

Metrics for code 'functional-ness' in Scala

Dissertation proposal

August 9, 2015

Master Student Name: Alexandru Matei

Supervisor Name : Marius Minea

Dissertation Domain: Software engineering - Functional Programming

Contents

1	Introduction	1
2	Problem statement	3
3	Theoretical Foundations	3
3.1	Immutability	4
3.2	Referential transparency, pure functions	5
3.3	High-order functions	5
3.4	Monads	7
3.5	Lazy evaluation	7
4	State of the art - Literature study	8
4.1	Metrics overview and their use in FP languages	8
4.2	Static analysis of functional programs- K. van den Berg	9
4.3	Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs - R.Harrison et. al.	10
4.4	Software Metrics: Measuring Haskell - Chris Ryder and Simon Thompson	10
4.4.1	Pattern metrics	11
4.4.2	Distance metrics	11
4.4.3	Callgraph attributes	12
4.4.4	Function attributes	13
4.5	Bullshit	14
4.6	Combining Functional and Imperative Programming for Multi-core Software: An Empirical Study Evaluating Scala and Java	14
4.7	Metrics for modularization assessment of Scala and C# systems	15
4.8	Coordinator guidance	15
4.8.1	Problem definition	15
4.8.2	General critique	16
4.8.3	References	18
4.9	Static analysis tools for Scala	19
5	Conclusions and Future work	19

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).

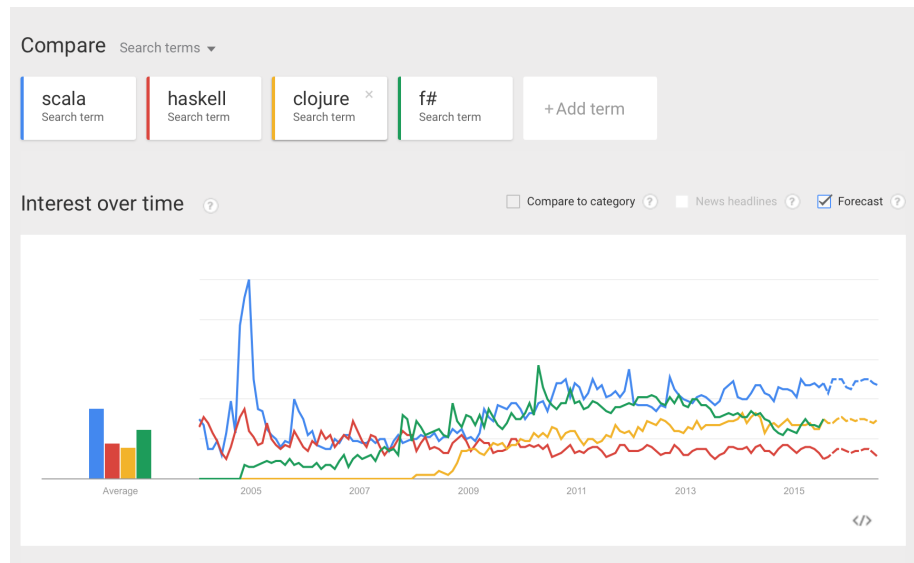


Figure 1: Google search hits for programming languages - 2004 to July, 2015

There is a growing interest in functional programming languages, because of their expressive power and the possibility to reason about correctness of programs. There are claims on the readability of functional programs, for example: 'In many cases the functional programming style yields more elegant and comprehensible programs than the imperative programming style' (Springer & Friedman, 1990) and 'Functional programming leads to programs which are exceptionally clear and concise and to the prospect of greatly increased software reliability and development speed' (Bailey, 1990). In a case study on the productivity of programming in a functional programming environment, some of these claims have been confirmed (Sanders, 1989). Moreover, the learning of programming in a functional programming style should have advantages over learning in an imperative style (Springer & Friedman, 1990; Bailes & Salzman, 1989). [vdBvdB95]

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 .

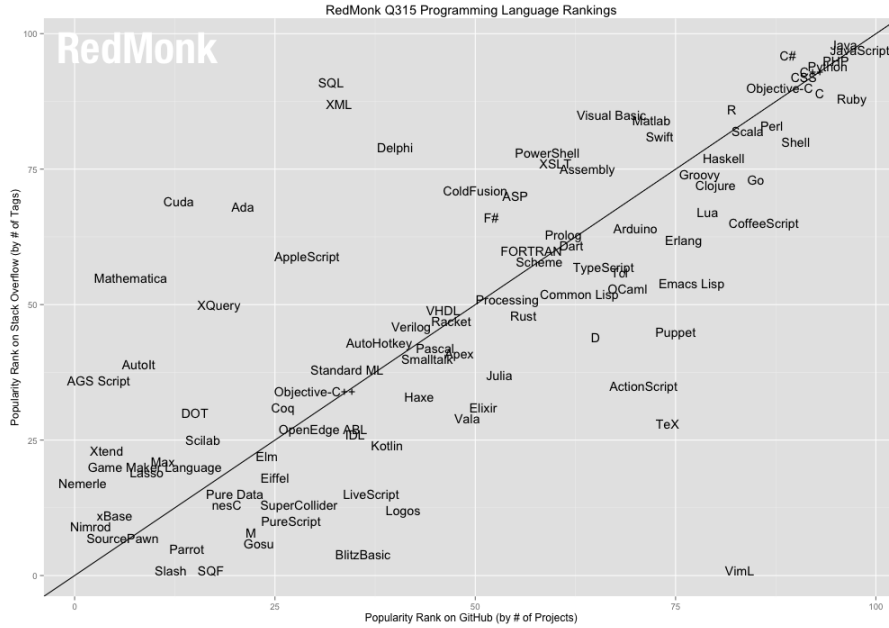


Figure 2: Redmonk rank of programming languages - June, 2015

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 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.

WOULD YOU BE INTERESTED IN TAKING A FOLLOW-UP COURSE?

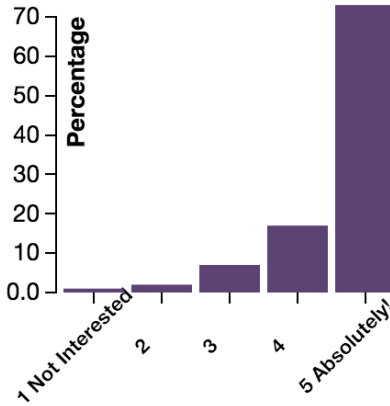


Figure 3: Feedback FP in Scala course - interest in future courses on Scala (FP)

2 Problem statement

Scala is a hybrid programming language 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 transparency, 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 threads 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 syntactic 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 are immutable but one can change some of fields to be mutable, although it is not recommended.

Declaring a variable as a 'val' prevents it from being reassigned; still, this doesn't guarantee that the object it is referring 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.immutable` respectively. By default, Scala always picks immutable collections.

```
val x = 4
x = 5 //Reassignment 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 transparency, 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:

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

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 novel 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 which work with order 0 data. So order 1 data are the well known functions that every programming language supports. Functions with an order greater than 1 are called higher-order functions and they fall at least in one 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:

```
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 correlation with software modularity was also conducted by B. Muddu et. al. [MABP13]. The proposed metric was studying the coupling between high order functions in different modules. It might be useful to make use also of their modularity metric when assessing 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.

```
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 initialization but later (when evaluation occurs) you suffer a performance penalty.

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 assess one's code functional-ness but we didn't have the opportunity to provide a proper definition for the concept of a metric and its existing state of affairs in software industry. An easy definition 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 numerous; 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 correlate 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.

4.2 Static analysis of functional programs- K. van den Berg

K. van den Berg performed a comparison study regarding code comprehensibility of program code: how easy is it to a developer to read and understand a program in order to find and fix some errors or to adapt to changing requirements. The languages that were studied are Pascal and Miranda, the first being the representative for ‘imperative languages’, the last representing ‘functional languages’. One of the metrics he chose to use are Halstead and McCabe’s, which were adapted for both programming paradigms.

Halstead metrics can be expressed as :

- η_1 - the number of unique operators
- η_2 - the number of unique operands
- N_1 - the total number of operators
- N_2 - the total number of operands
- η'_2 - the number of different input/output parameters

Using this terminology, one can define the following:

- $\eta = \eta_1 + \eta_2$ - the vocabulary of a program
- $N = N_1 + N_2$ - the length of a program as total number of tokens
- $V = N * \log_2 \eta$ - the size of a program (volume)
- $V' = (2 + \eta'_2) * \log_2(2 + \eta'_2)$ - the potential volume of a program
- $L = V'/V$ - program level or level of implementation
- $E = V/L$ - effort to generate a program

McCabe’s metrics for cyclomatic complexity focus on the decision structure of a program. The program is viewed as a decision graph G , having a unique entry and exit point. Every edge in the graph represents a decision branch while every node is a decision point. This metric is related to the difficulty of testing a program. The metric can be expressed as :

- e - the number of edges in the program graph
- n - the number of nodes
- $v(G) = e - n + 2$ - the cyclomatic complexity number

The study of Curtis (1981) shows empirical evidence that the complexity metrics of McCabe and Halstead relate to the psychological complexity, as expressed in the difficulty in understanding and modifying software. The relation between syntactic complexity and cognitive complexity has been investigated by Khalil & Clark (1989). Shen, Conte & Dunsmore (1983) give a critical review of Software Science. They show that the effort E , as defined by Halstead, correlates with the understandability of programs. . [vdBvdB95].

What we take for our study of K. van den Berg's paper is the adoption of Halstead and McGabe's metrics for functional programs. Using these metrics we could assess the complexity of our functional program and we could correlate it with other metrics.

4.3 Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs - R.Harrison et. al.

Another study that targeted a comparison between imperative and functional languages was that of R. Harrison et. al. [HSDL96] . After comparing C++ with SML they reached to the conclusion that SML code is more reusable than C++ but it also takes longer to be tested; one of the reason why testing takes so much was that SML needed to be reloaded before tests were run.

Their approach was to find a correlation between development metrics such as : number of know errors (KE), time to fix errors(TKE), number of modification requests (MR), time to implement modifications(TMR) and a subjective assessment of complexity(SC) and some code metrics; the code metrics of their choice where the number of non-commented lines (ncsl), number of distinct functions (N'), number of distinct library functions (L), number of distinct domain functions(D), depth of function hierarchy chart, number of function declarations, number of function definitions.

A nice metric for reusability that we can further use in our pursue of code functional-ness is reusability expressed as the ratio between number of library functions called (L) and number of distinct functions (N'): $R = L/N'$.

4.4 Software Metrics: Measuring Haskell - Chris Ryder and Simon Thompson

In their paper, Ryder and Thompson propose a collection of software metrics of Haskell programs. They can be divided into:

- pattern metrics - measuring pattern matching complexity
- distance metrics - scoping length, size, etc.
- callgraph attributes
- function attributes

4.4.1 Pattern metrics

- Pattern size (PSIZ) - number of components of the pattern's abstract syntax tree (AST)
- Number of pattern variables (NPVS) - pattern variables represent new identifiers that are added to the program scope
- Number of overridden/overriden pattern variables (NOPV) - one can mistakenly override some variables already presented in scope when pattern matching
- Number of pattern constructors (PATC)
- Number of wildcards used (WILD) - wildcards may convey information about the structure of items in the pattern, e.g. the position of constructor arguments.
- Depth of nesting - how many patterns are nested in one pattern matching expression e.g. `Car(Wheels(Michelin(_)), Motor(Wolkswagen(100CP(_))))`
 Sum of the dept of nesting (SDNP)
 Maximum depth of nesting (MDNP)

4.4.2 Distance metrics

- Number of new scopes (NNS) - measure the distance between declaration and usage and how many new scopes were introduced in between; this should indicate how complex the name-space is at that use
- Number of declarations brought into scope (NDBS) - how many declarations have been introduced into the name-space 4

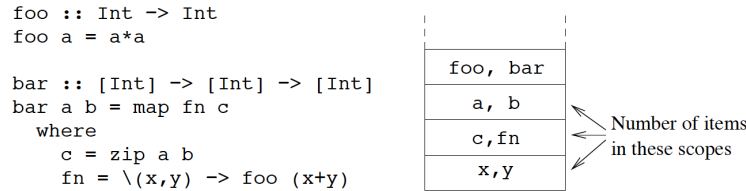


Figure 4: Measuring distance by the number of declarations brought into scope

- Number of source lines - spatial distance in the source code; measuring the distance between the use of an identifier and the import statement that brings it into scope, plus the distance between the declaration and the start of the module in which it is defined. 5

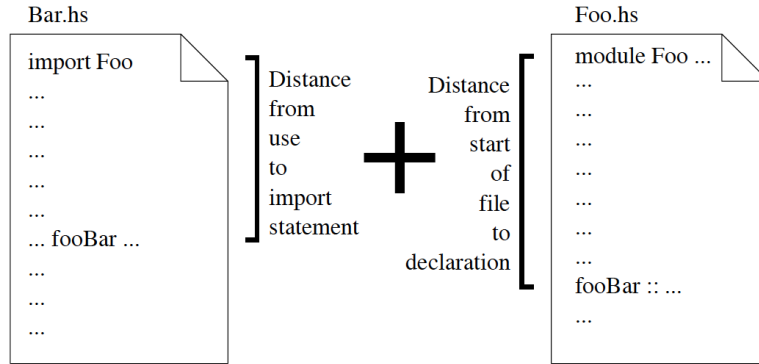


Figure 5: Measuring distance between use of identifier and import statement

- Number of parse tree nodes - count the number of parse tree nodes on the path between two points of the parse tree

4.4.3 Callgraph attributes

- Strongly connected component size (SCCS) - A strongly connected component (SCC) is a subgraph in which all the nodes (functions) are connected (call) directly or indirectly to all the other nodes; as the size of the SCC increases, the number of changes is likely to increase as well, because a change to a single function may cause changes to other functions in the SCC. SCCS is a measure of coupling, similar to imperative and OO coupling metrics such as the Coupling between object classes (CBO), still it measures indirect coupling between functions, not objects (CBO)
- Indegree (IDEG) - The indegree of a function in the callgraph is the number of functions that use it. Functions with high IDEG values may be more important, because they are heavily reused in the program and therefore changes to them may affect much of the program.
- Outdegree (OUTDEG) - The outdegree of a function in the callgraph is the number of functions it calls.
- Arc-to-node ratio (ATNR) - The arc-to-node ratio is a useful indicator of how busy a graph is.
- Callgraph Depth (CGDP)
- Callgraph Width (CGWD) - The subgraph of a function may be cyclic but can be transformed into a tree by breaking its cycle 6

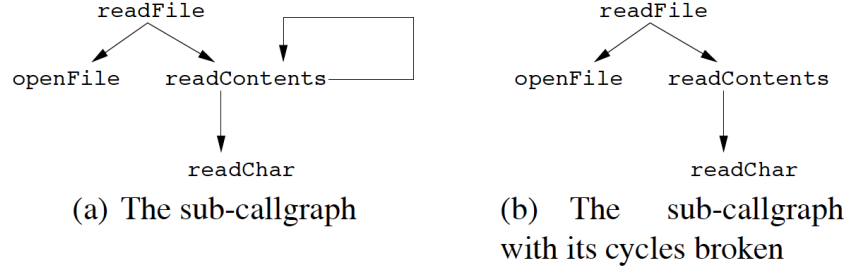


Figure 6: Breaking a cycle of a subgraph

4.4.4 Function attributes

- Pathcount (PATH) - Pathcount is a measure of the number of logical paths through a function
 - Operators (OPRT) - number of function operators
 - Operands (OPRD) - number of function operands
- The last two are a measure of function size. A large function is more likely to be complex than a small one.

In order to validate their metrics, Ryder and Thompson performed analysis over a program's lifetime bug fixes and refactorings. Their conclusions: OUTD metrics is correlated with the number of changes and also SCCS has a significant correlation. They also studied cross correlation between metrics presented above 7.

C1	Sum of the number of scopes Sum of the number of declarations Sum of the number of source lines Maximum number of source lines Average number of source lines Sum of the number of parse tree nodes Maximum number of parse tree nodes Average number of parse tree nodes	C3	NPVS SPDP PSIZ PATC (<i>in Refactoring program only</i>)
		C4	Average number of scopes Maximum number of declarations Maximum number of scopes Average number of declarations
C2	CGDP CGWD	C5	OPRD OPRT

Figure 7: Strongly correlated metrics

4.5 Bullshit

In order to study the maintainability, 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 on 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 correlate them to some process development metrics and this is the hardest part, since process activities are difficult to quantify; most of the research in FP metrics is targeting maintainability, 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 structures could be restructured to FP patterns. Huiqing Li propose a refactoring tool for functional programs written in Haskell [LRT03]; starting from here, we can imagine correlating our metrics with possible refactorings suggestions, of course, this goal is rather ambitious but it's worth a little more study on the matter.

4.6 Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java

Pankratius et. al. performed a comparison study on Scala and Java's parallel programming features and performance [PSG]. Their study shows that writing Scala code results in fewer lines of code than in Java and it preserves the application performance overall. One problem they notice is that Scala's support for parallelism is not as good as Java's. On top of that, Scala seems to require more debugging in testing effort which is quite problematic. One would expect that functional code should lead to fewer, easy to test lines of code.

An interesting point in their conclusion is that the top software developers participating in their study wrote a hybrid program, half functional and half imperative. So what would be the right balance between FP and imperative for a 'good Scala program'?

val, list, map, filter, flatmap, foreach, ::: (list concatenation), :: (list cons

operator) indicate a functional style. We count the occurrences of all such constructs in each project and calculate the percentage of how many belong to the imperative class and how many belong to the functional class. Figure 6 summarizes the results of this analysis for each subject in the study. [PSG]

4.7 Metrics for modularization assessment of Scala and C# systems

Basavaraju Muudu et. al. [MABP13] propose a series of metrics for Scala that target modularity with respect to Functional Programming features like : referential transparency, functional purity, first order functions and also Object Oriented features such as inheritance. They give the following definition for a module: ' A module is a logical collection of related files which can be modified and tested independently. A well modularized system will have low inter-module coupling and high intra-module cohesion. It helps the designers to add new modules easily.[MABP13]. As one can see, this definition does not imply an exact rule for discovering modules, so the system can contain a variable number of modules, depending how one delimits a module.

The technique they chose to use for delimiting system modules 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 metrics for referential transparency - function purity can be easily taken out of the modularity context and be expressed in a general manner, at the level of the whole software system; then a simple description for referential transparency would be expressed as:

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

Where 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.

4.8 Coordinator guidance

4.8.1 Problem definition

Cum ziceam, intrebarea cheie e la ce folosesti aceste metrice.

Punerea problemei in sec. 2 nu e clar definita/motivata: "Scala is a hybrid programming lanaguage [...] there is no mechanism in the language that enforces a functional style [...]"

Si atunci, unde, de ce, cat ar trebui sa scrii functional? Cand Radu Marinescu a inceput sa lucreze definind metrici pentru a detecta probleme de proiectare, acele probleme de proiectare erau in buna parte deja definite, dar trebuiau cuantificate si apoi detectate automat.

Care e stilul de cod bun pe care l-ar propune expertul mentionat de tine? Cum arata un program bine scris in Scala ca limbaj hibrid? Trebuie sa fie perfect functional sau nu? Sigur, Meijer enumera cateva caracteristici functionale. Probabil codul predominant functional e bun, dar nu inseamna ca acela obiectual e prost sau are acele "code flaws" si "bad smells" de care vorbesti in concluzie.

Deci cred ca ar trebui sa te lamuresti (nu in studiul asta) cum arata "codul ideal" pe care vrei sa-l masori. Sa si trebuiasca sa il definesti, sa validezi prin studii ce e bun si nu, si apoi sa gasesti metrici pentru caracterizare/detectie e prea mult.

4.8.2 General critique

O sa ma refer deci la partea de *studiu bibliografic*, care la tine incepe in sec. 4 si hai sa vedem ce aflam (exceptie: aflam in 3.2 ca [MABP13] defineste RTI). Deci: - vdBerg: code complexity in Miranda - Harrison: code modularity in SML - Ryder: metrics for Haskell - Muddu: metrics for Scala Dar NU aflam nici concret ce metrici se folosesc, nici ce concluzii s-au tras pe baza lor.

Apoi vine clasificarea (apropos, ce am scris in mail despre flow-graph si functional-related tu ai scos din context si interpretat altfel: eu am spus ca asta ar fi in completare la bulinele tale, NU o impartire in doua categorii!). Ai cele 4 categorii. E bine ca le-ai extras sintetic si nu pe articol. Dar ce sunt 'pattern-related metrics'? Si de ce ar fi aplicabile mai mult la Haskell ? Cititorul afla foarte putin de aici, ramane mai mult cu o intrebare.

Discutia mai detaliata o ai despre Muddu et al., dar tu descrii mai mult structura pe module decat metrici (singura metrica e cea din 3.2).

Si aici se cam termina studiul bibliografic (mai aflam ca Li a refactorizat ceva Haskell, dar nu ce si cum).

Facand rezumatul rezumatului, esenta e: - o enumerare de metrici (cele 4 buline), fara a *defini* de fapt concret aproape nicicare din ele (exceptie: RTI in 3.2) - trei limbaje pe care altii le-au analizat (Miranda, Haskell, Scala), in doua din cazuri, legat de modularizare.

Daca incerc sa *inteleg*, analizand lista de metrici, constat ca ultimele 3 din 4 buline nu sunt specific functionale, si despre prima bulina, nu aflu nimic; sunt insa si metrici functionale (enumerarea dinainte, dar nu e clar cum le definesc – de exemplu, in 3.2 spui "Further study is needed on elaborating how the pure function should be dened." desi aparent inainte ai citat o definitie clara (deci nu mi-e clar ce e clar si ce nu e clar si de ce).

Pe de alta parte, am avut discutii repetate pe mail, tu mi-ai facut un rezumat al lui Harrison et al., eu ti-am facut rezumatul meu, etc. Unde se regasesc astea

?

Am *citit* in mare toate articolele pe care ti le-am recomandat, si am *retinut* niste informatii, unele interesante, chiar daca nu toate strict legate de intentia ta, dar totusi relevante. Insa in studiul tau gasesc foarte putin care sa ma faca sa *aflu* si sa *inteleg* ce au facut altii. Uite, Pankratius et al (ICSE'12) scrie: "The study of [Harrison et al.] on SML versus C++ done 15 years ago was not conducted with a multi-paradigm language on multicore and unfortunately had implementations that were difficult to compare; however, that study reports similar results to ours that subjects need more effort to test functional code as opposed to imperative code."

E o fraza, dar aflu o informatie utila despre ce au concluzionat altii. Desigur, e relevanta pentru studiul lor (legat de programatori), si nu pentru tine, dar e un exemplu despre cum un articol (care NU e un studiu bibliografic!) ma invata totusi, intr-un rand, despre alt articol.

Sunt dezamagit si pentru ca am petrecut destul de mult timp cautand, citind (a fost placut si instructiv) dar si scriind mesaje, timp in care poate as fi putut sa imi fac sinteze pe alta tema pe care as putea poate sa si public. Iar primul care spune 'am lucrat destul' esti tot tu. Ar mai fi de comentat la concluzii, dar nu ma prea simt motivat in acest context.

Deci, nu vreau sa te imping la un perfectionism exagerat, inteleg ca ai probabil multe de facut, am auzit de la un student zvonuri ca se poate preda studiul si la toamna (nu o sa cercetez sau comentez), pe scurt: ce ai scris imi spune din pacate foarte putin ca studiu bibliografic; ori acesta e scopul sau.

cele bune,

Marius

Salut,

nu cred ca e extraordinar de mult de scris. E bine ca ai clasificat metricele, cred ca se mai poate imbunatati, de exemplu erau flowgraph metrics sau metricele de tip functional (ref. transparency etc.) definite de Muddu et al.

Apoi vine intrebarea cheie la ce se folosesc si la ce vrei sa le folosesti tu. Ryder vrea sa le coreleze cu erorile, dar e foarte incipient (poate e mai mult in teza, desi articolul e ulterior). van den Berg & Broek vor tot ceva similar, dar par ca discuta in alte articole. Harrison compara FP vs. OO, Pankratius discuta experienta programatorilor, Muddu et al analizeaza modularitatea.

Tu vrei sa vezi cat de functional e codul, dar trebuie sa te hotarasti ce sa faci cu informatia asta, mai ales ca Scala fiind un limbaj hibrid, (destul de) rar se programeaza complet functional in el.

O ora e normal daca studiezi articolul in detaliu; e mult pe articol, pentru ca nu tot ce gasesti e realmente relevant. Eu as citi introducerea, concluzii, rezultate experimentale, si apoi daca prezinta interes, tehnica, metodologia etc.

Obs: As folosi mai degraba metrice legate de (procesul de) dezvoltare vs. metrice de cod decat proces/produs, pare mai clar.

Ce as retine: - metrice utile: numar de functii apelate / de biblioteca, raportul dintre ele; numarul de functii definite; adancimea ierarhiei de functii (poate si celelalte) - diferenta in raportul L/N* Secundar (nefiind obiectivul

tau): metrice de eroare (erori / SLOC); discutabil (depinde de mediu, si studiul e vechi: timpul de testare); suprinzator: erori de interfata ???

Pentru concluzia "alegere in functie de gust" nu e nevoie de articole. Partea cu numarul de modificari nu o remarcasem.

Ce as cauta rapid: daca exista elemente suplimentare/clarificari in [1]: "The results of these preparatory studies have been reported in detail [1, 2]".

Observatii: raritatea referintelor e un element potential problematic. Practic, ai 2 (grupuri de) articole de acum 20 de ani (van den Berg si Harrison), unul de acum 10 ani (Ryder et al), si putin recent (Muddu, Kiraly). O explicatie logica e ca FP e inca insuficient de larg folosita, iar cercetarea in metrice e puternic determinata de practica. Pe de alta parte e bine ca Meijer a propus asta ca o tema de master.

Insa trebuie sa-ti pui problema ca metricile (si in general orice numere) nu au impact ca atare, ci doar daca sunt utile la ceva. E usor sa scrii articole in care sa *constati* ca ceva numere sunt diferite, si ceva de constatat se gaseste intotdeauna (asa sunt evaluarile experimentale superficiale). Ceva mai bine e daca poti sa *explici* numerele gasite (aici iar e pericolul de a enunta niste teorii/pareri nefondate). Realmente util e daca pe baza numerelor poti face ceva (la metrice: defecte de proiectare, refactorizare, focus pe testare, etc.)

Deci probabil merita de explorat domeniul 'refactoring (to) functional' pentru a face corelari / a gasi puncte de impact.

4.8.3 References

O teza mai veche despre metrice in Haskell https://kar.kent.ac.uk/14117/1/SOFTWARE_MEASUREMENT

O analiza pentru proiecte in C#/F#: complexitate ciclomatica, dependente <http://fsharpforfunandprofit.com/posts/cycles-and-modularity-in-the-wild/> (eu l-am putut accesa doar via archive.org). Vezi si ulterior <http://evelinag.com/blog/2014/06-09-comparing-dependency-networks/>

O discutie despre posibile metrice intr-un limbaj declarativ <https://sdqweb.ipd.kit.edu/publications/pdfs/kar>

Ryder & Thompson'07: Software Metrics: Measuring Haskell <https://kar.kent.ac.uk/14265/1/tfp-2005-paperFinal.pdf>

—————- Muddu et al. '13: Metrics for modularization assessment of Scala and C# systems (daca nu ai acces via UPT la IEEEExplore ti-l trimit).

CITIT -> Facut rezumat despre modularizare dar nu si despre metrice REZUMAT FACUT despre toate metricile

—————- Tot pe linia de comparare OO - FP, grupul lui Harrison citat de Berg & Broek are un raport tehnic in care printre altele discuta si ceva metrice: <http://cct.brookes.ac.uk/staff/resources/rachelharrison/report3-Harrison-95.pdf>

CITIT -> Concluzie legata de re folosirea codului SML vs C++ : nici o alta diferenta in rest REZUMAT FACUT

—————- Mai ai de exemplu si studiul asta <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2> care se axeaza pe programatori dar discuta tangential si caracteristici functionale/imperative de cod si folosirea lor. M-ar bucura sa aud ca ai cautat activ

si ai mai gasit si tu ceva relevant.

REZUMAT FACUT

Cum ziceam, caută pe Google Scholar să vezi cine a mai citat pe Ryder respectiv van den Berg (nu sunt multe). Vineri gasisem niste articole din 2010 de la Budapesta care nu m-au atras pe moment, dar uite o sinteza din 2013 [http : //ami.ektf.hu/uploads/papers/finalpdf/AMI42from29to44.pdf](http://ami.ektf.hu/uploads/papers/finalpdf/AMI42from29to44.pdf) si teza de doctorat a autorului [http : //www.tnks.inf.elte.hu/vedes/KiralyRolandTeziseken.pdf](http://www.tnks.inf.elte.hu/vedes/KiralyRolandTeziseken.pdf) Deasemenea cred ca merita sa mai cauti prin articolele de refactorizare functionala (chiar daca nu o sa faci asta) pentru ca refactorizarea se face cu un scop, de a imbunatati anumite caracteristici (masurabile), deci implicit sunt gandite pornind de la o metrica.

CITIT

4.9 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: indentation, 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 functional-ness 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.
- [PSG] Victor Pankratius, Felix Schmidt, and Gilda Garretn. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java.
- [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.