Class 13 - Higher-Order Functions and Collections in Scala

Higher-order functions provide a powerful mechanism for abstracting over common computation patterns in programs. This mechanism is particularly useful for designing libraries with rich interfaces that support callbacks to client code. We will study these mechanisms using the example of Scala’s collection libraries.

Higher-Order Functions in Scala

As a simple motivating example, suppose that we want to write a Scala function \(\text{sumInts}\) that takes the bounds \(a\) and \(b\) of a (half-open) interval of integer numbers \([a,b)\) and computes the sum of the values in that interval. For example, \(\text{sumInts}(1, 4)\) should yield 6. The following recursive implementation does what we want:

```scala
def sumInts(a: Int, b: Int): Int = {
  require(a <= b)
  if (a < b) a + sumInts(a + 1, b) else 0
}
```

Now, consider the following function \(\text{sumSqr}\) that computes the sum of the squares of the numbers in an interval \([a,b)\):

```scala
def sumSqr(a: Int, b: Int): Int = {
  require(a <= b)
  if (a < b) a * a + sumSqr(a + 1, b) else 0
}
```

The functions \(\text{sumInts}\) and \(\text{sumSqr}\) are almost identical. They only differ in the summand that is added in each recursive call. In the case of \(\text{sumInts}\) it is \(a\), and in the case of \(\text{sumSqr}\), it is \(a \times a\). We can write a higher-order function \(\text{sum}\) that abstracts from these differences, by taking a function \(f\) as additional argument, which captures the computation performed in the summand:

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

The function type Int => Int of the parameter f indicates that f is a function that takes a value of type Int and maps it again to an Int.

We can now define the function sumSqs by first defining a function square that squares an integer number, and then applying sum to square:

def square(i: Int) = i * i
def sumSqs(a: Int, b: Int) = sum(square, a, b)

In order to make the use of higher-order functions more convenient, Scala supports writing anonymous functions (aka function literals), similar to JavaScript. In Scala, anonymous functions take the general form:

(x1: T1, ..., xn: Tn) => body

where the xi are the parameters of the function, the Ti are the associated types, and body is the body of the function. We can thus define the functions sumInts and sumSqs using anonymous functions as follows:

def sumInts(a: Int, b: Int) = sum((i: Int) => i, a, b)
def sumSqs(a: Int, b: Int) = sum((i: Int) => i * i, a, b)

Tail Recursion

If we apply the function sumInts to larger intervals we observe the following:

scala> sumInts(1, 10000)
java.lang.StackOverflowError
...

The problem is that a call to a function requires the Scala runtime environment to allocate stack space that stores the arguments of the call in memory and any intermediate results obtained during the evaluation of the function body. In terms of our environment-based interpreter, the allocation of stack space corresponds to the extension of the current environment with the bindings for the formal parameters of a function to the actual arguments of the call. In particular, we have to remember the old environment until after the call has been evaluated so that we can continue to use it for evaluating the context of the call expression. Similarly, the stack space that is needed for a function call in Scala remains allocated until the call returns. For recursive procedures, the required stack space thus grows linearly with the recursion depth, which in the case of sum is given by the size of the interval b - a. Since the Scala runtime environment only reserves a relatively small amount of memory for the call stack, a call to sum for large intervals runs out of stack space. This is signaled by a StackOverflowError exception.
Can we implement the function \texttt{sum} so that it only requires constant space? To this end, consider the following \textit{imperative} implementation of \texttt{sum}, which uses a \texttt{while} loop and mutable variables to perform the summation:

\begin{verbatim}
def sum(f: Int => Int, a: Int, b: Int): Int = {
  require(a <= b)
  var acc = 0
  var i = a
  while (i < b) {
    acc = f(i) + acc
    i = i + 1
  }
  acc
}
\end{verbatim}

This implementation requires only constant space, since it involves only a single function call and the execution of a loop iteration for the summation does not allocate memory that persists across iterations. Unfortunately, this implementation uses mutable variables, which makes the implementation more difficult to reason about. However, we can turn the imperative \texttt{while} loop into a recursive functional procedure by hoisting the loop counter \texttt{i} and accumulator \texttt{acc} to function parameters:

\begin{verbatim}
def sum(f: Int => Int, a: Int, b: Int): Int = {
  require(a <= b)
  def loop(acc: Int, i: Int): Int = {
    if (i < b) loop(f(i) + acc, i + 1) else acc
  }
  loop(a, 0)
}
\end{verbatim}

Note how the function \texttt{loop} closely mimics the \texttt{while} loop in the imperative implementation without relying on mutable variables.

The function \texttt{loop} has an important property: the recursive call to \texttt{loop} in the \texttt{then} branch of the conditional is the final computation that is performed before the function returns. That is, in the recursive case, the function directly returns the result of the recursive call. We refer to functions in which all recursive calls are of this form as \textit{tail-recursive} functions. Contrast the new implementation of \texttt{sum} with our original implementation, which added \texttt{f(a)} to the result of the recursive call and was therefore not tail-recursive.

Since a tail-recursive call is the final computation that is performed when evaluating the body of a tail-recursive function, the stack space that is allocated for the current call can be reused by the recursive call. In particular, the memory that is needed to store the arguments of the current call can be reused to store the arguments of the recursive call. By reusing the current stack space, we effectively turn the recursive procedure back into an imperative loop. We refer to this optimization as \textit{tail call elimination}. Like most compilers for functional languages, the Scala compiler automatically eliminates tail calls. Thus, tail-
recursive functions are guaranteed to execute in constant stack space. With tail
call elimination we get the best of both worlds: we obtain the efficiency of an
imperative implementation and the simplicity of a functional implementation.

Rerunning `sumInts` with the new implementation of `sum` yields:

```scala
scala> sumInts(1, 10000)
res0: Int = 49995000
```

**Curried Functions in Scala**

Reconsider our definition of `sumInts` and `sumSqrs` in terms of `sum`:

```scala
def sumInts(a: Int, b: Int) = sum((i: Int) => i, a, b)
def sumSqrs(a: Int, b: Int) = sum((i: Int) => i * i, a, b)
```

One annoyance with these definitions is that we have to redefine the parameters
`a` and `b` which are simply passed to `sum`. We can avoid this by redefining `sum`
as a curried function that first takes the function `f` applied to the values in the
interval and then returns a function that takes the bounds of the interval `a` and `b`.

There are various ways to define curried functions in Scala. One way is to
define the nested function explicitly by name using a nested `def` declaration
and then returning that function:

```scala
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumHelp(a: Int, b: Int): Int = {
    require(a <= b)
    def loop(i: Int, acc: Int): Int = {
      if (i < b) loop(i+1, f(i) + acc) else acc
    }
    loop(a, 0)
  }
  sumHelp
}
```

Using the curried version of `sum` the definition of `sumInts` and `sumSqrs` can
be simplified:

```scala
def sumInts = sum(i => i)
def sumSqrs = sum(i => i * i)
```

Note that when we apply curried higher-order functions to anonymous func-
tions, then the compiler can often infer the parameter types. This simplifies the
definitions even further:

```scala
def sumInts = sum(i => i)
def sumSqrs = sum(i => i * i)
```

In our curried version of `sum`, the function `sumHelp` is not recursive and
is directly returned after being declared. We can thus simplify the definition of
`sum` further by turning `sumHelp` into an anonymous function:
def sum(f: Int => Int): (Int, Int) => Int =
  (a: Int, b: Int) => {
    require(a <= b)
    def loop(i: Int, acc: Int): Int = {
      if (i < b) loop(i+1, f(i) + acc) else acc
    }
    loop(a, 0)
  }

Since curried functions are so common in functional programs, the Scala language provides a special syntax for them. Instead of nesting the function declarations, we can write a curried function by providing the parameters of the nested function in a separate parameter list:

```scala
def sum(f: Int => Int)(a: Int, b: Int): Int = {
  require(a <= b)
  def loop(i: Int, acc: Int): Int = {
    if (i < b) loop(i+1, f(i) + acc) else acc
  }
  loop(a, 0)
}
```

If we partially apply a curried function written in this form, we have to tell the compiler explicitly by terminating the partial application with the underscore symbol _ The definitions of `sumInts` and `sumSqs` thus look as follows in this case:

```scala
def sumInts = sum(i => i)_
def sumSqs = sum(i => i * i)_{
```

### Functional Lists

An important use case of higher-order functions is to realize callbacks to client code from within library functions. We discuss this scenario using the specific example of the class `List` in the Scala standard library.

Functional lists are one of the most important data structures in functional programming. A functional list is an immutable sequence of data values of some common element type, e.g., a sequence of integer numbers or a sequence of strings. Since the data structure is immutable, it enables a shared representation in memory of lists which have common sublists. This yields very space-efficient, high-level implementations of algorithms if the data structure is used correctly.

We can define lists of integers recursively using an algebraic data type as follows:

```scala
sealed abstract class IntList
case object Nil extends IntList
case class Cons(head: Int, tail: IntList) extends IntList
```
That is, a list is either empty, denoted by Nil, or a cons cell consisting of an integer head and the rest of the list tail.

The generic class List[A] in the Scala standard library generalizes this data structure to lists over an arbitrary element type A. The empty list is also denoted by Nil and a cons cell is denoted head :: tail. We can thus construct lists as follows:

```scala
scala> val l1 = 1 :: (4 :: (2 :: Nil))
l1: List[Int] = List(1, 4, 2)

scala> val l2 = "apple" :: ("banana" :: Nil)
l2: List[String] = List(apple, banana)
```

Note that the cons operator :: is right-associative, so the parenthesis in the above example can be omitted:

```scala
scala> val l1 = 1 :: 4 :: 2 :: Nil
l1: List[Int] = List(1, 4, 2)
```

As expected, we can use pattern matching to deconstruct lists into their components:

```scala
scala> val h :: t = l1
h: Int = 1
t: List[Int] = List(4, 2)
```

First-Order Functions on Lists

Using pattern matching and structural recursion, we can define simple functions that manipulate lists. For example, the function concat takes two lists l1 and l2, and returns the list obtained from appending l2 to l1:

```scala
def concat[A](l1: List[A], l2: List[A]): List[A] =
  l1 match {
    case Nil => l2
    case h :: t => h :: concat(t, l2)
  }
```

Note that concat is a generic function that is parameterized by the type A of the elements stored in the lists. Here is an example of how to use concat:

```scala
scala> concat(List(3,4,1), List(2, 6))
res0: List[Int] = List(3,4,1,2,6)
```
Using `concat` we can define another function that reverses a given list:

```scala
def reverse[A](l: List[A]): List[A] = 
  l match {
    case Nil => Nil
    case h :: t => concat(reverse(t), List(h))
  }
```

```
scala> reverse(List(3,4,1,2))
res0: List[Int] = List(2,1,4,3)
```

The above implementation of `reverse` is not very efficient. Its running time is quadratic in the length of the list `xs`. Moreover, the function is not tail-recursive and hence requires linear space in the length of `xs`. We can rewrite the function so that it is tail-recursive and runs in linear time and constant space:

```scala
def reverse[A](xs: List[A]): List[A] = {
  def rev(l: List[A], acc: List[A]): List[A] = l match {
    case Nil => acc
    case h :: t => rev(t, h :: acc)
  }
  rev(xs, Nil)
}
```

### Higher-Order Functions on Lists

From the above examples we can see that functions operating on lists follow a common pattern: they traverse the list, decomposing it into its elements, and then apply some operation to each of the elements. We can extract this common pattern and implement them in more general higher-order functions that abstract from the specific operations being performed.

A particularly common operation on lists is to traverse a list and applying some function to each element, obtaining a new list. For example, suppose we have a list of `Double` values which we want to scale by a given factor to obtain a list of scaled values. The following function implements this operation:

```scala
def scale(factor: Double, l: List[Double]): List[Double] = 
  l match {
    case Nil => Nil
    case h :: t => factor * h :: scale(factor, t)
  }
```

A similar operation is implemented by the following function, which takes a list of integers and increments each element to obtain a new list:

```scala
def incr(l: List[Int]): List[Int] = 
  l match {
    case Nil => Nil
```

7
The type of operation that is performed by `scale` and `incr` is called a *map*. We can implement the map operation as a higher-order function that abstracts from the concrete operation that is applied to each element in the list:

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

The function `map` is parametric in both the element type `A` of the input list, as well as the element type `B` of the output list. That is, a map operation transforms a `List[A]` into a `List[B]` list by applying an operation `op: A => B` to each element in the input list. Note that the order of the elements in the input list is preserved.

We can now redefine `scale` and `incr` as instances of `map`:

```scala
def scale(factor: Double, l: List[Double]) = 
  map(l)(x => factor * x)
def incr(l: List[Int]) = map(l)(_ + 1)
```

Note that Scala provides a syntactic short form for defining anonymous functions by replacing variables in expressions by underscores. This is often useful to obtain succinct code when using higher-order functions. For example, the Scala compiler will automatically expand the following code to the above definitions of `scale` and `incr`:

```scala
def scale(factor: Double, l: List[Double]) = 
  map(l)(factor * _)
def incr(l: List[Int]) = map(l)(_ + 1)
```