Metrics for code 'functional-ness' in Scala
Dissertation proposal

July 15, 2015

Master Student Name: Alexandru Matei
Supervisor Name : Marius Minea
Dissertation Domain: Software engineering - Functional Programming

Contents

1 Introduction

Functional programming has started to gain more traction in the last years, thanks to the growing adoption of Scala and Haskell in the software industry; we can see that Google trends shows a rising interest in functional programming languages (see figure 1).

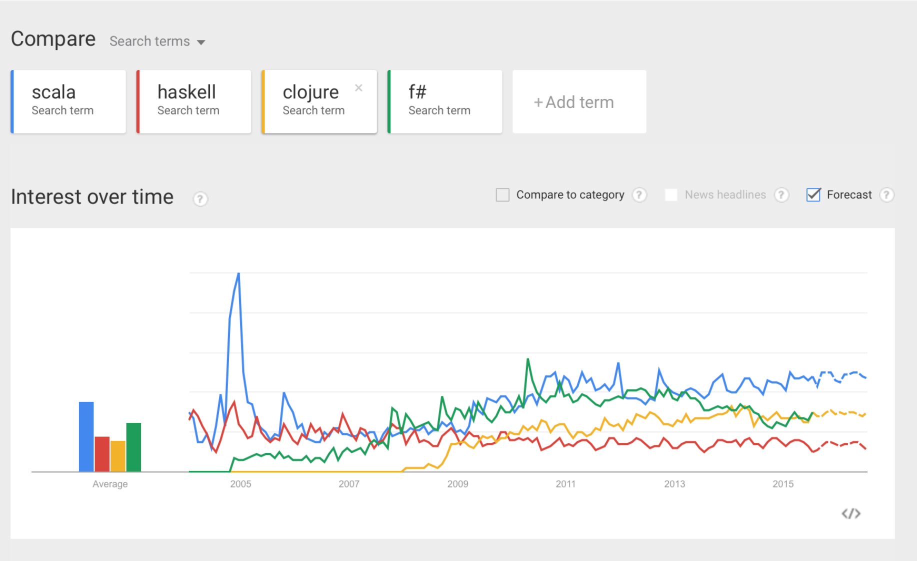

Figure 1: Google search hits for programming languages - 2004 to July, 2015

Scala is one of the leading functional programming languages, over the last couple of years being adopted by large companies such as Twitter(2009), Linkedin(2010), The Guardian, Foursquare. If we look at 'RedMonk Programming Languages Rankings'[O'G15], which is based on StackOverflow and Github analysis, Scala occupies a worthy 14th position, being the first FP language in the top .

One of its success factors is Scala's compatibility with Java Virtual Machine hence code interoperability with Java, and the benefits that come along from Java's well-established ecosystem.

Another one might be that Massive Open Online Courses (MOOCs) like Functional Programming in Scala [MO15] held by the creator of Scala, Martin Odersky and Reactive Programming are one of the most popular online courses; the feedback from developers is really encouraging(see figure 3).

Another reason why functional programming comes in handy nowadays is that multiprocessor architectures become ubiquitous and developers need to get more familiar with distributed and multi-threaded computing. Programming with shared state has proven to be difficult to reason about and functional programming offers to save a lot of headaches by removing shared state from

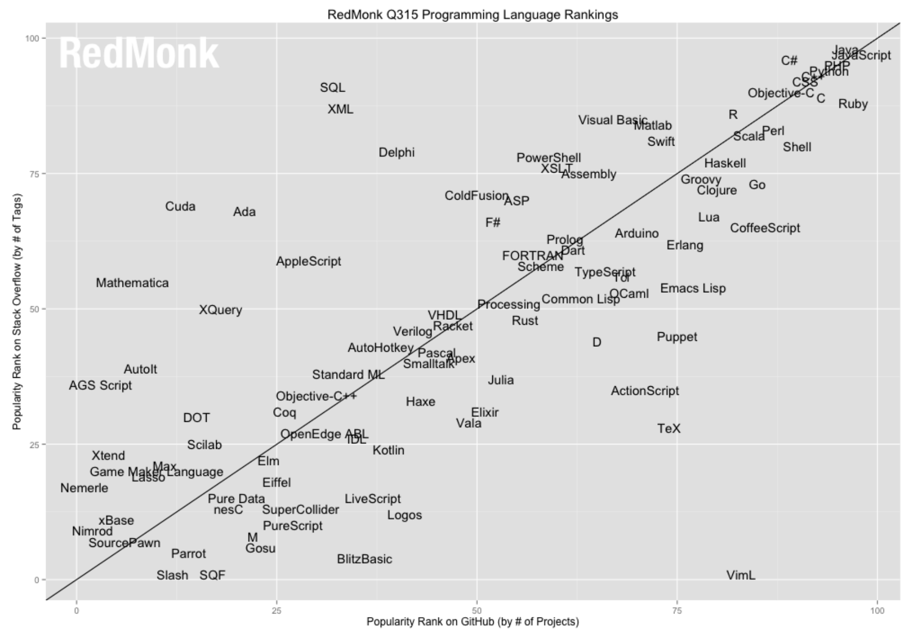

Figure 2: Redmonk rank of programming languages - June, 2015

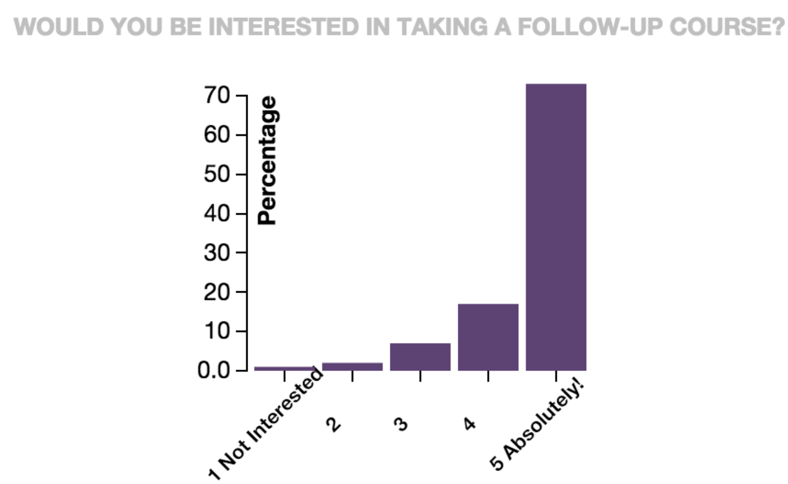

Figure 3: Feedback FP in Scala course - interest in future courses on Scala (FP)

the equation and providing better concurrency abstractions (Futures, Actors). We should also consider the exceptional distribution capabilities of map-reduce that functional programming offers. Although the programming paradigm was born more than 80 years ago, functional programming seems like a totally new way of thinking about software to modern day developers, having lots of secret gems left to be discovered.

2 Problem statement

Scala is a hybrid programming lanaguage combining both functional and object oriented elements [Ode15]. Writing Scala code does not imply you are writing functional code; there is no mechanism in the language that enforces a functional style of programming; so one can say they write code in Scala so they use functional programming when in fact their code base is entirely procedural. I was also put in such a position, being one of the many developers that switched from an Object Oriented language(Java /C#/C++xs etc.) to Scala. So how can one figure out if his Scala code follows the functional programming principles or not? The classic approach is to ask a functional programming expert for code review, if you are lucky enough to have access to such valuable resources.

The solution we propose is a set of metrics that should be able to quantify to what degree the code is making use of the functional paradigm. But what qualifies a code as being functional? What are the characteristics of functional code? In the following section we will elaborate the main characteristics of functional programming languages.

3 Theoretical Foundations

First of all we need to establish the main elements of functional languages and we will do so by taking Scala as an example. Only then we can elaborate further on the metrics and decide which ones are to be considered for the study.

The question we will try to answer in this body of work is 'how much does a method use functional programming concepts?' The main elements we will look at are: immutability, referential trasparency, high-order functions, monads, laziness.

3.1 Immutability

If a variable/object is immutable then its value never changes. That is a useful guarantee if one plans on going concurrent: any such value can be safely shared amongst treads since the value is read-only. Another advantage is the reduced aliasing between different parts of the program; one can say that immutable objects are easier to reason about and also their API is straight forward because you don't need to reason about internal state changes - everything is transparent compared to mutable objects that introduce opaqueness in reasoning about their possible states at different moments of time .

Scala encourages immutability through case classes, which provide a syntatic sugar for creating immutable objects (data structures). Case classes are regular classes which export their constructor parameters and which provide a recursive decomposition mechanism via pattern matching.

```
abstract class Pet
case class Dog(name:String) extends Pet
case class Cat(name:String) extends Pet
case class Hippo(name:String, weight:Int) extends Pet

//Decomposition
def printPetType(pet:Pet): String= pet match{
  case Dog => "it's a Dog"
  case Cat => "it's a Cat"
  case Hippo => "it's a Hippo"
}
```

By default, case classes ar immutable but one can change some of fields to be mutable, although it is not recommended.

Declaring a variable as a 'val' prevents it for being reassigned; still, this doesn't guarantee that the object it is reffering to is immutable. Best transparency is obtained with using both val's and immutable objects: this is the recommended approach also when dealing with concurrency. Scala provides both mutable and immutable collections that can be found in packages scala.collection.mutable and scala.collection.immmutable repectively. By default, Scala always picks immutable collections.

```
val x = 4
x = 5 //Reasignment to val generates error
```

So, in order to create an immutable Scala object, it is necessary to have all the fields declared as vals + all the values to be immutable objects in turn. Case classes make it easier to accomplish that in as few lines of code as possible.

3.2 Referential tranparency, pure functions

An expression is referentially transparent (RT) if it can be replaced by its resulting value without changing the behavior of the program. This must be true regardless of where the expression is used in the program. Programming without side effects leads to referential transparency. An example:

```
def f(x:Int)= x* 3
val z = f(3)
\\Now, whenever we use z in the code, we can safely
\\ replace it with f(3) without changing the result of the program
```

Pure functions evaluate to the same result given the same argument value(s) and don't have any side effects. A definition combining both concepts can be found in 'Functional Programming in Scala' by Chiusano and Bjarnason : 'A function f is pure if expression f(x) is referentially transparent for all referentially transparent values x'.

Examples of pure functions in Scala include:

- Methods on immutable collections such as map, drop, filter, take
- Methods like split, length on the String class
- Mathematical functions such as add, multiply ...

As a rule of thumb in Scala if a function has return type Unit, then most probably it has side effects and it is not pure.

A measure for the degree of referential transparency in a program can be expressed as:

$$RTI = \frac{|\{m \in M / Pure(m)\}|}{|M|}$$

Where RTI- Referential Transparency Index , M - set of all functions/methods of the software and Pure(m) - predicate that says if a function is pure or not. In this case, a value close to 1 corresponds to improved referential transparency while one close to 0 says that the property is not satisfied at all. [MABP13]. Further study is needed on elaborating how the pure function should be defined.

3.3 High-order functions

The central concept of functional languages as pointed out by John Hughes [Hug89] in 'Why Functional Programming Matters' is higher- order functions ; Hughes argues that when high order functions are combined with laziness techniques they greatly increase the modularity of software by providing novative ways (compared to other structured programming techniques) of 'gluing' modules together so programs can become more concise and easier to reason about.

Before we begin talking about high-order functions we should first understand what does an order of a function mean:

- Order 0: Non function data
- Order 1: Functions with domain and range of order 0
- Order 2: Functions with domain and range of order 1
- Order k: Functions with domain and range of order k-1

So order 0 is represented by numbers, lists, characters, etc. Order 1 are functions wich work with order 0 data. So order 1 data are the well known functions that every programming language supports. Functions with an order grater than 1 are callled higher-order functions and they fall at least in on of the following categories:

- they take other functions as parameters

- they return a function as a result

Being able to pass functions as parameters and return them as results means that functions are first-class citizens and is one of the core concepts that make a functional language.

An example is the following apply function written in Scala, which takes a function and an integer as parameters and produces a function as a result:

```scala
def apply(f: Int => String, v: Int) = f(v)
```

Classical higher-order functions over lists :

- Mapping: Application of a function on all elements in a list
- Filtering : Collection of elements from a list which satisfy a particular condition
- Accumulation: Pair wise combination of the elements of a list to a value of another type
- Folding: Reducing a list over some function with accumulator

[http://people.cs.aau.dk/ normark/prog3-03/pdf/higher-order-fu.pdf]

A study about high-order functions metrics and their corelation with software modularity was also conducted by B. Muddu et. al. [MABP13]. The proposed metric was studying the coupling between high order functions in diferent modules. It might be useful to make use also of their modularity metric when assesing code functional-ness.

3.4 Monads

Monads are a central design pattern of functional programming. They've been successfully used to abstract over well known problems in programming [Jon01]: non-determinism as expressed by list, time/ concurrency, mutable state (see IO Monad Haskell), exception handling, foreign language calls.

In Category Theory, a Monad is a functor equipped with a pair of natural transformations satisfying the laws of associativity and identity. In Scala, monads are just a parametric type M[T] with two operations, flatMap (or bind) and unit which also preserves associativity and identity.

```scala
trait M[T]{
  def flatMap[U](f:T=>M[U]): M[U]
}

def unit[T](x:T): M[T]
```

3.5 Lazy evaluation

Lazy evaluation or call-by-need is the opposite of eager evaluation(call-by-value) and it is an evaluation strategy which delays the evaluation of an expression until it is needed and only then computes the result and caches it for further evaluations.

One advantage is the memory saving due to postponing the evaluation but it comes with a performance cost: you get faster intialization but later (when evaluation occurs) you suffer a performance penality.

Scala supports laziness by introducing the lazy keyword and call by name parameters. By default it supports strict evaluation in contrast with Haskell which is lazy by default.

```
lazy val product = 100 * 30 // not evaluated
println(product.toString) // evaluated


object Test {
   def main(args: Array[String]) {
        delayed(time());
   }

   def time() = {
      println("Getting time in nano seconds")
      System.nanoTime
   }
   def delayed( t: => Long ) = {
      println("In delayed method")
      println("Param: " + t)
      t
   }
}
```

4 State of the art - Literature study

In the first section we talked about elaborating a series of metrics, in order to asses one's code functional-ness but we didn't have the opportunity to provide a proper definition for the concept of a metric and it's existing state of affairs in software industry. An easy definiton would be that metrics offer a quantitative measure of a software property.

4.1 Metrics overview and their use in FP languages

"When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind:

it may be the beginnings of knowledge but you have scarcely in your thoughts advanced to the stage of Science." (Lord Kelvin)

A metric is a measurement function, and a software quality metric is "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality [KMB04]. To some extent, metrics could be regarded as compiler warnings, signaling that a part of your code needs more testing. Software metrics are used in Software Engineering for helping with software development activities such as testing and refactoring, performance optimization, debugging, cost estimation, etc. Some common software metrics include : source line of code (LOC), number of functions, cyclomatic complexity, code coverage, cohesion, coupling.

Study of software metrics has been an active area of research since early 70' targeting mostly object oriented and imperative languages [RT05]; as for functional languages, the number of published papers is not the numoerous; some of the prior work on metrics for functional languages was started almost 20 years ago by K. van den Berg in 1995 [vdBvdB95] who proposed a set of metrics for evaluating code complexity of Miranda functional programming language and Harrison which studied code modularity for SML [HSDL96]; 10 years later, Ryder and Thompson proposed some metrics for Haskell [RT05] whereas Muddu et. al. developed metrics for Scala [MABP13]. We can corelate the few number of papers on FP with the fact that FP does have yet such large adoption in software industry compared to object oriented and imperative languages. So we can devide prior work on metrics into flow-graph metrics and functional-related metrics: referential transparency, high-order functions, pure functions[MABP13]

Most commonly used metrics for functional languages in the papers we mentioned about are:

- Pattern related metrics - applies mostly to Haskell
- Scoping metrics -how many scopes does a function introduce, the number of declatations brought in scope
- Call graph metrics - strong connections, in degree, out degree, depth, width, arc-to-node ratio
- Function atributes - path count, number of operands vs operators

In oder to study the maintability, Basavaraju Muddu et. al. introduce a technique for breaking software into logical modules [MABP13] The technique they chose was Modularizing by Inverse-Depth heuristic, which is based of the project's file directory structure; they start at the source root folder and compute for each branch the maximum depth; then starting from the bottom up, they assign a module up to a certain depth; in their study, they use inverse depth of 1 for selecting modules, arguing that these are the closest approximation to actual logical modules. We would like to try other modularization techniques:

package based, SBT (Scala Build Tool) modularization and relate them to our problem: establishing code functional-ness: would a good 'functional-ness' value for our proposed metric lead also to improved modularity?.

The biggest problem with code metrics is not the definition of new ones but the extraction of meaningful information from them;code metrics without semantics attached to them are just plain numbers; we need to corelate them to some process development metrics and this is the hardest part, since process activities are difficult to cuantify; most of the research in FP metrics is targeting maintanability, code quality. We would like to see how functional property metrics impact refactorings; is there a way to detect 'functional code smells'?.

Given previous papers on measuring programs written in functional languages, we would like first to find means to measure (if possible) all functional properties described in the previous chapter and try to relate them to both code quality and possible refactorings; it would be nice to study if some object-oriented structres could be restructred to FP patterns. Huiqing Li propose a refactoring tool for functional programms written in Haskell[LRT03]; starting from here, we can imagine corelating our metrics with possible refactorings suggestions, of course, this goal is rather ambitious but it's worth a little more study on the matter.

4.2 Static analysis tools for Scala

There are already a couple of static analysis tools written for Scala : ScalaStyle, ScapeGoat, Wart remover, Linger and Scala Abide. These tools are looking possible errors that might appear in Scala code and also checks for a certain style of coding. Some of the checks they perform are: identation, illegal imports, multiple declared strings, null appearances, redundant if statements, cyclomatic complexity, unreachable catch statements, unexpected recursive definitions, unassigned variables, shadowing etc.

Some metrics that could turn out to be useful for our study are the presence, absence of vars and cyclomatic complexity; ScalaStyle checks that classes and objects do not define mutable fields and that functions do not define mutable variables (VarFieldChecker, VarLocalChecker) [sca15] and it also has a metric for cyclomatic complexity. At least from an implementation perspective it would be worthwhile to take them into account.

5 Conclusions and Future work

In this paper we've looked at the main characteristics defining functional code and discussed the existing body of work regarding metrics inside functional languages. The state-of-the-art leaves a lot of room for new investigation to detect flaws. This is will be the main purpose of our research.

The leading question of this work as presented in section 1 is 'How can we determine the degree of code "functional-ness" '? What code patterns can we define as not being functional, how can we detect them and what can we do about them?

In most practical situations we cannot give binary answers on the functionalness of a codebase. Instead we will make use of metrics to quantify this aspect. The first step will be to conduct a study on which functional elements can be measured and which not. We will need to establish the granularity of what we want to investigate: intra-method level, inter-method, class-level. Afterwards, we will design a series of metrics to measure functional properties of the program.

The second step will be to implement a plugin in one of the main technologies used in the Scala ecosystem to make these measurements.

The third step will be to apply our plugin against a substantially large codebase and evaluate its effectiveness in detecting code flaws and “bad smells”. Our theoretical model should be complete enough to give reasons and refactoring suggestions for these when we detect them.

References

[HSDL96] R. Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis. Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering J.*, 11(4):247–254, July 1996.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[Jon01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2001.

[KMB04] Cem Kaner, Senior Member, and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[LRT03] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 27–38, New York, NY, USA, 2003. ACM.

[MABP13] Basavaraju Muddu, Allahbaksh M. Asadullah, Vasudev D. Bhat, and Srinivas Padmanabhuni. Metrics for modularization assessment of scala and c# systems. In *4th International Workshop on Emerging Trends in Software Metrics, WETSoM 2013, San Francisco, CA, USA, May 21, 2013*, pages 35–41, 2013.

[MO15] Heather Miller and Martin Odersky. *Functional Programming Principles in Scala: Impressions and Statistics*, 2012 (accessed July 10, 2015).

[Ode15] Martin Odersky. *What is Scala?*, 2012 (accessed July 10, 2015).

[O'G15] Stephen O'Grady. *The RedMonk Programming Language Rankings: June 2015*. RedMonk, 2015.

[RT05] Chris Ryder and Simon J. Thompson. Software Metrics: Measuring Haskell. In *Trends in Functional Programming*, pages 31–46, 2005.

[sca15] scalastyle.org. *Scalastyle: Implemented Rules*, 2014 (accessed July 10, 2015).

[vdBvdB95] Klaas van den Berg and P. M. van den Broek. Static analysis of functional programs. *Information & Software Technology*, 37(4):213–224, 1995.