Functional Programming in Scala

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

09, 2016

Outline

Function are values, i.e., a function can be

	Value	Function
• Anonymous	3	x => x + 1
• Assigned to a variable	x = 3	f = x => x + 1
• Passed as input/output parameter	f(3)	f(x => x + 1)
• Created dynamically	3 + 4	$f \circ g$

- Imperative languages $\Rightarrow$ Von Neumann Architecture
  - Efficiency
- Functional languages $\Rightarrow$ Lambda Calculus
  - A solid theoretical basis that is also closer to the user, but
  - relatively unconcerned with the architecture of the machines on which programs will run

- A mathematical function is
  - a mapping of members of one set, called the domain set, to another set, called the range set
- A **lambda expression** specifies the parameter(s) and the mapping of a function in the following form
  λ(x) x * x * x
  for the function cube   cube(x) = x * x * x
- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression
  (λ(x) x * x * x)(2)   which evaluates to 8

Higher-order Functions

- A higher-order function is one that either takes functions as parameters or yields a function as its result, or both
- For example,
  - Function composition
  - Apply-to-all
  - Forall/Exists
  - Insert-left/Insert-right
  - Functions as parameters
  - Closures

Function Composition

A function that

- takes two functions as parameters and
- yields a function whose value is the first actual parameter function applied to the application of the second

  $f \circ g = f : (g : x)$

  For $f(x) = x + 2$; $g(x) = x * x$; $f \circ g\ (x) = x * x + 2$

Example in Scala,

```
val f = (x:Double) => x + 2
val g = (x:Double) => x * x
val h = f compose g
h(3)
val k = f andThen g
k(3)
```

A functional form that

- takes a single function as a parameter and
- yields a list of values obtained by applying the given function to each element of a list of parameters

$$\alpha f :< x_1, x_2, ..., x_n >=< f : x_1, f : x_2, ..., f : x_2 >$$

For h(x)=x*x $\Rightarrow$ αh:(1,2,3) yields (1,4,9)

Example in Scala,

```
List(2,3,4).map((x:Int) => x * x)
def inc (x:Int) = x + 1
List(4,5,6).map(inc)
```

A functional form that

- takes a single **predicate** function as a parameter and
- yields a value obtained by applying the given function to each element of a list of parameters and take the **and/or** of the results

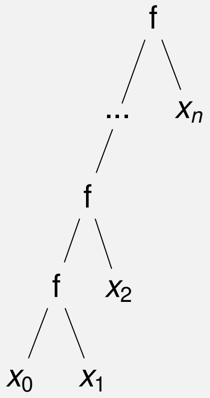
$$\forall f :< x_1, x_2, ..., x_n >= \bigcap f : x_i$$
$$\exists f :< x_1, x_2, ..., x_n >= \bigcup f : x_i$$

Example in Scala,

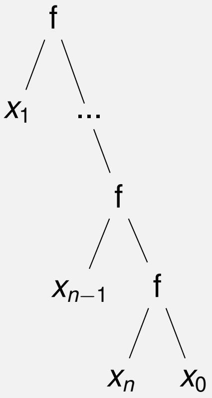
```
def isEqualToThree(x:Int) = x == 3
List(2,3,4).forall(isEqualToThree)
               // yield false
List(2,3,4).exists(isEqualToThree)
               // yield true
```

Insert Left / Insert Right

$/f :< x_0 >, < x_1, x_2, ..., x_n >$



$\backslash f :< x_0 >, < x_1, x_2, ..., x_n >$



Example in Scala

```
List(2,3,4).foldLeft(0)((a,b) => a+b) // yield 9
List(2,3,4).foldLeft(1)((a,b) => a*b) // yield 24
List(2,3,4).foldLeft("A")((a,b) => a + b)
                 // yield "A234"
List(2,3,4).foldRight("A")((a,b) => a + b)
                 // yield "234A"
```

Functions as Parameters

In user-defined functions, functions can be passed as parameters.

```
def apply(x:Int)(f:Int=>Int) = f(x)
val inc1 = (x:Int) => x + 1
val sq = (x:Int) => x * x
val fl = List(inc1,sq)
fl.map(apply(3)) //yield List(4,9)
```

Closure [1]

"An object is data with functions. A closure is a function with data." - John D. Cook

```
def power(exp:Double) =
            (x:Double) => math.pow(x,exp)
val square = power(2)
square(4) //yield 16.0
val cube = power(3)
cube(3) //yield 27.0
```

Closure = function + binding of its free variables

$f : X_1 \times X_2 \times ... \times X_n \rightarrow Y$

curry: $f : X_1 \rightarrow X_2 \rightarrow ... \rightarrow X_n \rightarrow Y$

Example in Scala

```scala
def add(x:Int, y:Int) = x + y
add(1,3)
add(1) add(1)(3)
def plus(x:Int)(y:Int) = x + y
plus(1)(3)
val inc1 = plus(1) _
inc1(3)
val addCurried = (add _).curried
val plusUncurried = Function.uncurried(plus _)
```

Read more on Partially Applied Functions [2]

Immutability

- Immutable: Cannot change
- In Java, strings are immutable
  "Hello".toUpper() doesn't change "Hello" but returns a new string "HELLO"
- In Scala, **val** is immutable
  val num = 12
  num = 10 // wrong
- Pure functional programming: No mutations
- Don't mutate—always return the result as a new value
- Functions that don't mutate state are inherently parallelizable

Example on Immutability

```
abstract class IntStack
   def push(x: Int): IntStack =
                     new IntNonEmptyStack(x, this)
   def isEmpty: Boolean
   def top: Int
   def pop: IntStack
class StackEmpty extends IntStack
   def isEmpty = true
   def top = error("EmptyStack.top")
   def pop = error("EmptyStack.pop")
class IntNonEmptyStack(elem: Int, rest: IntStack)
                           extends IntStack
   def isEmpty = false
   def top = elem
   def pop = rest
```

- Body of a function is an expression, i.e. evaluating to a value
- If there are many expressions, the value of the last executed expression will be returned
- No **return** required

Example in Scala,

```
def fact(x:Int):Int =
   if (x == 0) 1 else x * fact(x - 1)
val s = for (x <- 1 to 25 if x*x > 50) yield 2*x
```

- Like **switch**, but much more powerful

Syntax:
<exp > **match** {
case <pattern 1 > => <exp1>
case <pattern 2 > => <exp2>
...
}
Example in Scala,

```
def mathTest(x : Int): String = x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
}
```

- With recursive functions, return type is obligated
  def fact(n:Int)**:Int** = **if** (x == 0) 1 **else** n * fact(n - 1)
- Need **def** because the name is used on the right
- Iteration (while, for) can always be expressed as recursion

Example in Scala,

```
def mem(x:Int,lst:List[Int]):Boolean = lst match {
    case List() => false
    case head :: tail => if (x == head) true
                         else mem(x,tail)
}
```

Lazy evaluation [4]

- Expressions are **eagerly** evaluated, by default, where they appeared
- Lazy evaluation means the expression is just evaluated when the associated variable is firstly referred.
  lazy val x = 1 + y
  The expression 1 + y is evaluated just when x is firstly used
- Pass-by-name parameter
  def foo(b:Boolean,x**:=>**Int,y**:=>**Int) = if (b) x else y
  foo(a==0,1,b/a)

Type Inference

- Scala is strongly typed. Any value has a type guarantee
- Just like C++:
  ```
  char* greeting = "Hello";
  greeting = 42;//Type Error
  ```
- But without having to declare types:
  ```
  val greeting = "Hello";
  ```
- Contrast with scripting languages such as JavaScript
  ```
  var greeting = 'Hello';//This is JavaScript
  greeting = 42;//Ok
  alert(greeting.length);//Runtime Error
  ```

Type Inference (cont'd)

- Can override inferred type (only to a supertype, of course).
  ```
  var greeting : Any = "Hello"
  greeting = 42//Ok
  ```
- Parameter type must be declared
  ```
  def mistery (x) = 42 * x // Error
  def mistery (x : Int) = 42 * x // Ok
  ```
- Return type can be inferred
  If x is Int, then 42 * x is also Int
- Exception, recursive function requires return type declared

Parameter Inference from Context

- When a function parameter type is known, an anonymous function can be supplied without specifying its parameter types
  def twice(f: (Int)=>Int,x:Int) = f(f(x))
  twice(x=>42*x, 3) // Ok, x:Int is inferred from context
- Very useful when calling library functions
  List(1,2,3).filter(x=> x%2==0)
  - List[A].filter(p:(A)=>Boolean) : List[A]
  - A is Int since List(1,2,3) is a List[Int]
  - p: must be (Int) => Boolean
  - X must be Int

Parameter Simplification

- Ok to omit () around a single inferred parameter

  ```
  List(1, 2, 3).filter(x => x % 2 == 0)
  List(1, 2, 3).sortWith((x,y) => x > y)
  ```

  // need () with 2 or more parameters
- Use _ for a parameter that occurs only once in a body

  ```
  List(1, 2, 3).filter(_ % 2 == 0)
  List(1, 2, 3).sortWith( _ > _)
  ```

Summary

- Functional programming languages use **function application**, **conditional expressions**, **recursion**, and **functional forms** to control program execution instead of imperative features such as variables and assignments
- Purely functional languages have advantages over imperative alternatives, but their **lower efficiency** on existing machine architectures has prevented them from enjoying widespread use

References I

[1] Methods and Closures, http://www.artima.com/pins1ed/functions-and-closures.html, 19 06 2014.

[2] Function Currying in Scala, http://www.codecommit.com/blog/scala/function-currying-in-scala, 19 06 2014.

[3] Case classes and pattern matching, http://www.artima.com/pins1ed/case-classes-and-pattern-matching.html, 19 06 2014.

[4] Control Abstraction, http://www.artima.com/pins1ed/control-abstraction.html, 19 06 2014.