Scala: Change My Mind

CMPT 383 Assignment 2

Alec Pershick

Introduction

For many, Java is considered one of the best programming languages out there. It has its limitations; however, and a rival language built from the ground-up has been increasing in popularity. With the rise of Big Data and Machine Learning in the past decade, Scala, short for Scalable Language, has been making an impact on the programming world. Based solely off the name, you can probably imagine that the main purpose of the language is scalability. For large-network or business-critical applications, Scala is the perfect option as it incorporates much of the largely used Java language but builds upon it in a highly scalable and efficient way. Although it is compiled to run on the Java Virtual Machine, it is quite different from the Java that you know and love. Throughout the rest of the paper, I will be persuading you to move away from your old outdated Java habits, and move to a scalable, more reliable, more productive language in Scala.

Paradigms

Scala's predominant paradigm is functional programming, although it also supports object-oriented programming as well. This is great for businesses or groups that already have software built with Java, because then as a developer you could introduce new elegant solutions using functional programming for a new module or program but still have the Java code working fluently in the system. One major benefit of its functional paradigm is the immutability aspect it provides. An immutable object is "thread-safe" which means different threads may access it without the risk of a deadlock or race condition preventing your code from executing properly. Apart from immutability, Scala provides almost everything you could want in a functional programming language. It provides syntax for defining anonymous functions, allows functions to be nested, supports currying, and has higher-order functions.

However, Scala can also be just as useful as an object-oriented language as well. It considers almost everything as an object like in Python, and has classes, methods, fields, and traits that it utilizes beautifully.

Scala is also statically typed, although unlike most statically typed languages, it does not expect you to provide redundant type information. Type inference provides all the needed type information needed unless specified in an error message or for increased code readability.

Parallelism

Concurrency is something that fits nicely with the Scala programming language in many ways. The main reason is that Scala provides a way to keep your code thread-safe by forcing you to code immutable objects. This eliminates many of the errors that usually come along with programming in parallel with threads. Scala has many options for concurrent programming, one of those being a Scala Future. Generally you would use futures when you want to compute something in the background in an efficient, asynchronous, non-blocking way.

Another way to run code in parallel is by using Akka. Akka is an Actor concurrency model where you define a set of workers that wait for messages (signals) and then act on them in a specified way. Akka is useful for when you want to set up a more robust background job processing framework.

The last Scala parallel computing method is to use stream processing. This is used when you have events in a stream that need to be continually processed in real-time. Database updates or click events on a website are some examples of use cases for stream processing. Major frameworks such as Apache Spark use stream processing for its cluster computing with big data and machine learning algorithms.

Compilation Process

Scala is compiled the same way as Java for the most part. The Scala compiler converts the source code into JVM bytecode and does not do much optimization. Once it is converted into bytecode it is basically just directly converted to machine code for the architecture it is being run on. This is known as Just-in-Time (JIT) compilation. The JIT in JVM does not optimize the code

as well as in other languages however, mainly because it has to be fast. In order to avoid recompiling the code, it only optimizes the most frequently executed parts of the code. Similarly to Haskell, it also implements some lazy evaluation in its compilation process.

Syntactic Sugar-free

I found the syntax quite fussy but also interesting in Scala. For example, case sensitivity is a huge deal in Scala. The entire language is case-sensitive (which most languages are) but something that is not as common is that the filenames must match the object's name in the program. This is not just convention; the compiler will spit the program back at you if they do not match. I had some trouble doing some demo programs where it took me quite a while to notice that was why it was not compiling. The weird thing is that it was not throwing any errors at me in the prompt. Not sure if that is common behavior but that could get annoying.

Conveniences

Although there are some annoyances with syntax, there are also some really nice things for Scala programmers to enjoy. One such thing is working with types. Declaring variables in Scala can be done by using either the *var* or *val* keywords. *Var* declares it as a regular mutable variable, whereas *val* is an immutable variable. *Var myVar :Int;* is a valid declaration of an integer variable in Scala, however we can reduce it to just *var myVar;* and let the type inference do its job. These can also be declared inside classes as fields which can be quite useful.

Comparisons to Java

There are quite a few similarities between Scala and Java, but many differences as well. One being type inference that was mentioned previously, others include nested functions, Domain Specific Language (DSL) support, traits, closures, and integration with concurrency and machine learning tools like Apache Spark.

There is also a major difference in average lines of code written in these languages. Figure 1 shows an example of a simple program written in both Java and Scala. The Java

implementation took around 50 lines of code, whereas the Scala took 12. This is not always the case, but generally Scala provides the ability to write more elegant, readable code than Java.

Cons

Are there any downsides to using Scala? Well, yes. For one, since it runs on the JVM it cannot implement any kind of true tail-recursive optimization. There are workaround for this, such as using *@tailrec* to reap some benefits, but not to the extent of other functional programming languages such as Haskell. Many programmers also consider Scala to be marginally harder to learn that other common languages, however I think it is mainly due to it not being as new or as highly used. The other issue is that inexperienced Scala programmers will not know the consequences with writing certain implementations of Scala code. For example, if I write a quicksort in Scala using an easy functional implementation, the performance will take a huge hit when compared to a typical Java quicksort. The problem is that it is easy to use certain "easy" solutions to problems that are not the most efficient solutions in Scala. However, the good news is that if you care about performance, you can actually turn your quicksort implementation into a more efficient version than Java using different methods in Scala as seen in Figure 2 and Figure 3.

Conclusion

Scala is a great language to learn in order to improve both your functional skills as a developer, and to improve or update existing Java code with smarter more elegant solutions. Even though Scala has been around for nearly 15 years, it feels like it is just beginning to grow and reach its potential in the new age where data science is rising exponentially every year. Some major companies are incorporating Scala code into their software now including LinkedIn, Amazon, and Twitter as well as many others. It does have its work cut out for it though, as other functional languages are on the rise including Kotlin who is on an upward swing in the last year or so. I will be interested to see who reigns supreme in the functional world; I hope it is Scala.

Appendix

Figure 1

Java version:

```
public class User {
    private String name;
    private List<Order> orders;

    public User() {
        orders = new ArrayList<Order>();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Order> getOrders() {
        return orders;
    }

    public void setOrders(List<Order> orders) {
        this.orders = orders;
    }
}
```

```
public class Order {
    private int id;
    private List<Product> products;

    public Order() {
        products = new ArrayList<Product>();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public List<Product> getProducts() {
        return products;
    }

    public void setProducts(List<Product> products) {
        this.products = products;
    }
}

public class Product {
    private int id;
    private String category;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
```

```
    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }
}
```

Now the Scala version:

```
class User {
    var name: String = _
    var orders: List[Order] = Nil
}

class Order {
    var id: Int = _
    var products: List[Product] = Nil
}

class Product {
    var id: Int = _
    var category: String = _
}
```

Figure 2

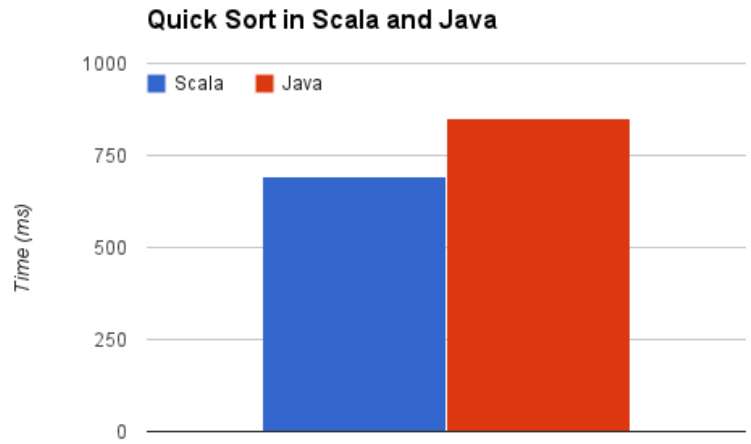

Figure 3