Functional Programming in Scala

Jacek Wasilewski

Scala

What is Scala? (1)

- Scala
  - modern **multi-paradigm** programming language
  - designed to express common programming patterns in a
    - concise,
    - elegant
    - type-safe way
  - **object-oriented** and **functional**

What is Scala? (2)

- Pure object-oriented - every value is an object
- Functional - every function is a value
- Provides
  - anonymous functions
  - higher-order functions
  - nested functions
  - support for currying
  - pattern matching model
  - …

What is Scala? (3)

- Compatible with Java – it runs on JVM
- Can simplify Java code
- Scala and Java are interoperable- you can integrate the code we write in Scala with our code in Java and vice versa

Scala History

- 2001
  - École Polytechnique Fédérale de Lausanne (EPFL)
  - Martin Odersky
- 2003
  - internal release
- 2004
  - public release
  - Java platform
- Gained attention and is widely used – distributed computing

Scala Basics

Expressions (1)

- Three ways of defining
  - with `def`
  - with `val`
  - with `var`
- Example:

```
def size = 5
val width = 10
var height = 20
```

- `def` and `val` expressions can not be re-assigned
- Re-assignments possible for `var` (not functional!)

Expressions (2)

- Inferred type
  `val size = 5`
- Explicit type
  `val size: Int = 5`
- Scala supports basic types
  `Byte, Short, Int, Long, Char, String, Float, Double, Boolean`
- No need to import a package for these
- Equivalents to Java basic types

Functions (1)

- Functions = expressions or sets of expressions
- Use `def` to define a function - `var` and `val` do not work
- Function can have a list of parameters
  - each parameter needs a type
- It is optional to specify the returning type.
- Last expression is the returning expression
  - possible to use `return` keyword – returning type needs to be specified
    - not purely functional
  - if missing returning expression, `Unit` type is returned – equivalent of `void`

Functions (2)

always

optional but parameters always with types

optional

```
def f(a: Int, b: Int, c: Int): Int = {
    val sum = a + b
    sum * c
}
```

body

usually optional

this will be returned

Equivalent:

```
def f(a: Int, b: Int, c: Int) = (a + b) * c
```

Parameters Evaluation (1)

- ```
  def square(x: Double) = x * x
  def sumOfSquares(x: Double, y: Double) =
      square(x) + square(y)
  ```
- ```
  square(4)
  square(2 + 2)
  ```
- Is there a difference?
- Depends on the evaluation type
  - call-by-value – `def square(x: Double) = ...`
  - call-by-name – `def square(x: => Double) = ...`

Call-by-value Evaluation

```
sumOfSquares(3, 2 + 2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25
```

```
def square(x: Double) = x * x
def sumOfSquares(x: Double,
                 y: Double) =
    square(x) + square(y)
```

Call-by-name Evaluation

```
sumOfSquares(3, 2 + 2)
square(3) + square(2 + 2)
3 * 3 + square(2 + 2)
9 + square(2 + 2)
9 + (2 + 2) * (2 + 2)
9 + 4 * (2 + 2)
9 + 4 * 4
9 + 16
25
```

```
def square(x: => Double) = x * x
def sumOfSquares(x: => Double,
                 y: => Double) =
    square(x) + square(y)
```

Expressions Again

- Three ways
  - with `def`
  - with `val`
  - with `var`
- `def` and `val` can not be re-assigned.
- Difference?
  - `def` stores the expression – like function
  - `val` stores the evaluated value only – like variable

Conditional Expressions

- Java's **`if-else`**
- ```
  def abs(x: Double) = {
    if (x >= 0)
      x
    else
      -x
  }
  ```
- If you use **`if`**, specify **`else`**
- ```
  def abs(x: Double) = {
    if (x >= 0)
      x
    -x
  }
  ```
- Not equivalent to
  ```
  def abs(x: Double): Double = {
    if (x >= 0)
      return x
    return -x
  }
  ```

Nested Functions (1)

```
def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001

def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x))
        guess
    else
        sqrtIter(improve(guess, x), x)

def sqrt(x: Double) = sqrtIter(1.0, x)
```

Nested Functions (2)

```
def sqrt(x: Double) = {
    def isGoodEnough(guess: Double, x: Double) =
        abs(square(guess) - x) < 0.001
    def improve(guess: Double, x: Double) =
        (guess + x / guess) / 2
    def sqrtIter(guess: Double, x: Double): Double =
        if (isGoodEnough(guess, x))
            guess
        else
            sqrtIter(improve(guess, x), x)
    sqrtIter(1.0, x)
}
```

Recursion

- Recursive functions **call themselves** from their bodies.
- ```
  def sum(n: Int): Int = {
      if (n < 1)
          0
      else
          n + sum(n – 1)
  }
  ```
  ```

Tail Recursion (1)

```
def gcd(a: Int, b: Int): Int = {
    if (b == 0)
        a
    else
        gcd(b, a % b)
}
```

```
gcd(21, 14)
gcd(14, 21 % 14)
gcd(14, 7)
gcd(7, 14 % 7)
gcd(7, 0)
7
```

Tail Recursion (2)

```
def sum(n: Int): Int = {
    if (n < 1)
        0
    else
        n + sum(n – 1)
}
```

```
sum(5)
5 + sum(4)
5 + (4 + sum(3))
5 + (4 + (3 + sum(2)))
5 + (4 + (3 + (2 + sum(1))))
5 + (4 + (3 + (2 + (1 + sum(0)))))
5 + (4 + (3 + (2 + (1 + 0))))
15
```

Tail Recursion (3)

- Different evaluations
  - `gcd` forms a **block**
  - `sum` forms a **triangle**
- Shape of evaluation approximates the memory usage
- `gcd` is a **tail-recursive** function

Tail Recursion (4)

- Differences
  - `gcd` does recursion with changing parameters
  - `sum` does recursion with modifying result
- Impacts the performance
- Aim to not modify the result when writing recursive functions
  - this leads to nested re-writings

Tail Recursion (5)

```
def sum(n: Int): Int = {
    if (n < 1)
        0
    else
        n + sum(n – 1)
}
```

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int):
Int = {
        if (n < 1)
            acc
        else
            sumInt(acc + n, n – 1)
    }
    sumInt(0, n)
}
```

Tail Recursion (6)

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int):
Int = {
        if (n < 1)
            acc
        else
            sumInt(acc + n, n – 1)
    }
    sumInt(0, n)
}
```

```
sum(5)
sumInt(0, 5)
sumInt(5, 4)
sumInt(9, 3)
sumInt(12, 2)
sumInt(14, 1)
sumInt(15, 0)
15
```

Tail Recursion (7)

- How to made recursion tail-recursive? (some rules of a thumb)
  - Convert your function into an nested function
  - Add an accumulator to the definition of your nested function
  - Make sure that your nested function does not use outer function
  - Call your nested function as the last thing of our outer function
  - Decide how the accumulator should be initialised
  - In the inner function, decide how the accumulator should be updated
  - One of the inner function branch should return accumulator
  - Check if the condition for final return is correct.
  - If more than one recursive call is needed, you might need more accumulators.

Tail Recursion (8)

```
def sum(n: Int): Int = {
    if (n < 1)
        0
    else
        n + sum(n – 1)
}
```

Tail Recursion (9)

```
def sum(n: Int): Int = {
    if (n < 1)
        0
    else
        n + sum(n – 1)
}
```

```
def sum(n: Int): Int = {
    def sumInt(n: Int): Int = {
        if (n < 1)
            0
        else
            n + sumInt(n – 1)
    }
}
```

Convert your function into an nested function.

Make sure that your nested function does not use outer function.

Tail Recursion (10)

```
def sum(n: Int): Int = {
    def sumInt(n: Int): Int = {
        if (n < 1)
            0
        else
            n + sumInt(n – 1)
    }
}
```

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int):
Int = {
        if (n < 1)
            0
        else
            n + sumInt(acc, n – 1)
    }
}
```

Add an accumulator to the definition of your nested function.

Tail Recursion (11)

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int): 
Int = {
        if (n < 1)
            0
        else
            n + sumInt(acc, n – 1)
    }
}
```

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int): 
Int = {
        if (n < 1)
            0
        else
            n + sumInt(acc, n – 1)
    }
    sumInt(0, n)
}
```

Call your nested function as the last thing of our outer function.

Decide how the accumulator should be initialised.

Tail Recursion (12)

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int):
Int = {
        if (n < 1)
            0
        else
            n + sumInt(acc, n – 1)
    }
    sumInt(0, n)
}
```

```
def sum(n: Int): Int = {
    def sumInt(acc: Int, n: Int):
Int = {
        if (n < 1)
            acc
        else
            sumInt(acc + n, n – 1)
    }
    sumInt(0, n)
}
```

In the inner function, decide how the accumulator should be updated.

One of the inner function branch should return accumulator.

Higher-order Functions (1)

- Functions are **first-class values** (or citizens)
  - they act like **any other value**
- **Higher-order functions** can
  - **accept** functions as parameters
  - **return** functions as a result

Higher-order Functions (2)

- Type
  ```
  (Int, Int) => Int
  ```
- Taking
  ```
  def g(y: Int, func: (Int, Int) => Int): Int = func(y, y)
  ```
- Returning
  ```
  def h(func: Int => Int): (Int => Int) = func
  ```

Higher-order Functions (2)

- Sum all integers between two given numbers:
```
def sumInts(a: Int, b: Int): Int =
    if (a > b) 0 else a + sumInts(a + 1, b)
```
- Sum the squares of all integers between two given numbers:
```
def square(x: Int): Int = x * x
def sumSquares(a: Int, b: Int): Int =
   if (a > b) 0 else square(a) + sumInts(a + 1, b)
```
- Sum the cubes of all integers between two given numbers:
```
def cube(x: Int): Int = x * x * x
def sumSquares(a: Int, b: Int): Int =
   if (a > b) 0 else cube(a) + sumInts(a + 1, b)
```

Higher-order Functions (3)

- Expression:

$$\sum_{a}^{b} f(n)$$

- Code:

```
def f(x: Int): Int = ...
def sum(a: Int, b: Int): Int =
   if (a > b) 0 else f(a) + sum(a + 1, b)
```

Higher-order Functions (4)

- Code:

```
def sum(f: (Int) => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f, a + 1, b)

def id(x: Int): Int = x
def square(x: Int): Int = x * x
def cube(x: Int): Int = x * x * x

sum(id, 2, 5)
sum(square, 2, 5)
sum(cube, 2, 5)
```

Anonymous Functions

- Few one-liners
```
def id(x: Int): Int = x
def square(x: Int): Int = x * x
def cube(x: Int): Int = x * x * x
```
- Naming them might be an over-head
- Possible to define them when used
```
sum(x => x, 2, 5)
sum(x => x * x, 2, 5)
```
- And with specified type
```
(x: Int) => x * x
```

Currying (1)

- ```
  def sum(f: Int => Int, a: Int, b: Int): Int =
      if (a > b) 0 else f(a) + sum(f, a + 1, b)
  def sumInts(a: Int, b: Int): Int = sum(x => x, a, b)
  def sumSquares(a: Int, b: Int): Int = sum(x => x * x, a, b)
  ```
- Need to pass **`a`** and **`b`** but these are not important
- Can we do better?
  ```

Currying (2)

- Currying - technique of transforming a function
  - takes multiple parameters
  - returns function that take less parameters
- Also a way of creating functions based on other functions
- Not limited to functions you define – you can wrap other as well

Currying (3)

- Code
```
def mult(x: Int, y: Int) = x * y
mult(1, 2) // 2
mult(3, 7) // 21
```
- Curried
```
def mult(x: Int)(y: Int) = x * y
mult(1)(2)
mult(3)(7)
```
- Or
```
def mult2(x: Int) = (y:Int) => x*y
mult2(1)(2)
mult2(3)(7)
```
- Then
```
def mult3(x: Int) = mult2(2)(x)
mult3(1)
```
- Equivalent
```
def mult3 = mult(2) _
mult3(1)
```
- Try
```
mult(2)
mult(2) _
(mult(2) _)(3)
def m = mult(2)
def m = mult(2) _
m(3)
```

Currying (4)

- Before:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f, a + 1, b)
def sumInts(a: Int, b: Int): Int = sum(x => x, a, b)
```

- After:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f)(a + 1, b)
def sumInts(a: Int, b: Int): Int = sum(x => x)(a, b)
def sumInts = sum(x => x) _
```