Software Engineering Properties of Functionally Enabled Languages

Georgios Gousios
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
Email: gousiosg@aueb.gr

***Abstract*—A new trend in programming languages and system design is the use of constructs derived from the functional language field. Startups requiring fast product turnover and large corporations looking for increased maintainability are exploring the use of new, purely functional (such as Erlang or Haskell) or functionally-enabled (such as Scala and Ruby) languages, on the basis of decreased complexity and higher productivity. Despite the apparent increase in their use, the software engineering properties, including the alleged advantages, of such languages are largely underexplored. In this paper, we discuss the issues that prohibit the use of classic complexity and productivity metrics and present the rationale behind a new set of metrics that targets this increasingly important set of languages.**

***Keywords*-Functional Languages, Software Engineering, Metrics, Complexity, Productivity**

I. Introduction and Motivation

Arguably, the bulk of code today is written in the imperative style. This fact has led software engineering researchers to put particular emphasis on the study of the properties of this programming paradigm. Over several years of efforts, a large body of knowledge has been assembled [1], consisting of metrics, quality and cost models and a plethora of tools, while the interplay of development processes and structured software is one of the hot topics of current research efforts.

Due to a variety of reasons, including the advent of multicore architectures, the rising rate of information production [2] and the necessity to reach the market fast, currently, large corporations and start-ups alike are investigating alternative programming and information storage models. At the same time, a number of languages that combine the functional and object-oriented paradigms or are purely functional have emerged and are beginning to capture the interest of developers. Moreover, popular scripting languages that have already incorporated functional characteristics form the basis of large scale web sites and have delivered novel mechanisms to fast web development. The result of the combination of new programming needs with the availability of new tools allowed functional programming ideas to proliferate.

Functional programming has been an active field of research for more than 50 years, at least since the publication of McCarthy's seminal paper on Lisp [3]. In the mean time, the mainstream programming paradigm has shifted from assembly programming to Fortran, then to structured programming and finally to object orientation. Despite being one of the favourite research topics of theoretical language designers, the concepts underlying functional programming, even though simple in theory, represent a major departure from what most programmers use in every day practice. Side effect free programming, variables that cannot really vary, lack of shared state, monads and pattern matching can confuse at first even battle-hardened object-oriented software designers; however, the very same concepts allow functional programs to be short and concise, while enabling them to be inherently parallelisable, a very important feature in the multicore era.

During the last few years, we are witnessing a slight but noticeable shift towards functional programming. Scripting languages, notably Python and Ruby, pioneered the introduction of functional concepts, such as list comprehensions and lambda functions, to mainstream programming. A new wave of programming languages, developed to overcome the expressiveness and complexity limitations exhibited in mainstream languages, have promoted functional constructs, such as type safe pattern matching, higher order functions and single assignment variables, to first class citizens (Scala). New, purely functional, languages have emerged to fill in the remaining gaps (F#), often introducing significant advancements in their field of specialisation (Erlang). We collectively refer to those languages with the term *functionally enabled.*

As Table I shows, the use of functionally enabled lan-

Table I
Metrics of language usage from Ohloh.net, indicating increasing use of functionally enabled languages. Measurements are approximate and depend on voluntary registration of projects with the measurement service.

Metric	PHP	Scala	Python	Ruby	Haskell	F#	Erlang
Number of commits ($\times 10^3$)							
Jun 08	52	800	52	18	700	20	700
Jul 10	44	1600	42	18	500	200	1200
Lines of Code ($\times 10^6$)							
Jun 08	51	0,01	18	9	0,04	0,001	0,03
Jul 10	49	0,3	15	9	0,02	0,01	0,05
Number of contributors							
Jun 08	3800	55	4000	1200	90	5	50
Jan 10	3700	140	3100	900	50	25	90

guages is on the rise. However, as functional programming has not caught mainstream attention yet, even though it does make an appearance in specialised fields such as financial engineering [4], [5], it has been living below the radar of software engineers. Indeed, very little research, if any, has been carried out on how software is written with the functional or the object-functional paradigm. Both in the scientific literature [6], [7], [8] and in opinion writings [9], [10], the consensus is that functionally enabled languages allow experienced developers to express algorithms more intuitively and result in significantly more concise and maintainable code. As the problem has not been explored yet, some authors beg to differ [11, Chapter 12].

In this work, we discuss the shortcomings of the current state of software engineering research in the field of functional and functionally enabled languages (which, in our definition, also includes popular scripting languages). We formulate a set of open research questions and provide indications of how those could be potentially answered. Our aim is to study whether functionally enabled languages keep up to the promises made by their designers, namely whether their functional characteristics improve developer expressiveness and reduce code complexity. In studying those attributes, we will also devise new ways of assessing code complexity in the presence of functional constructs and methods for comparing the productivity between programming paradigms.

II. Charisteristics of Functionally Enabled Languages

Functionally enabled languages include, or allow the straightforward definition of, special syntactic constructs that are seldom part of generic imperative programming languages. In this section, we analyse the syntactic constructs that renders the development with functionally enabled languages unique and examine their effect on current software engineering practices. Based on the observations made, we also formulate a set of research questions and propose ways to explore them.

Syntactic constructs are those elements of a programming language that enable programmers to express abstract computations in series of steps. Syntactic constructs are either directly implemented at the language level (for example, loop constructs in imperative languages, pattern matching in functional languages) or composed by combining lower level syntactic constructs (e.g. monads). A unique characteristic of functionally enabled languages is that they allow the composition of powerful constructs due to their support for abstracting computation descriptions from computation implementation, through the use of high order functions. The following list presents a collection of syntactic constructs that are unique to functional programming and form part, in various degrees, of many functionally enabled languages.

- Higher Order Functions are constructs that can receive functions as input and/or produce functions as a result. Higher order functions are used to provide generic computation primitives that work irrespective of the underlying data and computational details.
- Lambda Functions are the basic constructs of λ-calculus, the theoretical basis of functional programming [12]. In programming languages, they are implemented as anonymous functions that are passed as arguments to higher order functions or perform a computation in place of their definition.
- Closures are functions defined within the body of other functions and which share variables with their enclosing counterparts. They are often used as parameters to generic high order functions in order to make a computation specific to a context.
- List Comprehension is a syntactic construct that allows the definition of a list based on existing lists and a filtering function. It is used to dissect data or to initialise copies of lists.
- Pattern Matching is a technique for processing data based on its structure or contents. It is used as a method of dispatching certain operations based on characteristics of processed data structures.
- Immutable Variables are variables which cannot be assigned after initialisation. In most common functional languages, variables are by default immutable. This enables side effect free execution, since function calls cannot affect program state, which consequently enables lock-free parallelisation of programs adept to divide and conquer strategies (most data parallel programs).
- Monads are high level composable computation descriptions that are used in functional environments to contain computations with side effects [1]. Monads form the basis for safe sharing of program state in pure functional languages.

Table II presents an overview of the features that are special to functional programming, as they apply to the most popular functionally enabled languages. The table shows that functional constructs have proliferated from the pure functional Haskell language, which predated all other languages shown in the table, to the other languages as well. This means that, while typical languages tend to retain their original orientation, they also have special syntactic constructs that must be examined separately. For example, in the case of Scala, while typical object oriented design patterns can be implemented, their implementation is quite different from typical object oriented languages, exactly due to its functional characteristics. Indicative of the fact is the example in Table II, where Scala's pattern matching and type

[1]In functional programming, a computation is said to have side effects if during its execution, it somehow affects the program's shared state, e.g. by writing to a file or raising a runtime exception.

Table II
FUNCTIONAL PROGRAMMING FEATURES IN VARIOUS MAINSTREAM LANGUAGES

Feature	**Object Oriented**	**Scripting**		**Pure Functional**		
	Scala	Python	Ruby	Haskell	F#	Erlang
High order functions	✓	✓	✓	✓	✓	✓
Lambda functions	✓	✓	✓	✓	✓	✓
Closures	✓	✓	✓	✓	✓	✓
List comprehensions	✓	✓	⋋	✓	✓	✓
Lazy evaluation	⋋	⋋	⋋	✓	⋋	
Pattern Matching	✓	⋋		✓	✓	✓
Immutable variables	✓		⋋	✓	✓	✓
Monads	⋋	⋋	⋋	✓	✓	

✓: full support, ⋋: partial support or implemented in library

inference helps reduce boilerplate code by 50% compared to Java in the implementation of the omnipresent Factory design pattern. On the other hand, the implementation in Ruby is smaller than Java but lacks the type safety features of Scala (one can pass arbitrary values to the `create` method at runtime).

The example presented above is indicative; apart from reduced code size, there are other software engineering properties that are enabled by functional characteristics. As an example, immutable variables and side effect free functions help programs become inherently paralelisable; problems that can be parallelised are usually more easily expressed in functionally enabled languages, as there is no shared state to be guarded with explicit synchronisation. The lack of shared state also allows easier debugging, since a function call trace contains all data affected by the failing operation. Moreover, high order functions enable the expression of generic computation primitives irrespective of the underlying data; such programming style has been the basis of large data processing algorithms, such as Google's MapReduce [13] (modelled along Lisp's `map` and `reduce` list processing high order functions).

III. OPEN RESEARCH TOPICS

A. Programming practices in functionally enabled languages

Syntactic constructs are used and combined by programmers to convert architecture-level designs of systems into series of executable operations. The organised use of syntactic constructs leads to a programming practice or style, a repetitive application of specific syntactic constructs to express certain classes of algorithmic steps. The programming practices employed in common programming paradigms have been extensively analysed in the literature [14], [15], [16]. Functional programming practices are distinctive and usually only used in special program cases. The research question that emerges is what types of designs are more adept to functional solutions and what syntactic constructs do programmers use to express such designs. To answer this question, one has to identify and categorise all functional programming constructs (the list presented above and in Table II is not exhaustive) by means of a systematic review and then devise methods to automatically extract them from existing source code in order to conduct an extensive exploratory study.

Combining syntactic constructs is not an arbitrary operation; common design problems call for similar or pattern-like solutions. Design patterns offer a convenient way to capture, document and disseminate existing knowledge from a given area in a consistent and accessible format. In object-oriented programming, design patterns have been identified early on and their thorough documentation [14] has lead to widespread adoption. In this context, an important research question is whether design patterns exist in source code developed with functionally enabled languages and, it they exist, what is the degree of their application in existing software. To answer this question, one has to work on identifying related design patterns in both the literature and in practice and develop a mechanism to automatically identify them in existing code. An exploratory study will confirm the existence and document the applicability of the identified patterns in real programs.

Furthermore, most languages are used in the context of specific application domains; for example `C` is used mostly in systems-level software while Java has almost monopolised the application server market. What problem domains are functionally enabled languages good for? How are they used to solve particular classes of problems? Are the functionally enabled languages particularly able for specific kinds of problems? Answers to these questions could be found by the classifying the relevant literature, to identify documented uses of functional languages in specific projects, and by performing exploratory case studies on metadata from project hosting sites.

B. Complexity in functionally enabled languages

The majority of problems that software engineers face today are, by nature, complex. Software complexity is a term that encompasses several aspects of software design and implementation quality. Software complexity usually arises when applying suboptimal algorithmic and structural designs on complex problem spaces. Excessively complex code or systems can lead to faults and consequently complexity

Table III
SINGLETON-BASED FACTORY PATTERN IN JAVA (LEFT) AND SCALA (MIDDLE) AND RUBY (RIGHT).

```java
public interface Car() {...}
public RaceCar implements Car {...}
public NormalCar implements Car {...}
public class CarFactory {
    private CarFactory instance;

    private CarFactory() {}

    public void getInstance () {
        if (instance == null)
            instance = new CarFactory();
        return instance;
    }

    public Car create(String type) {
        if (type.equals("Race"))
            return new RaceCar();
        else if (type.equals("Normal"))
            return new NormalCar();
        else
            throw new Exception();
    }
}

Car myCar = CarFactory
    .getInstance().create("Race");
```

```scala
trait Car {...}
class RaceCar extends Car {...}
class NormalCar extends Car {...}
object CarFactory {
 def apply(String type) {
  type match {
   case "Race" => new RaceCar();
   case "Normal" => new NormalCar();
   case _        => throw new Exception;
  }
 }
}
val myCar = CarFactory("Race")
```

```ruby
require 'singleton'
class RaceCar < Car
class NormalCar < Car
class Car
  include Singleton
  def create name
    case
      when name == "Race"
        RaceCar.new
      when name == "Normal"
        NormalCar.new
      else
        raise type
    end
  end
end
myCar = Car.create("Race")
```

measures have been proposed as fault predictors [17], [18], [19] or as maintainability estimators [20].

Researchers have long been trying to evaluate complexity and therefore metrics that assess complexity at various levels of analysis have been proposed. At the algorithmic level, McCabe's cyclomatic complexity [21] examines the control flow graph of a function and calculates a measurement of its complexity by enumerating the number of possible execution paths a program function has. At the module level, Henry and Kafura's Information Flow metric [22] relates a modules complexity to the number of cross-references between the module and other modules. In a similar fashion, Sneed proposed and evaluated [23] several inter-module data flow metrics. Moreover, object oriented languages exhibit additional structural characteristics (i.e. inheritance, state encapsulation); the Chidamber and Kemerer suite of metrics [24] provide insights on the structure of object oriented programs. On the functional languages front, little research was done [25], [26] and the majority of it consists of adaptations of existing metrics to the purely functional language Haskell.

Functionally enabled languages however have special syntactical constructs that are not accounted for in complexity metrics, especially those evaluating complexity at the algorithmic or structural level. As an example, Table III-B presents the same simple algorithm (recursive Fibonacci sequence calculation) implemented in 4 different languages, 3 of which can be classified as functionally enabled. The Scala and Erlang implementations use pattern matching, while the Haskell implementation uses list comprehension and lazy evaluation. In trying to apply control flow based metrics, such as McCabe's cyclomatic complexity, to assess complexity, we hit at least three walls: (i) is pattern matching functionally equivalent to a branch statement? (ii) The Haskell version has no branches (the Scala version could be written similarly), does this mean that its complexity is 0? and (iii) what is the unit of application (similar to a function in structured programming) for applying the metric?

The situation also extends to structural metrics, even though in this case, the primary factor affecting metric applicability is the unit of information hiding employed by the language, and not its functional characteristics. For example, in the case of Scala, and partially Ruby, the Chidamber and Kemerer metrics could apply unchanged, as they are both object oriented languages in principle. Pattern matching (especially, typed pattern matching as in Scala's case classes) and side effect free execution (where state is encapsulated in monads) make structure metrics less useful in object functional and pure functional environments, respectively.

Research Opportunities: From the description provided above, it becomes apparent that a new approach that encapsulates the functional characteristics of functionally enabled languages is required. What we propose is an approach based on control flow analysis [27]. Control flow analysis is a suite of analytical techniques that approximate runtime control graphs of functional and object oriented programs [28]. Recent advances [29] have (theoretically) bridged the gap between data flow analysis in the object oriented and functional paradigms. While control flow analysis for structured

Table IV
DIFFERENT WAYS OF CALCULATING FIBONACCI SEQUENCES

```
int fib (int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n − 1) + fib (n − 2);
}
```

Standard recursion in C

```
def fib (n:Int) = fib_tr (n, 0, 1)

def fib_tr (n: Int, b: Int, a: Int): Int =
    n match {
        case 0 => b
        case _ => fib_tr(n − 1, a, a + b)
    }
```

Pattern matching and tail recursion in Scala

```
fib(0) −> 0 ;
fib(1) −> 1 ;
fib(N) when N > 0 −>
        fib(N − 1) + fib(N − 2).
```

Pattern matching in Erlang

```
fib :: [Int]
fib = 0 : 1 : [ a + b |
        (a, b) <− zip fib (tail fib)]
```

List comprehension and lazy evaluation in Haskell

programming is a relatively simple task, in the object oriented and functional paradigms dynamic dispatch and high order functions, respectively, do not allow the construction of accurate control flow graphs, the basis for metrics such as McCabe's complexity. By applying such tools, and expanding accordingly a researcher could investigate the complexity of the control flow in functionally enabled languages, as the analysis will go beyond the employed syntactic constructs. If the call graph is reconstructed or approximated, a possible complexity metric could simply count the number of states in the call graph, per unit of execution or per unit of information hiding.

An alternative approach would entail the construction and parsing of a program's parse tree, using tools from the language's compiler or interpreter suite. In that case, a complexity metric would be more straightforward to calculate, for example by counting the number of statements that a compiler considers as branches in the program's abstract syntax tree. This method however is language and compiler specific, potentially lacking the generality of the former one.

After the implementation of the complexity evaluation algorithm, it must be validated, while its usefulness must be demonstrated and evaluated. For validation, we advocate cross examination of the metric results by experienced developers in a controlled experiment setting. The usefulness of a metric can be evaluated empirically. Using a suitable method, for example the Goal-Question-Metric approach [30], a researcher can formulate a set of research questions that use the results of the metric to predict a behaviour of the system. A good starting point would be the correlation of the metric results at the module level with the number of bugs affecting the module.

C. Productivity and estimation with functionally enabled languages

One of the most commonly cited reasons for developing a project with a functionally enabled language is that of increased productivity over an imperative language. The consensus is that functional constructs enable faster and cleaner algorithmic expression, while they help cut on boilerplate code (e.g. field accessors, structure initialisation) found in imperative languages. Moreover, functionally enabled languages are considered to exhibit increased statement density and therefore the programmer can do more with a statement in comparison to procedural languages. While such claims are omnipresent, to the best of our knowledge, they have not been evaluated scientifically.

Productivity is a recurring discussion in all processes that involve inputs and outputs. In economic terms, productivity is the ratio of output to input, the output of a process divided by the effort required to produce it. In [31], programmer productivity is defined as the ratio of the delivered source lines of code to the total effort in man-months required to produce the delivered program. Input and output in software engineering processes are frequently addressed with output usually measured in lines of code [31], [32], [33]. As the lines of code metric cannot be determined safely before the end of the project, function point analysis usually complements it. Even though lines of code is a ubiquitous measure of work volume, its use for productivity measurements is not without criticism. The arguments against it included its inability to cater for semantic differences across languages, the fact that not all statements are of the same complexity and the fact that software developers today do a lot in the context of a software project apart from coding.

Related to productivity is the issue of project estimation.

The problem can be roughly explained with the following question: Given a software project description, how can we estimate its cost and duration? The prevailing current practice involves the use of pre-calibrated regression models with estimated size measures as sole input. Examples of such models include COCOMO [34] and SLIM [35]. Both models have been calibrated with data from existing projects and moreover COCOMO in its second iteration includes configuration support for a large set of project parameters. However, both models are very sensitive to the language used for the implementation. Specifically, COCOMO II explicitly defines lines of code as logical statements, which in the case of functionally enabled languages, are believed to be considerably more dense. Moreover, the same model, uses a static conversion matrix between function points and lines of code, again signifying the effect of the programming language on the prediction result. There is no written report of the application of either model on the prediction of systems developed with functionally enabled languages.

Research Opportunities: Discovering a method to objectively measure the expressiveness of functionally enabled languages is a very important research question. One way to tackle it would be to compare functional statements with functionally equivalent statements in generic languages. On the small scale, a researcher will need to identify functional syntactic constructs and re-implement them with generic procedural code. After confirming the applicability of the approach, the researcher will need to conduct a large scale analysis of existing software to quantify the extend of source code savings (if any).

A complementary approach entails the evaluation of how many lines are required to develop a function point, in a way similar to how COCOMO interchanges function points to source lines of code using pre computed equivalence tables for several programming languages. This approach requires access to rich process data (requirement documents, among others) for projects developed with functionally enabled languages and therefore it may not be applicable to data freely available for empirical research. Industry reports would be of great importance on that front.

Finally, in order to examine whether programming with functional languages is indeed faster, a controlled experiment with real developers could be very useful. The experiment should be designed to employ professional developers to work on a non-trivial algorithmic implementation in a limited timeframe, initially using their language of choice and then (after having fully understood the problem) using a functionally-enabled language they have never used before. The aim would be to measure whether the functionally enabled language will enable them to write faster or will make debugging faster. A questionnaire could then be used after the experiment to further elicit the developer's experience.

IV. Research Design

The proposed work is part of a research project that aims to shed light on why and how functionally enabled languages are gaining traction. Our interest in the project derives from the field of software engineering. The project's high level goal is to investigate how concepts derived from functional programming affect software development today. To achieve the stated goal, we will examine two major, and in many ways orthogonal, programming techniques, object orientation and functional programming, in order to establish (or not) the relative strengths of the functional paradigm. To examine real cases, we will use existing freely available OSS, written in popular functionally enabled languages, such as Ruby, Scala and Erlang.

To tackle the research problem, we decompose it in the following units of work:

1) Establishment of the functional programming concepts and idioms in use by existing and emerging languages.
2) Identification of how those concepts are used and combined with other paradigms in practice, i.e. the patterns of combining functional and imperative programming.
3) Assessment of the effect of functional concepts on software and algorithmic design. Does mixing functional and imperative concepts help in reducing software complexity and thereby increase maintainability?
4) Assessment of the effect of functional concepts on programmer productivity. Is it really "faster" to write software in the functional style?
5) Recommendations based on the knowledge acquired from the previous steps. What did we learn and how can we apply it in real projects?

Our research effort will be mostly empirical, as we will examine existing systems and practices. To tackle the first work item we will perform a systematic literature review [36]. The majority of the remaining work consists of observational studies. We plan to conduct several large scale exploratory case studies to establish the measurements required for tackling work items 2 and 3, and also confirmatory case studies, possibly in collaboration, to answer specific details of work item 3. Moreover, to answer the question of whether functionally enabled languages are faster to develop with, we plan to perform a controlled experiment with live subjects.

To investigate our hypotheses, we will use freely available data from the OSS ecosystem. A large number of both trivial and non-trivial systems written in functionally enabled languages, including the implementation of the languages' libraries, exists in various open repositories, such as Google Project Hosting ($>$1000 projects available), GitHub (> 5000 projects available) and SourceForge (> 500 projects available). Using methods described in our earlier work and elsewhere [37], [38], we will mirror both product data (source code repository) and process data (email archives and/or bugs, depending on availability). This work will be performed early on in the project's cycle.

A large part of the proposed work involves experimentation with existing software systems. For that, we will build on our previous work [39] on large scale empirical software engineering experimentation and will extend the Alitheia Core platform [37] to support the languages that we will work with. Apart from benefiting from Alitheia Core's deep analytics on software repository data, thus obtaining a historical and social network perspective, our work will also benefit from an easily extensible repository of data originating from several hundred of OSS projects and the future developments of the platform.

V. Conclusions and Future Work

Functionally enabled languages form an emerging trend in software development, for reasons that reportedly have to do with faster development turnover and more straightforward multiprocessor programming. Despite the increasing volume of software being written in such languages, their properties have not been investigated by the software engineering community. In this paper, we provide the first account of the issues that emerge when trying to apply traditional software engineering tools (e.g. complexity and productivity metrics) on functionally enabled languages, along with potential solutions.

To the best of the author's knowledge, the proposed work is the first to study the emerging trend of writing software in functionally enabled languages. As such, we believe that we will be able to uncover interesting facts and answer the questions of why developers are increasingly preferring those languages for new projects. Our focus will be on performing large scale studies in order to assess the current state of practice and construct a solid shared body of code for experimentation with such languages, similar to the one in we constructed in previous work.

Currently, we are investigating ways to implement the complexity metric, as described in Section III-B.

References

[1] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, Eds., *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[2] P. Lyman and H. R. Varian, "How much information 2003?" University of California at Berkeley, Tech. Rep., 2003.

[3] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[4] Y. Minsky and S. Weeks, "Caml trading — experiences with functional programming on Wall Street," *Journal of Functional Programming*, vol. 18, no. 4, pp. 553–564, Apr 2008.

[5] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard, "Commercial uses: Going functional on exotic trades," *Journal of Functional Programming*, vol. 19, no. 1, pp. 27–45, Jan. 2009.

[6] J. Hughes, "Why functional programming matters," *Computer Journal*, vol. 22, no. 2, pp. 98–107, 1989.

[7] P. Hudak and M. P. Jones, "Haskell vs. Ada vs. C++ vs. AWK vs. ...an experiment in software prototyping productivity," Yale University, Dept. of CS, New Haven, CT, Tech. Rep., Jul 1994.

[8] K. W. Ng and C. K. Luk, "A survey of languages integrating functional, object-oriented and logic programming," *Microprocessing and Microprogramming*, vol. 41, no. 1, pp. 5 – 36, 1995.

[9] P. Graham, *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly Media, 2004.

[10] J. Spolsky, *Joel on Software*. Apress, Aug 2004.

[11] D. Spinellis and G. G. (editors), *Beautiful Architecture: Leading Software Engineers Explain How They Think*. Sebastopol, CA: O'Reilly Media, Inc, 2009.

[12] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–150.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison Wesley, 1994.

[15] B. Meyer, *Object-oriented software construction*, 2nd ed. Prentice-Hall, 2000.

[16] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, "Object-oriented analysis and design with applications," 2007.

[17] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison Wesley Professional, 2003.

[18] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[19] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 91–100.

[20] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[21] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, pp. 308–320, 1976.

[22] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, pp. 510–518, 1981.

[23] H. Sneed, "Understanding software through numbers: A metric based approach to program comprehension," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 6, pp. 405–419, 1995.

[24] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, Jun 1994.

[25] K. den Berg, "Software measurement and functional programming," Ph.D. dissertation, University of Twente, June 1995.

[26] C. Ryder and S. Thompson, *Trends in Functional Programming*. Kluwer Academic Publishers, September 2005, ch. Software Metrics: Measuring Haskell.

[27] J. Midtgaard, "Control-flow analysis of functional programs," University of Aarhus, BRICS Report Series RS-07-18, December 2007.

[28] O. Shivers, "Control flow analysis in Scheme," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA: ACM, 1988, pp. 164–174.

[29] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis," in *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2010, pp. 305–315.

[30] V. Basili, C. Caldiera, and D. H. Rombach, "Goal question metric paradigm," in *Encyclopedia of Software Engineering*. New York: John Wiley and Sons, 1994, vol. 2, pp. 528–532.

[31] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal*, vol. 16, no. 1, pp. 54–73, 1977.

[32] K. D. Maxwell and P. Forselius, "Benchmarking software-development productivity," *IEEE Softw.*, vol. 17, no. 1, pp. 80–88, 2000.

[33] J. Asundi, "The need for effort estimation models for open source software projects," in *5-WOSSE: Proceedings of the fifth workshop on Open source software engineering*. New York, NY, USA: ACM, 2005, pp. 1–3.

[34] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, D. J. R. Ray Madachy, and B. Steece, *Software cost estimation with COCOMO II*. NJ: Prentice-Hall, 2000.

[35] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Softw. Eng.*, vol. 4, no. 4, pp. 345–361, 1978.

[36] B. Kitchenham, "Procedures for performing systematic reviews," Software Engineering Group, Keele University, United Kingdom and Empirical Software Engineering, National ICT Australia Ltd, Australia, Tech. Rep., 2004.

[37] G. Gousios and D. Spinellis, "A platform for software engineering research," in *MSR '09: Proceedings of the 6th Working Conference on Mining Software Repositories*, M. W. Godfrey and J. Whitehead, Eds. IEEE, May 2009, pp. 31–40.

[38] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories*, M. W. Godfrey and J. Whitehead, Eds., 2009, pp. 11–20.

[39] G. Gousios, "Tools and methods for large scale software engineering research," Ph.D. dissertation, Athens University of Economics and Business, Athens, Greece, July 2009.