Scala Introduction (II-IV)

Functional Programming

Jesús López-González

Senior Software Architect @ Habla Computing

2nd April 2020

@jeslg jesus.lopez@habla.dev

Functional Programming

Objectives

- Grasp the fundamentals of FP by means of Scala
- Learn the basics of algebraic data types (ADTs)
- Understand the implications of having functions as first-class citizens
- Get used to the syntax that simplifies dealing with functions

Functional Programming

So, What is Functional Programming? (1/2)

“Programming With Pure Functions”

*A **pure function** is a function that has the following properties:*

1. *Its return value is the same for the same arguments*
2. *Its evaluation has no side effects*

Functional Programming

So, What is Functional Programming? (2/2)

```scala
def pure(a: Int, b: Int): Int = a + b

var res: Int = 0

def impure(a: Int, b: Int): Int = {
  res = a + b
  a + b
}
```

Functional Programming

Immutability

```scala
val x = 0
x = 1
```

From "Research Topics in Functional Programming" ed. D. Turner, Addison-Wesley, 1990, pp 17–42.

Why Functional Programming Matters

John Hughes
The University, Glasgow

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write and to debug, and provides a collection of modules that can be reused to reduce future programming costs. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute significantly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). We conclude that since modularity is the key to successful programming, functional programming offers important advantages for software development.

1 Introduction

This paper is an attempt to demonstrate to the larger community of (non-

Functional Programming

Session Structure

- Algebraic Data Types (ADTs)
  - Case classes
  - Pattern Matching
- Lambda expressions
- Syntactic Sugar

Functional Programming

Session Structure

- **Algebraic Data Types (ADTs)**
  - Case classes
  - Pattern Matching
- Lambda expressions
- Syntactic Sugar

Functional Programming

Algebraic Data Types

```scala
sealed trait IList {
  def prepend(i: Int): IList = new Cons(i, this)
}

class Cons(val head: Int, val tail: IList) extends IList
class End() extends IList
```

Functional Programming

Case Classes & Pattern Matching

```scala
sealed trait IList {
  def sum: Int = this match {
    case Cons(h, t) => h + t.sum
    case End() => 0
  }
}

case class Cons(head: Int, tail: IList) extends IList
case class End() extends IList
```

Functional Programming

Session Structure

- Algebraic Data Types (ADTs)
  - Case classes
  - Pattern Matching
- **Lambda expressions**
- Syntactic Sugar

Características de PF

Lambda Expressions (1/2)

```scala
val incr: Function1[Int, Int] = new Function1[Int, Int] {
  def apply(i: Int): Int = i + 1
}
```

Características de PF
Lambda Expressions (2/2)

```scala
sealed trait IList {
  def map(f: Function1[Int, Int]): IList = this match {
    case Cons(h, t) => Cons(f(h), t.map(f))
    case End() => End()
  }
}

case class Cons(h: Int, t: IList) extends IList
case class End() extends IList
```

Functional Programming

Session Structure

- Algebraic Data Types (ADTs)
  - Case classes
  - Pattern Matching
- Lambda expressions
- **Syntactic Sugar**

Syntactic Sugar

Operators

```
scala> val lista1 = Cons(1, Cons(2, Nada()))
lista1: Cons = Cons(1,Cons(2,Nada()))

scala> lista1 contiene 2
res0: Boolean = true

scala> lista1 ++ lista1
res1:Lista = Cons(1,Cons(2,Cons(1,Cons(2,Nada()))))

scala> (:: operator as prepend)
```

Syntactic Sugar

Default Parameters for Constructor

```
case class Cons(
  cabeza: Int,
  resto: Lista = Nada()) extends Lista

scala> Cons(1, Cons(2))
res0: org.hablapps.curso.azucar.Cons = Cons(1,Cons(2,Nada()))
```

Syntactic Sugar
Variadic Methods

```scala
object Lista {
  def crear(es: Int*): Lista = {
    if (es.isEmpty)
      Nada()
    else
      Cons(es.head, crear(es.tail: _*))
  }
}

scala> Lista.crear(1,2,3)
res0:Lista = Cons(1,Cons(2,Cons(3,Nada())))
```

Syntactic Sugar

The apply method

```scala
object Lista {
  def apply(es: Int*): Lista = {
    if (es.isEmpty)
      Nada()
    else
      Cons(es.head, apply(es.tail: _*))
  }
}

scala> Lista(1,2,3)
res0: Lista = Cons(1,Cons(2,Cons(3,Nada())))
```

Syntactic Sugar
Lambda Expression

```
(x: Int) => x + 1

(x: Int, y: Int) => "(" + x + ", " + y + ")"
```


(*) Ejemplo extraído de la [sección de Funciones Anónimas](#) del tutorial oficial de Scala

Syntactic Sugar
Placeholder Lambdas

```
scala> val l = List(1, 2, 3)
l: List[Int] = List(1, 2, 3)

scala> l.map(_ + 1)
res0: List[Int] = List(2, 3, 4)
```

section4-azucar/code/Lista.scala

Takeaways

- Functional programming is programming with pure functions
- Algebraic data types are encoded as a "sum" of case clases
- Functions are treated as first-class citizens, which enables *higher order functions*
- Syntactic sugar is convenient to dulcify expressions
- Dotty has introduced many features towards the functional side
- This is just the beginning: *type classes*, *DSLs*, *generic programming*, etc.