# Comparison of functional programming in Scala and Go

Sebastian Ester, Concepts of programming languages

# Table of Contents

# Introduction

The subject of this study work is a comparison between the two programming languages Scala and Go, in which the focus is on functional programming.
In the first step we will take a look at the history of both languages, for which applications they are used and what the design goals were.
The second chapter covers the basics of each language. This includes the setup, data types and functions.
After getting familiar with the basic mechanisms, the main part of this study work is to compare the capabilities of Scala and Go in terms of functional programming. Highlighted aspects are immutable data, functions of higher order, pattern matching and lazy evaluation.
As it turns out, Go has only little support for functional programming - only functions of higher order are available by default. Scala, on the other hand, is much more powerful, because it contains all features you would expect from a functional programming language.

# History

Scala is a statically typed and compiled language which runs on the JVM and combines two concepts: functional and object oriented programming. It was designed by Martin Odersky, a professor at the École Polytechnique Fédérale de Lausanne in Switzerland, and was first released in 2003. He wanted to find out if functional and object oriented programming would fit together ergonomically. There were no pain points from other languages that should be fixed in Scala. It was only designed for research purposes.
Using the JMV as a runtime environment had two reasons: Firstly, Martin Odersky was strongly involved in the development of one of the first Java compilers, so he knew the JVM very well. The second reason is that functional programming implies dealing with immutable data and thus many volatile objects in memory. So a high performance garbage collector, which is provided by the JVM, is essential. For further details, see [scalaInterviewA] and [scalaInterviewB].

Go is also a statically typed and compiled language, but it has a very different background story than Scala. As described in [goFaq], Go was designed by the three Google developers Robert Griesemer, Rob Pike and Ken Thompson and has evolved from the frustration about other languages. Google addresses the issue that you had to choose between safety, performance and the ease of programming. If you choose for example efficient execution, think of C++ with its enormous compile times and the susceptibility for memory leaks. At this time, there was no language that combined the three points.
Also concurrent programming, which became more and more important due to multi core processors and distributed systems, is not supported by the mainstream languages in a way that is fun for the developer.
Google wanted to eliminate these issues with Go so that programmers can focus on their business logic rather than fighting with the language itself.

Go and Scala are both general purpose programming languages with a wide range of applications where they can be used. From backend development to microservice architectures. In Scala even mobile applications for Android can be written.

# Basics

This chapter contains the most important basic elements of Go and Scala. It is the base for the functional programming aspects discussed later on.

As already said, Scala and Go are both statically typed and compiled languages. The compiled Scala code runs on the JVM, while the Go compiler creates a single executable, which is specific for the available hardware architecture.
Compiling and running Scala code via commandline is okay at the beginning, but if the project gets more complex, it is recommended to use a build tool like "sbt" (see [scalaBuild]). Go, on the other hand, has no need for any build tool, because managing go code from the commandline works just fine, even for larger projects.
A possible IDE for Scala and Go is Intellij (with the required Plugins). For Go, there is even a separate IDE named GoLand. For experimental purposes, Scala provides a REPL for the command line and for Go there is an online playground (https://play.golang.org/). All provided examples in this work can be tested in these two environments.
Unless otherwise specified, the information about Scala and Go relates to [scalaTour] and [goTour] hereafter.

Now let's start with some basic mechanisms.
In Scala there are two types of variables: constant variables, called "values", and variables themselves. Formally spoken, values are named results of expressions. Go supports these two variable types, too, with the limitation that only primitive data types (see below) can be declared as a constant.

*Variables and values in Scala*

```
val x = 5
x = 7   // compile error
var y: Int = 5
y = 7   // ok
```

*Variables and constants in Go*

```
package main

import "fmt"

func main() {
    x := 5
    var y int
    y = 5
    const z = 5

    fmt.Println(x)
    fmt.Println(y)
    fmt.Println(z)
}
```

Variables and values in Scala and Variables and constants in Go show the syntax for defining variables and constants/values in each language. As already said, values in scala are constant and thus a reassignment won't compile. The type specification is in both languages optional because it can be inferred by the compiler. Go brings additional syntactic sugar for variable definition with the ":=" operator. The first assignment in Variables and constants in Go has the same effect as the following two lines together. It combines declaration and initialization of a variable in one statement.

# Data types

In Scala, every data type is an object. Also the "primitive" types like Int or Bool are objects and there is no distinction between value and reference.

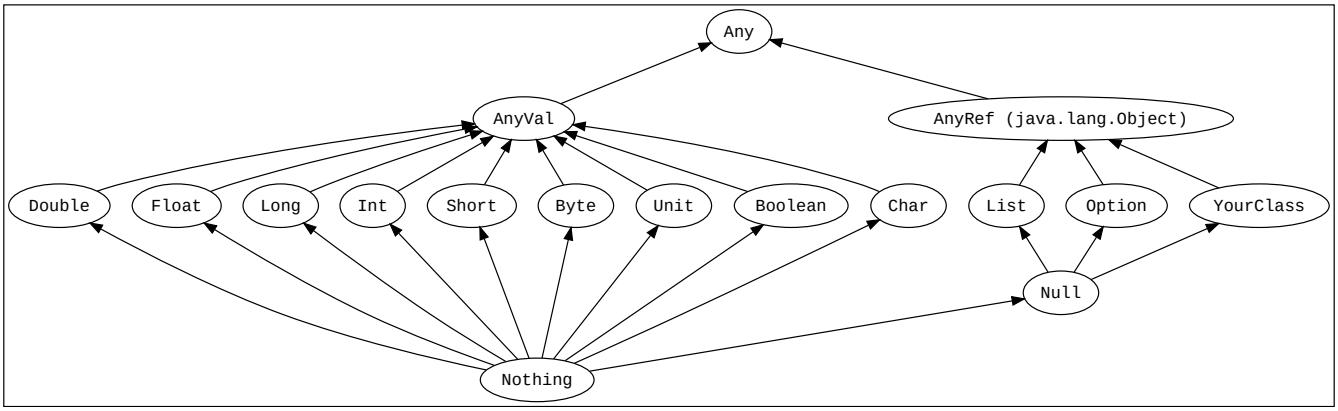


*Figure 1. Type hierarchy in Scala (see [scalaTour])*

Type hierarchy in Scala (see [scalaTour]) shows the hierarchy of objects in Scala. As you can see, the basic data types are indeed objects, but they are treated differently than the "real" objects in the way that they are non-nullable. Any other object (including the user defined classes) inherits from AnyRef which corresponds to java.lang.Object and can therefore be null.
A type that maybe not every developer is familiar with is "Unit". It becomes handy at functions, but more on that later. Compound data types are realized with classes and objects, but these are not discussed further in this paper.

In opposition to Scala, Go does distinguish between basic and compound/object data types (see below). Go has the following basic data types:

- bool
- string
- int[8/16/32/64]
- uint[8/16/32/64]
- byte (alias for uint8)
- rune (alias for int32 - represents a unicode code point)
- float32/64
- complex64/128

Go has additionally the concept of pointers, which represent the address of a value. This feels quite

odd, because the use of pointers in C and C++ is very tricky and error-prone. But the developers of Go have learnt from the mistakes of C and C++ and made pointers safe and easy to use. For example returning a pointer from a function is no problem. The syntax of pointers is the same as in C (except that the pointer type is written *[data type]) and an example is shown in the next chapter.
Compound data types are realized in Go with "structs", which are also well-known for C and C++ programmers. How they work is also not discussed further.

For better readability, both languages allow type aliasing with the keyword "type".

# Functions

There is a distinction between functions and methods in Scala and Go, because they support both object orientation. But for this paper, only functions are relevant, and therefore this chapter will cover the syntax for defining them in both languages. Particularities relating to functional programming will be covered later on.

Functions in Scala are actually anonymous functions, respectively function literals. They are assigned to a value and can then be called by the value's name.

*Function definition in Scala*

```
// single line function
val square = (x: Int) => x * x

// multi line function
val squareAndDouble = (x: Int) => {
    val squared = x * x
    squared * 2
}

// function call
val result = squareAndDouble(5)
```

Function definition in Scala shows two function definitions and how to call a function in scala. What might be conspicuous is that there is no "return" statement, as most developers are probably used to from other programming languages. Well, there is actually a return statement, but by convention, it is never used in Scala. The returned value from a function is the result of the last executed expression. For this reason, the last executed expression of a function (even if it's within an "if" statement) must match the specified return type of the function.
If the function body consists of only one line or one expression, respectively, the curly braces may be omitted.

Another material characteristic of scala functions is that every function must return a value. Accordingly, there is no keyword like "void". This is the point where the previously mentioned datatype Unit becomes handy. There are situations where you really dont't need to return a value from a function, and returning a dummy value just to satisfy the compiler isn't the most beautiful style. Returning the type Unit simply means that there is actually no return value.

Syntactically, Go orientates itself more towards the mainstream languages like Java. Curly braces

for the function body are obligatory and there is also the well-known return keyword. A further difference to Scala is that a function may return no value. But there is also no keyword like void, in Go you just omit the return type.

*Function definition in Go*

```
package main

import "fmt"

func main() {
    // function call
    result := square(5)
    fmt.Println(result)

    var a, b = 10, 11
    swapWithPointers(&a, &b) // a = 11, b = 10 now
    fmt.Printf("%d %d\n", a,b)

    _, b = swapWithReturnValues(a, b) // b = 11 again
    fmt.Printf("%d %d", a,b)
}

func square(x int) int {
    return x * x
}

// swaps the values of two integers using pointers
func swapWithPointers(x, y *int) {
    *x, *y = *y, *x
}

// swaps the values of two integers using multiple return values
func swapWithReturnValues(x, y int) (int,int) {
    return y, x
}
```

Function definition in Go shows an example of two simple functions written in Go. As you can see in the function swapWithPointers, Go allows multiple assignments in one line, which makes the code more concise if you don't overdo it with the number of assignments.
This concept leads one to suppose that returning multiple values from a function is also possible, which is shown in the function swapWithReturnValues. An underscore means that the corresponding returned value is ignored.

# Functional programming

In functional programming, functions are treated as mathematical expressions, which take some input parameters and produce some output that depends only on the input. For this reason, functions must not have any side effects. Or generally speaking, any kind of hidden state must be

avoided according to [goFunctional]. This idea implies a different style of programming, and the following chapters describe how the main parts are implemented in Scala and Go.

# Immutable data

In order to achieve statelessness in functions, it is very handy to work with immutable data. It reduces the risk of accidental side effects from the outset.
In Scala, every collection type, like a List or Map, is immutable by default. It is possible to use mutable collections, too, but Map creation in Scala shows that you simply don't want to use them, because it's unpleasant.

*Map creation in Scala*

```
val immutableMap = Map("Alice" -> 25, "Bob" -> 42)
var mutableMap = scala.collection.mutable.Map("Alice" -> 25, "Bob" -> 42)
```

As you can see, Scala strongly encourages you to use its functional programming features. And on the basis that all collection types are immutable, it is recommendable to use immutable data types everywhere, because there is simply no reason not to do it, as you can see in the following.

Scala supports the handling of immutable data (especially collections) with dedicated methods. List methods in Scala shows a small excerpt of those methods. None of them changes the original list, but creates a new value which is derived from the original list. There are many further methods, that work according to this principle, including the prominent "Map" functionality, which is part of the next chapter.

*List methods in Scala*

```
val list            = List(1,2,3,4,5,6)
// get first value of the list
val head            = list.head         // = 1
// get last value of the list
val tail            = list.tail         // = 6
// prepend 0 to the list
val largerList      = 0 :: list         // = List(0,1,2,3,4,5,6)
// replace the third element of the list
val modifiedList    = list.updated(2,9) // = List(1,2,9,4,5,6)
```

Data types in Go are by default mutable (except strings) and only the primitive data types like bool or int can be declared as a constant. So it's the developer's responsibility to ensure the immutability of compound data types. Even if this reduces the comfort a bit, it is nevertheless possible to do so.

*Immutable data in Go*

```
package main

import "fmt"

func main() {
        number := 5
        number = number + 1     // wrong
        numberInc := number + 1  // right
        fmt.Println(numberInc)

        numbers := []int{1,2,3,4}
        numbers[2] = 7                                          // wrong
        modifiedNumbers := replaceArrayElement(numbers, 2, 9)   // right
    fmt.Println(modifiedNumbers)
}

func replaceArrayElement(array []int, index int, newElement int) []int {
    arrayCopy := make([]int, len(array))
    copy(arrayCopy, array)
    return append(append(arrayCopy[:index], newElement), arrayCopy[index+1:]...)
}

func head(array []int) int {
    return array[0]
}

func tail(array []int) int {
    return array[len(array)-1]
}
```

Immutable data in Go shows some possibilities to make mutable data immutable (see [goFunctional]). Assignments of primitive data types are simple to deal with; you just create a new variable for each modification. In order to protect complex data structures or collections against modifications, more logic is needed to realize this. In the example, such an operation is shown exemplary in the function "replaceArrayElement". A deep copy of the array is needed to protect the original array against modification.
Of course, such a method would have to be written in a generic way to be usable. Nevertheless, the example shows that it takes not too much effort to realize immutable data in Go.

If you think one step further, the actual problem is not implementing utilities for immutable data, but to forbid mutable data. There is no support for this, so you would have to write your own style checker to enforce a purely functional style of programming in Go. Scala, on the contrary, has a built-in "enforcement" of immutable data by making mutable data very inconvenient to use.
If you look at the performance, both languages have the capabilities to deal with umpteen of copies because of their garbage collectors. Passing large objects around is no problem as well, as Scala passes only the address of an object, and in Go you can use pointers. But again, in Go the developer must take action to avoid performance leaks, Scala does that automatically for you.

Under the aspect of dealing with immutable data, Go has the capabilities of doing it, but leaves too much responsibilities to the developer, so that it's simply not practicable at justifiable expense. Scala, on the other hand, uses immutable data by default in many places and makes it as hard as possible for the developer not to use it.

# Higher order functions

Functions are first class citizens in Scala and Go, which means that they are equal to usual data types. Accordingly, they can be passed into functions as parameters or can be returned out of functions. A prominent example of a function which takes another function as a parameter is the Map function, which modifies the elements of a collection, whereby the logic is located in the passed function.

*Higher order functions in Scala (Part 1)*

```
val list            = List(1,2,3,4)
val doubledList     = list.map(_ * 2)   // = List(2,4,6,8)
// alternative notation
val doubledListAlt  = list.map(x => x * 2)

val square      = (x: Int) => x * x
val squaredList = list.map(square)  // = List(1,4,9,16)

val execute     =
    (funcToExecute: (Int, Int) => Int, x: Int, y: Int) => funcToExecute(x, y)
val result      = execute(_ + _, 4, 5) // = 9
// alternative notation
val resultAlt   = execute((x, y) => x + y, 4, 5)
```

If you look at the first example in Higher order functions in Scala (Part 1): that's what i call concise code. Conveniently, Scala makes the Map function available for all collections. And for the reason that the compiler can infer the type of the elements contained in the list, no more code is necessary to describe the desired behaviour. Of course the use case is not always that simple, so it is also possible to pass a function to Map that was previously defined (see second example).

The syntax to write an own function that takes another function as a parameter is described in the third example. To the left of the first arrow are the input types of the function (two integer values in this case), surrounded with parentheses. If there is only one input parameter, the parentheses can be omitted.
To the right of the first arrow is the return type of the function (also an integer value). In order to pass a function that adds two integers, the previously introduced shorthand notation with two underscores can be used. A more verbose notation must be used if named parameters are required in the function body (last line of the example).

Go does not offer as much comfort as Scala, because it does not provide an implementation of the Map function. Fortunately, the implementation is no big deal, as shown in Higher order functions in Go (Part 1).

*Higher order functions in Go (Part 1)*

```
package main

import "fmt"

type Any interface{}

func main() {
    list := []Any{1,2,3,4}
    mapper := func(element Any) Any {
        return element.(int) * 2
    }
    result := Map(mapper, list)
    fmt.Println(result)
}

func Map(mapper func(Any) Any, input []Any) []Any {
    ret := make([]Any, len(input))
    for index, element := range input {
        ret[index] = mapper(element)
    }
    return ret
}
```

In the given example, the Map function is written in a generic way, so that arrays of any type can be passed as a parameter. For better readability and less typing, the empty interface is replaced by the type alias Any. To let the compiler know about what type was actually passed into the function, an explicit cast has to be made in the passed mapper function. It is not mandatory to bind the mapper function to a variable, it can also be passed directly to Map using the same syntax.
Unlike Scala, there is no shorter notation for passing functions. The func keyword, named parameters and curly braces are always mandatory.

An example of a function which is being returned out of another function is shown in Higher order functions in Scala (Part 2) and Higher order functions in Go (Part 2).

*Higher order functions in Scala (Part 2)*

```
val getGreeter = () =>
        (firstName: String, lastName: String) => s"Hello $firstName $lastName"
val greeter = getGreeter()
val greeting = greeter("John", "Doe")  // = "Hello John Doe"
```

Scala is again very concise in its syntax. "getGreeter" is a function without parameters (indicated by the empty parentheses) that returns a function with two strings as input parameters and a string as output. The return type of getGreeter can be omitted, because it's inferred by the compiler. Also noteworthy is the syntax of concatenating string values with string literals by the dollar sign.
Calling getGreeter assigns the returned function to a new value named "greeter", which is now itself a function. Now, greeter can be called with two strings (a name) and the result is a greeting for the

respective name.

*Higher order functions in Go (Part 2)*

```
package main

import "fmt"

func main() {
    greeter := getGreeter()
    greeting := greeter("John", "Doe")
    fmt.Println(greeting)
}

func getGreeter() func(string, string) string {
    return func(firstName string, lastName string) string {
        return "Hello " + firstName + " " + lastName
    }
}
```

Higher order functions in Go (Part 2) implements the same functionality as the corresponding example in Scala. Go's syntax is more verbose, but in essence there is not much overhead compared to Scala.

In terms of higher order functions, Scala and Go offer the same possibilities to the developer. Also closures, which are not part of the examples, are possible in both languages. It is a matter of taste if you prefer a uniform style throughout the whole codebase like in Go, or the concise notation in Scala that might perhaps be hard to read sometimes (p.ex. for beginners).

# Incomparable features

This chapter is a small outlook to features of Scala - with regard to functional programming - that aren't supported in Go. Of course they could be implemented somehow in Go, but they are not part of the language itself and thus no comparison is made here.

**Pattern matching**
Imagine pattern matching as some sort of regular expressions at a different level. Here, values are checked against a specific pattern, and depending on the result of the check (true or false), a specific execution branch is chosen.

*Pattern matching in Scala*

```
def matchTest(x: Any): String = x match {
    case s: String => s
    case i: Int => i.toString
    case _ => "No match"
}
```

Pattern matching in Scala shows a simple example, where a parameter of any type is checked

against three patterns:

- String: if the parameter is of type string, this execution branch is chosen
- Int: if the parameter is an integer, this execution branch is chosen and the integer is additionally converted into a string to match the return type
- _: matches everything, so this is the default branch if none of the two patterns matched before

There are much more patterns, but this would be beyond the scope of this work.

**Lazy evaluation**
Most programming languages (including Go) practice the concept of eager evaluation, meaning that the arguments of a function are evaluated before the function is called. In contrast, lazy evaluation means that all arguments are only evaluated when they are actually needed (see [eagerVsLazy]). A good example for demonstrating lazy evaluation are Scala Streams. They behave like lists, but their elements are computed lazily.

*Lazy evaluation in Scala*

```
val stream          = Stream.from(1)
val mappedStream    = stream.map(_ * 2)
mappedStream(4)  // = 10
```

It is even possible to create infinite streams, like it's shown in Lazy evaluation in Scala. Without the values being computed lazily, this wouldn't work. Also operations on streams are lazy, so it's possible to create a new stream with the Map function (in the example, the values of the original stream are doubled). Only when the values of a stream are actually needed, meaning when there is an access operation (like the last statement in the example), they are eventually computed. For further details, see [scalaStream].

# Summary

Not only the background of Scala and Go is very different, but also in many other aspects the languages differ from each other. Beginning with the type system, Scala is purely object orientated while Go follows an approach similar to C with various integers, pointers and structs for compound data types.
Function definition and function calls are realized quite similarly in both languages, whereby each has its special features. Scala functions must return a value, but this is done without a return statement. Furthermore, if you actually don't want to return something, there is the special type Unit. Go, on the other hand, allows the assignment of multiple values at once as well as returning multiple values from a function.

Another point is that Scala offers you a wide range of syntax, depending on the complexity of the use case. Simple use cases require only minimal code, which fosters the ease of programming. Go is less flexible here, it allows only little syntactic deviations, which makes the code look quite uniform. There is no better or worse here, it's a matter of taste which style you prefer.

In the area of functional programming, the different design goals of the two languages are clearly

noticeable. Scala was designed with functional programming in mind and supports all features you'd expect from a functional programming language.
Go, on the other hand, has a different focus. The only functional feature that is available out of the box is higher order functions. All other aspects that make up a functional language, for example immutable data, pattern matching, etc., are not available and would have to be implemented subsequently at great expense. Also the language doesn't provide any possibility to enforce a functional coding style. As you see, Go is just not designed for functional programming, so everything that goes beyond functions of higher order would ruin the ease of programming the designers wanted to achieve.

# References


- [scalaInterviewA] Paul Krill. Scala designer cites goals for JVM language alternative. https://www.infoworld.com/article/2620033/application-development/scala-designer-cites-goals-for-jvm-language-alternative.html (visited at 02.01.2019)
- [scalaInterviewB] Bill Venners and Frank Sommers. The Goals of Scala's Design. https://www.artima.com/scalazine/articles/goals_of_scala.html (visited at 02.01.2019)
- [goFaq] Frequently Asked Questions (FAQ). https://golang.org/doc/faq (visited at 02.01.2019)
- [scalaBuild] Alvin Alexander. How to compile Scala code with 'scalac' and run it with 'scala'. https://alvinalexander.com/scala/how-to-compile-scala-source-code-scalac-examples (visited at 02.01.2019)
- [scalaTour] Tour of Scala. https://docs.scala-lang.org/tour/tour-of-scala.html (visited at 02.01.2019)
- [goTour] A Tour of Go. https://tour.golang.org (visited at 02.01.2019)
- [goFunctional] Geison. Functional Go. https://medium.com/@geisonfgfg/functional-go-bc116f4c96a4 (visited at 02.01.2019)
- [eagerVsLazy] Eager vs. Lazy Evaluation. Higher-Order Functions. http://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture6.htm (visited at 02.01.2019)
- [scalaStream] Alvin Alexander. How to use the Scala Stream class, a lazy version of List. https://alvinalexander.com/scala/how-to-use-stream-class-lazy-list-scala-cookbook (visited at 02.01.2019)