Concepts of Programming Languages
Lecture Notes: Functional Programming

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Functions over Lists

Last lecture, we looked at typed languages and Scala as a multi-paradigm language that combines functional and object-oriented programming. In this lecture, we discuss functional programming in more depth, again using Scala as an example. We start by inspecting an example to print the elements of a list.

```
1  def printList (xs:List[Int]) : Unit = xs match
2    case Nil    => ()
3    case y::ys =>
4      println(y) // can format: println("0x%02x".format(y))
5      printList (ys)
6
7  val xs = List(11,21,31)
8  printList (xs)
9  // 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.

```
1  def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
2      case Nil    => ()
3      case y::ys =>
4          f (y)
5          foreach (ys, f)
6
7  val xs = List(11,21,31)
8  foreach (xs, println)
```

Now it becomes easy to make variations.

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

// also possible
val printHex = (x:Int) => println ("ox%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

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

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 ("ox%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 => ("ox%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 ("ox%02x".format(x))
// is compiled to xs.foreach (x => println ("ox%02x".format(x)))

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

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

```
3     case y::ys => sum (ys, z + y)
4
5   val xs = List(11,21,31)
6   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.

```
1   def sum (xs:List[Int], z:Int = 0) : Int =  xs match
2       case Nil    => z
3       case y::ys => y + sum (ys, z)
4
5   val xs = List(11,21,31)
6   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.

Option Types

Option types are a principled approach to missing data. Option[T] resembles List[T] with a length of at most 1. Java, for a long time, emphasized programming with exceptions in order to report anything that deviates from a successful result.

```
1   // the Java way
2   def getDirs1 (dirName : String) : List[java.io.File] =
3       val dir = new java.io.File (dirName)
4       val xs = dir.listFiles
5       if xs == null then throw new java.io.FileNotFoundException
6       xs.nn.toList.map (_.nn).filter (_.isDirectory)
```

Exceptions, however, are best used to report exceptional circumstances (such as a broken network connection); missing data is quite an expected result. Programming with optionals allows us to document missing data and allows our users to appropriately react to it (as opposed to the Java habit of just passing on exceptions).

```
1   def getDirs2 (dirName : String) : Option[List[java.io.File]] =
2       val dir = new java.io.File (dirName)
3       val xs = dir.listFiles
4       if xs == null then return None
5       Some(xs.nn.toList.map (_.nn).filter (_.isDirectory))
```

With optionals, clients no longer suffer from the convolutional way of providing a result from multiple execution traces. An option is a type that may have some result or nothing. Scala knows several option types:

- None is the empty option
- Nil is the empty list
- null is a reference to nothing

Unit is not an option type, it only has a single value (always has nothing). Even though Scala has null, we often pretend it does not exist. More recent languages, such as Swift and Kotlin, identify None and null; these languages distinguish nullable and non-nullable types. Java also includes an optional, but its intended use is narrowed to library methods whose return types needed a clear way of communicating the absence of a result and null is overwhelmingly likely to cause errors.