Computer Programming with Scala

Week 3: Functional Programming (FP)

Martin Quinson

November 2015

école
normale
supérieure
rennes

Named and Default Parameters

Refering paramters by name

```
scala> def speed ( distance: Double , time: Double ): Double =
       distance / time
speed: ( distance: Double , time: Double ) Double

scala> speed (100 , 20)
res0: Double = 5.0
scala> speed ( time = 20 , distance = 100)
res1: Double = 5.0
```

Default Parameter values

```
scala> def speed ( distance: Double=0.1 , time: Double ): Double =
       distance / time
speed: ( distance: Double , time: Double ) Double

scala> val Bolt = speed (time=0.00266111)    # 9.58s = 0.00266111h
Bolt: Double = 37.578303790523506
```

Local Functions

- Functions can be defined within other functions
- Functions are only visible in surrounding scope
- Inner function can access namespace of surrounding function

```
def filterEven(name: String, li:List[Int]):List[Int] = {
  def isEven(i:Int) = {
    println(name + " contains " + i)
    (i%2 == 0)
  }
  li match {
    case Nil => Nil
    case x::xs if (isEven(x)) => x::filterEven(name, xs)
    case x::xs                =>    filterEven(name, xs)
  }
}
scala> filterEven("my list", List(1,2,3,4,5) )
my list contains 1
my list contains 2
my list contains 3
my list contains 4
res0: List[Int] = List(2, 4)
```

Higher Order Functions

First Class Functions $\leadsto$ Functions are regular values

- Can be assigned to a variable
- Can be passed as arguments to functions
- Can be returned by other functions

Higher Order Functions = Functions taking function as parameter

- Powerful abstraction mechanism

```
def my_map (lst: List[Int] , fun: Int => Int) :List[Int] =
   for (l <- lst) yield fun (l)

val numbers = List (2 ,3 ,4 ,5)
def addone ( n : Int ) = n + 1

scala> my_map ( numbers , addone )
res0: List[Int] = List (3 , 4 , 5 , 6)
```

Higher Order Functions on class List

Filtering and Partitioning

- Functions as (named) values

```
val li = List(1, 2, 3, 4, 5)
def isEven (n: Int) = n%2 == 0
scala> li filter isEven
res0: List[Int] = List(2, 4)
```

- With an anonymous functions

```
scala> li filter (i => i%2 == 0)
res1: List[Int] = List(2, 4)
scala> li filter (_%2 == 0)
res2: List[Int] = List(2, 4)
```

```
scala> li partition (_%2 == 0)
res3: (List[Int], List[Int]) = (List(2, 4),List(1, 3, 5))
```

- Also defined: `find`, `takeWhile`, `dropWhile` and span. Check the doc

Mapping over elements

```
scala> li map (_ + 1)
res4: List[Int] = List (2, 3 , 4 , 5 , 6)
scala> li foreach (x => print(x + ", ") )
1, 2, 3, 4, 5,
```

Folding `List :/` and `:\`

- Reduce all elements into a single value using the provided function

```
scala> def sum(xs: List[Int]): Int = (0 /: xs) (_ + _)
scala> sum( List(1,2,3,4) )
res0:Int = 10    # = 0 + 1 + 2 + 3 + 4
```

Folding `List` `:/` and `:\`

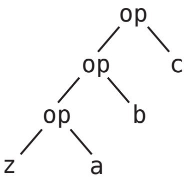
- Reduce all elements into a single value using the provided function

```
scala> def sum(xs: List[Int]): Int = (0 /: xs) (_ + _)
scala> sum( List(1,2,3,4) )
res0:Int = 10    # = 0 + 1 + 2 + 3 + 4
```

```
scala> def sumRight(xs: List[Int]): Int = (0 \: xs) (_ + _)
scala> sum( List(1,2,3,4) )
res0:Int = 10    # = 0 + 4 + 3 + 2 + 1
```

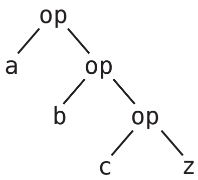
- $\boxed{\texttt{(z :/ xs) (op)}}$ `z`: initial value, `xs`: list, `op`: operation to apply

`(z :/ List(a,b,c)) (op)`



Fold Left

`(z :\ List(a,b,c)) (op)`



Fold Right

- Same result if `op` is associative; performance may vary

Partially Applied Functions: Functions as Objects

- Passing _ in place of parameter list creates a partially applied function
- Function Object automatically built by the compiler

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int

scala> sum(1, 2, 3)
res0: Int = 6
                                          # This creates an object of type
scala> val a = sum _                      # <function3> (because sum takes
a: (Int, Int, Int) => Int = <function3>   # 3 parameters)

scala> a(1,2,3)                           # Apply parameters to partially
res1: Int = 6                             # applied function => function call

scala> a.apply(1,2,3)                     # Exactly as before
res2: Int = 6

scala> val b = sum(1, _:Int , 3)          # Here, only one parameter remains
b: Int => Int = <function1>               # free. Thus the type <function1>
```

Function Objects and Implicits

- Underscore optional in contexts that require a function (and only there)

```
scala> someNumbers.foreach(print)  # no need to write (print _) here
1234

scala> val c = sum
<console>:5: error: missing arguments for method sum...
follow this method with ‘_’ if you want to treat it as a partially applied function
```

- Haskell doesn’t require the _ for the partially applied function (implicit)
- But Scala targets Java developers $\rightsquigarrow$ needs to detect missing parameters
- Thus the need for _ in general context
- _ is still optional where it can be no mistake

Closures

- Free variable: variable without a value; Bound variable: variable with a value

Closure = when a function refers to an external free variable

```
scala> var more = 1
scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function1>
scala> addMore(10)
res0: Int = 11
```

- This function object is a closure, because it encloses (packs) the free variables
- In scala, captures the variables, not the values (Java captures constants)

```
scala> more = 3 ; addMore(10)
res1: Int = 13
```

Building Closures

```
scala> def makeIncreaser(more: Int) = (x: Int) => x + more
makeIncreaser: (more: Int)Int => Int

scala> val inc9999 = makeIncreaser(9999)
inc9999: (Int) => Int = <function1>
```

- Each closure gets its own copy of the captured elements

Other Considerations

Code Factorization with Higher Order Functions

```scala
def withOdd(nums: List[Int]): Boolean={
  var exists = false
  for (num <- nums)
    if (num % 2 == 1)
      exists = true
  exists
}
```

```scala
def withOdd(nums: List[Int]): Boolean=
     nums.exists(_%2 == 1)
```

- Q1: Implement List.length with :/
- Q2: List.reverse() with :/
- Q3: Type of `((x:Double) => x+1)`
- Q4: Write a function that adds 1 to every elements of a List[Int]
- Q5: Define $S = \{a \times 2 \mid a \in [1, 100] \wedge a < 99 \wedge a^3 > 9\}$
- Q6: Explain `((_:Double)+2)` and `(_:String).size`

Tail Recusion Optimization

- Scala can optimize every tail recursive functions into a while loop
- Works only for basic forms (not mutually recursive, not partially applied)

Lazy variables `lazy val ui = ...`

- Only evaluated on need (usually, scala values are evaluated when defined)

Currying

- Defining functions with multiple parameter lists

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (x: Int)(y: Int)Int

scala> curriedSum(1)(2)
res5: Int = 3
```

- You are actually defining two functions back to back

```
scala> def first(x: Int) = (y: Int) => x + y
first: (x: Int)(Int) => Int

scala> val second = first(1)
second: (Int) => Int = <function1>
```

- Currying and Partially applied function

```
scala> curriedSum(1)
<console>:14: error: missing arguments for method curriedSum; follow this method wi

scala> curriedSum(1)_
res6: Int => Int = <function1>
```

- This explains the :/ syntax

Function Composition

```
def f(s: String) = "f(" + s + ")"
def g(s: String) = "g(" + s + ")"
```

compose makes a new function that composes its parameters: f(g(x))

```
scala> val FoG = f _ compose g _
FoG: String => String = <function1>

scala> FoG("yah")
res0: String = f(g(yah))
```

andThen does the same in the reverse order: g(f(x))

```
scala> val FthenG = f _ andThen g _
FthenG: String => String = <function1>

scala> FthenG("yah")
res1: String = g(f(yah))
```

PartialFunction

- It's a function that is not defined for every parameter value
- It is not a Partially Applied Function

```
scala> val one: PartialFunction[Int, String] = { case 1 => "one" }
one: PartialFunction[Int,String] = <function1>

scala> one.isDefinedAt(1)
res0: Boolean = true

scala> one.isDefinedAt(2)
res1: Boolean = false
```

- You can chain PartialFunctions with orElse

```
scala> val two: PartialFunction[Int, String] = { case 2 => "two" }
two: PartialFunction[Int,String] = <function1>
scala> val three: PartialFunction[Int, String] = { case 3 => "three" }
scala> val wildcard: PartialFunction[Int, String] = { case _ => "something else" }
scala> val partial = one orElse two orElse three orElse wildcard
partial: PartialFunction[Int,String] = <function1>

scala> partial(5)
res1: String = something else

scala> partial(3)
res2: String = three

scala> partial(2)
res3: String = two

scala> partial(1)
res4: String = two
```

case class and Pattern Matching

Defining a case class

```
abstract class Tree
case class Branch(left: Tree, right: Tree) extends Tree
case class Leaf(x: Int) extends Tree
```

Declaring a value

```
val t = Branch(Branch(Leaf(1), Leaf(2)), Branch(Leaf(3), Leaf(4)))
```

Pattern Matching

```
def sumLeaves(t: Tree): Int = t match {
  case Branch(l, r) => sumLeaves(l) + sumLeaves(r)
  case Leaf(x) => x
}
```

Matching on Variable Declaration

```
scala> val b = Branch(Leaf(1), Leaf(2))
b: Branch

scala> val Branch(_, l) = b
l: Tree = Leaf(2)
```

Parametrized types

Defining a Tree[String] (without duplication)

```
abstract class Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[A](x: A) extends Tree[A]

scala> val t = Branch(Branch(Leaf("a"), Leaf("b")), Branch(Leaf("c"), Leaf("d")))
t: Branch[String] = Branch(Branch(Leaf(a),Leaf(b)),Branch(Leaf(c),Leaf(d)))
```

- Tree is a trait while `Tree[Int]` is a type

Parametrized types

Defining a Tree[String] (without duplication)

```
abstract class Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[A](x: A) extends Tree[A]

scala> val t = Branch(Branch(Leaf("a"), Leaf("b")), Branch(Leaf("c"), Leaf("d")))
t: Branch[String] = Branch(Branch(Leaf(a),Leaf(b)),Branch(Leaf(c),Leaf(d)))
```

- `Tree` is a trait while `Tree[Int]` is a type

The `Option` type

- When you search for a value in a list, you don't know whether you'll find it
- An `Option[A]` can either be a `Some` (containing a value) or a `None`

```
val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

scala> capitals get "France"
res0: Option[java.lang.String] = Some(Paris)

scala> capitals get "North Pole"
res1: Option[java.lang.String] = None
```

- Q: Define the follwing methods over Tree[A]: find, map, foreach, filter

Variance

- Would you say that a `Tree[Int]` is-a `Tree[Any]`?
- Is it ok to provide a `Tree[Int]` where a `Tree[Any]` was expected?

Variance

- Would you say that a `Tree[Int]` is-a `Tree[Any]`?
- Is it ok to provide a `Tree[Int]` where a `Tree[Any]` was expected?
- Intuitively, yes, but by default, Scala generic types are nonvariant
- If your type `Tree` is covariant (flexible), just say so:

```
trait Tree[+T] { ... }    # a Tree[Int] is indeed a Tree[Any]
```

- In some cases, you can tell that your type is contravariant

```
trait Tree[-T] { ... }    # WRONG! a Tree[Any] cannot be a Tree[Int]!
```

- Purely functionnal types are often covariant

Variance

- Would you say that a `Tree[Int]` is-a `Tree[Any]`?
- Is it ok to provide a `Tree[Int]` where a `Tree[Any]` was expected?
- Intuitively, yes, but by default, Scala generic types are nonvariant
- If your type `Tree` is covariant (flexible), just say so:

```
trait Tree[+T] { ... }    # a Tree[Int] is indeed a Tree[Any]
```

- In some cases, you can tell that your type is contravariant

```
trait Tree[-T] { ... }    # WRONG! a Tree[Any] cannot be a Tree[Int]!
```

- Purely functionnal types are often covariant

Mutable Data is often not Covariant

```
class Cell[+T](init: T) { # WRONG
  private[this] var current = init
  def get = current
  def set(x: T) { current = x }
}
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get   # WOOOOPS
```

- This would sets the string to 1!
- Type system actually prevents this

```
Cell.scala:7: error: covariant type T
occurs in contravariant position in
type T of value x
def set(x: T) = current = x
           ^
```

Variance and subtyping

```
class Animal { val sound = "rustle" }
class Bird extends Animal { override val sound = "call" }
class Chicken extends Bird { override val sound = "cluck" }
```

Specialization: You need a `Bird` and have a `Chicken`. That's OK.

- This is the Liskov Substitution Principle

```
scala> val a:Bird = new Chicken
a: Bird = Chicken@1fcaea93
```

```
scala> a.sound
res8: String = cluck
```

- But you cannot use an `Animal` in place of a `Bird`

Variance and subtyping

```
class Animal { val sound = "rustle" }
class Bird extends Animal { override val sound = "call" }
class Chicken extends Bird { override val sound = "cluck" }
```

Specialization: You need a `Bird` and have a `Chicken`. That's OK.

- This is the Liskov Substitution Principle

```
scala> val a:Bird = new Chicken
a: Bird = Chicken@1fcaea93
```

```
scala> a.sound
res8: String = cluck
```

- But you cannot use an `Animal` in place of a `Bird`

Function parameters are contravariants

- Can't use a function that takes a `Chicken` for a function that takes a `Bird`
  - It would choke on a `Duck`; But a function that takes an `Animal` is OK

```
scala> val getTweet: (Bird => String) = ((a: Animal) => a.sound )
getTweet: Bird => String = <function1>
```

Variance and subtyping

```
class Animal { val sound = "rustle" }
class Bird extends Animal { override val sound = "call" }
class Chicken extends Bird { override val sound = "cluck" }
```

Specialization: You need a `Bird` and have a `Chicken`. That's OK.

- This is the Liskov Substitution Principle

```
scala> val a:Bird = new Chicken                scala> a.sound
a: Bird = Chicken@1fcaea93                     res8: String = cluck
```

- But you cannot use an `Animal` in place of a `Bird`

Function parameters are contravariants

- Can't use a function that takes a `Chicken` for a function that takes a `Bird`
  - It would choke on a `Duck`; But a function that takes an `Animal` is OK

```
scala> val getTweet: (Bird => String) = ((a: Animal) => a.sound )
getTweet: Bird => String = <function1>
```

Function return value are covariant

- Need a function that returns a `Bird`? A function returning a `Chicken` is OK

```
scala> val hatch: (() => Bird) = ((_) => new Chicken )
hatch: () => Bird = <function0>
```

Polymorphism Bounds

Refine your polymorphism

```
scala> def cacophony[T](things: Seq[T]) = things map (_.sound)
<console>:7: error: value sound is not a member of type parameter T
       def cacophony[T](things: Seq[T]) = things map (_.sound)
                                                        ^


scala> def biophony[T <: Animal](things: Seq[T]) = things map (_.sound)
biophony: [T <: Animal](things: Seq[T])Seq[java.lang.String]

scala> biophony(Seq(new Chicken, new Bird))
res5: Seq[java.lang.String] = List(cluck, call)
```

- biophony takes any T that *is-a* Animal

Polymorphism Bounds

Refine your polymorphism

```
scala> def cacophony[T](things: Seq[T]) = things map (_.sound)
<console>:7: error: value sound is not a member of type parameter T
       def cacophony[T](things: Seq[T]) = things map (_.sound)
                                                         ^


scala> def biophony[T <: Animal](things: Seq[T]) = things map (_.sound)
biophony: [T <: Animal](things: Seq[T])Seq[java.lang.String]

scala> biophony(Seq(new Chicken, new Bird))
res5: Seq[java.lang.String] = List(cluck, call)
```

- biophony takes any T that *is-a* Animal

Lower bound: List[T] defines ::(elem T) but also ::(U >: T)

```
scala> val flock = List(new Bird, new Bird)
flock: List[Bird] = List(Bird@7e1ec70e, Bird@169ea8d2)

scala> new Chicken :: flock
res6: List[Bird] = List(Chicken@56fbda05, Bird@7e1ec70e, Bird@169ea8d2)
scala> new Animal :: flock
res7: List[Animal] = List(Animal@56fbda05, Bird@7e1ec70e, Bird@169ea8d2)
```

More on Polymorphism

Quantification: When one you don't care about one type

```
scala> def count[A](l: List[A]) = l.size
count: [A](List[A])Int                          # A is useless

scala> def count(l: List[_]) = l.size
count: (List[_])Int                             # It's thus omitted
```

More on Polymorphism

Quantification: When one you don't care about one type

```
scala> def count[A](l: List[A]) = l.size
count: [A](List[A])Int                        # A is useless

scala> def count(l: List[_]) = l.size
count: (List[_])Int                           # It's thus omitted
```

Structural Types: specify type requirements by interface structure

```
scala> def foo(x: { def get: Int }) = 123 + x.get
foo: (x: AnyRef{def get: Int})Int

scala> foo(new { def get = 10 })
res0: Int = 133
```

- x parameter must be provided with a get function to be concrete
- The second class makes an ad-hoc anonymous class

More on Polymorphism

Quantification: When one you don't care about one type

```
scala> def count[A](l: List[A]) = l.size
count: [A](List[A])Int                          # A is useless

scala> def count(l: List[_]) = l.size
count: (List[_])Int                             # It's thus omitted
```

Structural Types: specify type requirements by interface structure

```
scala> def foo(x: { def get: Int }) = 123 + x.get
foo: (x: AnyRef{def get: Int})Int

scala> foo(new { def get = 10 })
res0: Int = 133
```

- x parameter must be provided with a get function to be concrete
- The second class makes an ad-hoc anonymous class

Erasure

- Unfortunately, the JVM erases every type specialization
- From `List[Int]`, only `List[_]` remains at runtime

Take home messages

Functional Programming

- Avoid mutable values, prefer expressions over statements
- Higher Order: pass functions as parameters (to factorize behavior)
- Partially Applied Functions: Function objects as first-class citizens
- Closures: functions that encapsulate some external state
- Currying: functions with multiple parameter lists
- Parametrized types: containers such as `Tree[A]`
- Variance permits to refine what we expect (the type system to our rescue)

FP in Scala

- Having both OOP and FP is nice and funny, but that's a lot of tools
- Getting used to them requires a lot of practice
- Some Scala choices are very debatable (it targets the Java ecosystem)