Programming in Scala

Lecture Three

Angelo Corsaro

13 October 2021

Chief Technology Officer, ADLINK Technology Inc.

Table of contents i

1. Reviewing Fold

2. More on Scala Classes

3. Algebraic Data Types

4. Case Classes and Pattern Matching

Reviewing Fold

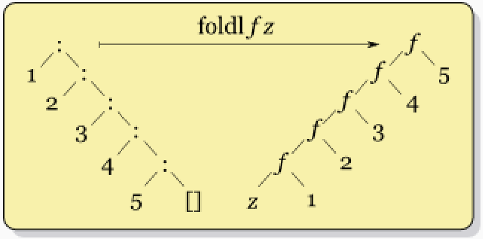
Folding

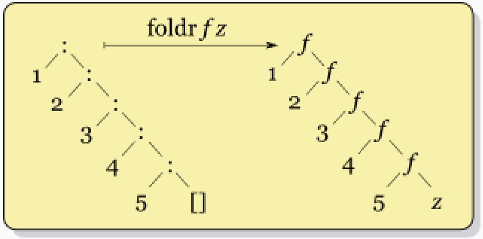
In functional programming, **fold**, refers to a family of *higher-order functions* that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value.

Folds can be regarded as consistently replacing the structural components of a data structure with functions and values.

Two variants of **fold** exist, namely **foldl** and **foldr**.

Visualizing folds





Folding Lists

The functions we just wrote for lists can all be expressed in terms of fold operator.

```scala
def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)

def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)

def reversel(ys: List[Int])= ys.foldLeft(List[Int]())((xs: List[Int], x: Int) => x :: xs)

def reverser(ys: List[Int]) = zs.foldRight(List[Int]())((x: Int, xs: List[Int]) => xs ::: List(x) )
```

foldr and foldl

```scala
def reverse(xs: List[Int]): List[Int] = xs match
        case Nil => Nil
        case y::ys => reverse(ys) ::: List(y)

def
val xs = List(1,2,3,4,5)
val ys = reverse(xs)

val zs = ys.foldLeft(List[Int]())((xs: List[Int], x: Int) => x :: xs)

val ks = zs.foldRight(List[Int]())((x: Int, xs: List[Int]) => xs ::: List(x) )
```

More on Scala Classes

Scala Classes Refresher

In the first lecture we saw the basic mechanism provided by Scala to declare Classes. Back than we had seen *closures* without realizing it, thus let's take a fresh look at some of the examples:

```scala
class Complex(val re: Double, val im: Double):
  override def toString = s"$re+i$im"


class Rational(a: Int, b: Int):
  assert(b != 0)
  val num: Int = a / gcd(a, b)
  val den: Int = b / gcd(a, b)

  def this(n: Int) = this(n, 1)
```

Abstract Classes

In Scala the **abstract** modifier is used to denote an abstract class. In other terms, a class that cannot be instantiated.

An abstract class *may have* abstract members that don't have an implementation (definition).

```scala
abstract class Shape:
   def scale(a: Float): Shape
   // Parameter-less methods: Two takes:
   def show(): Unit

   def area: Float
   def perimeter: Float
```

Parameter-less Methods

Parameter-less methods can be declared with or without parenthesis.

The convention is that parenthesis are used for methods that cause side-effects – like show above.

Parenthesis are not used to methods that compute properties. This way there **uniform access** between attributes and methods.

Extending a Class

The *Shape* abstract type had defined some of the characteristics shared by some geometrical shapes.

A concrete shape can be defined by extending the abstract Shape class as follows:

```scala
class Circle(val radius: Float) extends Shape:
   def scale(a: Float): Shape = new Circle(a * radius)
   def show(): Unit = print(s"Circle($radius)")

   def area: Float = Math.PI.toFloat*radius*radius
   def perimeter: Float = 2*Math.PI.toFloat*radius

```

Extending a Class – Cont

Thanks to the concept of uniform access provided by scala, we can implement parenthesis-less methods with vals. Below is an example:

```scala
class Circle(val radius: Float) extends Shape:
   def scale(a: Float): Shape = new Circle(a * radius)
   def show(): Unit = print(s"Circle($radius)")

   def area: Float = Math.PI.toFloat*radius*radius
   val perimeter: Float = 2*Math.PI.toFloat*radius

```

Methods or Vals?

The **area** and **perimeter** have been defined as parenthesis less methods.

As our shape are immutable, a valid question is if we could define them with **vals** and compute them only once, as opposed to every-time the method is called.

The good news is that Scala supports the concept of *abstract val*, let's see how to leverage them.

Abstract val

Along with abstract methods, an abstract class can also have abstract `vals`.

These are values which are declared but not defined.

If we were to leverage abstract values our Shape class could be declared as follows:

```scala
abstract class Shape:
   def scale(a: Float): Shape
   // Parameter-less methods: Two takes:
   def show(): Unit

   val area: Float
   val perimeter: Float
```

Implementing abstract val

Implementing an abstract val is as simple ad defining it as show in the code example below:

```
class Circle(val radius: Float) extends Shape:
   def scale(a: Float): Shape = new Circle(a * radius)
   def show(): Unit = print(s"Circle($radius)")

   val area: Float = Math.PI.toFloat*radius*radius
   val perimeter: Float = 2*Math.PI.toFloat*radius

```

Observation

Now, as we are using *val* the shape area is computed only once, at the time of initialization. This is an improvement compared from the previous example. Additionally, the client code cannot tell whether we are implementing the area and the perimeter with a val or with a parenthesis less method which is very good.

Yet... We are still spending the time even if the client code is never calling that operation. In an ideal world we would want to compute it once and only if needed... Can we do that?

lazy val

Scala defines the **lazy** modifier to indicate a value that should be computed lazily only when accessed.

If we want o do so, we need to change is the definition of the val in the `Circle` class, and as of **Scala 3** ensure that the members we are overriding are either declared as `def` or as `lazy val`.

```
class Circle(val radius: Float) extends Shape:
   def scale(a: Float): Shape = new Circle(a * radius)
   def show(): Unit = print(s"Circle($radius)")

   lazy val area: Float = Math.PI.toFloat*radius*radius
   lazy val perimeter: Float = 2*Math.PI.toFloat*radius
```

Overriding

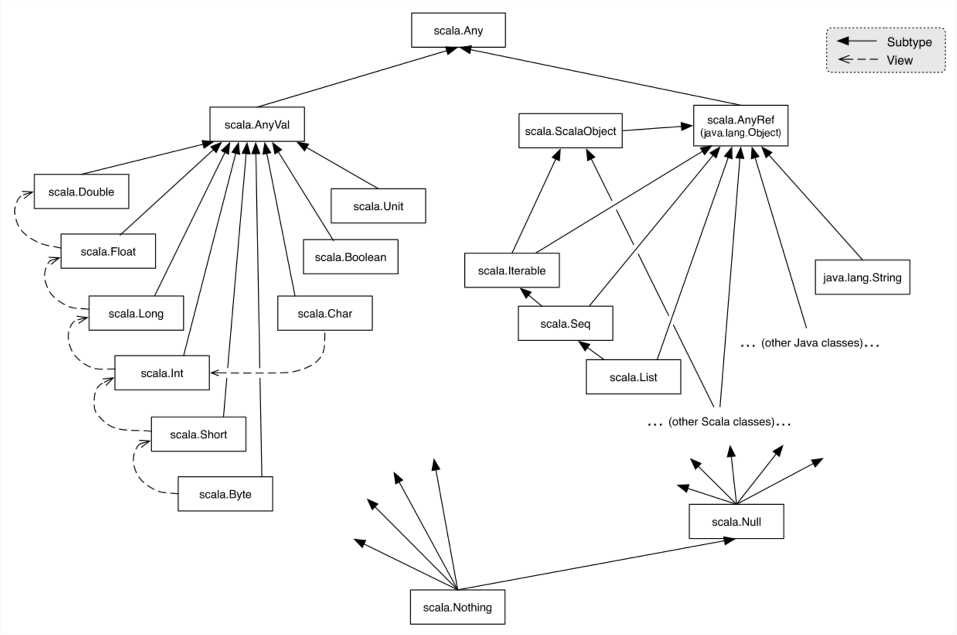
A subclass can override both *methods* as well as *val* defined in the parent class.

The super-class constructor can also be explicitly called as part of the extend declaration as shown in the example below:

```scala
package fr.esiee.fpic.labs:

    class FastCircle(r: Int) extends LazyCircle(r.toFloat):
        override lazy val perimeter: Float = Math.PI.floatValue * (r << 1)
        override def show(): Unit = print(s"O($radius")
```

Scala Type Hierarchy



Algebraic Data Types

Algebraic Data Type

In type theory and commonly in functional programming, an **algebraic data type** is a kind of composite type. In other term a type obtained by composing other types.

Depending on the composition operator we can have **product types** and **sum types**

Product Types

Product types are algebraic data types in which the *algebra* is *product*

A **product type** is defined by the conjunction of two or more types, called fields. The set of all possible values of a product type is the set-theoretic product, i.e., the Cartesian product, of the sets of all possible values of its field types.

Example

```
data Point = Point Double Double
```

In the example above the product type Point is defined by the Cartesian product of $Int \times Int$

Sum Types

The values of a sum type are typically grouped into several classes, called variants.

A value of a variant type is usually created with a quasi-functional entity called a constructor.

Each variant has its own constructor, which takes a specified number of arguments with specified types.

Sum Types Cont.

The set of all possible values of a sum type is the set-theoretic sum, *i.e.*, the disjoint union, of the sets of all possible values of its variants.

Enumerated types are a special case of sum types in which the constructors take no arguments, as exactly one value is defined for each constructor.

Values of algebraic types are analyzed with pattern matching, which identifies a value by its constructor or field names and extracts the data it contains.

Example

```
data Tree = Empty
          | Leaf Int
          | Node Tree Tree deriving (Show)

> let t = Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
> :t t
t :: Tree
```

Case Classes and Pattern Matching

Case Classes

Scala **case classes** provide a way to *mimic* algebraic data types as found in functional programming languages.

The Scala version of the *Tree* type we defined in Haskell is:

```scala
enum ITree:
    case Empty
    case Leaf(v: Int)
    case Node(left: ITree, right: ITree)

    override def toString: String = this match
        case ITree.Empty => ""
        case ITree.Leaf(v) => v.toString
        case ITree.Node(l, r) => "(" + l + ", " + r + ")"

val t = Node(Leaf(1), Node(Leaf(2), Leaf(3)))
```

Case Classes / Cont.

If you noticed the *Tree* algebraic data type was declared through case classes that did not have named attributes. In other term, there were no **val** or **var**.

That does not mean that you cannot declare **val** or **var** as part of a case class. It has more to do with the fact that case classes are worked upon using pattern matching.

Pattern Matching

Scala pattern matching has the following structure:

selector `match` *alternatives*

A pattern match includes a series of *alternatives*, each starting with the keyword **case**.

Each alternative includes a *pattern* and one or more expressions, which are evaluated only if the pattern matches.

Let's see what are the kinds of patterns supported by Scala.

Wildcard Pattern

The *wildcard pattern*, denotated by _, matches anything – thus its appelation.

Example

```
def isEmpty(xs: List[Int]) = xs match {
        case List() => return true
        case _ => return false
}

def isSingleElementList(xs: List[Int]) =  xs match {
        case List(_) => true
        case _ => false
}
```

Constant Pattern

A constant pattern matches only itself. Any literal, such as `18`, `42`, `"Ciao"`, `true` can be used as a constant.

Any **val** or **singleton object** can also be used as a constant.

Example

```
def toString(d: Int) = d match {
        case 0 => "Zero"
        case 1 => "Uno"
        // ...
        case 9 => "Nove"
}

```

Variable Pattern

A variable pattern matches any object, just like a wildcard. But unlike a wildcard, Scala binds the variable to whatever the object is.

You can then use this variable to act on the object further.

Example

```
def checkZero(d: Int) =
        d match {
                0 => "zero"
                v =>  v + " is not zero"
        }
```

Constructor Pattern

A constructor pattern looks like `Node(Leaf(1), Node(Leaf(2), _))` . It consists of a name (`Tree`) and then a number of patterns within parentheses.

Assuming the name designates a case class, such a pattern means to first check that the object is a member of the named case class, and then to check that the constructor parameters of the object match the extra patterns supplied.

Example

```scala
def triangleTree(t: Tree) =
   t match {
      case Node(Leaf(_), Leaf(_)) => True
      case _ => False
   }
```

Please notice that Sequence and Tuple patters are just a special case of constructor patterns.

Exercise

Define the following functions for our `Tree` type:

1. `height` which computes the tree height.
2. `sum` which is the function that computes the sum of all the elements of the tree.
3. `fold` which is a function that applies a generic binary operator to the tree, where the zero is used when for empty nodes.

Typed Pattern

Typed pattern are used to test and cast types.

Example

```scala
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```

Pattern Guards

A pattern guard comes after a pattern and starts with an **if**. The guard can be an arbitrary boolean expression, which typically refers to variables in the pattern.

If a pattern guard is present, the match succeeds only if the guard evaluates to true.

Example

```
// match only positive integers
case n: Int if 0 < n => ...

// match only strings starting with the letter `a'
case s: String if s(0) == 'a' => ...
```

Other use of Patterns

Patterns can be used also in assignments, such as in:

```scala
val (a, b, c) = tripetFun(something)
```

Patterns can also be used in for expressions:

```scala
for ((key, value) <- store)
  print("The key = " + key + "has value = " + value)
```

Exercises

- Define `sum`, `sub`, `mul` and `div` for `ITree`
- Can you identify a common strucutral way of operating on this type?
- Can you abstract it using higher order functions – such as fold?
- Extend `ITree` to hold values also in nodes and not just on leafs
- Implement the basic aritmetic operations for this type (as above)
- Can you make some the functions defined so far Tail-Recursive?