Software Metrics: Measuring Haskell

Chris Ryder* and Simon Thompson**

Computing Laboratory, University Of Kent, UK
`http://www.cs.kent.ac.uk/~cr20/`

Abstract

Software metrics have been used in software engineering as a mechanism for assessing code quality and for targeting software development activities such as testing or refactoring at areas of a program that will most benefit from them.
Haskell has many tools for software engineering, such as testing, debugging and refactoring tools, but software metrics have been neglected.
The work presented in this paper identifies a collection of software metrics for use with Haskell programs. These metrics are subjected to statistical analysis to assess the correlation between their values and the number of bug fixing changes occurring during the development lifetime of two case study programs. In addition to this, the relationships between the metric values is also explored, showing how combinations of metrics can be used to improve the accuracy of the measurements.

1 INTRODUCTION

Currently, most software engineering research for functional programs is focused on tracing and observation techniques, although recent work [6] has also looked at refactoring for functional programs. Such work is a valuable addition to the field, but can be difficult to apply effectively to large programs because of the difficulty of choosing appropriate application points.

In order to make effective use of such techniques it is typically necessary to concentrate their application into areas of a program most likely to contain bugs. However, the task of selecting such areas is often left to human intuition. Imperative and object oriented languages have used *software measurement* (also known as *software metrics*) to aid this task [4] [5], and so this work examines their applicability to functional programs written in Haskell.

1.1 Prior Work

Software metrics have been an active area of research since the early 70's. As such, there is a large body of prior work covering software metrics for imperative and OO programming, a good overview of which is provided by Melton [7].

Some of the early metrics work attracted criticism for its lack of validation, but in recent years this criticism has been addressed, see for instance [2]. Barnes

*`C.Ryder@kent.ac.uk`
**`S.J.Thompson@kent.ac.uk`

and Hopkins [1] attempted to address the issue of validation of Fortran metrics by examining the correlation between metric values and the number of bug fixes over the development lifetime of a large numerical library. Their work provides a model for the work presented in this paper.

Surprisingly, there is very little work exploring software metrics for functional languages. One of the few pieces is a PhD thesis [10] which examines the use of software metrics to compare the quality of software written in Miranda[1] with that written in Pascal. However, that work found little consensus among programmers on how to assess the quality of Miranda programs and is not discussed further here.

1.2 Overview of this paper

The remainder of this paper is divided into the following sections:-

- Section 2 introduces some metrics that can be used with Haskell.
- Section 3 describes the way in which we attempt to validate the metrics.
- Section 4 presents the results from the validation of the metrics.
- Section 5 discusses the problems encountered during the project.
- Section 6 presents the conclusions we draw from this work.

2 WHAT CAN BE MEASURED

There is a large body of work describing metrics for imperative languages. Some of those metrics, such as *pathcount* which counts the number of execution paths through a function, may directly translate to Haskell. Other features of Haskell, such as pattern matching, may not be considered by imperative metrics so it is necessary to devise metrics for such features. In this section we discuss a selection of metrics that cover a range of the commonly-used Haskell features.

2.1 Patterns

Because patterns are widely used in Haskell programs it is interesting to investigate how they affect the complexity of a program. To do this it is necessary to consider which attributes of patterns might be measured, and how these attributes might affect the complexity. We discuss these and comment on their applications case by case now:-

- *Pattern size*. There are many ways one might choose to measure the size of a pattern, but the simplest is to count the number of components in the abstract syntax tree of the pattern. The assumption is that as patterns increase in size they become more complex.

[1] Miranda is a trademark of Research Software Ltd.

- *Number of pattern variables*. Patterns often introduce variables into scope. One way in which this might affect complexity is by increasing the number of identifiers a programmer must know about in order to understand the code.

- *Number of overridden pattern variables* or *Number of overriding pattern variables*. Variables introduced in patterns may override existing identifiers, or be overridden by those in a `where` or `let` clause. Overriding identifiers in this manner can be confusing and often leads to unintended program behaviour if the compiler is unable to indicate the conflict. It therefore appears that counting the number of pattern variables involved in overriding may indicate potential points of error.

- *Number of constructors*. Patterns are often used when manipulating algebraic data types, by using the constructors of the data type in the pattern. Like counting the number of pattern variables, the hypothesis is that increasing the number of constructors in a pattern increases the amount of information a programmer must consider to understanding the pattern.

- *Number of wildcards*. When initially considering patterns it was debated whether or not wildcards in patterns should be measured. It was suggested that wildcards should be ignored because they explicitly state that the item they are matching is of no interest. However, wildcards often convey important information about the structure of items in the pattern, e.g. the position of arguments to a constructor. Because of this uncertainty it was decided that we should measure the number of wildcards to clarify their effect.

- *Depth of nesting*. Nesting of patterns is used frequently but can lead to complicated patterns. When measuring the depth of nesting one must consider how to measure the depth in patterns that contain more than one nested pattern. For instance, `[(a,b),(c,d)]` contains two nested patterns. One method is to take the maximum depth of all the nested patterns. Another method is to take the sum of the depths of the nested patterns. Taking the sum of the depths also measures how much nesting is taking place, and so might also be considered to be measuring the size of the pattern.

2.2 Distance

In all but the most trivial program there will be several declarations which will interact in some way. Inevitably there will be a distance between where things are used and where they are declared and one might hypothesise that the larger that distance, the greater the chance that an error will occur in the way that one uses that declaration. There are a number of ways in which a metric might measure distance and these are described now.

- *Number of new scopes*. One way to measure the distance between the use and declaration of an identifier is by how many new scopes have been introduced

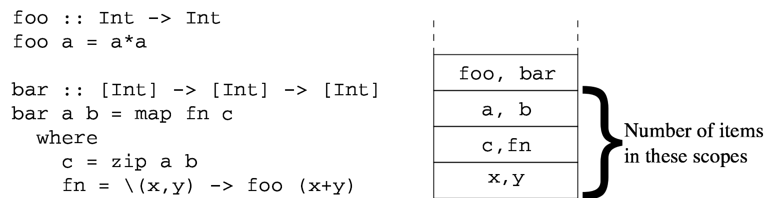

FIGURE 1. Measuring distance by the number of declarations brought into scope for the function `foo`.

between the two points. This gives a “conceptual” distance which might indicate how complex the name-space of the program is at that point. A more complex name-space may make it easier to introduce errors.

- *Number of declarations brought into scope.* An extension to the previous distance metric is to count how many declarations have been introduced into the name-space by any new scopes, not including the scope containing the declaration. This gives an idea of how “busy” the nested scopes are. This technique is illustrated in Figure 1.
- *Number of source lines.* The previously described distance metrics have measured the “conceptual” distance between use and declaration, however it is also important to consider the “spatial” distance in the source code. The assumption is that the further away items are in the source code, the harder it is to keep track of how they should be used. The simplest and most obvious way to measure distance is by the number of source code lines.
- *Number of parse tree nodes.* A problem with counting the number of source code lines as a measure of distance is that source code lines contain varying amounts of program code. One way to overcome this problem is to count the number of parse tree nodes on the path between two points of the parse tree. This may give a more consistent measure of the amount of code between the use and the declaration.

When measuring distance using scopes, the distance across module boundaries is reasonably straightforward to calculate because any imported identifiers will be in a scope of their own at the top level.

When measuring distance in the source code it is less clear how distance across module boundaries should be calculated. For this work we have chosen to measure the cross-module distance by measuring the distance between the use of an identifier and the import statement that brings it into scope, plus the distance between the declaration and the start of the module in which it is defined. This is illustrated in Figure 2. This method reflects the number of lines the programmer might have to look through, first finding the module the identifier is imported from, then finding

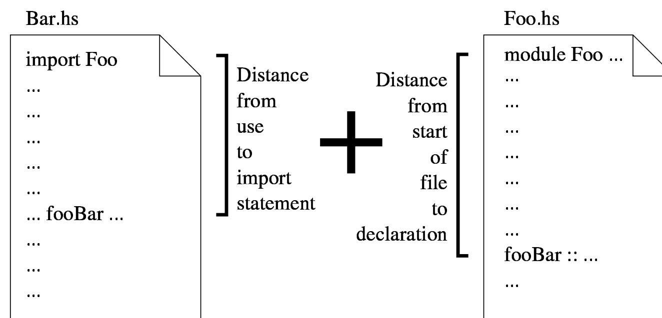

FIGURE 2. Measuring distance across module boundaries.

the identifier in the imported module. There are other variations of this method, for instance one might measure only the distance in the imported module.

Because a function is likely to call several functions there will be several distance measures, one for each called function, which must be aggregated in some way to produce a single value for the calling function. This work examines three methods: summing, taking the maximum and taking the mean.

2.3 Recursion

A common programming idiom in Haskell is recursion. Because it is common it is interesting to investigate its effect on complexity. There are two ways that a function can be recursive which one might term "trivial" and "non-trivial". In trivial recursion a function directly calls itself, whereas in non-trivial recursion the function calls itself only indirectly by means of a chain of one or more function calls. Non-trivial recursion is much less obvious in the source code. Some of the attributes that could be measured from recursive functions are described here.

- *Binary measure of recursion*. It is not always obvious if a function is recursive, because its cyclic callgraph may be large. In such cases it may be useful to know if a function is recursive without knowing how the function is recursive. Such an indication of recursion can be thought of as a binary recursion metric because it returns a value of either zero or one.

- *Number of recursive paths*. Functions may have more than one recursive path, for instance, consider Example 1. In the function `foo` there is one recursive path, i.e. it calls itself once. In both the `bar` and `fib` functions there are two recursive paths, i.e. they each call themselves twice. An increased number of recursive paths in a function may indicate greater complexity.

- *Number of trivial recursive paths* and *Number of non-trivial recursive paths*. Trivial recursive paths may be easier to understand than non-trivial recursive paths, so it is interesting to count each separately.

- *Sum or Product of the lengths of the recursive paths.* It is interesting to measure the length of the recursive paths within a function because longer paths might indicate increased complexity. If a function has more than one recursive path one must decide how to aggregate the length measures of each path. For this work we have chosen to take the product and the sum of the lengths because these methods also take account of the number of recursive paths as well as the lengths.

Example 1 (Recursive paths in functions).

```
foo :: String -> String
foo []     = []
foo (c:cs) = toLower c : foo cs

bar :: String -> String
bar [] = []
bar (c:cs) | isUpper c = toLower c : bar cs
           | otherwise = toUpper c : bar cs

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

2.4 Callgraph Attributes

Since function calls form a crucial part of Haskell it appears that some interesting properties may be measured from the callgraph of a Haskell program. Some of these are described below.

- *Strongly connected component size.* Because parts of a callgraph may be cyclic it is possible to find the strongly connected components of a callgraph. A strongly connected component is a subgraph in which all the nodes (functions) are connected (call) directly or indirectly to all the other nodes of the subgraph. Because all the functions that are part of a strongly connected component depend directly or indirectly upon each other, one might expect that as the size of the component increases, the number of changes is likely to increase as well because of the larger number of dependencies. It is worth noting that this metric may also be thought of as a measure of the degree of recursion in a program.

- *Indegree.* The indegree of a function in the callgraph is the number of functions which call it. This means that functions with high indegree values may be more important, because changes to them may affect large parts of the program.

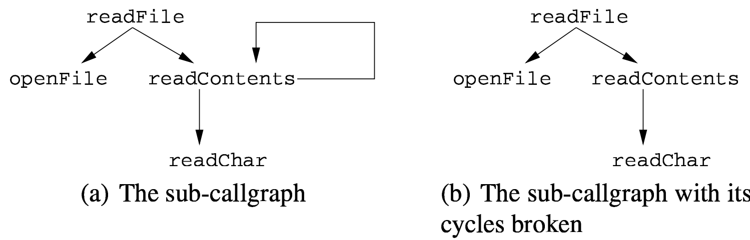

FIGURE 3. An example of a sub-callgraph for a function `readFile`.

- *Outdegree*. The outdegree of a function in the callgraph is the number of functions it calls. One might assume that the larger the outdegree, the greater the chance of the function needing to change, since changes in any of the called functions may cause changes in the behaviour of the calling function.

- The sub-callgraph for a single function. It is possible to separate out the subgraph that represents the callgraph 'rooted' at a single function in the program. One might think that the greater the complexity of this subgraph, the more likely the function is to be changed. Because of this there are several attributes one might measure from these subgraphs.

  - *Arc-to-node ratio*. The arc-to-node ratio is a useful indicator of how "busy" a graph is. If a callgraph has a high arc-to-node ratio, there is greater complexity in the interaction of the functions, and therefore one might hypothesis, a greater chance of errors occurring.
  - *Depth* and *width*. The subgraph of a function may be cyclic, but can be transformed into a tree. For instance the subgraph for a function called `readFile` is illustrated in Figure 3(a). In order to turn it into a tree it is necessary to break any cycles in the callgraph, as is shown in Figure 3(b). Such a tree represents all the direct or indirect dependencies of the function, and as such it is interesting to measure the size of this tree. Two common measures of size are the depth and the width of the tree. The deeper or wider the tree grows, the more complex the callgraph is likely to be.

2.5 Function Attributes

As well as the specific attributes highlighted in previous sections, one may also measure some more general attributes. These measurements are based upon measurements for imperative programs.

- *Pathcount*. Pathcount is a measure of the number of logical paths through a function. Barnes and Hopkins [1] showed pathcount to be a good predictor

of faults in Fortran programs, so it is interesting to investigate the results of pathcount measures for Haskell programs. Implementing pathcount for Haskell programs is mostly straightforward, although there are some places where the pathcount value is not obvious. For instance, consider Example 2.

In this example there are three obvious execution paths, one for each pattern expression, but there is also a fourth, less obvious execution path. If the second pattern `(x:xs)` matches, the guard `x > 0` will be tested. If this guard fails execution will drop through to the third pattern expression, creating a fourth execution path. Although this is a contrived example, this kind of "hidden path" can occur quite easily, for instance by omitting an `otherwise` guard.

Example 2 (Hidden execution paths when using patterns and guards).

```
func :: [Int] -> Int
func []       = 0
func (x:xs)   | x > 0 = func xs
func (x:y:xs) = func xs
```

- *Operators* and *Operands*. Having talked of various measurements previously it is important not to ignore less sophisticated measures such as function size. The larger a function, the more complex it is likely to be.

  There are many ways to measure program size, Van Den Berg [10] used a variation of Halstead's [5] operator and operand metrics in his work with Miranda. For this work we have updated Van Den Berg's metrics for Haskell. We define all literals and identifiers that are not operators as operands. Operators are the standard operators and language keywords, such as `:`, `++`, `where`, etc. Delimiters such as `()` and `[]`, etc, are also included as operators. The number of operands and the number of operators were implemented as separate metrics, but they are really part of a connected pair.

In this section we have presented a selection of metrics which cover a wide range of attributes of the Haskell language. With the exception of the *Number of wildcards* metric, these metrics would be expected to increase in value as the complexity increases.

The measures introduced here are all measuring distinct attributes, and it may be that some of these can be combined to produce more sophisticated and accurate measures. However it is important to validate these "atomic" metrics before attempting to combine them. The methodology used to validate these measures is described in the next section and the results are discussed in Section 4.

3 METHODOLOGY

To analyse the measurements described in Section 2 a number of case studies were undertaken. The general idea of the case studies was to follow the methodology of

Module	Min Size (LOC)	Max Size (LOC)	Num. Changes
Board	86	220	9
Main	25	27	38
Solve	39	101	7
Stack	26	31	0
GPegSolitaire	228	350	78
TPegSolitaire	98	177	16
		Total Number of Changes	148

TABLE 1. Summary of the Peg Solitaire case study program.

Barnes and Hopkins [1] and take a series of measurements from a program over its development lifetime, and then to correlate those measures with the number of bug fixing or refactoring changes occurring during that time. Metrics that correlate well with the bug fix counts may be good indicators of targets for testing or refactoring.

We experienced some difficulty in finding suitable case study programs, as is described later in Section 5. The two programs eventually chosen for the case study are both products of another research project at the University of Kent. The programs were both maintained in a CVS repository, giving easy access to the change history of the programs.

3.1 Peg Solitaire Case Study

The first case study program was a Peg Solitaire game [9] with both textual and graphical interfaces, consisting of a number of modules which did not necessarily all exist at the same point in time. A summary of the sizes of the modules is shown in Table 1.

3.2 Refactoring Case Study

The second case study program was a tool for refactoring Haskell programs [6]. The program used a library for parsing Haskell code which was not examined in this study, so only the code that manipulated parse trees was analysed. This program was approximately twice the size of the Peg Solitaire program. Details of the sizes and number of changes for each module are shown in Table 2.

3.3 Analysing change histories and metrics

The change histories of the two programs were manually examined to determine the nature of the changes (feature addition, bug fix, etc) and the number of bug fixing changes occurring for every function during the development lifetime was recorded.

Module	Min Size (LOC)	Max Size (LOC)	Num. Changes
EditorCommands	198	213	4
PFE0	332	337	2
PfeRefactoringCmds	18	24	5
PrettySymbols	23	23	0
RefacAddRmParam	142	434	56
RefacDupDef	62	157	19
RefacLocUtils	201	848	88
RefacMoveDef	322	796	56
RefacNewDef	77	478	58
RefacRenaming	67	236	23
RefacTypeSyn	20	21	0
RefacUtils	764	1088	126
ScopeModule	222	222	0
TiModule	140	140	0
Main	36	103	7
		Total Number of Changes	444

TABLE 2. Summary of the Refactoring case study program.

The metrics described in Section 2 were then run on each version of each program, and the maximum value of each metric was taken for every function. These measurements were then correlated with the number of bug fixing changes for each function using the statistical macros of an Excel spreadsheet.

Metrics whose values increase with increased complexity would be expected to show a positive correlation with the number of changes, while metrics whose values decrease with increased complexity are likely to have negative correlations.

4 RESULTS

This section presents some of the results of correlating the measurements taken from the case study programs with their change histories. We also investigate the correlation between the metric values to see if any metrics are related. The full and detailed results for this work are not presented here due to space constraints, but are analysed in detail in the thesis by Ryder [8]. Instead, the following main observations are discussed:

- *Outdegree* is correlated with the number of changes.
- All the distance metrics show similar levels of correlation.
- Callgraphs tend to grow uniformly in both width and depth.

Metric	Correlation (r)
Peg Solitaire	
Outdegree	0.4783
Strongly connected component size	0.3446
Refactoring	
Sum of number of scopes	0.632
Maximum number of scopes	0.6006
Number of pattern variables	0.5927
Number of operands	0.5795
Outdegree	0.5723

TABLE 3. Summary of highest correlation values from the case study programs.

- Recursion is used very little in the case study programs.
- Most of the pattern metrics are measuring the size of a pattern.

4.1 Correlation of individual metrics

The first results we analysed were those taken by correlating metric values against the number of changes. Table 3 summarises the highest statistically significant correlation values obtained from the two case study programs.

The first observation from these results was that, for most of the metrics, there was no statistically significant correlation in the data taken from the Peg Solitaire program. Only the *Strongly connected component size* and *Outdegree* metrics showed correlation values that were statistically significant at the 5% level, with correlation values of 0.3446 and 0.4783 respectively.

Conversely, the Refactoring program showed statistically significant correlations for all the metrics except for the recursion metrics, the *Strongly connected component size* metric and the *Indegree* metric.

The *Indegree* measure is not statistically significant for either program, and so one can assume that the indegree of a function has little effect on the complexity, at least as far as these case studies are concerned.

The Refactoring program has only a small amount of recursion in its functions, all of which is of the trivial variety, while the Peg Solitaire program has some non-trivial recursive functions. This means that the *Strongly connected component size* values for the Refactoring program will be either zero or one, while the values for the Peg Solitaire program will have a wider range of values. This may explain why this metric is significant for the Peg Solitaire program but not for the Refactoring program.

The recursion metrics show no statistically significant correlation for either program. This may be because there were relatively few recursive functions in the

two programs, just 21 out of 539 functions for the Refactoring program and 35 out of 233 functions for the Peg Solitaire program.

None of the distance measures were significant at the 5% level for the Peg Solitaire program, however they were all significant for the Refactoring program. Most of the measures resulted in correlations between 0.4 and 0.55, but the greatest correlation was provided by the *Sum of the number of scopes* metric, with a correlation of 0.632. These results seem to confirm that the greater the distance between where something is used and where it is declared, the greater the change of an error occurring in how it is used. The results also seem to suggest that it does not matter too much how the distance is measured, with the "semantic" measures having slightly higher correlation values than the "spatial" measures on average.

From the callgraph measures, the *Outdegree* metric provided the greatest correlation for both programs, with a correlation value of 0.4783 for the Peg Solitaire program and 0.5723 for the Refactoring program. This provides some evidence that functions that call lots of other functions are likely to change more often that functions that do not call many functions.

Of the other callgraph measures, *Strongly connected component size* has a significant correlation for the Peg Solitaire program, but not for the Refactoring program, as has been discussed earlier. None of the other callgraph measures have significant correlation for the Peg Solitaire program, but they do have significant correlations of varying degrees for the Refactoring program, from 0.3285, for the *Width* measure, to 0.4932 for the *Depth* measure.

The results for the function attributes showed that although none of the metrics were significant at the 5% level for the Peg Solitaire program, the operands and operators measures were significant at the 10% level. For each program the operands and operators measurements showed very similar correlation values. The pathcount measure showed a small correlation of 0.286 for the Refactoring program.

4.2 Cross-correlation of metrics

Having looked at the correlation of metric values with the number of changes, it is interesting to look at the correlation between metric values, which might indicate relationships between the attributes being measured.

Initially, the cross-correlation between metrics of the same class is examined, but later we examine correlation across metrics of different classes. Table 4 shows the clusters of metrics which appear to be strongly correlated.

The cluster formed by the pattern metrics, C3 in Table 4, implies that the pattern metrics are measuring a similar attribute, most likely the size of a pattern.

The distance measures form two clusters, C1 and C5 in Table 4. Cluster C1 suggests there is little difference between measuring distance by the number of source lines or by the number of parse tree nodes, and shows that measuring the sum of the number of scopes or declarations in scopes does not give much more information than measuring the number of source lines. This might be because

Cluster	Metrics
C1	Sum of the number of scopes
	Sum of the number of declarations
	Sum of the number of source lines
	Maximum number of source lines
	Average number of source lines
	Sum of the number of parse tree nodes
	Maximum number of parse tree nodes
	Average number of parse tree nodes
C2	Callgraph depth
	Callgraph width
C3	Number of pattern variables
	Sum of depth of patterns
	Pattern size
	Number of constructors in pattern*
C4	All recursion metrics
C5	Average number of scopes
	Maximum number of declarations
	Maximum number of scopes
	Average number of declarations
C6	Number of operands
	Number of operators

*Only correlated in the Refactoring case study program.

TABLE 4. Strongly correlated metrics for the case study programs.

declarations that are further away in scope tend to be further away in the source code. Likewise, as the number of declarations increase, so the distances between declarations and where they are used tends to increase.

Cluster C5 shows that the distance measured by the maximum or average number of scopes or declarations in scope is not strongly correlated with measuring the sum of the number of scopes or declarations in scope. One explanation for this would be that the identifiers used in a function are generally a similar distance from their declarations, for instance, all the uses of a pattern variable in a function might have a similar distance measure. This would cause the average and maximum values to be similar between functions, while the sum measure would vary much more.

Examining the cross-correlation of the callgraph metrics, cluster C2 in Table 4, shows that apart from the *Depth* and *Width* metrics, there is very little correlation between this class of metrics. This seems to confirm that they are measuring distinct attributes of callgraphs. The correlation between the *Width* and *Depth* metrics is interesting because it seems to suggest that callgraphs for individual functions tend to grow uniformly in both depth and width, and that they very rarely end up long and thin or short and wide.

The cluster C6 is unsurprising since these metrics are really part of a pair of interconnected metrics. However, the pathcount does not appear to be part of the cluster, showing that it is unlikely to be measuring the size of a function.

4.3 Cross-correlation of all the metrics

If the clusters of strongly correlated metrics are replaced with a representative of each cluster, it is possible to analyse the correlation between the various classes of metrics. For this work, each cluster was represented by the metric with the highest

C1	Number of pattern variables Number of operands Sum of number of scopes
C2	Maximum number of scopes Outdegree

TABLE 5. Summary of cross-correlation between metrics of different classes for the Refactoring program.

correlation value in the cluster.

The measurements from the Peg Solitaire case study showed no correlation between the various metrics, while the cross-correlation for the Refactoring case study is shown in Table 5.

The correlation between *Number of pattern variables* and *Number of operands* seen in cluster C1 of Table 5 is probably because variables are counted as operands, so an increase in the number of pattern variables will necessarily entail an increase in the number of operands.

The correlation with the *Sum of number of scopes* measure is less clear. It suggests that as the number of pattern variables increases, so the distance to any called functions, measured by the sum of the number of scopes, increases. This is most likely because pattern variables are often introduced where new scopes are constructed.

Cluster C2 of Table 5 suggests that the largest distance to any function called from any single function will increase as the number of called functions increases. This may be because as more functions are called, they will tend to be further away, since the called functions can not all be located in the same place.

4.4 Regression analysis of metrics

In order to obtain a greater correlation with the number of changes it may be possible to combine a number of metrics together. Determining the best combination of metrics can be done using a regression analysis. The regression analysis of the results from both case studies showed that statistically significant correlation can be achieved for both programs, with correlation values of 0.583 for the Peg Solitaire program and 0.6973 for the Refactoring program, which are higher than any of the individual metrics correlation values.

The coefficients for the metrics from the Peg Solitaire program show that the largest contribution, with a coefficient of 0.4731, comes from the *Outdegree* metric, suggesting that the most important attribute is the number of direct dependencies.

The coefficient for the *Sum of number of source lines* distance metric, -0.2673, is negative which suggests that if the functions used are a long way away in the source code it is less likely to introduce errors. This result may be caused by cross-

module function calls, which imply that the calling function is using some well defined and stable interface, and hence is less likely to have to be changed as a result of the called function being changed. This may imply that when measuring attributes across modules, the behaviour of the attributes might be different than when considering them inside a single module.

The coefficients from the Refactoring program regression analysis shows that the largest contribution by some margin comes from the *Sum of number of scopes* metric with a coefficient of 0.315. This suggests that, for the Refactoring program, it is important to know how complicated the name-space is for each function.

5 DIFFICULTIES ENCOUNTERED IN THIS WORK

The biggest difficult encountered during the course of this work was finding suitable programs to use as a case studies. Candidate programs needed to have source code stored in a CVS repository with a change history that had enough changes to allow for meaningful analysis.

Most of the programs investigated had no clear separation between bugfixing changes, refactoring changes and feature addition changes, with different types of changes being committed to CVS in the same commit. Because of this it was necessary to manually inspect each change in the CVS history of the programs to determine the type of change, a very time-consuming and error prone procedure.

Because of the need to manually inspect changes it was necessary to choose programs that were small enough to be able to inspect manually, but this can cause problems if there are too few changes for statistically significant results to be obtained. The Peg Solitaire program suffers from this to some extent.

6 CONCLUSIONS AND FURTHER WORK

In this paper we have described a number of software measurement techniques that can be used on Haskell programs. Using two case study programs we have shown that it may be possible to use some of these measurements to indicate functions that may have an increased risk of containing errors, and which therefore may benefit from more rigorous testing.

By analysing the cross-correlation of the metrics we have shown that some of the metrics measure very similar, or closely related attributes. The regression analysis of the metrics has shown that combining the measurements does increase the correlation with the number of changes, and therefore the accuracy of the metrics. From this we can see that there is no single attribute that makes a Haskell program complex, but rather a combination of features. However, good estimates can be obtained using only the *Outdegree* metric, which counts how many functions are called by a given function, suggesting that most of the complexity lies not with individual functions, but rather in the interaction of the functions.

6.1 Further Work

It is important to be realistic with the findings in this paper. They are based upon two Haskell programs, which may not be representative of Haskell programs in general. To further clarify these results it would be necessary to repeat these studies on a larger range of programs, although the time and effort involved in manually inspecting the change histories of the programs may be prohibitive.

What can be achieved much more easily is to further analyse the relationships between the metrics. This further analysis has been performed as part of the thesis by Ryder [8], but is not included here due to space constraints.

To extend this work we would like to integrate the ideas of software metrics into the HaRe [6] refactoring tool. The aim of such a project would be to use metrics to target the refactorings, in line with Fowlers' [3] work on "bad smells".

REFERENCES

[1] 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.

[2] Lionel Briand, Khaled El Emam, and Sandro Morasca. Theoretical and empirical validation of software product measures. Technical Report ISERN-95-03, Fraunhofer Inst. for Experimental Software Engineering, Germany, 1995.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. http://www.refactoring.com/.

[4] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey P. Siy. Predicting fault incidence using software change history. *Software Engineering*, 26(7):653–661, 2000.

[5] Maurice H. Halstead. *Elements of Software Science*. Elsevier, New York, USA, 1977.

[6] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Johan Jeuring, editor, *Proceedings of the ACM SIGPLAN 2003 Haskell Workshop*, pages 27–38, Uppsala, Sweden, August 2003. ACM Press.

[7] Austin Melton, editor. *Software Measurement*. International Thompson Publishing, London, UK, 1996.

[8] Chris Ryder. *Software Measurement for Functional Programming*. PhD thesis, Computing Lab, University of Kent, Canterbury, UK, August 2004. Available on-line at http://www.cs.kent.ac.uk/pubs/2004/2236.

[9] Simon Thompson and Claus Reinke. A case study in refactoring functional programs. In Roberto Ierusalimschy, Lucilia Figueiredo, and Marcio Tulio Valente, editors, *Proceedings of 7th Brazilian Symposium on Programming Languages*, pages 1–16, Ouro Preto, MG, Brazil, May 2003. Journal of Universal Computer Science, Springer-Verlag.

[10] Klaas Van den Berg. *Software Measurement and Functional Programming*. PhD thesis, University of Twente, Department of Computer Science, P.O.Box 217, 7500 AE Enschede, the Netherlands, June 1995.