Chapter 5
Functional Programming

5.1 Introduction

Previous chapters have focussed on the Object-Oriented side of the Scala language. However, Scala is a hybrid Object Oriented (OO) and functional programming (FP) language. In this chapter, we will now look at functional programming and its advantages and disadvantages.

5.2 What Is Functional Programming?

Wikipedia describes Functional Programming as:

> … a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

There are a number of points to note about this definition. The first is that it is focussed on the computational side of computer programmes. You might consider that obvious, but at least half of what we have looked at around objects and classes has been focussed on the representation of domain concepts and data within those concepts. Thus there is a difference of emphasis between the functional programming world and the Object-Oriented programming world.

Another thing to note is that the way in which the computations are represented emphasises functions that generate results based on data and computations. These functions only rely on their inputs and generate a new output. They do not rely on any side effects and do not depend on the current state of the program. Taking each of these in turn:

1. *Functional programming aims to avoid side effects*. A function should be replaceable by taking the data it receives and inlining the result generated (this is

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_5

referred to as referential transparency). This means that there should be no hidden side effects of the function. Hidden side effects make it harder to understand what a program is doing and thus make comprehension, development and maintenance harder. Pure functions have the following attributes:

- the only observable output is the return value.
- the only output dependency are the arguments.
- arguments are fully determined before any output is generated.

2. *Functional programming avoids concepts such as state*. Lets us take these as separate issues. If some operation is dependent upon the state of the program or some element of a program, then its behaviour may differ depending upon that state. This may make it harder to comprehend, implement, test and debug. As all of these impacts on the stability and probably reliability of a system, state-based operations may result in less reliable software being developed. As functions do not (should not) rely on any given state (only upon the data they are given) they should as a result be easier to understand, implement, test and debug.
3. *Functional programming promotes immutable data*. Functional programming also tends to avoid concepts such as mutable data. Mutable data is data that can change its state. By contrast immutability indicates that once created, data cannot be changed. In Scala Strings are immutable. Once you create a new string you cannot modify it. Any functions that apply to a string that would conceptually alter the contents of the string, result in a new String being generated. Scala takes this further by having a presumption of immutability that means that by default all data holding types are immutable. This ensures that functions cannot have hidden side effects and thus simplifies programming in general.
4. *Functional programming promotes declarative programming* (and is in fact a subtype of declarative programming), which means that programming is oriented around expressions that describe the solution rather than focus on the imperative approach of most procedural programming languages. These languages emphasise aspects of how the solution is derived. For example, an imperative approach to looping through some container and printing out each result in turn would look like this:

```
int sizeOfContainer = container.length
for(int I = 1 to sizeOfContainer) do
   element = container.get(i)
   print(element)
enddo
```

whereas a functional programming approach would look like:

```
container.foreach(print)
```

Functional programming has its roots in the lambda calculus, originally developed in the 1930s to explore computability. Many functional programming languages can thus be considered as elaborations on this lambda calculus. There have been numerous pure punctional programming languages including Common Lisp, Clojure and Haskell. Scala allows you to write in a purely functional programming style or to combine functions with objects. Care needs to be taken when doing this that the principles of functional programming, and thus the advantages of functional programming, are not undermined. However, when used judiciously functional programming can be a huge benefit for, and an enhancement to, the purely Object-Oriented world.

To summarise then:

Imperative Programming is what is currently perceived as traditional programming. That is, it is the style of programming used in languages such as C, C++, Java and C#. In these languages a programmer tells the computer what to do, e.g. $x = y + z$. It is thus oriented around control statements, looping constructs and assignments.

Functional Programming aims to describe the solution, that is, *what* the program needs to be doing (rather than *how* it should be done).

5.3 Advantages to Functional Programming

There are a number of significant advantages to functional programming compared to imperative programming. These include:

1. **Less code**. Typically a functional programming solution will require less code to write than an equivalent imperative solution. As there is less code to write, there is also less code to understand and to maintain. It is therefore possible that functional programmes are not only more elegant to read but easier to update and maintain. This can also lead to enhanced programmer productivity as they spend less time writing reams of code as well as less time reading those reams of code.
2. **Lack of (hidden) side effects (Referential Transparency)**. Programming without side effects is good as it makes it easier to reason about functions (that is a function is completely described by the data that goes in and the results that come back). This also means that it is safe to reuse these functions in different situations (as they do not do unexpected things). It should also be easier to develop, test and maintain such functions.
3. **Recursion is a natural control structure**. Functional languages tend to emphasis recursion as a way of processing structures that would use some form of looping constructs in an imperative language.

Although you can often implement recursion in imperative languages it is often easier to do in functional languages. It is also worth noting that recursion is a very

expressive and a great way for a programmer to write a solution to a problem; however it is not as efficient at run time as looping. However, any expression that can be written as a recursive routine can also be written using looping constructs. Functional programming languages often incorporate tail end recursive optimizations to convert recursive routines into iterative ones at runtime. Essentially, if the last thing a routine does before it returns is to call another routine, rather than actually invoking the routine and having to set up the context for that routine, it should be possible to reuse the current context and to treat it in an iterative manner as a loop around that routine. This means that both the programmer benefits of an expressive recursive construct and the runtime benefits of an iterative solution can be achieved using the same source code. This option is typically not available in imperative languages.

- **Good for prototyping solutions**. Solutions can be created very quickly for algorithmic or behaviour problems in a functional language. Thus allowing ideas and concepts to be explored in a rapid application development style.
- **Modular functionality**. Functional programming is modular in terms of functionality (where Object-Oriented languages are modular in the dimension of components). They are thus well suited to situations where it is natural to want to reuse or componentise the behaviour of a system.
- **The avoidance of state-based behaviour**. As functions only rely on their inputs and outputs (and avoid accessing any other stored state) they exhibit a cleaner and simpler style of programming. This avoidance of state based behaviour makes many difficult or challenging areas of programming simpler (such as those used in concurrency applications).
- **Additional control structures**. A strong emphasis on additional control structures such as pattern matching, managing variable scope, tail recursion optimizations.
- **Concurrency and immutable data**. As functional programming systems advocate immutable data structures it is simpler to construct concurrent systems. This is because the data being exchanged and accessed is immutable. Therefore multiple executing thread or processes cannot affect each other adversely. The Akka Actor model builds on this approach to provide a very clean model for multiple interacting concurrent systems.
- **Partial evaluation**. Since functions do not have side effects, it also becomes practical to bind one or more parameters to a function at compile time and to reuse these functions with bound values as new functions that take fewer parameters.

5.4 Disadvantages of Functional Programming

If functional programming has all the advantages previously described, why isn't it the mainstream force that imperative programming languages are? The reality is that functional programming is not without its disadvantages, including:

- *Input–output is harder in a purely functional language*. Input–output flows naturally align with stream style processing, which does not neatly fit into the data in, results out, nature of functional systems.
- *Interactive applications are harder to develop*. Interactive application is constructed via request response cycles initiated by a user action. Again these do not naturally sit within the purely functional paradigm.
- *Continuously running programs* such as services or controllers may be more difficult to develop, as they are naturally based upon the idea of a continuous loop.
- *Functional programming languages have tended to be less efficient on current hardware platforms*. This is partly because current hardware platforms are not designed with functional programming in mind and also because many of the systems previously available were focussed on the academic community where out and out performance was not the primary focus. However, this has changed to a large extent with Scala and the functional language Heskell.
- *Not data oriented*. A pure functional language does not really align with the needs of the primarily data-oriented nature of many of today's systems. Many (most) commercial systems are oriented around the need to retrieve data from a database, manipulate it in some way and store that data back into a database. Such data can be naturally represented via objects in an Object Oriented language.
- *Programmers are less familiar* with functional programming concepts and thus find it harder to pick up function-oriented languages.
- *Functional programming idioms are often less intuitive* to (traditional) programmers than imperative idioms (such as lazy evaluations) which can make debugging and maintenance harder.
- Many functional programming languages have been viewed as *Ivory tower languages* that are only used by academics. This has been true of some older functional languages but is increasingly changing with the advent of languages such as Scala.

5.5 Scala and Functional Programming

Scala overcomes many of the disadvantages of functional programming by providing a hybrid environment in which you can use the Object-Oriented features of the language to represent concepts, data-rich elements, etc. and use functions to express behaviour oriented aspects of a program. It thus provides a *best of both worlds* approach to your choice of programming language constructs.