Functional Programming in Scala

Raj Sunderraman

Programming Paradigms

- **imperative programming**
  modifying mutable variables, using assignments, and control structures such as if-then-else, loops, continue, return, etc inspired by Von Neumann architecture of computers.

- **functional programming**
  programming without mutable variables, assignments, loops, other imperative control structures; programming with functions; functions become values that are produced, consumed, and composed; functions can be passed as parameters and returned as values.

- **logic programming**
  programming in logic; use logical deductions to run a program; programs are a set of logical rules and facts; solutions focus on “what” aspect of the problem and let the system figure out “how” to solve them.

Orthogonal to

- **Object-oriented programming**

```
MacBook-Pro:~ raj$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_31).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 87 + 145
res0: Int = 232

scala> def size = 2
size: Int

scala> 5*size
res1: Int = 10

scala> def pi = 3.14
pi: Double

scala> def radius = 21.5
radius: Double

scala> (2 * pi) * radius
res2: Double = 135.02
```

```
scala> def square(x: Double) = x * x
square: (x: Double)Double

scala> square(2)
res3: Double = 4.0

scala> square( 5 + 4)
res4: Double = 81.0

scala> square(square(4))
res5: Double = 256.0

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

scala> sumOfSquares(3,4)
res6: Double = 25.0

scala> def power(x: Double, y: Int): Double = scala.math.pow(x,y)
power: (x: Double, y: Int)Double

scala> power(2,3)
res7: Double = 8.0
```

Evaluation of Function Applications (call-by-value strategy)

- Evaluate all function arguments from left to right
- Replace the function application by the function's right hand side, and at the same time
- Replace the formal parameters of the function by the actual arguments

e.g.

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

Substitution Model

- The expression and function evaluation described earlier is called the *substitution model*
- The evaluation *reduces the expression to a value*
- The substitution model is formalized in the lambda-calculus
- Does every expression evaluate to a value in finite steps?

No.

```
def loop: Int = loop

loop
loop
…
…
```

New Evaluation Strategy
(call-by-name strategy)

- Do not reduce the function arguments to values but delay evaluating until very end (call-by-name strategy)

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

call-by-name and call-by-value both result in the same value if

- the reduced expression consists of pure functions
- and both evaluation strategies terminate

call-by-value evaluates arguments once
call-by-name delays evaluation and can get lucky in not evaluating if argument not needed!

CBV vs CBN

Consider the following function:

```
def test(x: Int, y: Int) = x * x
```

For each of the following function applications, indicate which strategy has fewer reduction steps:

test(2,3)
CBV, CBN: test(2,3) = 2*2 = 4

test(3+4,8)
CBV: test(3+4,8) = test(7,8) = 7*7 = 49
CBN: test(3+4,8) = (3+4)*(3+4) = 7*(3+4) = 7*7 = 49

test(7,2*4)
CBV: test(7,2*4) = test(7,8) = 7*7 = 49
CBN: test(7,2*4) = 7*7 = 49

test(3+4,2*8)
Try this on your own…

Scala's Evaluation Strategy

- Scala normally uses CBV
- But if type of function parameter is preceded by => Scala uses CBN

Example:

```
def constOne(x: Int, y: =>Int) = 1

constOne(1+2, loop) = constOne(3,loop) = 1

constOne(loop, 1+2) = constOne(loop,1+2) = constOne(loop,1+2) =…..
```

CBV, CBN, Termination

- As long as both methods terminate, the end result is the same
- What if termination is not guaranteed?
  - if CBV terminates then CBN will also terminate
  - not vice versa

Example of CBN terminating but CBV not terminating

```
def first(x: Int, y: Int) = x
```

recall

```
def loop:Int = loop
```

first(1,loop) under CBN will terminate but will loop forever under CBV

val vs var vs def

val and var use CBV semantics; i.e. expression is evaluated on definition
val defines a fixed value
var defines a variable - which can be modified

```
val x = (1+2)
val y = 3
var z = 2*y
x = 9 //  error
z = 12 // ok
```

def uses CBN semantics; i.e. expression is evaluated when needed

e.g.

def x = 1 + 2
at this point x is not evaluated to 3

val y = 2 * x
at this point 2*(1+2) is evaluated to make y = 6

Conditional Expressions

Scala provides an if-then-else for expressions to be able to choose from two alternatives.

```
def abs(x: Int) = if (x >= 0) x else -x
```

Here (x >= 0) is a predicate with type Boolean

Boolean expressions can be composed of

```
true
false
!b
b && b
b || b
```

and of the usual comparison operations

```
<=, >=, <, >, ==, !=,
```

Rewrite Rules for Booleans/if-then-else expressions

!true —> false
!false —> true
true && e —> e
false && e —> false
true || e —> true
false || e —> e

&&, || use short-circuit evaluation (second operand need not be evaluated)

<u>Rewrite rule for</u>

if (b) then e1 else e2

if (true) then e1 else e2 —> e1
if (false) then e1 else e2 —> e2

Exercise

Implement functions and and or using conditional expressions such that

and(x,y) == (x && y)
or(x,y) == (x || y)

Solution:

```
def and(x: Boolean, y: Boolean): Boolean =
  if (x) then y else false

def or(x: Boolean, y: Boolean): Boolean =
  if (x) then true else y
```

Value Definitions

Similar to function parameters, the distinction by-name and by-value is available in definitions.

```
def square(x: Int): Int = x*x
val x = 2
val y = square(x)
```

Here y refers to 4 and not square(2)

```
def loop: Boolean = loop
```

```
def x = loop // OK
val x = loop // infinite loop
```

Example Program: Square Root using Newton's Method

To compute sqrt(x):

- Start with initial estimate y (let y = 1)
- Repeatedly improve the estimate by taking the mean of y and x/y

- Let x = 2

Estimate	Quotient	Mean
1	2/1 = 2	1.5
1.5	2/1.5= 1.333	1.4167
1.4167	2/1.4167=1.4118	1.4142
1.4142 …..		

Example Program: Square Root using Newton's Method

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

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

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

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

Note: Recursive functions in Scala need an explicit return type.

Remark: This version is not very accurate for very very small numbers such 0.001, 0.1e-20 and for very large numbers it may not terminate! e.g. for 1.0e20

Blocks and Lexical Scope

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (goodEnough(guess, x)) guess
    else sqrtIter(improve(guess,x), x)

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

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

  sqrtIter(1.0, x)
}
```

A block is delimited by braces {…}
It contains a sequence of definitions or expressions
The last element of a block is an expression that defines the block
A block is itself an expression (whose value is the value of the last expression)

Blocks and Visibility

Example:

```
val x = 2
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
} + x
```

A block is delimited by braces {…}
It contains a sequence of definitions or expressions
The last element of a block is an expression that defines the block
A block is itself an expression (whose value is the value of the last expression)

In the above example, result will have a value of 18.

Definitions in inner blocks “shadow” definitions in outer blocks with same name.
So, definitions in outer boxes are visible in inner blocks if they are not shadowed

sqrt function - take 3

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (goodEnough(guess)) guess
    else sqrtIter(improve(guess))

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

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

  sqrtIter(1.0)
}
```

The x: Double in the parameter of the outer block is “visible” in the inner blocks.

Semi Colons at end of statements

Semi colons at the end of lines are in most cases optional

```
val x = 1;
```

But,

```
val x = 5
val y = x + 1; y*y
```

Multi-line Expressions

```
longExpression
+ longExpression
```

will be treated as 2 expressions. So, write

```
(longExpression
+ longExpression)
```

or

```
longExpression +
longExpression
```