Concepts of Programming Languages

Lecture Notes: Functional Programming

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Scala is a multi-paradigm language that combines functional and object-oriented programming. In this lecture, we discuss pattern matching, list processing functions, and compare the difference between methods and functions in Scala in more depth.

Pattern Matching

When expressing contracts, we often want to describe properties about the resulting data. These can be as simple as comparing values (e.g., $x \geq 0$), but usually require us to describe the expected shape of the result in more detail.

For example, when faced with the following description of how to process a list, a large language model may generate code that we want to safeguard with a contract.

`In Scala, implement a method that takes a list of at least three numbers and returns it with the first three sorted in ascending order.`

Large language model response:

```
def f(numbers: List[Int]) : List[Int] = {
  require(numbers.length >= 3, "The list must contain at least 3
       elements")
  val (firstThree, rest) = numbers.splitAt(3)
  val sortedFirstThree = firstThree.sorted
  sortedFirstThree ++ rest
} ensuring { case xs => ??? }
```

The `???` operator throws a `NotImplementException` and it is our task to describe our requirements more formally with a suitable contract. In this example, we are interested in testing the shape of the list and that the first three elements are in ascending order.

In a traditional imperative programming language style we often access the elements of the list by index like below, or using projections:

```
ensuring { case xs =>
  val x1 = xs(0) // xs.head
  val x2 = xs(1) // xs.tail.head
  val x3 = xs(2) // xs.tail.tail.head
  x1 <= x2 && x2 <= x3
}
```

This style results in lengthy contracts that do not clearly express the intended shape of the result. *Pattern matching* is an alternative way of decomposing a complex data structure into its elements: it emphasizes the shape of the result and simultaneously branches on shapes and binds variables.

```
ensuring {
  case xs =>
    val x1 :: x2 :: x3 :: Nil = xs
    x1 <= x2 && x2 <= x3
}
```

Even more concisely, we can express the pattern immediately in the signature of the function that we pass to `ensuring` as follows:

```
ensuring {
  case x1 :: x2 :: x3 :: Nil =>
    x1 <= x2 && x2 <= x3
}
```

Pattern matching *binds variables* to the components of a data structure according to the specified shape. For example, when computing the sum of two numbers in a tuple of type `(Int, Int)`, a traditional approach may use a conditional expression to check for the tuple being non-null, and then accesses the tuple elements by their index as below:

```
def sum(p: (Int,Int)) : Int =
  if p==null then throw MatchError(p)
  val x = p(0)
  val y = p(1)
  x + y
```

Using pattern matching, we can express the same behavior by listing *cases* for the expected shapes, and in each case, listing the variables that stand for the components of the complex data structure.

```
def sum(p: (Int,Int)) = p match
  case (x,y) => x+y
```

Types are optional:

```
def sum(p: (Int,Int)) = p match
  case (x: Int, y: Int) => x+y
```

Pattern matching *branches* between multiple cases. This is useful when a data structure can take one of multiple shapes (e.g., an empty list vs. a list with at least one element) and we want to produce different results for those shapes. In the following example, we print the first element of a non-empty list; otherwise we print that the list is empty. In an imperative programming approach, we typically use a conditional expression to distinguish the cases. Each branch of the conditional expression then performs a different computation, as below:

```
def printHead(xs: List[Int]) : String =
  if xs == Nil then "List is empty"
  else
    val y: Int = xs.head
    val ys = xs.tail
    s"List is non-empty, head is $y"
```

The same functionality can be achieved with pattern matching:

```
def printHead(xs: List[Int]) : String = xs match
  case Nil              => "List is empty"
  case (y: Int) :: ys => s"List is non-empty, head is $y"
```

The code matches on the shape of list `xs`. If `xs` is the empty list (`case Nil`), then we return the string `"List is empty"`; else (`case (y: Int) :: ys`), we have a list with head `y` and tail `ys` and we print the head. Unused variables (tail `ys` above) can be omitted from the pattern shape using the wildcard operator _ as below:

```
def printHead(xs: List[Int]) = xs match
  case Nil     => "List is empty"
  case y :: _ => s"List is non-empty, head is $y"
```

Patterns can be nested to take any arbitrarily complicated shape:

```
def f (xs: List[(Int,String)]) = xs match
  case Nil                 => "List is empty"
  case _ :: Nil            => "List has one element"
  case _ :: (x,_) :: _ => s"The second int is $x"
```

The code above lists three cases:

- `case Nil` matches the empty list to return string `"List is empty"`
- `case _ :: Nil` matches a list with exactly one element
- `case _ :: (x, _) :: _` matches any list of at least two elements, and it binds the `Int` of the second element to variable `x`

Functions over Lists

We start by inspecting an example to print the elements of a list.

```
def printList (xs:List[Int]) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    println(y) // can format: println("0x%02x".format(y))
    printList (ys)

val xs = List(11,21,31)
printList (xs)
// 11 21 31
```

In the example above, we see two ways of printing the elements of the list (unformatted vs. formatted).

What if we now want to apply some other function to every element of the list? The basic setup of the recursive algorithm wouldn't change, only the specific operation that we apply at each element does. We can describe that abstract idea of processing every element with a recursive algorithm that, in addition to the list being processed, takes a function to be applied to each element as an argument.

```
def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
    case Nil   => ()
    case y::ys =>
```

```
      f (y)
      foreach (ys, f)

val xs = List(11,21,31)
foreach (xs, println)
```

Now it becomes easy to make variations.

```
def printHex (x:Int) = println("0x%02x".format(x))
foreach (xs, printHex)
```

But do we really care about the elements in the list? An additional improvement makes the element type a type parameter of the method.

```
def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
    case Nil   => ()
    case y::ys =>
      f (y)
      foreach (ys, f)
```

Finally, we may not even always want to define the functions that we apply to each element. For this, Scala supports Lambda expressions (anonymous functions):

```
foreach (xs, (x:Int) => println("0x%02x".format(x)))
foreach (xs, println("0x%02x".format(_)))

// also possible
val printHex = (x:Int) => println ("0x%02x".format(x))
foreach (xss, printLength)
```

We do not need to implement `foreach` ourselves, Scala collections provide it!

The examples above use both type and value parameters: type parameters are in square brackets, whereas value parameters are in round brackets. All type parameters must be declared before value parameters. Functions themselves are of function type: for example, X=>Int is the type of a function taking an argument of type X and returns a result of type Int. In Lambda expression, types are often unnecessary if Scala can infer them (type inference is smarter on methods than functions).

List comprehensions. From mathematics, we might be familiar with *set comprehensions* of the form

$$\{(m,n) \mid m \in \{0,\ldots,10\} \wedge n \in \{0,\ldots,10\} \wedge m \leq n\} \enspace .$$

List comprehensions of a similar form are included in many programming languages, such as SETL, Haskell, Scala, and JavaScript.

Scala provides another builtin special syntax to express `foreach` using list comprehensions (named *for-expressions* in Scala).

```
for x <- xs do println ("0x%02x".format(x))
```

We are now inspecting a (less-than-optimal) way of expressing imperative loops with our foreach implementation, by using a variable in scope:

```
def sum (xs:List[Int]) : Int =
    var result = 0
    xs.foreach ((x:Int) => result = result + x)
    result
```

Later, we'll see how *folds* provide a better way of expressing such ideas.

A note on equality: Java uses builtin operators for reference equality, and a method for value equality; Scala has methods for both, the operator symbol method == for value equality and method eq for reference equality.

Transformers. A frequent operation on collections is the modification of elements in the collection. To this end, *transformers* are functions to build a list of modified elements while traversing a collection recursively (unlike above where we only print elements but do not manipulate them).

```
def transform (xs:List[Int]) : List[String] = xs match
    case Nil    => Nil
    case y::ys => ("0x%02x".format (y)) :: transform (ys)
```

A transformer is expected to take one cons cell as input and produce another cons cell as output. Just like foreach, there is a builtin way of applying transformers to collections: map.

Scala again provides special notation to apply transformers in a for-expression:

```
for x <- xs do println ("0x%02x".format(x))
// is compiled to xs.foreach (x => println ("0x%02x".format(x)))

for x <- xs yield "0x%02x".format(x)
// is compiled to xs.map (x => "0x%02x".format(x))
```

Filtering. Often, we want to apply a function only to elements satisfying a certain condition, omitting the remaining elements in the output collection.

```
def filter [X] (xs:List[X], f:X=>Boolean) : List[X] = xs match
  case Nil             => Nil
  case y::ys if f (y) => y :: filter (ys, f)
  case _::ys           =>      filter (ys, f)

val zs = (0 to 7).toList
filter(zs, ((_:Int) % 3 != 0))
```

Again in special for-expression notation:

```
for z <- zs; if z % 3 != 0 yield z
// compiles to zs.filter (z => z % 3 != 0)

for z <- zs; if z % 3 != 0 yield "0x%02x".format(z)
// compiles to zs.filter (z => z % 3 != 0).map (z => "0x%02x".format(z))

for z <- zs; if z % 3 != 0 do println ("0x%02x".format(z))
// compiles to zs.filter (z => z % 3 != 0).foreach (z => println ("0x%02x".format(z)))
```

```
def flatten [X] (xs:List[List[X]]) : List[X] = xs match
  case Nil    => Nil
  case y::ys => y ::: flatten (ys)

val xss = List(List(11,21,31),List(),List(41,51))
```

In the above implementation of flatten, we use operator ::: .

Multiple iterators. For-expressions are quite general and can combine nested iterators in a single pass.

```
val xss = List(List(11,21,31),List(),List(41,51))
for xs <- xss;
     x <- xs yield (x, xs.length)
// result: List[(Int, Int)] = List((11,3), (21,3), (31,3),
   (41,2), (51,2))
```

We can also compute the cross product of independent iterators.

```
val xs = List(11,21,31)
val ys = List("a","b")
for x <- xs;
    y <- ys yield (x, y)
// result: List[(Int, String)] = List((11,a), (11,b), (21,a),
   (21,b), (31,a), (31,b))
```

Currying

We say that functions are *first-class* if they can be

- declared within any scope
- passed as arguments to other functions, and
- returned as results of functions.

Higher-order functions are functions that can take other functions as arguments (e.g., map etc.). Methods that take multiple arguments can be defined in the "usual" way with multiple formal arguments, or in a *curried* way as higher-order definitions that take one argument at a time and produce methods that take the remaining arguments.

```
// paired method
def add1(x:Int, y:Int) = x+y
add1(11, 21)

// curried method
def add2(x:Int)(y:Int) = x+y // takes an Int, returns method of
   type (y: Int): Int
add2(11)(21)

// paired function
```

```
val add3 = (x:Int, y:Int) => x+y

// curried function
val add4 = (x:Int) => (y:Int) => x+y // takes an Int, returns a
    function of type Int=>Int

// can mix notations: method returns a function
def add5(x:Int) = (y:Int) => x+y
```

Both paired and curried notation support partial function application:

```
// paired
val add1p = add1(4, _)
add1p(1) // result 5

// curried
val add4p = add4(4)
add4p(1) // result 5
add4p(2) // result 6
```

Functions and methods in Scala can also be created explicitly by instantiating the appropriate classes:

```
def a (x:Int) = x + 1;
val b = (x:Int) => x + 1;
val c = new Function[Int,Int] { def apply(x:Int) = x + 1 }
val d : PartialFunction[Any, Int]  = { case i: Int => i + 1 }

val fs = List(a,b,c,d)
for f <- fs yield f(4)
```

In summary:

- def defines a method, parameter types explicit
- => defines a function, parameter types inferable
- Functions are objects with method apply (e(args) ===e. apply (args))

Folds

MapReduce is a programming model for processing and generating data sets with a parallel, distributed algorithm. It requires a main process that performs filtering and sorting (the map step) and a summary operation that collects and combines results (the reduce step). Below is an example of counting the number of occurrences of each word in a set of documents using MapReduce in Scala.

```
def map(name: String, contents: String) =
    // name: document name (irrelevant here)
    // contents: document content
    for w <- contents do
        emit (w, 1)

```

```
def reduce(word: String, partialCounts: Iterator) =
    var sum = 0
    for pc <- partialCounts do
        sum = sum + pc
    emit (word, sum)
```

The MapReduce framework is a distributed implementation of a recursive algorithm. For example, we can sum up the elements of a list of integers as below.

```
def sum (xs:List[Int], z:Int = 0) : Int =  xs match
  case Nil    => z
  case y::ys  => sum (ys, z + y)

val xs = List(11,21,31)
sum (xs)
```

The algorithm takes a list of (remaining) elements and a partial result, and aggregates the partial result with the result of processing a single element, until no more elements are left to process. It computes the sum in a *forward fashion*, passing the aggregated result to the next step. In a *backwards fashion* as below, the aggregation is postponed until all elements are processed.

```
def sum (xs:List[Int], z:Int = 0) : Int =  xs match
    case Nil    => z
    case y::ys  => y + sum (ys, z)

val xs = List(11,21,31)
sum (xs)
```

Both have in common that they are using an accumulator. Scala has builtin fold operations that allow us to traverse collections while accumulating results. Operation foldLeft performs accumulation in a forward fashion, foldRight in backwards fashion. foldLeft is *tail recursive*, which means that the base case is the first element, the recursive call is on the tail, and the accumulator is applied to the head and the accumulated result. foldRight is *recursive into an argument*, which means that the base case is the last element, the recursive call is on the tail, and the accumulator is applied to the head and the result of the recursion. Folds are a universal concept that can be used to compute many different functions on lists, such as summing up elements, appending a list to another list, flattening, or reversing.