Class 14 - Higher-Order Functions and Collections in Scala

We have seen that we can often identify common patterns in functions on data structures and implement them in generic higher-order functions. We can then conveniently reuse these generic functions, reducing the amount of code we have to write. Today, we will look at the most general patterns for performing operations on collections, namely fold operations.

Folding Lists

As a motivating example, consider the following function, which computes the sum of the values stored in a list of integers

```scala
def sum(l: List[Int]): Int = {
  case Nil => 0
  case h :: t => h + sum(t)
}
```

Consider a list `l` of `n` integer values:

```
d1 :: d2 :: ... :: dn :: Nil
```

Then unrolling the recursion of `sum` on `l` yields the following computation

```
d1 + (d2 + ... (dn + 0)...)  
```

That is, in the \(i\)th recursive call, we add the current head \(d_i\) to the sum of the values in the current tail, where we consider the sum of an empty list `Nil` to be 0. If we represent this computation as a tree, we obtain

```
         +
        / \
       +   +
      /     /
     d1    d2
     /     /   +
    /     /     /
   d2   +   d3
   /     /   /
  d3   +   d4
  /     /
 d4   +
    /
 d5
```

```
0
```
We can now generalize from the specific computation performed by the represented expression. That is, in the general case we have a list of values of type A instead of Int. Then, instead of the specific initial value 0 for the empty list, we are given an initial value $z$ of some type $B$. Finally, instead of adding the current head to the sum of the current tail of the list, we apply a generic operation $\text{op}$ in each step. The operation $\text{op}$ takes the current value $d_i$, which is of type $A$, and the result of the computation on the tail, which is of type $B$, and returns again a value of type $B$. The resulting expanded recursive computation is then represented by the following tree:

$$
\begin{align*}
\text{op} & \quad \text{op} \\
\text{op} & \quad \text{op} \\
\text{op} & \quad \text{op} \\
\text{op} & \quad \text{op} \\
\end{align*}
$$

We refer to this type of computation as a fold of the list because the list is traversed and recursively folded into a single value. Note that the tree is leaning towards the right. We therefore refer to this type of fold operation as a fold-right. That is, the recursive computation is performed in right-to-left order of the values stored in the list.

The following higher-order function implements the fold-right operation:

```scala
def foldRight[A,B](l: List[A])(z: B)(op: (A, B) => B): B = 
  l match {
    case Nil => z
    case h :: t => op(h, foldRight(t)(z)(op))
  }
```

We can now redefine $\text{sum}$ in terms of foldRight:

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

Many of the other functions that we have seen before perform fold-right operations on lists. In particular, we can define $\text{concat}$ using foldRight as follows:

```scala
def concat[A](l1: List[A], l2: List[A]): List[A] = 
  foldRight(l1)(l2)(_ :: _)
```

Even the higher-order function $\text{map}$ is just a special case of a fold-right:

```scala
def map[A, B](l: List[A])(op: A => B): List[B] = 
  foldRight(l)((Nil: List[B]))((h, l) => op(h) :: l)
```

Note that due to limitations of Scala’s type inference algorithm, we have to manually annotate the type $\text{List}[B]$ of the empty list constructor $\text{Nil}$ that we use to build the resulting list of the map operation.

All the above operations on lists have in common that they combine the elements in list and the result of the recursive computation in right-to-left order.
We can also consider fold operations that perform the computation in reverse order:

\[ \text{op}(d_n, \text{op}(d_{n-1}, \ldots, \text{op}(d_2, \text{op}(d_1, z)) \ldots)) \]

The corresponding computation tree then looks as follows:

Note that the tree is now leaning towards the left and the elements are combined in left-to-right order. We therefore refer to this type of computation as a \textit{fold-left}. The following function implements the generic fold-left operation:

```scala
def foldLeft[A,B](xs: List[A])(z: B)(op: (B, A) => B): B = xs match {
  case Nil => z
  case h :: t => foldLeft(t)(op(z, h))(op)
}
```

Since addition is associative and commutative, we can alternatively define \textit{sum} using \textit{foldLeft} instead of \textit{foldRight}:

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

In fact, this definition of \textit{sum} is more efficient than our previous implementations because \textit{foldLeft} is tail-recursive, whereas our implementation of \textit{foldRight} is not. In general, only one of the two types of folds can be used to implement a specific operation on lists. For example, we can express \textit{reverse} in terms of a fold-left as follows:

```scala
def reverse[A](xs: List[A]): List[A] = foldLeft(xs)((Nil: List[A]))((ys, x) => x :: ys)
```

If we replaced \textit{foldLeft} by \textit{foldRight} in this definition, we would not obtain the correct result. The resulting list would be structurally identical to the input list.

### Scala’s Collection Classes

Since higher-order functions on collections are so incredibly useful for writing concise code, the data structures in the Scala standard API already provide a large number of these functions. These functions are implemented as methods of the corresponding collection classes. For example Scala’s \textit{List} class already provides methods \textit{foldLeft}, \textit{foldRight}, \textit{map}, etc.

As with any programming language, I advise you to study the Scala standard API carefully so that you get an overview of the provided functionality and so that you do not “reinvent the wheel”.
To get a glimpse of the expressive power of the functions implemented in the collection classes, consider the following code snippet. The code defines a list of integers and a list of strings and then folds the two lists into a single string. This string consists of a comma separated sequence of strings, where each string is a pair of elements from the two lists concatenated together using the colon symbol:

```scala
scala> val l1 = List(1, 2, 3)
l1: List[Int] = List(1, 2, 3)

scala> val l2 = List("a", "b", "c")
l2: List[String] = List(a, b, c)

scala> ((l1,l2) zipped) map (_ + ":" + _) reduce (_ + "," + _)
res0: String = 1:a, 2:b, 3:c
```

It is instructive to re-implement this code snippet in a language such as Java to appreciate how much more comprehensive the functional implementation is.