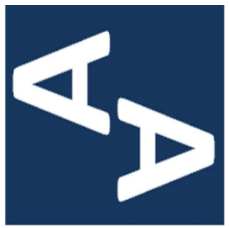
Elimina la corrupción: *programación funcional pura con Scala*



Juan Manuel Serrano
Habla Computing
juanmanuel.serrano@hablapps.com
@jmshac

Objetivos

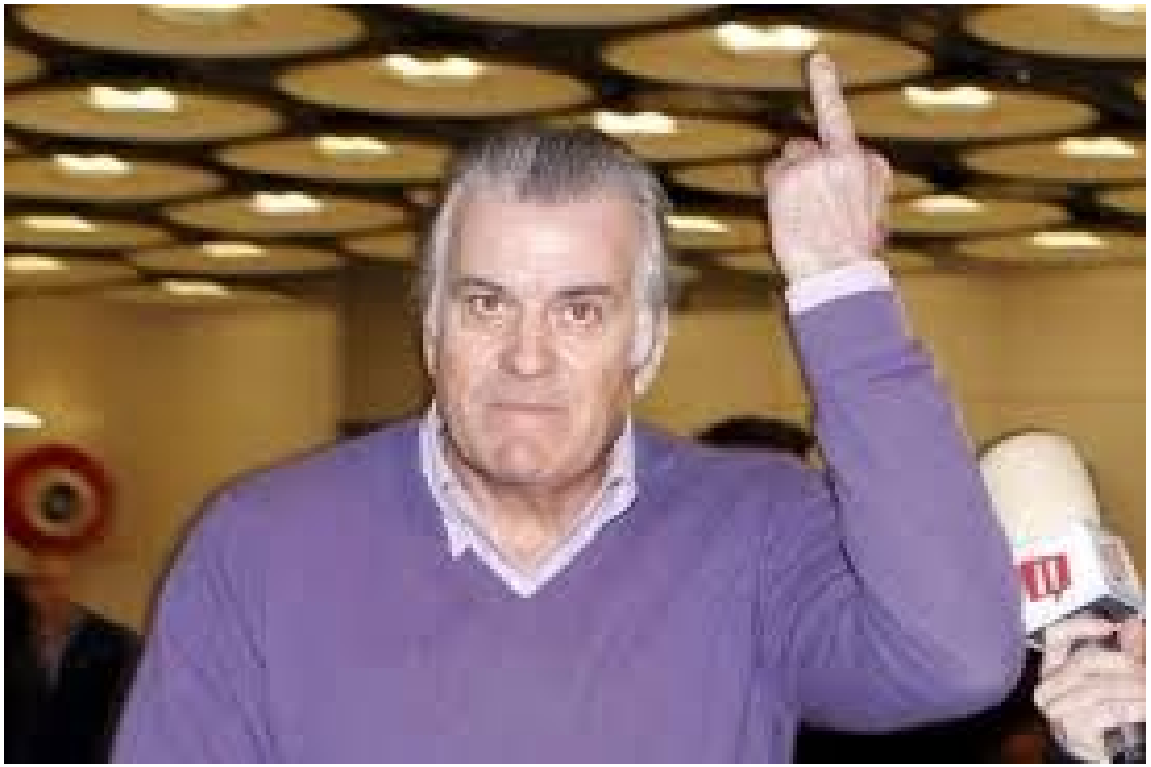
Martin Odersky

Phil Wadler @ OOPSLA07

¿Qué es la programación funcional?

En gran medida, programar con **funciones puras**: funciones que reciben *valores* de entrada, devuelven valores de salida, y **no hacen nada más**

El problema: los efectos colaterales

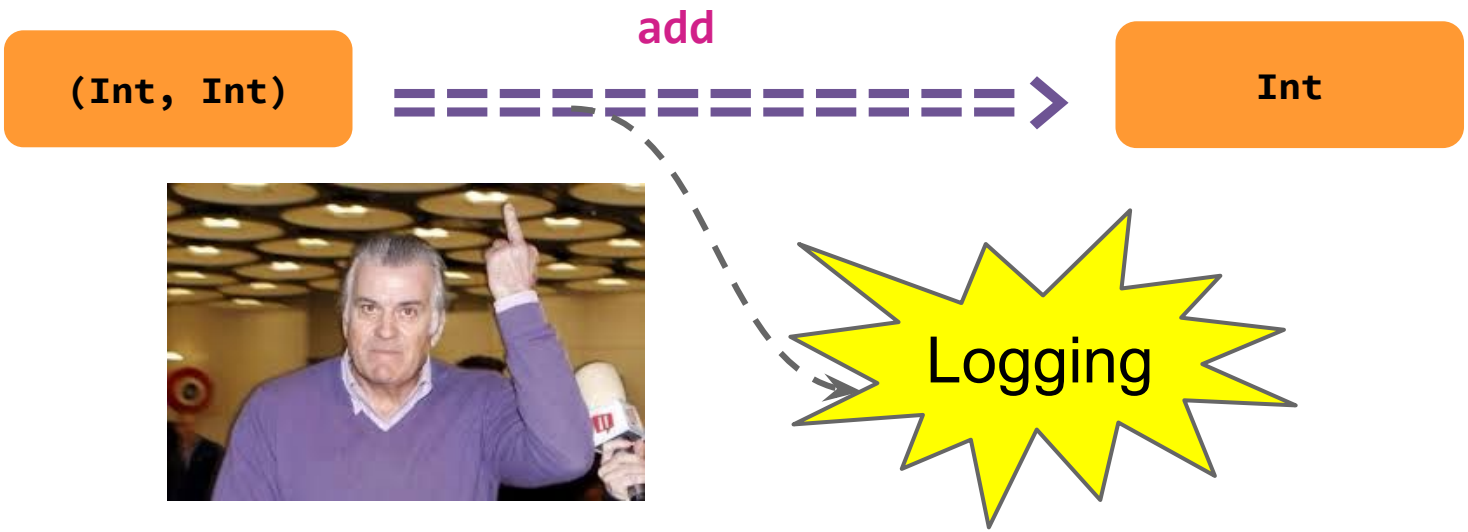


Una función pura: libre de efectos colaterales

```
def add(x: Int, y: Int): Int =
  x + y
```

Una función impura: corrupta (no declara todo lo que hace realmente)

```
def add(x: Int, y: Int): Int = {
  val z = x + y
  println(s"adding $x + $y = $z")
  z
}
```



Los efectos colaterales como encarnación del mal en el software

Si no puedes fácilmente

- Comprender
- Probar
- Arreglar
- Optimizar
- Reutilizar
- Componer
- ...

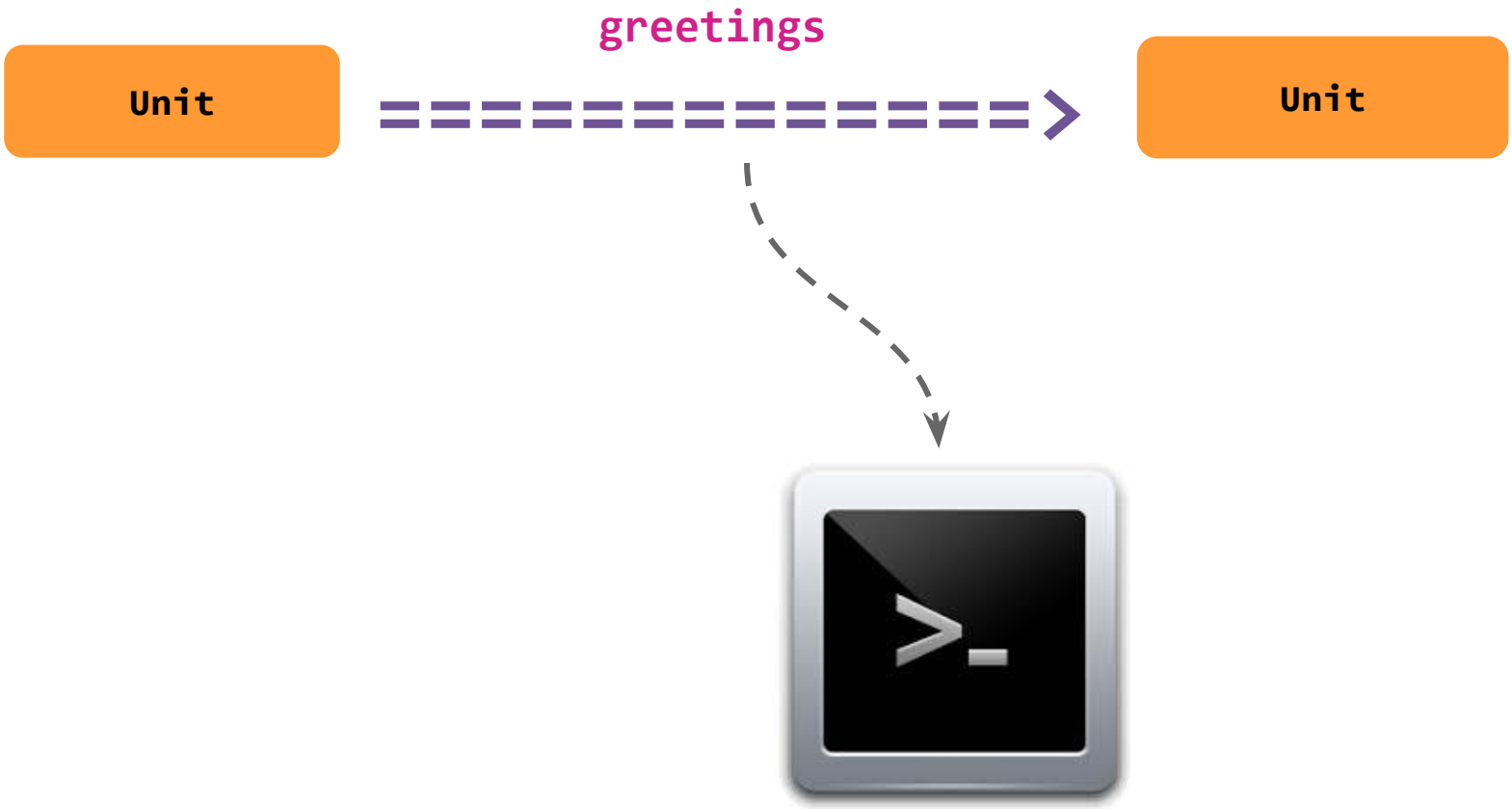
un programa …


SIDE EFFECT-FREE
SEGREGATION
SIDE EFFECTFUL
DECOUPLING

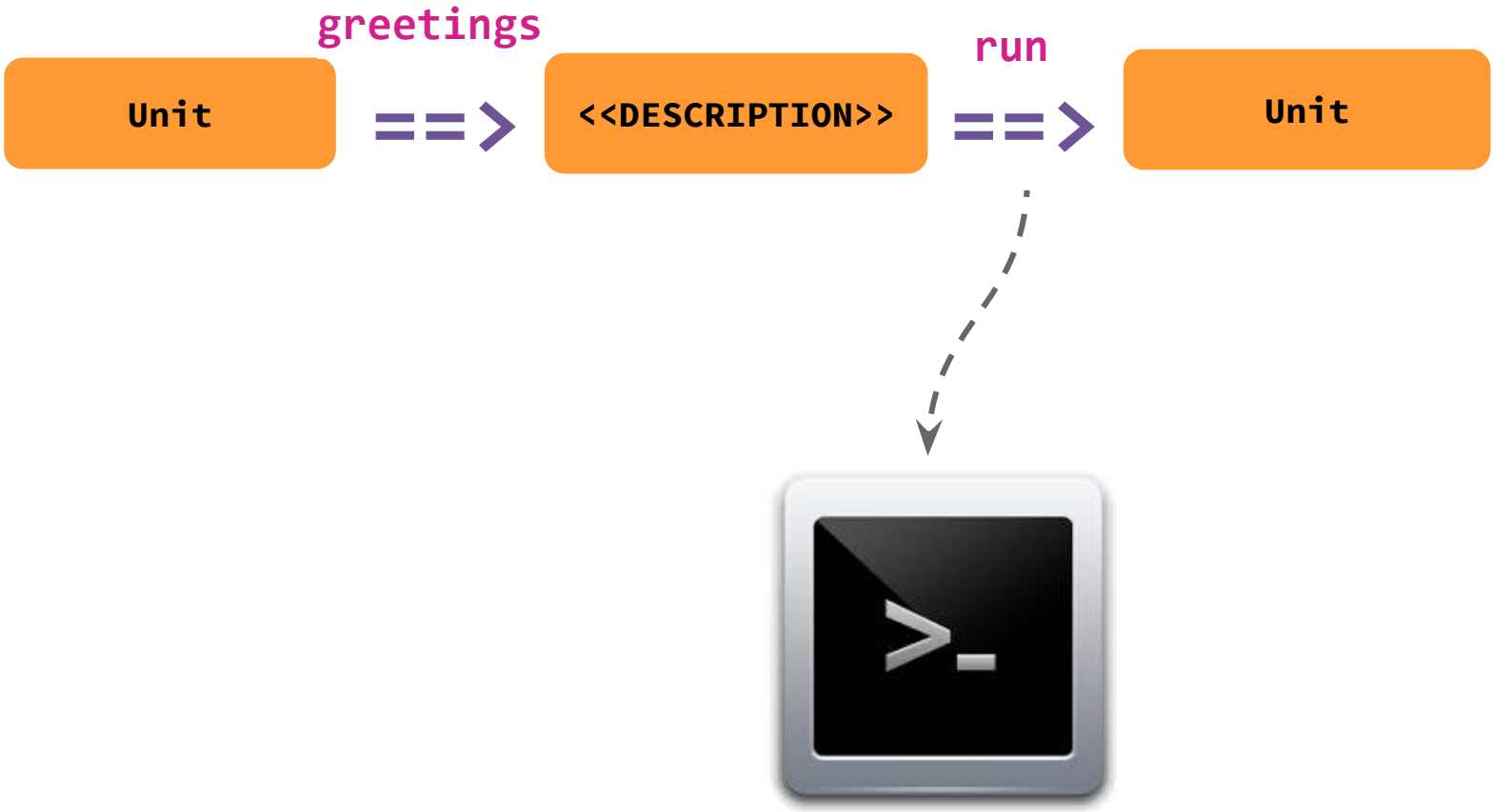
PURIFICACIÓN

- Lenguajes, programas e intérpretes
- Programas parametrizados
- Programas secuenciales
- Programas dependientes del contexto
- Programas “dulcificados”

La programación funcional es programación declarativa



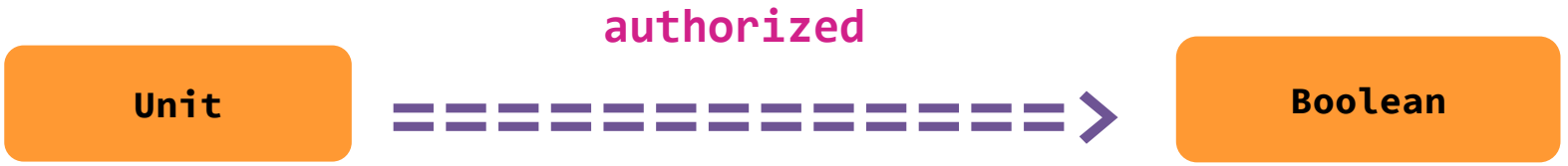
La programación funcional es programación declarativa



Smart constructors & for-comprehensions


```
def authorized: IOProgram[Boolean] = for{
      user <- print("user:")
      pw   <- read
      r    <- print("password:")
} yield r
      user <- read
      pw   <- print("password:")
      r    <- user=="me" && pw=="hola123"
```

Las semejanzas con el programa OO son evidentes ...




```
def authorized: Boolean = {
  val user = print("user:")
  val pw   = read
  val r    = user=="me" && pw=="hola123"
  r
}
```

Evaluación

MasterEFFECT


Unit
==>
Boolean
OO INTERFACES
<<IMPLEM.1>>
<<IMPLEM.2>>
<<IMPLEM.3>>


Unit
==>
IOProgram
[Boolean]
ALGEBRAIC DATA TYPE
interpreter1
interpreter2
interpreter3

¿Quién ofrece mejor desacoplamiento?

Slick 3.0

OO Interfaces *leak* non-functional concerns

Conclusiones

- Tómate en serio el problema de los efectos de lado: tómate en serio la programación funcional
- Las funciones de orden superior (map, filter, fold, etc.), la inmutabilidad, etc., no ayudan en este sentido
- Lo esencial son los lenguajes y los intérpretes: que nuestro lenguaje sea monádico o no es secundario
- Hay grandes similitudes con la programación típica orientada o objetos

¡Pero hay que escribir mucho más!

```scala
sealed trait IOProgram[T]{
  def flatMap[S](cont: T => IOProgram[S]): IOProgram[S] =
    Sequence(this, cont)
  def map[S](f: T => S): IOProgram[S] =
    flatMap( t => Value(f(t)))
}
case class Effect[T](effect: IOEffect[T]) extends IOProgram[T]
case class Sequence[S,T](
  program: IOProgram[S],
  cont: S => IOProgram[T]
) extends IOProgram[T]
case class Value[T](t: T) extends IOProgram[T]
```

¡Pero hay que escribir mucho más!

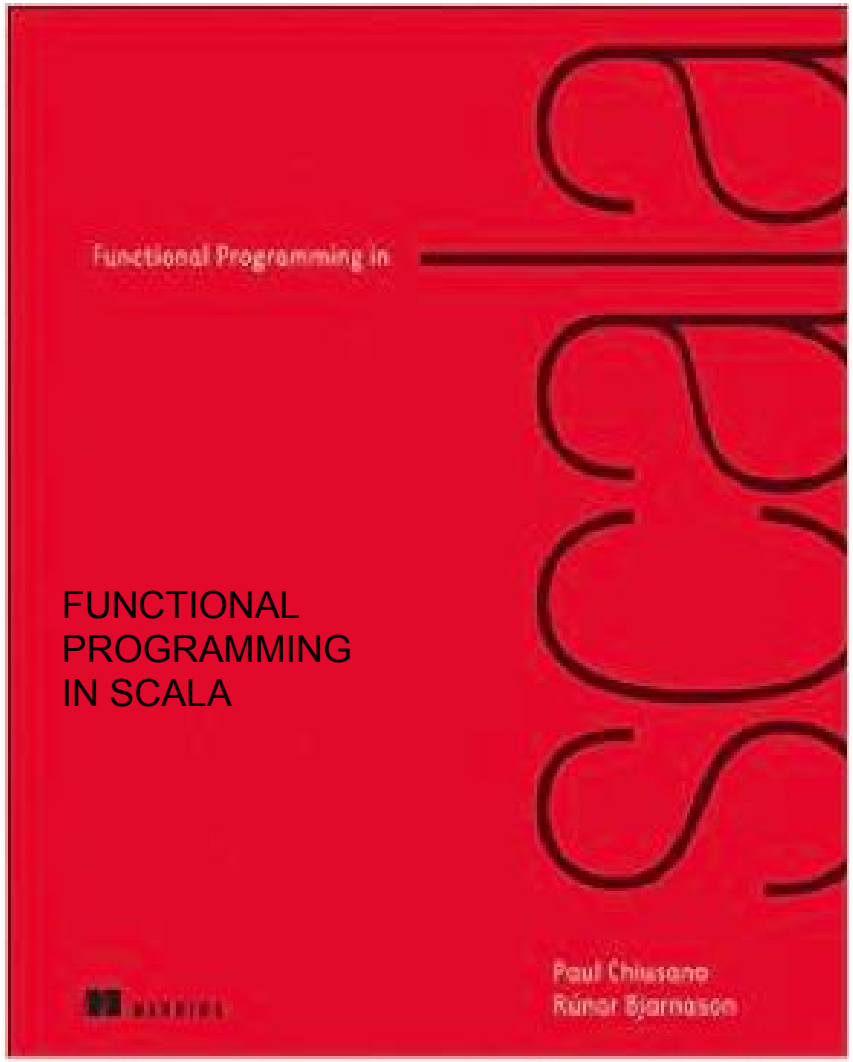
```
object IOProgram{
  def print(msg: String): IOProgram[Unit] =
    Effect(Print(msg))
  def read: IOProgram[String] =
    Effect(Read())
}

// Interpreter
  def run[T](program: IOProgram[T]): T =
    program match {
      case Effect(effect) => runEffect(effect)
      case Sequence(p1,cont) =>
        val r = run(p1)
        run(cont(r))
      case Value(v) => v
    }
}
```

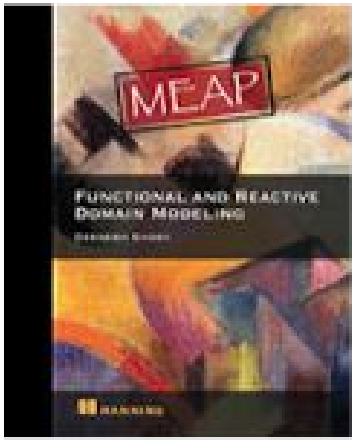
Scalaz/Cats **to the rescue ...**

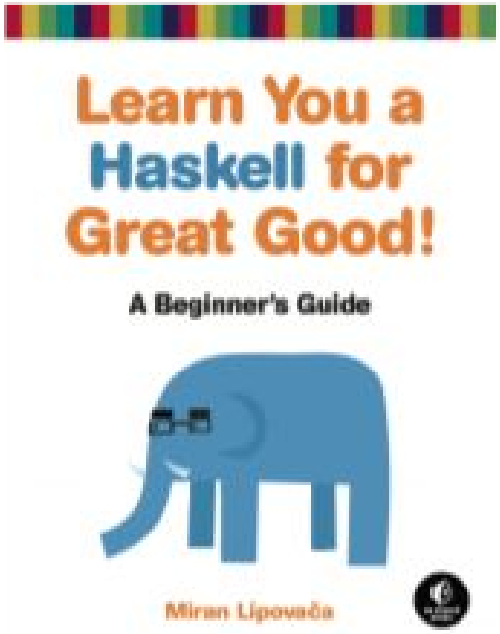
```scala
type IOProgram[A] = Free[IOEffect, A]

def run[U](program: IOProgram[U]): U =
  program.foldMap(runEffect)
```



FUNCTIONAL
PROGRAMMING
IN SCALA





Thanks, and see you soon!

www.meetup.com/Scala-Programming-Madrid

https://github.com/hablapps/funinscala-codemotion-15

juanmanuel.serrano@hablapps.com
@jmshac