Scala Notes

Peter Thompson

September 5, 2019

Contents

Part I

Functional Principles

Chapter 1

Functional Paradigms

1.1 Functional Programming

Mutation is where an attribute of a variable can change while the identity of the variable is maintained. For example, could define a polynomial class, then set a certain coefficient to a particular value.

Functional programming is a programming strategy that avoids mutation/reliance on state information. Immutable values are used. These can be transformed, but the idea is to minimise side effects. We don't want to pass an argument to a function that will then modify that argument. Input goes in, return value comes out with inputs unchanged.

The restricted definition of a functional programming framework is one in which there are no mutable variables, assignments, or imperative conttrol structures In a wider sense, functional programming can be carried out in any language that allows the construction of elegant programs that focus on functions

In scala, functions are first class objects. They can be treated and passed around just like any other variable (as in python)

1.2 call-by-name, call-by-value

Functions arguments in scala can be handled two ways. The argument can be called by name, or called by value.

```
//call by value
def CBVfunc(a:Int):Int = {...}
```

Function CBVfunc takes an int, returns an Int. The integer argument a is evaluated when the function is called.

```
//call by name
def CBNfunc(a: =>Int): Int = {...}
```

Function CBNfunc takes an integer argment (a), which is evaluated when it needs to be (or not at all!).

Not all functions terminate, infinte loops are a thing

Both call by value and call by name will reduce to the same outcome provided

- the reduced expression consists of pure functions (no state information/-side effects?)
- both evaluations terminate (no infinte loops)

if call by value terminates, then call by name will also converse is not guarenteed: call by name termination does not imply call by value termination in call by name, unused arguments are not evaluated

- could have a function that takes two arguments. The second argument is not used (always)
- could pass a non-terminating input to the CBN function, which is not used. no big deal
- call by value will try to evaluate it and get stuck

Below is an example of a function that will terminate when arguments are called by name but not when called by value.

```
// an infinite loop, this run indefinitely when evaluated
def loop = loop

// call by value function, arguments are evaluated once when the
    function is called
def mooseCBV(a: Int, b: Int):Int = a

// this evaluates loop, starting the infinte loop...
mooseCBV(1,loop)

// same as above, but using call-by-name (=>)
def mooseCBN(a: =>Int, b: => Int) = a

// returns 1, the loop is not evaluated
mooseCBN(1,loop)
```

Scala uses call by value by default, unless the function arguments are defined with `=>`

1.3 Conditionals

Boolean operations don't always need to evaluate the right hand operand (short circuit evaluation)

```
true && e -> e
false && e -> false
true || e -> true
false || e -> e
```

Things can be defined by name or by value. so `def x =loop` is a function, it is not evaluated untill it needs to be (call by name).

`val x =loop` evaluates to loop (call by value) immediately. this will kill your scala session/repl.

1.4 Recursion

Recursive functions must always have their return type explicitly defined (to make the compiler's life easier).

1.4.1 Tail Recursion

If a function calls itself as its last action, the stack frame can be reused Essentially it acts the same as a loop

If a functions last action is to call a function, (maybe the same, maybe different function), then the stack frame can be used - this is a tail-call (tail recursion is recursive tail-calling).

Tail recursive factorial example(works)

```
def factorial(N:Int) = {
    @scala.annotation.tailrec
    def currentProd(n:Int, prod:Int) :Int = {
        if (n==0) prod
        else currentProd(n-1,n*prod)
    }
    currentProd(N,1)
}
```

A lot of loops can be replaced by tail recursion. Usually this involves defining an inner function for the actual recursion, which accepts an accumulator argument in addition to other parameters. Recursive invocations of the inner function pass the current value of the accumulator, or return something when termination condition is met. For tail recursion (or recursion in general), it seems helpful to define the termination conditions at the very begining, then figure out the remaining logic.

1.5 Blocks and Scope

A block is defined by curly braces `{}`

Definitiones inside a block are invisible outside the block. Definitions from outside the block are visible inside, provided they have not been shadowed. A

lot of object oriented functionality can be warngled from scopes and closures. Methods defined within a class constructor have acesss to the parameters of that constructor, even if those parameters are not assigned to fields of the class. For example:

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean = if (x < elem) left.contains(x)
    else if (x > elem) right.contains(x)
    else true
    ...
```

the method `contains` refers to `elem`.

1.6 higher order functions

functions are first class values they can be passed and returned a function that does this is called a higher order function

this can be used to factor out common procedures. For example,

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

defines a function sumFunc, that takes two integers and a function that takes an Int and returns an Int (`Int => Int`) For example, we could sum all squares or cubes between 2 and 5 by calling

```
sumFunc(2,5,square)
sumFunc(2.5.cube)
```

The notation `A => B` is a function type. it is a type that defines a mapping from type A to type B (by a function)

1.7 anonymous functions

strings exist as literals. We can just write "abc", and the compiler knows it to be a string. We don't need to do `def str ="abc"; println(str)` instead `println("abc")` works just fine Same can be done with functions, we don't always need to define a function, we can define anonymous functions as needed. (same as lambda functions in python)

these are defined like this

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

the type of x can be omitted if it can be inferred.

anonymous functions are syntactic sugar: `(x:Int)= x*x` and `def f(x:Int)= x*x`; `f` evaluate to the same.

tail recursive sum

```
def sum(f: Int => Int, a:Int, b:Int) = {
    @scala.annotation.tailrec
    def doSum(total:Int, aval:Int):Int = {
        if (aval > b) total else doSum(total + f(aval),aval+1)
    }
    doSum(0,a)
}
```

Chapter 2

currying

from the scala docs

> "Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments."

For example, consider a function `sum`, which takes a function `f` and returns a function taking two integers as parameters (the bounds).

```
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
}
```

The function sum now takes a function, and returns a function (return type is `(Int,Int)=> Int`). the returned function will take two Ints and return one. In this case, when an initial function (f) is passed into sum, it will return a function that sums the initial function (f) within the supplied bounds.

so, we could do

```
def sumCubes = sum((x:Int) => x*x*x)
def sumSquares = sum((x:Int) => x*x)
// and then invoke them
sumSquares(2,3) // 13
sumCubes(4,7) // 748
```

Alternatively, we could invoke the returned function directly

```
val moose = sum((x:Int)=>x*x) (2,3) //13
```

Again, there is some syntactic sugar for currying. For example

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

is equivalent to the definition of sum above, but without the definition of the inner function. When invoked as `sum(func)` the return type will be a function that takes two integers as parameters: the bounds a and b of the original sum function (the second parameter list). As mentioned above, when a function taking multiple parameter lists is invoked with one or more parameter lists absent, then the return type is a function that takes the missing parameter lists.

2.1 multiple parameter lists

Functions can be specified with multiple sets of parameter lists

```
def f(args_1)...(args_n) = E
```

For $n > 1$, this is equivalent to

```
def f(args_1)...(args_n-1) = {def g(argsn) = E ; g}
// or
def f(args_1)...(args_n-1) = (args_n => E)
```

Carrying this through gives

```
def f = (args1 => (args2 => ...(argsn => E)))
```

named after Haskell Brooks Curry (same guy Haskell language is named after).

As an example, we can write a function that calculates the product of values of a function for points on an interval. We can then define a factorial function in terms of these products.

Our original product and factorial functions look like this

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

def factorial(n:int) = product((x:Int) =>x)(1,n)
```

Now we will use currying to generalise. In this case we will calculate the sum and product of squares (rather than factorials)

```
def CumulativeFunctionOperation(operation: (Int,Int) => Int,
    initVal:Int)(f:Int=>Int)(a:Int,b:Int):Int = {
```

```
    if (a>b) initVal else
        operation(f(a),CumulativeFunctionOperation(operation,initVal)(f)(a+1,b))
}

def sum2:(Int=>Int)=>(Int,Int)=>Int =
    CumulativeFunctionOperation((x:Int,y:Int)=> x+y,0)

sum2(x=>x*x)(2,3) // 13

def prod2:(Int=>Int)=>(Int,Int)=>Int =
    CumulativeFunctionOperation((x:Int,y:Int)=> x*y,1)

prod2(x=>x*x)(2,3) // 36
// factorial
def fact(n:Int) = prod2(x=>x)(1,n)
```

CumulativeFunctionOperation is a form of map-reduce. The operation being apply is a reducer (it takes two inputs and returns a single value). The supplied function is the mapper, for sum2 and prod2 this is the anonymous function `x=>x*x` which computes squares, for fact this is the identity function `x=>x`. The bounds of CFO define the sequence that we are map/reducing. This is pretty neat. These functions are in the worksheet "curying.sc" under week2/misc_worksheets project

2.2 Aside - functional implementation of sets

The week 2 assignment was interesting. A set can be implemented as a function that returns true if it contains the supplied argument. The union of two sets is then the "or" of their charateristic functions, and the intersection the "and". The implementation of a "map" function took me a while to figure out. Instead of thinking of map as `[f(x) for x in set]`, we define the mapped characteristic as a check to see whether the initial set contains (any) element that would map to x. It's a bit backwards.

```
def map(s: Set, f: Int => Int): Set = (x:Int) => exists(s,(y:Int)=> f(y)
    == x)
// exists returns true if any element in the set s satisfies the
    supplied condition, false otherwise.
```

The code for this assignment is under `week2/funsets/src/main/scala/funsets/FunSets.scala`

Chapter 3

Classes

Classes are a type of object that contain methods (member functions) and fields (member data). A class representing a rational number is defined below. Some more examples related to classes are in the rational worksheet (`week2/misc\_worksheets/rationals.sc`).

```
class Rational(x:Int, y:Int) {
  def method(args:Int) = block
}
```

The class definition is the constructor - a new instance of Rational can be created using `val rat =new Rational(2,3)`. Class methods are public by default, private methods should be prefaced with the `private` declaration. Overriding methods should be prefaces with the `override` declaration.

In a class method, the instance of the class can be referenced with `this` (In lectures, the argument to class methods is often "that", so multiplication could be written `new Rational(this.numer*that.numer,this.denom*that.denom)`)

Class fields can be explicitly assigned in the constructor

```
class foo(A: Int) {
  val memberInt = A
}
```

or alternatively the fields can be defined implicitly in the constructor - `class foo(val memberInt: Int){}` . Both of these definitions are equivalent.

Instances of a class are objects. This can be ambiguous, as objects are a thing in Scala (see below)

3.1 Aside - require and assert

Require is a function that can be used to make sure the class being initialised meets some conditions, for example see the rational worksheet (week2/misc_worksheets/rationals.sc).

We ırequire that a class be initialised with a nonzero denominator. require takes a condition and a string as arguments. If the condition is not met, then an `illegalArgumentException` is thrown and the string is printed.

The `assert` function behaves similarly. Assert also takes a condition and a string, but throws an AssertionError when the condition is not met. The intent is that require enforces a precondition on the caller of a function, while assert is used to check the code of the function itself. Require is to make sure the function is called as intended, assert is to check that things are not borked in the internals.

3.2 Alternate constructors

Alternate constructors can be created by defining "this" as a function that takes alternate arguments (by signature), and then invoking the primary constructor appropriately (again, using this) For example, an alternate constructor for Rational could be defined (within Rational) as `def this(x:Int)=this(x,1)`. Then `val moose =new Rational(3)` will be mapped to `Rational(3,1)`.

3.3 Objects

Objects are defined similarly to classes, but using the keyword `object` instead of `class`. Objects can also inherit from (extend) other classes and traits. The difference is that objects are singleton, that is, only a single instance of an object may be created. Objects may be referenced in the scope they are defined in, or imported from another package.

3.3.1 companion objects

If an object has the same name as a class then it is a companion object. In scala, class or static methods do not exist. Things like class methods can be defined in a companion object, class instances can always access the methods of their companion objects

This example is taken from the scala docs `https://docs.scala-lang.org/tour/singleton-objects.html`

```
import scala.math._
case class Circle(radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}
object Circle {
  private def calculateArea(radius: Double): Double = Pi * pow(radius,
      2.0)
}
val circle1 = new Circle(5.0)
circle1.area
```

```
}
```

Objects can be useful for defining factory methods.

3.4 Defining operators/ infix notation

Any method with a parameter can be used like an infix operator. `r add s` is equivalent to `r.add(s)`, `r less s` to `r.less(s)`, etc.

Operators may also be used as identifiers. Valid identifiers in scala can have the following forms

- alphanumeric - starts with letter, followed by letters/numbers
- symbolic - start with operator symbol `+:?~#`, followed by other operator symbols
- underscore - counts as letter
- alphanumeric can end in _ followed by operator symbols

so `x_, x_+*, x1, *, vector_++, counter_=` are all valid identifiers. Things like `+, -, <, >` can just be defined as class methods.

unary operators should be defined using the prefix `unary_`. For example, there is a distinction between the binary version of "-" (subtraction) and the unary "-" (negation). `-Foo` is equivalent to `Foo.unary_-()`

3.5 operator precedence

Details on operator precedence can be found here: `https://docs.scala-lang.org/tour/operators.html`

```
a + b ^? c ?^ d less a ==> b | c
```

can be rewritten (parenthesised) as

```
((a + b) ^? (c ?^ d)) less ((a ==> b) | c)
```

3.6 Classes, hierarchies, and dynamic binding

Classes (and objects) can inherit from existing classes or traits.

```
class baseAdder() {
  def addTwo(x:Int) = x +2
}
class badAdder() extends baseAdder {
  def addThree(x:Int) = x + 4
```

```
  override def addTwo(x: Int) = x +1
}
val moose = new baseAdder()
moose.addTwo(2) // Int = 5
val caribou = new badAdder()
caribou.addTwo(1) // Int = 2
caribou.addThree(4) //Int = 8
```

3.6.1 abstract classes

Abstract classes, like in c++, are classes with definitions that lack implementations. They are prefaced with the keyword `abstract`. If a class is defined as abstract, then it can not be instantiated using `new` (compiler will give an error).

```
abstract class IntSet {
    def incl(x:Int): IntSet
    def contains(x:Int) Boolean
    def union(other: IntSet): IntSet
}
```

3.6.2 traits

Traits are like interfaces in Java. They can declare methods and fields but provide no implementation. Traits take no parameters, and can not be instatiated.

```
trait Pet {
  val name: String
}
```

3.6.3 subtyping

We use the extend keyword to derive a class.

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean = if (x < elem) left.contains(x)
    else if (x > elem) right.contains(x)
    else true
  def incl(x:Int): IntSet = if (x < elem) new
      NonEmpty(elem,left.incl(x), right)
    else if (x > elem) new NonEmpty(elem,left, right.incl(x))
    else this
  override def toString = s"{ ${left} ${elem} ${right} }"
  override def union(other: IntSet) =
      left.union(right).union(other).incl(elem)
}
class Empty extends IntSet {
```

```
  def contains( elem: Int): Boolean = false
  def incl(x:Int): IntSet = new NonEmpty(x,new Empty, new Empty)
  override def toString = "."
}
```

this implies that both empty and nonempty meet all the criteria of intset (implement union, contains and incl), but may have additional functionality. Instances of thee classes can be used whenever IntSets are required. IntSet is the **superclass** of both Empty and NonEmpty (which are **subtypes** of IntSet). If no extend clause is given when defining a class, then the standard java class "object" is assumed.

In java and scala a class can only have one superclass (not true in python or c++). A class may only extend from a single class (or trait). If there are several natural (potential) superclasses, use traits instead.

```
trait Planar {
  def height: Int
  def width: Int
  def surface = width*height
}
class Square extends Shape with Planar with Movable with ...
```

3.6.4 dynamic binding

Scala can use dynamic binding. The type of an object is determined at runtime. For example, a function f may take an instance of class A as an argument, and invoke `A.method()`. There may be a subclass B derived from A, which can override `A.method()`. At runtime, the type of the object passed to f is checked, and either A.method or B.method is invoked. Static binding is where the type of the object is known, for example

```
val theInstance = new A()
theInstance.method()
```

will invoke `A.method()`.