Programming in Scala

Lecture Two

Angelo Corsaro, PhD

14 September 2020

Chief Technology Officer, ADLINK Technology Inc.

Table of contents i

Table of contents ii

Table of contents iii

Why Scala

Why Should You Learn Scala?

There are several good reasons for learning scala, the most relevant are in my opinion the following:

- Scala can be seen as a better Java, thus if you target the JVM, it is probably the best programming language you can use.
- Scala gives you a smooth path toward approaching and progressing in functional programming.

Why Should You Learn Scala? – Cont.

- There are an increasing number of important companies adoption scala as one of their mainstream langauges, examples are **LinkedIn**, **Twitter**, **Netflix**, **Apple**, and **AirBnb**.
- There an increasing number of high-impact projects adopting Scala, such as **Apache Spark**, **Apache Flink**, **Akka**, and the **Play! Framework**.
- Scala – and in general functional languages – programmers usually get access to more insteresting jobs and earn more money.

Salaries as November 2018

Java programmer Salary

Based on 856 salaries

Year Month Week Hour

£45,000 / Year

The average **Java Programmer** salary in UK is **£45,000** per year or **£23.08** per hour. Entry level positions start at **£23,400** per year while most experienced workers make up to **£76,500** per year.

Scala programmer Salary

Based on 94 salaries

Year Month Week Hour

£47,100 / Year

The average **Scala Programmer** salary in UK is **£47,100** per year or **£24.15** per hour. Entry level positions start at **£23,400** per year while most experienced workers make up to **£80,070** per year.

Haskell programmer Salary

Based on 49 salaries

Year Month Week Hour

£80,000 / Year

The average **Haskell Programmer** salary in UK is **£80,000** per year or **£41.03** per hour. Entry level positions start at **£35,000** per year while most experienced workers make up to **£136,000** per year.

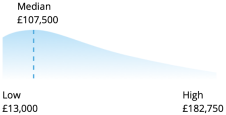
Ocaml programmer Salary

Based on 16 salaries

Year Month Week Hour

£107,500 / Year

The average **Ocaml Programmer** salary in UK is **£107,500** per year or **£55.13** per hour. Entry level positions start at **£13,000** per year while most experienced workers make up to **£182,750** per year.



Type Parameterization

Type Constructors in Scala

Beside first order types, Scala supports Type Constructors, – a kind of higher order types.

In the case of Scala, a parametric class is an n-ary type operator taking as argument one or more types, and returning another type.

Example
The list time we have seen this far, is a type constructor declared as:

$$class \quad List[+T]$$

Thus *List[Int]* and *List[String]* are two instances of the List[+T] type constructor.

Functional Programming Recap

Functional Programming

Functional Programming is a method of program construction that:

- Emphasises functions and their applications as opposed to commands and their execution.
- Uses simple mathematical notation that allows problems to be described clearly and concisely.
- Has a simple mathematical bases that supports equational reasoning about the properties of a program.

A functional programming languages is guided by two main ideas, **functions as first-class values** and **no side-effects**.

Functions as First-Class Values

In a functional programming language:

- A function is a value as an `Integer` or a `String`.
- Functions can be passed as arguments, returned as values as well as stored into variables.
- Functions can be named or anonymous and can be defined inside other functions.

No Side-Effects

- In a functional programming language, the result of applying some arguments to a functions should only depend on the input. In other terms, applying the same input to a given function always gives the same output.
- Functions that satisfy this property are know to be *referentially transparent*.
- Functional programming languages ecnourage the use of *immutable* data structures and *referentually transparent* functions.

Note: it is worth comparing the functional approach with the imperative style programming were everything is based on mutable data and side-effects.

Haskell Quick Start

The Haskell Programming Language

Haskell is a pure, lazy, functional programming language first defined in 1990.

The programming language was named after Haskell B. Curry, who was one of the pioneers of λ-calculus, a mathematical theory of functions that has been an inspiration to designers of a number of functional programming languages.

The latest version of the language is Haskell-2010 and a working group was established in 2016 to define Haskell-2020.

The Haskell's home is `http://www.haskell.org`

Functions in Haskell

Functions in Haskell

Haskell's notation to denotate a function f that takes an argument of type X and returns a result of type Y is:

```haskell
f :: X -> Y
```

Example

For example:

```haskell
sin :: Float -> Float
add :: Integer -> Integer -> Integer
reverse  :: String -> String
sum :: [Integer] -> Integer
```

Notice how Haskell's functions declaration is extremely close to that used in mathematics. This is not accidental, and we will see the analogy goes quite far.

Getting Started with Haskell

The best way to get started with Haskell is to unstall `stack` from `https://docs.haskellstack.org/en/stable/README/`.

Once installed you can start Haskell's **REPL** and experiment a bit:

```
> stack repl
Prelude> let xs = [1..10]
Prelude> :t xs
xs :: (Num t, Enum t) => [t]
Prelude> :t sum
sum :: (Num a, Foldable t) => t a -> a
Prelude> sum xs
55
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> foldl (+) 0 xs
55
Prelude> foldl (*) 1 xs
3628800
```

Functions in Scala

Functions in Scala

Scala's notation to denote a function f that takes an argument of type X and returns a result of type Y is:

```
def f(x: X): Y
```

Example

For example:

```
def sin(x: Float): Float
def add(a: Integer, b: Integer): Integer
def reverse(s: String): String
def sum(xs: Array[Integer]): Integer
```

Notice how Haskell's functions declaration is extremely close to that used in mathematics. This is not accidental, and we will see the analogy goes quite far.

Functions are Objects

In Scala function a function:

$$def \quad f(t_1 : T_1, t_2 : T_2, ..., t_n : T_n) : R$$

is are represented at runtime as objects whose type is:

$$trait \quad Fun[-T_1, -T_2, ..., -T_n, +R] \quad extends \quad AnyRef$$

Local Functions

Some times, it is useful to define *named* functions that don't pollute the global namespace but instead are available only within a given function scope.

Scala makes this possible by defining local functions, in other terms, functions that are defined inside another function.

```scala
def outer(x: X): Y = {
  def inner(a: A): B = {
    // function body
  }
  // ...
  val b = inner(a)
  // ...
}
```

Tail Recursion

Factorial Definition

The mathematical definition of factorial is as follows:

$$factorial(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n * factoriak(n-1) & \text{otherwise} \end{cases}$$

Factorial Definition in Scala – Take 1

If you'd tried to write the factorial, you probably ended up writing something like this:

```scala
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n-1)
```

Please notice that for clarity sake I am not asserting $n >= 0$. This should be checked for production code.

Factorial Definition in Scala – Take 2

If you'd tried to write the factorial, you probably ended up writing something like this:

```scala
def factorial(n: Int): Int = n match {
        case 0 => 1
        case _ if n > 0 => n*factorial(n-1)
}
```

This version looks a bit more like the mathematical definition, which is nice.

Functions in Haskell

Factorial Definition in Haskell – Take 1

If you'd tried to write the factorial, you probably ended up writing something like this:

```haskell
factorial :: Integer -> Integer
factorial 0  = 1
factorial n = n * (factorial n-1)
```

This version looks a bit more like the mathematical definition, which is nice.

Factorial Definition in Haskell – Take 2

If you'd tried to write the factorial, you probably ended up writing something like this:

```
factorial :: Integer -> Integer
factorial n  = case n of
        0 -> 1
        _ -> n* (factorial (n-1))
```

This version looks a bit more like the mathematical definition, which is nice.

Evaluating the Factorial

Below is the equational substitution for `factorial 3` as well as its evaluation stack.

$$\textit{factorial} \quad 3 = 3*(\textit{factorial} \quad 2) = 3*2*(\textit{factorial} \quad 1) = 3*2*1*(\textit{factorial} \quad 0) = 3*2*1*1 = 6$$

...
1
1*(factorial 0)
2*(factorial 1)
3*(factorial 2)
factorial 3
...

Tail Recursion

The implementations of the factorial we have seen thus far take a linear number of stack frames to evaluate.

This is not desirable. A recursive functions that does not evaluate with a bound the stack size may fail at run-time because of stack overflows.

For efficiency and run-time robustness, whenever possible, is is best to write recursive functions so that they are tail recursive.

Definition
Tail-recursive functions are functions in which all recursive calls are tail calls and hence do not build up any deferred operations.

Exercise: Tail Recursive Factorial

Let's find a tail recursive definition of the factorial. Let's see how to get there and how inner function can help out.

Anybody wants to show his/her solution?

Tail Recursive Factorial

```scala
def factorial(n: Int): Int = {
  @tailrec
  def afact(a: Int, n: Int): Int =
    if (n == 0) a else afact(a * n, n - 1)

  afact(1, n)
}
```

As you can see from the fragment above we are carrying state along calls through an accumulator. This is a technique used often to transform a function into tail recursive.

The `@tailrec` annotation is used to tell the Scala compiler that this call is supposed to be *tail-recursive*. If this is not the case, the compiler will raise a warning.

First Class Functions

First-Class Functions

Scala has *first-class functions*. Beside being able to define named functions, it is possible to write functions as unnamed literals and to pass them as values.

Example

```scala
val iadd = (a: Int, b: Int) => a + b
val isub = (a: Int, b: Int) => a - b

val i = iadd(1,2)

val ibinOp = (op: (Int, Int) => Int, a: Int, b:Int) => op(a,b)

ibinOp(iadd, 1, 2)
ibinOp(isub, 1, 2)
```

Currying

Do we need multiple-arguments functions?

Those with an imperative programming background are lend to think that a function in general can have n-arguments.

Thus they think of a generic function as: $def \quad fun(a : A, b : B, c : C, ...) : \quad X$

But is this really needed? Is this the right abstraction?

Currying

Definition
Let $f : X \to Y$ denote **a function** f from X to Y. The notation $X \to Y$ then denotes **all functions** from X to Y. Here X and Y may be sets, types or other kind of mathematical objects.

Given a function $f : (X \times Y) \to Z$, where $X \times Y$ denotes the Cartesian products of X and Y, currying constructs, or makes a new function, $curr(f) : X \to (Y \to Z)$.

That is, $curry(f)$ takes an argument of type X and returns a function of type $Y \to Z$. **uncurrying** is the opposite transformation.

Looking again at Haskell's function declaration

Let's look again at the add function defined earlier:

```
add :: Integer -> Integer -> Integer
```

This function can be re-written as follows, to make more explicit the currying:

```
add :: Integer -> (Integer -> Integer)
```

In other terms, Haskell functions are single-parameters functions.

Technically, the add function above takes an `Integer` parameter and returns a function that `Integer` and returns and `Integer`. The function add can be applied as follows:

```
Prelude> add 1 2
3
Prelude> (add 1) 2
3
```

Looking again at Haskell's function declaration – cont.

Also notice that the function:

```haskell
add :: (Integer, Integer) -> Integer
```

Is a single parameter functions that takes a **tuple** of two `Integer` and returns an `Integer`.

The function `add` can be applied as follows:

```
Prelude> add (1, 2)
3
```

Haskell's function declarations – again

In general a Haskell function has the following form:

$f :: A \rightarrow B \rightarrow C \rightarrow ... \rightarrow Z$

When seeing this declaration, you should think as if it was parenthesized as follows:

$f :: A \rightarrow (B \rightarrow (C \rightarrow ...(Y \rightarrow Z)$

Also notice that:

$f :: (A \rightarrow B) - > C$

Is a function that takes single parameter of type $A \rightarrow B$ (a function from A in B) and returns a C

Currying in Scala

Scala provides support for curryed functions, but these have to be explicitly declared as such.

In general, given a function of **n** arguments:

```
def fun(a: A, b: B, c: C, ...) : X
```

The curryed function is delcared as follows:

```
def fun(a: A)(b: B)(c: C) ... : X
```

Thus, in Scala, differently from Haskell, you have to decide at declaration time if a function is curryed or not.

The syntax is a bit cluttered when compared to Haskell, but the semantics is the same.

Be Patient...

Now you may starting thinking that functional programmers are eccentric, or even a bit insane... But be patient and in a few slides you'll find out the power and elegance of curryed functions.

Partial Application

Definition
Given a function:

$f :: T_1 \to T_2 \to T_3 \to ... T_n \to T$

If we apply this function to arguments:

$e_1 :: T_1, e_2 :: T_2, ..., e_k :: T_k, (k < n)$

the result type is given by *canceling* the k types T_1 to T_k, which give the type:

$g :: T_k + 1 \to T_k + 2... \to T_n \to T$

The resulting function g is obtained by partially applying $k < n$ arguments to f.

Partial Application in Haskell

Partial application in Haskell extremely straightforward, you just have to provide ($k < n$) parameters to the function application.

Example

```haskell
iadd :: Integer -> Integer -> Integer
iadd a b = a + b
iadd 1 2
3
iinc = iadd 1
:t iinc
iinc :: Num a => a -> a
iinc 10
11
```

Partial Application in Haskell

Example

```haskell
ibinOp :: (Integer -> Integer -> Integer) -> Integer -> Integer -> Integer
ibinOp op a b = op a b

isum = ibinOp iadd
imul = ibinOp (*)
inc = ibinOp (+) 1
double = imul 2
```

Partial Application in Scala

Partial application in Scala quite similar to Haskell, with the difference that you have to add a placeholder _ to indicate the fact that other parameters are missing.

Thus given a curryed function:

$$def \quad fun(t_1 : T_1)(t_2 : T_2)(t_3 : T_3)...(t_n : T_n) : T$$

The partial application of the first $k < n$ parameters is done as follows: $fun(t_1)(t_2)(t_3)...(t_k)_T$

This evaluates to a function with the following signature: $def \quad fun(t_k + 1 : T_k + 1)(t_k + 2 : T_k + 2)...(t_n : T_n) : T$

Partial Application in Scala: Example

Example

```
def cadd(a: Int)(b:Int): Int = a + b
cadd(1)(2)

def csub(a: Int)(b:Int): Int = a - b
val cinc = cadd(1)_
cinc (10)

val ibinOp          = (op: (Int, Int) => Int, a: Int, b:Int) => op(a,b)

ibinOp(cadd, 1, 2)
ibinOp(isub, 1, 2)


def cbinOp(op: (Int, Int) => Int)(a: Int)(b:Int) = op(a,b)

val inc = cbinOp (iadd) (1) _

inc(1)
```

Reflections on Partial Applications and Currying

Currying is instrumental for **Partial Application**, but it also has other uses in Scala to introduce high level abstractions, such as new control flow that seem as if they were built-in the language.

Partial application is extremely useful in library design and in higher-order programming. You can use partially-applied higher-order functions to easily customize behaviour of your code and of libraries.

looping in Scala

Let's assume that we wanted to add a **loop** construct to scala.

Recall that Scala's `for` construct should not be used for looping since as we will see, it translates to `map` and `flatmap` and can be quite expensive as an iterative construct.

Let's use what we've learned thus far to implement a looping construct.

We would wont the looping construct to look like this:

```
loop (3) { println("Looping...") }
```

This should produce:

```
Looping...
Looping...
Looping...
```

Looping in Scala

The `loop` function should be defined as the following curried function:

```scala
@tailrec
def loop(n: Int)(body: => Unit): Unit = {
  if (n > 0) {
        body
        loop(n-1)(body)
  }
}
```

Lambda and Closures

Lambdas

Definition
A **lambda function** is an anonymous function, in other terms a function definition that is not bound to an identifier.

Example

```
val timesTwo = (a: Int) => 2*a
```

Closure

Definition
A **closure** is an anonymous function, in other terms a function definition that is not bound to an identifier, which captures free variables from the environment.

Example

```scala
val x = 10
val plusX = (a: Int) => a + x
```

Notice that a as a closure resolves free variables from the environment, is a good state to carry around context.

Lists

Working with Lists

List are one of the canonical functional data structures.

Let's explore Scala's List (see `https://www.scala-lang.org/api/current/scala/collection/immutable/List.html`) and let's see how higher order programming makes list processing extremely elegant.

Exercise: Computing on Lists

Given a list of `List[Int]`, write functions that compute:

- the length
- the sum of its elements
- the product of its elements
- the inverted list
- the maximum
- the minimum

The only list operations you can use are `isEmpty`, `head`, `tail`, and either append or the operator `:::` that concatenates two lists. See `https://www.scala-lang.org/api/current/scala/collection/immutable/List.html`.

Computing on Lists with Pattern Matching

```scala
val xs = List(1,2,3,4)

def sum(xs: List[Int]): Int = xs match {
  case y::ys => y+sum(ys)
  case List() => 0
}

def mul(xs: List[Int]): Int = xs match {
  case y::ys => y*mul(ys)
  case List() => 1
}

def reverse(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case y::ys => reverse(ys) ::: List(y)
}
```

Folding

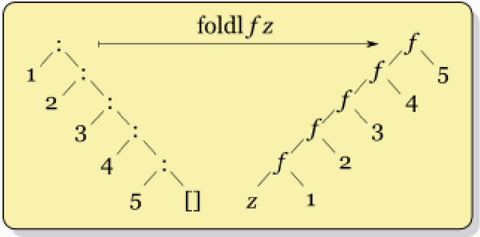
Folding

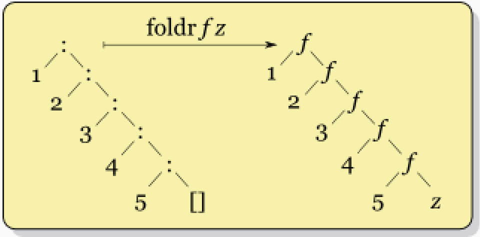
In functional programming, **fold**, refers to a family of *higher-order functions* that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value.

Folds can be regarded as consistently replacing the structural components of a data structure with functions and values.

Two variants of **fold** exist, namely **foldl** and **foldr**.

Visualizing folds





Folding Lists

The functions we just wrote for lists can all be expressed in terms of fold operator.

```
def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)

def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)

def reversel(ys: List[Int])= ys.foldLeft(List[Int]())((xs: List[Int], x: Int) => x :: xs)

def reverser(ys: List[Int]) = zs.foldRight(List[Int]())((x: Int, xs: List[Int]) => xs ::: List(x) )
```

foldr and foldl

```scala
def reverse(xs: List[Int]): List[Int] = xs match {
        case List() => List()
        case y::ys => reverse(ys) ::: List(y)
}
val xs = List(1,2,3,4,5)
val ys = reverse(xs)

val zs = ys.foldLeft(List[Int]())((xs: List[Int], x: Int) => x :: xs)

val ks = zs.foldRight(List[Int]())((x: Int, xs: List[Int]) => xs ::: List(x) )
```

Homeworks

Reading

From the **Programming in Scala** book you should read:

- Chapter 8
- Chapter 9
- Chapter 16