

# Scala on Android: the-current-state-of-the-art

Chris Howell

March 2, 2016



## **Abstract**

This document features a review of the viability of an alternative programming language as the basis for an Android application. I will present a test case, a review of the development process, an in-depth study of an alternative programming language in comparison to Java and a reflective review detailing my experience and opinion on the subject matter.

# Contents

|          |                                        |           |
|----------|----------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                    | <b>5</b>  |
| 1.1      | Background . . . . .                   | 5         |
| 1.2      | Why Scala . . . . .                    | 5         |
| 1.3      | Measuring Success . . . . .            | 6         |
| <b>2</b> | <b>Development Process</b>             | <b>6</b>  |
| 2.1      | Setting Up The Environment . . . . .   | 7         |
| 2.2      | SBT and Gradle . . . . .               | 8         |
| 2.3      | Conscript and giter8 . . . . .         | 8         |
| 2.4      | SBT project structure . . . . .        | 8         |
| 2.5      | Memory overhead and APK size . . . . . | 8         |
| 2.6      | Reflection . . . . .                   | 9         |
| <b>3</b> | <b>Scala on Android</b>                | <b>10</b> |
| 3.1      | Immutability . . . . .                 | 11        |
| 3.2      | Objects and Classes . . . . .          | 12        |
| 3.3      | Case Classes . . . . .                 | 12        |
| 3.4      | Pattern Matching . . . . .             | 13        |
| 3.5      | Option type . . . . .                  | 14        |
| 3.6      | Implicit conversion . . . . .          | 14        |
| 3.7      | Traits . . . . .                       | 15        |
| 3.8      | Futures and Promises . . . . .         | 17        |
| 3.9      | Actors . . . . .                       | 17        |
| <b>4</b> | <b>Type System</b>                     | <b>17</b> |
| 4.1      | Implicit Definition . . . . .          | 18        |
| 4.2      | Bounded Type . . . . .                 | 18        |
| 4.3      | Variance Annotation . . . . .          | 19        |
| 4.4      | Partial Application . . . . .          | 19        |
| 4.5      | First Order Functions . . . . .        | 19        |
| 4.6      | Higher Order Functions . . . . .       | 19        |
| 4.7      | Type Inference . . . . .               | 20        |
| 4.8      | Type Functions . . . . .               | 20        |
| 4.9      | Higher Kinded Types . . . . .          | 20        |
| 4.10     | Yield Operators . . . . .              | 20        |
| 4.11     | Operator Overloading . . . . .         | 20        |
| <b>5</b> | <b>Kotlin</b>                          | <b>21</b> |
| 5.1      | Comparison with Scala . . . . .        | 22        |
| 5.2      | Type System . . . . .                  | 22        |
| 5.2.1    | Null values . . . . .                  | 22        |
| 5.2.2    | Type projections . . . . .             | 22        |

|          |                                 |           |
|----------|---------------------------------|-----------|
| 5.3      | Java interoperability . . . . . | 22        |
| 5.4      | Developer support . . . . .     | 22        |
| <b>6</b> | <b>Project Reflection</b>       | <b>23</b> |
| 6.0.1    | PROS . . . . .                  | 24        |
| 6.0.2    | CONS . . . . .                  | 24        |
| <b>7</b> | <b>Conclusions</b>              | <b>24</b> |

# 1 Introduction

## 1.1 Background

This report describes my research into an alternative programming language for use on Android. I hope that my research will highlight the viability of using alternative languages when creating applications and the additional benefits the alternative language can offer in contrast to Java. I wanted to spend the next few paragraphs talking about the reason why I have decided on this project and what it means to me.

I picked this project because it is an area of personal interest to me. With Java being the first language I learnt in detail, and generally the first language I turn to when building new applications. As my experience in programming has grown, I have started to branch out into learning other languages, with these including PHP, JavaScript, Haskell, C++ and finally Scala.

With Android application programs on the increase I felt like my skills and experience have reached the point where I am now competent enough to create Android applications. Understandably I was frustrated to know that I would be forced to continue using Java when there are many other languages available as a computer programmer.

Fortunately the Java developers design choice of write once, run anywhere enables alternative languages that compile down into Java byte code to also run on Googles version of the JVM called Dalvik. Any programming language that executes on the JVM, can also execute on Dalvik. In recent years the JVM has seen a growth in the number of languages that are capable of

## 1.2 Why Scala

While Java is indisputably the current go-to language when developing android applications, I hope that my test case highlights the possibility that android developers have when creating applications. The android applications byte code executing on the VM increases the prospect of an alternative language that will one day dethrone Java. It would require no significant change in hardware/software as the VM is already well established.

The test code I have prototyped for this project implements Scala on Android to create a distributed messenger system. As I have stated in my discussion of the development process, while the initial set-up can require some initial configuring, once this stage has been complete you can take advantage of a less verbose language than Java.

While Scala is sometimes referred to as Java without the semicolon, this is a statement that could not be further from the truth. Scala is, in my opinion, currently one of the best options for developers to blend object oriented principles with functional programming when writing code. Because it is neither a complete imperative language nor a purely functional language, it is essentially a new programming paradigm that redefines the rules when coding. It is an expressive language that provides a much richer type system than Java currently supports.

A strong type system like Scala's is an effective method of ensuring the code stays bug free during runtime, with most of the errors being caught during compilation.

### 1.3 Measuring Success

- A working test case.
- A review of the development process with a discussion of any issues that arose from this development period.
- A detailed discussion of the pros and cons of using Scala on Android with code examples and comparison to Java.
- A review/comparison of the Scala and Java type systems.
- A discussion of the programming language Kotlin.
- A summary and conclusion stating my opinion on the future of Android application development.

## 2 Development Process

During my research stage of the project I came across one recurring theme in many of the experiences of other developers keen for an alternative to Java on Android. The recurring theme is the difficulty in setting up the development environment. From my own personal experience I can agree with this still being the case.

I am of the opinion that until the setup process has been streamlined to enable a developer to spend more time coding their application than configuring the environment, there will be no significant leap forwards in a new language taking prominence over Java. I believe this situation will remain the same until Google decide to throw their support behind a successor/alternative language to Java.

What is interesting and I will cover in greater detail in a later section is that JetBrains are currently working on seamlessly incorporating their Kotlin programming language into IntelliJ IDEA. This is an extremely interesting development as

its a programming language written by programmers for programmers. It shares a lot of similarities to Scala in that it is a hybrid functional/imperative language.

It should be noted that I have only tested one alternative language on Android, in this case Scala, so I cannot state for a fact if it is easier or harder setting up the environment when using a language like Groovy. From reviewing some experiences by other developers of setting up the development environment to utilize Groovy the general experience seems to be slightly easier than Scala. Arguably because Groovy is an interpreted language means it has worse performance than Scala, but this falls outside the subject area for this report.

In recent years a large number of programming languages have appeared that run on the JVM, I believe this is in part to developers wanting to branch out from Java. The opportunity that is provided is that the JVM is already well established, so any machine capable of running the JVM or Java code is also capable of running any of these alternative languages. With the fact that the JVM is already a well-supported method of launching applications that are machine independent it is the logical approach for developers to use this as a base for building new programming languages.

The next few sections of this report will cover what is in my opinion the easiest method for setting up an android application that uses SBT to compile and launch the application.

## **2.1 Setting Up The Environment**

Ideally when setting up the environment it is advisable to first download sbt to use as a command line build tool for compiling and debugging your projects. Sbt is a flexible, powerful and easy to use build tool written in Scala for Scala projects.

One of the major advantages of using SBT is that it allows a developer to state the dependencies for the project and will then manage these by downloading all the required libraries. This approach saves you having to add the individual JAR files to the class path when importing Akka actors etc.

Another reason why it is an advantage to use SBT is because it features incremental compilation, with this meaning that after the initial project compilation only classes that have been altered in some way need to be compiled again. This saves vast amounts of time and almost completely removes the issue of slow compilation for android apps written in Scala.

Along with SBT the Scala programming language needs to be downloaded and install onto the development machine. The latest iteration of Scala is version 2.11.7.

After installing Scala the development machine also need to the Android SDK install along with the appropriate API level. You will need to correctly set the path the Android SDK or this will cause issues on both Linux and Windows operating systems.

## **2.2 SBT and Gradle**

## **2.3 Conscript and giter8**

The easiest method for quickly setting up and compiling SBT-based android applications is via the command line build tool conscript and a giter8 template. It is possible to use the command line to create the project structure, this approach is more time consuming than downloading a template.

Conscript is a build tool that enables a developer to quickly download and build templates created by giter8 and other groups. This approach is the least confusing when building Scala-based Android applications. The main reason for this is that they quickly provide the developer with a basic working template without having to configure the dependencies etc.

Once conscript has been downloaded and install from the GitHub account it allows for the quick creation of new sbt-based android template via the command line. This project contains no code, but does help to quickly get a new project started without having to manually implement the basic project structure.

This approach is excellent for reducing the setup stage and for quickly getting a new app featuring Scala and SBT running with minimal effort.

## **2.4 SBT project structure**

The standard android application is often built using the Gradle build system to manage the project dependencies. The basic structure of an SBT android project structures differs slightly from a more standard gradle build with the difference being that dependencies for the project are all placed in the SBT build file.

## **2.5 Memory overhead and APK size**

A topic of much discussion is that the android apk file requires all its dependencies to be included in a single apk file. Unfortunately, this requires the whole Scala language to be included into the apk file when it is created. While in the past this did cause the base apk file size to be exceedingly large, often reaching 8MB+ for



the base project, this fortunately is no longer a real issue when using Scala.

```
Ignoring unused library classes...
  Original number of library classes: 3217
  Final number of library classes:    577
Shrinking...
Removing unused program classes and class elements...
  Original number of program classes: 7583
  Final number of program classes:    1583
```

One of the features of Proguard is that it can remove classes that aren't needed when compiling the apk file. One interesting feature of SBT is that we also define the Proguard configurations inside our build.sbt file. By using Proguard I was able to reduce the APK by a significant amount to a much more reasonable size. As the above image shows the number of classes removed was in excess of 6000. This is a significant number of redundant classes that aren't included in the final APK and helps reduce the memory overhead significantly.

```
[info] [testProject-debug.apk] Install finished: 538.71KB in 4.46s. 120.84KB/s
[info] Starting: Intent { cmp=my.android.project/.helloWorld }
[success] Total time: 21 s, completed 06-Feb-2016 12:53:48
```

As you can see it was possible to have the base APK file for this project, including the Scala Library and Akka actors compiling to a relatively tiny 538.71KB. This is a more than reasonable base APK size and I believe it is approximately the same base size when creating a Java project. We can also see that the total time it took to compile the project on a clean build was 21 seconds. As Proguard caches some compilation information when you recompile the program the resulting APK is slightly larger unless you clean the cache before building the project again.

## 2.6 Reflection

I believe that the main lesson I learnt from the development process is that it is entirely possible to write fast and responsive android applications using Scala as opposed to Java. While the memory overhead of using Scala was a problem when developers first started utilizing Scala on android, this is no longer a real issue in the development phase as Proguard removes the redundant classes from the Scala library when the APK is being generated.

Another well documented issue is the compilation time required to compile the project, with this often being in excess of two minutes. Fortunately, this is another non-issue at this stage of creating Scala Android applications. Because SBT feature incremental compilation the only classes that are compiled again are those that have been changed in some way. On average I waited no more than a few seconds for a class to compile after I had made some changes.

```
> compile
[info] Generating R.java
[warn] Warning: AndroidManifest.xml already defines debuggable (in http://schemas.android.com/apk/res/android); using existing value in manifest.
[info] Compiling 1 Scala source to C:\Users\Chris\testproject\target\android-bin\classes...
[success] Total time: 3 s, completed 06-Feb-2016 13:17:19
```

Using conscript and giter8 to quickly create the template project structure was an effective way of quickly overcoming the lack of official Android Scala support from IDE creators like JetBrains and the Android O/S owners Google. It takes a matter of second to quickly get a Java-based android project ready for coding when using an IDE like IntelliJ IDEA, and using Conscript/Giter8 approach you can reproduce this base template creation in a quick and easy fashion.

I would say overall while it was initially frustrating to get a project up and running using Scala, once I had created my first project the subsequent projects were much simpler to create as the environmental variables had already been correctly set and the languages/ tools had been installed and configured.

It should be noted that I wrote all of the code for this project using Notepad++ with the Scala Plugin. When I did try and open the program using IntelliJ IDEA it complained that the android packages werent recognized. For me this wasnt a problem as I didnt find it an issue to write my code in Notepad++. To build and test my code I used the command line and SBT. None of the reported issues by the IDEA IDE were reported or caused the application not to run when I build it from the command line with SBT.

### 3 Scala on Android

Some examples are given below to highlight some of the options available using Scala.

- Immutability
- Objects and classes
- Case classes

- Pattern matching
- Option types
- Implicit conversion
- Traits
- Futures and promises
- Actors

This list provides some of the features available to Scala developers that you won't find in the Java programming language. This may change in time as Java borrows more and more from Scala. At its core Java will always be an imperative and verbose programming language as this is deeply ingrained in the makeup of the language.

It should also be noted that while this list tries to capture the essence of the power that programmers can harness with Scala, this list should not be considered exhaustive as the language is large and extensive. All the examples shown are runnable on my Samsung A3 phone and are targeted for the Android API Level 16.

### 3.1 Immutability

One of the real cornerstones of functional programming is the approach to global state. The imperative approach is to have large numbers of global variables that help define the changing state of the programming. This approach can often lead to unexpected results when data is effected in some unforeseen way and this can often lead to runtime errors.

For example, Java requires a developer to write lots of codes usually using if/else statements to check that the data matches the expected result. If the developer doesn't include these checks at the appropriate points in a program, then the result will often be runtime errors. One method of ensuring shared data isn't corrupted is by utilizing synchronized methods, but again this can lead to verbose and hard to understand code.

A functional programming languages approach is to reduce the amount of global variables to the absolute minimum amount required, with this often being none at all. This approach to changing state results in less runtime error as each method created can be considered a pure function in that it only has control over local variables.

Scala provides the choice of allowing global variables to effect changing state in a traditional imperative manner. This is not the preferred choice when coding in

Scala and favouring immutability is encouraged unless absolutely required. When immutability is applied to code then generally it is less buggy and unlikely to crash during runtime due to unexpected data.

## 3.2 Objects and Classes

Classes work in a similar fashion to Java in that they are essentially the blueprint that contains all the methods accessible to that class. These blueprints can then be reused multiple times to create object instances from. The main difference between Scala classes and Java classes would be that Java favours global variables and mutable state, while Scala classes often feature no global variables and favours immutability. This means that Scala classes are likely to be less buggy during runtime as functions only effect local variables.

While Scala classes are similar to Java classes where these two languages do differ is that Scala allows you to create a singleton object. The closest comparison to this you can find in Java in declaring every method and variable inside a class static to essentially create a static object. Scala objects allow you to create a single instance of the object by using the object keyword instead of the class keyword.

## 3.3 Case Classes

Case classes are classes with attributes that are created in a clear and concise syntax. They offer a powerful and flexible way of quickly defining case classes that are easily combinable with pattern matching.

In the image below an abstract class called Message is defined which features two case classes that both contain String values. At a later date it will be a trivial matter to revisit and revise these attributes as required. While the value of this approach may not be immediately apparent the Akka actor model supports pattern matching on the receive function being called. With this approach I will be able to apply pattern matching to the message case class to determine the appropriate response.

```
abstract class Message
case class send(text:String) extends Message
case class received(text:String) extends Message
```

The next image shows the more verbose approach employed by Java, with this implementation still missing the code for the abstract Message superclass and the Received class. With the code included for all three required classes the verbose

nature of Java becomes apparent as significantly more code is required to achieve the same result as Scala case classes.

```
public class send(String text) extends message{  
    public String text;  
    public message(String text){  
        this.text = text;  
    }  
}
```

### 3.4 Pattern Matching

In addition to feature imperative if and else statements Scala also features pattern matching, with this being the more traditional approach to selection in functional programming languages. Pattern matching is often combined with recursion in functional programming; this approach is an alternative to iteration.

One powerful option that Scala allows is the ability to pattern match against classes, with the image below showing an example of this. The function takes an instance of the Message class that was defined earlier in the code and then uses pattern matching to determine which variation of the class was received as the input.

```
def match(m:Message):String = m match{  
    case send(t) => "Message sent!"  
    case received(t) => "Message Received"  
}
```

Using pattern matching to decode case classes is common practice and provides a flexible way to write functions that react according to the concrete instance of the

abstract class.

While I have mainly focused on discussing pattern matching on types you can also pattern match on input data such as numbers or characters.

### 3.5 Option type

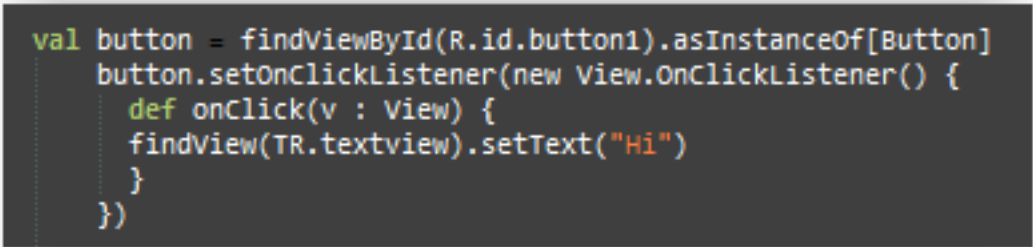
One useful feature of the Scala programming language is the use of Option types. This feature works in a similar way to Haskell's Maybe Monad resulting in the removal of the null pointer exception from occurring.

This is a common exception that often occurs in Java program development when an unexpected null is returned when data is required. As returning null values are heavily incorporated into Java programs, this approach has carried over into Android application development and has created the same issue for Android developers.

Scala excludes null values and instead implements the Option type. The Option type allows for the possibility of some data to be returned or none. This approach generally results in less error-prone code being written that has null pointer exceptions appearing unexpectedly. It also removes the need for try/catch blocks, with this resulting in cleaner code.

As of Java 8 option types are available in the Java language as well. They generally work in a similar way to Scala Options, but one of their failings is that they don't force the developer to ensure there is a procedure in place in the event of no data being returned, with this resulting in an exception being thrown.

### 3.6 Implicit conversion



```
val button = findViewById(R.id.button1).asInstanceOf[Button]
button.setOnClickListener(new View.OnClickListener() {
    def onClick(v : View) {
        findViewById(R.id.textview).setText("Hi")
    }
})
```

This image shows one way of creating a new button in an activity and setting the on click response for when the button has been pressed. Unfortunately, this approach is still verbose and so similar to Java it is essentially Java without the semi colons. Fortunately Scala does provide an interesting method of reducing the

verboseness of creating button a button on click event.

```
implicit def onClickListener(f: (View => Unit)): View.OnClickListener = {  
  new View.OnClickListener() {  
    override def onClick(v: View) {  
      f(v)  
    }  
  }  
}
```

Scala implements implicit functions that take implicit parameters as inputs. In this case the function takes an input value `f`, which is itself a function that takes a view and returns a unit. This approach allows for the creation of an implicit definition for the button on click event. In this example the button is passed a lambda function for the button `onClickListener`. This means that only one instance of the button listener needs to be defined and then whenever the developer wants to create a new button click event they can use this.

This approach reduces the amount of code required by a significant amount as it is no longer required to define the actions of a new listener for every button. Ideally, implicit click event listener can then be placed into a separate trait; this will allow it to then be utilized across the whole program when and where the functionality is required.

```
val button = findViewById(TR.button1)  
button.setOnClickListener((v : View) => {  
  findViewById(TR.textview).setText("Hi")  
}))
```

As you can now see all we need to do now when we want to define the buttons behaviour is write the code instead of the boilerplate required to create a new on click listener for each button whenever we want to define some behaviour. The implicit function is available to any button within scope and is ideally placed inside the trait so that it is reachable wherever needed in the project.

### 3.7 Traits

A Scala trait works in a similar manner to a Java interface, but differs in the level of abstractness that is offered. For example, in Java an interface will only contain the method names, the method inputs and the return type. This is probably as

abstract as is possible and is Java's answer to how to implement multiple inheritance.

By only allowing to directly inherit from one superclass at a time, the only way it is possible to simulate multiple inheritances is by incorporating interfaces and having to write the concrete methods in each class including that interface. This highlights the verbose nature of Java as a concrete implementation of the interface needs to be coded whenever a class implements them. Scala traits allow for a concrete method implementation to be defined and utilized wherever needed.

Scala traits are similar to interfaces in that they contain the method stubs, but differ in that they also allow the code for these methods to be given a concrete implementation. Then using the `with` keyword a Scala Class can implement as many traits as required. By using this approach to inheritance Scala Classes can include multiple traits in a single class and reuse their provided functionality whenever needed. If method name clashes are detected between the traits that a class is implementing, then they are given priority in order of right to left, so this is something a developer should always consider when including multiple traits in a class.

```
trait onClickTrait {  
  
  implicit def onClickListener(f: (View => Unit)): View.OnClickListener = {  
    new View.OnClickListener() {  
      override def onClick(v: View) {  
        f(v)  
      }  
    }  
  }  
}
```

As you can see I have now removed the implicit button click function and placed it inside its own trait. This means that even though I no longer have this method inside the class displayed below, I can still attach this to the buttons contained inside this class because I have included the `onClickTrait`.

This approach would be impossible to implement in Java because you could only extend one class and would be forced to write an interface without the concrete implementation of the functionality, with this forcing you to then write that functionality inside any class that uses the interface.

```
class helloWorld extends Activity with TypedActivity with onClickTrait
```



### 3.8 Futures and Promises

One interesting piece of functionality provided by the Scala language is the use of futures. A Scala future is a non-blocking piece of data that has yet to be determined. By stating that a value is a future, the program can continue to operate until the value is actually needed. By using futures the code is non-blocking as it isn't required to wait for the value to be resolved until it is absolutely needed.

### 3.9 Actors

While languages like Java feature thread-based concurrency Scala omits this approach in favour of the actor-based concurrency system. One of the major drawbacks of using mutable state and thread-based concurrency is that the developer has to spend large amounts of time reasoning about how to reduce issues such as deadlock or thread starvation from occurring.

By using the actor model approach to concurrency Scala allows the developer to write concurrent, non-blocking programs. The actor model features actors that carry out some predefined functionality. The actors each have an inbox of received messages and process these messages sequentially.

An actor has one main function that needs to be overridden. This function is the receive function which handles how the actor responds to different messages. Combined with case classes this is a powerful system for handling concurrency.

The Akka framework allows actors to be remotely distributed across multiple physical locations via the use of remote actors in a peer-to-peer fashion. This is one of the more complicated aspects to set-up on Android as it requires merging multiple Akka JAR file reference configurations into a single configuration file.

## 4 Type System

The true advantage of Scala comes in its rich and extensive type system. I believe one of the key aspects for why Scala could be a much more effective language when programming Android applications is the extensive type system it provides. In this respect Java simply cannot compete as its type system isn't as expressive in comparison to Scala's type system. While Scala allows for clear and elegant use of its type system, Java has a verbose imperative approach to its type system, with this often resulting in less elegant type declarations and code.

While Java 8 has certainly made a move towards implementing some of the most common examples of the functional paradigm, but at best these are simply the most common features of any language that has functional programming features. One example would be that Java has only just introduced the lambda function as an input for a method, while even C++ has featured lambdas since version 11.

The java type system is built around everything being a class and even in Java 8 no fundamental change has taken place to this idea. Take for example the lambda function, while on the surface it looks like you are just defining a function that is executed, in reality this code is converted into a class before being executed.

Scala has a much bigger and richer type system than Java. Scala provides a truly flexible type system that has much greater depth than Java and allows a skilled developer to create code of a much higher quality than Java. Some of the features of the Scala type system are listed below.

- Implicit Definition
- View Bound Type
- Variance Annotation
- Partial Application
- Type Inference
- Type Functions
- Higher Kinded Types
- Operator Overloading

## 4.1 Implicit Definition

## 4.2 Bounded Type

In Scala using bounded types a type can be specified as having some relation to another type. For example we have type A and type B. Using bounded types the developer can specify to the compiler that there is a relationship between type A and type B. This allows functions to have a certain type as an input, but then treat that input as if it is a different type to the input type.

This relationship between two types can be expressed using the symbol  $\leq$  for upper type bounds and  $\geq$  for lower type bounds. While this might seem like a simplified method of expressing relationships between two types, it is in fact a flexible

and powerful tool for declaring relationships between types in the function parameters. So for example, if the developer needs to state the relationship is that the function input type A is related in some way to type B, then the developer can simply write in the function parameters as `A <: B`. This means that when providing the function with its input it can receive an input of type A, but treat it as if its an input of type B.

Bounded types in this manner are how Scala provides its interoperability with Java. Using bounded types the Scala compiler can treat Java classes and code as if it is Scala code. When applying the example given to Java and Scala classes how works becomes clearer. By stating that the input type is type A, in this case a Java class, and then stating the relationship is `<:` to B

### 4.3 Variance Annotation

Scala features declaration-site variance, with this meaning that the variance of a type is stated in the type parameter. For example, we have an input of type `[T]`, in the type signature for this type we want to state that any input of type `[T]` is a subclass of type `[C]`. Using Scala declaration-site variance we can then explicitly state that type `[T]` is a subset of type `[C]` by adding a plus symbol. So now when stating the input of this function is `[T]` the developer would write `[T+]`. This means that any input of type `[T]`, is a subclass of t

### 4.4 Partial Application

### 4.5 First Order Functions

### 4.6 Higher Order Functions

Higher order functions come in one of three variations. The first variation is a function that has one of more of its inputs parameters defined as a function that returns some value. The second variation doesn't take a function as an input, but instead returns a function as it's output. The final variation takes one or more functions as an input parameter and returns a function as an output.

Two extremely common higher order functions are `map` and `filter`. Both these functions take other functions as inputs and then apply them to some supplied data. Usually these two functions are passed lambda functions as inputs, with this allowing the developer to state the function only when it is required. Alternatively, a type function or predefined function are capable of being passed as inputs.

```
def freq(f : Char => String => Int, c : Char, s : String) = f(s, c)
```

## 4.7 Type Inference

Scala features an advanced compile time type inference system. Advanced type inference like Scala provides lets the compiler calculate the type signature of variables. One of the benefits of type inference as good as Scals, is that nearly all bugs in the code are caught when compiling and not during runtime. A strong type inference system reduces the amount of required code as obviously type values dont need to be stated by the developer.

```
val str = String
```

```
String str = String;
```

Java in comparison requires the developer to explicitly state the type of the variable as the compiler isnt as strong at type inference as the Scala compiler.

Unfortunately the Scala compiler cannot infer the types in recursive functions.

## 4.8 Type Functions

When defining a type signature in Scala, we often see functions being passed as type parameters. One useful feature of the Scala type system is that the developer can actually define the function as a type and then when declaring a type parameter that is actually a function, the predefined function can be passed as the input. This is useful when multiple functions take the same function as a type parameter, as this function only needs to be defined once and can then be passed around to whatever input requires it.

## 4.9 Higher Kinded Types

### 4.10 Yield Operators

### 4.11 Operator Overloading

One interesting feature that Scals type system implements is the use of operator overloading. Operator overloading allows a developer to define a symbol as the name of a function. While the advantage of this may not be immediate apparent, it is a flexible system as it allows developers to still use a symbol-based approach when writing their functions.

For example, the developer creates a new object type called T. Objects of type T are in some way comparable to each other, although this has not currently been coded by the developer. Using implicit conversion the developer can then write a new function that takes two objects of type T and in some way compares them and

returns the result to the developer. Using operator overload the developer can name the implicit function `+`.

Now when the developer writes the functionality that requires two objects of `T` being compared to each other the only requirement is using the `+` symbol. The compiler will realise that the default `+` symbol function does not know how to compare two objects of type `T`, but an implicit function has been written for using the `+` symbol to compare objects of type `T`.

It should be noted that while operator overloading is a flexible system for defining symbol-based functions, it can often lead to some frustration. In some cases it can make the code harder to understand to other developers as in some instances using keywords is easier to reason about and understand than using symbols, this is especially apparent in larger programs which feature multiple operator overloads.

## 5 Kotlin

While Scala is evidently an extremely powerful and flexible language at present it has only found a small corner of the android development market to occupy. While my test case highlights the strengths that the language can bring to Android application development, without real support from Google it will be extremely difficult to find widespread use. Currently Scala on Android requires tweaking by the developer and this can have an impact on productivity, with hard to find errors when proguards tree shake removes a required class.

Kotlin is a relatively new language, with version 1.0 only recently being released, that is created by Jet Brains the creators of IntelliJ IDEA IDE. It is another language that runs on the Java Virtual Machine. This development is extremely important as not only do JetBrains offer easy integrations with their IDE and Kotlin, but also because Kotlin implements many features found in Scala, the language syntax offers a similar level of expressiveness.

While still being a relatively new language, having only just released version 1.0, it is an extremely interesting development. Kotlin is a language with similarities to Scala and could be considered Androids answer to Apples Swift language. Both language share a syntactic similarity with Scala and implement a large number of the features found in Scala.

One of the more important aspects of this development is that Android Studio is based on IntelliJ IDEA IDE. The reason this is important is that Jet Brains are providing as easy integration as possible with their IDE and Kotlin. One of the biggest barriers to using Scala on Android is the lack of support from Google, with

this requiring the developer to spend time tweaking the project structure to support the use of Scala on Android.

## **5.1 Comparison with Scala**

As stated Kotlin and Swift both share more than a passing resemblance to Scala, both syntactically and in their type systems.

## **5.2 Type System**

Its type system, while being expressive, is not as capable as Scalas type system. This may change in subsequent versions as new features are implemented into the language and others are deprecated. As Kotlin's type system matures it may reach a similar level of expressiveness as Scala.

### **5.2.1 Null values**

Kotlin has a different approach to dealing with null values than language likes Scala and Java. While Scala features option types for handling values that could be null, this type is simply a wrapper around a value and can still return null values.

Kotlin by default doesn't allow null values to be returned.

### **5.2.2 Type projections**

## **5.3 Java interoperability**

One of the main aims of Kotlin is to provide seamless integration with Java and all its libraries. One interesting way that Jet Brains have implemented this into the Kotlin language is by having their IDE automatically convert Java source code into Kotlin code. This is an excellent approach as it allows developers to quickly convert old Java code into Kotlin syntax without having to refactor all the code from one language into another. This ease of use when switching from Java code to Kotlin is a

## **5.4 Developer support**

As previously stated the developers of Kotlin are the same group that created the IDEA IDE, which is the base for Android Studio. Google dropped support for Android programming on the Eclipse IDE and are supporting Android Studio as the primary IDE for Android application development. Because of this

## 6 Project Reflection

Generally I am happy with the progress I have made during my study of Scala on Android. I was able to complete a working test case to a satisfactory standard and was able to provide an in-depth discussion of the Scala language features and of its type system.

It should be stated that I had to alter my application from a messenger app that made use of remote actors in favour of an encryption application that utilized PHP and a more traditional client/server relationship for moving data over the network. The main reason for this was the difficulty in setting up remote actors on Android.

Using proguard is vital when developing Android applications written in Scala. Its treeshake to reduce the number of classes and keep the APK size to a minimum is vital to reducing the memory overhead. Unfortunately, this also leads too hard to locate issues when attempting to utilize libraries like Akka remote actors, as vital classes that are required by these libraries can unfortunately be remove in the treeshake.

While I do believe that given more time I could have implemented remote actors, some developers have had success, I felt that I would be better spending my time on creating a more complete application, with good backend functionality and a good UI experience for the user.

I was able to implement the actor model inside my application to handle the internal concurrency. This highlights that even though the Akka framework is generally more suited to server applications, the actor model approach to concurrency is still a good fit for Android app development. As Android applications require good use of concurrency, the actor model provides an excellent method for creating concurrent non-blocking

Ideally in the future a more lightweight version of the actor model will be made available for use in Android development. With good support from Google using the actor model for concurrency could become the new standard for writing concurrent Android applications. In my opinion this is the likely future of concurrency on Android, but only if Google chooses to support it.

While I do still stand firm in my belief that Scala is a much more expressive and powerful language than Java, if I was to state my opinion on the future of Android application development I would place Kotlin as the most likely candidate to provide a true alternative to Java for Android development.

Kotlin for example has a relatively small size when included in Android applica-

tions, with this being the clear winner of the memory overhead provided by Scala on Android. Because of the smaller base library size the developer doesn't require proguard or its treeshake to remove classes and methods. By not require proguard to remove classes, this removes the trouble of classes relying on other classes that have been removed, with this allowing the developer to spend more time coding and less time fixing proguard related issues.

#### 6.0.1 PROS

#### 6.0.2 CONS

## 7 Conclusions

## References

- [1] J. Howie, *Generalised triangle groups of type  $(3, 5, 2)$* , <http://arxiv.org/abs/1102.2073> (2011).