Research Study - Metrics for code 'functional-ness' in Scala

Alexandru Matei

July 13, 2015

Contents

1 Introduction

Functional programming started to get 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).

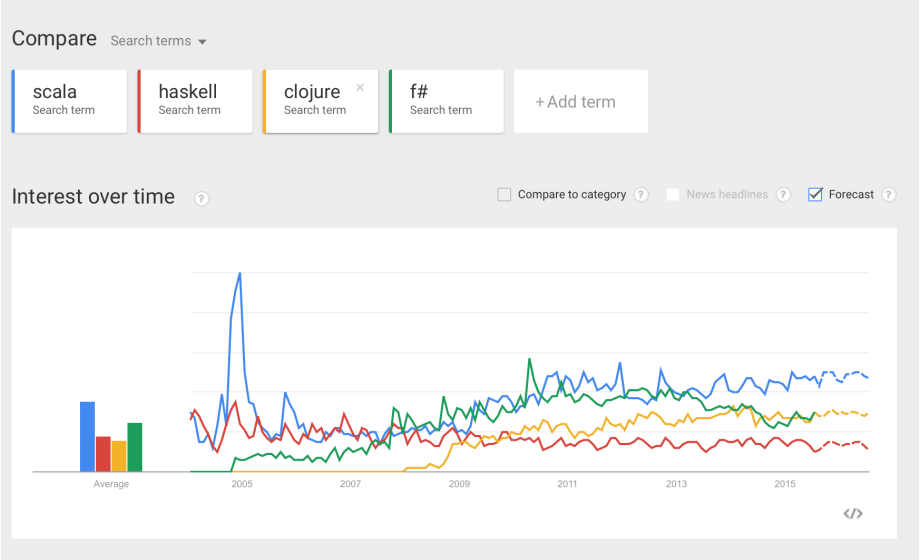

Figure 1: Google search hits for programming languages - 2004 to July, 2015

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 top .

One of its success factors is Scala's JVM compatibility and code interoperability with Java, and the benefits that come along from Java's well-estabilished eco-system. Another one might be that 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 FP 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 pretty difficult to reason about and functional programming offers to save a lot of headaches by removing shared state from the equation and providing better concurrency abstractions (Futures, Actors) and we should also

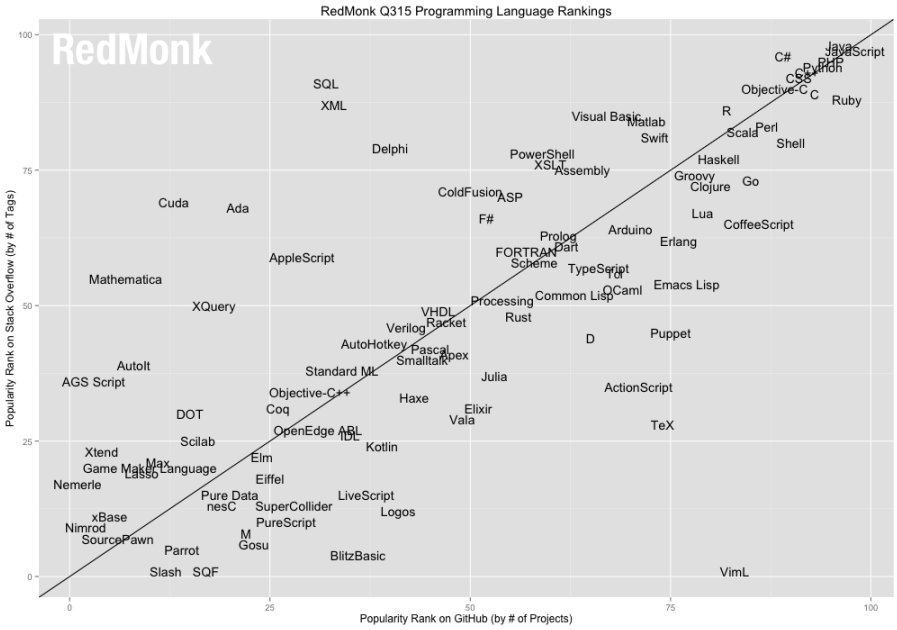

Figure 2: Redmonk rank of programming languages - June, 2015

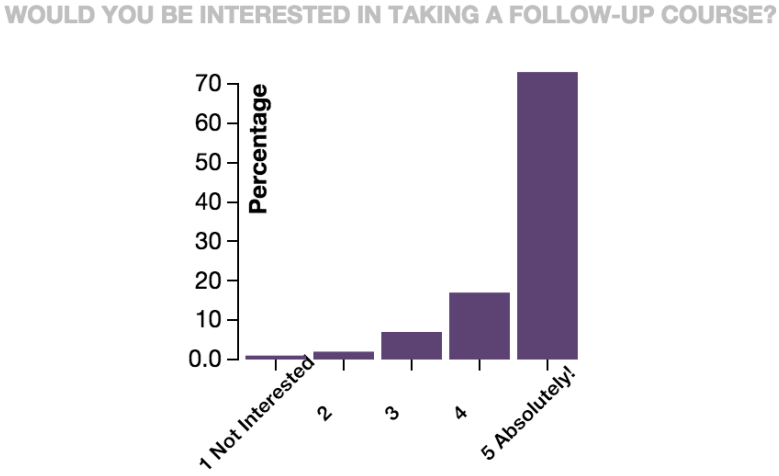

Figure 3: Feedback FP in Scala course - interest in future courses on Scala (FP)

consider the exceptional distribution capabilities of map-reduce that FP offers. Although the programming paradigm was born more than 80 years ago, FP seems like a totally new way of thinking about software to modern day devel-

opers, having lots of secret gems left to be discovered.

2 Problem description

Scala is a hybrid programming lanaguage being both a functional language and an object oriented one (cite http://www.scala-lang.org/what-is-scala.html). Using the features of functional programming should lead to less code which in turn usually leads to less bugs and better understandability. K. van den Berg Phd. thesis concludes that 'The results of the experiments show that the programs in a functional lan- guage took significantly less time to develop and were considerably more con- cise and easier to understand than the corresponding programs written in an imperative language' [vdBvdB95].

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 he writes code in Scala so he uses FP when in fact his code base is entirely procedural. I was also put in such a position, being one of the many developers that switched from an OO language(Java /C#/C++xs etc.) to a functional language such as Scala. So how can one figure out if his Scala code follows the FP 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. Wouldn't it be nicer (and cheaper) to have a free tool that does the code review job for you?

The solution we propose is an automatic tool checker that would be able to asses one's code 'functional-ness'; it should be able to guide the Scala programmer through the coding process and help him understand what features of FP he is using or missing on. We can imagine having a DSL for choosing which are the targeted FP properties for the analysis and receiving an detailed report on the analysis with further guidelines.

But what means qualifies a code as being functional? What are the specifics of a functional code? These are some of the questions that we have to answer first, in order to come up with metrics for code functional-ness.

It is in this context that we start our study on functional programming and the pursue of searching and understanding its features.

3 Methodology

First of all we need to establish the characteristics of FP languages, 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.

We get to ask ourselves the following questions: 'how much does a method uses functional programming concepts¿ Does it contain only immutable data, so no shared state? Does it use higher order functions for data processing? Does a method use advanced features such as monadic comprehension? Does a method use functional-like constructs such as anonymous inner classes?

3.1 Immutability

If a variable/object is immutable then it's value never changes. That is a really usefull 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 more 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 evaluates to the same result given the same argument value(s) and they don't have any side effects. A better definition 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 trasparent 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 its return type Unit, then most probably it has side effects.

3.3 High-order functions

One of the features pointed out by John Hughes [Hug89] in 'Why Functional Programming Matters' are higher- order functions and lazy evaluation; Hughes argues that these two concepts 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. One other design pattern that was first approached in Haskel and targets modularity by emphasizing on composition is represented by monads which model the concept of a category (see 3.4).

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

An example is the following apply function written in Scala, which takes a function (defined on integers that returns a string) and an integer as parameters and returns the function application over the integer value:

```
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
- Zipping: Combination of two lists to a single list

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

3.4 Monads

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.

```
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 lazyness 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 Literature study

Current research on functional programming static analysis include Klass van den Berg [vdBvdB95] which studies the use cases of control-flow graphs in Miranda, a pure functional language, same as Haskell; the main focus is on program coprehensibility and they conclude that structured function definitions appear to be easier to understand by novice programmers.

Chris Ryder an Simon Thompson propose a series of metrics for Haskell functional programming language in order to reason about functions that may have an increased risk for errors and might need more rigurous testing [RT05].

Regarding Scala programming language, Basavaraju Muddu studies metrics for assessing Scala modularity; the study focuses on referential tranparency, functional purity and high order functions, some of the aspects that we will also use in our metrics to determine code functional-ness [MABP13]. To some extent, modularization metrics are a subset of the functional-ness metrics.

As for the industry there are already a couple of static analysis tools written for Scala :

- 'ScalaStyle' that is a scala style checker (implemented rules can be found at http://www.scalastyle.org/rules-0.7.0.html)
- ScapeGoat plugin (https://github.com/sksamuel/scalac-scapegoat-plugin) - signals suspicious language usage in code
- Wart remover https://github.com/puffnfresh/wartremover - linting tool
- HairyFotr/linger https://github.com/HairyFotr/linter
- Scala Abide https://github.com/scala/scala-abide - linting tool

5 Research questions

We have to first decide upon the patterns that are most relevant and study about the compiler support for performing such analysis (Scala compiler plugins). Next section presents some of the targeted FP features that we will try to measure in order to compute a metrice.

Scala provides an API to extend its compiler through Scala compiler plugins; how they work is that they insert custom functionality at different stages of the compilation; of course this could lead to errors, if used improperly. Unfortunately, there are only a limited amount of resources online (http://lampwww.epfl.ch/ magarcia/ScalaCompilerCornerReloaded/) that talk about how to create plugins.

There is already a scala build tool plugin named

The only paper published on FP analysis was 'Static analysis of functional programms' ; still, we could not find a paper that handles code 'functional-ness' the way we presented above.

References

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

[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).

[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.

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