# Natural Language Programming Analysis in Scala

Éric Zbinden
`eric.zbinden@epfl.ch`

Supervisors:
Philippe Sutter(`philippe.sutter@epfl.ch`)
Philipp Haller(`philipp.haller@epfl.ch`)
Prof. Viktor Kuncak(`viktor.kuncak@epfl.ch`)

EPFL
Laboratory for Automated Reasoning and Analysis (LARA)
`http://lara.epfl.ch/`
Programming Methods Laboratory (LAMP)
`http://lamp.epfl.ch/`

January 9, 2012

## Contents

## 1 Introduction

...
For Java this was done by

## 2 Project Overview

The main idea of this project is to apply an analysis similar to Høst and Østvold [5] but to Scala. Scala contains possibilities that JAVA don't contains; functional programming, local import. So I tried to focus on theses particularities as much as possible.

## 3 Implementation

The first thing is to retrieve information about a code. The best way is to create a plug-in for the Scala compiler.

### 3.1 Compiler plug-in

My plug-in, named Scala-names, is inserted into the compiler phases right after phase 7: *refchecks*. It use the abstract syntax tree produced in previous phases to extract all objects named by the programmer. It could be variables names, objects names, methods names, parameters names or types names.

By lack of time, the analysis is run only on method names. But the plug-in could easily be improved to also analyze other objects. The plug-in then check for every method definition found if that method have a given list of features. See 36 for more details on features. Then the plug-in output for every method a list of "1" or "0" depending on this method satisfy or not a feature and the method name with source and position.

### 3.2 Features

A feature is a boolean property that a method may satisfy or not.

1. Return type is a collection :
   The return type is a collection. To be enough general to match a return of a List, a Set or a Map, it's implemented to match any class that extends the Traversable trait [4].

2. Return type is an Object :
   This feature match any return type that extends the AnyRef class [2].

3. Return type is Unit :
   This feature match if the return type is Unit.

4. Return type is boolean :
   Straight forward, this feature match boolean return type.

5. Return type is an Integer :
   Same as feature 4 with Integer return type.

6. Return type is a String :
   Same as features 4 and 5 with String return type.

7. Method take no parameter :
   This method is a variant of feature 8 .

8. Method is declared without parenthesis :
   In Scala when a programmer declare a function that take no argument, it's possible to write with or without parenthesis. With parenthesis indicate that this function have side effect; modifying data, printing out on std. In the AbstractSyntaxTree produced by the compiler, the parameters of a method are returned as a List of List. If the first list is empty, then it indicate that this method is declared without parameters and no parenthesis and should therefore have no side-effect. If the first list is of size one and contains an empty list, we have here a method declared without parameters but with parenthesis and thus with possible side-effect. And if the first list have a bigger size than one, it indicate we are dealing with a currified function.

9. Method body contains IF statement :
   The method body contain an IF branch. Note that the Scala compiler translate WHILE block and DO-WHILE block with a IF branch. These two case should not match this feature as the programmer did not write any IF branch. IF guard statement in pattern matching are not taken in account as IF branch. However an IF branch inside a pattern matching right hand side will match.
   IF branch matching inside pattern matching right hand side:

   ```
   ls match {
     case Nil => false
     case x :: xs => if(x==0) true else false
   }
   ```

IF guard in pattern matching

```
ls match {
  case Nil => false
  case x :: xs if(x==0) => true
  case x :: xs => false
}
```

10. Method body contains WHILE statement :
The method body contains a WHILE statement. DO-WHILE statement are also considered as WHILE statement.

11. Method body contains TRY-CATCH statement :
The method body contains a TRY-CATCH block. If a method body contains only the TRY block, it will also match.

12. Method body contains pattern matching :
The method body contains pattern matching. If match if the method body contains a MATCH statement.

13. Method body contains explicit THROW statement :
The method body contains a THROW statement. It match only if it's explicitly declared. A NumberFormatException raised by a wrong string applied to *.toInt* will not match.

14. Method is currified :
This feature is a variant of feature 8.

15. Method is self-recursive :
This method call it-self in it's body. The feature don't match for other method even with surcharged identifiers.

16. Method name is a verb :
This feature match if the method name is a verb. It could be an infinitive or a conjugated form. For this feature, it use the WordNet 2.1 database [1] to determine it.

17. Method name is a noun :
This feature match is the method name is a noun. As in feature 16, it use also the WordNet 2.1 database [1].

18. Method name is a camel case phrase :
An unique word is not always meaningful to name a method. As C convention use underscore-separated words like "end_of_file", Java and Scala convention

use camel case. Camel case is a practice of writing several words composed without white space but with the first letter of each word in uppercase. The first letter of the first word may or not be in uppercase.
This feature split the method name by the non-letter characters and by uppercase letters. Then it reconstruct potential acronym like in 19. Then the feature will match if the split is composed at least of two words and if the second and followings words begin with an uppercase letter. Example:

```
aCamel++Case_PhraseWithXML => List(a, Camel, Case, Phrase, With, X, M, L)
                           => List(a, Camel, Case, Phrase, With, XML)
                           => true
```

Note that *an_Underscore_Separated_Phrase* will match this feature as all words, except the first, begin with an uppercase letter.

19. Method name contains an acronym :
An acronym is defined by the aggregation of uppercase letter and digit. It must start with a letter. Non-letter nor digit characters are considered, as in feature 18, as blank and thus discarded. Inside a camel case phrase, the acronym is construct correctly: the acronym inside *XMLAsString* is reconstruct as *XML*. But as the underscore will also considered as a blank, two acronym separated by an underscore will create one acronym instead of two.
This feature will match if the method name contains at least one word that is an acronym.

20. Method name match an abstract phrase construction :

21. Method name contains *is* pattern :
This feature match if the method name contains the word *is*. *is* can be written *is* or *Is*. As in features 18 and 19, the method name is split into words. Then if at least one word is *is*, then the feature will match.

22. Method name contains *get* pattern :
Straight forward, it's implemented the same way as feature 21.

23. Method name contains *set* pattern :
Same as features 21 and 22.

24. Method name contains *contains* pattern :
Same as features 21, 22 and 23.

25. Method name is a valid JAVA name :
    A valid JAVA method name is a series of JAVA letters or JAVA digit that begin with a JAVA letter [6] and that is not a JAVA keyword [7].

26. Method name is an operator :
    An operator is defined as a following of characters that are neither a letter neither a digit.

27. Method return type is completely contained into the method name :

28. Method return type is partially contained into the method name :
    This feature is a variant of feature 28. For composed type name, as programmer are often lazy, they often write it partially. One would by example write only "Tree" instead of "AbstractSyntaxTree". This feature will match in such case as previous feature will not. It will also match if the type is completely contained, like 28.

29. Method is right associative :
    Scala language is left associative. But a method name that finish with the *:* character is right associative.

    ```
    1 :: List(2, 3) => List(2, 3)::.(1)
                    => List(1, 2, 3)
    ```

    This feature match if the last character of the method name is *:* .

30. Method is declared into another method :
    Scala language permit declaration of functions inside function. It's a good way to declare subroutine that should not be used outside of this function. This feature match if the method body contains a method declaration.

31. Method body contains inner method definition :
    This feature is the opposite of feature 30 and match if the method body contains a method declaration.

32. Method is overriding another method :
    This feature match if this method override another method.

33. Method is abstract :
    This feature match if the method body is empty.

34. Method is public :
    This feature match if this method is public. Note that inner function are always public but can't be accessed from outside.

35. Method is static :
    This feature match if this method is static.

36. Method name finish with "s" :
    The last character of the method name is *s*.

37. Method name finish with "ss" :
    This two lasts characters of the method name are both *s*.

## 3.3 Clustering

With the output of the Scala-names plug-in, I filled the $k$-means algorithm [9] hoping the result will be interesting. The implementation of the algorithm follow [9] description with a random partition. K-means is sensible to the initial partitioning and its output may be different. The output may also contains empty clusters. As k-means reach a stable point fast, It's generally recommended to run several time the algorithm to be sure to obtain a fine result.
First question, how many cluster should we use ? A too small number of clusters will aggregate different groups of methods and a too high number of clusters will separate methods in a same group. I think the answer to this question really depend on the data set and what we want to find. See 4.1 for answer based on this project.
At the end of the algorithm, we obtain clustered data. Fine, but the center of cluster is a raw position, as it's the average position of all its members. To obtain a position more readable an more meaningful, I discrete the position of the clusters depending on a threshold to obtain a position only composed of: 0, 1 or ?. If for a given dimension the value is lower than the threshold, the value is lowered to 0. Respectively if the value is higher than $1 - threshold$, the value is raised to 1. And if the value is higher than the threshold and lower than $1 - threshold$ the value is set to undefined: ?. See figure 1 for an example on two dimensions.
Then I do one more step of k-means to check the effect of the discretization to the clusters. Here another question is raised: what's a relevant threshold ? It should be small enough to be as most correct but big enough to have a minimum of undefined values. See the respond in section 4.1.2.

# 4 Experimental Results

## 4.1 Clustering

To answers the two questions of section 3.3, I ran 100 times the k-means algorithm on the output of the Scala-names plug-in of the Scala library with different numbers

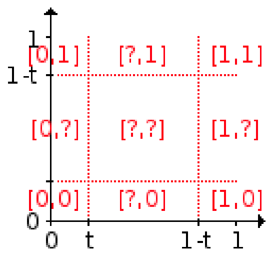
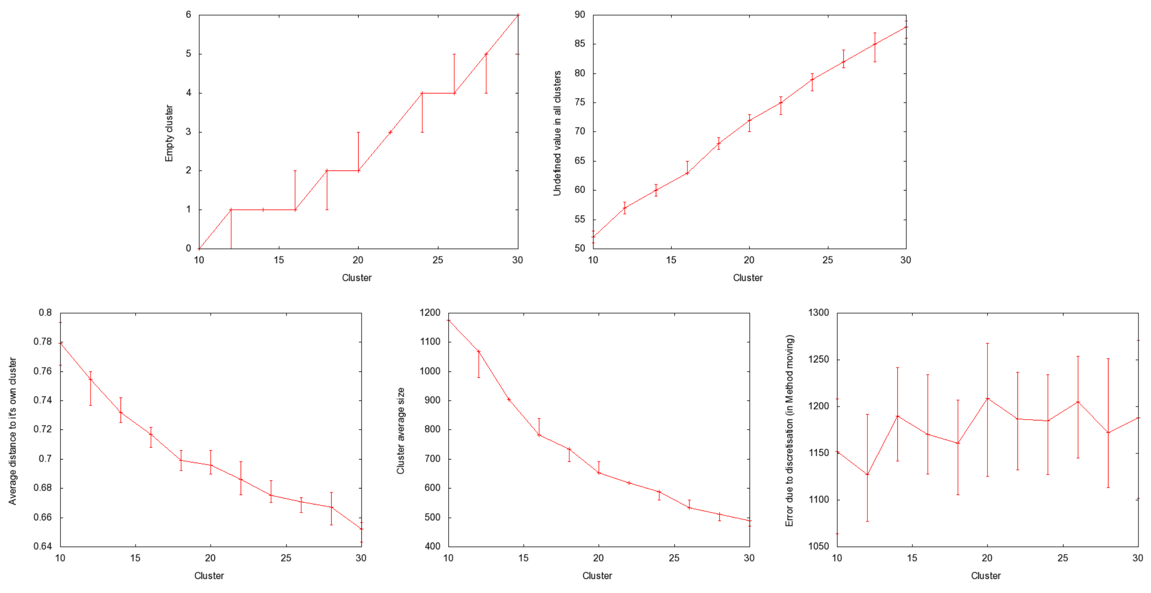


Figure 1: Discretization of a cluster of two dimensions

of clusters. I took five mean metrics with confidence interval of 95%: the number of empty clusters, the number of undefined values in all clusters, the average distance of methods to their respective cluster, the cluster average size and the error due to discretization modeled by the number of methods moving after a one-step check.

### 4.1.1 Number of clusters

For this series of experiments, the threshold was fixed at 0.15 .



1. As the number of clusters increase, the number of empty clusters increase too. With 30 clusters we reach 6 empty clusters. It represent a percentage

of empty cluster of 20%, which is pretty high. As The computational time of the algorithm also increase with a higher number of cluster, it suggest that a lower number of clusters is better.

2. To be independent of the first metric, the second it is calculated only with non-empty cluster. The number of undefined values in the clusters also increase with the cluster number on a basis of $\frac{2}{3}$. This is bad as we can't deduce anything from them.

3. The third metric seams not really relevant. If the distance of a method to it's own cluster decrease, that's good, as the cluster is a better representation of its members. But as the result for ten cluster is already below a distance of one, a so small difference have no impact.

4. As the fourth metric show the average cluster size, we can pretend that a cluster with a smaller number of members will be more accurate. My experience say the same as during the development of the project I get as a first result of k-means, with ten clusters, a pretty strange result. As all the ten clusters have a negative result for the pattern matching feature, I looked into some files at random, saw that were true for those files and stated: "*Yes, that's true, the Scala library do not contains any pattern matching.*" Off course, that's wrong. But pattern matching is found only in 8% of all methods, making this feature less representative than feature with a higher probability. On the other hand, as pointed by the second metric, the number of undefined values increase which try to say that to obtain a smaller granularity we need to cut some concentrated points.

5. The last metric seams unstable and really depend on the initial partitioning. But we need to consider the library size: 11'752 method declarations. The variation of the mean is of 1% and the number of method that move to another cluster stay on a percentage of 10% of all methods. I noted that this percentage of moving methods was lower with a lower cluster dimension. We can conclude that the number of clusters don't impact on the error due to discretization.

On the basis of theses results, I decided to take a number of cluster of fifteen to shrink the cluster size at maximum without to suffer of bad effects of undefined values.

### 4.1.2 Threshold

Here again, I ran hundred time the same experience, but this time a fixed number of cluster of fifteen on different threshold.

### 4.2 Features

### 4.3 Correlation

### 4.4 Others

## 5 Conclusion

## 6 Future and Related Work

### 6.1 Adding New Features

The Scala compiler can infer type. So programmers don't need to always specify the return type of a method. In some case the type could be a good indication to the reader to understand the code. But in other case like the method *toString()* we don't really need to see the type to know its type. Adding a feature returning if the type is declared or inferred, may obtain information about code quality.

### 6.2 Compose With Others Analysis

Method name are not the only "object" that the programmer name. He also name variables, class, parameters and types. An analysis of these "objects" can also find out to be interesting. By example, the feature 37 give nothing with method name but could highlight something on variables. Cross analysis on all objects can also demonstrate some results.

## References

[1] *WordNet, A lexical database for English.* Princeton University, New Jersey, USA, March 2005. `http://wordnet.princeton.edu/wordnet/`.

[2] Anyref class. [3]. consulted January 6, 2012, `http://www.scala-lang.org/api/current/index.html#scala.AnyRef`.

[3] *Scala Standard Library 2.9.1 final*, 2011. `http://www.scala-lang.org/api/current/index.html#package`.

[4] Traversable trait. [3]. consulted January 6, 2012, `http://www.scala-lang.org/api/current/index.html#scala.collection.Traversable`.

[5] Einar W. Høst and Bjarte M. Østvold. Debuging method names. In *Proceedings of 2009 ECOOP - Object-Oriented Programming*, Genova, Italy, July 6–10 2009.

[6] Sun Microsystems. Java identifiers. [8]. `http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.8`.

[7] Sun Microsystems. Java keywords. [8]. `http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.9`.

[8] Sun Microsystems. The java language specification, third edition. Santa Clara, California, USA, 2005.

[9] Wikipedia. k-means clustering. consulted January 5, 2012, `http://en.wikipedia.org/wiki/K-means_clustering`.