On Scala
(or: Why Static Typing Doesn't Have to Suck)

By Michael Neale & Thomas Lee

Abstract

In this paper we discuss Scala as an alternative to increasingly popular, dynamically typed JVM languages such as JRuby and Groovy. There are developers who are rightly concerned about the move to dynamically typed languages for their enterprise projects. With impressive support for both object-oriented and functional paradigms, Scala provides many of the expressive benefits of a dynamically typed language with the reassurance and safety of a traditional static type system. We will compare Java with Scala and, in turn, Scala with Ruby to demonstrate that static typing is not itself a constraint on nor an indicator of the expressive capabilities of a programming language.

Introduction

The relative stagnance of Java The Language has the wider community asking "What's next?". What's interesting is that many Java developers are -- more often than not -- finding solace in the arms of dynamic languages like JRuby, Groovy, Jython and Clojure. The dynamic nature of these languages often allows developers to better express their immediate intention without having to work around the language.

While these languages have often met with success, there are many lamenting the absence of IDE tooling for these dynamic languages, questions about the long-term maintainability of projects using dynamic languages and the difficult-to-debug errors that can occasionally surface when one relies on a dynamic type system.

Scala is, in the authors' opinion, the answer to many of these concerns.

What is Scala?

Scala is an statically typed, expressive, multi-paradigm language for the JVM that has steadily been gaining traction over the past few years. It is much less verbose than Java while still enforcing type correctness -- something often missed by statically inclined developers in a dynamic world. It supports imperative, object-oriented and especially functional paradigms and comes with a standard library that augments the Java standard library. In addition to all of this, any existing Java code can be reused from within Scala without any special effort -- and since Scala code compiles down to JVM bytecode, with some care it is possible to use Scala classes from within Java code too.

Before We Start: The "Unit" Type

Before we get too deep in the examples, it's probably worth introducing something in Scala that may be unfamiliar to Java and/or Ruby developers:

When a method declaration in Scala indicates that its return type is a "Unit", it is analogous to a Java method declaration return type of "void". The details of this and the distinction between void and Unit are outside the scope of this paper, but the distinction can be safely ignored for all the examples we present here.

Comparing and Contrasting Java to Scala

The original creator of Scala, Martin Odersky, was involved in early efforts to "modernise" Java (pre Java 5), which eventually turned into Java 5 and included a "dumbed down" implementation of generics. Martin was frustrated with the limitations inherent in the Java language specification, and decided a "fresh start" with a modern language, but with similar goals (and identical runtime infrastructure) was warranted - and thus Scala was born.

Some of the features of Scala that are often viewed as "missing" from Java include:

- Anonymous functions / closures
- Type-safe pattern matching
- Everything is an object (1.floatValue will actually compile!)
- Limited type inference
- Operator overloading (operators are just methods!)

Without the requirement of source-level backwards compatibility that has prevented Java from taking radical steps forward, Scala has language-level support for features that the Java community have been requesting for years.

First Class and Anonymous Functions vs. Anonymous Inner Classes

Saying that a language supports first class functions is indicates that the language allows us to use functions (and, in Scala, methods) as values. That is, we can assign functions to variables or pass them as arguments to other functions. Why is something like this useful? First, let's look at a simple function that accepts another function as an argument.

```
class Node(id : String, children : List[Node]) {
  def id() = this.id
  def children() = this.children
}

def visit(nodes : List[Node], visitorFunction : (Node) => Unit) = {
  for (node <- nodes) {
    visitorFunction(node)
    visit(node.children(), visitorFunction)
  }
}
```

Since Scala functions can also be used as values, we need to be able to express function type signatures if we're going to be able to do any sort of useful type checking on functions. To that end, we can indicate that values passed in as the second parameter to the *visit* method (*visitorFunction*) must be functions that accept a Node as an argument and returns Unit. i.e.:

```
(Node) => Unit
```

So it's now obvious that the *visit* function takes a list of nodes and a function that accepts a single node as a parameter. What can we do with a function like *visit*?

```
def printId(node : Node) = println(node.id())

visit(listOfNodes, showId)
```

This will print (to the console) the id of each node and its children. What if we want to use this same function in an SWT or Swing application to add the nodes to a drop down list using the same *visit* algorithm?

```
def addToComboBox(node : Node) = someCombo.add(node.id())
```

```
visit(listOfNodes, addToComboBox)
```

For simple tasks we can also make use of Scala's language-level support for *anonymous functions*, which we'll cover later on in this paper.

Actors vs. Classical Threads & Locking

An increasingly important topic is that of safe concurrent programming techniques, due to modern architectures "scaling horizontally" in terms of the number of processors or cores available to an application. Java has excellent support for multi threaded programming using shared memory. The main issue with this style of programming is that it very very hard to do correctly. Libraries have been built which provide a "share less" architecture (such as java.util.concurrent) which certainly help in this area. A popular well supported style of concurrency is based on "message passing" (made popular in recent times by Erlang).

Scala builds on the foundations in the JVM but by using its preference for immutable data structures, it allows other models for concurrency. One of the most successful is the "actor" framework: it is not a special syntax or feature of scala, so much as a library.

The actor library concept was "borrowed" from Erlang: think of actors as a process which can consume messages from an inbound mailbox:

```
//how to create a new actor instance:
val myactor = actor {
                loop {
                  receive {
                    case MyMessage(content) => println(content)
                    case i: Int => println("can match on type")
                    case _ => println("catch everything else !")
                  }}}
```

Pattern matching is used to match inbound "messages" (messages are just objects, immutable by convention) - each actor instance has its own inbound mailbox. Actors may be invoked in a async/non blocking fashion (using "!") or in a blocking fashion (using !?). This style of actor shown is able to use a thread pool where available, and a thread does not have to be allocated per actor, allowing thousands of actors to exist (they are not a resource constrained by the system, unlike threads). Actors can also be used to "gate keep" access to a single resource, as an alternative to locking/synchronization, and results in code with less dead locks and similar issues.

```
myactor ! MyMessage("hello world ") //asynchronous - MyMessage is a case
class
myactor !? 42 //call and await a response - with optional timeout
```

Traits and Type Aliases vs. Interfaces

Traits have a lot in common with Java interfaces. In fact, one can use a trait in Scala exactly as they would a Java interface. The big differentiating factor of traits is that trait methods can be implemented in the traits themselves. That means that we can actually provide *functionality* to classes inheriting traits, similar to abstract classes in Java. However, traits cannot define data: they may only call their own methods or the methods of the traits that they inherit from. By implementing traits in this way, Scala gives us a relatively safe way to support multiple inheritance and inheritance of implementation -- if you're into that sort of thing!

Traits can also be used as needed instead of tying them to your class declaration:

```
class Person(firstName : String, lastName : String) {
  def name() = firstName + " " + lastName
}
```

```
class Gibbon(name : String) {
  def name() = this.name
}

def showName(thingy : HasName) = println(thingy.name())

val tom = new Person("Tom", "Lee") with HasName
val mike = new Person("Michael", "Neale") with HasName
val fluffy = new Gibbon("Fluffy the Gibbon") with HasName

List(tom, mike, fluffy).foreach(showName(_))
```

Note that although we use the *"new ... with ..."* syntax here, it's entirely possible to use traits as you would base classes / interfaces in Java i.e. the *"... extends ..."* syntax.

Another option that Scala provides is the concept of *type aliases*. Often we don't really care about the fact we're implementing an interface (or trait!) called X or Y, we just care about a method or two that the said interfaces provide to us. Enter type aliases! Type aliases, when used as type declarations for method parameters or generic collections, allow us to say *I don't care about the type of the object, so long as it gives me these methods.* This is very powerful stuff, allowing us to bypass a lot of the "blah blah blah" typically required by a statically typed language like Java without circumventing the cushion of our type system. We'll cover type aliasing in an upcoming chapter.

Comparing and Contrasting Scala with Ruby

Languages like Ruby and Groovy have become the poster children for alternative JVM languages due to their inherently expressive nature. Quite often this "expressiveness" is attributed to dynamic languages in general, but a language like Scala can offer the best of both worlds.

Duck Typing

The typical approach to managing the cognitive load imposed by dynamic languages is the application of *duck typing,* by which the developer effectively makes some implicit, reasonable assumptions about the properties exhibited by the objects being passed to the method he is implementing.

```
class Duck; def quack(); puts "Quack"; end; end
class Goose; def quack(); puts "Honk"; end; end
class Tricycle; def applyBrakes(); puts "Screeech!"; end; end

def quack(quackable); quackable.quack; end

quack(Duck.new) # OK
quack(Goose.new) # OK
quack(Tricycle.new) # runtime error
```

In Scala, we can do the same thing, except that we are explicit about the assumptions we're making. The sort of type checking provided by Scala in scenarios like this can catch the silly mistakes that keep the statically inclined away from dynamically typed languages:

```
class Duck {
  def quack() = println("Quack")
}
```

```
class Goose {
  def quack() = println("Honk")
}

class Tricycle {
  def applyBrakes() = println("Screeech!")
}

def quack(quackable : {def quack() : Unit}) = quackable.quack()

quack(new Duck) // OK
quack(new Goose) // OK
quack(new Tricycle) // compile time error! (Tricycle does not have a quack()
method!)
```

Note that the length and noise of these two examples are actually quite similar -- but in Scala, you are bound to discover this mistake long before the code ever reaches your client. In the Ruby example, you can never be as certain. If the inline type information seems a little ugly, you can use *type aliasing* like so:

```
type Quackable = {def quack() : Unit}
```

With this new Quackable type alias, we can then redefine the quack method like so:

```
def quack(quackable : Quackable) = quackable.quack()
```

The semantics are exactly the same as before: *quack* will accept any object that provides a quack() method.

map, filter and fold*

Rubyists are likely to be familiar with the *map*, *delete_if* and *inject* methods available for collection objects like Array. Type-safe Scala equivalents are available in the form of *map*, *filter* and *foldLeft*. Here are some examples of these functions using Scala's syntax for *anonymous functions*:

```
List(1, 2, 3).map((x : Int) => x + 5)

List(1, 2, 3).filter((x : Int) => x != 2)

List(1, 2, 3).foldLeft(1)((x : Int, y : Int) => x * y)
```

As you can see, anonymous functions can create a little visual noise if not used with caution. The wildcard operator can be used to create anonymous functions with less noise:

```
List(1, 2, 3).map(_ + 5)

List(1, 2, 3).filter(_ != 2)

List(1, 2, 3).foldLeft(1)(_ * _)
```

Ruby's "open objects" vs. Scala implicits

In Ruby a common practice is to dynamically enhance a class i.e. add to or change the behaviour of objects without modifying the original class declaration. While Scala doesn't permit this sort of promiscuity to the extent allowed by such languages, it does have limited support

for things like this. Scala provides a feature called *implicits* to allow developers to add methods to objects without necessarily needing access to the original class declaration. C# provides similar functionality with its support for *extension methods*.

To use an implicit define a function similar to the following:

```
implicit def myWrapper(orig: OriginalClass) = new EnhancedClass(orig)
```

In the above definition, a function called "myWrapper" is created which takes an instance of OriginalClass, and produces an instance of EnhancedClass (taking in the "orig" instance). As this is an implicit function, you don't invoke it directly, but let the compiler weave it in. When this definition is in scope (eg defined in the same class, or imported), then you are able to write code that appears as if you are invoking a method on OriginalClass that does not exist ! The compiler allows for this by detecting there is no such method, but then noting that in scope there is an implicit conversion that would provide the desired method (in the above case by wrapping it):

```
class OriginalClass(val name: String, val age: Int)
class EnhancedClass(val orig: OriginalClass) {
  def amIHot() = orig.name == "mic" && age < 42
}

implicit def myWrapper(orig: OriginalClass) = new EnhancedClass(orig)

val tom = new OriginalClass("tom", 25)
//
// Because the myWrapper implicit is in effect, this works!
//
tom.amIHot()
```

Again, we must stress that implicits are checked at *compile time*!

Other Neat Scalaisms

The Scala standard library includes a bunch of other stuff that we really don't have time to cover here, including:

- Parser combinators
- XML constructs
- A wrapper for the Swing UI toolkit

Also worth noting is scalaz, which provides a bunch of useful stuff: http://code.google.com/p/scalaz/

Conclusion

Hopefully you've seen enough in this paper to want to give Scala a go. It's a fantastic compromise for those who are frustrated with Java's shortcomings but don't feel comfortable about making the jump to a dynamic language. Scala is truly a breath of fresh air for the Java platform and it's proof that Java's limitations aren't so much the problems of the JVM as the Java language itself.