

Programming in Scala

Lecture Two

Angelo Corsaro

December 5, 2017

Chief Technology Officer, ADLINK Technology Inc.

Table of contents

1. Functional Programming Recap
2. Haskell Quick Start
3. Functions in Haskell
4. Functions in Scala
5. Exploring Inner Functions and Tail Recursion
6. Functions in Haskell
7. First Class Functions

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 denote a function f that takes an argument of type X and returns a result of type Y is:

```
1 f :: X -> Y
```

Example

For example:

```
1 sin :: Float -> Float
2 add :: Integer -> Integer -> Integer
3 reverse :: String -> String
4 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 uninstall stack from <https://docs.haskellstack.org/en/stable/README/>.

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

```
1  > stack repl
2  Prelude> let xs = [1..10]
3  Prelude> :t xs
4  xs :: (Num t, Enum t) => [t]
5  Prelude> :t sum
6  sum :: (Num a, Foldable t) => t a -> a
7  Prelude> sum xs
8  55
9  Prelude> :t foldl
10 foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
11 Prelude> foldl (+) 0 xs
12 55
13 Prelude> foldl (*) 1 xs
14 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:

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

Example

For example:

```
1 def sin(x: Float): Float
2 def add(a: Integer, b: Integer): Integer
3 def reverse(s: String): String
4 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.

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.

```
1  def outer(x: X): Y = {  
2    def inner(a: A): B = {  
3      // function body  
4    }  
5    // ...  
6    val b = inner(a)  
7    // ...  
8  }
```

Exploring Inner Functions and Tail Recursion

Factorial Exercise

Last lecture I had asked you to look into finding 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?

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:

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

Please notice that for clarity sake I am not asserting $n \geq 0$. 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:

```
1 def factorial(n: Int): Int = n match {  
2   case 0 => 1  
3   case _ if n > 0 => n*factorial(n-1)  
4 }
```

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:

```
1 factorial :: Integer -> Integer
2 factorial 0 = 1
3 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:

```
1 factorial :: Integer -> Integer
2 factorial n = case n of
3     0 -> 1
4     _ -> 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.

$$\begin{aligned} \text{factorial } 3 &= 3 * (\text{factorial } 2) = 3 * 2 * (\text{factorial } 1) = \\ &3 * 2 * 1 * (\text{factorial } 0) = 3 * 2 * 1 * 1 = 6 \end{aligned}$$

...
6
3*2*1*(factorial 0)
3*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 as recursive functions for which the stack is not bound may fail at run-time because of stack overflows.

For efficiency and run-time robustness, whenever possible, it 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.

Tail Recursive Factorial

```
1 def factorial(n: Int): Int = {  
2   @tailrec  
3   def afact(a: Int, n: Int): Int =  
4     if (n == 0) a else afact(a * n, n - 1)  
5  
6   afact(1, n)  
7 }
```

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

```
1  val iadd = (a: Int, b: Int) => a + b
2  val isub = (a: Int, b: Int) => a - b
3
4  val i = iadd(1,2)
5
6  val ibinOp = (op: (Int, Int) => Int, a: Int, b: Int) => op(a,b)
7
8  ibinOp(iadd, 1, 2)
9  ibinOp(isub, 1, 2)
```

Currying

Do we need multiple-arguments functions?

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

Thus they think of a generic function as: $\text{def fun}(a : A, b : B, c : C, \dots) : X$

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

Definition

Let $f : X \rightarrow Y$ denote **a function** f from X to Y . The notation $X \rightarrow Y$ 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 $f : (X \times Y) \rightarrow Z$, where $X \times Y$ denotes the Cartesian products of X and Y , currying constructs, or makes a new function, $\text{curr}(f) : X \rightarrow (Y \rightarrow Z)$.

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

Looking again at Haskell's function declaration

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

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

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

```
1 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 an Integer. The function add can be applied as follows:

```
1 Prelude> add 1 2
2 3
3 Prelude> (add 1) 2
4 3
```

Looking again at Haskell's function declaration – cont.

Also notice that the function:

```
1 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:

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

Haskell's function declarations – again

In general a Haskell function has the following form:

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

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

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

Also notice that:

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

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 curried functions, but these have to be explicitly declared as such.

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

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

The curried function is declared as follows:

```
1 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 curried 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:

$$f :: T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \dots T_n \rightarrow T$$

If we apply this function to arguments:

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

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

$$g :: T_{k+1} \rightarrow T_{k+2} \dots \rightarrow T_n \rightarrow T$$

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

```
1 iadd :: Integer -> Integer -> Integer
2 iadd a b = a + b
3 iadd 1 2
4 3
5 iinc = iadd 1
6 :t iinc
7 iinc :: Num a => a -> a
8 iinc 10
9 11
```

Partial Application in Haskell

Example

```
1  ibinOp :: (Integer -> Integer -> Integer) -> Integer -> Integer -> Integer
2  ibinOp op a b = op a b
3
4  isum = ibinOp iadd
5  imul = ibinOp (*)
6  inc = ibinOp (+) 1
7  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 curried function: $deffun(t_1 : T_1)(t_2 : T_2)(t_3 : T_3) \dots (t_n : T_n) : T$

The partial application of the first $k < n$ parameters is done as follows:

$fun(t_1)(t_2)(t_3) \dots (t_k)_T$

This evaluates to a function with the following signature:

$defafun(t_k + 1 : T_k + 1)(t_k + 2 : T_k + 2) \dots (t_n : T_n) : T$

Partial Application in Scala: Example

Example

```
1  def cadd(a: Int)(b: Int): Int = a + b
2  cadd(1)(2)
3
4  def csub(a: Int)(b: Int): Int = a - b
5  val cinc = cadd(1)_
6  cinc (10)
7
8  val ibinOp          = (op: (Int, Int) => Int, a: Int, b: Int) => op(a,b)
9
10 ibinOp(cadd, 1, 2)
11 ibinOp(isub, 1, 2)
12
13
14 def cbinOp(op: (Int, Int) => Int)(a: Int)(b: Int) = op(a,b)
15
16 val inc = cbinOp (iadd) (1) _
17
18 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 want the looping construct to look like this:

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

This should produce:

```
1 Looping...
2 Looping...
3 Looping...
```

looping in Scala

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

```
1 @tailrec
2 def loop(n: Int)(body: => Unit): Unit = {
3     if (n > 0) {
4         body
5         loop(n-1)(body)
6     }
7 }
```

Homeworks

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

- Chapter 8
- Chapter 9