Chapter 24
Partially Applied Functions and Currying

24.1 Introduction

This chapter looks at two ways in which functions (and in fact methods) in Scala can comprise components of reuse within a software system. These two approaches are partial application of functions and Currying. The two approaches represent variations on a theme. In both cases they allow a function with one or more parameters to have one or more of those parameters bound to a specific value to create a new function with one or more fewer variables.

At an abstract level, consider having a function that takes two parameters. These two parameters, x and y, are used within the function body with the multiplying operator in the form x * y. For example, we might have:

$$\text{operation} = (\text{x}, \text{y}) = > \text{x} * \text{y}$$

This operation might then be used as follows:

$$\text{total} = \text{operation}(2, 5)$$

which would result in 5 being multiplied by 2 to give 10. Or it could be used:

$$\text{total} = \text{operation}(10, 5)$$

which would result in 5 being multiplied by 10 to give 50.

If we needed to double a number we could thus reuse operation many times, for example

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_24

$$\text{operation}(2, 5)$$
$$\text{operation}(2, 10)$$
$$\text{operation}(2, 6)$$
$$\text{operation}(2, 151)$$

All of the above would double the second number. However, we have had to remember to provide the 2 so that the number can be doubled. However, the number 2 has not changed between any of the invocations of operation. What if we fixed the first parameter to always be 2, this would mean that we could create a new function that apparently only takes one parameter (the number to double). For example, let us say we could write something like:

$$\text{double} = \text{operation}(2, *)$$

Such that we could now write:

$$\text{double}(5)$$
$$\text{double}(151)$$

In essence double is an alias for operation, but an alias that provides the value 2 for the first parameter and leaves the second parameter to be filled in by the future invocation of the double function.

This is essentially what Scala provides, although it has two mechanisms through which this style of parameter binding can be implemented. The two approaches are Partially Applied functions and Currying.

Note that what is said in the rest of this chapter for functions is also true for methods. The only difference is that methods are always part of the class or object in which they are defined. However it is possible to partially apply and curry both functions and methods.

24.2 Partially Applied Functions

A Partially Applied function in Scala is a function where one or more of its parameters have been *applied or bound* to a value, resulting in the creation of a new function one fewer parameters than the original. For example, let us create a function operation based on the example presented above in Scala.

```
val operation = (x: Int, y: Int) => x * y
```

This operation does exactly what the previous example did. It takes two parameters and multiples them together. We can thus invoke it in the normal manner:

```
println(operation(2, 5))
```

The result of executing this statement is:

```
10
```

However, we can bind the first parameter to 2 so that it will always double the second parameter and store the resulting function reference into a property double. For example,

```
val doubleIt = operation(2, _: Int)
```

Double now references a function that takes one parameter an Int. Note that the '_' underbar indicates that this is a placeholder for future values and that the type that will be used with this placeholder is Int. This allows for overloading of functions and distinguishing between such overloading (overloading relates to functions with the same names but different parameter types).

To invoke this function we merely need to provide the parameter value to be double, for example

```
println(doubleIt(5))
```

The result of executing this line is once again:

```
10
```

It is also possible to have more than one parameter bound and more than one parameter left unbound. These parameters can be intermixed with any parameter in any position being bound or unbound as required.

The following listing shows a simple summation function that takes three integers and adds them together. It also shows a situation where the first and the third parameters are bound and the resulting function assigned to the val partialSum. The function referenced by partialSum takes a single Int and adds the value passed into the values 1 and 3.

```
var sum = (a: Int, b: Int, c: Int) => a + b + c
println(sum(1, 2, 3))
val partialSum = sum(1, _: Int, 3)
println(partialSum(2))
```

The result of executing this listing is:

```
6
6
```

Partially Applied functions are very useful for creating new functions from existing functions. The other approach is Currying which is described below.

24.3 Currying

24.3.1 Introduction to Currying

An alternative approach to Partially Applied functions is to use a technique/pattern known as Currying in the Scala/Functional world. The name Currying may seem obscure but the technique is named after Haskell Curry (for whom the Haskell programming language is named).

Currying allows new *language* structures to be created. The language features can resemble normal function invocations or can resemble language constructs. This allows the constructs developed to be presented to client code as if they are merely an extension to Scala (which in many ways they are). For example, structures such as:

```
transaction {
…
}
```

can be created where transaction is a function that has been curried such that it is linked to an appropriated database, etc.

A curried function is a function that is applied to *multiple* argument lists (in contrast a Partially Applied function has one argument list). Note that functions can have one or more parameter lists.

24.3.2 Defining Multiple Parameter List Functions

All the function examples we have seen so far have used a single argument list with multiple parameters. For example this function takes two parameters (x and y) and multiplies them together. It is used with `println` in this example to print out the result of multiplying 2 and 3:

```
class Basic {
  def sum(x: Int, y: Int) = x + y
}

object BasicTest extends App {
  val test = new Basic
  println(test.sum(2, 3))
}
```

In Scala we can rewrite this function as a multiple argument function, on which case each argument list takes a single argument. The arguments can still be used in the body of the function (and thus we can still multiple x and y together):

```
package com.jjh.scala.curry

class CurryTest {
  def sum(x: Int)(y: Int) = x + y
}

object CurryTest extends App {
  val test = new CurryTest
  println(test.sum(2)(3))
}
```

Both of these functions return the value 6, and both allow you to pass in two parameters to the function. However, in effect the second version results in two function invocations back to back. It is a bit like chaining two functions together. The first function invocation takes a single Int parameter x and returns a function value for the second function. The second function takes the Int parameter Y and applies it to the first functions result.

24.3.3 Using Curried Functions

Why is this useful? It is because we can provide a value to use for the *first* function before we wish to provide a value for the *second* function. As the result of providing the value for the first function is to return a function value to use with the second, we can essentially *store* that functions and its subsequent invocation for later use. For example, given the following function definition in the class CurryTest:

```
package com.jjh.scala.curry

class CurryTest {
  def sum(x: Int)(y: Int) = x + y
}
```

This function has multiple argument lists and could be called directly as shown above.

However we could *curry* the function such that we provide the first argument but not the second. Note that in comparison with Partially Applied Functions where any argument can be applied, when Currying it is only the left-hand side arguments that can be applied. Thus in this case we could not supply the second argument rather than the first. Also note that because of this Currying does not require that you specify the types of the omitted parameters.

The approach used with multiple argument lists is to provide the arguments from the left and to use '_' to indicate that the remainder of the function invocation has been left out:

```
val plusOne = test.sum(1)_
println(plusOne(10))
```

The plusOne function is now a curried function that takes the y argument and adds it to the value 1 which was provided for the x argument when the plusOne val was defined. Note that as far as the client code of plusOne is concerned this is just a normal single parameter function.

The definition of plusOne used above is actually a shorthand form for the definition of the curried function. All of the following are also variations on the definition of the curried function:

```
import test._
def plusTwoLL(y:Int):Int = sum(2)(y: Int)
def plusTwoL(y:Int) = sum(2)(y: Int)
def plusTwo(y: Int) = sum(2)(y)
def plusTwoS(y: Int) = sum(2)_
def plusTwoRS = sum(2)_
```

The explanation for of the variations on the plusTwo function is provided below:

- The function definition plusTwoLL is the *longest longhand* form of the function where we specify the parameter and the return type of the newly created function as well as the binding of the first parameter to 2.
- The plusTwoL function is a *longhand* form which infers the return type.
- The plusTwo method allows the type of the second parameter in the second parameter list to be inferred (this is allowed in Currying—specifying the type is optional here; this was not the case for Partially Applied functions).
- The plusTwoS definition is a *shorthand* form where the second parameter list is replaced by the underbar ('_') placeholder.
- The plusTwoRS is the *real shorthand* form where the parameter type of the newly created function is also inferred.

24.3.4 Building Domain-Specific Languages

If you need to create a framework to be used by Scala programmers, but want to provide a flexible way to configure that framework and allow the developers to view the end results as merely part of the Scala environment, then Currying has many advantages.

For example, the use of Currying allows you to build new language structures based on multiple argument lists. For example you can then exploit the '{}' syntax for a single parameter. If this parameter is itself a function, the end result is a construct that looks like a language feature:

```
write (file) {
  writer => writer.println(new Date)
}
```

The above is a function *write* that has two argument lists: one that takes a file and one that takes a function. The round bracket syntax could be used for both parameters. However, by using the '{}' syntax for the last parameter this looks more like a language structure! This is an alternative syntax for parameters that can be used for the last set of parameters lists.

We can create the write example presented earlier by implementing a function or a method with multiple parameters. In this example, we are using a method write:

```
object Printer {
  def write(file: File)(op: PrintWriter => Unit) {
    val writer = new PrintWriter(file)
    try {
      op(writer)
    } finally {
      writer.close
    }
  }
}
```

This method has two parameter lists defined: one parameter list that takes a file and one parameter list that takes a function. This function takes a PrintWriter and returns Unit.

This could then be used as follows:

```
import Printer._
val file = new File("data.txt")
write (file) (writer => writer.println(new Date))
```

Note that the second parameter lists contain a function definition that takes a parameter of type PrintWriter and calls the method println on that print writing. The result is that the current Date is written into whatever writer is passed to this function. Inside the write method, the function passed a PrintWriter that is bound to the file passed in within the first parameter list.

Of course we could also exploit the '{}' syntax which means that we could also write:

```
val file = new File("data.txt")
    import Printer._

  val file = new File("data.txt")
  write (file) {
    writer => writer.println(new Date)
  }
```

which makes the write function appear more of a language construct than a straight function or method call.

However, we could take this further. If we are creating a logging system in which we always want client code to write to the same log file we could use Currying to create a fileWriter function that is guaranteed to write the file we wish to specify:

```
import Printer._

val fileWriter = write(file)_
fileWriter {
  writer => writer.println(new Date)
}
```

Client code can now invoke the fileWriter merely providing the function that will determine the actual value to write. If this is returned to client code via a factory or explicitly imported, they will merely see this as a part of the language.

In fact this can be written more concisely yet using the implicit parameter syntax in which an underbar is used to represent a parameter passed into the function and used within that function:

```
  fileWriter {
  _ println new Date
}
```

which is very clean and simple to present to client code.

The advantages of Currying are:

- It allows new *language* constructs to be created.
- It allows constructs to be bound at runtime to specific objects, etc. If this approach is combined with the Factory pattern, then the details can be hidden from client code.

The drawbacks of Currying include:

- Partially Applied functions are more flexible in terms of which parameters are applied and which are not yet applied.