Chapter 1

Software Metrics: Measuring Haskell

Chris Ryder[1], Simon Thompson[1]

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 [PJ03] has many tools for software engineering, such as testing, debugging and refactoring tools, but software metrics have been neglected.

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 of two case study programs. In addition, the relationships between the metrics are also explored, showing how combinations of metrics can be used to improve their accuracy.

1.1 INTRODUCTION

Currently, most software engineering research for functional programs is focused on tracing and observation techniques, although recent work by Li and others [LRT03] has also looked at refactoring for functional programs. Such work is a valuable addition to the field, but can be hard to effectively apply 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 [GKMS00, Hal77, FP98], and so this work examines the applicability of metrics to functional programs written in Haskell.

[1]Computing Laboratory, University Of Kent, Canterbury, Kent, CT2 7NF, UK;
Email: C.Ryder@kent.ac.uk, S.J.Thompson@kent.ac.uk

1.1.1 Prior Work

Software metrics have been an active area of research since the early 70's so there is a large body of prior work for OO and imperative languages, such as that by Fenton, Pfleeger and Melton [FP98, Mel96]. Some of the early work attracted criticism for its lack of validation, but in recent years this has been addressed, for instance by Briand and his co-workers [BEEM95]. Barnes and Hopkins [BH00] addressed the issue of validation by examining the correlation between metric values and the number of bug fixes over a programs development lifetime.

Surprisingly, there is little work exploring metrics for functional languages. One of the few pieces is a thesis by Van Den Berg [VdB95] which examines the use of metrics to compare the quality of software written in Miranda[2] with that written in Pascal. However, little consensus was found among programmers on how to rate the quality of Miranda programs, so it is not discussed further here.

1.1.2 Motivation

The motivation for investigating software metrics for functional programming languages comes from three common software engineering tasks, software testing, code reviews and refactoring.

Currently, these tasks rely on either human intuition, e.g. to decide which refactoring to apply to a function, or brute force, e.g. by reviewing every function. Each of these tasks can be helped by using software metrics to concentrate programmer's effort on areas of the program where most benefit is likely to be gained. For instance, functions which exhibit high metric values might be tested more rigorously, may be subject to an in-depth code review, or may be refactored to reduce their complexity. Conversely, functions which exhibit low metric values may not require as much testing, reviewing or refactoring. Targeting programmer effort using metrics in this manner can improve the quality of software by making more efficient use of programmer's time and skills.

In many ways metrics are analogous to compiler warnings. They indicate unusual features in the code, but there may be legitimate reasons for those features. Like warnings, metrics give a hint that part of the code may need to be inspected.

1.1.3 Overview of this paper

The remainder of this paper is divided into the following sections: Section 1.2 introduces a selection of metrics that can be used with Haskell. Section 1.3 describes the way in which we attempt to validate the metrics. Section 1.4 presents the results from the validation of the metrics. Section 1.5 presents the conclusions we draw from this work.

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

1.2 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.

At the time this project was started we were unable to implement type-based metrics, such as measuring the complexity of a function's type, because we were unable to find a stand-alone implementation of the Haskell type system. Therefore the metrics presented here are a first step in assessing metrics for Haskell. Recently, the Glasgow Haskell Compiler (GHC) [MPJ04] has begun to provide a programming interface to its internal components, such as its type checker. This allows type-based metrics to be implemented, which we hope to pursue in future.

In the remainder of this section we present a selection of the Haskell metrics we analysed and discuss their relationship to imperative or OO metrics.

1.2.1 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 case by case now:-

- *Pattern size* (*PSIZ*). 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.

- *Number of pattern variables* (*NPVS*). 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 comprehend the code. Studies [Boe81, McC92, FH79] have shown that at least 50% of the effort of modifying a program is in comprehending the code being changed.

- *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` clause for instance. Overriding identifiers can be confusing and can lead to unintended program behaviour, particularly if the compiler is unable to indicate the conflict because the identifiers have the same type. Therefore one hypothesis is that high numbers of variables involved in overriding may indicate potential points of error.

- *Number of constructors* (*PATC*). Patterns are often used when manipulating algebraic data types by using the constructors of the data type in the pattern. Like NPVS, the hypothesis is that the higher the number of constructors in a pattern the more information a programmer needs to consider to understand it.

```
foo :: Int -> Int
foo a = a*a

bar :: [Int] -> [Int] -> [Int]
bar a b = map fn c
  where
    c = zip a b
    fn = \(x,y) -> foo (x+y)
```

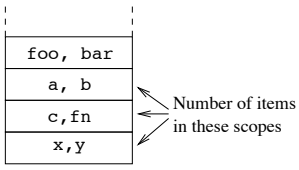

FIGURE 1.1. Measuring distance by the number of declarations brought into scope for the function `foo`.

- *Number of wildcards* (*WILD*). When initially considering patterns it was suggested that wildcards should be ignored because they state that the item they are matching is of no interest. However, wildcards convey information about the structure of items in the pattern, e.g. the position of constructor arguments. Therefore it was decided that we should measure WILD to clarify their effect.

- *Depth of nesting*. Patterns are frequently nested, which can lead to complicated patterns. When measuring the depth of nesting one must consider how to measure the depth in patterns such as `[(a,b),(c,d)]`, which contain more than one nested pattern. This study uses two ways, *Sum of the depth of nesting* (*SPDP*) and *Maximum depth of nesting*, however the sum method may also be measuring the size of the pattern.

1.2.2 Distance

In all but the most trivial program there will be several declarations which will interact. The interactions between declarations are often described by *def-use* pairs [RW85]. For instance, the def-use pair (a,b) indicates that b uses the declaration a. Metrics that use def-use pairs are most often concerned with the testability of programs, because def-use pairs indicate interactions that might require testing.

When one considers a def-use pair, there will inevitably be a distance between the location of the use and the declaration in the source code. One hypothesis is that the larger the distance, the greater the probability that an error will occur in the way that declaration is used. Distance may be measured in a number of ways:

- *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 between the two points. This gives a "conceptual" distance which may indicate how complex the name-space is at that use. This leads to a hypothesis that a more complex name-space may make it harder to avoid introducing 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. This technique, illustrated in Figure 1.1, may give an idea of how "busy" the nested scopes are.

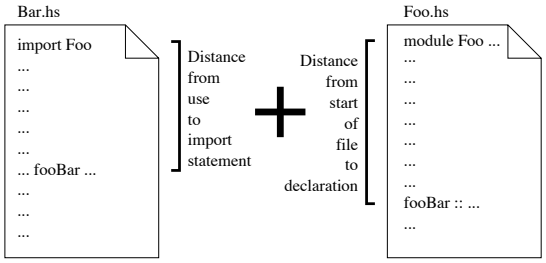

FIGURE 1.2. Measuring distance across module boundaries.

- *Number of source lines.* The distance metrics described previously have measured the "conceptual" distance, however it is also important to consider the "spatial" distance in the source code. The hypothesis is that the further away items are in the source code, the harder it is to recall how they should be used. The simplest way to measure spatial distance is by the number of source lines.
- *Number of parse tree nodes.* A problem with counting source lines as a measure of distance is that source 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 instead. This may give a more consistent measure of the amount of code between the use and the declaration.

Measuring distance between modules using scope-based measures is straightforward, because imported identifiers will be in a top level scope of their own.

When measuring distance using source lines it is less clear how distance between modules 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 1.2. This method reflects the number of lines a programmer may have to look through, first finding the module the identifier is imported from, then finding the identifier in the imported module. A variation of this method might be to measure only the distance in the imported module, for instance.

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.

1.2.3 Callgraph Attributes

Because 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* (*SCCS*). Because parts of a callgraph may be cyclic it is possible to find the strongly connected components. A strongly connected component (SCC) is a subgraph in which all the nodes (functions) are connected (call) directly or indirectly to all the other nodes. Because all functions that are part of a SCC depend directly or indirectly upon each other, one might expect that as the size of the SCC increases, the number of changes is likely to increase as well, because a change to a single function may cause changes to other functions in the SCC. This is often known as the *ripple effect*.

  SCCS is a measure of coupling, similar to imperative and OO coupling metrics such as the *Coupling between object classes* (CBO) metric used by Chidamber and Kemerer [CK94]. The main difference is that CBO measures only direct coupling between objects, e.g. `A` calls `B`, while SCCS also measures indirect coupling between functions, e.g. `A` calls `X` which calls `B`.

- *Indegree* (*IDEG*). The indegree of a function in the callgraph is the number of functions which call it, and thus IDEG is a measure of reuse. Functions with high IDEG values may be more important, because they are heavily reused in the program and therefore changes to them may affect much of the program. This metric is inspired by the *Fan-In* metric of Constantine and Yourdon [YC79], which measures how many times a module is used by other modules. Thus IDEG is Fan-In used on individual functions, rather than whole modules.

- *Outdegree* (*OUTD*). The outdegree of a function in the callgraph is the number of functions it calls. One might assume that the larger the OUTD, 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. Like the IDEG metric, the OUTD is inspired by the work of Constantine and Yourdon, in this case by their *Fan-Out* metric which measures the number of modules used by a module. As with IDEG, the OUTD metric is used on individual functions, rather than on whole modules.

It is possible to isolate the subgraph that represents the callgraph rooted at a single function. One hypothesis is that the greater the complexity of the subgraph, the more likely the function is to change, because it is harder to comprehend the subgraph. Therefore one might measure several attributes from these subgraphs:

- *Arc-to-node ratio* (*ATNR*). The arc-to-node ratio is a useful indicator of how "busy" a graph is. If a callgraph has a high ATNR, there is greater complexity in the interaction of the functions, and therefore one might hypothesise, a greater chance of errors occurring. This is similar to the FIFO metric suggested by Constantine and Yourdon, but FIFO looks only at the direct dependents and dependencies of the module being measured, while ATNR looks at the complexity of all the interdependencies of the entire subgraph.

- *Callgraph Depth* (*CGDP*) and *Callgraph Width* (*CGWD*). The subgraph of a function may be cyclic but can be transformed into a tree by breaking its cycles, as is illustrated in Figure 1.3. Such a tree represents all the direct or

(a) The sub-callgraph (b) The sub-callgraph with its cycles broken

FIGURE 1.3. An example of a sub-callgraph for a function `readFile`.

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. The deeper or wider the tree grows, the more complex it is likely to be.

The depth and width metrics described here are inspired by the *Depth of Inheritance Tree* (DIT) and *Number of Children* (NOC) metrics suggested by Chidamber and Kemerer for OO programs.

The DIT metric measures how deep a class is in the inheritance hierarchy of an OO program. The deeper a class is, the greater the number of methods it is likely to inherit, and hence the harder it is to predict its behaviour. In Haskell we model this by measuring the depth of the subgraph, because a deep subgraph is likely to be hard to comprehend than a shallow subgraph.

The NOC metric is the number of immediate children in the inheritance hierarchy of the class being analysed. The number of children indicates how much the class is reused, and thus how important the class is to the design of the program. Our width measure looks superficially similar to the NOC metric, but in fact measures dependencies, much like our depth metric. The Haskell metric most closely resembling NOC is the IDEG metric, which also measures reuse.

1.2.4 Function Attributes

As well as the specific attributes highlighted in previous sections, one may also measure some more general attributes such as the following.

- *Pathcount* (*PATH*). Pathcount is a measure of the number of logical paths through a function. Barnes and Hopkins showed pathcount to be a good predictor of faults in Fortran programs, so it is interesting to investigate pathcount metrics for Haskell programs. Implementing pathcount for Haskell is mostly straightforward, although there are some places where the pathcount value is not obvious. For instance, consider Example 1.1.

  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 1.1 (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* (*OPRT*) and *Operands* (*OPRD*). Having discussed various metrics previously it is important not to ignore less sophisticated measures such as function size. A large function is more likely to be complex than a small one.

  There are many ways to measure program size. Van Den Berg used a variation of Halstead's [Hal77] operator and operand metrics in his work with Miranda. This work updates Van Den Berg's metrics for Haskell by defining 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. Although OPRT and OPRD were implemented as separate metrics, they are really 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 WILD metric, these metrics are expected to increase in value as the complexity increases.

The metrics 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.

1.3 VALIDATION METHODOLOGY

To validate the metrics described in Section 1.2 a number of case studies were undertaken. For this work we followed the methodology described by Barnes and Hopkins and took a series of measurements from a program over its development lifetime, and then correlated 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.

A limitation of this method of validation is that all bug fixing changes are considered to be of equal importance. In reality it is likely that some bug fixes might be considered "trivial" because they were easy to implement or had only minimal impact on the operation of the program, while others may be considered to be much more serious because they were hard to implement or had a significant impact on the operation of the program. It is not clear how the relative seriousness of a bug fix should be incorporated into this analysis, for example, should serious bug fixes be counted as multiple trivial bug fixes? or should trivial bug fixes be

discarded? Furthermore, it would be difficult to objectively assess the seriousness of the changes. Therefore, we do not include "bug seriousness" in our analysis.

We experienced some difficulty in finding suitable case study programs. Candidate programs needed to have source code stored in a CVS repository with a change history that contained enough changes to allow for meaningful analysis.

Most of the programs investigated had no clear separation between bug fixes, refactorings and feature additions, with different types of changes often being committed to CVS in the same commit. Unfortunately it is not possible to automatically classify these changes, e.g. by assuming small changes are bug fixes and large changes are feature additions, with any degree of accuracy because the sizes of the changes are not uniform. For instance, a feature addition may involve lots of small changes to lots of functions, and thus be indistinguishable from a collection of small bug fix changes, while conversely a bug fixing change may require a large change to a function and thus look like a feature addition.

Because it was not possible to automatically classify changes 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 process. However, this issue only affects the validation process, not the use of the metrics.

The need to manually inspect changes necessitated choosing programs that were small enough to be able to inspect manually within a reasonable amount of time, but choosing smaller programs causes problems if there are too few changes for statistically significant results to be obtained. The first of our case study programs, a Peg Solitaire program described later, suffers from this to some extent.

The use of a revision control system that uses fine grained commits, such as `darcs`, may encourage programmers to clearly and individually record bug fixes.

The two programs 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 histories. The programs were developed separately from our work and we had no influence in their development, other than to request that changes be committed to the CVS repository individually, making it significantly easier to classify the types of the changes.

1.3.1 Peg Solitaire Case Study

The first case study program was a Peg Solitaire game [TR03] with both textual and graphical interfaces, consisting of a number of modules which did not necessarily all exist simultaneously. The module sizes are shown in Table 1.1.

1.3.2 Refactoring Case Study

The second case study program was a tool for refactoring Haskell programs [LRT03]. The program used a parser library which was not examined in this study, therefore only the code that manipulated parse trees was analysed. Table 1.1 shows this program was approximately twice the size of the Peg Solitaire program.

	Module	Min Size (LOC)	Max Size (LOC)	Num. Changes
Peg Solitaire	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
Refactoring	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 1.1. Summary of the Peg Solitaire and Refactoring case study programs.

1.3.3 Analysing change histories and metrics

The change histories of the two programs were manually examined to determine the nature of the changes, such as feature additions, bug fixes, etc, and the number of bug fixing changes occurring for every function during the development lifetime was recorded. It is important to note that the programs did compile after every change, therefore the change counts do not include errors that would have been caught by the compiler, except where they are part of a larger change.

The metrics described in Section 1.2 were then run on each version of each program, and the maximum value of each metric was taken for every function. The measurements were then correlated with the number of bug fixing changes for each function using the statistical macros of Excel. Although it is possible that taking the maximum value may introduce artifacts, the metric values for a given function tend not to change very often so we do not believe this to be a problem.

1.4 RESULTS

This section presents some of the results of correlating the measurements taken from the case study programs with their change histories. 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

	Metric	Correlation (r)	r^2
PS	OUTD	0.4783	0.229
	SCCS	0.3446	0.119
RE	Distance by the sum of number of scopes	0.632	0.399
	Distance by the maximum number of scopes	0.6006	0.361
	NPVS	0.5927	0.351
	OPRD	0.5795	0.336
	OUTD	0.5723	0.328

TABLE 1.2. Highest correlations for Peg Solitaire (PS) and Refactoring (RE).

increased complexity are likely to have negative correlations.

We also investigated the correlation between different metrics to see if any metrics were 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 [Ryd04]. Instead, the following main observations are discussed:

- The OUTD metric 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.
- Most of the pattern metrics are measuring the size of a pattern.

1.4.1 Correlation of individual metrics

The first results we analysed were those taken by correlating metric values against the number of changes. Table 1.2 summarises the highest statistically significant correlation values obtained from the two case studies, as well as their r^2 values. The r^2 values show the proportion of the variance in common between the metric and the number of bug fixes. This gives an indication of the influence of the correlation on the number of bug fixes. For instance, consider the OUTD metric in Table 1.2. It has an r^2 value of 0.229, which states that there is 22.9% of the variance in common between OUTD and the number of bug fixes. In the rest of this section correlation values will be followed by their r^2 values in parenthesis, e.g. the correlation and r^2 values of OUTD will be shown as 0.4783 (0.229).

These results show that, for most of the metrics, there was no statistically significant correlation in the data taken from the Peg Solitaire program. Only the SCCS and OUTD metrics show correlation that was statistically significant at the 5% level, with values of 0.3446 (0.119) and 0.4783 (0.229) respectively.

Conversely, the Refactoring program shows statistically significant correlations for all the metrics except for the SCCS and IDEG metrics.

IDEG, which measures reuse, is not statistically significant for either program, so one can assume that the reuse of a function has little effect on its complexity.

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 (0.16) and 0.55 (0.303), but the greatest correlation was provided by the *Distance by the sum of the number of scopes* metric, with a correlation of 0.632 (0.399). These results seem to confirm that the greater the distance between where something is used and where it is declared, the greater the probability 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 stronger correlation with the number of bug fixes than the "spatial" measures on average.

However, we do not know what text editor was used in the development of these case study programs. It may be that a "smart" editor that allows the programmer to jump directly to definitions may reduce the effect of distance.

From the callgraph measures, OUTD provided the greatest correlation for both programs, with a correlation value of 0.4783 (0.229) for the Peg Solitaire program and 0.5723 (0.328) for the Refactoring program. This provides some evidence that functions that call lots of other functions are likely to change more often than functions that do not call many functions. This is also known to occur for the related *Fan-Out* OO metric described previously in Section 1.2.3.

Of the other callgraph measures, SCCS has significant correlation for the Peg Solitaire program, but not for the Refactoring program, as was discussed earlier. Although none of the other callgraph measures have significant correlation for the Peg Solitaire program, they do have significant values for the Refactoring program, ranging from 0.3285 (0.108) for CGWD, to 0.4932 (0.243) for CGDP.

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 OPRD and OPRT measures were significant at the 10% level. For each program the OPRD and OPRT measurements showed very similar correlation values. The PATH measure showed a small correlation of 0.286 (0.082) for the Refactoring program.

1.4.2 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 1.3 shows the clusters of metrics which appear to be strongly correlated.

The cluster formed by the pattern metrics, C3 in Table 1.3, 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 C4 in Table 1.3. 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 declarations that are further away in scope tend to be further away in the source code. Likewise, as the number of declarations increases, so the distances between

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

TABLE 1.3. Strongly correlated metrics for the case study programs.

declarations and where they are used tend to increase.

Cluster C4 shows that distance measured by the maximum or average number of scopes or declarations in scope is not strongly correlated with distance by the sum of the number of scopes or declarations in scope. One reason for this might be that the identifiers used in a function are generally a similar distance from their declarations, e.g. 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 1.3, shows that apart from the CGDP and CGWD 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 CGWD and CGDP metrics is interesting because it seems to suggest that callgraphs for individual functions tend to grow uniformly in both depth and width.

The cluster C5 is unsurprising since these metrics are really part of a pair of interconnected metrics. However, the PATH metric does not appear to be part of the cluster, showing that it is unlikely to be measuring the size of a function.

1.4.3 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 correlation value in the cluster. The measurements from the Peg Solitaire case study showed no correlation between the various classes of metrics, while the cross-correlation for the Refactoring case study is shown in Table 1.4.

The correlation between NPVS and OPRD seen in cluster C1 of Table 1.4 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, the distance to any called functions, measured by the sum of the number of scopes, also increases. This may be because pattern variables are often introduced where new scopes are constructed.

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

TABLE 1.4. Cross-correlated metrics for the Refactoring program.

Cluster C2 of Table 1.4 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.

1.4.4 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. 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 (0.34) for the Peg Solitaire program and 0.6973 (0.487) for the Refactoring program, which are higher than any of the individual metrics correlation values.

The coefficients of the regression analysis for the Peg Solitaire program show that the largest contribution, with a coefficient of 0.4731, comes from OUTD, 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 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 suggests that cross-module calls may need to be measured differently to intra-module calls.

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.

1.5 CONCLUSIONS AND FURTHER WORK

In this paper we have described a number of software metrics 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 metrics to indicate functions that may have an increased risk of containing errors, and which may therefore benefit from more rigorous testing.

Unfortunately, because we were only able to assess two case study programs there remain questions about the general applicability of these metrics to Haskell

programs. The authors would therefore welcome contributions of Haskell programs that would make suitable case studies in order to further expand this analysis. Nevertheless, we were still able to show interesting results.

By analysing the cross-correlation of the metrics we have shown that some of the metrics measure similar or closely related attributes. The regression analysis of the metrics has shown that combining the measurements does increase the correlation, 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 OUTD metric, which measures the dependencies of a function. This suggests that, in common with OO and imperative programs, most of the complexity in a Haskell program lies not within individual functions, but rather in their interaction. We note also that the OUTD metric does not appear to be cross-correlated with the measures of function size, OPRD and OPRT, therefore this result is unlikely to be caused simply by larger functions being more likely to contain bugs.

Overall, this preliminary study using mostly translations of imperative or OO metrics has shown that metrics *can* be used on Haskell programs to indicate areas with increased probability of containing bugs. The success of this preliminary work encourages further exploration, in particular, by designing metrics to analyse Haskell specific features which may provide better predictors of bug locations.

As part of the thesis by Ryder, the results of this preliminary study of metrics have been used to experiment with visualisation tools. These tools aim to exploit the metrics to aid programmers in exploring the source code of their programs, demonstrating one area where metrics can be of use.

1.5.1 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 clarify these results further 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, but is not included here due to space constraints.

The metrics described in this paper are mostly translations of imperative or OO metrics, but Haskell programs contain features not analysed by such metrics, e.g. a powerful type system, higher-order and polymorphic functions, etc. Although we were unable to implement metrics for these features during this project, recent developments in the Haskell community, such as the GHC API [MPJ04, RT05] and Strafunski[LV03], have now made it possible. Therefore, one area to expand this work is the design and evaluation of metrics for these advanced Haskell features.

We would also like to integrate the ideas of software metrics into the HaRe [LRT03] refactoring tool. The aim of such a project would be to use metrics to target refactorings, in line with Fowlers' [FBB+99] work on "bad smells".

REFERENCES

[BEEM95] L. Briand, K. El Emam, and S. Morasca. Theoretical and empirical validation of software product measures. Technical Report ISERN-95-03, Fraunhofer Inst., Germany, 1995.

[BH00] D.J. Barnes and T.R. Hopkins. The evolution and testing of a medium sized numerical package. *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, January 2000.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FH79] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: report to our respondents. *Proc. of GUIDE 48*, 1979. The Guide Corporation.

[FP98] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[GKMS00] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Soft. Eng.*, 26(7), 2000.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[LRT03] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. *Proceedings of the 2003 Haskell Workshop*, 2003. ACM Press.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. *Proc. of PADL'03*, vol. 2562 of *LNCS*, 2003. Springer-Verlag.

[McC92] C. McClure. *The Three Rs of Software Automation*. Prentice-Hall, 1992.

[Mel96] A. Melton, *Software Measurement*. Thompson Computer Press, 1996.

[MPJ04] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. Microsoft Research, Cambridge, UK. http://www.haskell.org/ghc/.

[PJ03] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. http://www.haskell.org/definition/.

[RT05] C. Ryder and S. Thompson. Porting HaRe to the GHC API. Technical Report 8-05, Computing Laboratory, University of Kent, UK, October 2005.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4), 1985.

[Ryd04] C. Ryder. *Software Measurement for Functional Programming*. PhD thesis, Computing Lab, University of Kent, UK, 2004.

[TR03] S. Thompson and C. Reinke. A case study in refactoring functional programs. *Proceedings of 7th Brazilian Symposium on Programming Languages*, May 2003. Journal of Universal Computer Science, Springer-Verlag.

[VdB95] K. Van den Berg. *Software Measurement and Functional Programming*. PhD thesis, Uni. of Twente, Department of Computer Science, Netherlands, 1995.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.