A Study of Scala Repositories on Github

Ron Coleman
Computer Science Department
Marist College
Poughkeepsie, New York, United States

Matthew A. Johnson
Computer Science Department
Marist College
Poughkeepsie, New York, United States

***Abstract*—Functional programming appears to be enjoying a renaissance of interest for developing practical, "real-world" applications. Proponents have long maintained that the functional style is a better way to modularize programs and reduce complexity. What is new in this paper is we test this claim by studying the complexity of open source codes written in Scala, a modern language that unifies functional and object programming. We downloaded from GitHub, Inc., a portfolio of mostly "trending" Scala repositories that included the Scala compiler and standard library, much of them written in Scala; the Twitter, Inc., server and its support libraries; and many other repositories, several of them production-oriented and commercially inspired. In total we investigated approximately 22,000 source files with 2 millions lines of code and 223,000 methods written by hundreds of programmers. To analyze these sources, we developed a novel compiler kit that measures lines of code and adaptively learns to estimate the cyclomatic complexity of functional-object codes. The data show, first, lines of code and cyclomatic complexity are positively correlated as we expected but only weakly which we did not expect with Kendall's τ=0.258–0.274. Second, 75% of the Scala methods are straight-line, that is, they have the lowest possible cyclomatic complexity. Third, nearly 70% of methods have three or fewer lines. Fourth, the distributions of lines of code and cyclomatic complexity are both non-Gaussian (*P*<0.01), which is as surprising as it is interesting. These data may offer new insights into software complexity and the large-scale structure of applications including but not necessarily limited to Scala.**

Keywords—Functional programming; Scala; GitHub.com

I. Introduction

Functional programming appears to be enjoying a renaissance of interest for writing practical applications. The turn toward functional programming is evident in recent introductions of new functional languages, revival of old ones, incorporation of functional semantics in non-functional languages, publications of trade texts focused on functional programming, proliferation of open source communities and tools dedicated to functional programming, and adoption of functional approaches by some firms in industry. While reasons for the newfound enthusiasm are likely varied, some proponents have argued that elaboration of the lambda calculus is well suited to writing modular programs that reduce complexity.

What is new in this paper is we tested this latter claim and developed an experimental compiler kit to analyze the modularity and complexity of Scala, a modern language that unifies functional and object styles [1,2]. (While the focus is on Scala, the concept we present is more general and we posit adaptable to the functional style, whether in a pure functional language or a language like Java that recently incorporated lambda expressions beginning with version 8.) We then downloaded from GitHub.com [1] a portfolio of mostly "trending" Scala repositories that contain millions of lines of source in tens of thousands of files with hundreds of thousands of methods written by hundreds of programmers. A robust analysis of this data indicates that lines of code (LOC) and cyclomatic complexity (*M*) [3] are positively correlated, as we expected, but only weakly which we did not expect. In other words, LOC and *M* are clearly related though not necessarily interchangeable as suggested elsewhere in the literature for programs written with imperative languages. While we do not yet know if this new finding is unique to Scala, robust variability statistics indicate *M* is a more reliable estimate of complexity compared to LOC, confirming the distinction of the two metrics, at least for the Scala repositories on GitHub. The data furthermore shows an interesting non-Gaussian preponderance of short, straight-line methods, which also surprised us. That is, we assumed as a null hypothesis that LOC and *M* would be normally distributed about a mean value which they aren't. These new findings may offer insights into software complexity and the large-scale structure of programs including, but not necessarily limited to, Scala.

II. Background

Functional programming for much of its history has thrived largely in academic obscurity [4,5]. That may be changing. A renaissance of interest in "real-world" applications of functional programming, languages, and styles has emerged in recent years [6,7,8,9,10,11,12]. Some investigators have observed that the renewed enthusiasm for functional programming is partly a response to the "free lunch is over" dilemma posed by the advent of commodity multicore systems [13,14]. Others like Twitter, Inc., have switched to functional programming, and Scala in particular, for the advantages Scala purports to offer for scalability. [2] Yet functional programming proponents have long maintained that mathematical expressiveness of the functional style lends itself to modularization and reducing program complexity [15]. That there have been no empirical studies to support these latter suppositions has not stopped language designers, developers, and authors from arguing for more functional programming.

We don't fault functional programming enthusiasts. There isn't even a consensus regarding what software complexity *is*, a conundrum in our view reminiscent of asking what beauty *is*,

[1]See GitHib, Inc., https://github.com/trending?l=scala, accessed: 6 June 2014
[2]See C. Metz, "Twitter jilts Ruby for Scala," The Register, http://www.theregister.co.uk/2009/04/01/twitter_on_scala/, accessed 4 Jun 2014

which Immanuel Kant tackled more than two centuries ago [16]. Perhaps the relationship between software complexity and aesthetics and matters of taste is more than a philosophical one for if software complexity were "in the eyes of the beholder" it might account for the 100-plus different metrics, computational and cognitive, that propose to blindly quantify what is desirable and undesirable in code, the irony notwithstanding [17,18]. Our point is only to suggest that rather than inventing yet another metric for functional programs, we believed it more productive as an experiment to start with existing metrics, off-the-shelf, so to speak; refactor them only if needed; and see what the source code is telling us.

As a candidate, *M* has its downsides, being imperfect and dated [19,20]. Furthermore there is no research on how to apply *M* to functional programs, which differ in some fundamental ways from imperative programs for which *M* had been originally developed. Still *M* remains the most widely known and often applied metric, standing singularly for its diverse implementations[3] and published risk assessments by the Software Engineering Institute [21].

LOC, as a simple measure of complexity, is similarly dated and inadequate [22]. Some modern languages furthermore present semantic challenges for measuring LOC because of nested definitions and structures. Nevertheless the enduring importance LOC, despite their limitations, are evident in modern source editors, IDEs, operating systems, etc. which would be incomplete from a programming point of view without line counting facilities.

Hatton observed for FORTRAN and C that *M* and LOC were statistically correlated, declaring *M* "effectively useless" [23]. Perhaps Hatton made this claim because LOC was so obvious and simple that there had to be a better approach, although for our purposes we show this view of LOC is naïve at best. We don't disagree with Hatton in principle; we would simply state the matter differently. Namely, we expect only as a working hypothesis that any other measure of software complexity is positively correlated with LOC since this view comports with commonsense and anecdotal experience.

III. Why Scala?

We were motivated to study Scala for a few reasons.

1) Scala blends functional and object-oriented styles, which stood out for us as representative of the forward-looking, modern turn toward practical functional programming.

2) Scala is a Java Virtual Machine (JVM) language. This is complementary to the first item above and it means Scala runs virtually everywhere (e.g., desktops, browsers, cellphones, tablets, and GPUs [24], and furthermore interoperates with a large installed base of legacy Java codes. Thus, a study of Scala may be of interest to a broader audience of programmers and researchers.

3) Scala repositories are readily accessible as open source. Some of these repositories, as we show, are large, sophisticated, and deployed in a commercial / production capacity.

4) The Scala open source community provided us with the requisite tools to develop our own tools. We are referring mainly to the Scala plugin for Eclipse (see below) that was created and over the years improved by the Scala community.

5) We are Scala programmers. We have used Scala for teaching and research purposes and we were thus curious to know how our anecdotal experience compared with empirical data.

IV. Sclastic

The main problems, conceptual and programming, were how to apply LOC and *M* to Scala. In summary, the issue for LOC is how to interpret inner definitions. The issue for *M* is handling standard library and user-defined high-order predicate functions.

Thus, we sought to implement an experimental compiler kit capable of solving these problems for a large sample of Scala source codes. We call this kit, *Sclastic* since it operates in an "elastic" manner, that is, it learns to dynamically estimate *M* by discovering the signatures of high-order functions that take predicate (i.e., Boolean-returning) function objects as parameters and storing this information in a database that Sclastic consults during a separate pass.

Sclastic is itself mostly written in Scala and the source is hosted on GitHub. [4] Sclastic is *not* in the portfolio of repositories we analyze.

At a high level, the main body of Sclastic has three phases, each comprising one or more passes. The first phase, the de-commenter, removes comments and empty lines from the input file, which it stores as an in-memory stream of string objects. The second phase, the parser, filters the string stream and identifies lexical objects, methods, scopes, and *decision points*, which we describe below. The final phase, the method compiler, analyzes the parse stream and calculates lines counts and estimates *M* for each method.

This three-phase process outputs a list of Scala objects that other parts of the Sclastic kit analyze for statistical and report generation purposes.

A. Definitions

For the purposes of this paper, we have the following definitions.

- Line. A Scala line is a sequence of zero or more characters terminated by a newline character or the end

[3] For only a smattering of languages see F. Kline, "Cyclomatic Complexity Viewer," http://www.codeproject.com/Articles/10705/Cyclomatic-Complexity-Viewer, accessed 12 Aug 2013; G. Wilson, "Cyclomatic complexity for Python code," http://thegarywilson.com/blog/2006/cyclomatic-complexity-for-python-code/, 9 Jul 2006, accessed 6 Jun 2014; G. Wilson, "Cyclomatic complexity for Python code," http://thegarywilson.com/blog/2006/cyclomatic-complexity-for-python-code/, 9 Jul 2006, accessed 6 Jun 2014; SonarSource, http://www.sonarqube.org/, accessed 6 Jun 2014; and Cyvis, "Software Complexity Visualiser," http://cyvis.sourceforge.net/cyclomatic_complexity.html, accessed 6 Jun 2014

[4] See "Sclastic," http://github.com/roncoleman125/Sclastic, accessed 6 June 2014

of file. In the degenerate case, an empty line has only a newline character or is the end of file. A line may contain one or more comments. A de-commented line is a line with all comments removed.

- Method. A method is a Scala class member function. The function may return an object or it may be void returning in which case it may also be called a procedure. The method may have zero or more formal parameters. Every method is composed of at least one non-empty line.

B. *Line counts*

Per the definition above, counting lines in Scala source, in the simplest cases, is simple. Consider the snippet below.

```
1: class A {
2:   def evens(input: List[Int]):
         List[Int] = {
3:     input.filter(p => p % 2 == 0)
4:   }
5: }
```

Snippet 1

The `evens` method of class `A`, given a list of integers, returns a new list with only the even numbers from the input list. (Note: Scala ignores indents and most whitespace. We added line numbers only for readability; they are not part of Scala syntax.) In this case, the line count of `evens` which Sclastic reports, is the number of lines between and including the inner curly braces of `evens`, that is, three.

Curly braces, however, are often optional in Scala. Consider the snippet of class `B` below.

```
1: class B {
2:   def evens(input: List[Int]) =
         input.filter(p => p % 2 == 0)
3: }
```

Snippet 2

This `evens` method is functionally equivalent to the one from class `A`. However, Sclastic compiles `evens` in this case giving a line count of one.

Counting lines is even subtler since Scala permits inner definitions of classes and methods. That is, it is possible to define classes within classes, methods within methods, and combinations thereof with arbitrary nesting depth. Consider the snippet below, which implements `evens` with the closure, `iseven`.

```
1: class C {
2:   val TWO = 2
3:   def evens(input: List[Int]):
         List[Int] = {
4:     def iseven(a: Int): Boolean =
            a % TWO == 0
5:     input.filter(p => iseven(p))
6:   }
7: }
```

Snippet 3

While the "nominal" line count of `evens` is four (i.e., lines 3 – 6), the "effective" line count is three (i.e., lines 3, 5 and 6). The nominal and effective line count of `iseven` is one (i.e., line 4).

Note furthermore that class `C` has an implied constructor which initializes the member, `TWO`, whenever an instance of `C` comes into existence. (In Scala, a `val` type is a read-only "value" or constant, `final` in Java.)

Sclastic interprets constructors as initializer methods. Thus, while the constructor's nominal line count is seven (i.e., lines 1 – 7), its effective line count is three (i.e., lines 1, 2, and 7).

We define the nominal line count to be the number of lines of a lexical scope including inner definitions. The effective line count is the number of lines of a lexical scope not including inner definitions. For purposes of this paper, we use only the effective line count.

C. *Hard signatures*

The cyclomatic complexity given by McCabe [5] is

$$M = E - N + 2\,P \qquad (1)$$

where E and N are the number of edges and nodes, respectively, in the program flow graph and P is the number of exit points for a given method.

A simplification is to use predicate counting [5,33]. It counts *decision points*, i.e., statements that contain Boolean expressions where an alternate path though the code might be selected. If π is a function, which returns the number of decision points within a method, then the cyclomatic complexity can also be calculated as follows:

$$M = \pi + 1 \qquad (2)$$

McCabe [5] shows that Equations 1 and 2 give the same result for a method whe P=1. In both cases, the smallest value, M=1, means the method consists only of a "basic block" or "straight-line" code.

Using Equation 2 makes calculating M straight forward for languages like Java. In this case, the decision point signatures are the selection and looping statements: *if*, *switch-case*, *for*, *while*, and *do-while*. Since these statements may also contain Boolean expressions connected by logical-and and/or logical-or operators, respectively, *&&* and | | are also decision points in the context of selection and looping statements. For instance, we count an *if* statement as one decision point while we count an *if* statement with an embedded && or | | as two decisions points.

There is yet another simplification which we assume. Namely, if the Boolean expression in the selection or loop is constant true (i.e., there is no decision), it is still counted as a decision point, even though the Boolean expression will never be false.

As the table below suggests, there is an incomplete correspondence between Java and Scala decision point signatures.

TABLE I. JAVA AND SCALA DECISION POINT SIGNATURES

Java	Scala
if	if
for	for (possibly)
while	while
do-while	N/A
switch-case	match-case
&&, ||	&&, ||
N/A	higher-order functions

First, we ignore the *do-while*. It doesn't exist in Scala.

As for the Scala *for* statement, it behaves like the Java *for-each* statement. That is, it operates on a collection and visits every element unconditionally. The Scala `for` statement *may* contain an *if* keyword. However, this case is covered by the *if* signature in Table 1.

We say the above signatures are "hard" in the sense that their signatures are part of the language. Furthermore, we hard-code them in a program table we call the "book" which Sclastic searches when it parses the input source.

D. Hard signatures

Scala also makes decisions in the context of higher-order functions that take Boolean-returning function objects. We call these higher-order functions, *predicate contexts*.

Consider Snippet 1. The function literal, `p => p % 2 == 0`, determines whether an element of the `List` collection is even. The problem is that we must search the source for all references to predicate contexts like `filter`. As we show, the Scala standard library has hundreds of such methods, the signatures of which we can put in the book with the hard signatures.

Doing so solves only part of the problem. It does not allow for the Scala standard library to incorporate new predicate contexts or refactor old ones. Furthermore, a programmer may extend the Scala standard library and add new predicate contexts or create new classes and predicate contexts that are independent of the Scala standard library.

Our solution to these problems was to make two passes over the input during the parser phase. During the first pass the parser identifies method definitions that are predicate contexts, that is, "soft" signatures, which the parser adds to the book. During the second pass, the parser queries the book to identify decision points, hard and soft. (In practice, there are in fact two books, a "hard" one, which is "hard-coded" into Sclastic, and a "soft" one, which is created dynamically and stored in a datbase. A configuration switch tells Sclastic to generate the soft one and stop or load the soft one and continue analyzing the source.)

E. Soft signature miss rate

The book may still be incomplete. Namely, decision points that reference predicate contexts, which are not in the portfolio of repositories, will not be in the book. One solution is to inflate the portfolio with repositories until the book is "closed," namely, all references to predicate contexts are contained in the book. We consider this approach definitive but impractical. The universe of Scala repositories is likely large and not necessarily completely hosted by GitHub.

We have chosen instead to model the potential severity of the problem by estimating the probability of a soft signature not being in the book when it is needed—the *soft signature miss rate*. First, we have the probability of declared imports that that do not have corresponding package exports in the portfolio. This is the *package miss rate*. However, the soft signature miss rate is likely a fraction of the miss package rate since not every imported package contains predicate contexts. For instance, Java imports will not have predicate contexts. In general, the majority of Scala repositories in the portfolio do not contain predicate contexts. Thus, the soft signature miss rate is a joint probability, namely, the missing package rate times the probability that a package has predicate contexts, assuming the two are mathematically independent.

For a random sample of repositories, we model the soft signature miss rate, S, as follows:

$$S \approx k \times w \tag{3}$$

where k is the observed missing package rate and w is the observed fraction of repositories that contain packages with predicate contexts. We observe the parameters, k and w, using the law of succession [25] and frequency data extracted from the portfolio.

Finally, there are reasons we suggest to exclude the Scala compiler / standard library repository from the portfolio. However, our analysis always includes in the book soft signatures from the entire portfolio of repositories, that is, including the Scala repository.

V. EXPERIMENTAL DESIGN

In this section, we describe the experimental design and give summary statistics for the portfolio.

A. Data

We created a portfolio of all the Scala repositories that GitHub identified as trending. This term, "trending," is GitHub terminology, which by GitHub's definition means a repository that "the GitHub community is most excited about". The important thing is that GitHub selects these repositories when we specify the language, "scala." GitHub returns the respective "trending" repositories as hyperlinks on several web pages.

The portfolio starts as a collection of downloaded zip files which are inputs to Sclastic. The portfolio includes the Scala compiler / standard library which are written largely in Scala[5]; the Twitter, Inc., server and libraries [6]; several, large commercially inspired repositories such as Lift [26] and Akka[7]; and many smaller and lesser known repositories for computational finance, graphics, games, networking, web

[5]See "Scala: Object-Oriented Meets Functional," http://scala-lang.org, accessed 6 Jun 2014
[6]See "Twitter is built on open source software," http://twitter.github.io/, accessed 30 Jun 2013
[7]See "Akka," http://akka.io/, accessed 6 June 2014

services, crypto-graphics, and artificial intelligence, among others.

In the case of Twitter, Inc., of its 42 repositories on Twitter's home page on GitHub[6], three were found to be "trending" and the rest we included in the portfolio for the sake of curiosity and possible future research. The other exception was Casbah[8], a repository we had used for initial testing.

We downloaded 262 repositories in total. 85% of the repositories were trending and the rest, 15%, were non-trending, being the 39 Twitter repositories and Casbah. This portfolio consisted of 21,596 source files with 2,391 KLOC (1,519 KLOC with comments and empty lines removed), and 223,493 methods.

The book has 1,187 soft signatures. 471 or approximately 40% are from the Scala repository. The remaining 60% are made from 77 other repositories. The portfolio exports 3,667 unique packages and imports 89,131 packages, 81,285 or 91.2% of which Sclastic found in the portfolio.

The table below gives statistics on the ten largest repositories in the portfolio ranked by number of methods. (All counts are ×1000.)

TABLE II. TEN LARGEST REPOSITORIES IN THE PORTFOLIO.

	Repository	Methods	% tot.	Raw LOC	Stripped LOC
1	Scala	57	26	401	247
2	Scala Test	17	7	300	170
3	Delite	9	4	62	41
4	Lift	9	4	106	58
5	Akka	8	4	105	66
6	SBT-0.13	7	3	45	36
7	Spire-2.10.0	5	2	23	17
8	Scalaz-Seven	5	2	42	28
9	Finagle	5	2	56	41
10	BIDMat	4	2	16	14

These ten largest repositories account for 56% of the methods and 47% of the executable LOC in the portfolio.

B. Setup

We used Eclipse[9], Indigo service release 2 to develop and run Sclastic with the Scala 2.92 compiler and the Scala IDE plugin 3.0.0[10].

We have one Korn shell script. It computes the package miss rate given a list of imports and a list of imports that have no declared class or package in the source. We also have one C program. It calculates, Kendall's τ [27] and MADM [28] statistics.

C. Nonparametric methods

A visual inspection of the distributions of M and LOC suggested the data probably were non-Gaussian. This was in fact confirmed by the Kolmogorov-Smirnov (K-S) test [27]. Thus, we used only robust statistical measures like the K-S test. Two other methods we use are Kendall's τ and *median absolute deviation from the median* or MADM. Kendall's τ is a rank-based measure of correlation, a nonparametric analogue of Person's r. MADM is a rank-based measure of variability which might be considered a nonparametric analogue of the coefficient of variation. We calculate both of these statistics using the kendall.c program included in the Sclastic repository. The interested reader may wish to consult the source code and/or the literature for more details about these statistics.

D. Scatter plots

Our intensions for the scatterplots were to paint a picture of the qualitative relationship between M and LOC. However, since both M and LOC are discrete integer values, we found a simple scatterplot gives a terrace-like picture, obscuring many data points that may be overlaid by other data points. This loses much information. The scatterplots we use attempt to correct this problem by rendering data points at not at x=M×q and y=LOC×q but x=$\Omega(M\times q+\eta_0)$ and y=$\Omega(\text{LOC}\times q+\eta_1)$. Here q is a scaling constant that converts the respective value to pixel units (q is the same in both cases); η_0 and η_1 are uniform random deviates on the interval [-0.50, 0.50); and Ω rounds to the nearest integer. In other words, we render each point without bias within one scaled unit of its location in the chart.

VI. RESULTS

Thus, per Equation 3 we estimate k = 0.088 and w = (1+78) / (262+2), namely, in accordance with the law of succession [25]. Our "best guess" of the soft signature miss rate is $S \approx 0.026$.

All results are based on source after the comments and empty lines have been removed. Furthermore, since the Scala complier/standard library repository is by far the largest in the portfolio, we analyzed the portfolio with this repository and without it to check for any possible bias the Scala repository may have had on the overall results. The table below gives the summary statistics for the portfolio with and without the Scala repository.

TABLE III. SUMMARY STATISTICS WITH AND WITHOUT THE SCALA REPOSITORY

	Portfolio	w/o Scala repos.
τ	0.258	0.274
Median M	1.0	1.0
Median LOC	2.0	2.0
MADM (M)	0.0	0.0
MADM (LOC)	1.0	2.0
Hard decision points	126,432	96,732
Soft decision points	122,202	110,996

The scatter plots below include the Scala compiler/standard library repository and excludes it respectively.

[8]See "MongoDB," http://10gen.com, accessed 6 June 2014
[9]See "Eclipse," http://eclipse.org, accessed 15 Feb 2013
[10]See "Scala IDE for Eclipse," http://scala-ide.org/, accessed 12 Aug 2013

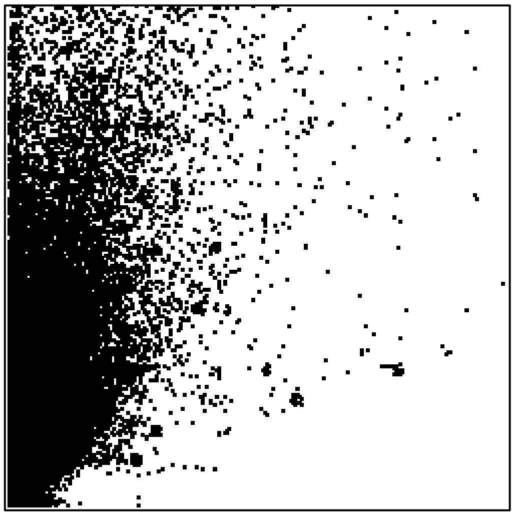
Fig. 1. Scatter plot M (horizontal axis) vs. LOC (vertical axis) including the Scala compiler/standard library repository. Both axes have range [0,50].

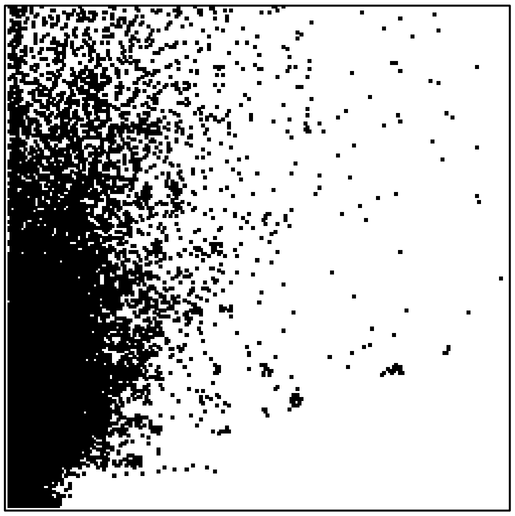
Fig. 2. Scatter plot *M* (horizontal axis) vs. LOC (vertical axis) excluding the Scala compiler/standard library repository. Both axes have range [0,50].

M is the horizontal axis and LOC, the vertical axis. The ranges of *M* and LOC on each axis are 0 - 50 (inclusive) which account for >99% of the data points.

The table below gives the distribution of the first ten *M* per method values across the entire portfolio.

TABLE IV. DISTRIBUTION OF M MEASUREMENTS

M	Freq.	% of total	cum. %
1	167,717	75.1	75.1
2	25,527	11.0	86.1
3	10,969	4.9	91.0
4	6,013	2.6	93.6
5	4,124	1.8	95.4
6	2,287	1.0	96.4
7	1,606	0.7	97.1
8	1,071	0.5	97.6
9	911	0.4	98.0
10	678	0.3	98.3

The chart below gives the *M* per method distribution plotted on a log-log scale.

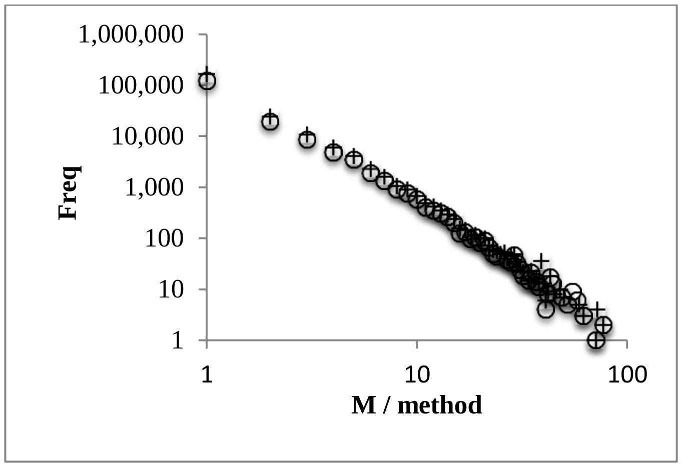

Fig. 3. Distribution of *M* / method plotted on log-log scales including (+) and excluding (o) the Scala complier/standard library repository.

The table below gives the distribution of the first ten *M* per method values

TABLE V. DISTRIBUTION OF LOC MEASUREMENTS

LOC	Freq.	% of total	cum. %
1	100,692	45.5	45.5
2	31,221	14.1	58.6
3	14,212	8.7	68.3
4	9,994	6.4	74.7
5	7,197	4.5	79.2
6	2,287	3.3	82.5
7	1,606	2.7	85.2
8	5,990	2.1	87.3
9	4,700	1.7	89.0
10	3,935	1.5	90.5

The chart below gives the LOC per method distribution plotted on a log-log scale.

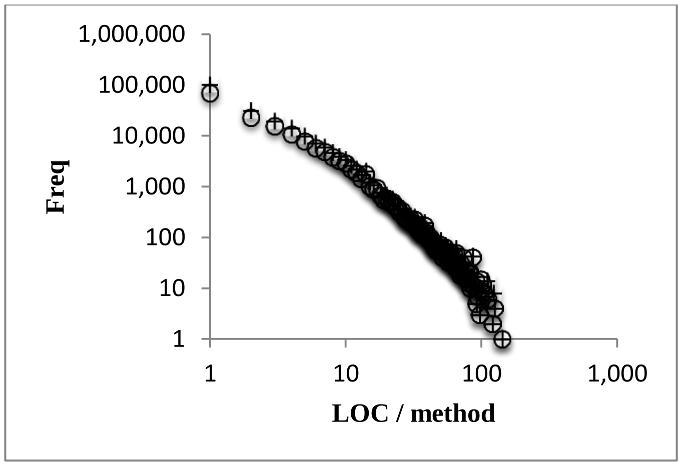

Fig. 4. Distribution of LOC / method plotted on log-log scales including (+) and excluding (o) the Scala complier/standard library repository

VII. Discussion

In this section we discuss and interpret the results.

A. *Soft signature miss rate*

We had noted that estimated soft signature miss rate, *S*, is less than 3%. This suggests that the book, which as we mentioned always contains the Scala standard library, is mostly complete as it contains an overwhelming majority of all the soft signatures required by the portfolio to reliably estimate *M*.

B. *Correlations*

Table 3 shows weak correlation between *M* and LOC including the Scala repository. The correlation also remains weak without the Scala repository. We interpret the weak correlation with and without Scala repository this to mean that the Scala repository does not appear to bias the *M* and LOC correlation. That the correlation is positive agrees with the commonsense expectation we posited at the outset. However, that the correlation is weak tell us that *M* is not "effectively useless" in relation to LOC as Hatton wrote.

The positive but weak correlation would seem to suggest that *M* and LOC are measuring related but nevertheless different phenomena in the source. Some evidence in support of this conjecture is the MADM statistics. That MADM(M) < MADM(LOC) in general indicates that *M* is a more stable measure with less variability compared to LOC. This stands to reason since the range of *M* tends to be smaller than that for LOC. Indeed, this would explain the vertical layering of the scatterplots in Figure 1 and Figure 2. (Note: since $M>0$ and LOC>0, we find no data points on the $x=0$ or $y=0$ axis.) In other words, these data don't contradict Hatton [23] but they also don't fully support it. LOC and *M* are not interchangeable and both metric may be needed to provide a more complete picture of the complexity of Scala codes.

C. *Hard and soft decision points*

We note in Table 3 that there are nearly as many soft decision points as hard ones. The hard-to-soft ratio with the Scala repository is 1.03 and without it, 0.87. This fraction might indicate that overall programmers are exploiting the blend of functional and object styles in Scala, which would make sense. That the Scala repository employs fractionally more hard decision points (126,432-96,732=29,700) than soft ones (122,202-110,996=11,206) is noteworthy as the hard-to-soft ratio is 2.65. We offer only as conjecture the possibility that the Scala repository doesn't reference its own standard library in relative terms. The standard library would be designed and implemented more for reuse by others.

D. *Distributions*

Although the median $M=1$ in Table 3, Table 4 shows that slightly more than 75% of methods have $M=1$. Although the median LOC=2, Table 5 shows nearly 70% of methods have LOC≤3. In other words, most of the code is highly modular and mostly simple. As we pointed out, the K-S test indicates that both of these distributions are non-Gaussian ($P < 0.01$).

In our opinion, this is perhaps even more interesting and surprising. First, on its face, this data tends to agree with claims of functional programming proponents, that is, functional programming encourages highly modular coding. It does not, at least, seem to contradict them. Whether this is unique to Scala or the functional style is unknown. Second, it could be argued that the short and simple methods are mainly "getters" and "setters". We don't know; Sclastic does not distinguish getters and setters from other methods. However, we doubt this is the explanation for the preponderance of short, straight-line methods since Scala obviates the use of such boilerplate in general. Another explanation to consider is programmers are merely following the published style guides by Scala language designers and Twitter, Inc.[11] The problem with this idea is the style guides are only those: guides. Furthermore Scala is a relatively new language and the style guides, as far as we know, are even newer.

There is yet another possibility to account for these distributions. As we pointed out, the distributions and *M* and LOC are non-Gaussian. This was the reason we used robust methods of statistical analysis. First, the charts in Figure 3 and Figure 4 strongly resemble one another. Again, this suggests that with (+) or without (o) the Scala compiler / standard library repository, the general statistical pattern persists. Second, the distributions resemble those distributions of physical and aesthetic phenomena known to follow power-laws [29]. That is, the explanatory model has the form of a homogenous power-law, namely, $f(x) = c\, x^{\alpha}$ where c and α are constants. This notion was tested by [30] which found power-laws offered the best, most parsimonious explanation for distributions and *M* and LOC. The reader will note that, indeed, if we plotted, $\log f(x) = \alpha\, log(x) + \beta$ we would obtain a line with slope α and intercept $\beta = log(c)$. Figure 3 and Figure 4, in this case, $\alpha < 0$, suggest that.

Here we wish to go further and speculate that the *M* and LOC type-distributions as presented in this paper may not be unique to Scala *per se*. Rather, they may be a statistical characteristic of other languages, when studied in the large as the case of our portfolio of Scala repositories. However, similar distributions for other languages have not been reported elsewhere in the literature, which leaves open a research for further study.

VIII. Conclusions

The results we have give in this paper point in a few different directions for future research. One of these is to confirm our findings for other functional programming languages where open source is concern. In this way, we have the opportunity to study possibly many other repositories. We gave a list of candidate languages in the "Background" section. Another direction is to study a language like Java. The promise of Java is we would likely find many repositories on GitHub. Finally, a study of Java repositories, being largely object-oriented at this time (Java 8, which supports lambda expressions, was released in March 2014), offers an opportunity to make some assessment and comparison of the relative contributions of functional and object styles in the data we presented here for Scala.

[11]See "Scala Style Guide," http://docs.scala-lang.org/style/, accessed 9 June 2014 and "Effective Scala," http://twitter.github.io/effectivescala/, accessed 9 June 2014

ACKNOWLEDGMENTS

The authors thank the reviewers for their helpful feedback.

REFERENCES

[1] M. Odersky, L. Spoon, B. Venners, Programming in Scala: A Comprehensive Step-by-Step Guide, Artima, 2011

[2] M. Odersky, T. Rompf, "Unifying Functioal and Object-Oriented Programming with Scala," CACM, vol. 57, no. 4, April 2014

[3] T. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, Decemember 1976

[4] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages, ACM Computing Surveys, vol. 21, no. 3, 1989

[5] G. O'Regan, A Brief History of Computing, Springer, 2010

[6] C. Emerick, B. Carper, C. Grand, Clojoure Programming, O'Reilly, 2012

[7] M. Fogus, Functional JavaScript: Introducing Functional Programming with Underscore.js, O'Reilly, 2013

[8] M.R. Hnsen, H. Rischel, Functioal Programming Using F#, Cambridge University Press, 2013

[9] G. Michaelson, An Introduction to Functional Programming Through Lambda Calculus, Dover, 2011

[10] S. St. Laurent, Introducing Erlang, O'Reilly, 2013

[11] S. Thompson, Haskell: The Craft of Functional Programming, Pearson Education Ltd, 2011

[12] D. Wampler, Functional Programming for Java Developers: Tools for Better Concurrency, Abstraction, and Agility, O'Reilly, 2011

[13] H. Sutter, The Free Lunch is Over: The Fundamental Turn Toward Concurrency in Software, Dr. Dobbs Journal, vol. 30, no. 3, 2005

[14] R. Coleman, U. Ghattamaneni, "Parallel Collections: A Free Lunch?," Journal of Computer Science and Engineering," vol. 17, issue 2, 2012

[15] J. Hughes, "Why Functional Programming Matters," Research Topics in Functional Programming, ed. Turner D, Addison-Wesley, 1990, pp. 17–42

[16] I. Kant, The Critique of Judgment (1790), translation by J. C. Meredith, Oxford University Press, 1978

[17] E. Weyuker, "Evaluating Software Complexity," IEEE Transactions on Software Engineering, vol. 14, no. 9, Sep. 1988

[18] D. Tran-Cao, G. Lévesque, J. Meunier. "A Field Study of Software Functional Complexity Measurement." Proceedings of the 14th International Workshop on Software Measurement, 2004

[19] G. Gill, C. Kemerer C, "Cyclomatic Complexity Metrics Revisited: An Empirical Study of Software Development and Maintenance," CISR WP No. 218, Sloan WP No. 3222-90, 1990

[20] N. Pataki, A. Sipos, Z. Porkolab, "Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric," Proc. of ECOOP 2006 Doctoral Symposium and PhD Students Workshop, 2006

[21] SEI, "C4 Technology Reference Guide, Software Engineering Institute," Carnegie Mellon, 1997

[22] C. Archer "Measuring Object-Oriented Software Products," Software Engineering Institute, Carnegie Mellon, 1995

[23] L. Hatton, "The role of empiricism in improving the reliability of future software," TAIC, 2008

[24] N. Nystrom, W. White, K. Das, "Firepile: GPU Programming in Scala," GPCE, 23 Oct 2011

[25] E.T. Jaynes, Probability Theory: The Logic of Science, Cambridge, UK, Cambridge University Press, 2003

[26] T. Perrett, Lift in Action: The Simply Functional Web Framework for Scala, Manning Publications, 2011

[27] J. Conover J, Practical Non-Parametric Statistics, Wiley, 1995

[28] D.C. Hoaglin, F. Mosteller, J.W. Tukey, Understanding Robust and Exploratory Data Analysis, Wiley-Interscience, 2000

[29] M. Schroeder, Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise, Dover, 2009

[30] R. Coleman, M. Johnson, "Power-Laws and Structure in Functional Programs", Proceedings of the 2014 International Conference on Computational Science & Computational Intelligence, Las Vegas, NV, 10 – 13 Mar, 2014, IEEE Computer Society CPS.