[IOTransformation] {
  def read                : IOState => (IOState,T) = ???
  def write(msg: String): IOState => (IOState,Unit) = ???
}
```

Computaciones con estado
Mónada State




```
type State[S,T] = S => (S, T)

type IOTransformation[T] = State[IOState,T]
```

Scalaz State:
Computaciones con estado

```scala
trait MonadState[F[_], S] extends Monad[F] { self =>
  // primitive effects
  def get: F[S]
  def put(s: S): F[Unit]

  // derived operations
  def gets[A](f: S => A): F[A] =
    map(get)(f)
  def modify(f: S => S): F[Unit] =
    bind(get)(s => put(f(s)))
}
```

Scalaz State:
Computaciones con estado

```
type IOTrans[A] = State[IOState, A]

implicit object PureIO extends IO[IOTrans] {
  val ms: MonadState[IOTrans,IOState] =
    StateT.stateMonad[IOState]

  def read: IOTrans[String] =
    for {
      s <- ms.get
      _ <- ms.put(s.copy(reads = s.reads.tail))
    } yield s.reads.head
  ...
}
```

Scalaz State:
Computaciones con estado

```
implicit object PureIO extends IO[IOTrans] {
  val ms: MonadState[IOTrans,IOState] =
    StateT.stateMonad[IOState]

  ...
  def write(msg: String): IOTrans[Unit] =
    for {
      s <- ms.get
      _ <- ms.put(s.copy(writes = msg :: s.writes))
    } yield ()
}
```

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - **Ejercicio 3 - Uso de mónadas**
- Sección 6 - Conclusión del curso

Ejercicio 3

Ejercicio 3: Utiliza los operadores de las mónadas para implementar los programas de IO que vimos en el tema de lenguajes

tema4-libs/ejercicio3/Ejercicio3.scala

El ecosistema funcional de Scala

- Sección 0 - El ecosistema funcional de Scala
- Sección 1 - Scalaz intro
- Sección 2 - Patrón de diseño de type classes
  - Ejercicio 1 - Uso de monoides
- Sección 3 - Scalacheck intro
- Sección 4 - Leyes de los monoides con Scalacheck
  - Ejercicio 2 - Probar una instancia de monoide
- Sección 5 - Mónadas en Scalaz
  - Ejercicio 3 - Uso de mónadas
- **Sección 6 - Conclusión del curso**

It’s all about modularity

DATATYPE GENERICS

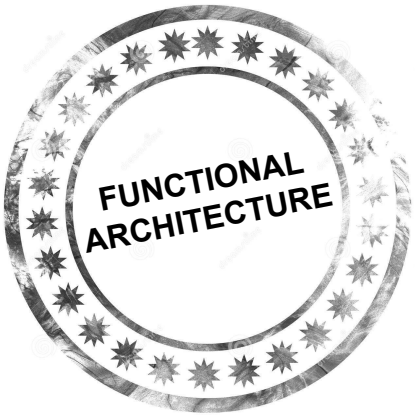
PURE FUNCTIONS

TYPE CLASSES

HIGHER-ORDER FUNCTIONS

PARAMETRIC POLYMORPHISM

FUNCTIONS


FUNCTIONAL
ARCHITECTURE


PURE
FUNCTIONS
programs
LANGUAGES
INTERPRETERS

Type classes For The Win!

```scala
trait FileIOInstructions[P[_]]{
   def readFile(path: String): P[String]
   def writeFile(path: String, contents: String): P[Unit]
   def deleteFile(path: String): P[Unit]
 }
```

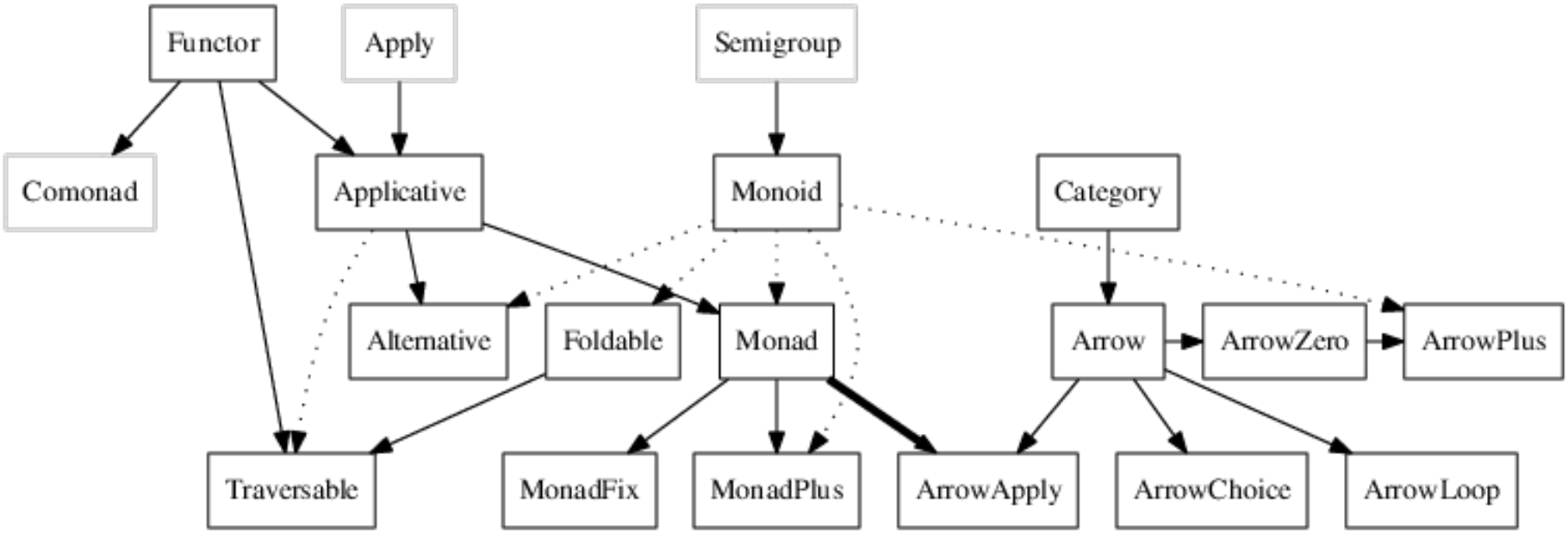
```scala
trait YourLanguage1[P[_]]{
   def ???(): P[???]
   ...
}
```

```scala
trait IOInstructions[P[_]]{
   def read(): P[String]
   def write(msg: String): P[Unit]
 }
```

```scala
trait YourLanguage2[P[_]]{
   def ???(): P[???]
   ...
}
```

```scala
def remove[P[_]](implicit
  IO: IOInstructions[P],
  FS: FileIOInstructions[P],
  M:  Monad[P]): P[Unit] =
  for{
    _    <- IO.write("File to remove?")
    path <- IO.read()
    _    <- FS.deleteFile(path)
  } yield ()
```

Do NOT start from *scratch* !



https://wiki.haskell.org/Typeclassopedia

Conclusiones: ¿Por qué Scala?

- Soporte de técnicas de PF:
  - Genericidad *Higher-kind*
  - Implícitos
  - Lambdas
  - Sintaxis (*for-comprehension*, *context bound*, ...)
- Ecosistema
  - Compatible con Java
  - Spark, Shapeless, Scalaz, Play, Akka, ...