Java Talk, 25.11.2014

A short Introduction into Functional Programming with Scala

Peter Salhofer, FH JOANNEUM

Functional Programming?

- In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the **evaluation of** mathematical **functions** and **avoids changing-state and mutable data**. [Wikipedia]

A Pure Function:

- The function always evaluates the same result value given the same argument value(s). The function result value **cannot depend on any hidden information or state** that may change as program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices.
- Evaluation of the result **does not cause** any semantically observable **side effect** or output, such as mutation of mutable objects or output to I/O devices
  [Wikipedia]

Functional Programming in Practice ...

- Avoid variables
- Everything is a function
- Immutable data structures over mutable ones
- Recursion over Iteration
- Functional constructs for collection types

Scala

- Scala is a language that favours functional programming in combination with OOP
- It runs in the JVM and can therefore be used with any Java class

Core Features

- Great Type Inference
- Concurrency & Distribution (native support for Promises and Futures)
- Traits (≈Interfaces with concrete methods)
- Pattern Matching (≈mature switch-case)
- Higher Order Functions (incl. Currying)
- Implicits (≈Dependency Injection for Arguments)

Type Inference

We shall not use variables!

```
var text:String = "Some Text"
```

Type follows the name

We prefer values (constants)

```
val text = "Some Text"
```

The Type is inferred from the vaule!

The Limits Of Type Inference

- Parameters of a function declaration need have types!
- The return type of recursive functions needs to be declared!

Function Declarations

```
def add(a:Int,b:Int):Int = {
  return a+b
}
```

The Java-to-Scala approach

```
def add(a:Int,b:Int) = a+b
```

Idiomatic Scala:

- Return Type of function is inferred
- Return keyword is never used, since the last expression is returned automatically
- One-liners do not need curly braces

Functions vs. Procedures

„=“ → Function

```
def add(a:Int,b:Int) = a+b
```

„{...}“ without „=“ → Procedure → Return type = Unit

```
def printSum(a:Int,b:Int) {print(a+b)}
```

Standard Lists are Immutable!

Immutable List: Cannot grow or be changed in any way!

```
val list = List("U","B","I")

println(list(1))  //"B"
```

Elements are accessed via parenthesis

```
list.foreach(println)

//or

list foreach println
```

Syntactic Sugar: Accessing methods does not require the „.“-operator. If a method expects only one parameter, parenthesis are optional

Constructing Lists

The „cons“-Operator: Creates a new List adding an element (= left operand) to the beginning (=head) of an existing list.
`Nil` = empty list
General form: `head :: tail`

```
val list1 = "I" :: Nil  //List("I")

val list2 = "B" :: list1  //List("B","I")

val list = "O"::"B"::"I"::Nil //List("O","B","I")
```

Deconstructing Lists, and .. and … and …

```
implicit val phoneBook = List(("44333442","Julia"),
    ("34223322","Seppl"),
    ("45645646","Sandra"),
    ("33122221","John")
  )

def findNumber(name:String)(implicit book:List[(String,String)]):Option[String] =
  book match  {
       case (number,person)::_ if person == name => Some(number)
       case _::tail => findNumber(name)(tail)
       case Nil => None
  }

def printResult(result:Option[String]) = result match {
  case Some(number) => println(s"The number is $number")
  case None => println("Number wasnt found")
}

printResult(findNumber("Sandra")) //The number is 45645646
printResult(findNumber("Hugo")) //Number wasnt found
```

Closures

- Closures or Anonymous Functions or Function Literals have the following syntax:

```
(parameterlist) => body
```

- E.g.:

```
(a,b) => a+b  (Parameters might require types!)
```

- Their Type is:

```
(type_1,type_2, …,type_n) => return_type
```

Using Closures with Lists

```
val list = List(23,21,65,12,58,33)

val big1 = list.filter((v) => v > 30)  //List(65, 58, 33)

//or

val big2 = list.filter(_ > 30)  //List(65, 58, 33)
```

Syntactic Sugar: If parameter names are not needed (=every parameter occurs exactly once and in the same order as in the parameter list) the parameter list can be ommitted and parameters replaced by wildcard character

List: map

```
val netPrice = List(100,80,140,70)
```

```
val grossPrice = netPrice.map(_*1.2)
//List(120.0, 96.0, 168.0, 84.0)
```

List: reduce

```
val price = List(100,80,140,70)

val total = price.reduce(_+_) //390
```

List: foldLeft/foldRight

```
val list = List("A","B","C","D")

val leftToRight = list.foldLeft("")(_+_) //ABDC
val rightToLeft = list.foldRight("")(_+_) //ABCD


val foldRight = list.foldRight("")((entry,result) => entry+result) //ABCD
val foldLeft = list.foldLeft("")((result,entry) => entry+result) //DCBA
```

Fold: The Swiss-Army-Knife of the Functional Programmer!

```
def findNumber(name:String)(implicit book:List[(String,String)]):Option[String] =
    book match  {
        case (number,person)::_ if person == name => Some(number)
        case _::tail => findNumber(name)(tail)
        case Nil => None
    }
```

Fold is often used to replace recursion.

```
def findNumber(name: String)(implicit book: List[(String, String)]): Option[String] =
    book.foldLeft(None:Option[String])((r,e) => if (e._2 == name) Some(e._1) else r)
```

Phonebook Extended: Phonebook.txt

```
John Doe, Alte Poststraße 147, 8020 Graz, 0664 4432334
Julia Miller, Hauptplatz 1, 8010 Graz, 3342234
#Comment
Jesse James, Dietrichsteinplatz 3a, 8010 Graz, 5673234
...
```

Phonebook Extended

Regular Expression

```
import scala.io.Source

val line = """(.*?), ?(.*?), ?(\d{4,5}) (.*?), ?([\d -/]*)$""".r
```

Case Class: A Scala „POJO“ in one line

```
case class Entry(name:String,street:String,zip:String,city:String,number:String)
```

Creates a stream of lines

```
def readBook(file:String) =
  Source.fromFile(file).getLines().foldLeft(Map[String,Entry]())((b,l) => l match {
    case line(name,street,zip,city,number) => b + (name -> Entry(name,street,zip,city,number))
    case _ => b
  })
```

RegEx can be nicely used in pattern matching