ISSN: 2319-8753

International Journal of Innovative Research in Science, Engineering and Technology

Vol. 1, Issue 1, November 2012

# A Study of Software Functional Programming and Measurement

**V.Thangadurai[1], Dr.K.P.Yadav[2] , Dr.K.Krishnamoorthy[3]**

Research Scholar, Department of CSE, Monad University, Hapur (UP), India

Director, Mangalmay Institute of Technology, Greater Noida (UP), India

Professor, Dept. of CSE, Sudharsan Engineering college, Pudukkottai, Tamilnadu,India

**Abstract:** This paper presents an investigation into the usefulness of software measurement techniques, also known as software metrics, for software written in functional programming languages such as Haskell. Statistical analysis is performed on a selection of metrics for Haskell programs, some taken from the world of imperative languages. An attempt is made to assess the utility of various metrics in predicting likely places that bugs may occur in practice by correlating bug fixes with metric values within the change histories of a number of case study programs.

This work also examines mechanisms for visualizing the results of the metrics and shows some proof of concept implementations for Haskell programs, and notes the usefulness of such tools in other software Engineering processes such as refactoring. This research makes the following contributions to the field of software engineering for functional programs. A collection of metrics for use with functional programs has been identified from the existing metrics used with other paradigms. The relationship between the metrics and the change history of a small collection of programs has been explored. The relationships between the individual metrics on a large collection of programs have been explored. Visualization tools have been developed for further exploring the metric values in conjunction with program source code.

**Keywords**: software Measurement technique, Statistical analysis, Metrics, Haskell Problems.

## I. INTRODUCTION

Functional programming has been an active area of research for many years, but relatively little of this research has been directed towards the software engineering aspects of functional programming. This may partly account for the slow adoption of functional programming in "real world" software development, along with a lack of robust libraries and other such issues discussed by Wadler. Functional programming languages can be a very efficient tool for implementing complex systems. However, if the system requires debugging or performance tuning it is not necessarily straightforward to

ISSN: 2319-8753

International Journal of Innovative Research in Science, Engineering and Technology

Vol. 1, Issue 1, November 2012

test for and track down bugs or performance bottlenecks. Currently, most research in the area of software engineering for functional programming is focused on debugging techniques such as tracing, or data abstraction mechanisms such as the work of Hudak and his co-workers on monad transformers and that of Swierstra and his co-workers on combinatory libraries. Work has also been done on design methodologies for functional programming, for instance the work of Russell [84], and more recently into other development activities such as Refactoring [60] of functional programs. Work on debugging techniques is a valuable addition to the field, but mostly such work is based on runtime observation of a program, for instance the use of execution tracing. This works very well for small scale programs, but on non-trivial programs it may take an inordinate amount of time to run the program, and of course, there is no guarantee that every section of code will be executed. In such situations it would be useful to be able to concentrate the testing and debugging effort into those areas of a program that are most likely to benefit from the attention. Currently, there is no easy way to make such decisions, although runtime profiling tools are often used to help direct development effort at performance bottlenecks. In the world of imperative and object-oriented languages, software measurement, also known as software metrics, has been used for many years to provide developers with additional information about their programs. Such information can give programmers important indications about where bugs are likely to be introduced, or about how easy parts of a program are to test, for instance. This can be extremely valuable in easing the testing process by focusing programmers' attention on parts of the program where their effort may provide the greatest overall benefit, which in turn can help ease the whole process of validating software. The example of the imperative and object-oriented communities suggests that software metrics could provide a useful complement to the existing debugging tools available to functional programmers today. Some of the measurement techniques from imperative and object-oriented languages may transfer quite cleanly to functional languages, for instance the path count metric which counts the number of execution paths through a piece of program code, but some of the more advanced features of functional programming languages may contribute to the complexity of a program in ways that are not considered by traditional imperative or object oriented metrics. It may therefore be necessary to develop new metrics for certain
aspects of functional programs.

## II. SOFTWARE MEASUREMENT

Developing software is a complex and expensive process. New processes are continually being developed to improve the quality of software and reduce the costs involved in its construction, but although software development is maturing, many activities are still ad hoc and rely upon personal experience. This is particularly highlighted by Shull and his co-workers in. A significant amount of effort is spent on avoidable rework, although the amount of effort decreases as the development process matures. Avoidable rework is any rework that is not caused by changing requirements, e.g. fixing software defects such as coding errors. Finding and fixing software defects after delivery is much more expensive than fixing defects during early stages of development. The reasons for this are illustrated nicely by Smith. If errors are not detected early, much extra work must be performed in testing, diagnosing, and fixing the defect. Because of this it is important to find and fix defects as early as possible, and as such, any tools that can aid in the detection of defects can be a significant benefit. Studies such as the work of Shull and his co-workers have shown that most avoidable rework comes from a small number of software defects, and that code inspections, or peer reviews, can catch more than half of the defects in a product.

• Why not inspect all code? Conducting a peer review of millions of lines of code is time consuming and expensive.
• Why not just do lots of testing? Testing is time consuming and therefore expensive. According to Peters and Pedrycz typically as much as 30 to 50% of the software development budget is spent on testing. Additionally, it can be very hard to develop a comprehensive test suite, which can result in only partial testing of the system.
• Why not look at past defect history? The idea of using

The great promise of software metrics is that they may provide a concrete method to quantify the "quality" of software, and therefore the likelihood of defects appearing in particular sections of program code. However there has been little rigorous work to verify that software metrics can deliver on this promise. Barnes and Hopkins attempted to provide some much needed rigorous analysis of software metrics by analysing the evolution of a large library of numerical routines written in Fortran through a series of releases. They measured path count metric values for each routine in the library, in each release, and counted the number of defects over the lifetime of the library. Their statistical analysis of this data showed some encouraging signs, as it produced a statistically significant correlation between the path count values and the number of defects found over the system evolution. While Barnes and Hopkins' work concentrated on a Fortran library, their approach is equally well applicable to programs written in any programming language and therefore seems a good model in which to validate the metrics for functional program that are presented in this thesis. For this to be a useful exercise it was necessary to find a Haskell program that had a comprehensive change history, such as that available for any project stored in some form of version control software such as CVS , and which was also of a large enough size to allow some achievable (statistically significant) confidence in any statistical analysis. There are two approaches to finding a suitable program to use as a case study. The first method is to find a real-world program from a source such as the haskell.org CVS repository. Such a program would have the advantage of being a true reflection of the use of Haskell. However it is necessary that changes to the program are committed to the repository regularly, and that bug fixing changes in particular are committed individually, in order to be able to reliably separate out such changes for analysis. The alternative approach to finding a suitable case study program is to write a program of our own specifically to use as a case study. This has the advantage of allowing complete control of how changes in the program are logged, but may run the risk of being too small to have any statistically significant confidence in the statistical analysis. For the work presented in this thesis we adopted both approaches, which are now described in more detail.

ISSN: 2319-8753

International Journal of Innovative Research in Science, Engineering and Technology

*Vol. 1, Issue 1, November 2012*

## III. CONCLUSION

The investigation of software metrics for functional programming languages, in particular Haskell, has been little studied despite the interest in software metrics in other programming disciplines. Therefore this thesis attempts to address this gap with the following contributions. • A collection of metrics for use with functional programs has been identified from the existing metrics used with other paradigms. The relationship between the metrics and the change history of a small collection of programs has been explored.

• The relationships between the individual metrics on a large collection of programs have been explored.

• Visualization tools have been developed for further exploring the metric values in conjunction with program source code.

• Several of the metrics presented are strongly correlated. This suggests they are measuring closely related attributes, such as the number of patterns in a function and the number of scopes in a function. Analyzing the correlation between metrics led to the following observations.

• In the selection of metrics studied in this thesis, there does not appear to be a single metric that, for all programs, gives a good correlation with the number of bug-fixing or refactoring changes, although “Out degree” (c3), a measure of the number of functions called by a given function, can give reasonable predictions for most programs. Instead, combinations of metrics can be used to give increased correlation, and therefore more accurate predictions.

The visualization systems described in this thesis are currently proof of concept implementations, but they have shown that combining metrics with visualization is more useful than simply presenting the user with a list of metric values. Implementing the visualizations as a Haskell library has provided flexibility and extensibility for end users to customize visualizations to their own needs, as well as providing easy mechanisms to extend the library with new visualizations in the future.

## REFERENCES

[1] Eighth IEEE Symposium on Software Metrics, Ottawa, Canada, June 2002. IEEE Computer Society Press.
[2] Eighth IEEE Symposium on Software Metrics, Industrial Practices Proceedings, Ottawa, Canada, June 2002. IEEE Computer Society Press.
[3] Christopher Ahlberg and Ben Shneiderman. Visual information seeking using the FilmFinder. In Conference Companion on Human Factors in Computing Systems, pages 433–434, Boston, Massachusetts, USA, 1994. ACM Press.
[4] Edward B. Allen. Measuring graph abstractions of software: An information-theory approach. In Eighth IEEE Symposium on Software Metrics, pages 182–193, Ottawa, Canada, June 2002. IEEE Computer Society Press.
[5] An introduction to the Mac OS X Dock. Apple Computer, Inc. Cupertino, CA, USA. http://www.apple.com/support/mac101/tour/4/, April 2005.
[6] Erik Arisholm. Dynamic coupling measures for object-oriented software. In Eighth IEEE Symposium on Software Metrics, pages 33–42, Ottawa, Canada, June 2002. IEEE Computer Society Press.
[7] Richard Bache and Monika Mullerburg. Measures of testability as a basis for quality assurance. Software Engineering Journal, 5(2):86–92, 1990.
[8] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. Journal of Visual Languages and Computing, 6(2):119–133, 1995.
[9] Thomas Ball and Stephen G. Eick. Software visualization in the large. IEEE Computer, 29(4):33–43, 1996.
[10] D.J. Barnes and T.R. Hopkins. The evolution and testing of a medium sized numerical package. In H.P. Langtangen, A.M. Bruaset, and E. Quak, editors, Advances in Software Tools for Scientific Computing, volume 10 of Lecture Notes in Computational Science and Engineering, pages 225–238. Springer-Verlag, Berlin, Germany, January 2000.
[11] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. IEEE Transactions on Software Engineering, 13(12):1278–1296, 1987.
[12] Benoit Baudry, Yves Le Traon, and Gerson Suny´e. Testability analysis of a UML class diagram. In Eighth IEEE Symposium on Software Metrics, pages 54–63, Ottawa, Canada, June 2002. IEEE Computer Society Press.
[13] Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston, MA, USA, 2000. http://www.extremeprogramming.org/.
[14] D. E. Bell and J. E. Sullivan. Further investigations into the complexity of software. Technical Report MTR-2874, MITRE, 1974.

[15] Robert V. Binder. Design for testability in object-oriented systems. Communications of the ACM, 37(9):87–101, 1994.
[16] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. OpenGL Programming Guide. Addison-Wesley, Boston, MA, USA, 2004.
[17] Frederick P. Brooks. The Mythical Man-Month and Other Essays on Software Engineering. Addison-Wesley, Boston, MA, USA, 1995.
[18] J. Cain and R. McCrindle. Program visualisation using C++ lenses. In Proceedings of the Seventh International Workshop on Program Comprehension, pages 20–32, Pittsburgh, Pennsylvania, USA, May 1999. IEEE Computer
Society.
[19] David Carlson. Eclipse Distilled. Addison-Wesley, Boston, MA, USA, 2005. http://www.eclipse.org/.
[20] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. In Ricardo Pena and Thomas Arts, editors, Implementation of Functional Languages: 14th International Workshop, IFL 2002, LNCS 2670, pages 165–181, Madrid, Spain, September 2002.
[21] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), pages 268–279, Montreal, Canada, September 2000. ACM Press.
[22] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. IEEE Computer, 27(8):44–49, 1994.
[23] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. Software Engineering Metrics and Models. Benjamin-Cummings Publishing Co., Inc., Boston, MA, USA, 1986.
[24] M. Dao, M. Huchard, T. Libourel, C. Roume, and H. Leblanc. A new approach to factorization - introducing metrics. In Eighth IEEE Symposium on Software Metrics, pages 227–236, Ottawa, Canada, June 2002. IEEE Computer Society Press.