Functional Programming in Scala
Part II

Raj Sunderraman

Tail Recursion

Evaluating a Function Application

function call f(e1,…,en) is evaluated as follows:

- evaluate expressions e1,…,en resulting in values v1,…,vn
- replace f(e1,…,en) by body of function in which actual arguments replace formal parameters of f.

More formally,

def f(x1,…xn) = B

f(v1,…,vn) —> [v1/x1,…,vn/xn]B
where [v1/x1,…,vn/xn]B stands for expression B in which all occurrences of xi are replaced by vi.

[v1/x1,…,vn/xn] is called a substitution.

Rewriting Example (gcd)

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

      gcd(14,21)
—>  if (21 == 0) 14 else gcd(21, 14%21)
—>  if (false)14 else gcd(21, 14%21)
—>  gcd(21, 14%21)
—>  gcd(21, 14)
—>  if (14 == 0) 21 else gcd(14, 21%14)
—>  if (false) 21 else gcd(14, 21%14)
—>  gcd(14, 21%14)
—>  gcd(14, 7)
—>  if (7 == 0) 14 else gcd(7, 14%7)
—>  if (false)14 else gcd(7, 14%7)
—>  gcd(7, 14%7)
—>  gcd(7, 0)
—>  if (0 == 0) 7 else gcd(0, 7%0)
—>  7
```

Tail Recursion is better

If a function calls it self as its last action, the function's stack can be reused.
This is called tail recursion

Tail recursive functions are essentially iterative processes

In Scala, tail-recursive functions can be annotated

```
@tailrec
def gcd(a: Int, b: Int): Int = …
```

Tail recursive version of factorial:

```
def fact(answer: Int, n: Int): Int =
  if (n == 0) answer else fact(n*answer, n - 1)

def factorial(n: Int): Int =
  fact(1,n)
```

Higher Order Functions

Functional languages treat functions as *first-class values*.

i.e. like any other value, a function can be passed as a parameter and returned as a result.

This provides a flexible way to compose programs.

Functions that take other functions as parameters or return functions as results are called *higher-order functions*.

Higher Order Functions - Example

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

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

def fact(x: Int): Int = if (x == 0) 1 else x * fact(x-1)

def sumCubes(a: Int, b: Int): Int =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)

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

Can we factor out the function and reduce all of these to a single function?

Higher Order Functions - Continued

```
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(id, a, b)
def sumCubes(a: Int, b: Int): Int = sum(cube, a, b)
def sumFactorials(a: Int, b: Int): Int = sum(fact, a, b)
```

where

```
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if (x == 0) 1 else x * fact(x - 1)
```

Type A => B is the type of a function that takes an argument of type A and returns a result of type B.

So, Int => Int is the type of functions that map integer to integers

Anonymous Functions

Passing functions as parameters leads to the creation of many small functions.

It is tedious to have to define (using def) and name these functions.

e.g.

```
def str = “abc”;
println(str);
```

vs

```
println(“str”)
```

Can we not have function literals just like String literals?

Anonymous functions are basically function literals

Anonymous Function Syntax

Examples:

```
(x: Int) => x * x * x
```

(x: Int) is the parameter of the function and x * x * x is the body.

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

In general:

(x1: T1, x2: T2, …, xn: Tn) => E   can be expressed using def as follows:

```
{ def f(x1: T1, x2: T2, …, xn: Tn) = E; f }
```

Using anonymous functions, we can write earlier sums in a shorter way:

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```

Products, Factorials, etc

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

def product(a: Int, b: Int): Int =
 if (a > b) 1 else a * product(a+1,b)

def factorial(n: Int): Int = product(1,n)

```
object SumProduct {
  def operate(f: (Int, Int)=>Int, ident: Int, a: Int, b: Int): Int =
    if (a > b) ident else f(a, operate(f, ident, a+1,b))

  def sum(a: Int, b: Int): Int =
    operate((x, y)=>x+y, 0, a, b)

  def product(a: Int, b: Int): Int =
    operate((x, y)=>x*y, 1, a, b)

  def main(args: Array[String]) {
    println(sum(1, 6))
    println(product(1, 6))
  }
}
```

Currying

Motivation:

```
def sumInts(a: Int, b: Int): Int = sum(x => x, a, b)
def sumCubes(a: Int, b: Int): Int = sum(x => x*x*x, a, b)
def sumFactorials(a: Int, b: Int): Int = sum(fact, a, b)
```

parameters a and b get passed on to sum() without any modifications.
Can we get rid of these parameters?

Function returning functions:

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

sum is a function that returns another function, sumF. sumF applies the given function parameter f and sums the results.

Currying continued

With the new definition of sum, we can define

```
def sumInts = sum(x => x)
def sumCubes = sum(x => x*x*x)
def sumFactorials = sum(fact)
```

and use them as follows:

```
sumCubes(1,10) + sumFactorials(1,5)
```

We can even avoid the middlemen sumInt, sumCubes etc…

```
sum(cube)(1,10)
```

sum(cube) returns the sum of cubes function and this function is next applied to arguments (1,10).

Function applications associate to the left:

```
sum(cube)(1,10) = (sum(cube))(1,10)
```

Multiple Parameter Lists - Scala Syntax

Special Syntax in Scala (the following is equivalent to the nested sumF):

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

<u>In general</u>

def f(args1)…(argsn) = E, n>1 is equivalent to

def f(args1)…(argsn-1) = { def g(argsn) = E; g}
where g is a new function symbol or even shorter:

def f(args1)…(argsn-1) = (argsn => E)

Repeating this n times, we get
def f = (args1 => (args2 => … (argsn => E)…))

“Currying”

Example

Given

def sum(f: Int => Int)(a: Int, b: Int): Int = …

what is the type of sum?

Answer:

(Int => Int) => (Int, Int) => Int

Since function types associate to the right, this can be rewritten as

(Int => Int) => ((Int, Int) => Int)

Problem - Higher Order Function

Write a product function similar to sum.
Generalize sum and product using HOFs

```
object SumProduct {
  def operate(f: (Int, Int)=>Int, ident: Int, a: Int, b: Int): Int =
    if (a > b) ident else f(a,operate(f,ident,a+1,b))

  def sum(a: Int, b: Int): Int =
    operate((x,y)=>x+y, 0, a, b)

  def product(a: Int, b: Int): Int =
    operate((x,y)=>x*y, 1, a, b)

  def main(args: Array[String]) {
    println(sum(1, 6))
    println(product(1, 6))
  }
}
```

Example: Finding Fixed Points

A number x is called a <u>fixed point</u> of a function f if

$f(x) = x$

Very useful concept in computer science! For some functions, we can locate the fixed point by starting with an <u>initial estimate</u>, and then applying f in a <u>repetitive</u> manner:

$x, f(x), f(f(x)), f(f(f(x))), \ldots$

until the value does not vary any more (or the change is sufficiently small)

Scala Program to compute Fixed Points

```
val tolerance = 0.0001

def isCloseEnough(x: Double, y: Double) =
  abs((x-y)/x) < tolerance

def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
      val next = f(guess)
      println(next)
      if (isCloseEnough(guess, next)) next
      else iterate(next)
  }
  iterate(firstGuess)
}
```

sqrt(x) can be expressed in terms of a fixed point as follows:

sqrt(x) = the number y such that y * y = x
           = the number y such that y = x/y
           = fixed point of function (y => x/y)

```
def sqrt(x: Double) = fixedPoint(y => x/y)(1.0)
```

Example: Finding Fixed Points

The previous solution goes into an infinite loop!
Values oscillate between 2 and 1!

To fix this, use the function (y => (y + x/y)/2) which takes the average of guess and next guess.

```
def sqrt(x: Double) = fixedPoint(y => (y + x/y)/2)(1.0)
```

This produces:

sqrt(2) = 1.4142135623746899

Functions as Return Values

This example illustrates return vales as functions:

Recall sqrt(x) is a fixed point of function (y => x/y)

The iterative algorithm converges to a solution by averaging successive values. This technique of stabilizing by averaging can be generalized into a function!

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2
```

```
def sqrt(x: Double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

This expresses the algorithm precisely!