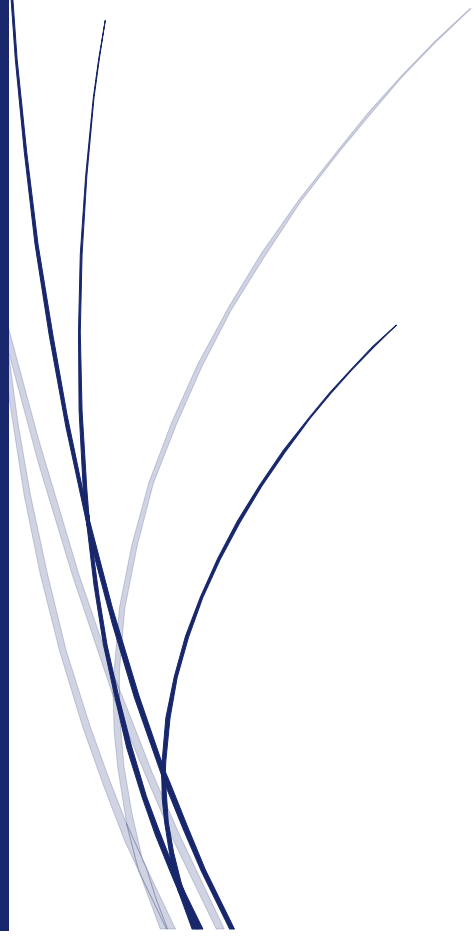
Tallinn University of Technology
Department of Computer Engineering

30.11.2015

Scala

A Scalable language

Essay
Analysis of programming languages
(IAG0450)



Petra Krnáčová
156604IV

CONTENTS

INTRODUCTION

Scala programming language is quite new language, its design started in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland by Martin Odersky, who had previously worked on current generation of javac, the Java compiler [1]. It was publically released in 2004 for the first time.

It is a programming language for general software applications which is statically typed and multi-paradigm programming language that runs on Java virtual machine [2]. Multi-paradigm programming language means that it allows programmers to mix multiple programming styles such as object-oriented, imperative and functional programming.

Originally, it has two compiler implementations. First implementation translates Scala into Java byte code, second into .NET [3]. However .NET implementation stopped being supported in 2012 [4] and in 2015 first no experimental version of the Scala to JavaScript compiler was released [5]. Current stable release of Scala is 2.11.7. which was released in June 2015 [4].

Because of language interoperability with Java, Java libraries can be used directly in Scala source code and vice-versa.

Nowadays, Scala is gathering the programmers and the companies. In 2012, it was voted the most popular JVM scripting language at JavaOne conference. Programming language popularity measures place Scala in the first 50 most popular languages (TIOBE index 2013 – 31st, RedMonk 2013 – 12th). Some interesting works which use Scala are: Twitter, Coursera and LinkedIn [4].

ESSENTIAL CHARACTERISTICS

WELL DEFINED SYNTACTIC AND SEMANTIC DEFINITION OF LANGUAGE

Even though Scala is not “officially” standardized, like for example C++ with version specifications as: ANSI C89, ISO C90, ISO C99, ISO C11 by ISO or ANSI authorities. It is de facto (as Java) standardized by its own Scala Language Specification (SLS) [23].

All syntax of language is represented in EBNF notation and then semantic is described in detail.

A lot of Scala’s design decisions were inspired by shortcomings of Java. Its design was also inspired by languages such as Eiffel, Erlang, Haskell, Lisp, Pizza, Standard ML, OCaml, Scheme, Smalltalk and Oz [4].

Let’s examine some differences between Java and Scala according to [4].

- Scala does not require semicolons to end statements.
- Value types are capitalized.

- Parameter and return type follow, as in Pascal.
    - `num: Int`
- Local or class variables must be preceded by `val` – immutable or `var` – mutable variable
- Type cast operator is different than in Java - `foo.asInstanceOf[Type]`
- Methods can be called without class reference operator `.` and parentheses
    - `thread.send(signo)` has the same meaning `thread send signo`
- Array references are written like functions calls
    - `Array(i)`
- Generic types are written as `List[String]` instead of `List<String>`
- Scala has singleton class `Unit` instead of pseudo-type void
- Methods definitions are preceded with keyword `def`
- Scala has no `static` variables or methods – instead it uses singleton objects
- Default visibility in Scala is public – it doesn't have `public` keyword
- Parameters from Scala class constructor are saved as class parameters, it is not necessary to declare them explicitly
- For expressions are similar to Haskell or Python, usage of yield makes a collection generator from for loop
    - `val s = for (x <- 1 to 25 if x*x > 50) yield 2*x`
    - `Result - Vector(16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50)`

Because Scala allows functional programming, we can find these functional tendencies in it:

- **No distinction between statements and expressions** – everything is an expression
- **Type inference** – compiler can deduce the type of variables, function return values and other expressions, so the types can be omitted
- **Anonymous functions** – similar to lambda functions in Java 8, you can create inline function without name it
- **Immutable variables and objects**
- **Lazy evaluation** – Scala evaluates expressions as soon as they are available, it is possible to change this behavior by using a keyword `lazy`
- **Delimited continuations (since 2.8)**
- **Higher-order functions**
- **Nested functions**
- **Currying** - evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.
- **Pattern matching** - can be viewed as an extensible version of a switch statement, where arbitrary data types can be matched (rather than just simple types like integers, booleans and strings), including arbitrary nesting.

Scala is also an object-oriented language in the sense that every value is an object. Data types and behaviors of objects are described by classes and traits. Traits are Scala's replacement for Java's interfaces. Traits are similar to mixin classes in that they have nearly all the power of a regular abstract class, lacking only class parameters (Scala's equivalent to Java's constructor parameters), since traits are always mixed in with a class.

RELIABILITY

Scala programs are running on JVM, which means that it's machine independent language, which means that Scala is reliable regarding to machine dependent errors. It also uses features used in Java such as garbage collector which makes Scala reliable regarding to wrong memory management errors.

Scala is statically typed language which means that type checking is done by the compiler before running the program. Its immutable variables and objects increase the reliability of programs too.

It also has a great support for runtime checking. Same as in Java, Scala uses exceptions to handle the errors. But differently to Java, it only has unchecked exceptions – even for I/O exceptions – so it totally depends on programmer which exceptions he will handle. It makes him more vulnerable to make a mistake.

Exception can be handled in typical try-catch-finally block, or it is possible to use Scala's `control.Exception`.

Example usage [23]:

```
val x1 = catching(classOf[MalformedURLException]) opt new URL(s)
```

But Scala has more error handling mechanisms [24]:

- Use `Option` when you are not sure if an instance of object will be returned. It returns an instance of `Some(A)` if `A` exists, or `None` if it does not.
- Construct `Either` is a disjoint union construct. It returns either an instance of `Left[L]` or an instance of `Right[R]`. It's commonly used for error handling, where by convention Left is used to represent failure and Right is used to represent success.
- Construct `Try` is similar to `Either`, but instead of returning any class in a Left or Right wrapper, it returns `Failure[Throwable]` or `Success[T]`. It's an analogue for the try-catch block: it replaces try-catch's stack based error handling with heap based error handling.
    - Don't confuse it with try-catch block
    - Example: (symbol <- is used in for-construction to separate pattern from generator.)

```
val sumTry = for {
   int1 <- Try(Integer.parseInt("1"))
   int2 <- Try(Integer.parseInt("2"))
} yield {
   int1 + int2
}
```

Although Scala provides a lot of mechanisms to check errors, so it can be considered as a reliable language, it also has some features such as implicit conversions, or not implemented null-safety (its implementation was abandoned because of recommend usage of own `Option` classes [16]) which can violate reliability.

FAST TRANSLATION

Speed of translation is one of the greatest weakness of Scala programming language. As I previously mentioned, Scala is compiled to Java byte code and run on JVM, but according to Martin Odersky [6], Scalac (Scala's compiler) manages about 500-1000 lines per second which may seem that it's 10 times slower than Javac (Java's compiler) does, but 1000 lines of code written in Scala correspond to about 2000-3000 lines of code written in Java, which means that in Scala there is more functionality per line.

Why the compilation is slow?

- Scala is statically typed programming language.
- Type inference is costly.
- Type checking is occurring twice – once according to Scala's rules, second time after erasure according to Java's rules.
- There are about 15 transformation steps to go from Scala to Java.
- Optimization.
- Startup of the compiler Scalac is slow (4-8 seconds) because of loading a huge amount of classes (because of complexity of the language) and searching for classpath for all root packages and files.

A solution for improving the speed of compilation is usage of fsc (Fast offline compiler) which, as is written in the description: *"The fsc tool submits Scala compilation jobs to a compilation daemon. The first time it is executed, the daemon is started automatically. On subsequent runs, the same daemon can be reused, thus resulting in a faster compilation. The tool is especially effective when repeatedly compiling with the same class paths, because the compilation daemon can reuse a compiler instance."* [7] Fsc plugin can be integrated into most of the commonly used IDE's, such as NetBeans, Eclipse and IntelliJ.

EFFICIENT OBJECT CODE

Efficient object code means that performance of the program will be good (fast execution, small memory consumption). Scala compiler emits Java byte code, but it doesn't mean that the efficiency characteristics must be necessarily the same as Java has. It depends on how effectively the Scala compiler transforms source code into Java byte code. Then VM is a limitation, because it converts Java byte code into machine code for the computer

architecture on which it is being run (just in time compilation), so on this level Java and Scala are the same.

The level of code optimization is low with just in time compilation, because it has to be fast. In the moment, JVM uses HotSpot dynamic compilation. Rather than convert all byte codes into machine code before they are executed, HotSpot first runs as an interpreter and only compiles the "hot" code - the code executed most frequently. Then, it optimizes only the parts of the code which are executed frequently, to avoid recompiling. That means, that a program could have different performance each time it is run (whether the particular code was optimized in between the runs or not) [9].

So let's go back to the Scala compiler, because the major differences between Java and Scala can be found there.

"The Scala compiler is complementing the traditional optimizations with a boxing optimization (that removes unnecessary boxing and unboxing operations), a type-propagation analysis (that obtains more precise types for values on the stack and local variables) and a copy-propagation optimization (that has a simple heap model handling common object patterns like boxed values and closures)." [10]

Benchmarks [11], [12] shows that performance of Scala and Java is similar, depends on the task and quality of source code, even though both languages allow you to write a program with poor performance if you do not understand syntax, semantic and the optimization process well.

Is Scala efficient? Let's compare with another languages like C++. Note that current results hardly depends on quality of implementation of algorithm.

Task: Computation of k-nucleotide frequencies [11]

Language	Time (in s)	Memory (in KB)	CPU usage
Scala	**6.77**	223 624	**20.28**
C	12.17	189 420	36.18
C++	7.16	**157 828**	24.08
Java	11.29	**1 118 288**	38.66
Ada	11.91	278 332	25.60
Fortran	**23.90**	194 140	**61.70**

Task: Fannkuch-Redux benchmark [11]

Language	Time (in s)	Memory (in KB)	CPU usage
Scala	15.23	**36 820**	59.62
C	**9.16**	3 624	**35.83**
C++	13.12	**2 032**	51.38
Java	**17.41**	33 032	**68.64**
Ada	11.25	4 116	44.84
Fortran	13.98	10 536	55.74

Task: Reverse complement of DNA

Language	Time (in s)	Memory (in KB)	CPU usage
Scala	**1.36**	**477 268**	1.80
C	**0.50**	251 028	**0.76**
C++	0.58	**214 852**	0.96
Java	1.27	315 296	**2.68**
Ada	0.78	201 036	0.92
Fortran	1.01	248 636	1.01

Task: Hans Boehm's GCBench

Language	Time (in s)	Memory (in KB)	CPU usage
Scala	13.65	**518 700**	19.53
C	**3.25**	**156 764**	10.17
C++	**37.97**	199 720	**37.94**
Java	5.67	356 656	**8.00**
Ada	5.37	179 164	17.86
Fortran	6.02	184 420	19.17

ORTHOGONALITY

In the book The Art Of Unix Programming [13] orthogonality is described such as: "*Orthogonality is one of the most important properties that can help make even complex designs compact. In a purely orthogonal design, operations do not have side effects; each action (whether it's an API call, a macro invocation, or a language operation) changes just one thing without affecting others.*"

It also means, that you are able to use various language features in arbitrary combinations with consistent results [14]. Great example of violation of orthogonality is that in C returning array from function is permitted, but passing the array as an argument is allowed.

Not propagation of side effects in programming languages is typically achieved by separation of concerns and encapsulation [14].

Even though Scala is very complex language, which provides lot of different features and constructs and the syntax of Scala is flexible and may be hard to master, it is also quite general and orthogonal. Martin Odersky described in video [15] that Object-Oriented paradigm and Functional Programming paradigm are orthogonal to each other, and Scala unifies them into Object-Functional paradigm.

It also means, that features of functional programming and features of objected-oriented programming don't affect each other in the way that functional language isn't less functional because it's also object-oriented and vice-versa. But it is flexible in the way, that it allows the programmer to combine both styles as he wants without inconsistent results.

Martin Odersky also said that it was one of the main goals in designing Scala to make it orthogonal, complex, flexible and compact [16]. One of the Scala features strongly connected with orthogonality is that Scala is immutable. Compactness in Scala could be demonstrated in contrast of Java, as I previously mentioned, Java program with 2000-3000 lines could be reduced by usage of Scala into 1000 lines of code. On the other hand some condensed statements are hard to read and understand for average programmer.

In the book Programming in Scala [17] you can find a definition of Scala as a language with high cohesion and orthogonal abstractions. One of the features for sustaining orthogonal abstraction is **mixin-class** composition [18]. It is impossible to combine functionality of two classes into new one with single inheritance and interfaces. Scala's mixin-class composition allows reusing of the delta of a class definition (all new definitions that are not inherited). Keyword **trait** is used to define a class which can be used as a mixin.

Example [18]:

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
```

```
}

trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) { while (hasNext) f(next) }
}

class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s charAt i; i += 1; ch }
}

object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator
    val iter = new Iter
    iter foreach println
  }
}
```

"The Iter class in function main is constructed from a mixin composition of the parents StringIterator and RichIterator with the keyword with. The first parent is called the superclass of Iter, whereas the second (and every other, if present) parent is called a mixin." [18]

That behavior suppress redundancy which is not at all orthogonal.

MACHINE INDEPENDENCE

Because Scala's source code is translated into Java byte code and run in JVM, it is also platform independent language as Java is. Even though JVM needs to be installed before running an application, source code of Scala application is the same for every platform on which it will be run, because JVM provides a bridge between the code and underlying platform. JVM acts like a virtual platform on which the code is executed.

DESIRABLE CHARACTERISTICS

PROVABILITY

Provability (automatic proof that written application works correctly or not) is one of the characteristics that modern programming languages hardly achieve because of their complexity and freedom given to programmers. In Scala it's especially hard because of its syntax flexibility and multi-paradigm.

So as in the many programming languages (like Java), to know if the program works as intended, it is needed to perform some tests.

GENERALITY

Scala is a general purpose programming language. It is possible to write application for wide range of domains because of its flexible syntax, Java interoperability, direct XML support and support of both functional and object-oriented programming with all theirs benefits.

Scala has good support for abstraction which directly supports generality (if we see generality as a combining closely related constructs into a single more general one), for example it allows implicit conversions.

CONSISTENCY WITH COMMONLY USED NOTATIONS

Scala notations are inspired by notations which can be found in programming languages such as C, Java, Lisp, Python or Haskell [4]. Main differences between Scala's and Java's syntax (notations) were discussed in the chapter Well defined syntactic and semantic definition of language.

Guide for moving from Scala to Java can be found there: http://techblog.realestate.com.au/java-to-scala-cheatsheet/

Notations are logical and consistent. Writeability is good, although syntax constructions are more condensed than Java has, and it take some time to move from Java to Scala, Scala rewards programmer with more complex and more abstract constructions.

But programs in Scala could be hard to read (readability is not the most powerful feature of Scala) because it representation could be far from used pseudo-code notations.

SUBSETS

In Scala, it is possible to create different language subsets. One possibility is to restrict all Java (and XML) constructions and libraries. Then, because Scala is multi-paradigm language, it is possible to use only functional programming constructs or object-oriented programming constructs depending on a task.

UNIFORMITY

Simple definition of uniformity is that language constructs with similar meanings should look similar and language constructs with different meanings should look different [19].

Scala favors uniformity in for example while / for loops, in contrast to the Pascal, where while and repeat loops are not uniform, because repeat does not require begin-end keyword and in treating collections of primitives.

It also supports uniform access principle that states that variables and parameterless functions should be accessed using the same syntax. "*Scala supports this principle by not allowing*

parentheses to be placed at call sites of parameterless functions. As a result, a parameterless function definition can be changed to a val, or vice versa, without affecting client code." [21]

But because Scala is very flexible and complex language, which supports interoperability with Java it also has some non-uniform concepts.

Good example is allowed usage of Java keyword null, although Scala has its own idiom Option class [20]. Instead of returning one object when a function succeeds and null when it fails, function should instead return an instance of an Option, where the Option object is either: an instance of the **Some** class or an instance of the **None** class. None class makes null unnecessary, redundant and the construct with similar meaning looks different.

Example [20]:

```
def toInt(in: String): Option[Int] = {
    try {
        Some(Integer.parseInt(in.trim))
    } catch {
        case e: NumberFormatException => None
    }
}

toInt(someString) match {
    case Some(i) => println(i)
    case None => println("That didn't work.")
}
```

EXTENSIBILITY (EASE TO ADD FEATURES)

Scala is an extensible (scalable) language. It allows the programmer to define new types inside the objects (type definition), extend with libraries, both Java and Scala libraries, because of interoperability, it supports direct usage of XML syntax in regular Scala's source code.

Usage of SugarScala framework [22], which enables syntax extensibility for Scala, makes it usable for domain-specific languages embedding.

SUMMARY

Scala is new and progressive language, which means that backward compatibility is not assured, but bugs are progressively fixed and new features are added. It is running on JVM and supports Java interoperability. Because it is multi-paradigm language and has flexible syntax with good level of abstraction it gives lot of freedom to programmer to write his applications. Though, it has well defined syntax and semantic, it is reliable, orthogonal, scalable and extensible language. Harder readability (linked with complexity and abstraction) and slow translation could be considered as main flaws of this language.

BIBLIOGRAPHY

[1] https://en.wikipedia.org/wiki/Martin_Odersky

[2] https://dzone.com/articles/moving-java-scala-one-year

[3] http://www.artima.com/weblogs/viewpost.jsp?thread=163733

[4] https://en.wikipedia.org/wiki/Scala_(programming_language)

[5] http://www.scala-lang.org/news/2015/02/05/scala-js-no-longer-experimental.html

[6] http://stackoverflow.com/questions/3490383/java-compile-speed-vs-scala-compile-speed/3612212#3612212

[7] http://www.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/tools/fsc.html

[8] http://www.scala-lang.org/files/archive/spec/2.11/

[9] http://www.ibm.com/developerworks/library/j-jtp12214/

[10] http://wessexacademy.ch/wp-content/uploads/2013/09/scala_paper.pdf

[11] http://benchmarksgame.alioth.debian.org/u64q/scala.html

[12] http://shipilev.net/blog/2014/java-scala-divided-we-fail/

[13] http://www.amazon.com/Programming-Addison-Wesley-Professional-Computng-Series/dp/0131429019

[14] https://en.wikipedia.org/wiki/Orthogonality#Computer%5Fscience

[15] https://www.youtube.com/watch?v=01rXrI6xelE

[16] http://www.slideshare.net/Odersky/scala-evolution

[17] https://books.google.ee/books?id=AnurBQAAQBAJ&pg=PR15&lpg=PR15&dq=scala+orthogonal?&source=bl&ots=7K8Dh7r3Y_&sig=k-q5DwFfSHrRJNV7HRsLyix7LEE&hl=sk&sa=X&ved=0ahUKEwjireHq2KHJAhUCWCwKHWv6Ch84ChDoAQgmMAE#v=onepage&q=scala%20orthogonal%3F&f=false

[18] http://www.scala-lang.org/old/node/117

[19] http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/comparing/

[20] http://alvinalexander.com/scala/using-scala-option-some-none-idiom-function-java-null

[21] http://docs.scala-lang.org/glossary/#uniform-access-principle

[22] http://www.informatik.uni-marburg.de/~seba/teaching/thesis-jakob.pdf

[23] http://www.scala-lang.org/api/current/index.html#scala.util.control.Exception$