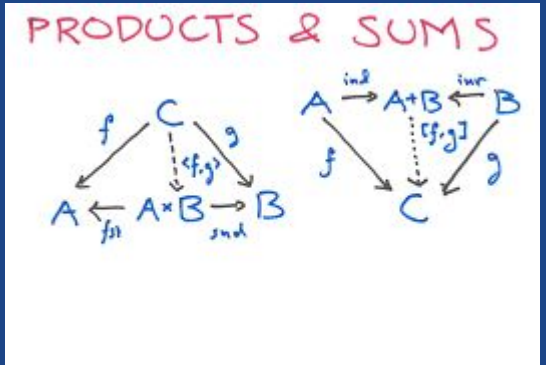
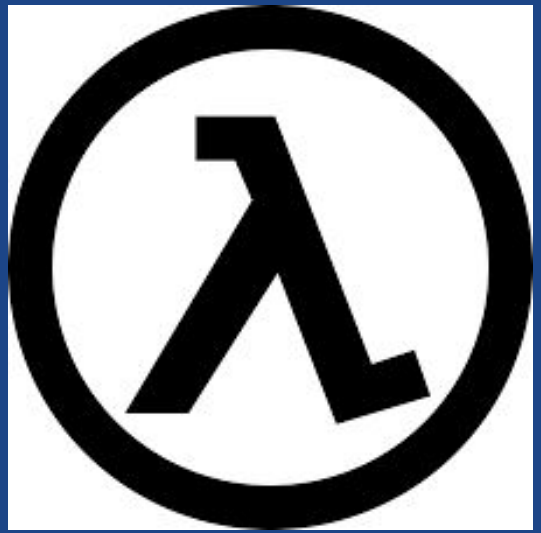
Being Fun parte 1

Monads
Monoids
Applicatives
Functors

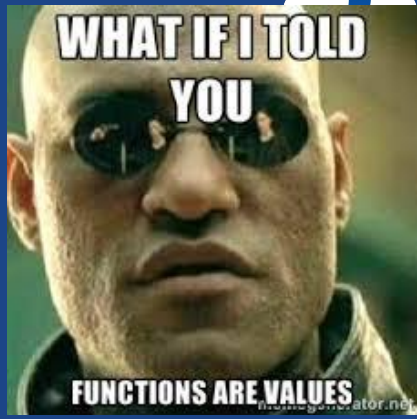


FP =>

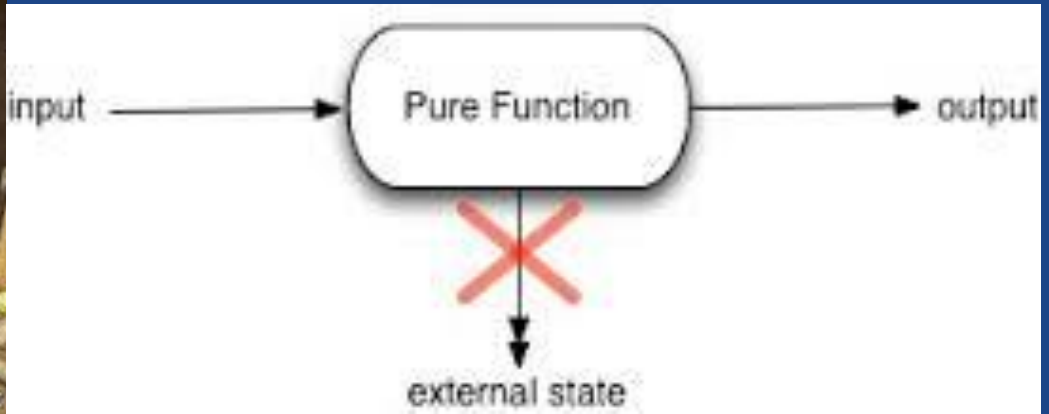


- Pure Functions
- Expressions
- Lazy Evaluation
- Referential Transparency
- Algebraic Data Types
- Higher Functions
- Transform and Composition
- Ability of Reasoning

functions



- Deterministico
- Sin efecto de lado
- Nos lleva a pensar que hacemos con el efecto de lado



Expression driven

- Statements se ejecutan en el momento que llega al ; (lenguaje imperativo c, c++, java, etc)
- Expressions se difiere su evaluación hasta que sea el momento indicado
- Expressions trae la transparencia referencial => se puede usar la expresión como un valor sin necesidad de evaluarse hasta el momento indicado (lazy evaluation y substitution model).
- Substitution Model => pensar como ecuaciones
  - val a = 10 + 5
  - val m = a + a => m = (10 + 5) + (10 + 5)
  - println (m) => println((10 + 5) + (10 + 5)) => println ((15) + (15)) => (println(30)) => efecto de lado 30 ;)

Algebraic Data Types

Ahhh, un case class… en scala

- Estructura de datos (no son Abstract Data Types)
- garantizan inmutabilidad
- Garantizan propiedades algebraicas como la igualdad, simetría y transitividad

Que es un Type?

- Un tipo representa una descripción de un conjunto de valores que cumplen todos con las mismas propiedades
  - más fácil: definición de un conjunto por comprensión (Números enteros)
  - todos los valores tienen un tipo

Qué otros tipos de tipos hay?

- Product Types
  - Producto cartesiano de todos los valores de dos o más tipos (es decir hay que multiplicar para saber cuántos valores pueden representarse con ese tipo)
  - Ejemplo: type Point2D = (Int, Int)
  - case class Hotel(id: Int, rooms: Int)
- Sum Types
  - Conjuntos Disjuntos (ya no es un producto cartesiano)
  - Either[Exception, Int]
  - sealed trait AddressType
    case object Home extends AddressType
    case object Business extends AddressType

https://gist.github.com/jdegoes/97459c0045f373f4eaf126998d8f65dc

Qué otros tipos de tipos hay?

- Type Constructors
  - Es un tipo que sirve para construir tipos o_O
  - Ej. List es un constructor de tipo que pertenece a la categoría de Listas que según el type variable que se le pase List[Int] o List[Boolean].

https://gist.github.com/jdegoes/97459c0045f373f4eaf126998d8f65dc

Qué otros tipos de tipos hay?

- Higher-Kinded Types (preparen la pipa…)
  - Type-Level Functions: Son funciones a nivel de tipos por el que dado un tipo se retorna otro tipo
    - List acepta un tipo Boolean y retorna el tipo List[Boolean]
  - Kinds: El tipo de los tipos (preparen los acidos...)
    - * representa el conjunto de todos los tipos
    - * => * representa un type-level function _ [_]
    - (* => *) => * Higher Kinded Monad[F [_]]
- Existentials (chamuyando un poco más)

  def largo[A](l: List[A]) = l.size (no importa que es A)

  def largo(l: List [_]) = l.size (mejor así)

  ojo con devolver la lista porque se perdió el tipo.

Higher Functions

- Wikipedia (al menos presenta una de estas características
  - receives functions as arguments
  - returns function as result
- cualquier otra función se llaman de primer orden
- Ejemplo clásico, la derivada es una función de orden superior que se aplica a una función de primer orden y como resultado da otra función de primer orden
- Ejemplo conocido: transformaciones o composición de funciones

Transformaciones

- map => transformar de un conjunto a otro
- flatten => transforma el codominio al tipo del dominio
- flatMap => map + flatten

```
val a = List(1,2,3)
val f: ( Int => Option[Int]) = x => if ( x == 0) None; else Some(x);
a.map(f(x)).flatten == a.flatMap(f(_)) …. $> true
```

Acumulando

- fold, foldLeft, foldRight (acumular en algún orden)
  - fold right problema (lazy evaluation => java.lang.StackOverflowError)
  - fold left (while)

```
override def foldLeft[B](z: B)(f: (B, A) => B): B = {
  var acc = z
  var these = this
  while (!these.isEmpty) {
    acc = f(acc, these.head)
    these = these.tail
  }
  acc
}
```

```
override def foldRight[B](z: B)(f: (A, B) => B): B =
this match {
  case Nil => z
  case x :: xs => f(x, xs.foldRight(z)(f))
}
```

g o f (g(f(x))) vs andThen

```
val sumaUno = (x: Int) => x + 1
val multiplicaPorDos = (x: Int) => x * 2
val cuadrado = (x: Int) => x * x
val elements = List(1,2,3,4,5)
```

```
scala> elements map sumaUno map multiplicaPorDos map cuadrado
res1: List[Int] = List(16, 36, 64, 100, 144)
```

```
scala> elements map (sumaUno compose multiplicaPorDos compose cuadrado)
res2: List[Int] = List(3, 9, 19, 33, 51)  => son distintos (recordar g aplica a f(x) )
```

```
scala> elements map (cuadrado compose multiplicaPorDos compose sumaUno )
res3: List[Int] = List(16, 36, 64, 100, 144) => se declaran al reverso o usar andThen
```

```
scala> elements map (sumaUno andThen multiplicaPorDos andThen cuadrado )
res4: List[Int] = List(16, 36, 64, 100, 144)
```

Y en el mundo real...

Side Effects

- Problema mezclar side effects con lógica de negocio:
  - Acoplamiento
  - Jode los Test
  - Oscurece lógica de dominio
  - No se puede componer código
- Objetivo: Aislar los efectos de lado
- Como: Utilizando Effects Wrappers

… effects wrappers como

Una monada:

- se lo ve como una herramienta que permite unir componentes de manera robusta
- encapsula valores de un tipo, creando un nuevo tipo asociativo a un algoritmo específico
- esto permite encadenar operaciones permitiendo armar tuberías de procesamiento de datos
- Tiene que cumplir ciertas propiedades para que pueda garantizar ese comportamiento

En criollo...

- Es un wrapper de un valor de un tipo o encapsula una ejecución posiblemente con efecto de lado que puede asociarse con otras mónadas y descriptivamente encadenar operaciones. (cheto)
- Define 3 cosas:
  - un constructor de tipo llamado unit en la jerga, es ej: el constructor del wrapper Success(x) de Try
    - def unit[A](x: A): M[A]
  - un operador que permite enlazar acciones
    - def flatMap[B](f: A => M[B]): M[B]

https://medium.com/@sinisalouc/demystifying-the-monad-in-scala-cc716bb6f534#.vfzzps8x7

Y en el mundo real...

Monadas al rescate!

- Ejemplo: Uso de Try[T] cuando se requiera hacer algo con efecto de lado.

```
def ioCall (...) : Try[T]
ioCall match {
        case Success(result) =>
        case Failure(exception) =>
    }
```

Al ser una (pseudo) monada, se puede hacer algo así

ioCall ***andThen*** storeInDB *andThen* logResult

Las monadas en Scala*

- Option[T] => Some(x) | None
- Try[T] => Success(x) | Failure(e)
- Future[T] => Success(x) | Failure(e)
- Either[T] => Left(exception) | Right(result)

¿Ven un patrón en comun?

*Odersky, no utilizó el concepto genérico de Monada, y las implementaciones que hizo no cumplen con todas las leyes. Para eso se creó scalaZ y Cats.

https://medium.com/@sinisalouc/demystifying-the-monad-in-scala-cc716bb6f534#.vfzzps8x7

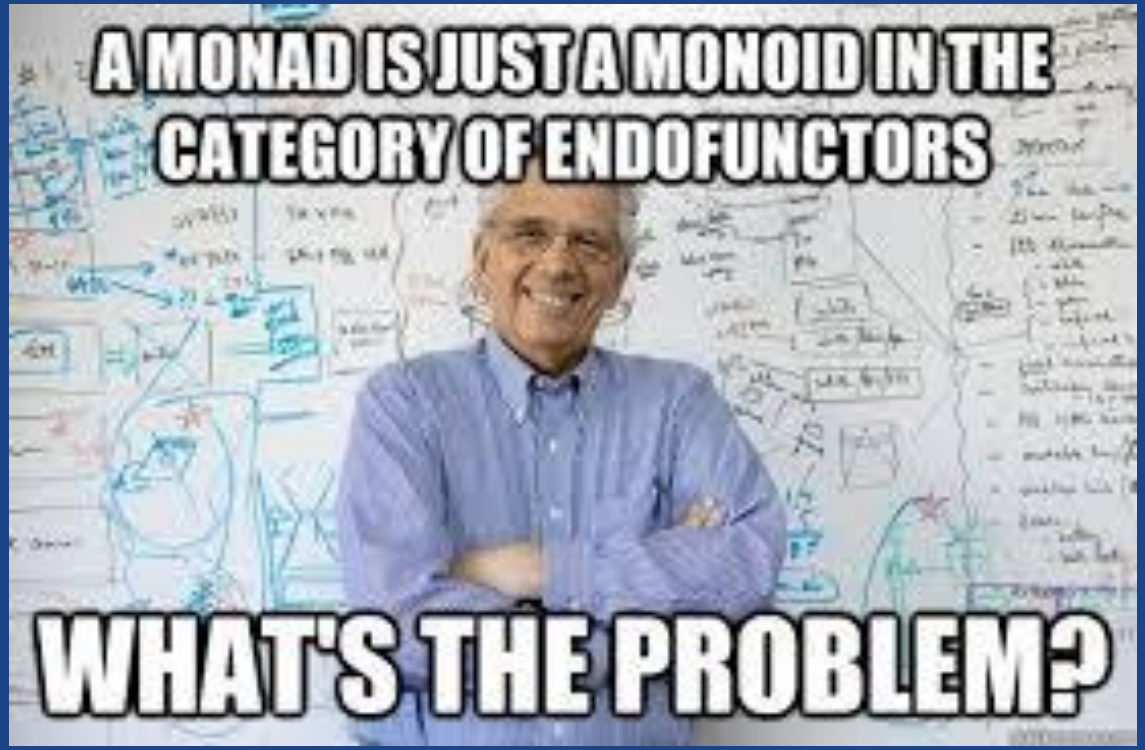
Herramientas en Scala

- allCatch Exceptions (capturando el efecto)

```
import scala.util.control.Exception._
allCatch.opt => Some(result) | None
allCatch.either => Right(result) | Left(exception)
allCatch.toTry => Success(result) | Failure(exception)
```

Y en el mundo algebraico

Una monada es



Questions?