



Kent Academic Repository

**Ryder, Chris (2004) *Software measurement for functional programming*.
Doctor of Philosophy (PhD) thesis, University of Kent.**

Downloaded from

<https://kar.kent.ac.uk/86321/> The University of Kent's Academic Repository KAR

The version of record is available from

<https://doi.org/10.22024/UniKent/01.02.86321>

This document version

UNSPECIFIED

DOI for this version

Licence for this version

CC BY-NC-ND (Attribution-NonCommercial-NoDerivatives)

Additional information

This thesis has been digitised by EThOS, the British Library digitisation service, for purposes of preservation and dissemination. It was uploaded to KAR on 09 February 2021 in order to hold its content and record within University of Kent systems. It is available Open Access using a Creative Commons Attribution, Non-commercial, No Derivatives (<https://creativecommons.org/licenses/by-nc-nd/4.0/>) licence so that the thesis and its author, can benefit from opportunities for increased readership and citation. This was done in line with University of Kent policies (<https://www.kent.ac.uk/is/strategy/docs/Kent%20Open%20Access%20policy.pdf>). If y...

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal*, Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our [Take Down policy](https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies) (available from <https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies>).

SOFTWARE MEASUREMENT FOR
FUNCTIONAL PROGRAMMING

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Christopher Ryder
August 2004

Contents

List of Tables	xiv
List of Figures	xx
List of Examples	xxi
Abstract	xxii
Acknowledgements	xxiii
1 Introduction	1
1.1 Metrics and Debugging	2
1.2 Metrics and the Development Process	3
1.3 Metrics and Visualisation	3
1.4 Overview of this Thesis	4
2 Software Measurement	9
2.1 Ott: A Brief History of Software Metrics	13
2.2 Metrics for Imperative Languages	15
2.2.1 Introduction to metrics for imperative languages	16
2.2.2 Measuring the size of an imperative program	16
2.2.3 Attributes of control flow in imperative languages	18
2.2.4 Metrics for software testing in imperative languages	22
2.2.5 Attributes of modularity in imperative languages	23
2.3 Metrics for Object-Oriented Languages	29

2.3.1	Classifying classes	29
2.3.2	Metrics for methods	30
2.3.3	Class properties	31
2.3.4	Dynamic coupling measures	32
2.3.5	Summary	34
2.4	Software Measurement for Functional Programming Languages . .	34
2.4.1	Readability	35
2.4.2	Validation	35
2.4.3	Summary	40
2.5	Metrics and Time	40
2.5.1	Time as a metric	41
2.5.2	Using time to validate metrics	42
2.5.3	Summary	44
2.6	Measuring Software Design	45
2.6.1	Assessing testability from class diagrams	45
2.6.2	Factorisation and metrics	46
2.6.3	Summary	47
2.7	Metrics for Re-engineering	47
2.7.1	Cohesion metrics for refactoring	48
2.7.2	Finding refactorings via change metrics	50
2.7.3	Summary	54
2.8	Generic and Miscellaneous Metrics	54
2.8.1	Predicting software scalability	54
2.8.2	Measuring system maintainability	56
2.8.3	Using information theory for metrics	57
2.9	Summary	58
3	Validation Methodology	61
3.1	A Brief Introduction to Haskell	62
3.1.1	Characteristics of Functional and Imperative Languages . .	62

3.1.2	Haskell Syntax	63
3.2	Implementing the Metrics: Medina	71
3.2.1	Medina Design	74
3.2.2	Extending Medina	87
3.2.3	Future Expansion	88
3.3	Experimental Methodology	91
3.3.1	First Attempt: Happy	92
3.3.2	Second Attempt: Peg Solitaire	93
3.3.3	An Additional Case Study: Refactoring	95
3.3.4	A Larger Body of Programs	96
3.3.5	Summary	99
4	Software Measurement for Haskell	101
4.1	Measuring the Complexity of Patterns	105
4.1.1	Number of pattern variables	111
4.1.2	Number of overridden or overriding pattern variables	112
4.1.3	Number of constructors	114
4.1.4	Number of wildcards	115
4.1.5	Depth of nesting	117
4.1.6	Pattern size	118
4.1.7	Interaction of pattern attributes	119
4.1.8	Summary	121
4.2	Measuring the Distance Between Declarations and Their Use	123
4.2.1	Distance metrics for multi-module programs	126
4.2.2	Distance by the number of scopes	128
4.2.3	Distance by the number of declarations in scope	128
4.2.4	Distance by the number of source lines	129
4.2.5	Distance by the number of parse tree nodes	130
4.2.6	Interaction of distance measures	130
4.2.7	Summary	133

4.3	Measuring Attributes of Recursive Functions	135
4.3.1	A binary indication of recursion	138
4.3.2	Number of recursive paths	139
4.3.3	Number of trivial and non-trivial recursive paths	141
4.3.4	Sum and product of recursive path lengths	142
4.3.5	Interaction of attributes of recursive functions	143
4.3.6	Summary	144
4.4	Measuring Attributes of Callgraphs	145
4.4.1	Measuring strongly connected components	147
4.4.2	Measuring the indegree of a function	148
4.4.3	Measuring the outdegree of a function	148
4.4.4	Measuring the size of a functions callgraph	149
4.4.5	Interaction of attributes of callgraphs	150
4.4.6	Summary	151
4.5	Measuring Miscellaneous Attributes of Functions	153
4.5.1	Measuring the pathcount of a function	154
4.5.2	Measuring the size of a function	155
4.5.3	Interaction of miscellaneous attributes of functions	156
4.5.4	Summary	157
4.6	Interaction of Attributes of Haskell Programs	158
4.6.1	Cross-correlation of Haskell metrics	158
4.6.2	Regression analysis of Haskell metrics	160
4.7	Summary	162
4.7.1	Statistical analysis	163

5 Trends and Characteristics of Haskell Metrics 164

5.1	Analysis of Cross-correlation of Metrics on a Larger Body of Programs	167
5.1.1	Cross-correlation of recursion metrics	168
5.1.2	Cross-correlation of callgraph measures	169
5.1.3	Cross-correlation of distance measures	169

5.1.4	Cross-correlation of pattern measures	171
5.1.5	Cross-correlation of miscellaneous measures	171
5.1.6	Cross-correlation between different classes of metrics	171
5.1.7	Summary	174
5.2	Typical Values of Metrics	176
5.2.1	Typical values of pattern metrics	176
5.2.2	Typical values of recursion metrics	177
5.2.3	Typical values of callgraph metrics	177
5.2.4	Typical values of distance metrics	178
5.2.5	Typical values of miscellaneous metrics	179
5.2.6	Summary	180
5.3	Summary	182
6	Software Visualisation	186
6.1	Visualisation for Gaining An Overview	190
6.1.1	Scaling, thumbnail views and greeking	191
6.1.2	Focus + Context	191
6.1.3	Zooming and fisheye lenses	194
6.1.4	Perspective wall	195
6.2	Visualisation for Seeing Specifics: Dynamic Queries	196
6.3	Visualising Time	199
6.3.1	Animation	199
6.3.2	Bracketing	200
6.4	The Use (and Abuse) of 3D	202
6.4.1	Abstract views	204
6.4.2	Real-world views	207
6.5	Visualising Software	209
6.5.1	Space-Filling software visualisation	209
6.5.2	SeeSoft	211
6.5.3	Tarantula	215

6.5.4	Information mural	217
6.5.5	Graph display of software	218
6.5.6	Software visualisation using C++ lenses	219
6.5.7	Analysing Java software using metrics and visualisation	221
6.6	Summary	222
7	Software Visualisation for Haskell	224
7.1	Designing Visualisation Systems for Haskell	225
7.1.1	Exploring a single Haskell module	226
7.1.2	Finding modules in a program which have unusual properties	228
7.1.3	Browsing the dependencies between functions and modules	229
7.1.4	Investigating the evolution of a module	236
7.1.5	Summary	237
7.2	Implementing Visualisation Systems for Haskell	239
7.2.1	An initial design	240
7.2.2	Using a web browser as a display engine	241
7.2.3	Implementing a tool for exploring a single Haskell module	242
7.2.4	Implementing a tool for browsing the dependencies between functions and modules	242
7.2.5	Implementing a tool for investigating the evolution of a module	252
7.3	Summary	252
8	Conclusions and Further Work	254
8.1	Summary of Conclusions	255
8.2	Future Work	258
A	Key of Metric Variable Names	261
B	Correlation of Metric Values and Changes	263
C	Cross-correlation of Metrics	268
D	Regression Analysis of Metric Values	275

E	Tables of Cross-correlation	281
E.1	Tables of Cross-correlation of Recursion Metrics	282
E.2	Tables of Cross-correlation of Callgraph Metrics	290
E.3	Tables of Cross-correlation of Distance Metrics	300
E.4	Tables of Cross-correlation of Pattern Metrics	315
E.5	Tables of Cross-correlation of Miscellaneous Metrics	328
F	Tables of Metric Values	334
G	Histograms of Metric Values	341
	Bibliography	359

List of Tables

1	Results from the validation of metrics for Miranda type expressions.	38
2	A selection of metrics and their properties.	60
3	A summary of the number of lines of code in the Medina library.	73
4	Information about the Peg Solitaire case study program. Note: The total sizes are approximate figures only, due to individual modules change sizes at different times during the evolution of the program.	94
5	Information about the Refactoring case study program. Note: The total sizes are approximate figures only, due to individual modules change sizes at different times during the evolution of the program.	96
6	A summary of the case study programs showing their sizes in lines of code and the number of functions and modules they contain. .	98
7	A summary of the typical correlation between various metrics. Continued by Table 8 on Page 184. See Appendix A for a key of metric names.	183
8	A summary of the typical correlation between various metrics. Continued from Table 7 on Page 183. See Appendix A for a key of metric names.	184
9	Measurements of pattern attributes and their correlations with change history.	264
10	Distance measures and their correlations with change history . . .	265
11	Measures of recursion and their correlations with change history.	266
12	Callgraph measurements and their correlations with the number of changes.	266

13	Measures of miscellaneous attributes of functions and their correlations with change history.	267
14	Correlation matrix for pattern attributes.	269
15	Correlation matrix for distance attributes.	270
16	Correlation matrix for recursion measurements.	271
17	Correlation matrix for callgraph measurements.	272
18	Correlation matrix for function measurements.	272
19	Correlation matrix for all measurements from the Peg Solitaire program.	273
20	Correlation matrix for all measurements from the Refactoring program.	274
21	Regression analysis of measurements of pattern attributes.	276
22	Regression analysis of distance metric measurements.	277
23	Regression analysis of recursion measurements from the Peg Solitaire program.	277
24	Regression analysis of callgraph measurements.	278
25	Regression analysis of function measurements.	279
26	Regression analysis of all measurements from the Peg Solitaire program.	280
27	Regression analysis of all measurements from the Refactoring program.	280
28	Correlation between recursion metrics and others for the CGI Library.	283
29	Correlation between recursion metrics and others for the Haskell Cryptographic Library.	283
30	Correlation between recursion metrics and others for the Haskell DSP Library.	284
31	Correlation between recursion metrics and others for FGL.	284
32	Correlation between recursion metrics and others for the Library of Geometric Algorithms.	285

33	Correlation between recursion metrics and others for Haddock. . .	286
34	Correlation between recursion metrics and others for Happy. . . .	286
35	Correlation between recursion metrics and others for Hat.	287
36	Correlation between recursion metrics and others for HaXml. . . .	287
37	Correlation between recursion metrics and others for HUnit. . . .	288
38	Correlation between recursion metrics and others for PCF imple- mentation.	288
39	Correlation between recursion metrics and others for Pretty Printer Library.	289
40	Correlation between recursion metrics and others for Typing Haskell in Haskell.	289
41	Correlation between callgraph metrics and others for the CGI Library.	291
42	Correlation between callgraph metrics and others for the Haskell Cryptographic Library.	292
43	Correlation between callgraph metrics and others for the Haskell DSP Library.	292
44	Correlation between callgraph metrics and others for FGL.	293
45	Correlation between callgraph metrics and others for the Library of Geometric Algorithms.	293
46	Correlation between callgraph metrics and others for GetOpt. . . .	294
47	Correlation between callgraph metrics and others for Haddock. . .	295
48	Correlation between callgraph metrics and others for Happy. . . .	296
49	Correlation between callgraph metrics and others for Hat.	296
50	Correlation between callgraph metrics and others for HaXml. . . .	297
51	Correlation between callgraph metrics and others for HUnit. . . .	297
52	Correlation between callgraph metrics and others for PCF imple- mentation.	298
53	Correlation between callgraph metrics and others for Pretty Printer Library.	298

54	Correlation between callgraph metrics and others for Typing Haskell in Haskell.	299
55	Correlation between distance metrics and others for the CGI Library.	301
56	Correlation between distance metrics and others for the Haskell Cryptographic Library.	302
57	Correlation between distance metrics and others for the Haskell DSP Library.	303
58	Correlation between distance metrics and others for FGL.	304
59	Correlation between distance metrics and others for the Library of Geometric Algorithms.	305
60	Correlation between distance metrics and others for GetOpt.	306
61	Correlation between distance metrics and others for Haddock.	307
62	Correlation between distance metrics and others for Happy.	308
63	Correlation between distance metrics and others for Hat.	309
64	Correlation between distance metrics and others for HaXml.	310
65	Correlation between distance metrics and others for HUnit.	311
66	Correlation between distance metrics and others for PCF implementation.	312
67	Correlation between distance metrics and others for Pretty Printer Library.	313
68	Correlation between distance metrics and others for Typing Haskell in Haskell.	314
69	Correlation between pattern metrics and others for the CGI Library.	316
70	Correlation between pattern metrics and others for the Haskell Cryptographic Library.	317
71	Correlation between pattern metrics and others for the Haskell DSP Library.	318
72	Correlation between pattern metrics and others for FGL.	319

73	Correlation between pattern metrics and others for the Library of Geometric Algorithms.	320
74	Correlation between pattern metrics and others for GetOpt.	321
75	Correlation between pattern metrics and others for Haddock.	321
76	Correlation between pattern metrics and others for Happy.	322
77	Correlation between pattern metrics and others for Hat.	323
78	Correlation between pattern metrics and others for HaXml.	324
79	Correlation between pattern metrics and others for HUnit.	325
80	Correlation between pattern metrics and others for PCF implementation.	325
81	Correlation between pattern metrics and others for Pretty Printer Library.	326
82	Correlation between pattern metrics and others for Typing Haskell in Haskell.	327
83	Correlation between miscellaneous metrics and others for the CGI Library.	329
84	Correlation between miscellaneous metrics and others for the Haskell Cryptographic Library.	329
85	Correlation between miscellaneous metrics and others for the Haskell DSP Library.	329
86	Correlation between miscellaneous metrics and others for FGL.	330
87	Correlation between miscellaneous metrics and others for the Library of Geometric Algorithms.	330
88	Correlation between miscellaneous metrics and others for GetOpt.	330
89	Correlation between miscellaneous metrics and others for Haddock.	331
90	Correlation between miscellaneous metrics and others for Happy.	331
91	Correlation between miscellaneous metrics and others for Hat.	331
92	Correlation between miscellaneous metrics and others for HaXml.	332
93	Correlation between miscellaneous metrics and others for HUnit.	332

94	Correlation between miscellaneous metrics and others for PCF implementation.	332
95	Correlation between miscellaneous metrics and others for Pretty Printer Library.	333
96	Correlation between miscellaneous metrics and others for Typing Haskell in Haskell.	333
97	Mean, Mode, Median and Standard Deviation values of pattern metrics.	335
98	Mean, Mode, Median and Standard Deviation values of recursion metrics.	336
99	Mean, Mode, Median and Standard Deviation values of callgraph metrics.	337
100	Mean, Mode, Median and Standard Deviation values of distance metrics (Part 1 of 2).	338
101	Mean, Mode, Median and Standard Deviation values of distance metrics (Part 2 of 2).	339
102	Mean, Mode, Median and Standard Deviation values of miscellaneous metrics.	340

List of Figures

1	Fixing defects early saves effort.	10
2	Examples of prime flowgraphs.	19
3	Example of an S-graph	20
4	Example coupling graph.	27
5	Plot of path count values against code size in Lines Of Code. Path count plotted on logarithmic scale.	44
6	An illustration of visualising distance measures by Simon, Steinbrücknet and Lewerentz.	49
7	An example of the Split Into or Merge With Superclass refactoring.	51
8	An example of the Split Into or Merge With Subclass refactoring.	51
9	A block diagram of the Medina library metrics sub-system. . . .	72
10	A block diagram of the Medina library visualisation sub-system.	73
11	A simplified diagram of the Medina library module structure. . .	74
12	An example of a <code>ParseTree</code> data structure.	76
13	An example of a <code>BaseAbstractSyntax</code> data structure.	78
14	An example of a <code>RawTaggedBindingList</code> data structure. Some nodes are not included in this diagram in order to clarify the struc- ture.	78
15	An example of an <code>IdentMap</code> data structure. Some nodes are not included in this diagram in order to clarify the structure.	80
16	An example of a <code>ModuleImportGraph</code> structure for the program in Example 2. Some parse tree nodes are hidden for clarity.	81

17	An example of a <code>CallGraph</code> structure for the program in Example 2.	81
18	An example of a <code>TotalCallGraph</code> structure for the program in Example 2.	82
19	An overview of the structure of the basic Medina visualisation components.	84
20	An overview of the design structure of the Medina library. Numbers in brackets indicate approximate code size.	88
21	Example of measuring distance by the number of scopes for the function <code>foo</code>	124
22	Example of measuring distance by the number of declarations brought into scope for the function <code>foo</code>	124
23	Example of measuring distance by the number of source code lines for the function <code>foo</code>	125
24	Example of measuring distance by the number of parse tree nodes for the function <code>foo</code>	125
25	An example of the use of scaling in an image browser, showing both a small scale (a) and a large (b) scale view.	192
26	Example of greeked text.	192
27	An example of a focus + context visualisation in SeeSoft.	193
28	Example of a fisheye lens over text.	194
29	An example of the Mac OS X Dock system using fisheye lenses. Note the sizes of the icons change as the mouse cursor is moved.	195
30	Example of perspective wall.	196
31	The Homefinder system of Williamson and Shneiderman.	197
32	The FilmFinder system of Ahlberg and Shneiderman.	198
33	The GCSpy system by Jones and Printezis.	200
34	A bracketing visualisation by Roberts.	201
35	An example of a Cone-Tree visualisation.	203
36	An example of the CallStax visualisation system.	205

37	An example of the FileVis visualisation system, showing a file containing some low-detail function representations.	205
38	An illustration of visualising distance metrics by Simon, Steinbrücknet and Lewerentz.	206
39	An example of the Software World system by Knight and Munro.	208
40	An illustration of SeeSys by Baker and Eick.	210
41	An illustration of the SeeSoft line representation.	212
42	An illustration of the SeeSoft pixel representation.	213
43	An example of the SeeSoft File Summary Representation.	214
44	An example of the Tarantula system by Eagan, Harrold, Jones and Stasko.	216
45	An example of data shown with and without the use of the information mural technique.	217
46	An example of graph display of software in which node width, height and colour are all used to indicate different attributes.	219
47	An illustration of the lens technique used by Cain and McCrindle.	220
48	An illustration of the Shimba system.	221
49	Final version of the focus + context tool. The pathcount metric is used to colour code functions in this example, although it is of course possible to use other metrics instead.	227
50	A tool for browsing multiple source files. This example use pathcount for colour coding the functions in the source files, although other metrics can be selected.	230
51	A file browser tool showing all functions which use the <code>map</code> function red. This is achieved using a simple binary metric to indicate usage of the <code>map</code> function.	231
52	Example of a callgraph for two modules.	232
53	Part of a callgraph in which edges are hard to distinguish.	233

54	An illustration of mouse overs in the module and callgraph browser. Edges are colour coded according to how many symbols are imported along them.	235
55	A browser for exploring the history of a source file. Functions are colour coded using the pathcount metric, but any of the Haskell metrics presented in this thesis can be used.	238
56	First version of the focus + context tool. Colour coding is performed using the pathcount metric, although other metrics could be used.	243
57	The first version of the module and callgraph browser.	245
58	The module and callgraph browser implemented in SVG using colour and line styles.	246
59	The final version of the module browser. Colours for the edges are chosen based on the number of symbols imported and used along those edges. It would also be possible to use metrics to colour code nodes. The general colour scheme used, such as the background colour, etc, can be configured to the users taste and needs. . . .	248
60	Popup window after clicking an edge.	249
61	Popup window after clicking a module node.	250
62	Histograms of “Arc-to-node ratio” values.	342
63	Histograms of “Strongly connected component size” values.	342
64	Histograms of “Indegree” values.	343
65	Histograms of “Outdegree” values.	343
66	Histograms of “Depth” values.	344
67	Histograms of “Width” values.	344
68	Histograms of “Binary recursion” values.	345
69	Histograms of “Number of non-trivial recursive paths” values. . .	345
70	Histograms of “Number of trivial recursive paths” values.	346
71	Histograms of “Number of recursive paths” values.	346
72	Histograms of “Sum of lengths of recursive paths” values.	347

73	Histograms of “Product of lengths of recursive paths” values.	347
74	Histograms of “Number of pattern variables” values.	348
75	Histograms of “Sum of depth of patterns” values.	348
76	Histograms of “Maximum depth of patterns” values.	349
77	Histograms of “Number of overridden or overriding pattern variables” values.	349
78	Histograms of “Number of constructors in pattern” values.	350
79	Histograms of “Pattern size” values.	350
80	Histograms of “Number of wildcards in pattern” values.	351
81	Histograms of “Pathcount” values.	351
82	Histograms of “Number of operands” values.	352
83	Histograms of “Number of operators” values.	352
84	Histograms of “Distance by the sum of the number of scopes” values.	353
85	Histograms of “Distance by the maximum number of scopes” values.	353
86	Histograms of “Distance by the average number of scopes” values.	354
87	Histograms of “Distance by the sum of the number of declarations in scope” values.	354
88	Histograms of “Distance by the maximum of the number of declarations in scope” values.	355
89	Histograms of “Distance by the average number of declarations in scope” values.	355
90	Histograms of “Distance by the sum of the number of source lines” values.	356
91	Histograms of “Distance by the maximum number of source lines” values.	356
92	Histograms of “Distance by the average number of source lines” values.	357
93	Histograms of “Distance by the sum of the number of parse tree nodes” values.	357

94	Histograms of “Distance by the maximum number of parse tree nodes” values.	358
95	Histograms of “Distance by the average number of parse tree nodes” values.	358

List of Examples

1	An example of an abstract data type	67
2	An example of a multi-module program.	80
3	Using pattern matching to introduce identifiers.	105
4	Using pattern matching with algebraic data types.	105
5	Using nested patterns.	106
6	Using wildcards in patterns.	106
7	Using field names with data structures in patterns.	107
8	Using patterns on the right hand side of functions.	107
9	Differences in the normal form of functions.	109
10	Overridden pattern variables.	113
11	Complications of using wildcards in patterns.	116
12	Nested patterns.	118
13	An example of trivial recursion.	135
14	An example of non-trivial recursive behaviour.	136
15	Examples of multiple recursive paths in functions.	139
16	Example of the difference between recursive paths and recursive calls in functions.	140
17	A function with a subtle pathcount value.	154

Abstract

This thesis 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 visualising 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 has been explored.
- Visualisation tools have been developed for further exploring the metric values in conjunction with program source code.

Acknowledgements

I would like to express my thanks to the Engineering and Physical Sciences Research Council for funding this work and to Microsoft Research for the time I spent at their Cambridge research lab in November 2000.

I would also like to extend my thanks to my supervisor, Professor Simon Thompson, for guiding me through this work and for the countless hours of proof reading.

My personal thanks must go to my mother, father and sister for their support throughout my university life. I could not have wished for better parents or a more supportive family and could not have completed this work without them.

And finally, to Kristina for pushing me to finish the corrections. Thank you.

For my parents

For my sister

For Kristina

Chapter 1

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 [98].

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 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 [61] on monad transformers and that of Swierstra and his co-workers [92] on combinator 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.

1.1 Metrics and Debugging

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 pathcount 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.

This thesis aims to examine both imperative and object-oriented metrics to assess their applicability to functional programming, and to develop new metrics for the areas which are not covered by traditional metrics.

1.2 Metrics and the Development Process

Functional programming currently lacks a commonly adopted design methodology, such as UML for object-oriented development. Proposals have been presented, for instance by Russell [84], but as yet such ideas have not been widely adopted. Instead, newer methodologies such as Extreme Programming [13] appear to better suit the current ad hoc methodologies used by functional programmers.

Although there is no formal recognition that functional programmers are using an Extreme Programming methodology, the ideas of performing unit tests early and often, and of continual incremental development, seems to be reminiscent of the typical development style used with functional languages.

A development process which has gained popularity with the rise of Extreme Programming is Refactoring [36]. Refactoring is the process of making changes to software which preserve behaviour but improve structure or design. Extreme Programming involves frequent refactoring as a program is written. However, deciding exactly which refactorings to apply and where to apply them relies on the intuition and experience of the programmer. Software metrics offer a way to discover parts of a program which have unusual characteristics, or “bad smells” in Fowler’s [36] terminology, and which therefore may be good targets for refactoring.

1.3 Metrics and Visualisation

Software measurement tools can generate large amounts of data from non-trivial programs. Because of this a natural extension to measurement tools is to add some form of visualisation system. This thesis therefore also discusses methods of

visualising and exploring software systems and metric values, presenting a selection of visualisation systems for Haskell programs implemented as a flexible and extensible library.

The combination of visualisation tools and metrics can be a useful tool for use with other development activities such as refactoring, where visualisation tools can be used to “browse” for refactoring targets or to indicate how much a program will be affected by a particular refactoring.

1.4 Overview of this Thesis

This thesis presents and analyses a selection of software metrics designed to measure attributes of programs written in the functional programming language Haskell which may not be considered by existing metrics for imperative or object-oriented languages. The work is then extended to show how these metrics can be incorporated into software visualisation systems in order to provide tools to programmers that can help their understanding of their programs and aid in the application of other development processes such as refactoring. This thesis provides the following contributions to the field of software metrics for functional programming languages.

- 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 has been explored.
- Visualisation tools have been developed for further exploring the metric values in conjunction with program source code.

The experiments and studies carried out to make these contributions reached a number of conclusions which are detailed briefly here.

- 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. Analysing the correlation between metrics led to the following observations.
 - The occurrence of non-trivial recursion, e.g. recursion in which a function does not directly call itself, in Haskell programs is quite unusual, and is associated with complex program behaviour. However the occurrence of trivial recursion, where a function directly calls itself, is common.
 - The callgraphs of functions in whole Haskell programs generally grow uniformly in both depth and width, rather than becoming long and thin, or short and wide.
 - Large functions tend to include a greater number of local declarations than small functions. This is most likely because local declarations allow one to attach names to parts of a large, perhaps complex, function. Therefore functions which are large, but do not have many local declarations may well be difficult to understand.
 - The functions used by a single function, `foo`, will tend to be located close to `foo` in the source files.
- 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 “Outdegree” (*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.

This implies that there is no single attribute that makes a Haskell program complex, rather that the complexity is a result of a number of attributes.

- Typically the metrics presented generate values that are distributed at the low end of their scales, suggesting that it should be possible to select thresholds to indicate when the various attributes are exhibiting unusual values, because any values above a threshold are very likely to indicate unusual behaviour that may warrant further investigation.
- Software metrics can generate large amounts of data and so there is a clear need for tools that can present the interesting points of the data to the user, while hiding the bulk of the uninteresting data.
- Visualisation tools, combined with software metrics, have applications in other parts of the software engineering process. Tools such as Tarantula [29] have shown the use of visualisation tools for analysing measurements about the testing process. Work in this thesis has shown that visualisation systems like the file browser tool, presented in Section 7.1.2 of Chapter 7, have uses in the refactoring process, for example, by indicating all the sections of a program's source code that may be affected by a particular refactoring.

The work in this thesis is presented in two main parts, covering software metrics and software visualisation respectively. The thesis is divided into a number of chapters which are introduced here.

- Chapter 2, *Software Measurement*, describes why software measurement is seen as important by the imperative and object-oriented (OO) development community, and presents various software metrics used in imperative and OO programming. The chapter also examines how metrics have been used to aid other development processes such as testing and refactoring.
- Chapter 3, *Validation Methodology*, describes the experimental methods used to assess the various metrics described later in Chapter 4, *Software*

Measurement for Haskell, and Chapter 5, *Trends and Characteristics of Haskell Metrics*, along with the rationale for the choice of programs used as case studies. The chapter also describes what characteristics are expected of a candidate case study program, and highlights the difficulty of finding suitable programs.

- Chapter 4, *Software Measurement for Haskell*, applies ideas from imperative and OO metrics to a functional programming language, Haskell. A selection of metrics for Haskell are presented and statistical analysis of the results of two case studies is performed to assess the correlation between metric values and changes (including bug fixes) in the evolution history of software projects. Statistical analysis is also used to determine correlation between metric values, indicating metrics which are closely related.
- Chapter 5, *Trends and Characteristics of Haskell Metrics*, extends the work presented in Chapter 4, *Software Measurement for Haskell*, to study a wider selection of case study programs in order to examine the relationships between the various attributes measured by the metrics. Further work is performed to discover the typical metric values that programs exhibit, in order to understand what constitutes an anomalous value of a particular metric.
- Chapter 6, *Software Visualisation*, discusses software visualisation systems that have been used to explore and investigate software systems, and examines the main issues involved in designing visualisation tools. Visualisation tools are particularly useful for exploring the results of software metrics, which will produce large amounts of data for non-trivial programs. In particular, the chapter discusses how visualisation systems for Java programs have been combined with metrics and with information about test cases.
- Chapter 7, *Software Visualisation for Haskell*, presents an implementation of a flexible library of visualisation components which can be combined to build various visualisation tools for exploring Haskell programs, drawing

inspiration from the work in Chapter 6. The systems presented offer ways to explore either single files or collections of files, support visualising the evolution of source files, and allow the end user to view and explore the module hierarchy and callgraph of a program. The library is intended to allow the end user to customise or create visualisation systems or to combine elements of different visualisations, thereby providing flexibility. The library is written in Haskell, ensuring that end users do not need to learn a new programming language, and uses HOpenGL [76] to display the user interface.

- Chapter 8, *Conclusions and Further Work*, summarises the thesis and lists the main conclusions from this work. The chapter also suggests several directions for possible future work, including closer ties to the refactoring and testing processes, and integration with evolutionary computing technologies.

This thesis also includes a number of appendices containing tables and graphs which are referenced in Chapter 4, *Software Measurement for Haskell* and Chapter 5, *Trends and Characteristics of Haskell Metrics*. These tables and figures were separated into appendices to preserve the clarity of those chapters due to the large space the tables and graphs occupy.

Chapter 2

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 [87].

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 [90], and are shown here in Figure 1. 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 [87] 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. These points raise a number of questions, such as those presented by Smith [90],

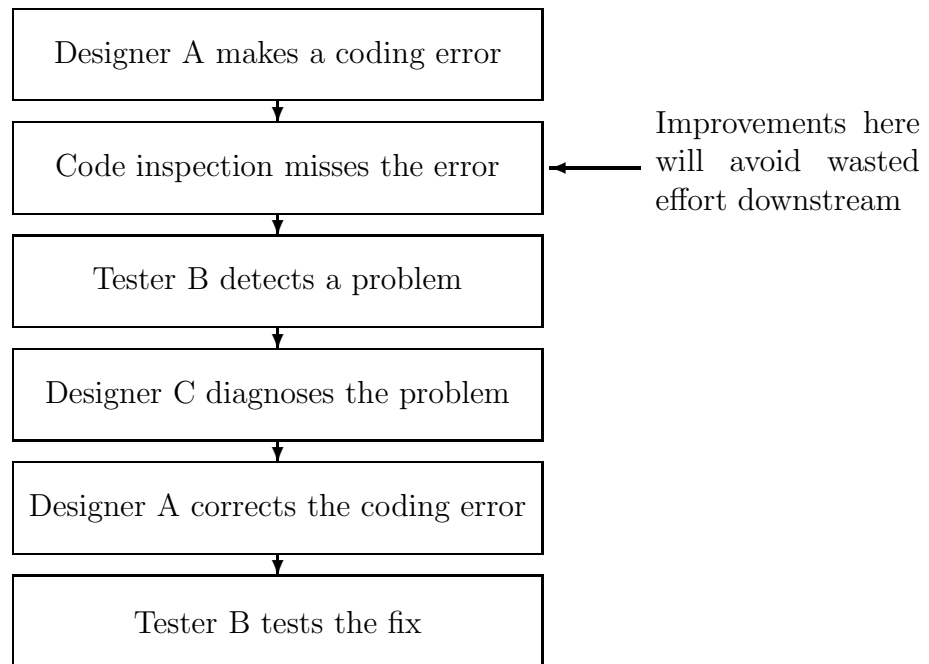


Figure 1: Fixing defects early saves effort.

about how one might reduce the number of software defects that survive the early stages of development:

- 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 [79] 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 past history of the system has been exploited in many processes and is particularly useful where there is a collection of projects which can be expected to share many characteristics, however this suffers from two drawbacks. Firstly, new code has no defect history so such processes may not help in this circumstance, and secondly, defect fixes often only address symptoms of the defect and can leave the real defect unfixed and undetected.
- How can these methods be combined? By combining a number of techniques it may be possible to take the positive aspects of the techniques while reducing the effects of the negative aspects, such as the time consuming nature of peer reviews.

Software measurement is a technique in which quantitative measures, often called *metrics*, are taken from the source code of a program. Typically metrics attempt to quantify how complex a piece of source code is to understand, modify or test. This notion of complexity should not be confused with the computational complexity of an algorithm, which is typically denoted using $O()$ notation. Computational complexity is concerned with runtime behaviour, rather than how easy or hard it is for a programmer to maintain.

One of the claims of software measurement is that it can help identify the parts of a system which are most likely to benefit from inspection. This allows resources to be focused where they will help most and allows them to be used in their most effective way, e.g. peer reviews of small sections of code, rather than large sections, or indicating parts of a program that may benefit from refactoring code changes.

Software measurement is increasingly becoming part of the development process in many software development companies. Many such companies participate in process certification programs such as ISO9000 [78] or Software CMM [77]. These certification programs indicate the quality and maturity of the development process used in a company, and many large software houses view these as an important benchmark. Interest in these programs is not confined only to the developers either, clients are also increasingly specifying certification levels as part of the tendering process. This in turn has created a drive by software development companies to achieve the necessary improvements in the development process to gain higher certification levels.

This increased desire to improve the quality of the software development process has correspondingly increased the acceptance of software measurement. For instance, Software CMM [77] highlights that engineers may have detailed insight, or *visibility*, into the state of a project due to their first-hand knowledge, but may only see a small part of a large project, while managers may lack visibility into the project, relying instead on periodic reviews. Level 4 of Software CMM requires that productivity and quality are measured, and that software processes are instrumented with well-defined and consistent measurements. The purpose of these measurements is to provide a quantitative foundation for evaluating the projects' software processes and products. While this thesis does not study process measurements it does study product measurements.

The remainder of this chapter will explore some of the work already performed in the software measurement field and discusses the various methods of validation that have been used in software measurement. There has been a large amount

of work in this area since its inception in the early 1970's, and as such the work presented in this chapter can only be a broad survey of the field. This chapter mostly covers static measures of software, but does briefly visit the area of dynamic measures. Dynamic measures are not considered in the rest of this thesis, and therefore all measures discussed should be assumed to be static measures unless otherwise stated. The remainder of this chapter is divided into the following parts.

- Section 2.1 presents a brief summary of the history of software measurement.
- Section 2.2 describes some of the metrics for imperative languages.
- Section 2.3 examines metrics for analysing object-oriented languages.
- Section 2.4 introduces previous work that applies software measurement techniques to functional programming languages.
- Section 2.5 investigates the role of time in the use of software metrics.
- Section 2.6 discusses the ways in which metrics have been used in quantifying aspects of design artifacts.
- Section 2.7 examines the use of software measurement to aid in the process of re-engineering software systems.
- Section 2.8 presents some metrics which do not fit in the classifications of the previous sections.
- Section 2.9 summarises the chapter.

2.1 Ott: A Brief History of Software Metrics

The history of software metrics makes an interesting lesson on the pitfalls of over zealous promotion of research results. In [73] Ott provides a detailed history of research in software metrics which it is worth summarising briefly in this section.

The first major work published in the field of software metrics was Halstead's work on "Software Science" [42] in the early 1970's, which is covered in more detail later in this section. Halstead wanted to form a science that would be the foundation of software engineering, much like physics is to electrical engineering. Halstead frequently drew analogies between software issues and concepts from other disciplines, such as psychology. Several researchers started looking at software metrics after Halstead's work, but the field was viewed with scepticism by many people. Criticism was levelled at both Halstead's experimental methods, in which he sometimes used inappropriate statistical methods, and at the theory on which "Software Science" was based. In particular, Halstead's tendency to take findings from disparate disciplines and use them out of context within the software engineering field attracted much criticism and this left something of a cloud over the field for some time.

Much of the early work following Halstead's was focused on measuring complexity. Such work is typified by the work of Bell and Sullivan [14], as well as that of McCabe [67] which is described later in this chapter. During the mid 70's several industrial projects started to look at software measurement. Their goal was to provide some way to quantify the software development process so that the effect of changes to the process could be monitored. This prompted many researchers during the late 70's and early 80's to look again at the field. Many "new" metrics were developed, however much of the computer science community still viewed software metrics with scepticism. Much of the criticism was still levelled at Halstead, but some more general concerns were raised about the validity of attempting to measure something as multi-faceted as software complexity by a single number.

Since then, the field of software measurement has progressed. Researchers in the field have learned from Halstead's mistakes and software measurement has gained a more solid foundation, both in statistical analysis and in empirical validation. An excellent set of guidelines for empirical research and statistical analysis for software metrics is provided by Kitchenham and her co-workers [52].

These guidelines are not described here, but are to be recommended to anyone involved in empirical studies of software.

This improved foundation has helped the use of software metrics to become more accepted. An indication of the acceptance of software measurement is indicated by a recent conference [1, 2] on software measurement at which a number of large companies, such as various telecommunications and aero-space manufacturers, were represented both as speakers and listeners.

The rest of this chapter presents a survey of the past and present work relating to software measurement, both in general and for specific programming paradigms.

2.2 Metrics for Imperative Languages

Until relatively recently most software has been written in imperative languages and correspondingly most research into software measurement has been conducted on such languages. There have been many metrics developed and it would be impossible to describe all of them in detail, however in the following sections a general overview of the different types of metrics will be presented. Fenton's [32] text provides descriptions of these metrics in greater depth than is possible here. The rest of this section is divided into the follow parts.

- Section 2.2.1 presents a brief introduction to software metrics for traditional imperative languages.
- Section 2.2.2 describes some of the measurement techniques used for quantifying program size.
- Section 2.2.3 surveys some of the useful metrics used to measure factors of control flow.
- Section 2.2.4 looks at how metrics relate to testing of imperative programs.
- Section 2.2.5 covers the metrics that can be used to help understand modularity in programs.

2.2.1 Introduction to metrics for imperative languages

A software program has many attributes that can be measured. Fenton classifies such attributes into two classes:

- *Internal attributes.* Those attributes of software products dependent only on the software product, such as size.
- *External attributes.* Those attributes measured only with respect to how the product relates to its environment, such as reliability.

Fenton stresses that although it is possible to learn through measurement that a particular attribute (for example “complexity”) of a program is particularly high, software measurement cannot tell you *how* to improve the program. Software measurement can however be used to give *clues* to improve software. Coupled with this point, Fenton observes that the term “complexity” is commonly used to capture the “totality” of internal attributes, but that it does not articulate any particular attribute of complexity very well. Fenton therefore argues that although people feel complexity should be summarised by a single value, a better approach is to learn how to quantify the attributes that contribute to complexity, such as those described in this chapter, and thus improve the quality of software systems in that way.

2.2.2 Measuring the size of an imperative program

One of the simplest attributes of a program is its size. Program size, although primitive, can be a useful measure of complexity because large pieces of code could be expected to be harder to understand and maintain than small pieces of code, all other considerations being equal.

There are many ways one might choose to measure the size of a program. Conte [23] claims that the most common definition is to count the number of lines of code where a line of code, or *LOC*, is defined as:

“...any line of program text that is not a comment or blank line, regardless of the number of statements on the line. This includes all lines containing program headers, declarations and executable and non-executable statements.”

Fenton [32] calls this definition of program size *Non-Comment Lines Of Code* or *NCLOC*. This measure does not consider the size of any comments in the program text, however comments may still be of interest because they are often a source of documentation for the adjacent functions or the containing source file. For this reason Fenton also defines *Comment Lines Of Code*, or *CLOC*, as the number of comment lines in the program text. From this, the total program size will be $LOC = NCLOC + CLOC$.

Fenton believes it is generally useful to gather both NCLOC and CLOC because they measure different attributes. NCLOC measures the “size” of the code in a program, whereas CLOC measures the amount of documentation contained in the program source code in the form of comments.

Fenton also notes that many people find it tempting to combine some notion of effort, e.g. the amount of “stuff” on a line, with the program length but he argues that program length should be kept “atomic” rather than creating a hybrid length measure. An example of such a hybrid measure is Halstead’s *program volume*.

Halstead’s program volume [42] is a measure of the size of a program. Halstead classifies symbols in a program into two classes, those that are used to specify actions are classed as *operators* while symbols that represent data are classed as *operands*. Halstead defines four basic measures from his symbol classifications:

η_1 The number of unique or distinct operators in the program.

N_1 The total number of operators in the program.

η_2 The number of unique or distinct operands in the program.

N_2 The total number of operands in the program.

From these elemental measures Halstead defines program volume as $V = N \times \log_2 \eta$, where N is the program length, defined as $N = N_1 + N_2$, which is essentially the number of symbols in the program, and η is the program *vocabulary*, defined as $\eta = \eta_1 + \eta_2$.

The program volume measure indicates the size of the smallest possible representation of the program in bits. This occurs because $\log_2 \eta$ indicates the number of bits needed to represent all the individual elements of the program, while N indicates how many elements there are in the program.

2.2.3 Attributes of control flow in imperative languages

One of the important aspects of imperative programs is their control flow, that is, the path of execution through a program. Control flow can often be non-trivial and programs with complex control flow can become difficult to comprehend, test, and maintain. It is therefore desirable to attempt to quantify and manage the control flow while a program is being developed, which might allow for easier maintenance in the future. There has been much work on using software measurement to quantify the control flow of programs, some of which will be discussed in the following sections.

The control flow of a program is usually modelled using *flowgraphs*. Flowgraphs are directed graphs which indicate the execution paths through a program. Some flowgraphs, such as if-then-else constructs, appear very regularly in program construction and have therefore been given special names and are known collectively as *prime flowgraphs*. These prime flowgraphs have been illustrated in Figure 2.

Flowgraphs can be combined through sequencing and nesting. The notation used for sequencing is $(F_1; F_2; \dots)$, e.g. $(P_0; D_0)$. The notation used for nesting is $F(F_1 \text{ on } x_1, F_2 \text{ on } x_2, \dots)$ which states that flowgraph F_1 is nested on node x_1 on the flowgraph F , and likewise F_2 is nested on node x_2 , e.g. $D_1(D_2 \text{ on } A)$. This is often abbreviated to $F(F_1, F_2, \dots)$. To illustrate this notation further, consider the flowgraph in Figure 3. This graph represents a simple program to read the

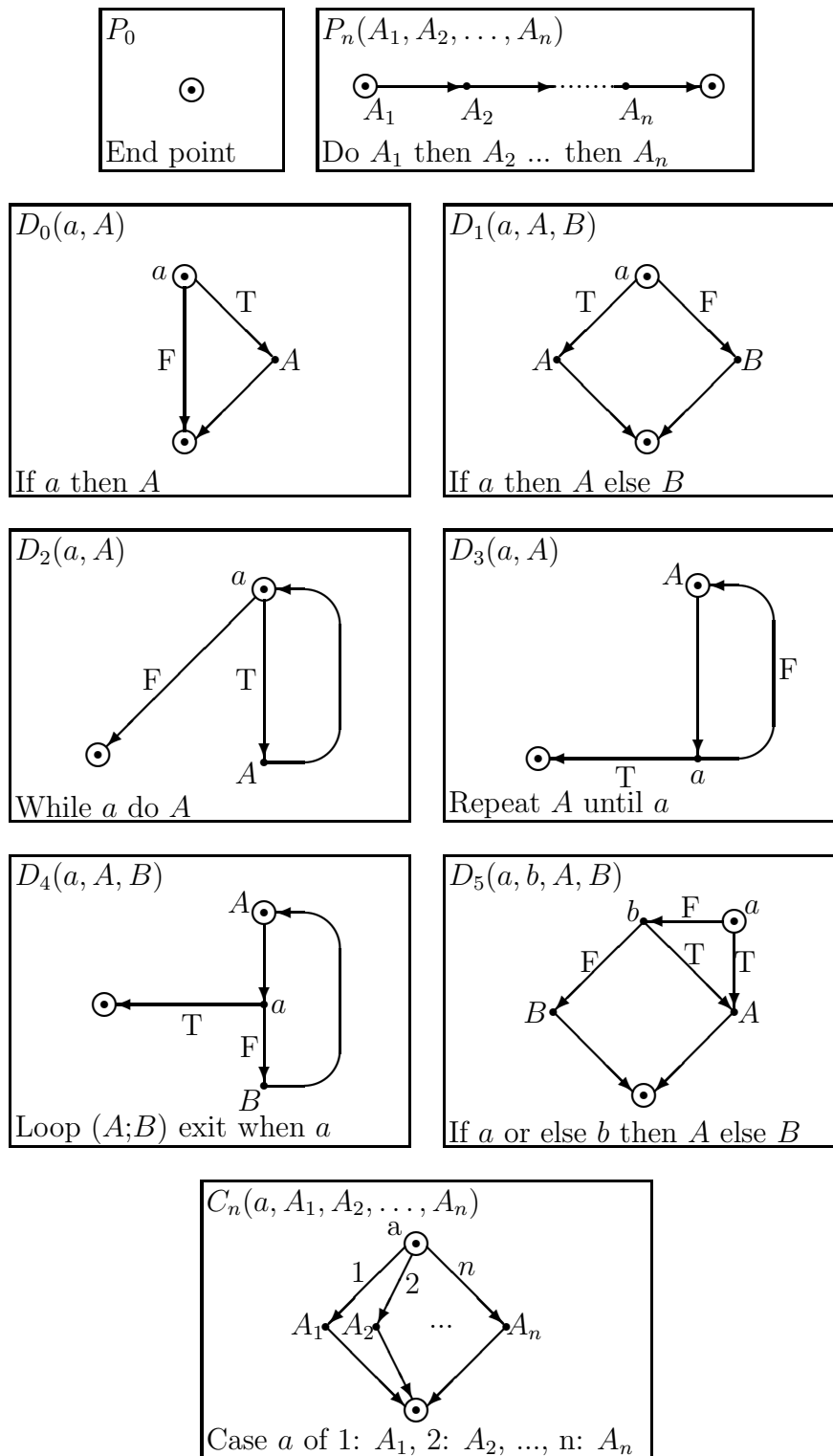


Figure 2: Examples of prime flowgraphs.

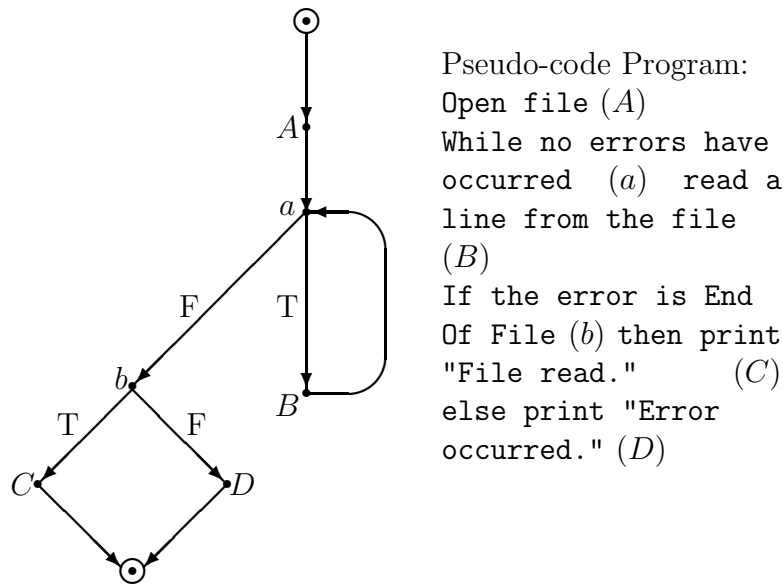


Figure 3: Example of an S-graph

contents of a file, and is constructed from a number of prime flowgraphs, such as a “while a do A” loop, $D_2(a, B)$, and an “if-then-else” structure, $D_1(b, C, D)$. These prime flowgraphs are composed in sequence, thus the whole flowgraph could be represented as the sequence $(P_2; D_2; D_1)$.

Fenton claims it is possible to use the flowgraph of a section of code to determine whether that code is “well structured”. This is done by first nominating a set S of prime flowgraphs. The choice of which prime flowgraphs to include in S might depend, for instance, upon the particular programming language being used, due to the different control structures available in different languages.

Any control flowgraph that is composed only from elements of set S is then classified as structured. Fenton calls such structured graphs *S-graphs*. Figure 3 illustrates an example of an S-graph.

Any flowgraph can be decomposed into the component flowgraphs and so associated with each flowgraph is a decomposition tree which describes how the flowgraph may be built from prime flowgraphs using sequencing and nesting. In [33] Fenton and Whitty describe a *Prime Decomposition Theorem* which states the following:

“Every flowgraph has a unique decomposition into a hierarchy of prime flowgraphs.”

This theorem and its proof describes how every program has some quantifiable degree of “structuredness” which can be characterised by its decomposition tree. This occurs because the only structures which cannot be decomposed are the prime flowgraphs and therefore, unless a program consists of only a single prime flowgraph, it must be possible to decompose the program to some extent.

Fenton believes that the Prime Decomposition Theorem can be used to help “restructure” code in an optimal manner by identifying those primes that are causing “spaghetti” code. This can be done by calculating the decomposition tree for the given program flowgraph. Any elements of the decomposition tree which are not part of the S set indicate sections of code which are unstructured and therefore are likely to cause “spaghetti” code.

Flowgraphs provide a source of many measures such as depth of nesting and complexity. One such well known and commonly cited metric is McCabe’s *cyclomatic complexity* metric [67]. Cyclomatic complexity was presented as a measure of the complexity of a program although, as will be seen in Section 2.2.4, it is also useful as a measure of testability.

Cyclomatic complexity is a measure of the complexity of the decision structure in a program. Programs are represented as a flowgraph G in which there is a single entry point and a single exit point. The metric is defined as $v(G) = e - n + 2$ with e , the total number of arcs, and n , the total number of nodes. This metric counts the number of independent paths through the program, where an independent path is any path through the program that introduces at least one new set of processing statements. The number of independent paths is a good indicator of the complexity of a program because generally a program becomes harder to understand as the number of paths increases. It can also be shown that for a graph in which all decision nodes have an outdegree of 2, that is, every node has two outgoing edges, $v(G)$ is equal to one plus the number of decision nodes in the flowgraph.

Although McCabe's work on Cyclomatic Complexity remains the most heavily cited and best known control flow measure, there have been many other control flow complexity measures defined by many researchers. These other control flow complexity measures are not discussed in this thesis, but a comprehensive analysis of these metrics is provided by Zuse [107].

2.2.4 Metrics for software testing in imperative languages

Effective software testing is recognised as being one of the hardest tasks in software engineering. Because of this it is important that software is written in such a way as to ease the testing process. Software measurement can have a role to play in helping achieve this goal.

Fenton is quick to point out that although it is possible to *measure* how difficult a program will be to test, software measurement *cannot* explain how to change a program to improve testability. Despite this, software measurement is still of use because it can indicate areas where a program may be difficult to test, and so can help direct programmer effort towards the areas of a program where it may be most needed and therefore most cost effective.

A measurement that is more commonly associated with complexity measures than testability is McCabe's cyclomatic complexity measure. Cyclomatic complexity measure, previously defined in Section 2.2.3, counts the number of linearly independent execution paths in a program and so forms an important indication of how difficult a program may be to test. Generally, the greater the number of paths, the more test cases are needed to fully test all paths of the program.

There are many test strategies such as those described and compared by Basili and Selby [11] and those featured in almost any text on software engineering. Because of the wide range of test strategies, it would be useful to be able to measure the number of test cases required for each strategy for a given program and it may be possible to use metrics for this purpose. For instance, McCabe's cyclomatic complexity can measure the number of test cases needed for structured testing. Each such metric is only an *indication* of testability since it is much easier

to calculate an estimate of the *number* of test cases needed than it is to generate the *actual* test cases.

Whatever testing strategy is used, it is important to know how much of the program is being executed by the test cases. This is known as *test coverage* or the *Test Effectiveness Ratio*. It is important to know the test coverage because it indicates how effective the set of test cases are. This is useful because there is little point in running a large number of test cases if most of them exercise the same execution path!

For a test designed to exercise objects of type T , where T might be loops, branches, etc, the Test Effectiveness Ratio for that test is defined as :

$$TER_T = \frac{\text{number of T objects exercised } \geq \text{once}}{\text{total number of T objects}}$$

To calculate the Test Effectiveness Ratio it is necessary to determine which objects were exercised for a specific set of test data. This may be difficult to achieve statically, so it is often necessary to perform some form of runtime execution tracing to be able to perform this measurement.

The use of metrics to predict, or give indications of, how easily a program may be tested has been explored by many researchers in many papers, such as those by Bache and Mullerburg [7], Freedman [37], Binder [15] and Woodward and Al-Khanjari [103], although there is often little in the way of validation presented with such work.

2.2.5 Attributes of modularity in imperative languages

Most modern programming languages provide some support for dividing a program into modules. Such modular programs are generally easier to understand and make better reuse of code, although Hatton [44] suggests that a program consisting of lots of small modules may be harder to understand than a program consisting of a smaller number of slightly larger modules. Yourdon and Constantine [106] claim that the shape, or *morphology*, of the overall program structure

can be a useful source of information about the design of a software system. *Callgraphs* are directed graphs in which nodes represent functions and edges represent calls between functions. Callgraphs can be constructed from a software system and provide many attributes such as width and depth that can be measured quite simply. A few of the more interesting measures they define are described below.

- *Size*. This can be the number of nodes, the number of arcs, or the sum of both. The size of a callgraph affects the complexity of the program. The greater the size of the callgraph, the more dependencies between different parts of the program there is likely to be, and so the greater the complexity.
- *Depth*. The length of the longest path from the root node to a leaf node.
- *Width*. The maximum number of nodes at any one level.
- *Arc-to-node ratio*. This is a connectivity density measure that increases as more connections, or function calls, are made and is similar to cyclomatic complexity. A high arc-to-node ratio indicates complex interactions between functions, and thus may highlight areas of program code that may benefit from re-engineering. Section 2.7 examines the use of metrics in the re-engineering process.

As well as the simple measures described above there have also been some more esoteric metrics designed for use on callgraphs. For instance, Ince and Hekmatpour [47] derived a novel measure they term *impurity*. Impurity is the extent to which a callgraph deviates from an n-ary tree. Their assertion is that the more a system deviates, the worse the architecture of the software system. However it is not completely clear that this is necessarily a desirable property because it appears to discourage code reuse, because their measure implies that all nodes should only be used once.

Code reuse is generally considered an important part of dividing a program into modules because reusing code can reduce both programming and testing effort. Fenton classifies reuse into two categories.

- *Public Reuse*. Reusing code which was constructed externally to the program. e.g. library code.
- *Private Reuse*. Reusing modules within the same program.

Both public and private reuse can be measured from a callgraph. Reuse may be measured either by the number of times a module is used or by the number of modules that use a particular module. Although these two measures seem to be the same, consider the situation where one module uses a second module multiple times, for example by using several functions that are exported by the second module. In this situation the two measures will then give different results.

Coupling

One of the intentions of dividing a program into modules is to prevent changes in one module affecting other modules. In this situation the extent to which two modules are coupled is an important consideration. If two modules are too tightly coupled it will not be possible to change either module independently. In the context of software measurement, coupling is normally considered between pairs of modules. The coupling of a system as a whole is called *global coupling*. Fenton classifies coupling between modules in an imperative language into six classes:

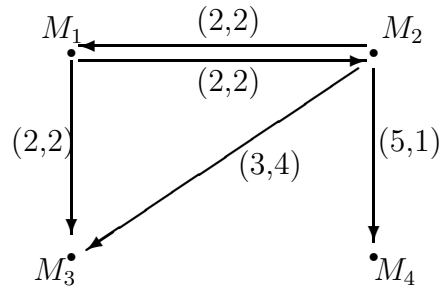
- *Content Coupling, R_5* . $(x, y) \in R_5$ if x changes things inside of y , for instance by changing values defined in y .
- *Common Coupling, R_4* . $(x, y) \in R_4$ if x and y both refer to the same global data. This can cause problems if the format of the global data is changed.
- *Control Coupling, R_3* . $(x, y) \in R_3$ if x passes a parameter to y with the intention of controlling the behaviour of y . An example might be if y is a function that draws shapes on a computer screen. Different parameters to y might cause it to perform different actions, such as drawing a triangle or a square.

- *Stamp Coupling, R_2 .* $(x, y) \in R_2$ if x and y both accept the same defined type as a parameter. This may create interdependencies between otherwise unrelated modules.
- *Data Coupling, R_1 .* $(x, y) \in R_1$ if x and y communicate by parameters which do not incorporate any control element. This occurs when x and y perform the same tasks regardless of the data passed between them. An example might be passing parameters to a mathematical function, the function performs the same actions regardless of the parameters passed. This is in contrast to control coupling in which different sets of actions may be performed depending on the parameters that are passed. Data coupling is the minimum necessary coupling for any communication between modules.
- *No Coupling, R_0 .* $(x, y) \in R_0$ if x and y are completely independent.

Fenton orders these classifications of coupling from R_0 being good to R_5 being bad. It is not usually possible to have only R_0 coupling between all modules, one exception might be a library of utility functions, so Fenton's ordering is intended to be treated only as a guideline.

It is also worth noting that in practice it may be difficult to exactly classify coupling by Fenton's classifications. For instance, determining between R_3 and R_1 coupling may be difficult or impossible to do mechanically and may therefore require manual inspection in order to decide the exact classification.

The coupling of a system can be modelled using a *coupling-model graph*. A coupling-model graph can provide a good method of visualising the coupling between modules. An example of such a graph is shown in Figure 4, where edges of the graph are marked with pairs, (x, y) , which indicates the type of coupling (R_x) and the number of occurrences (y) of that type, for instance the number of shared global variables for R_4 coupling. The visualisation of the module coupling may be of as much, or even more, use than any measures on coupling because it can allow developers to gain a high level overview of the interactions in the software system very quickly.



Where (x,y) shows y occurrences of coupling type R_x .

Figure 4: Example coupling graph.

Information flow

Generally a modular program will consist of modules with data flowing through them in a manner that is termed *information flow*. Measuring properties of information flow may be of interest to programmers because it can help to indicate where particularly complex interaction is occurring between functions, and thus where there may be complex algorithmic behaviour that is likely to make modifications to the program difficult to perform.

Flowgraphs can be used to model the information flow between modules. Fenton views information flow as one of several different types, described below.

- *Local Flow*. A module invokes a second module and passes information to the invoked module, or an invoked module returns a result to the caller.
- *Local Indirect Flow*. Occurs if an invoked module returns information which is subsequently passed to a second invoked module.
- *Global Flow*. Occurs if there is a flow of information from one module to another via a global data structure.

Using the information flowgraph of a program it is possible to measure many attributes that help to quantify the complexity of the program structure. One such complexity measure is the *Henry-Kafura* [45] measure. This measure uses the following two measures as its basis.

- *Fan-In*. The number of local flows that terminate at a module M .

- *Fan-Out*. The number of local flows which emanate from a module M .

Information from such attributes may help to identify modules that contribute to complex information flow, and that therefore make testing and modifications more difficult to perform.

The Henry-Kafura measure, shown in Equation 1, uses these attributes along, with a measure of the module size, to indicate the complexity of a module. The $fan-in(M) \times fan-out(M)$ component of the metric indicates the number of possible connections between an input source and an output destination, and is squared in the belief that the complexity in terms of the number of connections is more than linear. The assertion that complexity is non-linear is also supported by Brooks [17].

However the use of the Henry-Kafura measure to indicate complexity is a little questionable because a module with fan-out or fan-in of zero will have a complexity measure of zero. Henry and Kafura justify this notion by claiming that modules with zero fan-in or fan-out are in some sense isolated from the system as a whole and therefore have low complexity.

$$\text{complexity of module } M = length(M) \times (fan-in(M) \times fan-out(M))^2 \quad (1)$$

Henry and Kafura performed an industrial study comparing their measure with maintenance change data for the UNIX operating system. The study revealed a correlation between high measurement values and those modules known to be difficult to maintain. Shepperd [86] also performed a detailed study of the Henry-Kafura measure, comparing the Henry-Kafura measure with the development time of projects. Shepperd found that the Henry-Kafura measure did not have significant correlation with the development time. However Shepperd found that a hybrid measure, shown in Equation 2, did show a significant correlation with development time, although it is not clear why this should be so.

$$\text{complexity of module } M = (fan-in(M) \times fan-out(M))^2 \quad (2)$$

2.3 Metrics for Object-Oriented Languages

In previous sections a selection of useful metrics has been shown for imperative languages, and there are many more that have not been covered in this thesis due to the large body of work covering metrics for imperative languages. However in recent years Object-Oriented (OO) programming languages such as C++ and Java have become popular so it is interesting to examine the metrics available for such programming languages. Many of the metrics used for imperative languages can be applied to OO programs, but there are a number of additional attributes that are specific to OO programs which are therefore ignored by imperative metrics. Many metrics for use with OO programs are described by Lorenz and Kidd [62]. The remainder of this section is divided into the following parts.

- Section 2.3.1 discusses ways to classify the importance of classes.
- Section 2.3.2 presents work to measure properties of methods in an OO program.
- Section 2.3.3 examines classes and their properties.
- Section 2.3.4 looks at work that investigates the coupling between objects.
- Section 2.3.5 summarises this section.

2.3.1 Classifying classes

In OO programming the program is divided into classes which typically encapsulate both state and functionality. Lorenz and Kidd classify program classes into one of the following two categories.

- *Support Classes* are those classes which are not central to the task the program is to perform. It would be possible to develop a solution without them.

- *Key Classes* are those that are central to solving the problem, without which it would be impossible to develop a solution.

It is important not to confuse these two classifications with “application” and “utility” classes. Utility classes may be key classes, for instance a numerical library may contain utility classes which are key to solving the application problem. However, making this classification normally requires the insight of a program and would be difficult to perform accurately with an automated system.

With program classes classified into support and key categories it is possible to measure the ratio of support classes to key classes. This ratio gives an indication of whether the program is dominated by support vs key code. This may be useful to know because “show stopping” defects are more likely to occur in key classes than support classes, where defects may not be so serious.

2.3.2 Metrics for methods

Object-oriented languages use classes to encapsulate functionality and state. A class will normally have *attributes* (state) and *methods* (functionality) which operate on those attributes. Classes communicate by sending *messages*, which is achieved by invoking the methods of the target class.

Some metrics view methods simply as functions in an imperative language and in this way many of the metrics used for imperative languages can be used directly on methods in an OO program. For instance, method size may be measured by Lines Of Code, or the number of statements in the method body.

A more object-oriented view of method size, suggested by Lorenz and Kidd [62], may be to measure the number of message sends initiated by the method. The number of message sends gives a useful indication of how “busy” a method is, which might highlight methods with a high degree of coupling. This is similar to counting the number of function calls out of a particular function in an imperative language, and may be calculated either statically, by the number of lines which initiate message sends, or dynamically at runtime. Dynamic measures of coupling

between objects are discussed in greater detail in Section 2.3.4.

2.3.3 Class properties

Existing metrics for imperative languages are not necessarily best suited for measuring properties of classes as a whole. For instance, to measure the size of a class it would be possible to measure Lines Of Code but this is not necessarily a good indication of the size of a class. A better measurement might be to count the number of methods the class contains, which measures how large the interface to the class is, or how much state the class encapsulates, for instance by counting the number of attributes the class contains.

One of the most useful features of OO programming is *inheritance*. Inheritance allows a child class to inherit behaviour from a parent class, or *extend* the behaviour of the parent class. This feature adds genericity to OO languages, but can make the behaviour of a child class harder to understand because it may be necessary to understand the behaviour of the ancestor classes as well as that of the child class. Because inheritance is an important aspect of OO programming it is useful to have metrics to measure the impact of inheritance on program complexity.

Inheritance can be represented using directed graphs. From such graphs it is possible to measure attributes such as depth of inheritance and number of ancestors. It is worth noting that just representing the inheritance graph visually may be of significant help in understanding a system, as can be seen by the popularity of modelling tools such as UML, and can highlight places where the implementation of a software system does not match the original design. This is explored in greater detail in Section 2.6.

Along with the inheritance hierarchy, the effect of inheritance on a class is also of interest. When a class inherits functionality from a parent class it may also override methods. This information can be added to the flowgraph and allows information such as the number of inherited methods or the number of overridden methods to be counted. The number of overridden methods is a particularly

interesting metric because overriding methods can result in methods which are syntactically identical from the point of view of their type signature, but may have fundamentally different behaviour. This can make it very hard to correctly understand the behaviour of code that uses such overridden methods.

Because all these measures consist of counting occurrences of some property, they can be implemented quite simply as static measures, although it is also possible to perform these measurements dynamically, which is partly examined in Section 2.3.4. Additionally, many of these measures can be applied to design artifacts such as UML diagrams. This is discussed in Section 2.6.

2.3.4 Dynamic coupling measures

Software systems often need to be changed, e.g. for new or changing user requirements. It is therefore important that software systems are constructed in such a way that they may be changed with the minimum of disruption to existing code. One way to make the system easier to change is to ensure the amount of coupling between components is minimal. Because of this there have been many coupling metrics defined, such as those for imperative programs described in Section 2.2.5. These static measures of coupling have been shown to be useful indicators of software quality by the work summarised in Section 2.2.5. However properties of the dynamic behaviour of OO software can only be accurately gathered at runtime.

One of the advantages of measuring coupling dynamically instead of statically is that it allows the focus of attention to be limited to code used for a particular use case, which may be only a subset of the code contained in the classes associated with the use case. This is not possible with static measures which must consider the classes as a whole.

Arisholm [6] examines three different dimensions of dynamic collaboration between two components of a system, namely the direction of the coupling, the mapping of the coupling and the strength of the coupling. These are described in more detail below.

- *Direction*. One can measure the messages received by an entity, the *export coupling*, separately from the messages sent by an entity, the *import coupling*.
- *Mapping*. Messages are “received” through methods defined either within an object’s class or inherited from its parent classes. Because of this, although messages are mapped to a single *object*, they may be mapped to many *classes*. From this a distinction can be drawn between the objects sending and receiving the messages and the classes that implement the methods. This leads to two categories of coupling which can be measured separately, *object-level* coupling and *class-level* coupling.
- *Strength*. The strength of the coupling is a measure of the amount of association between the two entities. Arisholm uses three methods of measuring this strength:
 - *Number of dynamic messages*. A count of the total number of messages sent by an entity.
 - *Number of distinct method invocations*. The number of unique messages sent by an entity, ignoring duplicate messages.
 - *Number of distinct classes*. The number of unique classes to which the entity sends messages. Different messages sent to the same class are treated as a single message.

Unlike the classification of coupling described earlier in Section 2.2.5, these strength measures do not attempt to classify the *type* of the coupling, e.g. whether the messages contain some control element, but instead only count the number of occurrences, and so treat all types of coupling as equal.

From these three dimensions of coupling Arisholm developed a collection of twelve metrics. The preliminary results from his work suggested that a number of the metrics showed a positive correlation with the *change proneness* of a case study program. Further, the results suggested that the higher the object-level

export coupling of a class, the more objects depend upon that object and hence the class of which that object is an instance is more likely to be changed.

Dynamic measures of software open up many possibilities for measurement, however this thesis concentrates on static measures of software and the dynamic measures are presented here only for information.

2.3.5 Summary

In general, many of the imperative language metrics can be used for OO programs. Additional metrics can be used to measure attributes of the additional features of OO languages, such as inheritance and coupling. However some of these OO features must be measured dynamically at runtime rather than statically in order to gain accurate measurements.

2.4 Software Measurement for Functional Programming Languages

While there is much work examining the use of software measurement for imperative and object oriented languages, there appears to be little work exploring the use of software measurement in functional programming. One of the few such works is the PhD thesis of Van den Berg [96]. This thesis describes how functional programming using the language Miranda¹ is used for teaching purposes in the first year of the computer science course at the University of Twente. The introduction of functional programming into the curriculum provided the initial motivation for the thesis, namely to answer the question “Do students produce better programs when they learn functional programming instead of imperative programming?”. This question raises two issues, “Which criteria can be used to objectively assess the quality of programs?” and “How to compare quality aspects of programs written in different programming paradigms?”.

¹Miranda is a trademark of Research Software Ltd.

The rest of this section addresses these questions and is divided into the following subsections.

- Section 2.4.1 examines the readability of programs.
- Section 2.4.2 describes the methods Van den Berg used to validate his metrics.
- Section 2.4.3 summarises the conclusions Van den Berg drew from this work.

2.4.1 Readability

An important aspect of producing “a better program” is readability. Van den Berg used metrics such as program volume from Halstead’s Software Science and McCabe’s cyclomatic complexity metric to assess the readability of a program. Van den Berg carried out an experiment in which several students performed modifications to existing programs, some written in Miranda and others in Pascal, which were then ranked in order of readability by a group of experts. The metrics were then applied to the programs and the correlation was measured between the ordering of the metric results and the ordering from the expert’s opinions.

This case study showed that while there was a reasonably high correlation between the metrics and the experts view for imperative programs (Pascal), the correlation was significantly lower for functional programs (Miranda). Van den Berg found that for functional programs, the degree of agreement between the experts on the readability of the programs was low and cited this as a possible causes of the low correlation.

2.4.2 Validation

A common criticism of software measurement in the past has been the lack of rigorous validation. Van den Berg presents a case study demonstrating the experimental validation process he used. For the case study he used structure metrics

for Miranda type expressions. The structure of Miranda type expressions was represented by the grammar :

```
typeexp ::= Num | Bool | Char |
          Var num | L typeexp | T [typeexp] |
          F typeexp typeexp
```

With this grammar a large class of Miranda types, excluding algebraic and abstract types, can be represented. For instance the type :

```
(* -> bool) -> [*] -> ([*], [*])
```

would be represented in the grammar as :

```
(F (F (Var 1) Bool)
  (F (L (Var 1))
    T [(L (Var 1)), (L (Var 1))]))
```

From the grammar for type expressions Van den Berg derived the following set of axioms that a structure metric on types must fulfil.

$$m(L t) > m(t) \tag{3}$$

$$m(T [t_1, \dots, t_n]) > \max(m(t_1), \dots, m(t_n)) \tag{4}$$

$$m(T [t_1, \dots, t_n]) = m(T (\text{perm}[t_1, \dots, t_n])) \tag{5}$$

$$m(F t_1 t_2) = m(F t_2 t_1) \tag{6}$$

$$m(T [t_1, \dots, t_{n+1}]) > m(T [t_1, \dots, t_n]) \tag{7}$$

$$m(T [t_1, \dots, t_n]) > m(L t_i), i = 1, \dots, n \tag{8}$$

$$m(F t_1 t_2) > m(T [t_2, t_1]) \tag{9}$$

$$\text{where } n \geq 1$$

Axiom 3 states that the metric value for a list of elements of type t should be higher than for a single element of that type, and thus makes the assumption that lists add to the complexity of a type expression.

Axiom 4 states that the metric value for a tuple should be a greater value than any of the metric values for the types contained in the tuple. This makes the assumption that a tuple adds to the complexity of a group of types.

Axiom 5 states that the metric values for a tuple should not be affected by the order of the elements of a tuple.

Axiom 6 states that the metric values for a function type should not be affected by the ordering of the element types in the function type.

Axiom 7 states that the metric value for a tuple should increase as the number of elements in the tuple increases.

Axiom 8 states that the complexity of a tuple, e.g. `(Bool,Char)`, is greater than the complexity of a list of any of the component types, e.g. `[Bool]` and `[Char]`. The reason for this assertion is that to understand the tuple it is necessary to understand two types and the tuple constructor, while to understand a list it is only necessary to understand one type and the list constructor.

Axiom 9 states that the metric value for a function type should be greater than the metric value for a tuple type containing the same element types.

For the experiment a simple sum metric that conforms to the above axioms was defined in the following manner:

$$\begin{aligned}
 m(Num) &= C_N \\
 m(Char) &= C_C \\
 m(Bool) &= C_B \\
 m(Var\ n) &= C_{V(n)} \\
 m(L\ t) &= C_L + m(t) \\
 m(F\ t_1 t_2) &= C_F + m(t_1) + m(t_2) \\
 m(T\ [t_1, \dots, t_n]) &= C_T + m(t_1) + \dots + m(t_n)
 \end{aligned}$$

	Axiom	$t_{LHS}(\text{sec})$	$t_{RHS}(\text{sec})$
Eqn. 3	LHS > RHS	19.0	08.0
Eqn. 4	LHS > RHS	21.6	10.6
Eqn. 5	LHS = RHS	33.8	29.7
Eqn. 6	LHS = RHS	15.2	20.7
Eqn. 7	LHS > RHS	25.6	20.5
Eqn. 8	LHS > RHS	24.6	12.7
Eqn. 9	LHS > RHS	19.7	12.7

Table 1: Results from the validation of metrics for Miranda type expressions.

The experiment consisted of presenting a type expression to a subject who was then requested to produce a function with that type. The function need not produce any sensible output, merely conform to the type signature. The time was measured from the instant the subject was shown the type expression until the instant they completed the task. The subjects were 16 first year undergraduate students, each answering 40 questions. Each subject was shown the same questions, but in a random order. Showing individual type expressions to the subjects in a random order avoids the results being biased by a “learning effect”, whereby answering one question trains the subject for a later question, resulting in reduced time to answer the later question.

The type expressions used in the experiment were devised so that they would fit the axioms described above. For instance, the two type expressions `[char -> bool]` and `char -> bool` could be used to test Axiom 3.

Only those type expressions that were correctly answered were considered and those with extreme time values² were discarded. The average times taken to correctly complete the type expression questions were then used to test the axioms. The results from this experiment are shown in Table 1.

These timing results were then used to calculate coefficients, e.g. C_N , to be inserted in the metric described above. This resulted in a metric that can be used to assess the complexity of type expressions in Miranda programs.

²Those that differ by more than three times the standard deviation from the arithmetic mean.

Program structure is often thought to be an important aspect of good program construction. Van den Berg derived metrics for program structure using control flowgraphs. He then performed an experiment to determine programmers performance on structured versus unstructured function definitions of varying sizes.

The notion of structured and unstructured used in this work was based on Fenton's [32] control flow work. Van den Berg classifies flowgraphs for Miranda as structured or unstructured by the paths through the flowgraphs. A path through a flowgraph is a sequence of consecutive nodes from the start node to the stop node. Van den Berg defines a *D-structured* path as a sequence of pattern matching nodes followed by a sequence of guard nodes, and possibly an expression node and finally a stop node. He further defines a path that is not D-structured as *X-structured*. He then classifies a function as structured if all paths through its flowgraph are D-structured, otherwise the function is classified as unstructured.

The experiment was performed in a similar manner to the experiment described earlier for type expressions. The following conclusions were drawn from the experiment.

1. Subjects need significantly less time to obtain an answer to structured functional definitions than to unstructured functional definitions.
2. Subjects give correct answers to somewhat larger structured functional definitions significantly more often than they do to unstructured definitions of comparable size.
3. Subjects need significantly more time to obtain an answer to larger function definitions than to smaller ones.
4. Subjects give correct answers for larger structured function definitions significantly more often than they do for smaller ones.

The most interesting conclusion here is 4, which appears to show that subjects in the experiment were more careful in their answers to larger problems than they were for their answers to smaller problems. This is also suggested by conclusion

3 which shows that the subjects spent less time answering the smaller problems than they did for the larger problems.

2.4.3 Summary

Van den Berg concluded that it was not possible to make a general conclusion to the question ‘Do students who learn functional programming write better programs?’ on objective grounds, particularly as there was little agreement between experts on what constituted a readable Miranda program. However he did note that students who learnt functional programming tended to use more functions in imperative languages than those students who had not. No other work relating software measurement to functional programming is known to the author.

In general there has been little activity in the field of software engineering for functional programming. There has been a little work on design paradigms for functional programming by Russell [84] and Wakeling [99], but the majority of the software engineering research has been focused on tool support, such as support for Haskell in integrated development environments such as Eclipse [38] and Visual Studio [65].

By far the largest body of work in the area of software engineering for functional programming has been in the study of debugging and tracing tools, which are typified by work such as that of Runciman and his co-workers [20] on tracing program execution and that of Reinke [81] on visualising and animating such traces.

Most recently, work has also been started by Li, Reinke and Thompson [60] in the area of tool support for refactoring functional programs. The application of software metrics to refactoring is examined in Section 2.7.

2.5 Metrics and Time

Large software projects are often developed and maintained over many years. As such software ages it can become difficult to maintain, develop, and debug. The

history of the development of such a system, particularly the change history as might be contained in a revision control system such as CVS [35], can provide interesting data about the program which may be used with metrics in several ways. The remainder of this section is divided into the following parts.

- Section 2.5.1 examines work that measures attributes of the change history of a program.
- Section 2.5.2 shows how the change history of a program may be used to validate a metric.
- Section 2.5.3 summarises this section.

2.5.1 Time as a metric

In [40] Graves and his co-workers explored the effect of using the history of a program's development to predict where a program is likely to become unmanageable. They found that the change history contained more useful information than could be obtain from a single snapshot of the program. For their work they compared the following measures:

- *Number of Past Faults.* This method predicts the number of faults to be found in a module in the future using a constant multiple of the number of faults found over a past period of time. This provided a reasonably accurate prediction of the number of faults and proved to be difficult to improve upon.
- *Number of Deltas.* Using the number of changes to a module over its entire history provided a better prediction of the number of faults to be found than those measures that were generated from a single snapshot of the program such as lines of code. This measure may be related to the *Number of Past Faults* metric because there is likely to be changes, or *deltas*, after a fault has been discovered. Likewise, if there are a large number of deltas then there may be an increased probability of faults being introduced.

- *Average Code Age.* Combining the average age of code within a module with the number of deltas produced a measure which increased the accuracy of the number of deltas method.
- *Weighted Time Damp.* This method computes the fault potential of a module by adding contributions from each change made to a module such that the larger or more recent a change is, the greater the contribution, with recent large changes contributing the most. This method also incorporates a damping mechanism to avoid transient events such as a single large change from skewing the result.

Of all these methods, the weighted time damp metric provided the most accurate prediction of the number of faults likely to occur in the future. For their experiment a 1.5 million line subsystem of a telephone switching system was used. The metrics described above, along with other complexity and size metrics such as Lines Of Code, were used to predict the number of faults in the system and these results were compared with the actual fault occurrences. The Number of Past Faults metric was used as a benchmark against which the other methods could be compared. From these experiments they found that the change history provides much more useful and accurate predictions than simple metrics that are applied to a single snapshot of the program. Particularly good correlation with the number of faults was obtained when the data from the change history is combined with “snapshot” metrics such as Lines Of Code. An example of such a metric is the Weighted Time Damp measure.

2.5.2 Using time to validate metrics

Barnes and Hopkins [10] describe how they applied simple metrics such as pathcount to a software library written in Fortran and compared the results with the bug fixing changes appearing in the change history of the library. They found that there was a high correlation between routines which required post release maintenance and routines which exhibited a high pathcount value in excess of 10^5 . Their

results showed that 41% of all the bug fixes occurred in routines with a pathcount value in excess of 10^5 , while those routines accounted for only 16% of the total number of routines in the library. They therefore calculated that routines with a pathcount value greater than 10^5 were six times more likely to contain a bug than routines with a pathcount value of less than 10^5 .

One hypothesis that could account for such a result is that if bugs are distributed randomly throughout the program code, and that routines with larger pathcount values also have a larger number of lines of code, then those larger routines would be statistically more likely to contain bugs.

This hypothesis depends on there being a correlation between the size of a routine and its pathcount. To perform a quick test to see if there is a correlation, we plotted a graph of pathcount values against routine size, which is shown in Figure 5.

This graph shows that there is a trend for the pathcount values to increase with the routine size. Further analysis showed a statistically significant correlation of 0.4334 between the pathcount values and the size of a routine, measured in lines of code. However, this correlation is quite low, so it is still not clear if Barnes and Hopkins results could be caused by random placement of bugs. When considering the pathcount metric in more detail it seems clear that as a pathcount value increases, the number of lines of code must also increase because there is a limit to the number of execution paths that may be present in a given number of lines of code. It therefore seems likely that the correlation between function size and pathcount is caused by this relationship.

Because of this observation it is unclear if Barnes and Hopkins could have achieved similar results by using function size rather than pathcount values. One reason why pathcount may be a more discriminating predictor of faults than function size may be that pathcount measures cover a significantly larger range of values than the function size metric, and may therefore have a finer granularity.

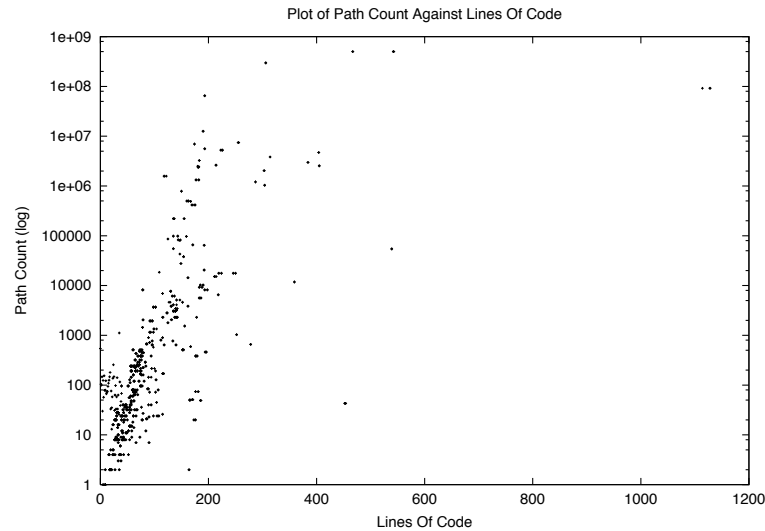


Figure 5: Plot of path count values against code size in Lines Of Code. Path count plotted on logarithmic scale.

2.5.3 Summary

Barnes and Hopkins provide a scientific approach to the validation of metrics, something that has sometimes been lacking in the field. Their method of using the change history of a library to allow validation of predictions based on metric values is an innovative use of the change history.

This section has also shown that the change history contains much useful information about the state of a software system and that combining metrics with software change history can make software measurement a more powerful tool.

Using change history as part of the measurement process may be relatively simple for many large software systems because such systems often employ a source code revision control system such as CVS, from which it is sometimes possible to extract information automatically. However it is worth noting that this can depend heavily upon the frequency of commits to the revision control system and the quality of the log messages associated with them. This is discussed more in Chapter 3.

2.6 Measuring Software Design

Software systems are growing increasingly complex so it is essential that software implementations are based on a good software design. This is highlighted in the work of Mens and his co-workers [68] which also notes that improved software development methodologies do not solve the problems of constructing complex software because such methodologies are usually used to implement more features within the same time frame, and so the overall complexity of the systems is not reduced.

Because of the importance of having a good design from which to build software systems, some researchers have begun to study how software measurement ideas can be applied to artifacts of the design process, such as UML diagrams, in order to assess the quality of a design. Design artifacts can be thought of as a higher-level source code, and artifacts such as class diagrams often employ graph abstractions that are commonly used in software metrics, so in many ways this is a natural progression for software measurement. The rest of this section is divided into the following parts.

- Section 2.6.1 examines measures of testability that can be applied to class diagrams.
- Section 2.6.2 examines ways metrics can be used to improve the interfaces provided by components of a system.
- Section 2.6.3 summarises this section.

2.6.1 Assessing testability from class diagrams

Testing of a software system can be an expensive and time consuming phase of the software development cycle, so it is important that the system being developed is as easy to test as possible. Ease of testability in a software system does not happen accidentally, instead it must be designed into the system from the beginning. The work presented by Baudry and his co-workers [12] aims to address the issue

of designing programs that are easier to test by applying ideas from software measurement to design artifacts such as UML class diagrams.

Class diagrams show the interdependencies or *relationships* between objects in the system. These relationships may be either *direct* or *transitive*. Direct relationships between objects A and B occur when the two objects directly interact with each other, while transitive relationships occur when the two objects interact via a whole sequence of direct relationships involving other objects.

However, it is important to realise that class diagrams can present only a partial view of the program. There is a distinction to make between *class interaction*, represented in the class diagrams, and *object interaction* at runtime. Class interactions present in the class diagram are only potential interactions, while object interactions are actual interactions between objects.

Using a class dependency graph as a model of the class interactions, it is possible to detect interactions of classes that may not be obvious from the class diagram. Additionally, the class dependency graph allows several metrics to be defined which quantify the complexity of the class interactions. Increased complexity in the interactions usually indicates increased difficulty in testing.

Having a means to measure the testing complexity of a design artifact such as UML class diagrams allows system architects to compare evolutions of the system design to establish the effects of changes on the testability. Such measures also allow designers to focus their efforts in areas where testing might be difficult, perhaps making more cost effective use of their time and reducing the cost of the testing phases of the development cycle.

2.6.2 Factorisation and metrics

Object-oriented programming encapsulates functionality in components called classes. Amongst the design issues inherent in constructing component based software is the problem of ensuring components offer good interfaces for collaboration, encapsulating neither too much nor too little functionality.

The issue of determining the best generalisation and specialisation relationships between components, and the factorisation that occurs when component based systems are designed, is considered by Dao and his co-workers in [24]. Their work presents a theoretical framework in which to assess the quality of a component hierarchy and derives metrics that measure aspects of the factorisation of such component based systems.

Although their framework is not detailed in this thesis, it is worth noting that their work may be useful in refactoring where functionality might be moved between classes to improve the public interface of components, and thus reduce the complexity of the system as a whole. However, it is important to ensure that moving items between classes does not adversely affect the readability of the program, which is an issue discussed by Moore [70, 71].

2.6.3 Summary

Creating a good design is an important part of the software development process. As such, artifacts of the design process such as UML diagrams should be considered as part of the program source, and therefore legitimate targets for software measurement. This has only recently begun to be recognised, and this section has presented some work that utilises this idea.

2.7 Metrics for Re-engineering

An intrinsic property of real-world software is that it evolves when new requirements are discovered and extra features are added. Such systems tend to become increasingly complex as they evolve and often drift from their original design, becoming cumbersome to maintain and difficult to extend.

The process of *refactoring* claims to address these issues by incrementally improving the internal structure of the systems while preserving the external behaviour of the system.

A good survey of the field of refactoring is provided Mens and his co-workers

in [68] and [69]. These papers raise several research questions, one of which is: “How can we determine where and why refactorings should be applied?” This seems an ideal application for software measurement.

The remainder of this section consists of the following parts.

- Section 2.7.1 shows how cohesion metrics can be used to suggest places where refactorings might be applied.
- Section 2.7.2 describes the use of metrics for determining where refactorings have been applied between versions of software.
- Section 2.7.3 summarises the section.

2.7.1 Cohesion metrics for refactoring

Cohesion metrics measure the degree to which separate parts of a software system, such as classes in an OO program, interact with each other. If two classes interact heavily they probably “belong together”. This can give a strong indication that the two classes might benefit from moving some items, such as attributes or methods, between the classes to produce a cleaner system design and implementation.

This use of metrics is demonstrated by Simon and his co-workers in [89] which presents four refactorings used in OO software and describes the symptoms, or *bad smells*, that indicate that the refactorings should be applied. The refactorings used in this work were:

- *Move method*. Move a method from class *A* to class *B*. Typically done when the method uses, or is used by, more features of *B* than *A*.
- *Move attribute*. Move an attribute from class *A* to class *B*. Typically done when the attribute is used more by *B* than *A*.
- *Extract class*. Extract functionality from a class and form a new class. This is done when a class performs two or more distinct functions in the program which should each be provided by separate classes.

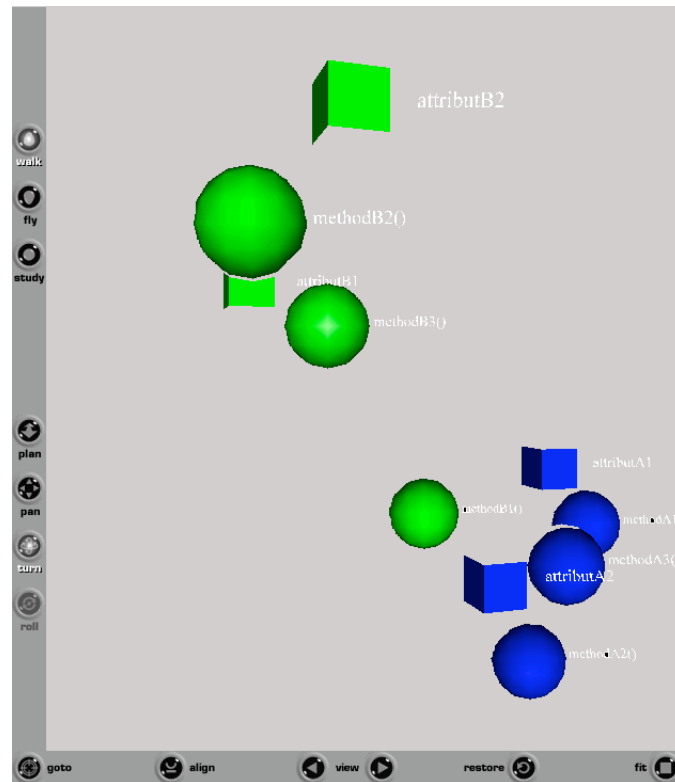


Figure 6: An illustration of visualising distance measures by Simon, Steinbrücknet and Lewerentz.

- *Inline class*. Merge a class A into a class B . Typically applied when a class is too small and does not contain much functionality, or when two classes are intimately coupled.

Cohesion measures were used to measure “distance” between items, however rather than simply display these distances as a list, Simon and his co-workers produced a VRML [43] model to display this information graphically.

This graphical display, which is illustrated in Figure 6 and described in detail in Chapter 6.4.1, shows elements such as attributes and methods in a three dimensional space. The class to which elements belong is shown by colour coding the elements. As a result of using the cohesion metrics to measure the distances between elements, elements that are strongly associated with each other will be clustered in the visualisation.

Ideally clusters will contain elements of the same colour, indicating that the strongly associated elements are packaged in a single class. However, if one finds a cluster that contains a small number of elements which are coloured differently from the majority of the elements in that cluster, it is a good indication that those elements might be suitable for a move refactoring.

Visualisation systems such as this offer a very clear view of the cohesion of the design. However, its usability is limited to small numbers of classes and elements and is not scalable because as the number of elements and classes increase the view can very quickly become messy and unintelligible.

2.7.2 Finding refactorings via change metrics

Reverse engineering is the process of determining the design from a functioning software system. Reverse engineering may be an integral part of the software development process because the implementation of the system may drift from the design, and it is important to understand why such drifts occur. One reason for an implementation to drift from its design is that various refactorings have been applied, such as those that move items between classes. However it is not always clear where such refactorings have been applied, so it would be useful to have a method of detecting where this has occurred.

One way to detect where refactorings have been applied is to use metrics. This is examined by Demeyer and his co-workers [26]. Their system uses a combination of change metrics on past versions of the software, and heuristics to identify parts of the system where refactorings are likely to have occurred and, for a small subset of refactorings, which refactorings have been applied. The refactorings that this system detects are described below.

- *Split Into or Merge With Superclass*. This refactoring either separates out some functionality of a class into a new superclass (Split Into Superclass), or merges a class into an existing superclass (Merge With Superclass). This is illustrated in Figure 7.

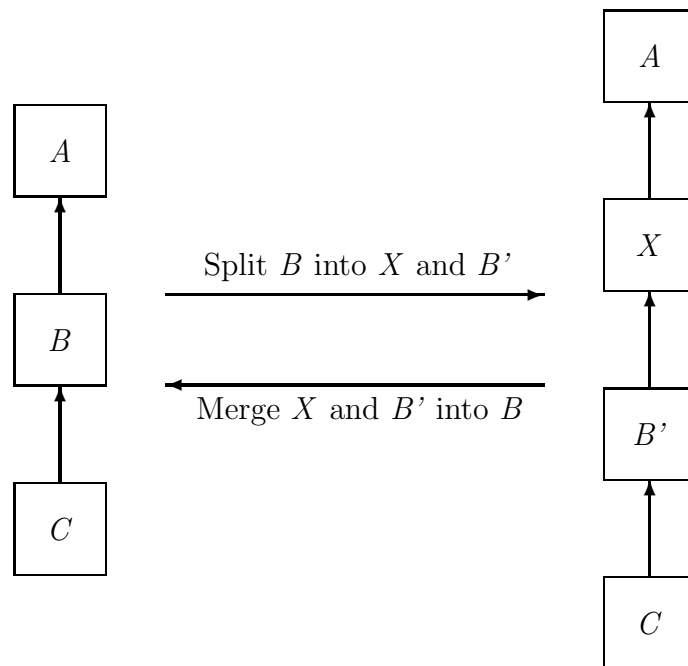


Figure 7: An example of the Split Into or Merge With Superclass refactoring.

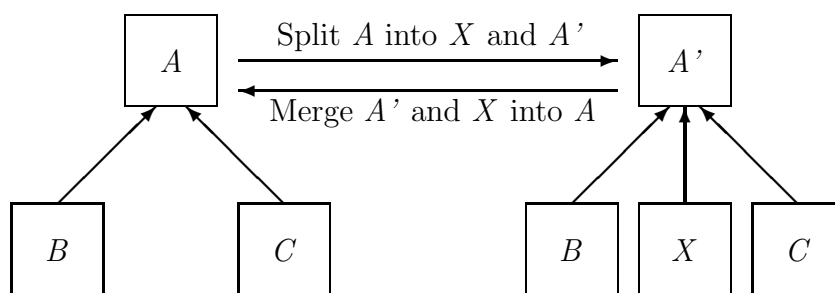


Figure 8: An example of the Split Into or Merge With Subclass refactoring.

- *Split Into or Merge With Subclass.* This refactoring either separates functionality from a class into a new subclass (Split Into Subclass) or merges a class into an existing subclass (Merge With Subclass). At a first glance this seems to be the same as the previous refactoring, “Split Into or Merge With Superclass”, but the two refactorings are subtly different in the way they affect the inheritance hierarchy. The “Split Into Superclass” refactoring creates a new superclass, while the “Split Into Subclass” creates a new subclass. This is illustrated in Figure 8.
- *Move to Other Class.* This refactoring moves an item such as a method from one class to another.
- *Split Method or Factor Out Common Functionality.* This refactoring will split a method into two separate methods (Split Method) or merge common functionality from two separate methods into a single method.

Change metrics measure attributes of changes between different versions of a program. This work considered the following three properties of the software.

- *Method Size.* Measuring changes in method size may indicate the splitting or merging methods. This work used three metrics to measure method size, “Number of message sends in method body”, “Number of statements in method body”, and “Lines of code in method body”. These metrics were discussed in Section 2.3.2.
- *Class Size.* Changes in the size of a class may indicate a shift of functionality between classes. The class size was measured using the “Number of methods”, “Number of instance variables” and “Number of class variables” metrics. These metrics were examined in Section 2.3.3.
- *Inheritance.* Changes to the class inheritance is a symptom of refactorings that optimise the class hierarchy, such as “Split Into Subclass”. Inheritance

was measured using “Hierarchy nesting level”, “Number of immediate children of a class”, “Number of inherited methods”, and “Number of overridden methods”.

To validate the metric and heuristic combination a series of three case studies were chosen. The case studies were select for the following reasons:

- *Accessible.* Source code of the case studies is publicly accessible, allowing others to reproduce the results.
- *Representative.* Each case study is a successful system that has undergone successive refactorings.
- *Independent.* Each case study was developed independently, which means the development process used for the programs should not influence the results.
- *Documented.* The features that were changed between different version of the case study programs are documented.

The case studies showed that the combination of using change metrics and heuristics generally focused on a small subset of the program code and reliably indicated the refactorings. This allows a reverse engineer to examine only the highlighted code, significantly reducing the effort of code inspection. It is important to note that this system only identifies where refactorings have *already* occurred and does not attempt to indicate where refactorings might take place in the future.

However their system does suffer from some drawbacks. The system uses names for matching items so it is vulnerable to items being renamed, which the system sees as separate removal and addition of items. This can lead to confusing results, although a more sophisticated system could handle renaming better. More problematic is that the system becomes less precise as the number of changes between versions increases. One way to solve this is to ensure only a small number

of changes occur between each version, for instance by committing changes to a CVS repository frequently. This is difficult to do after the fact so it is important to ensure this is done from the start of the development cycle. This is a recurring need when attempting to performing empirical studies of software measurement techniques.

2.7.3 Summary

This section has described the problems associated with software systems as they evolve, and has presented refactoring as a method to control this evolution. Further, it has shown work that attempts to use cohesion metrics which measure the extent to which items in a program “belong together” to decide where refactorings might be applied.

This section has also shown that at times it is useful to determine where refactorings have been applied in the past, in order to determine how and where a program has drifted from its design. Work has been presented that uses change metrics to determine where refactorings have been applied between two versions of a program.

2.8 Generic and Miscellaneous Metrics

In the previous sections a range of software measurement techniques have been discussed. This section presents a few other interesting measurement systems which do not easily fit into the classifications of those already described.

2.8.1 Predicting software scalability

Software measurement is typically used to highlight sections of program code that may cause problems during the software development cycle, for instance by increasing the difficulty of testing or decreasing the maintainability. However, it may also be of interest to measure other aspects of software systems such as

performance criteria. This issue is addressed by Weyuker and her co-workers in [100], which presents a *Performance Non-Scalability Likelihood* metric which can be used to predict the scalability of a software systems.

The Performance Non-Scalability Likelihood, or *PNL*, metric provides a probabilistic prediction of the likelihood of a software system meeting some chosen performance criteria under a specified workload. In order to apply the PNL metric it is necessary to first produce a simplified model of the software system by identifying the key operations, or *state variables*, of the system that constitute the *state* of the software system to be measured. The PNL metric analysis is performed on this simplified model.

To apply the PNL metric to the system model one must first decide upon a performance objective. An example of a performance objective might be a maximum response time that would be deemed acceptable, or the minimum number of transactions per second.

Next it is necessary to collect field usage data which describes how the system behaves in the real world for some observed *real* workload. This is called an *operational distribution* or an *operational profile* and is a probability distribution describing how the software system being studied is actually used during its operation.

Finally, the PNL metric can be computed for the system model using the operational distribution. The result from this analysis is a probability distribution describing the likelihood of the performance criteria being met for a given workload.

Being able to estimate when a workload will be too high for the performance criteria to be met can be extremely useful because it can be used to plan when system upgrades may be necessary, or may indicate that algorithms used by the software system might need improving.

However computing the PNL metric is not straightforward because the metric is not “fully automated”. It requires that a model of the system is provided which is likely to require manual analysis of the software system. More problematic is

the necessity to collect field data. This may be difficult for large systems or potentially impossible, for instance if the software system is not yet fully constructed. Collecting a suitable amount of field data may incur significant costs although if the process is included as an intrinsic part of the construction of the software system these costs may be modest, while providing the potential to avoid costly performance issues that may appear at a later date.

2.8.2 Measuring system maintainability

Studies have demonstrated that software's characteristics, history and associated environments are all useful in measuring the quality and maintainability of software systems. Consequently many metrics have been designed to assess the maintainability of a system. One such metric is HPMAS.

HPMAS, described by Coleman and his co-workers in [22], is Hewlett Packard's Hierarchical Multidimensional Assessment model, a software maintainability assessment system which examines three underlying attributes or *dimensions* of a system:

- The control structure.
- The information structure such as the data structures and information flow.
- Typography of the source code, such as the naming of identifiers and commenting.

Metrics are defined to measure the characteristics of each of the dimensions and an *index of maintainability* is derived for each dimension. The three dimension scores are then combined to form the *total maintainability index* for the system as a whole.

HPMAS uses a weighted deviation system such that metric values outside a specified optimum range are penalised further, helping to highlight problem areas in the system.

Validation of the HPMAS model was achieved by applying the model to large industrial systems provided by HP.

HPMAS can be used for assessing the maintainability of software systems and allows investigation of the effects of changes to the software on its maintainability. It is particularly useful for gauging the effectiveness of changes intended to enhance the maintainability, where it may be useful to know the cost effectiveness of the effort involved in such changes. In some ways HPMAS can be thought of as an “applied” metric because it is a hybrid metric that is the result of applying atomic metrics, such as those described elsewhere in this chapter, to real systems in long term studies.

2.8.3 Using information theory for metrics

A common abstraction used when dealing with software systems is a graph composed of nodes and edges in which nodes represent components of the system while edges represent interactions between those components.

As has been shown in previous sections, many software metrics such as size, length, complexity, coupling and cohesion use this abstraction when performing their measurements. These traditional measures generally count features as if they were all equal in the programmers mind, however information theory suggests that repetitive patterns have low information content, and therefore have a low cognitive load for the programmer. This suggests that using information theory as a foundation for these metrics may yield more accurate results than the traditional metrics.

Allen [4] suggests that mistakes in software design and implementation may be caused by cognitive overload, which occurs when the amount of information that the programmer must comprehend in order to make a decision is too large. Allen presents versions of the size, length, complexity, coupling and cohesion metrics that use information theory as a foundation. These metrics all measure the information content of their particular attributes of a graph structure in bits, except for the cohesion metric which provides a ratio of bits.

Although these are interesting theoretical measures, Allen presents no evidence of their use on actual programs and does not provide any validation of these metrics. Instead validation of the usefulness of these measures in practice is left to others.

2.9 Summary

This chapter has shown how, despite a somewhat turbulent start, software measurement has grown to become a recognised part of the software development process.

In the preceding sections it has been seen that there are many attributes that one might choose to measure from software systems, whatever the programming paradigm they are written in, or even from the design artifacts associated with the software. These metrics, which are summarised in Table 2, can help assess the quality of the software, can indicate parts of the program where testing will be difficult, or may indicate where one may wish to apply refactorings.

Metrics are not limited to just measuring attributes of the source code of a program, but may even be used to predict how well a system will scale. Such information is invaluable for planning when systems will need to be upgraded, or for otherwise addressing performance concerns.

As can be seen from Table 2, almost all the metrics can be applied to the source code of a program in an automated manner. The exception to this is the *Performance Non-Scalability Likelihood* (PNL) metric, which is applied to a model of the program. The construction of a model is typically a manual task. The metrics such as the *Number of Past Faults* and the *Weighted Time Damp* require a change history for the program being measured as well as the source code of the program. Typically this would be provided by a revision control system such as CVS, which would allow the metrics to be applied without manual intervention.

A number of the metrics, such as the *Callgraph Size* or the *Depth of Inheritance*, can be applied not only to the source code of a program, but also to the

design documents such as UML diagrams. This could prove particularly useful for assessing the architectural complexity of a program during the design phase of the development cycle, when changes to the design can be performed with relatively little cost.

One of the important considerations when designing software programs is to minimise the coupling and dependencies between the various components of the program. Experience has shown that minimising coupling makes software easier to test and maintain. It is not surprising then that many of the metrics presented in this chapter attempt to measure the number of dependencies and the degree of coupling between components. This can be seen in Table 2, which shows that nearly a third of the metrics are performing such measures.

From all this work, and the continuing work in the field, it seems clear that metrics will play an increasingly pivotal role in software development. However, one aspect where research into software measurement is lacking is in their application to functional programming. Functional programming is something of a niche in the world of software development, but is gradually emerging from the shadows.

With the emergence of functional programming as a technology comes a necessity for more rigorous development methods. As was highlighted earlier in this chapter, there is no widely used formalised design process for functional programming as there is, for instance, with UML for OO programming. However there is an awareness among functional programmers that techniques such as refactoring fit very nicely into a functional context. Because of this there is a need for tools to quantify functional programs and indicate situations where refactorings could result in improvements to the source code.

This is an ideal application for software measurement, and so the following chapters will study the use of software measurement in the functional programming language Haskell.

Metric	Applied To	Used to measure or predict
	System Model	
	Change History	
	Design Documents	
	Execution Traces	
	Source Code	
	Performance Criteria	
	Maintainability	
	Documentation	
	Encapsulation	
	Dependencies	
	Coupling	
	Undiscovered Defects	
	Test Coverage	
	Testing Effort	
	Complexity	
	Code Size	
LOC, NCLOC	✓	✓
CLOC	✓	
Program Volume	✓	
Depth of nesting	✓	
Cyclomatic complexity	✓	
Pathcount	✓	
Test Effectiveness Ratio	✓	
Callgraph size	✓	
Callgraph Depth	✓	
Callgraph Width	✓	
Callgraph Arc-to-node ratio	✓	
Coupling-model graph	✓	
Henry-Kafura	✓	
Number of message sends in a method	✓	
Number of methods in a class	✓	
Number of attributes in a class	✓	
Depth of inheritance	✓	
Number of ancestors	✓	
Number of inherited methods	✓	
Number of overridden methods	✓	
Number of past faults	✓	
Number of deltas	✓	
Average code age	✓	
Weighted time damp	✓	
Performance Non-Scalability Likelihood	✓	
HPMAS	✓	

Table 2: A selection of metrics and their properties.

Chapter 3

Validation Methodology

The work in this thesis presents a number of software metrics for use with Haskell programs. Chapter 2 described how, historically, work on software metrics has often been lacking in rigorous validation. Because of this lack a significant portion of this thesis is devoted to the validation and analysis of the metrics presented.

Before examining the results of this analysis it is important to understand how the validation was performed. Therefore this chapter presents the methodology used to perform this validation, and also describes the design and implementation of the metrics.

To further aid in understanding what measurements are performed by the metrics, a brief overview of the Haskell language is also provided. The chapter is thus divided into the following parts.

- Section 3.1 provides a brief introduction to the Haskell language and its features.
- Section 3.2 describes how the metrics and visualisation systems presented in this thesis were designed and developed.
- Section 3.3 explains the experimental methodology used to validate the metrics presented in this thesis, the results of which are discussed in Chapter 4 and Chapter 5.

3.1 A Brief Introduction to Haskell

Haskell is a standardised general purpose lazy functional programming language, which features a sophisticated and powerful static polymorphic type system, higher-order functions, user defined algebraic data types, a monadic I/O system, and many other innovations. As such Haskell is the culmination of many years of research on non-strict functional languages.

In this section the Haskell language will be described in general, but it is not the intention to provide a comprehensive specification of the language. Such a specification is provided by the Haskell 98 Report[80].

The rest of this section is divided into the following parts.

- Section 3.1.1 describes the general differences between imperative languages and functional languages such as Haskell.
- Section 3.1.2 presents an overview of the syntax of a Haskell program.

3.1.1 Characteristics of Functional and Imperative Languages

The exact features that categorise a language as functional is open to debate, but the most commonly agreed feature is that functions are treated as first class values that can be passed around as function arguments, returned as results, and generally manipulated in the same way as any other built in values. Contrast this with imperative or Object-Oriented (OO) languages which only allow references to functions to be passed, for instance function pointers in C or indirectly as collections of functions encapsulated by a class which can be instantiated as objects in Java.

Because functional languages allow functions to be passed around as values, they also allow the construction of functions at runtime. For instance, by using lambda expressions in Haskell.

Typically functional languages also restrict functions to pure behaviour by disallowing side effecting actions, although in some languages such as SML and LISP this restriction is not complete. Imperative and OO languages incorporate side effects as fundamental language behaviour using features such as global variables and object attributes. In contrast, functional languages provide many ways to achieve similar behaviour without the need for side effects. In a functional language the implicit state modifications caused by side effects are rendered explicit, helping to reduce the chances of unforeseen or unpredictable behaviour.

By enforcing the use of pure functions the functional languages are not constrained to a particular execution order, and so are typically non-strict. In particular it is common to use lazy evaluation strategies in which values are evaluated “on demand”. Haskell uses a lazy evaluation scheme such that functions and parameters are evaluated only as far as is needed for the result. The use of lazy evaluation makes it possible to perform computation using infinite data structures. The use of pure functions can also allow the compiler to perform extra optimisation phases, such as inlining or memoization. Furthermore, pure functions can aid in formally reasoning about program behaviour.

Functional languages also typically include a powerful type system which supports safe reusability and abstraction through features such as user defined concrete (or algebraic) data types, abstract data types, higher order functions and polymorphism.

3.1.2 Haskell Syntax

The majority of this thesis uses Haskell as an example of a functional programming language. It is therefore useful to have a broad understanding of the syntax of Haskell.

At the top-level Haskell programs consist of a set of *modules*, each of which is typically stored in a separate file. Each module normally starts with an interface declaration, which lists the functions and data types which are *exported*, meaning they are accessible from outside the module. A module interface declaration for

a module `FooBar` might look like this.

```
module FooBar
  ( foo
  , bar
  ) where
```

This interface declaration *exports* two functions `foo` and `bar`. If a module does not include an interface declaration, like this:

```
module FooBar where
```

then everything contained within the module is externally accessible. A module may be *imported* to gain access to its exported functionality. This is illustrated in the example below, in which the `FooBar` module imports the `Foo` and `Bar` modules.

```
module FooBar where
```

```
import Foo
import Bar
```

Modules contain a selection of *declarations*. Declarations may be functions, data types and type classes. Only a subset of the various possible declarations and language syntax will be described in this section, and if further details are required the Haskell 98 Report[80] should be consulted.

Function declarations

Functions in Haskell can be thought of as consisting of two parts, a type declaration and a definition. The type declaration specifies the types of the parameters and the return type of the function, e.g. `foo :: Int -> Int -> Double` declares the type of the function `foo` to be a function which takes two integer numbers and returns a floating point number. The `->` symbol is right associative, so for instance the type declaration for `foo` could be bracketed like so: `Int -> (Int -> Double)`. Similarly, function application is left associative, therefore the application of `foo`

could be bracketed like this: `(foo 1) 2`. The associativity allows for the useful feature of *partial application*, which allows the creation of function values. For instance, the value `foo 1` is the partial application of `foo` to the value `1`, and thus results in a function of type `Int -> Double`.

The definition of a function involves specifying the name for the function, its arguments, and its behaviour. This is illustrated below.

```
foo :: Int -> Int -> Double
foo a b = fromInteger (a + b)
```

Here the first argument is called `a` and the second `b`, and the behaviour on the right hand side of the `=` symbol is to add them together and convert the result into a floating point value.

This example illustrates a very basic function, but shows the general pattern of function definition in Haskell. Further examples of function declarations are shown and explained Chapter 4 of this thesis.

Data types

Haskell features several built-in data types, such as `Int` for integer numbers, `Double` for floating point numbers and `Char` for characters. However Haskell also allows the declaration of new data types by using the `data` keyword.

Data types consist of a name and one or more constructors. To illustrate this, consider the example below.

```
data Shape = Rectangle Int Int | Triangle Int Int Int
```

This example defines a data type called `Shape` to hold information about a shape, and contains two constructors, one to store information about a rectangle, and one to likewise represent a triangle. Both constructors take arguments, in this case indicating the lengths of the sides of the shapes.

Functions can be written to operate on such data types by matching against the constructors, for instance in this example:

```
perimeter :: Shape -> Int
perimeter (Rectangle w h) = 2 * w + 2 * h
perimeter (Triangle a b c) = a + b + c
```

When data types are used in multi-module programs the data types must be exported from its containing module. This is done by including the name of the data type in the export list of the module. For example:

```
module MyShapes
  ( Shape (..)
  ) where
```

It is important to note the use of `(..)` after the data type name. This causes all the constructors of the data type to be exported from the module. However it is usually good design practice to hide the internal representation of data structures when writing complex programs. This can be achieved in Haskell removing the `(..)` from the export list and instead adding only those constructors or functions which one wishes to be “published”. Example 1 illustrates how an abstract shape type might be defined.

This example allows users of the `MyShape` module to use the `Shape` data type in their code, but they may only construct such a data type using the `makeTriangle` and `makeRectangle` functions. Furthermore, this allows the internal representation of the `Shape` data type to be modified, for example if it is necessary to add extra constructors or extra parameters to existing constructors, without affecting users of the module.

Control structures

So far the methods of declaring functions and data types have been described, but no mention has been made of how to perform actions within a function. In this section the various control structures available within Haskell will be described.

Control structures in Haskell do not contain loop constructs, such as the “while” and “for” loops that are common in imperative languages to sequence

```
module MyShapes ( Shape
                  , makeTriangle , makeRectangle
                  , isTriangle , isRectangle
                  , rectWidth , rectHeight
                  , triSideA , triSideB , triSideC
                  ) where

data Shape = Rectangle Int Int | Triangle Int Int Int

makeTriangle :: Int -> Int -> Int -> Shape
makeTriangle a b c = Triangle a b c

makeRectangle :: Int -> Int -> Shape
makeRectangle w h = Rectangle w h

isTriangle :: Shape -> Bool
isTriangle (Triangle a b c) = True
isTriangle (Rectangle w h) = False

isRectangle :: Shape -> Bool
isRectangle (Rectangle w h) = True
isRectangle (Triangle a b c) = False

rectWidth :: Shape -> Int
rectWidth (Rectangle w h) = w
rectWidth (Triangle a b c) = fail "Not a Rectangle"

rectHeight :: Shape -> Int
rectHeight (Rectangle w h) = h
rectHeight (Triangle a b c) = fail "Not a Rectangle"

triSideA :: Shape -> Int
triSideA (Triangle a b c) = a
triSideA (Rectangle w h) = fail "Not a Triangle"

triSideB :: Shape -> Int
triSideB (Triangle a b c) = b
triSideB (Rectangle w h) = fail "Not a Triangle"

triSideC :: Shape -> Int
triSideC (Triangle a b c) = c
triSideC (Rectangle w h) = fail "Not a Triangle"
```

Example 1: An example of an abstract data type

or repeat groups of commands. Instead controlled looping is performed using recursion, which in many cases of “tail recursion” the compiler can optimise into loops.

Instead, control structures in Haskell consist only of mechanisms for branching execution. There are four basic branching mechanisms, *pattern matching*, *guards*, *case expressions* and *if-then-else*.

Pattern matching. Pattern matching has two purposes. Firstly it allows a function to provide alternative actions for different inputs by using patterns in place of argument names in the declaration, and secondly it provides for selecting parts of the input. For instance, consider the following trivial examples.

```
isZero :: Int -> Bool
isZero 0 = True
isZero a = False
```

```
rectangleWidth :: Shape -> Int
rectangleWidth (Rectangle w _) = w
```

The `isZero` function of this example illustrates the use of pattern matching to provide alternative actions. If the function is passed a zero then the first line will be executed, otherwise the second line will be evaluated.

The `rectangleWidth` function illustrates the use of pattern matching. It extracts only the width field of the record. The `_` character is a wildcard and is used as a place holder for the height field which is not used in this function. A pattern match can contain multiple wildcards.

Patterns are explored in more detail in Section 4.1 of Chapter 4. It is important to note the limitation of patterns, in that they are essentially “static” branches, and cannot perform dynamic branching. For example, it is not possible to use pattern matching to perform an action if one argument is greater than another. Such dynamic checks need a different form of branching, such as guards.

Guards. Guards, like pattern matching, allow selective evaluation of parts of a function definition, but unlike pattern matching, allow dynamic checks. For

instance, consider this function.

```
largest :: Int -> Int -> Int
largest a b | b > a      = b
            | otherwise = a
```

This example features two guards, the first tests for the situation where `b` is greater than `a`, while the second guard, `otherwise`, is a “default” or “catch-all” guard. Guards are tested in the order they are written, and Haskell allows as many guards as one wishes. It is not necessary to end with an `otherwise` guard, although it is good coding practice to do so. Guards are discussed in more detail in Section 4.5.1 of Chapter 4.

Case expressions. Case expressions perform an analogous task to pattern matching, but are allowed on the right hand side of a function definition or as part of a larger expression. Thus case expressions are for expressions what pattern matching is for function definitions. For instance, consider the following example.

```
isZero :: Int -> Bool
isZero a = case a of
             0 -> True
             b -> False
```

This example shows a straightforward use of `case` at the top level of a function definition to match against zero, or any number and return the appropriate value. Case expressions can also be used as part of a larger expression, for instance consider the following function.

```
shapeToString :: Shape -> String
shapeToString s =
  "A " ++ (case s of
            Rectangle w h -> "Rectangle"
            Triangle a b c -> "Triangle")
  ++ " shape"
```

This example will generate a string such as “A Triangle shape”, depending on the given input, and demonstrates how case expressions can be used as part of a larger expression, in this case part of a larger string concatenation (`++`) expression.

If-then-else. Just as case expressions provide a mechanism to use pattern matching within an expression, Haskell also provides a mechanism to use guard-like functionality within expressions by using `if-then-else` statements. An example is illustrated below.

```
largest :: Int -> Int -> Int
largest a b = if a > b then a else b
```

Similarly, as with case expressions, `if-then-else` constructs can be used as part of a larger expression. For instance in this example.

```
printEvenOdd :: Int -> String
printEvenOdd n =
  "n is an " ++ (if n `mod` 2 == 0 then "even" else "odd") ++ " number"
```

The control structures presented in this section may also be used in any combination, as is illustrated in the contrived example below.

```
longestSide :: Shape -> Int
longestSide :: Shape -> Int
longestSide (Triangle a b c)
  | a >= c    = if a >= b then a else b
  | b >= c    = b
  | otherwise = c
longestSide (Rectangle w h) = if w > h then w else h
```

Summary

This section has presented a brief overview of some of the top-level syntax of the Haskell functional programming language. This introduction is intended only to familiarise the reader with some basic constructs of the language, and where necessary this thesis will expand on this introduction. Further information about Haskell may be obtained from the Haskell website, <http://www.haskell.org/>, the Haskell language report[80], or from several introductory texts on functional programming using Haskell, such as [94] or [46]. A detailed formal definition of the Haskell language and syntax can be obtained from the Haskell 98 Language report[80].

3.2 Implementing the Metrics: Medina

The work presented in this thesis studies a selection of software metrics and visualisation techniques for use on Haskell programs. The metrics and visualisation systems used in this study were implemented especially for use on Haskell as a library called Medina. The library is written in Haskell and is designed to be extensible, reusable and generic. The design and implementation of the Medina library is an integral part of the work presented in this thesis, so it is therefore described in detail in the following sections.

Medina is implemented using Haskell 98 and the multi-parameter type class extension supported by GHC [66], and consists of a library of functions and associated data types to assist programmers in writing their own metrics or visualisation systems. The Medina library can be loosely divided into two groups of functions, those for implementing metrics and those for implementing visualisation systems, which share a common base consisting of a parser and various data structures to represent Haskell programs.

To aid in extending and testing the library, a selection of example metrics and visualisation systems is included. The example metrics supplied with Medina are those that were used to perform the studies presented in Chapters 4 and 5 of this thesis. The example visualisation programs supplied with Medina are those described in Chapter 7 of this thesis.

The metrics support in Medina includes the following systems and the overall architecture of the system is illustrated in Figure 9.

- Parser.
- Transformations that can be performed on various representations of abstract syntax trees.
- Generic abstract syntax tree traversal routines.
- Support for temporal operations, such as applying a metric to every version of a program, using CVS.

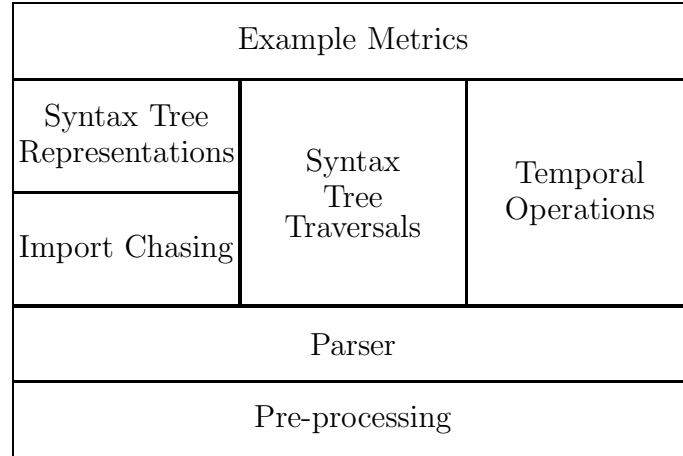


Figure 9: A block diagram of the Medina library metrics sub-system.

- Import chasing and processing of literal scripts.

The visualisation support includes the following systems and is illustrated in Figure 10.

- Basic functions for building GUI for OpenGL[16, 85] and SVG[30] output formats.
- An interface to GraphViz[31] for performing graph layout.
- Basic support for generating HTML.
- Basic support for generating GIF format images.

The Medina library consists of 99 modules, which in total contain approximately 30,000 lines of Haskell source code, including comments and Haddock documentation. The division of lines of code between the modules is summarised in Table 3.

The remainder of this chapter presents the design and structure of the library in greater detail and discusses how the library may be extended by a user. Finally, the chapter explains several directions in which the functionality of the Medina library could be enhanced.

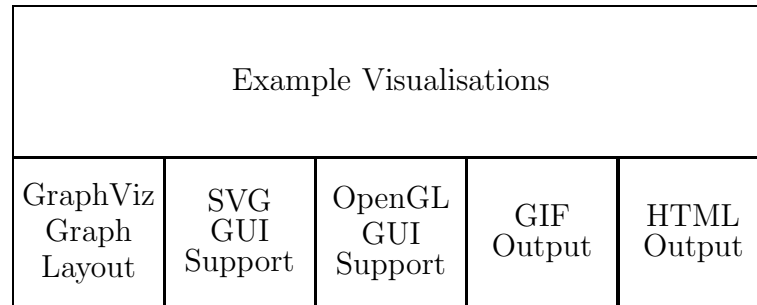


Figure 10: A block diagram of the Medina library visualisation sub-system.

Component	Approximate LOC
Front-end (parser and lexer)	8000
Data structures for representing programs	4500
Temporal operations (CVS integration)	800
Largest metric	650
Smallest metric	35
All metrics	6000
Visualisation systems	10000
Total lines of code	30000

Table 3: A summary of the number of lines of code in the Medina library.

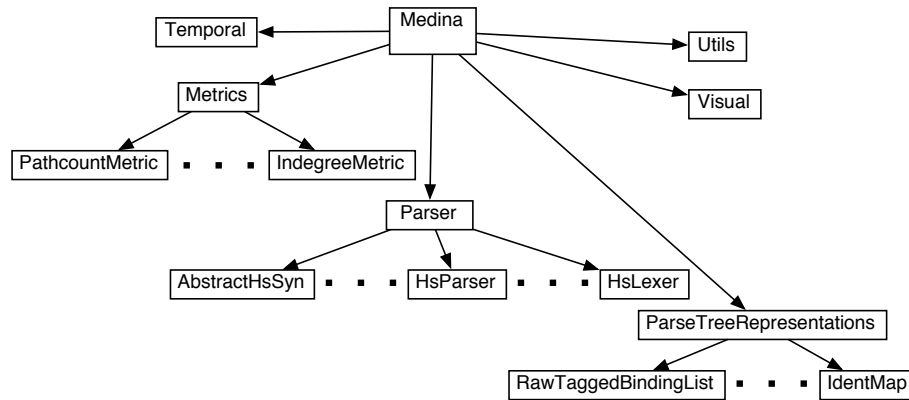


Figure 11: A simplified diagram of the Medina library module structure.

3.2.1 Medina Design

The Medina library consists of two loosely coupled halves, containing the metrics and the visualisation systems. The design of these two halves and the points of coupling between them are discussed in the following sections. A simplified overview of the module structure of the Medina library is provided in Figure 11.

Medina Metrics

The metrics half of the library consists of a “front-end” for the Haskell language, abstractions for traversing and manipulating abstract syntax trees, and alternative representations of Haskell programs, provided as alternatives to abstract syntax trees. Each of these components is described in detail now.

A Front-end for Haskell The “front-end” of the Medina library consists of three phases. The first phase performs “unlitting”, turning literal Haskell scripts (`.lhs`) into non-literate Haskell programs (`.hs`). The second and third phases are lexing and parsing respectively.

Medina provides two ways of performing unlitting. One method uses a pure Haskell 98 function to perform unlitting of a string containing the source code of the Haskell program. This method is completely portable and requires no

external support programs. However, this pure function does not support the use of preprocessor directives, such as CPP, embedded in Haskell programs, although in principal this could be added given further engineering effort.

Haskell compilers typically have some support for passing such programs through the appropriate preprocessor to expand these directives. Because this is common behaviour, the Medina library provides an alternative method of performing un-litling which requires the presence of GHC. This alternative method invokes GHC to perform the unlitling and removal of preprocessor directives, reading back the resulting output Haskell source code via a UNIX pipe. This method has the advantage of removing preprocessor directives, but requires the work to be performed within the IO monad and depends upon the presence of GHC on the user's system.

The parser and lexer used in Medina are based on those from Haddock[64], the Haskell documentation tool, and they provide syntactic support for various GHC extensions to the Haskell 98 specification. The parser and lexer are largely unchanged from those in Haddock, with the only modification performed being to make the data types used by the parser to represent the abstract syntax tree into abstract data types. Making the data structures abstract reduces the coupling between the front-end and the rest of the library, which may allow the library to use a different lexer and parser in the future, such as that from GHC, without affecting programs that use the Medina library.

Abstractions for traversing abstract syntax trees Writing metrics typically requires traversing the abstract syntax tree of a program. This can be a tedious task, so the Medina library attempts to provide some mechanisms to reduce this burden.

Medina does not contain a true generic programming system, such as Strafunski[58], but instead provides a collection of utility functions to perform common tasks, such as:

- Grouping all the bindings in a program by identifier.

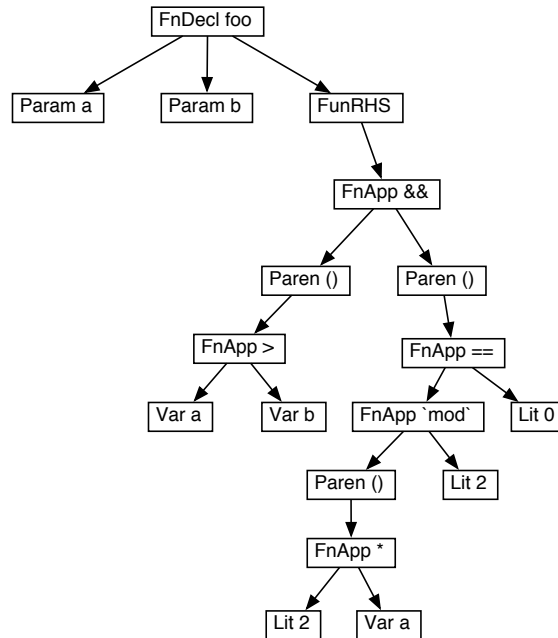


Figure 12: An example of a `ParseTree` data structure.

- Extracting a particular identifier from an abstract syntax tree.
- Selecting specific types of nodes from the abstract syntax tree.

Program representations in the Medina library Because the abstract syntax tree of a Haskell program is somewhat cumbersome to analyse, the Medina library provides a number of alternative representations which may prove easier to use than the raw abstract syntax tree. The various representations of the abstract syntax tree are described below.

- `ParseTree` is the basic abstract data structure generated by the parser and is the basis for all the other representations. The data structure forms a tree that mirrors the lexical structure of the program. For example, the structure for the function definition `foo a b = (a > b) && ((2*a) `mod` 2 == 0)` is shown in Figure 12.
- `BaseAbstractSyntax` is a concrete version of the `ParseTree` data structures

in which every node contains a field for storing meta data about the node. The meta data can be arbitrary, but every node must have the same type meta data. Meta data is useful for tasks such as holding intermediate values of metrics, or holding final metric values to be passed on to a pretty printer for colour coding in a visualisation.

Each node of the structure is also labelled with a unique identifier. These identifiers are attached by performing a depth-first traversal of the structure, during which every node of the structure is stamped with a unique integer. These unique identifiers do not convey information about the program being represented, such as binding or usage information, but are instead used only as references into the abstract syntax tree data structure. This allows for nodes to be extracted from the abstract syntax tree, processed, then inserted back into the tree. Functions are provided to perform this *merging* process. This can be useful when one of the other representations is used to generate a value which one wishes to be stored in the meta data of the `BaseAbstractSyntax`. For instance, one might use a more abstract representation such as an `IdentMap` (described in detail below) to calculate a metric value, but store the results as meta data in order to pass them to a pretty printer.

To illustrate the structure of a `BaseAbstractSyntax` data type, Figure 13 shows the `BaseAbstractSyntax` representation of the example `ParseTree` structure shown in Figure 12.

- `RawTaggedBindingList` represents a Haskell program as a list of the function and pattern bindings in the program source code. Each binding is represented by its abstract syntax. `RawTaggedBindingList` structures are generated from a `BaseAbstractSyntax`, and may be merged back into the source `BaseAbstractSyntax`. This is a useful abstraction because it is often necessary to examine only the bindings in a program in order to perform a measurement, ignoring all the type and data declarations. Figure

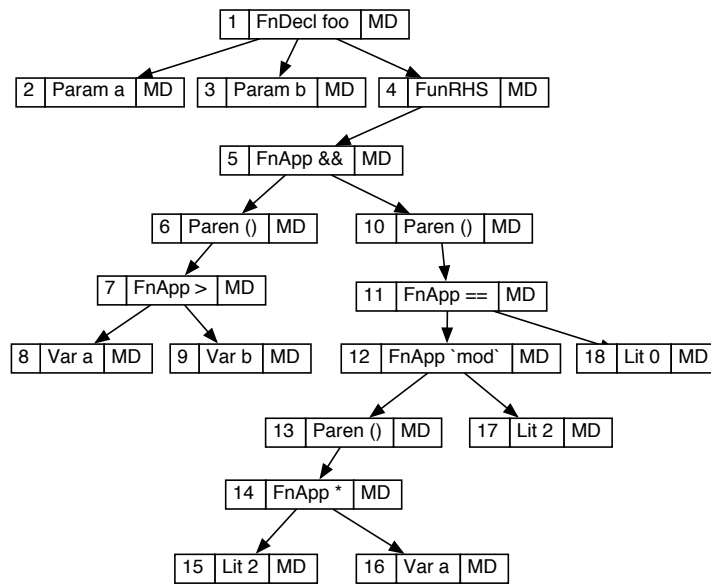


Figure 13: An example of a BaseAbstractSyntax data structure.

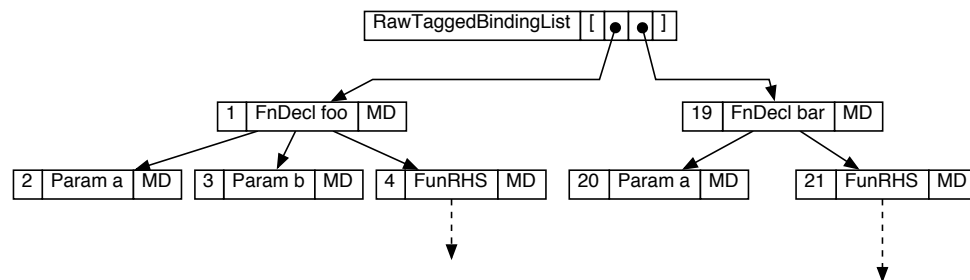


Figure 14: An example of a RawTaggedBindingList data structure. Some nodes are not included in this diagram in order to clarify the structure.

14 illustrates a `RawTaggedBindingList` structure for the following example program.

```
foo :: Int -> Int -> Bool
foo a b = (a > b) && ((2*a) `mod` 2 == 0)

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

- `IdentMap` is similar to the `RawTaggedBindingList`. It represents a Haskell program as a finite map from fully qualified identifiers to the corresponding abstract syntax tree nodes. Thus for every fully qualified identifier there is a list of AST nodes representing the elements of the identifier definition, such as its type signature and its bindings. This differs from the `RawTaggedBindingList` in two ways. Firstly it includes all elements that have an identifier, including type and data declarations, which are not included in a `RawTaggedBindingList`. Secondly, the `IdentMap` is a mapping from identifiers to abstract syntax, rather than a simple list of abstract syntax elements. This frees the metric writer from having to extract identifier names themselves, and also provides simpler mechanisms for applying metrics by using the standard operators on finite maps, such as `mapFM`. Figure 15 illustrates the `IdentMap` structure for the example program previously used to illustrate the `RawTaggedBindingList` structure in Figure 14.
- `ModuleImportGraph` represents the import hierarchy of a program as a directed graph with edges indicating import statements and nodes representing modules. Nodes in the `ModuleImportGraph` contain the `ParseTree` of the corresponding module, if it is available. The syntax might not be available because although Medina may know a module exists, due to an import statement that references the module, it may not know where to find its source code, for instance, the module may be part of a binary library shipped with a compiler. Figure 16 shows an example of a `ModuleImportGraph` structure for the program in Example 2.

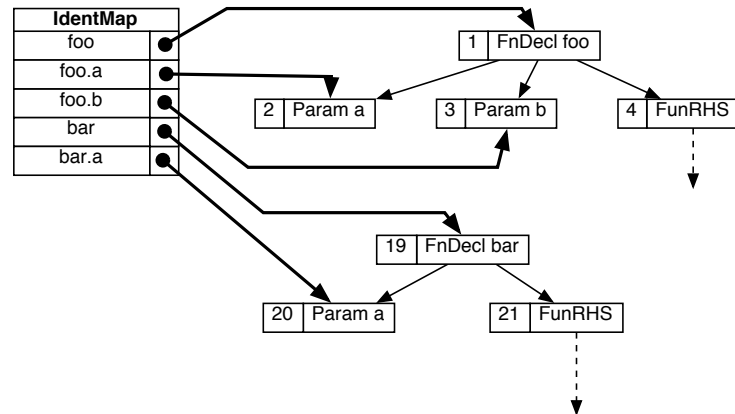


Figure 15: An example of an `IdentMap` data structure. Some nodes are not included in this diagram in order to clarify the structure.

```

module Bang where
bang :: Int -> Bool
bang a = a*a > 25

module Bar where
import Bang
bar :: Int -> Int -> Int
bar a b | bang a    = a
        | otherwise = b

module Foo where
import Bar
import Bang
foo :: Int -> Int -> Int
foo a b | bang b    = b
        | otherwise = foobar a b

foobar :: Int -> Int -> Int
foobar a b = bar (a+b) (a*b)
  
```

Example 2: An example of a multi-module program.

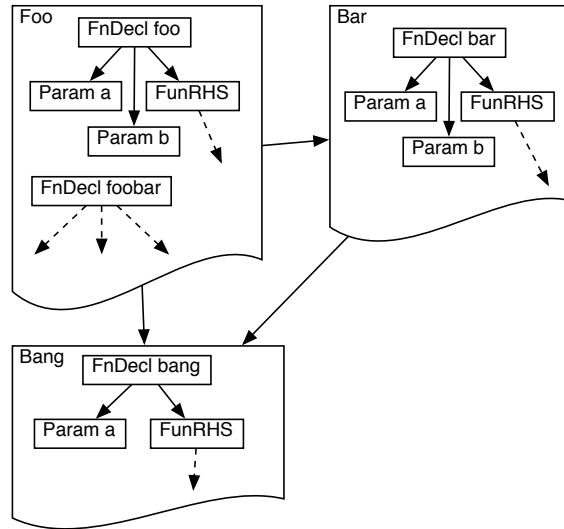


Figure 16: An example of a `ModuleImportGraph` structure for the program in Example 2. Some parse tree nodes are hidden for clarity.

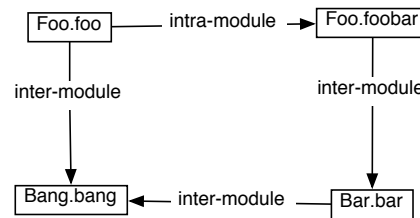


Figure 17: An example of a `CallGraph` structure for the program in Example 2.

- `CallGraph` represents the callgraph of a Haskell program as a directed graph. Nodes of the graph are marked with their fully qualified identifier, while edges are marked with their type, e.g. whether they are calling an identifier in the same module (intra-module), in another module (inter-module), or if the callee has not yet been processed (unresolved). The `CallGraph` structure of the program in Example 2 is illustrated in Figure 17.
- `TotalCallGraph` is a conglomeration of `ModuleImportGraph` and `CallGraph`. It represents Haskell programs as a directed graph in which nodes represent modules and edges represent import statements. The nodes of the graph,

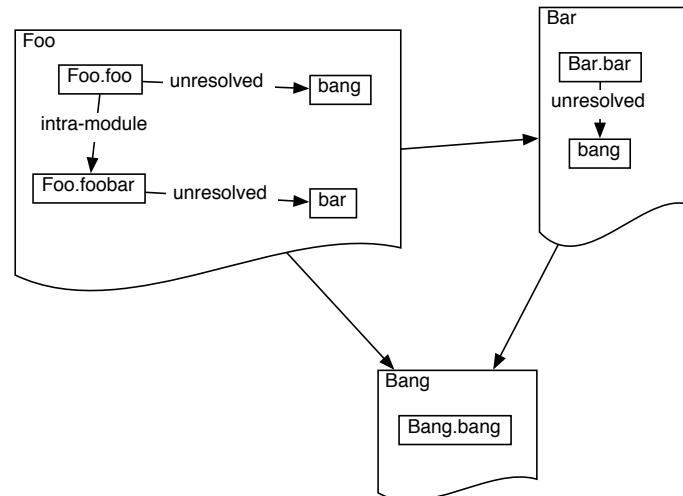


Figure 18: An example of a `TotalCallGraph` structure for the program in Example 2.

have the `CallGraph` structures of the corresponding module attached to them. Because `TotalCallGraph` is related to `CallGraph`, functions are also provided to convert a `TotalCallGraph` into a `CallGraph`. The `TotalCallGraph` structure for the program in Example 2 is illustrated in Figure 18.

The Medina library also includes a number of utility functions for converting between the various program representations described above. For instance, functions are provided to generate a `Callgraph` from a list of file names or `ParseTree`'s, or for generating a `IdentMap` from a `ParseTree`.

The example metric programs provided with the Medina library all use one or more of these abstractions, and the provision of the abstractions does make implementing metrics easier. One area that still requires some work is in additional support for traversing these various representations, such as by using a generic programming library like `Strafunski`[58].

Temporal operations using CVS To aid the work presented in this thesis, support was added to Medina to perform temporal operations such as applying a metric to every version of a program between two given dates. This support is

provided by an interface to CVS, a commonly used version control system.

CVS allows a user to record all versions of the source code of a program and at a later date retrieve specified versions. Medina includes a low-level interface to CVS which provides Haskell equivalents of all the CVS operations, as well as a higher level interface intended to be used by users of the Medina library. This higher level interface provides operations for mapping metrics over ranges of dates or version numbers of software in the CVS repository.

Medina Visualisation

The visualisation systems in the Medina library are designed to be modular and reusable. This allows the various visualisation components to be reused as part of more complex visualisation systems. Most of the visualisation systems are decoupled from the metrics and the abstract syntax tree, and instead use some form of generic input description. This allows the visualisation systems to be used with a wider range of inputs, which may extend beyond program visualisation. However, currently both the visualisation systems and metrics are combined into a single library image and are thus linked together at compile time. In future the library will be further decoupled, such that the visualisation and metrics systems are contained in separate library images, allowing the visualisation systems to be used without requiring the metric systems to be linked into the final executable.

The visualisation systems currently use HOpenGL to provide the user interface. Each visualisation component is designed to render its display into a separate panel, which allows the various components to be used together, in particular it allows components to be embedded within other components. The available components can be divided into two groups:

- GUI support, such as scrollbars, buttons, and menus.
- Basic visualisation components, from which more complicated visualisation systems can be built.

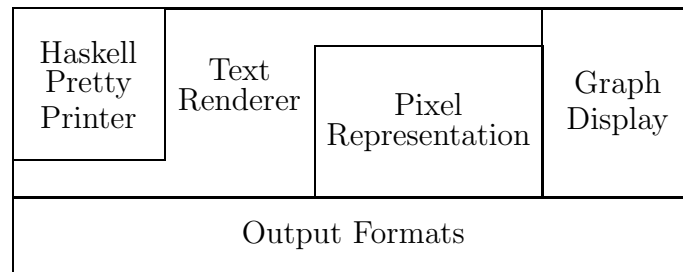


Figure 19: An overview of the structure of the basic Medina visualisation components.

The GUI support components are not discussed further in this work because they are likely to be familiar to most readers who have used GUI programs in operating systems such as Mac OS or Windows. The basic visualisation components are more interesting and are therefore described in more detail in the following section.

Basic visualisation components All of the visualisation components provided by the Medina library are built from a few basic components. These basic components can be combined in many ways to form more complex visualisation systems, such as those described in Chapters 6 and 7 of this thesis.

The Medina library provides the following basic components which are illustrated in Figure 19.

- Haskell pretty printer. This provides a mechanism for turning abstract syntax trees into text representations of the source Haskell program. The current pretty printer is quite basic so therefore the pretty printed Haskell program does not always look like the source Haskell program. However, if the abstract syntax tree and token stream contain enough information it

would be possible to replace the existing pretty printer with a more sophisticated pretty printer that can preserve the layout of the Haskell program, such as that used by the HaRe[60] tool.

The Medina pretty printer has support for encoding colour information in the textual output, enabling metric values embedded in the abstract syntax tree to be used for colour coding parts of the pretty printed source.

The output of the pretty printer can be passed directly to other visualisation components such as the text renderer or the pixel representation, both of which are described below.

- Text renderer. The text rendering system in the Medina library allows abstract descriptions of textual content to be display in a GUI component.

The abstract description used as input to the text renderer represents text as blocks of characters which share common attributes. Currently the only supported attribute is colour, but it would be possible to add other attributes such as font information, character styles such as bold and italic, or font size information. The blocks of characters can include line breaks.

The interface to the text renderer is overloaded such that there are various implementations that can be used via a common interface. Different implementations generate different output formats. Medina provides text renderers for generating output to plain ASCII text files, HTML files, GIF images, to the screen as a GUI component, and as a pixel representation. Pixel representations are described below.

- Pixel representation. The pixel representation is a specialised implementation of the text renderer interface, described previously. Pixel representations are described in detail in Section 6.1.1 of Chapter 6 and in Section 7.1.1 of Chapter 7 of this thesis, but are briefly introduced here. Pixel representations display characters as a solid area of colour, typically one pixel square, rather than displaying the character glyph. The result of rendering

text using a pixel representation is to dramatically reduce the amount of space used by the rendered text, while still preserving its shape.

Pixel representations are often used to form the backdrop of a scroll bar, or to give an overview of a data set.

- **Graph display.** The Medina library provides a component for displaying directed graphs. The component uses a static layout in which the nodes and edges are positioned at the time the component is created and are fixed thereafter. A dynamic layout would allow user interaction to move the nodes and edges from their initial positions, however this is not supported yet. The Medina graph display component provides facilities for zooming in and out of the graph, and for scrolling around the graph.

The graph display component allows the library user to control many aspects of the graph display by providing callback functions. Callback functions can be passed to the component to change how the edges and nodes are drawn, e.g. to have dashed lines instead of solid. The graph display component also allows callback functions to be attached to various GUI events, such as the user positioning the mouse cursor over an edge or node, or clicking on an edge or node, and other similar events. These are used to good effect in the Haskell module browser described in Section 7.1.3 of Chapter 7 of this thesis.

Example programs The Medina system includes a small selection of examples programs which demonstrate how the visualisation components in the library should be used, and which were also used for the work presented in Chapter 7 of this thesis.

These example visualisation programs currently use a fixed, hard-coded choice of metric to be visualised. It is only possible to select a different metric in these programs by editing the source code and re-compiling them.

However, these programs are intended only as example and proof-of-concept

implementations, and as such the Medina library supports the implementation of more realistic visualisation tools which, for instance, might allow the user to select at runtime which metric to use for the visualisation, or to dynamically switch between metrics as the visualisation is running, or even to display multiple metrics and visualisations simultaneously.

Coupling between metrics and visualisation

Although the metrics and visualisation parts of the Medina library are largely decoupled, there are a few points where the two parts are coupled.

The visualisation components take a generic input description and therefore do not depend on the metrics part of the library. Equally the metrics part of the library does not depend upon the visualisation part.

However, there are routines provided to convert metric data and abstract syntax tree representations into the generic descriptions that the visualisation components require. These conversion routines therefore cause coupling between the two halves of the library.

Furthermore, there are some utility modules that are shared between the two halves, such as some utilities for performing IO actions. These modules add further coupling between the two halves.

Therefore the overall structure of the Medina library is a diamond shape, with a shared base, two distinct sides, and then a top which enables the two sides to be used together. This structure is illustrated in Figure 20.

3.2.2 Extending Medina

The original goal of the Medina library was to enable users to build their own metrics using the metrics and tools built into the library. Currently this can be achieved by using the various abstract syntax tree representations to select the parts of a program that are to be measured, and then by writing functions to measure the required attributes. This is further eased by the provision of a small

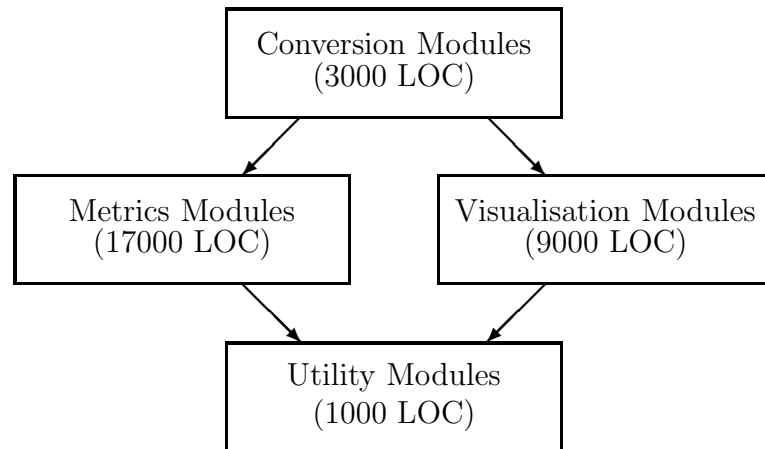


Figure 20: An overview of the design structure of the Medina library. Numbers in brackets indicate approximate code size.

selection of type classes and functions which provide for a limited form of generic traversal of the raw syntax trees. For instance, a function `selectPT` is provided to allow selection of specific nodes from a parse tree, such as in the following example that extracts all pattern nodes from a tree:

```

isPat PatNode = True
isPat _       = False

selectPT isPat parsetree
  
```

Inevitably however, this mechanism still requires the programmer to have fairly detailed knowledge of the abstract syntax tree of a Haskell program and may also require, depending on the complexity of the metric being implemented, a significant amount of tedious boilerplate code to be written.

Therefore, the Medina library does make steps towards its original goal of making it easier for programmers to implement their own metrics. However improvements can be made in this area, for instance by adding support for more powerful generic programming techniques using a library such as `Strafunski`[58].

3.2.3 Future Expansion

The Medina library presented with this thesis is a first version of a toolkit for implementing software metrics and visualisation systems for Haskell programs. This

first version has shown that there are a number of areas of the library which could be improved. These can largely be split into two areas, engineering improvements and integration with other systems, which will be discussed separately.

Engineering Improvements

The engineering improvements that could be made to the Medina library focus largely upon the usability of the library. Usability includes both usability for potential tool writers, implementing tools that use the library, and the usability for the end-user of such tools.

To increase the usability for tool writers, there are a number of changes that could be made. The most useful change would be to use a real generic programming library, such as Strafunski[58], to implement the abstract syntax traversal functions. This would not only result in a reduction in the size of the Medina library, but would also relieve tool writers from having to write as much boilerplate code for performing their metric analysis. The benefits of Strafunski for this type of work have already been demonstrated by HaRe, the Haskell refactoring tool.

A further change that may benefit tool writers is to divide the Medina library into two halves, one for the metrics functionality and the other for the visualisation systems. Such a division would have several benefits. Firstly, the API of the library is likely to be cleaner and simpler, allowing tool writers to examine only the documentation they need for their particular type of tool, be it visualisation or metric. Further more, such a division reduces the dependencies of metric tools, which would otherwise have dependencies on GUI libraries which would not actually be required. Reducing dependencies in software is important to make use of such tools easier.

The usability of the library for end users is mainly limited by the graphical interface library used for displaying the visualisations. Currently Medina uses the HOpenGL library, which offers no support for providing native look and feel for applications. The emerging standard GUI library for Haskell programs appears

to be wxHaskell, a binding to the cross-platform wxWidgets library. The wxHaskell library provides native look and feel across platforms, and so porting the visualisation systems to wxHaskell may significantly improve the usability of the visualisations.

However, such a port may not be simple. Some of the visualisation systems, most notably the callgraph visualisations, require careful optimisation to obtain acceptable user interface responsiveness when displaying complex programs. This is possible in HOpenGL due to the low-level nature of the library, but there is a danger that using a higher level user interface might reduce the opportunities for such optimisations. The optimisations performed by the Medina library are described in detail later in Section 7.2.4 of Chapter 7.

Integration with Other Systems

As well as the engineering enhancements described in the previous section, there are a number of directions in which Medina could be improved by integration with other tools that perform related tasks.

One such tool is HaRe [60], a tool for performing refactoring of Haskell programs. As has been described in Section 1.2 of Chapter 1 of this thesis, metrics can be used to indicate where a refactoring should be performed. Because of this it seems logical to integrate HaRe and Medina in some way. HaRe provides both a tool for performing refactorings, and an API for implementing refactorings and other such program analysis tools. One way to integrate Medina with HaRe would be to either combine their APIs, or port Medina to use the HaRe API. This may make it easier to implement tools such as visualisers to help a programmer find locations where they may wish to apply refactorings.

Further work could also investigate ways of combining the Medina library with tools such as QuickCheck [21] to help target testing effort, or Hat [20] to drive the process of tracing and debugging.

3.3 Experimental Methodology

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 [10] 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 pathcount 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 pathcount 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 [35], 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.

3.3.1 First Attempt: Happy

The original intention was to take a real-world Haskell program and analyse its change history and its metric values. To be sure of a comprehensive change history the program needed to be managed by some form of version control system, such as CVS. Several programs from the `haske11.org` CVS repository were examined and “Happy”, a parser generator similar to `yacc`, was chosen. Happy was chosen because it appeared to contain a change history that spanned several releases, consisted of approximately 5500 lines of code and so was large enough to have some confidence of meaningful statistical analysis, and yet was small enough for the code to be able to manually inspected. Manual inspection of the code is necessary when the purpose of a change to the code is not clearly marked as either a bug fix or the addition of a feature, and must therefore be determined manually.

The first task was to determine how many changes each function of the program had undergone during the program’s lifetime and to classify each change as either fixing a defect, adding a feature, or refactoring the code. To do this it was necessary to look at the change history of the program and determine where each change occurred and what the change was doing. It was hoped that the log messages that are added to each change in the CVS repository would enable the changes to be classified. Sadly this proved not to be the case because there would often be several changes committed with a single log message that covered both a defect fixing change and a separate feature addition change.

Because of this it was necessary to manually view and classify each of the changes to determine where and when the various types of changes occurred, a

long and tedious task. The result of this was a count of the number of bug fixes for each function in Happy.

However this showed that there were only a small number of bug fixing changes occurring during the lifetime of the program. Out of approximately 250 functions there were only 25 bug fixes, occurring in just 14 functions, with one function containing 12 of those bug fixes!

There are several reasons why there might be so few bug fixes appearing in the code. The code may be well written and therefore relatively defect free, or bug fix changes may have been missed or mis-classified as feature additions when manually reviewing the change history. The granularity of commits may also affect the number of bug fix changes that were found. For instance, a bug fix change may have been made but not committed to CVS, and then a feature addition may have overwritten the bug fix change before being committed to CVS. This would mean the bug fix change was never committed to CVS, and therefore never seen in our analysis.

Alternatively, Happy has undergone some relatively large changes that were implementing additional features during its lifetime. There are several such changes and it is possible that these changes are hiding underlying bug fixes, e.g. Function X contains a defect, but that defect was “fixed” when function X was changed to add feature Y.

Because of the extent of the feature changes it was deemed that Happy did not provide a good test case for the statistical work and this test case was abandoned.

3.3.2 Second Attempt: Peg Solitaire

After reflecting on the first attempt at a case study it was decided that a program would be specifically written for the case study. This decision was reached partly because of the effort involved in classifying changes in an unknown program and partly because a research project was starting at Kent to look at refactoring of functional programs [60]. One of the first tasks of that project was to write a program that the project team could use to familiarise themselves with the

Module	Min Size (LOC)	Max Size (LOC)	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
Totals:	502	906	148

Table 4: Information about the Peg Solitaire case study program. Note: The total sizes are approximate figures only, due to individual modules change sizes at different times during the evolution of the program.

Haskell language and which later could be used as a case study. It was felt that this metrics work could use the Refactoring group’s program. Their case study program was an ideal program for our own use because we had no input to the development of this program, other than to request a fine grained CVS commit policy which resulted in bug fixing changes being clearly marked.

The program used for the case study was an implementation of a peg solitaire game. The game includes both a textual and a graphical interface with which to play the game. The program consists of six modules and went through 41 revisions during its lifetime. Some details of the number of changes occurring in each module are presented in Table 4.

No reference is made to which file modules occur in because this sometimes changed during the history of the program. An example of this is where `Main` moved from the file `TPegSolitaire.hs` to its own file, `Main.hs`. Because of this it was decided that changes should be assigned to the module they occurred in rather than the file. This makes it easier to track metric values over time because we can ignore which file a module has been loaded from. It is also worth noting that not all the modules necessarily existed at the same point in time. For instance, later versions of the program did not contain the `Solve` module, while earlier versions did not contain the `GPegSolitaire` module.

To calculate the number of changes occurring in each module the source code

for the program was manually inspected using `TkCVS` [95], a graphical interface to the CVS revision control system. `TkCVS` provides an easy way to view the differences between two versions of a source file using a visual difference tool, and using this we classified each change throughout the program as either a bug fix, a feature addition, or a refactoring change. There were relatively few bug fix changes but a reasonable number of refactoring changes, some of which may also have included bug fixes. For the purpose of exploring metric values, the number of bug fix and refactoring changes were combined because both types of changes indicate reasons for closer examination of the relevant sections of code, for instance refactoring changes can be thought of as fixing bugs in the design of a program. Therefore this examination of metrics is evaluating the correlation of metric values with the number of “fixing” changes, where a fixing change may be either a bug fix or a refactoring.

The case study was run by applying metrics to each revision of the program over the course of its lifetime using the CVS integration of the Medina library, described later in Section 3.2. From these metric values the maximum value of the metric for each function in the program was taken as the “score” for the functions. The scores for the functions were then correlated with the number of bug fix and refactoring changes for each function using the statistical functions within Excel. The results of this experiment are discussed later in Chapter 4.

3.3.3 An Additional Case Study: Refactoring

As well as using the Peg Solitaire program as a case study it was felt that a larger piece of software should also be studied. As part of the Kent Refactoring group’s work they have written a tool to perform refactorings on Haskell program code. A pre-release version of this tool was chosen as a second case study. The tool used a library for parsing Haskell code which was not examined in this study, so only the code that manipulated parse trees was analysed. Details of the sizes and number of changes for each module of the program are presented in Table 5. This shows that the Refactoring case study program is approximately twice the size of

Module	Min Size (LOC)	Max Size (LOC)	Changes
EditorCommands	198	215	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
Totals:	2624	5122	444

Table 5: Information about the Refactoring case study program. Note: The total sizes are approximate figures only, due to individual modules change sizes at different times during the evolution of the program.

the Peg Solitaire program.

The techniques used for classifying changes in the Peg Solitaire program were again used for the Refactoring tool. Metric measurements and analysis were performed in the same way for both the Peg Solitaire program and the Refactoring tool. The results of these experiments are discussed later in Chapter 4.

3.3.4 A Larger Body of Programs

Basing the statistical work on only two case study programs is somewhat limiting, however the work involved in manually inspecting change histories is prohibitive. An alternative to looking at the number of bug fixing changes is to consider the relationships between metrics. In order to do this it is only necessary to take snapshots of programs and apply the metrics to those. This can be done quickly and automatically.

For this work a selection of programs was gathered from the list available at <http://www.haskell.org/libraries/> on the 28th of July, 2003. The following programs were chosen for this study, and their various characteristics are summarised in Table 6.

- CGI Library (CGI). A library for writing CGI programs for web servers. <http://www.cse.ogi.edu/~erik/Personal/cgi.htm>
- Haskell Cryptographic Library (CRYPTO). A library of cryptographic functions. <http://www.haskell.org/crypto/ReadMe.html>
- Haskell DSP Library (DSP). A library of digital signal processing functions. <http://haskelldsp.sourceforge.net/>
- FGL (FGL). A library of graph operations. <http://www.cs.orst.edu/~erwig/fgl/>
- The Library for Geometric Algorithms (GEOMLIB). A library of geometric functions. <http://www.dinkla.net/fp/cglib.html>
- GetOpt (GETOPT). A module for GNU/POSIX-like handling of command line arguments. http://www.pms.informatik.uni-muenchen.de/mitarbeiter/panne/haskell_libs/GetOpt.html
- Haddock (HADDOCK). A tool to generate documentation from Haskell source code. <http://www.haskell.org/haddock/>
- Happy (HAPPY). A parser generator for Haskell, similar to yacc. <http://www.haskell.org/happy/>
- Hat (HAT). A collection of tools for debugging of Haskell programs, using tracing. <http://www.cs.york.ac.uk/fp/hat>
- HaXml (HAXML). A library of XML tools, including parsing, pretty printing and transformations. <http://www.cs.york.ac.uk/fp/HaXml/>

Program	LOC	Functions	Modules
CGI	1426	214	17
CRYPTO	739	73	8
DSP	6973	523	78
FGL	3159	442	26
GEOMLIB	7236	869	22
GETOPT	198	16	1
HADDOCK	11789	1945	16
HAPPY	4779	400	14
HAT	15696	1661	57
HAXML	7294	817	28
HUNIT	1094	112	11
PRETTY	935	82	1
PCF	242	59	3
THIH	11259	563	38

Table 6: A summary of the case study programs showing their sizes in lines of code and the number of functions and modules they contain.

- HUnit (HUNIT). A unit testing framework for Haskell. <http://hunit.sourceforge.net/>
- Pretty printer library (PRETTY). Simon Peyton Jones industrial strength pretty printing library. <http://research.microsoft.com/~simonpj/downloads/pretty-printer/pretty.html>
- An implementation of untyped PCF in typed PCF (PCF). This software was donated by a colleague, Dr Stefan Kahrs, rather than selected from Haskell.org. http://www.cs.kent.ac.uk/people/staff/smk/PCF/Untyped_PCF.html
- Typing Haskell in Haskell (THIH). A Haskell program that implements a Haskell type checker. <http://www.cse.ogi.edu/~mpj/thih/>

Each of the metrics was run on each of the programs and the results were placed into a single spreadsheet. This was then used to calculate the cross-correlation information discussed in Section 5.1 of Chapter 5 and the mean, mode, median

and standard deviation values discussed in Section 5.2 of Chapter 5.

3.3.5 Summary

There were a number of problems encountered during the course of this work, and it is useful to discuss these here. The first problem was finding suitable programs to use as case studies. Ideally a program maintained in a CVS repository, with a comprehensive change history was needed. However most of the programs investigated had no clear separation between bug fixing changes, refactoring changes and feature additions, with different types of changes often being committed to CVS in the same commit. This made it necessary to manually inspect the code changes to determine the type of change, a very time consuming and error prone procedure.

Because of the need to manually inspect the changes it was necessary to choose programs that were small enough that it was possible to inspect them within a reasonable amount of time, but this can lead to problems because the programs may then be too small for statistically significant results to be obtained, as is seen for the Peg Solitaire program in the results of this experiment which are described in Chapter 4.

Programs which undergo frequent and/or large feature additions cause problems when analysing change histories because the large changes can often mask bug fixing changes. This can lead to artificially low bug fix counts, making such programs unsuitable as case studies. This problem can be partially relieved if a strict, fine grained commit policy is used, such that all bug fixes are individually committed to the repository. This allows bug fixing changes to be extracted regardless of the amount of feature additions present. However, this still does not help in the case where a feature addition “accidentally” fixes a bug, in which case the bug fix will go unnoticed. This may be unavoidable.

Investigating cross-correlation between metrics can be achieved much more simply than correlating metrics with bug fixes, and as such it is possible to examine both larger programs, and a greater number of programs than is feasible when

correlating with bug fixes. However this does not help to explain the relationship between metrics and bug fixes, and as such is not a substitute for longitudinal case studies that examine the correlation between bug fixes and metric values.

An ideal case study would consist of a program that has a rigid commit policy in which each change has been clearly labelled with its kind, such as bug fix, feature addition or refactoring, and individually committed into a version control system such as CVS. Such a program would have undergone several releases or revisions, and would have had enough defects, and therefore bug fixes, that it is possible to obtain statistically significant results from the program.

Chapter 4

Software Measurement for Haskell

Chapter 2 presented a selection of previous work on measuring a variety of aspects of software. Most of this work relates to imperative or object oriented programming languages. In this chapter some of the ideas from Chapter 2 are applied to the functional programming language Haskell, although the observations in this thesis may apply equally well to other functional languages such as SML and Clean.

There have been many software metrics defined over the years and hence Chapter 2 represents only a broad sample of the possible measurements one might make from imperative and OO programs. There are a number of these imperative and OO metrics which may transfer over to the functional programming domain with little or no modification, and these are briefly enumerated here.

- *Program size.* The various measures of program size, such as LOC and Halstead's program volume can be transferred to functional programs with little or no modification. These measures are used in Section 4.5.2 of this chapter.
- *Testing effort.* Metrics that measure the effort involved in testing a program, such as pathcount and cyclomatic complexity, can be applied to functional

programming languages, although as is described later in Section 4.5.1 implementing such metrics requires care to correctly consider all execution paths. The pathcount measure is analysed for Haskell in Section 4.5.1 of this chapter.

- *Callgraph measures.* Measurements taken from callgraphs, such as size, depth, width and arc-to-node ratio, can all be applied to functional programs without modification. These measures are examined for use with Haskell in Section 4.4 of this chapter.
- *Coupling measures.* Although it is possible to apply OO coupling measures, such as object level and class level coupling and coupling strength, from functional languages such as Haskell, some modifications are needed to map different language elements into the metric. For instance, OO coupling metrics normally measure coupling between classes, but Haskell classes are not classes in the OO sense, and instead one is most likely to measure coupling between modules or functions. Coupling between functions is examined briefly in Sections 4.4.2 and 4.4.3 of this chapter.
- *Design artifacts.* In principle metrics which operate on design artifacts such as UML diagrams should be applicable to the equivalent design artifacts of functional programming languages. Unfortunately, the functional programming world generally does not follow any formalised software design process, partly because the topic of designing functional programs is under-researched, with the thesis of Russell [84] and paper of Wakeling [99] constituting the majority of the work in the area. Because of this, the ideas presented in this section do not appear to have much applicability to Haskell program design at the current time.

This chapter presents versions of these metrics for use with Haskell, as well as other metrics designed to measure particular language features that may not be covered by existing imperative metrics.

Using the methodology described in Chapter 3, this chapter shows the correlation coefficients that result from correlating these measurements with the change history of two Haskell programs, Peg Solitaire and Refactoring, to determine if higher metric values are correlated with higher numbers of bug fixes. The two programs are relatively small, with Refactoring being approximately twice the size of Peg Solitaire. These programs were described previously in Chapter 3. Statistical significance is assumed to be at the 5% level unless otherwise stated.

Statistical analysis is also used to select the combinations of measurements that provide the highest correlation, and to investigate how the attributes being measured interact.

From this statistical analysis a number of observations are made about the relationship between the metrics and the subjective complexity of Haskell functions. The term *subjective complexity* is used as a notion of the complexity that might be perceived by a programmer attempting to understand or modify the given function. This should not be confused with *computational complexity* which is normally specified in $O()$ notation. Unlike computational complexity, subjective complexity cannot be concretely specified and instead metrics provide indications of the subjective complexity.

The remainder of this chapter is divided into the following sections.

- Section 4.1 investigates the complexity of pattern expressions in Haskell programs.
- Section 4.2 discusses attributes that can be measured by examining the distance between where identifiers are used and where they are declared in Haskell programs.
- Section 4.3 looks at the important property of recursion in Haskell programs.
- Section 4.4 presents some generic measurements that can be taken from the callgraph of a program and investigates their application to Haskell programs.

- Section 4.5 examines some generic measurements of program size and program complexity and applies those to Haskell programs.
- Section 4.6 investigates how the various metrics may interact and shows that several of the metrics are strongly correlated.
- Section 4.7 summarises the conclusions from the work in this chapter and suggests some ways to utilise this information.

Further analysis of the metrics presented in this chapter is performed in Chapter 5, which studies both the interaction between the metrics, and the typical metric values that are exhibited by a collection of Haskell programs.

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (n:ns) = n + sumList ns
```

Example 3: Using pattern matching to introduce identifiers.

```
data Shape = Rect Int Int | Triangle

isSquare :: Shape -> Bool
isSquare (Rect w h) = w == h
isSquare Triangle   = False
```

Example 4: Using pattern matching with algebraic data types.

4.1 Measuring the Complexity of Patterns

Pattern matching is widely used in Haskell. It provides a mechanism to selectively handle various classes of function arguments. For instance, consider Example 3.

In the `sumList` function there are two patterns. The first pattern matches an empty list, `[]`, and the second pattern matches the first element of the list, `n`, and the possibly empty tail of the list, `ns`. This pattern introduces two identifiers into the scope, which are then used on the right hand side. These identifiers may be overridden by definitions in a local definition such as a `let` expression or `where` clause.

Patterns are also invaluable in manipulating algebraic data types. Patterns may contain constructor names which cause the pattern to match only that constructor. Consider the program in Example 4.

In the `isSquare` function the first pattern will match if and only if the shape is a `Rect` while the second pattern will match if the shape is a `Triangle`. This indicates how patterns can be used to discriminate between constructors. The first pattern, which extracts the width and height from the `Rect`, also demonstrates how patterns may be used for selection as well.

By nesting patterns it is possible to perform complicated selection operations,

```
-- Colour as RGB values
type Colour = (Int,Int,Int)
data Shape = Square Colour | Triangle Colour

redValue :: Shape -> Int
redValue (Square (r,g,b))    = r
redValue (Triangle (r,g,b)) = r
```

Example 5: Using nested patterns.

```
redValue :: Shape -> Int
redValue (Square (r,_,_))    = r
redValue (Triangle (r,_,_)) = r
```

Example 6: Using wildcards in patterns.

such as those illustrated in Example 5.

Patterns may also contain wildcards, `_` which are used as place holders for parts of the pattern that are of no interest in the definition in hand. Using a wildcard is preferable to using some identifier because it explicitly states that the object matched by the wildcard is not used, and therefore does not need to be named. An alternative version of Example 5 that uses wildcards is shown in Example 6.

An alternative to using wildcards in patterns when matching against data structures is to add field names to the data type, which can greatly simplify the pattern expressions. This is illustrated in Example 7. Adding field names to data structures is also useful because well chosen field names can help document the data structures.

Patterns are not confined just to the left hand side of a function, but may also be used in the body of a function, for instance as part of a `let` or `case` expression. Example 8 illustrates this.

Because patterns are so widely used in all areas of functional programming in Haskell it is interesting to investigate how patterns affect a program's complexity,

```
data Colour = Colour {red, green, blue :: Int}
data Shape = Square Colour | Triangle Colour

redValue :: Shape -> Int
redValue (Square c)    = red c
redValue (Triangle c) = red c
```

Example 7: Using field names with data structures in patterns.

```
redValue :: Shape -> Int
redValue sh =
  let
    (r,g,b) =
      case sh of
        Square c -> c
        Triangle c -> c
  in
    r
```

Example 8: Using patterns on the right hand side of functions.

and whether this in turn affects its change history. To do this it is necessary to decide which attributes of patterns might be measured and how these attributes may affect the complexity.

There are several attributes that one might measure from patterns. These are discussed in detail in the following sections, but are briefly introduced case by case now.

- Number of identifiers introduced. Patterns often introduce identifiers into the scope. One way in which this might affect the complexity is to increase the number of identifiers a programmer must know about in order to understand the program code in question. This is described in more detail in Section 4.1.1.
- Number of overridden or overriding pattern variables. Patterns that introduce variables may override existing identifiers, or identifiers in patterns may be overridden by those in a `where` or `let` clause. Overriding identifiers in this manner can be very confusing, and can easily lead to erroneous behaviour when modifying the program. It therefore appears that measuring the number of pattern variables involved in overriding might help to predict potential points of error. This is described in more detail in Section 4.1.2.
- Number of constructors used. Patterns containing constructors are often used to manipulate algebraic data types. Like identifiers in patterns, this increases the number of items a programmer must know about, and therefore may increase the complexity. This is described in more detail in Section 4.1.3.
- 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, for example,

```
-- An alphabet consisting of only 3 characters
data SmallAlphabet = SA | SB | SC

-- An alphabet consisting of 26 characters
data LargeAlphabet = LA | LB | LC | .... | LZ

isSmallA :: SmallAlphabet -> Bool
isSmallA SA = True
isSmallA _  = False

isLargeA :: LargeAlphabet -> Bool
isLargeA LA = True
isLargeA _  = False
```

Example 9: Differences in the normal form of functions.

the position of arguments to a constructor. Because of this uncertainty it was decided that wildcards should be measured to see what effect they have. This is described in more detail in Section 4.1.4.

- Depth of nesting. Nesting of patterns is used quite often, for instance `[(a,b)]` contains the pattern `(a,b)` nested within the pattern `[...]`. However nesting can lead to complicated patterns so the depth of nesting seems to be a good target to measure. This is described in more detail in Section 4.1.5.
- Pattern size. Patterns obviously may be of different sizes, and it is likely that larger patterns are more complex than smaller patterns so pattern size seems an ideal candidate to measure. This is described in more detail in Section 4.1.6.
- Complexity of the normal form of a definition. Another potential indicator of the complexity of a definition containing patterns is the complexity of the normal form in which overlapping patterns have been removed, making all patterns exclusive. For instance consider the two functions in Example 9.

Although the `isSmallA` and `isLargeA` functions look to have similar complexities, it is worth considering the final pattern in each of these functions. In the `isSmallA` function the final pattern expression will be executed for two of the `SmallAlphabet` constructors, whereas in the `isLargeA` function the final pattern expression will be executed for twenty five of the `LargeAlphabet` constructors. This may make the latter function more complex than the `isSmallA` function.

Such a situation could be detected by examining the normal form of the definitions, however this requires the use of a type system which the framework used for these experiments currently lacks. For this reason this particular measure will not be considered any further in this study, and is left as future work.

These pattern measures cover a large range of the common usage of patterns in Haskell programs, and most of these measures cover a single attribute of patterns and therefore can be thought of as “atomic”.

For most of the pattern measures one would expect increased values to indicate increased complexity. The exception to this is the “Number of wildcards” measure, where one might expect increased values to indicate reduced complexity.

In the following sections these attributes will be analysed to see what effect they have on the complexity of patterns used in the case study programs described in Chapter 3. Specifically the rest of this chapter will be arranged in the following manner.

- Section 4.1.1 explores the “Number of pattern variables” measure.
- Section 4.1.2 investigates the “Number of overridden or overriding pattern variables” metric.
- Section 4.1.3 examines the “Number of constructors in pattern” metric.
- Section 4.1.4 discusses the “Number of wildcards in pattern” measure.

- Section 4.1.5 explores the effect of the depth of nesting of patterns upon the complexity.
- Section 4.1.6 examines the “Pattern size” metric.
- Section 4.1.7 discusses how the pattern metrics interact, and looks at the correlation between metric values.
- Section 4.1.8 presents the conclusions that can be drawn from measurements made on pattern expressions.

4.1.1 Number of pattern variables

Patterns are most often used to introduce variables. An example might be a variable representing a parameter of a function, or a pattern expression which matches a tuple with two elements, introducing two variables. It is not immediately clear how increasing the number of variables in this sort of pattern might affect the subjective complexity, that is, the ease of understanding the pattern. One hypothesis is that the presence of more pattern variables requires the programmer to comprehend more objects in order to understand the section of code in question, and so increases the complexity.

Having implemented a metric to measure this attribute, and applied it to the case study programs described in Chapter 3, the first impression was that a large proportion of the functions in the case study programs used patterns that contained pattern variables. The Peg Solitaire program used pattern variables in 199 of its 234 functions (85%), while the Refactoring program use pattern variables in 379 of its 540 functions (70%). This demonstrates how central patterns are to Haskell.

The metric values were correlated against the number of bug fixes, as described in Chapter 3. The correlation coefficient, shown in Table 9 in Appendix B, for this metric for the Peg Solitaire program was 0.0209, which is not statistically significant. The Refactoring program has a statistically significant correlation

value of 0.5927, showing a fair degree of correlation with the number of changes.

These results are confusing because the value for the Peg Solitaire program seems to suggest that the number of pattern variables has little effect upon the complexity, while the result for the Refactoring program does provide support for the hypothesis. Therefore it is possible to make the following tentative observation.

Observation 4.1.1 *The presence of a high number of pattern variables may indicate increased subjective complexity.*

4.1.2 Number of overridden or overriding pattern variables

Patterns that introduce variables may override existing identifiers, and identifiers in patterns may be overridden by those in a `where` or `let` clause. Overriding identifiers in this manner can be confusing, for instance in Example 10 it is not immediately clear which `a` is being used at any point in the function.

This type of overriding can be particularly troublesome if the conflicting definitions have the same type, which prevents the compiler from indicating possibly erroneous behaviour when modifying the program. For instance, consider the (contrived) `area` function in Example 10. If the `a` in the `where` clause is renamed, the function will compile with no errors, but will give incorrect results unless the `a` after the `=` is also changed.

It therefore appears that measuring the number of pattern variables involved in overriding might predict potential points of error. However, the implementation of the metric used in this work does not yet contain a type system, and so for this work it is not possible to check if the variables involved in the overriding have the same type, which would be interesting to factor into the measurement, and so this implementation of the metric treats all overriding in the same manner, regardless of the types involved. However it would be possible to re-implement the metric using a framework such as Programatica [41] to make use of type information in

```
data Shape
  -- A triangle described by its angles and lengths
  = Triangle {
    a      :: Float,
    angleA :: Float,
    b      :: Float,
    angleB :: Float,
    c      :: Float,
    angleC :: Float
  }

-- Area of a shape
area :: Shape -> Float
area (Triangle a aA b bB c cC) = a
  where
    a = 1/2 * (b * c * sin(aA))
```

Example 10: Overridden pattern variables.

the measurement.

The metric developed to measure this attribute was applied to the case study programs and the results showed that overriding of pattern identifiers occurred in only 9 out of the 234 functions of the Peg Solitaire program, and in 51 of the 540 functions that make up the Refactoring program.

The correlation results for the Peg Solitaire program, shown in Table 9 in Appendix B, do not exhibit any statistically significant correlation with the number of bug fixing changes, probably because of the low number of instances of overriding occurring in the program. However, the results from the Refactoring program show a statistically significant correlation of 0.3731. Thus the following observation can be made.

Observation 4.1.2 *Increasing the number of overridden pattern variables increases the subjective complexity of the function.*

One might claim that overriding pattern variables is an inherently undesirable programming paradigm and as such this metric may be a useful tool to highlight

such occurrences, so that they may be corrected by the programmer. Such a tool might combine this metric with a visualisation technique such as the pixel representation to quickly highlight such code (See for instance Sections 6.1 and 6.5 of Chapter 6), or form part of an automated refactoring tool such as HaRe [60].

4.1.3 Number of constructors

Patterns are commonly used in Haskell programs for manipulating algebraic data types by matching against constructor names. A possible metric is to count how many constructors are used in a pattern. There are two interesting and opposing hypotheses about the effect of this attribute on the subjective complexity of a function.

One hypothesis is that using more constructors in a pattern requires more objects to be understood in order to comprehend the pattern, therefore increasing the complexity. The alternative hypothesis is that if constructor names are chosen well they are descriptive, and therefore add an element of documentation to the pattern, possibly reducing the subjective complexity of the pattern.

The metric data for this attribute showed that the Peg Solitaire program used patterns that contained constructors in 33 of its 234 functions, while the Refactoring program used such patterns in 138 of its 540 functions.

The results in Table 9 in Appendix B show once more that there is no statistically significant correlation for the Peg Solitaire program, but that there is a small positive correlation of 0.3645 for the larger Refactoring program.

The correlation value for the Refactoring program suggests that the first hypothesis is more likely to be true, although the lack of correlation for the Peg Solitaire program makes it difficult to be certain.

However, a possible explanation for the difference in correlation values between the two programs might be the nature of the Refactoring program, which uses complicated algebraic data structures to represent abstract syntax trees of Haskell programs. Because these structures are complicated it is easy for errors to occur in

their use, which might contribute to the positive correlation between the number of constructors and the number of changes. This suggests that combining the number of constructors with some measure of the complexity of the corresponding data structures could produce a more accurate version of this metric.

In the smaller data structures of the Peg Solitaire program the use of well named constructors can add documentation to the source code, thereby making the patterns in the code less complex, which might explain the (statistically insignificant) negative correlation seen in the Peg Solitaire program.

4.1.4 Number of wildcards

Wildcards may initially lead one to believe that they make understanding a pattern simpler because wildcards indicate items that may not need to be considered, but it is also possible that thinking of wildcards in this manner could be misleading.

Consider Example 11. In the `redValue` function it is reasonably straightforward to remember the position of the field we are interested in, namely the red value. However in the `lineGreenValue` function it is much harder. In such an example it is very easy to get the `g` pattern variable in the wrong place, and because the fields are all of the same type the compiler is unable to indicate any errors if the variable is in the wrong place.

Because of this it is reasonable to expect that larger numbers of wildcards might *increase* the complexity of the pattern. To explore the impact of wildcards upon complexity a small metric was created to count the number of times wildcards were used in a pattern. The metric showed that wildcards were used in 23 of the 234 functions in the Peg Solitaire program and 162 of the 540 functions of the Refactoring program.

An alternative metric might calculate the ratio of the number wildcards to the number of variables, which would give an indication of whether a pattern is dominated by variables or by wildcards. However, only the basic count of the number of wildcards is used in this work.

```

-- A simple shape with a size and a colour in RGB values
data Shape
  = Triangle
    Int Int Int -- Length side A, B and C
    Int Int Int -- Fill colour RGB values

-- A shape with a size, a position, a border thickness,
-- and RGB colours for the border and for the fill colour.
data AdvancedShape
  = AdvancedTriangle
    Int Int Int -- Length side A, B and C
    Int Int     -- X and Y position
    Int         -- Line width
    Int Int Int -- Line colour RGB values
    Int Int Int -- Fill colour RGB values

redValue :: Shape -> Int
redValue (Triangle _ _ _ r _ _) = r

lineGreenValue :: AdvancedShape -> Int
lineGreenValue (AdvancedTriangle _ _ _ _ _ _ g _ _ _ _) = g

```

Example 11: Complications of using wildcards in patterns.

Table 9 in Appendix B shows that once again the results for the Peg Solitaire program show no statistically significant correlation, while the Refactoring program shows a correlation value of 0.3572. This seems to suggest that increasing the number of wildcards can increase the complexity, and so the following observation can be made.

Observation 4.1.3 *Large numbers of wildcards may indicate areas of increased subjective complexity.*

This seems to contradict the generally perceived wisdom that wildcards reduce complexity. However, an explanation that might account for this is that the functions that use patterns with wildcards may be manipulating complex data structures, and so changes may be due to the complexity of the data structures rather than the appearance of wildcards.

4.1.5 Depth of nesting

When measuring the depth of nesting one must consider how to measure the depth of nesting of 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 all the nested patterns.

Taking the maximum of the depths measures only how deeply nested the pattern is, while taking the sum of the depths effectively measures how much nesting is taking place, which may be more accurate. However it may be that taking the sum of the depths will actually be measuring the size of the pattern, in which case one would see a strong correlation between those two measures.

Consider Example 12. In this example the patterns in `pattern1` and `pattern2` would have the same maximum depth of nesting, both having one level of nesting, but would each have a different sum of depths, with the pattern in `pattern2` having a larger value than that in `pattern1`. Thus it would seem that the sum of the depths is more discriminating than the maximum depth.

```
type Tuple = (Int, Int)
type TupleList = [Tuple]

pattern1 :: TupleList -> Int
pattern1 [(a,b)] = a + b

pattern2 :: TupleList -> Int
pattern2 [(a,b),(c,d),(e,f)] = a + b + c + d + e + f
```

Example 12: Nested patterns.

Metrics for both measures of depth were written and the correlation results are shown in Table 9 in Appendix B. The Peg Solitaire program shows correlation values of 0.0582 and 0.071 for the maximum and sum depth measures respectively. These values are not statistically significant.

The Refactoring program showed significant correlation values of 0.4208 and 0.5692 for the maximum and sum respectively. This seems to show that taking the sum of the depths does indeed give a better prediction of the likelihood of bugs occurring than taking the maximum depth. These results allow the following observation to be made.

Observation 4.1.4 *Increasing the amount of nesting of patterns increases the complexity of expressions.*

4.1.6 Pattern size

The previous measures on patterns have concentrated on specific components of patterns and so it may be interesting to make a statement about a pattern as a whole. One simple way to do this is to talk about its size.

The size of a pattern could be measured in a number of ways. For instance, one could count the number of components in the abstract syntax tree, or one might take the depth of abstract syntax tree as a measure of its size. For the purpose of exploring the effect of pattern size on the number of changes, the number of

nodes in the abstract syntax tree of the pattern was chosen as the size measure.

The metric results in Table 9 in Appendix B show that the Peg Solitaire program had no significant correlation, while the Refactoring program had a correlation of 0.5423, which seems to suggest the following observation.

Observation 4.1.5 *Increasing the size of a pattern tends to increase its subjective complexity.*

4.1.7 Interaction of pattern attributes

In the previous sections several metrics have been presented, each of which measures a specific attribute of pattern expressions. On their own, none of these metrics provide a particularly high correlation with the number of changes. The intriguing question that arises from this is whether these attributes interact in any way to form a more accurate measure.

The first step to investigating any possible interactions is to see if any of the measures are correlated with each other. This can be done using a correlation matrix. The correlation matrices for the pattern measures taken from the Peg Solitaire and Refactoring programs are shown in Table 14 in Appendix C.

The correlation matrix for both programs show that several of the pattern metrics are strongly correlated. In particular, it appears that “Number of pattern variables” ($p1$), “Sum of depth of patterns” ($p2$) and “Pattern size” ($p6$) are strongly correlated in both programs. In the Refactoring program “Number of constructors in pattern” ($p5$) is also strongly correlated with this group. One explanation for this is that these measures may all be measuring the size of a pattern.

Looking at the correlation matrix again it appears that apart from the cluster of strongly correlated measurements the other metrics have minimal cross-correlation.

It is interesting to see how the metrics might be combined to provide a higher correlation with the number of changes. This can be done using a regression

analysis, however when variables in a regression analysis are strongly correlated the result can be inaccurate and so it is advisable to replace the strongly correlated metrics with a representative metric.

For this work “Sum of depth of patterns” ($p2$), for the Peg Solitaire program, and “Number of pattern variables” ($p1$), for the Refactoring program, were chosen as the representatives of the cluster of correlated metrics. These were chosen because they have the highest correlation with the number of changes out of the metrics that form the clusters.

Because the results for different metrics may not be measured in the same scales it is important to normalise them so that they all have equal weighting in the regression analysis. This is achieved by performing the analysis on the “z-scores” of the metric results. The z-scores normalise the metric values such that the z-scores have a mean value of zero and a standard deviation of one. The results of the regression analysis are shown in Table 21 in Appendix D.

The results of the regression analysis show the multiple correlation coefficient, R , to be 0.1584 for the Peg Solitaire program, which is not statistically significant, and 0.6015 for the Refactoring program. These values are higher than the highest individual correlation values for each program, although in the case of the Refactoring program, only marginally.

These results suggest that there is only a small amount of interaction between the metrics. The results from the Refactoring program show that 36% of the variance can be explained by the metrics so it is worth examining this regression analysis in more detail.

The coefficients shown in Table 21 in Appendix D for the regression analysis of the Refactoring program shows that the largest contribution comes from the “Number of pattern variables” ($p1$) metric. The correlation value for this metric is only slightly less than the multiple correlation coefficient. This suggests the following observation.

Observation 4.1.6 *The main influence on the subjective complexity of a pattern*

is its size, as indicated by the number of pattern variables.

It seems that for pattern attributes the other measurements do not significantly increase the correlation with the number of changes, although this may be caused by the relatively small number of occurrences of those attributes in the Refactoring program.

4.1.8 Summary

In this section it has been shown that there are several attributes of pattern expressions that can be measured. Each attribute measures a distinct component of a pattern expression that might add to a pattern's complexity. These attributes can therefore be thought of as atomic attributes. Analysis of these atomic attributes shows that some of the attributes are strongly correlated. These strongly correlated attributes appear to be measuring the size of a pattern in various ways.

The two case studies highlight the differing results that can occur for the measurements correlations in differing contexts. In the Peg Solitaire program the “Number of constructors in pattern” (*p5*) measurement had a slight negative correlation, while in the Refactoring program it had a positive correlation. This may be due to the differing uses of constructors in the two programs. In the Peg Solitaire program the constructors are used in simple data types that have little nesting, and as such the naming of the constructors helps to document the code. By contrast, the Refactoring program uses large, complex mutually recursive nested data types to represent parse trees. In this case the constructor names are sometimes generic and add little documentation to the code. The complex nature of the data types makes it easy to introduce errors, and hence the metric has a positive correlation.

Performing regression analysis suggests that the largest influence on the correlation with the number of changes is the size of the pattern.

For some of the metrics, the type of program from which a measurement is

taken may affect the correlation of the metric. For instance the “Number of constructors in pattern” (*p5*) measurement appears to exhibit different correlations between programs with complex data types, such as the Refactoring program, and programs with simple data types, such as the Peg Solitaire program. Because of this it seems that combining such a metric with some measures of the corresponding data type could produce a greater correlation, and therefore perhaps a more accurate prediction.

4.2 Measuring the Distance Between Declarations and Their Use

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 declared and where they are used and one might hypothesise that the larger the distance between where something is used and where it is declared, the greater the chance that an error will occur in the way one uses that item. It therefore seems that attempting to measure that distance may indicate potentially problematic areas of a program.

The distance between a declaration and where it is used can be measured in a number of ways, most of which fit into one of two categories. Those that measure the distance in the source code, such as the number of lines of code, which one might term “spatial” distance measures, and those that measure distance semantically, such as counting the number of new scopes introduced, which one might term “conceptual” distance measures.

Because a function may call several different functions, there will be several different distance measures, one for each called function, which must be aggregated in some way to produce a single value for the calling function.

There are numerous ways one might choose to aggregate distance values and for this work four methods were chosen, taking the sum of the distances, taking the maximum distance, taking the product of the distances and taking the mean distance. However, when these methods were used the product method often produced very high values (greater than 100 digits) which proved to be difficult to process and so the product method was discarded from this work.

There are numerous ways one might choose to measure the distance between the use and the declaration of identifiers, four of which were chosen for this work. These are discussed in detail later, but are briefly introduced case by case now:-

- Number of scopes (Figure 21). One method of measuring the distance is to count the number of scopes that one must examine to find the declaration

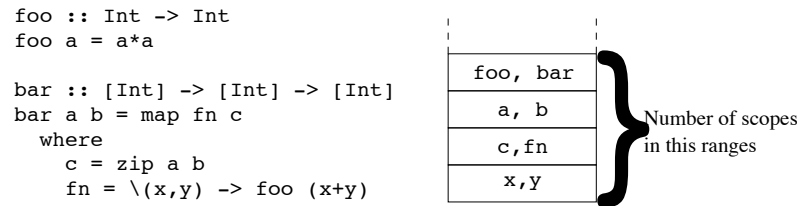


Figure 21: Example of measuring distance by the number of scopes for the function `foo`.

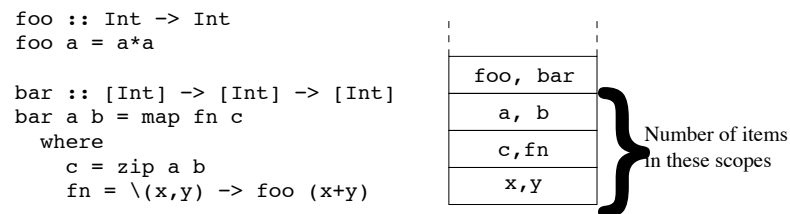


Figure 22: Example of measuring distance by the number of declarations brought into scope for the function `foo`.

of the identifier being called. This gives a “conceptual” distance measure which might indicate how complex the name-space of the program is at a particular point in the code. This method is discussed in more detail in Section 4.2.2.

- Number of declarations in new scopes (Figure 22). This is another “conceptual” distance measure that is an extension to the previous measure. In this measure the number of identifiers declared in the newly introduced scopes is counted. This gives an indication of how “busy” the name-space is. This method is discussed in more detail in Section 4.2.3.
- Number of source lines (Figure 23). This is a straightforward “spatial” distance measure counting the number of source code lines between the declaration and use. The implementation used in this work counts comment lines and whitespace as part of the source code lines, although with a different parser it would be possible to exclude comment lines from the counts.

```

1: foo :: Int -> Int
2: foo a = a*a
3:
4: bar :: [Int] -> [Int] -> [Int]
5: bar a b = map fn c
6:   where
7:     c = zip a b
8:     fn = \(x,y) -> foo (x+y)

```

Number of source lines between these two points, seven in this case.

Figure 23: Example of measuring distance by the number of source code lines for the function `foo`.

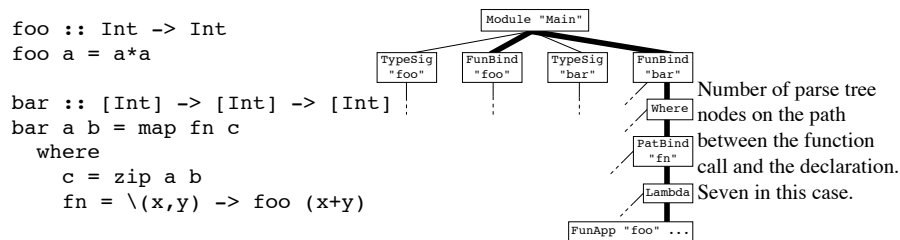


Figure 24: Example of measuring distance by the number of parse tree nodes for the function `foo`.

This method is discussed in more detail in Section 4.2.4.

- Number of parse tree nodes (Figure 24). This is an extension of the previous measure. Source code lines may contain differing amounts of code, so this measure counts the number of parse tree nodes on the path between two points in the parse tree. This is a “spatial” measure like the “Number of source lines” measure but gives a more consistent measure of the amount of code produced between the use and the declaration. This method is discussed in more detail in Section 4.2.5.

Something worth considering when discussing these metrics is how reorganising the source code of a program might affect the measures. Spatial measures, such as the “Number of source lines” measure, are likely to change significantly if functions in the source code are reordered. However, conceptual or semantic methods may not be so affected, for instance the “Number of new scopes” measure should be unaffected by reorderings.

4.2.1 Distance metrics for multi-module programs

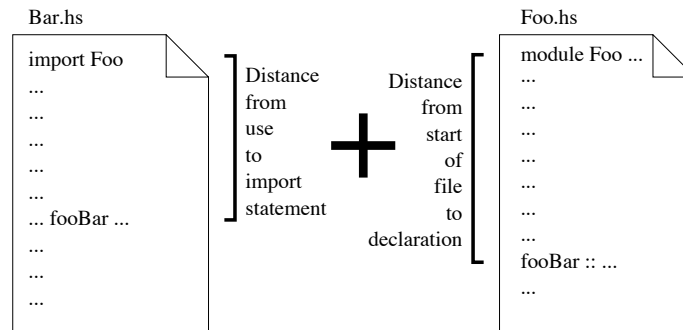
It is rare to find a Haskell program that consists of just a single file. Typically programs will be divided into modules, with each module contained in a separate file. The problem arising from this is how distance across module boundaries should be measured.

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

This method is implemented in the metrics used in this study, although an alternative method might consider the case where a function `foo` is imported into a module `A` from a module `B` and then re-exported. This forms a chain of imports that may make the function `foo` harder for the programmer to find. The length of this chain of imports could be factored into the cross-module distance measure in some way, but this is not yet implemented in the metrics used for this work.

When measuring distance in the source code the way to measure distance across module boundaries is less clear. One crude method would be to assign a constant distance measure to items defined in external files. This would indicate that such items are “further away”, but would give no indication of how much further.

A solution that attempts to give an indication of how much code the programmer might have to look through to find the declaration of a function is 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 below.



This method gives an indication of how much code the programmer might have to look through, finding the module the identifier is imported from, then finding the identifier in the imported module. There are other variations of this method, one might measure only the distance in the imported module, or only the distance to the import statement which brings the identifier into scope, for instance. Although the method illustrated here is used for this work, the best way to measure distance across module boundaries, particularly for spatial distances which are less obvious than conceptual distances, is largely an open question.

In the following sections these measures will be analysed to see how well they might correlate with the number of changes in the case study programs described in Chapter 3. This section will be organised like so

- Section 4.2.2 examines measuring distance by the number of scopes.
- Section 4.2.3 discusses measuring distance by the number of declarations introduced into scope between the use and the declaration.
- Section 4.2.4 explores measuring distance by the number of source code lines.
- Section 4.2.5 investigates measuring distance by the number parse tree nodes.
- Section 4.2.6 discusses the ways in which the various distance measures may interact, and examines the correlation between the various metrics.
- Section 4.2.7 presents the conclusions that can be drawn from the distance measures.

4.2.2 Distance by the number of scopes

One of the conceptual ways to think of the distance between a declaration and where it is used is to consider how many scopes are introduced in between. Measuring the number of scopes indicates how deeply nested the usage is, and how complex the name-space is. This measure may therefore provide an estimate of how widely one must look to fully understand the behaviour of the calling function.

Looking at the correlation between the various combinations of this measure and the number of changes, shown in Table 10 in Appendix B, it can be seen that taking the average distance to all used identifiers gives the highest correlation for the Peg Solitaire program, while the Refactoring program has its highest correlation when taking the sum of the distances. However, the correlation values for the Peg Solitaire program are not statistically significant at the 5% level.

Examining the result from the Refactoring program shows that the interesting measurements are the sum and maximum measures. These show very similar correlation values, 0.632 for sum and 0.6006 for maximum. This suggests that as more functions are called, meaning the sum measure will rise, the distance to the furthest declaration will tend to rise, meaning the maximum measure will increase.

4.2.3 Distance by the number of declarations in scope

An extension to measuring the distance by the number of scopes is to count how many declarations have been introduced into the name-space by any new scopes between the use and the declaration, not including the scope containing the declaration. This gives an idea of how “busy” the nested scopes are. This technique is illustrated in Figure 22 on Page 124.

Looking at the results for these measurements in Table 10 in Appendix B it seems that counting the number of declarations in the scopes, rather than simply counting the number of scopes, does not significantly alter the correlation of the metric with the number of changes, indeed the results from the Refactoring

program show that this *decreased* the correlation. This leads to the following tentative observation, which could be used to help target refactorings that lift definitions to “higher” scopes.

Observation 4.2.1 *It is the nesting of local definitions that complicates program structure rather than the number of local definitions.*

4.2.4 Distance by the number of source lines

So far only “conceptual” distance measures have been considered. The alternative way to measure distance is to consider the physical or “spatial” distance in the source code. This can be done in several ways, the most obvious of which is to measure the number of source code lines between the use and the declaration of an identifier. A metric was implemented to measure distance using this method and the results of applying this metric are presented in Table 10 in Appendix B.

Looking at these results the immediate observation is that the Peg Solitaire program has negative correlations while the Refactoring program has positive correlations, and that the Refactoring program has significantly higher correlation values. However, the measurements from the Peg Solitaire program are not statistically significant.

The opposite correlation trends of negative for the Peg Solitaire program and positive for the Refactoring program, may be explained by the `Main` module of the Peg Solitaire program. This module accounted for 38 of the 148 changes, despite having a maximum size of only 27 lines of code. A large number of the changes in this module were bug-fixes and refactoring changes occurring after the user interface of the program changed from a purely text-based interface to a dual text-based and graphical interface. This module may be a cause of the negative correlation trends for the Peg Solitaire program because this small module had a greater number of changes than many of the larger modules. However it should be remembered that the values for the Peg Solitaire program are not statistically significant, and as such one should be careful not to draw too many conclusions

from these results.

4.2.5 Distance by the number of parse tree nodes

A common problem of measuring program code size by the number of source code lines is that source lines may have varying amounts of program code. This problem may also affect distance measures that use the number of source lines. An alternative way to measure “spatial” distance is to count the number of parse tree nodes on the path between the point on the parse tree where the identifier is declared and the point where it is used, including any sub-trees rooted at any of the nodes on the path. This method includes the sizes of any intervening definitions between the calling function and the declared function, and may give a more accurate measurement of the amount of program code between the declaration and use than is obtained from counting the number of source code lines.

The correlation results for this measure, shown in Table 10 in Appendix B, indicate that for the Refactoring program there is very little difference in correlation between measuring distance by source line and measuring distance by the number of parse tree nodes. This might imply that on average the amount of program code per source line is reasonably consistent. The results for the Peg Solitaire program are not statistically significant.

4.2.6 Interaction of distance measures

Having presented some ways to measure distance between declarations and where they are used, it is important to ask how these attributes may interact. This is done for a larger selection of programs in Section 5.1, but it is interesting to look at this for the two case study programs here. The first step towards this is to produce a correlation matrix for the metric results. These are shown in Table 15 in Appendix C.

The correlation matrices for the two case study programs each show two clusters of strongly correlated measures. The clusters consist of the same metrics in

each programs, with the first cluster consisting of

“Distance by the sum of the number of scopes” (*d1*)

“Distance by the sum of the number of declarations in scope” (*d4*)

“Distance by the sum of the number of source lines” (*d7*)

“Distance by the maximum number of source lines” (*d8*)

“Distance by the average number of source lines” (*d9*)

“Distance by the sum of the number of parse tree nodes” (*d10*)

“Distance by the maximum number of parse tree nodes” (*d11*)

“Distance by the average number of parse tree nodes” (*d12*)

and the second cluster consisting of

“Distance by the maximum number of scopes” (*d2*)

“Distance by the average number of scopes” (*d3*)

“Distance by the maximum of the number of declarations in scope” (*d5*)

“Distance by the average number of declarations in scope” (*d6*)

The first cluster reaffirms that there is little difference between measuring distance by the number of source lines or by the number of parse trees 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 in the source code between declarations and where they are used tend to increase.

However, it may also be that programmers might tend to write functions close to where they are used in the source code. In future work it may be interesting to see if it is possible to permutate the source code of a program to produce a program which has minimal distance measures, or indeed to discover if a program is already “minimal”.

The second cluster shows that the distance measured by the maximum or average number of scopes or declarations in scopes is not strongly correlated with

measuring the sum of the number of scopes or declarations in scope. One explanation for this might be that the declarations, or variables, used in a function are generally similar distances 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 between functions because it also measures how many declarations are used.

Having looked at the correlation matrix for the distance measures it is possible to replace the clusters of strongly correlated metrics with single representative metrics and perform a regression analysis on the z-scores of the metric values to see if combining the measurements can increase the correlation with the number of changes.

The metric with the highest correlation with the number of changes was chosen from each cluster as the representative of that cluster. The chosen representative metrics are “Distance by the sum of the number of source lines” (*d7*) and “Distance by the average number of declarations in scope” (*d6*) for the Peg Solitaire program and “Distance by the sum of the number of scopes” (*d1*) and “Distance by the maximum number of scopes” (*d2*) for the Refactoring program. The results of the regression analysis are shown in Table 22 in Appendix D.

The regression analysis shows a multiple correlation coefficient (R) of 0.1957 for the Peg Solitaire program and 0.6829 for the Refactoring program, both of which are statistically significant. These are slightly higher than the correlation values of any of the individual measurements which suggests that there is a little interaction between them. However the regression for the Peg Solitaire program still only accounts for a small proportion of the variance of the number of changes.

The values for the Refactoring program show that distance measures can explain nearly 50% of the variance of the number of changes in that program, indicating that it may be possible to obtain good predictions from these measures. This suggests that these metrics might be usefully combined in a visualisation tool which could highlight functions with particularly high values of these metrics.

Looking at the coefficients from the analysis of the Refactoring program it seems that “Distance by the sum of the number of scopes” ($d1$) and “Distance by the maximum number of scopes” ($d2$) make reasonably even contributions to the correlation with the number of changes. This is probably because the two measures are partially correlated and therefore combining them adds only a small amount of information.

Interestingly, the coefficients from the analysis of the Peg Solitaire program show that the “Distance by the sum of the number of source lines” ($d7$) metric has a negative coefficient. This is interesting because it suggests that if the functions used are a long way away in the source code they are 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 suggest that when measuring attributes across modules, the behaviour of the attributes might be different than when considering them inside a single module.

4.2.7 Summary

This section has presented a selection of metrics that measure the distance between declarations and where they are used in a variety of ways. Performing these measurements has revealed a number of discrepancies between results for the Peg Solitaire and Refactoring programs, and possible explanations for these have been presented.

Investigating the cross correlation of the measures has shown that a large proportion of the measurements are strongly correlated, suggesting they are measuring the same or similar attributes.

Examining the regression analysis of the measures suggests that the attribute that contributes most to the complexity is the semantic, “conceptual”, distance in the program code, rather than the “spatial” distance. This suggests that errors are more likely to be introduced into programs when there are lots of nested scopes, although since there is a strong correlation between the metrics it is not clear from

these results if this is an artifact of a different attribute, such as program size.

The regression analysis also showed that in some circumstances the further away a called function is defined in the source code, the less likely an error is to occur. This implies that cross-module function calls should perhaps be treated differently to intra-module function calls.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

Example 13: An example of trivial recursion.

4.3 Measuring Attributes of Recursive Functions

The Haskell language does not provide an explicit loop construct, instead programs use recursion to achieve looping. In many cases such recursive functions can be optimised by the compiler into a loop. Because recursive functions are the only way to program loops in Haskell they are used extensively, and so it is interesting to look at what can be measured about recursive functions. This section considers only explicit recursion and does not consider other mechanisms for achieving looping, such as the use of `foldr` and other higher-order functions, which can be considered as “black boxes” that can be trusted. Instead this section concentrates on the visible structure of the code.

There are two ways in which a function may be recursive, which one might term “trivial” and “non-trivial” recursion. In trivial or “direct” recursion the function directly calls itself, as shown in Example 13.

Non-trivial or “indirect” recursion is potentially more complex. In a non-trivial recursive function, `foo`, a second function, `bar`, is called which in turn calls `foo`, producing a cyclic callgraph, or *strongly connected component*, although the strongly connected component may be larger in a real program. This type of recursion is demonstrated in Example 14.

Thus non-trivial recursion is much less obvious in the source code than trivial recursion, and therefore may affect the complexity of the functions involved. It is also worth noting that functions may have more than one execution path that cause recursive behaviour, and so a function may contain both trivially recursive execution paths and non-trivially recursive execution paths.

By generating callgraphs for a program it is possible to analyse the way in


```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (a:xs) = qsortLE a xs ++ a ++ qsortG a xs

qsortLE :: Int -> [Int] -> [Int]
qsortLE a xs = qsort [x | x <- xs, x <= a]

qsortG :: Int -> [Int] -> [Int]
qsortG a xs = qsort [x | x <- xs, x > a]
```

Example 14: An example of non-trivial recursive behaviour.

which functions may be recursive. There are several attributes that one might wish to measure from recursive functions. These will be briefly introduced here and then explained and analysed in detail later in this section.

- Binary measure of recursion. It is not always obvious if a function is recursive because its cyclic callgraph might 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 measure. This method is discussed in more detail in Section 4.3.1.
- Number of recursive call paths. A call path is a chain of function calls. e.g. Function `a` calls function `bar` which calls function `c`. If the call path is cyclic it indicates that the functions involved are recursive. A function may have more than one recursive call path and so functions with greater numbers of recursive paths may be more complex. This method is discussed in more detail in Section 4.3.2.
- Number of trivial recursive call paths and number of non-trivial recursive call paths. Trivial recursive paths, those where a function directly calls itself, may be easier to understand than non-trivially recursive paths, where a function calls a second function that calls the first, for instance, and so it is interesting to count each type of recursive path separately. This method is discussed in more detail in Section 4.3.3.

- Sum or Product of the lengths of the recursive paths. As the recursive paths within a function have a length it is interesting to measure these lengths, because it may be that longer path lengths indicate increased complexity. If a function has more than one recursive path one must decide how to combine the lengths of the paths. For this work we have chosen to take the product and the sum of the lengths because these methods take account of how many recursive paths there are as well as the lengths, although one might also choose to take the maximum, or the average lengths, for instance. When investigating distance measures in Section 4.2 the use of products to combine metric values caused problems with large values being produced. However, no such problems were encountered in the use of products for the recursion metrics because the values being combined were much smaller than those of the distance measures.

This method is discussed in more detail in Section 4.3.4.

In the following sections these measures will be examined in more detail, using the case study programs described in Chapter 3. There are also other measures that can be taken from the callgraph of a program that measure attributes of recursion, such as the size of any strongly connected components. Measurements of callgraph attributes are discussed in more detail later in Section 4.4.

The rest of this section is structured in the following manner.

- Section 4.3.1 looks at the binary measure of recursion.
- Section 4.3.2 investigates measuring the total number of recursive paths in a function.
- Section 4.3.3 examines ways of measuring the number of trivial and non-trivial recursive paths separately.
- Section 4.3.4 studies the lengths of the recursive paths present in a function.
- Section 4.3.5 discusses the possible interactions between the recursion measures, and in particular the degree of cross-correlation between them.

- Section 4.3.6 presents the conclusions that can be drawn from this study of these recursion metrics.

4.3.1 A binary indication of recursion

Sometimes it can be difficult to recognise that a function is in fact recursive because its strongly connected component may be quite large. Because of this it may be useful to have an indication that a function is recursive without actually knowing how that recursion occurs. To do this a metric was developed that returns either one or zero, indicating whether the function is recursive or not. The correlation values for this “Binary recursion” ($r1$) metric are shown in Table 11 in Appendix B.

The interesting observation from the results is that, for the Peg Solitaire program, this very simplistic metric appears to give the highest correlation of any of the recursion metrics. This suggests that it is more important to know that a function is recursive than to know exactly what makes the function recursive. Although it is not significant at the 5% level it is significant at the 10% level.

The results for the Refactoring program are less striking. There are only 21 out of the 540 functions that make up the Refactoring program that are recursive. Most of these recursive functions are small and have not been changed much, resulting in the low, statistically insignificant, correlation value seen.

Recursive callgraph paths can be either trivial or non-trivial. Closer inspection of the table shows that, for the Peg Solitaire program, there is a very similar correlation value for the “Number of trivial recursive paths” ($r3$), suggesting that the binary metric is mainly measuring the number of trivial recursive paths. This was confirmed when the raw data from the metrics was inspected, which showed that there were 28 functions with trivial recursive paths but only 7 functions with non-trivial recursive paths.

This is also shown in Section 5.2.2 of Chapter 5 which studies the values of the recursion metrics on a wider selection of programs, showing that typically the amount of non-trivial recursion is much lower than the amount of trivial recursion.

```
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)
```

Example 15: Examples of multiple recursive paths in functions.

4.3.2 Number of recursive paths

Functions may have more than one recursive path, so may be more complex if they have a greater number of recursive paths. For instance, consider the functions in Example 15.

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. A higher number of recursive paths in a function might indicate greater complexity. Although it is not entirely clear whether the recursion in `fib` is more complex or less complex than that in `bar`.

It is important to consider the subtle difference between counting the number of recursive paths and counting the number of recursive calls. Consider the contrived functions in Example 16. In the function `altCaseStr` there is only a single recursive call, to `altStr` which in turn calls `altCaseStr` again, creating the recursion. However, the execution path for that recursive call may take one of two possible paths, depending on whether the first character of the string passed to `altStr` is a lowercase character or not. Hence, `altCaseStr` has two *recursive paths* but has only a single *recursive call*. It is also worth noting that `altStr` has

```
-- Alternate case of characters in string,  
-- e.g. altCaseStr "hello" -> "hElLo"  
altCaseStr :: String -> String  
altCaseStr [] = []  
altCaseStr (c:cs) = toLower c : altStr cs  
  
altStr :: String -> String  
altStr [] = []  
altStr (c:cs)  
  | isLower c = toUpper c : altCaseStr cs  
  | otherwise = c : altCaseStr cs
```

Example 16: Example of the difference between recursive paths and recursive calls in functions.

both two recursive paths and two recursive calls.

This shows that by counting the number of recursive paths, rather than the number of recursive calls, all functions that make up the recursive behaviour should have the same metric value, since they will all have the same complexity of recursive behaviour. If on the other hand one were to measure the number of recursive calls, different metric values, and therefore different indications of the complexity of the recursive behaviour, might be obtained depending upon which function is being measured at the time.

One may speculate that the greater the number of recursive paths, the more the programmer must comprehend to understand the behaviour of the function. A metric was written to perform this measurement and the correlation values are shown in Table 11 in Appendix B.

The correlation value for the Peg Solitaire program suggests that the number of recursive paths does not affect the subjective complexity very much. Because there is no statistically significant correlation for this measure it is difficult to say anything concrete about the results for this program.

The correlation value for this measurement from the Refactoring program is the same as that of the binary recursion measure. This is because all the recursive functions in the Refactoring program have only a single recursive path, and so this

measure will produce identical results to the binary recursion metric. It is unclear whether this occurrence is specific to the Refactoring program or whether it is a general trend, however further studies, discussed in Section 5.1 of Chapter 5, suggest that this occurs in a number of other programs and therefore may not be unusual.

4.3.3 Number of trivial and non-trivial recursive paths

Previously the difference between trivial and non-trivial recursive paths was introduced. Trivial recursive paths are those in which there is only one function call, e.g. `foo` calls itself, while non-trivial recursive paths contain other function calls, e.g. `foo` calls `bar` which calls `foo`.

It is conceivable that non-trivial recursive paths may have a greater effect upon the subjective complexity, and therefore perhaps the number of changes, than trivial recursive paths because they are harder to recognise in the source code. To see if this is the case, metrics were written to calculate the number of trivial and non-trivial recursive paths and the results from each were correlated with the number of changes. The correlation results are shown in Table 11 in Appendix B.

As was described in Section 4.3.2, the recursive functions in the Refactoring program are all trivially recursive. Because of this there is no correlation value for the “Number of non-trivial recursive paths” ($r2$) measure and the “Number of trivial recursive paths” ($r3$) measure has the same correlation value as the “Binary recursion” ($r1$) measure.

The initial observations from the results for the Peg Solitaire program show no statistically significant correlation for either metric, suggesting that distinguishing between trivial and non-trivial paths does not increase the accuracy of the recursion metrics.

4.3.4 Sum and product of recursive path lengths

The metrics that have been discussed so far have considered only the type of recursive path that exists for each function, be it trivial or non-trivial, but have not considered the length of those paths. It is tempting to think that longer recursive paths are likely to be harder to recognise in the source code, and therefore more error prone, than shorter recursive paths.

The length of a call path can be measured in several ways, for instance one might measure the number of calls in the call path, or one might count the number of functions in the call path. For this work the later method was used, this means that a trivial recursive path will have a length of two, while non-trivial recursive paths will have a length of at least three. This method was chosen for ease of implementation because it fitted the internal representation of a call path best, however using either method should not affect the results, and so this is largely a pragmatic choice.

When a function has only one recursive path the metric value for that function is simply the length of the path, but for functions with more than one recursive path some way of combining the lengths is needed. For this work two methods were chosen, taking the sum of the lengths of the individual paths and taking the product of the lengths of the paths. The correlation results for these two measurements are show in Table 11 in Appendix B.

Once again the results for the Refactoring program show correlation values identical to that of the Binary recursion method for the same reasons described in Section 4.3.3.

The results for the Peg Solitaire program are slightly confusing. They show that the sum of the lengths has a small positive correlation with the number of changes, while the product of the lengths has a small negative correlation. However, as neither correlation value is statistically significant this difference may be meaningless.

4.3.5 Interaction of attributes of recursive functions

In the previous sections several ways of measuring aspects of recursion have been presented, none of which showed correlation that was statistically significant at the 5% level because of the insufficient variation in the nature of the recursive functions of the case study programs. It is therefore interesting to investigate whether combining the recursion measures might increase the correlation. To start this process it is worth examining the cross-correlation matrix for the recursion metrics, shown in Table 16 in Appendix C.

The correlation matrix for the Refactoring program has been shown for completeness, but will not be considered in this section because of the identical correlation values of the metrics for this program. This has occurred because the Refactoring program does not contain non-trivial recursion, causing all the recursion metrics, other than the “Number of non-trivial recursive paths” ($r2$) measure, to return the same values.

The correlation matrix for the Peg Solitaire program appears to show that the recursion measurements are all quite strongly correlated with the “Number of recursive paths” ($r4$) metric. This is probably because there is only a small number of functions with non-trivial or multiple recursive paths, which causes all the recursion metrics to produce very similar values, and hence be strongly correlated. Because of this it is not possible to draw any strong conclusions from the correlation matrix.

Because of the high degree of correlation between the measurements for the Peg Solitaire program the regression analysis may not be accurate, but it is presented in Table 23 in Appendix D for completeness.

The results from the regression analysis of the Peg Solitaire program show that the multiple correlation coefficient, R , is only slightly higher than that of the highest individual measurement. This is probably because the individual measurements are highly correlated, and so accumulating them adds very little extra information to the analysis.

4.3.6 Summary

In the previous sections several metrics have been presented to measure aspects of the recursive behaviour of functions. These metrics have shown that, for the two programs studied here, there are only a small number of recursive functions. Because of this it is tempting to assume that recursive programming is not in fact widely used, perhaps because of the availability of standard higher order functions such as `map` and `fold` which are well understood by functional programmers.

However, this observation is not reflected in the work in Section 5.2 of Chapter 5 which studies a wider selection of programs, so this is most likely peculiar to these programs.

The results of correlating the metric values with the number of changes has produced no statistically significant correlations. However this may be caused by the unusually small amount of recursion present in the two programs studied in this chapter.

4.4 Measuring Attributes of Callgraphs

Because function calls form such an intrinsic part of Haskell it would seem that some interesting properties could be measured from the callgraph of a Haskell program. These are described in detail in later sections, but are introduced briefly here.

- Strongly connected component size. Because sections of a callgraph may be cyclic it is possible to use traditional graph algorithms 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. All the functions that are part of a strongly connected component depend directly or indirectly upon the other functions of the component. Because of this one might expect that as the component size increases, the number of changes is likely to increase, since changes in one function may be more likely to cause changes in the other functions. This method is discussed in more detail in Section 4.4.1.
- Indegree. The indegree of a function in the callgraph is the number of functions which call it. This means that the indegree estimates how much code depends upon the given function. Therefore it may be that functions with high indegree values may be more important, because changes to them may affect large parts of the program. This method is discussed in more detail in Section 4.4.2.
- 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 be changed, because changes in any of the called functions may cause changes in the behaviour of the calling function. This method is discussed in more detail in Section 4.4.3.
- A sub-callgraph for a function. It is possible to separate out the subgraph that represents the callgraph of a single function from a callgraph of an entire

program. One might think that the greater the complexity of this subgraph, the greater the chance of the function being changed, since changes in any function that are part of the subgraph may cause changes to ripple through the subgraph. 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 likely to be greater complexity in the interaction of the functions of the callgraph, and therefore a greater chance of errors occurring. Therefore this measure may indicate situations where errors may be introduced.
- Depth and Width. The subgraph of a function can be represented as a tree that contains all the direct or indirect dependencies of the function, and it is possible 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.

These methods are discussed in greater detail in Section 4.4.4.

In the following sections these callgraph measures will be examined in more detail, using the case study programs described in Chapter 3. The rest of this section is organised as follows.

- Section 4.4.1 examines the strongly connected component measure.
- Section 4.4.2 studies the indegree measure.
- Section 4.4.3 discusses the outdegree measure.
- Section 4.4.4 explores ways to measure the size of a functions callgraph.
- Section 4.4.5 investigates the interaction between the various callgraph measures, and the cross-correlation between them.
- Section 4.4.6 presents the conclusions that can be drawn from the callgraph measures.

4.4.1 Measuring strongly connected components

In Section 4.3 callgraphs were presented as a way to measure attributes of recursive functions by finding the cyclic paths caused by recursion. A more general measure that extends this is to measure the size of the strongly connected components of the callgraph.

A strongly connected component of a graph is a set of nodes in which there is a path between any two nodes. Thus the strongly connected component in a callgraph is the set of functions which all call each other, either directly or indirectly. Measuring the size of the strongly connected component of a function might provide useful information about the interdependencies of the function.

If a function has a large strongly connected component then it may be more likely to suffer from changes in other functions of the program affecting its behaviour. Conversely a function with a small strongly connected component may be less likely to be affected. This makes the size of the strongly connected component an interesting attribute to measure.

A metric was written using the callgraph functionality in the Medina library, described in Section 3.2 of Chapter 3, and King and Launchbury's graph algorithms [51] to measure the size of the strongly connected component of each function in the case study program. The values from this measure were then correlated against the number of changes. The results of which are shown in Table 12 in Appendix B.

The correlation value for the Peg Solitaire program, 0.3446, suggests that there is some correlation between the size of the strongly connected component and the number of changes. It is conceivable that this is because changes in one element of the strongly connected component are likely to cause changes in other elements in the component.

However, the Refactoring program shows very little correlation between the size of the strongly connected component and the number of changes, and is only statistically significant at the 10% level. This is probably because there are

only a small number of recursive functions in this program, all of which contain only trivial recursion, and hence have equally sized, small, strongly connected components.

4.4.2 Measuring the indegree of a function

Because a function may be called from many places it is interesting to investigate whether that affects the complexity of the function. This can be done by measuring the indegree of the functions node in the programs callgraph. The indegree measures how many functions a given function is called by, somewhat like the Google ranking system [75].

Measuring this attribute gives a crude measure of how “important” the function is, with more important functions being used by more functions than less important functions. The correlation values from the two case study programs for this metric are shown in Table 12 in Appendix B.

The correlation results showed no significant correlation for either the Peg Solitaire program or the Refactoring program, suggesting that the indegree has little effect on the subjective complexity of a function.

4.4.3 Measuring the outdegree of a function

Having measured the indegree of functions it is also interesting to measure the outdegree. The outdegree measures how many functions are called by a given function. The greater the outdegree of a function, the more things that function directly depends upon. It may be that functions that depend upon large numbers of other functions are more likely to be changed, since changes in any of the called functions may require changes in the calling function.

The correlation results for this metric in Table 12 in Appendix B show that this metric is statistically significant at the 5% level in both programs, showing a reasonable correlation with the number of changes, with Peg Solitaire having a correlation of 0.4783 and Refactoring having a correlation of 0.5723. Thus it is

possible to make the following observation.

Observation 4.4.1 *Functions with a high outdegree value are more likely to be changed than functions with a low outdegree.*

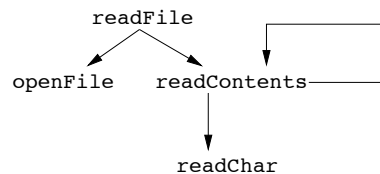
4.4.4 Measuring the size of a functions callgraph

Once a callgraph for an entire program has been created it is then possible to separate out a sub-graph which represents the callgraph for a specific function. It might be interesting to look at that sub-callgraph and examine whether attributes of the sub-callgraph make a difference to the complexity of the function.

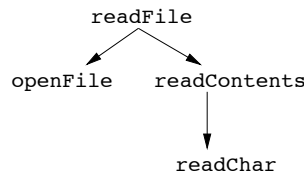
There are many attributes one might measure from the subgraph. Three attributes that are particularly interesting for this work are the depth, the width and the arc-to-node ratio.

The arc-to-node ratio is the ratio of the number of arcs, or function calls, to the number of nodes, or functions. This is interesting because it gives an idea of how “busy” the graph is.

To measure the depth and width of the sub-callgraph it is necessary to flatten any cycles in the graph, that is, to break any loops in the graph. When flattening cycles it is important to avoid producing duplicate nodes in the call “tree”. The intention of the tree is to represent all the direct and indirect dependencies of the function, and thus it is undesirable to include functions (nodes) multiple times. For instance the subgraph for a function called `readFile` might look like this.



The resulting flattened tree would look like this.



Since the tree represents the dependencies of the function, it may be that the deeper or wider the tree grows, the more likely the function is to be affected by changes in any of the other functions in the sub-callgraph. This may therefore be an indicator of functions that are likely to require frequent maintenance.

The correlation values for the depth, width and arc-to-node ratio metrics, shown in Table 12 in Appendix B, exhibit no statistically significant correlation for the Peg Solitaire program, but the Refactoring program shows reasonable correlation between these measures and the number of changes. The Refactoring program also shows some correlation between these metrics and the “Outdegree” (*c3*) metric, which one would expect as an increase in the number of called functions will necessarily make the size of the subgraph increase.

One possible explanation for the difference in correlation values between the two programs is to consider the size and complexity of the callgraphs of the two programs. The Refactoring program has a larger, more complex callgraph than the Peg Solitaire program, and therefore it may be subject to more errors when adding functionality, due to the extra “hidden” complexity. Examining the raw metric values showed that the Refactoring program tended to have larger values of depth, width and arc-to-node ratio than the Peg Solitaire program, which seems to add support to this theory.

4.4.5 Interaction of attributes of callgraphs

Having presented some measures that can be taken from a callgraph it is worth looking at how these might interact. Table 17 in Appendix C shows the correlation matrix for these measurements. This seems to suggest that most of the metrics are measuring different aspects of callgraphs, since there is little correlation between measurements.

The only exceptions to this are the “Depth” (*c4*) and “Width” (*c5*) measures, which are strongly correlated for both case study programs. This seems to suggest the following observation.

Observation 4.4.2 *Callgraphs for functions tend to grow uniformly in both depth and width, and very rarely end up long and thin or short and wide.*

The regression analysis for the callgraph metrics is presented in Table 24 in Appendix D. Since the “Depth” (*c4*) and “Width” (*c5*) metrics are highly correlated the “Depth” (*c4*) measurement was used to represent both metrics in the regression analysis.

The multiple correlation values, R , are a little higher than the greatest correlation of any of the individual measurements for each program. The R^2 value shows that the callgraph measures can explain a sizeable amount of the variance of the number of changes, 27% for the Peg Solitaire program and 36% for the Refactoring program.

Looking at the coefficients for the metrics it seems that the most important callgraph metric is “Outdegree” (*c3*) which is unsurprising because it has the highest correlation values of the metrics. The Peg Solitaire program also has a significant contribution from the “Strongly connected component size” (*c1*) metric, while the Refactoring program has a contribution from the “Depth” (*c4*) metric. This seems to suggest that for programs with non-trivial recursive behaviour the “Strongly connected component size” (*c1*) metric can be used to increase the accuracy of the callgraph measures.

4.4.6 Summary

In this section we have shown several attributes that can be measured from callgraphs. From looking at these attributes it seems that size measures such as the “Width” (*c5*), “Depth” (*c4*) or “Arc-to-node ratio” (*c6*) have some correlation with the number of changes for programs with complex callgraphs.

For programs with non-trivial recursive behaviour measuring the strongly connected component size can provide good correlations with the number of changes. However measures that give an idea of the dependencies of functions, such as the outdegree, give significantly higher correlations than any other single callgraph

measure for both programs.

The study of the “Indegree” (*c2*) and “Outdegree” (*c3*) measures has also shown an interesting comparison with object-oriented coupling measures. The object-oriented equivalents of these metrics are *export coupling* and *import coupling* respectively. Studies of these coupling measures, discussed in Section 2.3.4 of Chapter 2, have shown that it is the export coupling that offers the greatest correlation with the number of changes, which is opposite to the results shown in this work.

It is not clear why the object-oriented measures should produce opposite results to the Haskell measures. One reason might be that the object-oriented measures are dynamic and are therefore only measuring the interactions that actually take place during a particular execution, unlike the Haskell measures that are static and must therefore measure all possible interactions.

Another possible reason for the opposite behaviour may be that object-oriented programs tend to concentrate complexity into a few key objects which are distributed throughout the program and are used by many other objects, making the export coupling metric the better indicator of the complexity. Functional programs tend to have a top-down structure which concentrates complexity into “top level” functions which are not used by many functions, but use many other less complex functions to perform their task, making the outdegree metric the better indicator of complexity.

4.5 Measuring Miscellaneous Attributes of Functions

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. These are described in greater detail in later sections but are briefly introduced here.

- **Pathcount.** Pathcount is a measure of the number of logical paths through a function. The higher the pathcount value, the greater the complexity, in particular high pathcount values indicate code that may be hard to test because of the difficulty of ensuring all execution paths are exercised.

Barnes and Hopkins [10] produced encouraging results for Fortran programs showing that pathcount values were correlated with the number of changes, and so it is interesting to investigate the results of pathcount measures for Haskell programs. This method is described in more detail in Section 4.5.1.

- **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 [96] used a variation of Halstead's [42] operator and operand measures in his work with Miranda, so that method is used for this work. This method is discussed in greater detail in Section 4.5.2.

The remainder of this section is organised in the following manner.

- Section 4.5.1 explores the use of the pathcount metric.
- Section 4.5.2 looks at the effects of the size of functions on their complexity.
- Section 4.5.3 studies the cross-correlation between these miscellaneous measures to see if the measures are linked in any way.

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

Example 17: A function with a subtle pathcount value.

- Section 4.5.4 presents the conclusions that can be drawn from these miscellaneous metrics.

4.5.1 Measuring the pathcount of a function

In Barnes and Hopkins [10] work, pathcount values were correlated with the number of changes. The pathcount value is a measure of the number of logical execution paths through a particular piece of code. Because Barnes and Hopkins produced some encouraging results using pathcount measures that showed correlation between the pathcount and the number of bug fix changes in Fortran programs, it is interesting to see if pathcount metrics may produce similar results for Haskell programs.

To implement a pathcount metric for Haskell it is necessary to consider all the places in which the execution path may branch, which is mostly straightforward. Some obvious cases where branching occurs is in `if ... then ... else ...` and `case` expressions. However, there are also some more subtle situations where the number of execution paths is less obvious. Consider Example 17.

In this function 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 then the guard `x > 0` will be tested. If this guard fails, execution drops through to the third pattern expression, creating a fourth execution path. Thus the pathcount for this function is four. Although this is a contrived example, this kind of “extra path” can occur quite easily in real functions, by omitting an `otherwise` guard, for instance.

Pathcount values indicate the level of complexity in the program code, and in particular the amount of effort required to test the program, because they indicate the number of execution paths that must be exercised in order to fully test the code.

This may be of particular interest in relation to Haskell tracing and observation techniques, where increasing the number of execution paths may make it harder to utilise the techniques effectively.

The correlation results for this metric are shown in Table 13 in Appendix B. These indicate that there is no statistically significant correlation with the number of changes for the Peg Solitaire program, and only a small, but statistically significant, correlation of 0.286 for the Refactoring program. Thus the following observation can be made.

Observation 4.5.1 *The pathcount value of a function has little effect on the functions subjective complexity.*

4.5.2 Measuring the size of a function

With all the measures that have been presented previously it is important not to ignore less sophisticated measures such as function size. The larger a function is, the more complex it is likely to be.

There are many ways one might choose to measure program size. Van Den Berg [96] used a variation of Halstead's [42] operator and operand measures in his work with Miranda, and so for this work we have updated Van Den Berg's measures 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. Although the number of operands and the number of operators were implemented as separate metrics and are treated separately in the statistical analysis, they are really part of a connected pair and as such are likely to be strongly correlated, for instance,

the number of binary operators such as `++` and `:` is directly correlated with the number of operands.

The correlation values for these metrics in Table 13 in Appendix B show that the number of operands and the number of operators have very similar correlation values. Both show a slight positive correlation of 0.1099 and 0.1281 respectively for the Peg Solitaire program, which are statistically significant at the 10% level, while the Refactoring program shows a reasonable amount of correlation, 0.5795 for the operands measure and 0.558 for the operators measure, which are significant at the 5% level. Thus the following observation can be made.

Observation 4.5.2 *The chance of a function requiring bug fixing changes increases with the size of the function.*

One explanation for this observation may be that because Haskell programs are generally quite concise, large functions are likely to be significantly more complex than smaller functions.

4.5.3 Interaction of miscellaneous attributes of functions

Although the measures presented in this section do not appear obviously related it is worth investigating this further. The correlation matrix for these measurements, shown in Table 18 in Appendix C, indicates that “Number of operands” ($m2$) and “Number of operators” ($m3$) are highly correlated which is unsurprising since they are a pair of interconnected metrics. The correlation matrix also shows that the “Pathcount” ($m1$) measure is not strongly correlated with the “Number of operators” ($m3$) or “Number of operands” ($m2$) metrics.

This seems to indicate that the complexity of a function, as indicated by the number of execution paths, is not dependent upon the size of the function. This is similar to the observations about Barnes and Hopkins [10] work in Section 2.5.2 of Chapter 2, and probably shows that “Pathcount” ($m1$) is measuring the effort require to test the code, while the “Number of operands” ($m2$) and “Number of operators” ($m3$) metrics are measuring the programmer effort involved in writing

the functions. The particularly low correlation between these metrics in this work may be an indication of the concise nature of Haskell programs, which can express complex behaviour in a very small amount of program code.

Table 25 in Appendix D shows the regression analysis for these metrics, using “Number of operands” ($m2$) to represent the two size measures. The analysis seems to suggest that the “Pathcount” ($m1$) measure adds very little new information when combined with the “Number of operands” ($m2$) measurement. This is probably because there is little correlation between the pathcount values and the number of changes.

4.5.4 Summary

This section has presented measures derived from metrics for imperative languages. The size measures showed little correlation for the Peg Solitaire program, but did show a fair degree of correlation with the number of changes for the Refactoring program. The Refactoring program is approximately twice the size of the Peg Solitaire program, which may account for the increased correlation.

The “Pathcount” ($m1$) measure appears not to be correlated with the number of changes for the two programs in this case study. However this measure may still be of use for estimating testing effort for Haskell programs, but this use of the metric is not investigated in this work.

4.6 Interaction of Attributes of Haskell Programs

The preceding sections have presented several measurable attributes of functional programs. On their own, most of these attributes show only a small correlation with the number of changes, with only a small number having correlations greater than 0.5, so it is interesting to see how these measures might interact, and how combining these measurements might affect the correlation with the number of changes. The remainder of this section is arranged in the following manner.

- Section 4.6.1 explores the cross-correlation between all the metrics presented in the previous sections.
- Section 4.6.2 uses regression analysis to select combinations of the metrics that give the greatest correlation with the number of changes.

4.6.1 Cross-correlation of Haskell metrics

In the previous sections the cross-correlation between metrics of the same class was shown. These indicated metrics which were measuring similar or related attributes. In this section the cross-correlation of metrics of different classes will be performed.

To do this, cross-correlation matrices were produced for the metrics that have previously been chosen to be used in a regression analysis in a previous section. These metrics were selected so that the cross-correlations that have already been considered in the previous sections will not reappear in this analysis. For the recursion metrics, where the metrics all have a high degree of correlation, the “Binary recursion” (*r1*) metric has been chosen as the representative measurement.

Table 19 in Appendix C shows the correlation matrix for all the chosen metrics for the Peg Solitaire program, while Table 20 in Appendix C shows the correlation matrix for the Refactoring program.

The correlation matrix for the Peg Solitaire program seems to show no strong

correlation, that is, correlation greater than 0.75, between any of the measurements.

The correlation matrix for the Refactoring program, however, shows that the “Number of pattern variables” ($p1$), “Number of operands” ($m2$) and “Distance by the sum of the number of scopes” ($d1$) measures form one strongly correlated cluster, while the “Distance by the maximum number of scopes” ($d2$) and “Out-degree” ($c3$) measures form a second strongly correlated cluster.

Looking at the first cluster of strongly correlated metrics, it seems that the correlation between “Number of pattern variables” ($p1$) and “Number of operands” ($m2$) is probably understandable, because variables are counted as operands, and so an increase in the number of pattern variables will necessarily entail an increase in the number of operands. The correlation with the “Distance by the sum of the number of scopes” ($d1$) measure is less clear. It seems to suggest that as the number of pattern variables increases, so the distance to any called functions, as measured by the sum of the number of scopes, increases. A possible explanation for this is that patterns occur where new scopes are introduced, therefore as the number of pattern variables increases, the number of scopes will tend to increase as well. This increase in the number of scopes will cause the “Distance by the sum of the number of scopes” ($d1$) measure to exhibit higher values.

The second cluster of strongly correlated metrics suggests that the largest semantic or “conceptual” distance, measured by the “Distance by the maximum number of scopes” ($d2$) metric, to any function called from any single function will increase as the number of called functions increases. This may be because functions that call many other functions tend to have a greater number of local declarations, and therefore the semantic distance, measured by the number of newly introduced scopes, will also tend to increase.

The cross-correlation of the metrics is much easier to calculate than the correlation of metric values with the number of changes, and as such it is easier to study a larger number of programs. This is explored in Chapter 5.

4.6.2 Regression analysis of Haskell metrics

If one takes all the metrics and combines them it might be possible to achieve a greater correlation with the number of changes than is possible with a single metric. This can be investigated using a regression analysis.

To perform the regression analysis of the measurements the clusters of strongly correlated metrics were replaced by representative measurements, “Distance by the sum of the number of scopes” ($d1$) for the first cluster and “Distance by the maximum number of scopes” ($d2$) for the second. These were chosen as representatives because they had the highest individual correlation values of the measurements that made up the clusters. The results of the subsequent regression analysis are shown in Tables 26 and 27 in Appendix D for the Peg Solitaire and Refactoring programs respectively.

The regression analysis for the Peg Solitaire program shows that the multiple correlation coefficient, R , is 0.583 which suggests that the metrics are at least partly correlated with the number of changes. The R^2 value shows that over 30% of the variance of the number of changes can be explained by these metrics.

Looking at the coefficients from the regression shows that the largest contribution comes from the Outdegree metric, suggesting that the most important attribute is how many functions a given function directly depends upon.

The coefficient for the “Distance by the sum of the number of source lines” ($d7$) distance measure is -0.2673. This is interesting because it suggests that if the functions used are a long way away in the source code they are *less* likely to introduce errors. This observation may be caused by cross-module function calls, which imply that the calling function is using some well defined and stable interface, and therefore is less likely to have to be changed as a result of the called function being changed.

The regression analysis for the Refactoring program shows that the multiple correlation coefficient, R , is 0.6973, indicating a good degree of correlation with the number of changes. The R^2 value shows that nearly 50% of the variance of

the number of changes may be explained by these measures.

Examining the coefficients from the regression analysis shows that the largest contribution, by some margin, comes from the “Distance by the sum of the number of scopes” (*d1*) metric. This suggests that, for the Refactoring program, it is important to know how complicated the name-space is for each function.

These results could be used to implement a program that could highlight sections of program code that may have unusual behaviour, and which are therefore worth closer inspection.

4.7 Summary

This chapter has taken some of the ideas from Chapter 2 and presented a selection of metrics that measure a wide range of attributes of Haskell programs, such as recursion, patterns, callgraphs, and distance between declarations and where they are used.

There are, however, a number of other “targets” for which metrics could be devised, but which have not been covered in this chapter.

Functional languages often provide powerful abstraction mechanisms such as polymorphic and higher-order functions, or abstract data types. These languages features suggest that useful metrics could be defined to measure attributes of abstraction in programs.

For example, one might measure how polymorphic a function is by counting the number of different type variables present in the function’s type signature, or by counting how many different types the function is used as. Likewise one might measure how abstract a given algebraic data type is by counting the number of constructors which are exported for that data type. Similar metrics could also be created to measure attributes of higher-order functions and other abstraction mechanisms.

These extra measurements are not discussed further in this thesis because implementing such metrics typically requires a type system, which the Medina library, which is used to implement the metrics presented in this thesis, does not currently contain. However it would be possible to implement such metrics if the Medina library used a different “front-end”, such as Programatica[41], which includes a type system. Because of this, it was decided that these extended metrics should be left for future research.

4.7.1 Statistical analysis

As well as presenting a selection of metrics, this chapter has also attempted to perform statistical analysis of the results of applying them to two Haskell programs. This work has shown that some of the metrics are strongly correlated, indicating that the attributes they measure are related, for instance the number of patterns in a function is closely related to the number of scopes in the function. These correlations between metrics are examined in greater detail in Chapter 5.

Looking at the correlation between the metric values and the number of bug-fixing or refactoring changes each function has undergone has shown that in general there is no single metric which gives good predictions, although some of the metrics such as “Outdegree” (*c3*) can give reasonable predictions. Combining the metrics which exhibit the highest correlations can give better predictions. From this we can see that there is no single attribute that makes a Haskell program hard to understand, but rather that the subjective complexity lies in the combination of features.

Unfortunately, due to the time consuming nature of analysing program change histories, the analysis of correlation between metrics and bug-fixing changes is limited to only two programs. Because of this one must be careful not to over generalise the results of this study. To acquire greater confidence in these statistical results it would be necessary to carry out further case studies.

However, there are other interesting studies that can be performed with the metrics that were presented in this chapter. Therefore the work in this chapter is extended in Chapter 5 to examine a wider selection of Haskell programs.

Chapter 5

Trends and Characteristics of Haskell Metrics

Chapter 4 presented a selection of metrics that measure various individual attributes of Haskell programs and analysed the correlation between the individual metrics and the change history of two case study programs, as well as the cross-correlation between the various metrics.

This chapter extends that work by examining a wider selection of Haskell programs. Due to the time consuming nature of examining change histories this chapter does not compare metrics with change histories, but instead concentrates on the relationships between attributes measured by the metrics.

The methodology used for the work presented in this chapter was described in detail in Chapter 3. It is worth noting that the following case study programs all perform tasks which involve language processing.

- HADDOCK - A tool for generating API documentation from Haskell source code.
- HAPPY - A parser generator for Haskell, similar to yacc.
- HAT - A collection of tools for debugging Haskell programs by using tracing.
- HAXML - A library of tools for processing XML, including parsing, pretty

printing and transformations.

- HUNIT - A unit testing framework for Haskell.
- THH - A Haskell program that implements a type checker for the Haskell language.

Typically, when writing programs that involve processing of complex tree-like data structures such as parse trees, programs are constructed in a manner that closely follows the data structure. It is therefore possible that because these programs perform similar tasks they may cause the results to be biased. However for the work presented in this chapter, the presence of these programs does not appear to cause bias because these six programs appear to show the same common metric characteristics as the other eight programs. Therefore this issue is not addressed further.

This chapter also studies the *values* that the metrics produce in order to determine the typical values for the metrics and to discover what values may be classed as unusual, and which therefore indicate programs and functions which deviate from the typical style of Haskell programs.

The commonly held view of the style of typical Haskell programs can be summarised as the following.

Functional programs typically contain many pattern expressions, such as in the argument lists of functions, for manipulating data types and for control flow such as performing `if-then-else` style selective execution. Looping behaviour is typically achieved using groups of recursive functions. Functional programmers tend to break problems down into small parts, which results in programs which consist of many small functions “plumbed” together with a few large functions. Abstraction is usually achieved through the use of polymorphism and higher-order functions.

However, there is little hard evidence that this “folklore” is actually representative of real Haskell programs. Thus by examining the metric values it may be possible to provide a more concrete idea of the features that typify the “average” Haskell program.

The remainder of this chapter is divided into the following sections.

- Section 5.1 examines the interaction between the various metrics on a larger body of programs.
- Section 5.2 discusses the values that the various metrics typically show on a selection of programs and suggests ways in which these values may be incorporated into a tool.
- Section 5.3 summarises the conclusions from the work in this chapter and suggests some ways to utilise this information.

5.1 Analysis of Cross-correlation of Metrics on a Larger Body of Programs

The work in Chapter 4 concentrated on examining the correlation between the various metrics and the number of changes a program has undergone, but that work is based on only two case study programs.

To obtain a much clearer picture of the behaviour of the metrics it would be useful to analyse more programs. However, analysing the program change histories is a prohibitively time consuming process, as described in Chapter 3, so this has not been done in this study, and is instead left as future work.

An alternative method of analysing the metrics on a larger body of programs is to study the relationships between the metrics by analysing a single snapshot of each program. Because analysing a single snapshot of a program can be done quickly and mostly automatically it is possible to examine a larger sample of programs. This allows for greater confidence in any conclusions drawn from the study.

In the following sections the cross-correlation of the metrics will be analysed on a further fourteen Haskell programs. The methodology used for this work and the programs studied were described in Section 3.3.4 of Chapter 3. The cross-correlation matrices for each of the programs are shown in Appendix E, and will be examined in detail in the following sections.

- Section 5.1.1 examines the cross-correlation of the recursion metrics.
- Section 5.1.2 studies the cross-correlation of the callgraph metrics.
- Section 5.1.3 looks at the cross-correlation of the distance measurements.
- Section 5.1.4 investigates the cross-correlation of the pattern metrics.
- Section 5.1.5 studies the cross-correlation of the miscellaneous metrics.
- Section 5.1.6 examines the cross-correlation between metrics of different classes.

- Section 5.1.7 presents the conclusions that can be drawn from the study of the cross correlation.

5.1.1 Cross-correlation of recursion metrics

The tables of cross-correlation values for the recursion metrics, shown in Section E.1, indicate that only five of the fourteen programs feature any non-trivial recursion, as indicated by Tables 32, 33, 35, 36 and 40. However only one program, GETOPT, does not include any recursion. Thus one might make the following observation.

Observation 5.1.1 *The occurrence of non-trivial recursion in Haskell programs is quite unusual, and is associated with complex program behaviour. However the occurrence of trivial recursion is common.*

Because of this observation, and those discussed in Section 4.3 of Chapter 4, it is not surprising that for most of the programs in the study the recursion metrics were exactly correlated.

It is worth noting that four of the programs which did contain non-trivially recursive functions were language processing programs. As was described at the start of this chapter, the program structure of language processing programs often quite closely follows the structure of the data types used in the program.

However, three of the programs which did contain non-trivially recursive functions still exhibited strong correlation between the recursion metrics. As discussed in Section 4.3 of Chapter 4, this suggests that only a small proportion of the recursive functions in those programs are non-trivially recursive, adding another indication that such functions are unusual.

For the two remaining programs with non-trivial recursion, HADDOCK and HAT, it appears that “Number of non-trivial recursive paths” (r_2), “Number of recursive paths” (r_4) and “Sum of lengths of recursive paths” (r_5) are strongly correlated. This seems to suggest that for these programs there is a significant proportion of the recursive paths that are non-trivially recursive, because the number

of non-trivial paths is strongly correlated with the total number of paths. Both of these programs contain large parsers for the Haskell language which have complex recursive behaviour, and are therefore the probable cause of this correlation.

5.1.2 Cross-correlation of callgraph measures

In Section 4.4.5 of Chapter 4 the correlation between the “Depth” ($c4$) and the “Width” ($c5$) metrics was highlighted. Studying the cross-correlation values in Section E.2 it appears that these measures are strongly correlated in all fourteen of the programs in this study. This provides strong evidence for the following observation.

Observation 5.1.2 *The callgraphs of functions in Haskell programs generally grow uniformly in both depth and width.*

Apart from the “Depth” ($c4$) and “Width” ($c5$) measures there is little consistent strong correlation between any of the callgraph measures. This suggests that they are probably measuring distinct attributes.

5.1.3 Cross-correlation of distance measures

Examining the correlation between the distance measures in Section E.3, it appears that in all fourteen of the programs in this study the “Distance by the sum of the number of source lines” ($d7$) and the “Distance by the sum of the number of parse tree nodes” ($d10$) measures are strongly correlated. Therefore the following observation can be made.

Observation 5.1.3 *Lines in the source code of a Haskell program generally contain similar amounts of program code.*

The correlation matrices also show that the “Distance by the sum of the number of scopes” ($d1$) and the “Distance by the sum of the number of declarations in scope” ($d4$) metrics are strongly correlated in thirteen out of the fourteen programs. This is not surprising as the number of declarations in scope is necessarily

likely to increase as the number of scopes increases, however the high degree of correlation suggests the following observation.

Observation 5.1.4 *The number of declarations in scope tends to increase relatively evenly with the number of scopes.*

The one program that did not have a high degree of correlation between these two measures was HADDOCK. This program had a correlation of 0.5753 between these two measures, and as such there is still a fair degree of correlation between them. The HADDOCK program contains some large pieces of IO monad code, and these large `do` blocks may be a reason for the lower correlation.

A large group of the distance measures, consisting of “Distance by the maximum number of source lines” (*d8*), “Distance by the average number of source lines” (*d9*), “Distance by the maximum number of parse tree nodes” (*d11*) and “Distance by the average number of parse tree nodes” (*d12*), appears to form a cluster of correlated metrics in thirteen of the fourteen programs. In the other program the “Distance by the average number of source lines” (*d9*) and “Distance by the average number of parse tree nodes” (*d12*) were still correlated, as were “Distance by the maximum number of source lines” (*d8*) and “Distance by the maximum number of parse tree nodes” (*d11*). This observation again shows the correlation between number of source lines and number of parse tree nodes.

The correlation between the maximum and the average values of these measures may suggest that distances to called functions tend to be of similar lengths, but it may equally be a result of the average value naturally increasing with the maximum value.

This may also account for “Distance by the maximum of the number of declarations in scope” (*d5*) and “Distance by the average number of declarations in scope” (*d6*) being strongly correlated in eleven of the programs, with the remaining programs still having correlations greater than 0.5 between these two measures, and “Distance by the maximum number of scopes” (*d2*) and “Distance by the average number of scopes” (*d3*) being strongly correlated in ten of the programs.

5.1.4 Cross-correlation of pattern measures

In Section 4.1.7 of Chapter 4 the correlation between “Number of pattern variables” ($p1$), “Sum of depth of patterns” ($p2$) and “Pattern size” ($p6$) was discussed. Studying the correlation matrices in Section E.4, this correlation appears to hold across all fourteen of the programs, which emphasises that these measures all appear to be measuring the size of a pattern.

It is also worth noting that “Maximum depth of patterns” ($p3$) or “Number of constructors in pattern” ($p5$) are often correlated with these measures, indicating that these measures may also be measuring the size of a pattern.

5.1.5 Cross-correlation of miscellaneous measures

Investigating the miscellaneous measures shows that the “Pathcount” ($m1$) measure is not consistently correlated with any other metric. This suggests that the “Pathcount” ($m1$) metric is likely to be measuring a distinct attribute.

Looking at the other two miscellaneous measures, “Number of operands” ($m2$) and “Number of operators” ($m3$), shows that they are strongly correlated in thirteen of the programs. As has been discussed in Section 4.5.3 of Chapter 4, this is expected since the two measures form a connected pair.

What is curious, however, is the one program in which these two measures were *not* correlated, CRYPTO. This program includes some modules which contain large tables of literal values. This will lead to the “Number of operands” ($m2$) measure having significantly higher values than the “Number of operators” ($m3$) measure for those identifiers, which may be the cause of the lower correlation seen for this program.

5.1.6 Cross-correlation between different classes of metrics

In the previous sections the cross-correlation within the various classes of metrics has been examined for the fourteen programs. In this section the cross-correlation

between the different classes of metrics will be examined. Looking at the cross-correlation tables in Appendix E shows that there are a number of metrics of different classes which are correlated in at least some of the programs in this study.

The most consistent correlation between metrics of different classes is between “Distance by the sum of the number of scopes” ($d1$) and “Number of pattern variables” ($p1$), which are correlated in thirteen of the programs. In ten of those thirteen programs “Sum of depth of patterns” ($p2$) is also strongly correlated with these two measures.

One possible explanation for this correlation is that patterns are often used where new scopes are introduced, and so the measures will tend to increase together. The one program that did not exhibit this correlation is the THH program. This program contains several declarations that consists of large lists containing lots of nested function calls. These lists have quite high values for the “Distance by the sum of the number of scopes” ($d1$) measure, but only low values, often zero, for the “Number of pattern variables” ($p1$) measure because the declarations are not parameterised and therefore do not use patterns. This might explain the low correlation value for this program.

In eleven of the programs “Pattern size” ($p6$) and “Distance by the sum of the number of scopes” ($d1$) are strongly correlated. As was described above, this is probably a consequence of the tendency for patterns to occur where scopes are introduced.

The “Number of operands” ($m2$) and “Number of operators” ($m3$) measures are correlated with the “Distance by the sum of the number of scopes” ($d1$) measure in eleven programs. This suggests that as the size of a function increases, as indicated by the “Number of operands” ($m2$) and “Number of operators” ($m3$) measures, the number of new scopes introduced in that function increases as well, causing the “Distance by the sum of the number of scopes” ($d1$) measure to increase. This may be because larger functions tend to involve local declarations, e.g. in a `where` clause which introduces more scopes. This leads to the following

observation.

Observation 5.1.5 *Large functions tend to include a proportionally greater number of local declarations than small functions.*

This observation may provide an interesting way of targeting refactorings that lift declarations, since this observation suggest that larger functions might be more likely to have local declarations worth lifting.

Nine of the programs show a correlation between the “Number of operands” ($m2$), “Number of operators” ($m3$), “Number of pattern variables” ($p1$), “Sum of depth of patterns” ($p2$) and “Pattern size” ($p6$) measures. This seems to suggest the following observation.

Observation 5.1.6 *The size and complexity of pattern expressions within a function tends to increase as function size increases.*

This may be due to patterns often being part of the left hand side of function bindings, which will cause the number of patterns to increase as the number of function bindings increases.

There are also a number of metrics which are correlated less consistently, some of these are shown here:

- “Outdegree” ($c3$) and “Distance by the maximum number of scopes” ($d2$) are correlated in six of the programs.
- “Number of pattern variables” ($p1$) and “Distance by the sum of the number of declarations in scope” ($d4$) are correlated in six of the programs.
- “Number of pattern variables” ($p1$) and “Distance by the sum of the number of source lines” ($d7$) are correlated in five of the programs.
- “Sum of depth of patterns” ($p2$) and “Distance by the sum of the number of declarations in scope” ($d4$) are correlated in five of the programs.

Interestingly, the recursion metrics show no correlation with any other class of metrics in ten of the programs, and only inconsistent correlation in the remaining four programs. This seems to show that the recursion metrics are measuring a distinct class of attributes.

5.1.7 Summary

This section has examined the cross-correlation between the various metrics described in Sections 4.1 through 4.5 of Chapter 4 on a selection of fourteen programs collected mainly from the `Haskell.org` web site.

A number of observations have been made from this work, many of which confirm those made from studying the Refactoring and Peg Solitaire programs.

For instance, this study provided further evidence to support the observation made in Section 4.4.5 of Chapter 4 that callgraphs of functions generally grow uniformly in both depth and width.

The observations of Section 4.1.7 of Chapter 4, in which the correlation between “Number of pattern variables” ($p1$), “Sum of depth of patterns” ($p2$) and “Pattern size” ($p6$) was highlighted, are also evident in this study, providing more weight to the hypothesis that the pattern metrics are generally measuring the size of a pattern.

Section 4.3 of Chapter 4 observed that non-trivial recursion was quite rare. This study also agrees with that observation, showing that while trivial recursion is quite often present in the programs in this study, non-trivial recursion is quite unusual. Non-trivially recursive functions can be quite complex, and may not be easily identified by a programmer, so the “Number of non-trivial recursive paths” ($r2$) metric could possibly be used to direct a programmers attention to such functions in order to refactor them into a less complex arrangement.

This study has also shown that the recursion metrics do not show strong correlation with any of the other classes of metrics, and therefore are likely to be measuring a distinct attribute of the programs.

Examination of the distance measures on these programs showed that the distance metrics mostly exhibited the same clustering of strongly correlated distance metrics that was seen in Section 4.2.6 of Chapter 4.

Looking at the cross-correlation between different classes of metrics has shown that “Distance by the sum of the number of scopes” ($d1$) is often correlated with the “Number of pattern variables” ($p1$) and “Pattern size” ($p6$) metrics. This was also seen for the Refactoring program in Section 4.6.1 of Chapter 4, and is probably caused by the tendency for patterns to occur where new scopes are introduced.

The cross-correlation also seems to suggest that as the size of a function increases, the number of scopes in the function tends to increase, as does the number or complexity of the pattern expressions contained in the declaration.

5.2 Typical Values of Metrics

So far the correlation between metrics and the number of bug fix changes and the cross-correlation between the various metrics has been investigated. However, it is also interesting to consider the actual values of the metrics. If one is to use these metrics to make decisions about software it is necessary to have an idea of what values the metrics typically show. In this section the typical values of the metrics will be discussed. The methodology for this work, and the programs used for the study were described in Chapter 3.

Appendix F contains tables showing the mean, mode, median and standard deviation values for each of the metrics for each of the programs discussed in Section 5.1. These values give a summary of the metrics and Appendix G presents histograms of the values of the metrics for each program, providing similar information pictorially. The remainder of this section is organised in the following manner.

- Section 5.2.1 discusses the typical values of the pattern metrics.
- Section 5.2.2 examines the typical values of the recursion metrics.
- Section 5.2.3 looks at the typical values of the callgraph metrics.
- Section 5.2.4 studies the typical values of the distance measures.
- Section 5.2.5 explores the typical values of the miscellaneous metrics.
- Section 5.2.6 presents the conclusions that can be drawn from these investigations into the typical values of the metrics.

5.2.1 Typical values of pattern metrics

The mode values in Table 97 show that generally pattern metrics have very low values, typically zero or one. The standard deviation values seem to show that

most of the pattern metrics have quite a small range of values. The main exceptions are “Number of pattern variables” ($p1$), “Sum of depth of patterns” ($p2$) and “Pattern size” ($p6$), with “Pattern size” ($p6$) having the largest range of values.

The histograms for the pattern metrics show that most of the metrics show the same behaviour, with most of the metric values being in the first quarter of the graphs.

5.2.2 Typical values of recursion metrics

Looking at the values in Table 98, it seems that recursive functions are quite rare because all the metrics have mean and mode values of zero. The histograms for the “Binary recursion” ($r1$) measure, shown in Figure 68 show very well that the majority of the functions in the programs are not recursive. The histograms for “Number of non-trivial recursive paths” ($r2$) (Figure 69) shows that non-trivial recursion is very rare, with the exception of HADDOCK which contains a machine generated parser with complex recursive behaviour.

None of the metrics show particularly large standard deviations, with the exception of “Number of non-trivial recursive paths” ($r2$), “Number of recursive paths” ($r4$), “Sum of lengths of recursive paths” ($r5$) and “Product of lengths of recursive paths” ($r6$) when applied to the HADDOCK program, and “Sum of lengths of recursive paths” ($r5$) and “Product of lengths of recursive paths” ($r6$) when applied to the HAT program. Both of these programs are quite large and contain parsers which tend to have complex recursive behaviour and hence the lengths of recursive paths in these programs will be larger than those in the other programs which have much simpler recursive paths.

5.2.3 Typical values of callgraph metrics

The values from the callgraph measurements (Table 99) show that most of the callgraph metrics have only a small range of values. However the “Width” ($c5$)

measure shows a significantly wider range of values than the other callgraph metrics, particularly for the six program-processing programs. Likewise, the “Strongly connected component size” (*c1*) measure shows a wide range of values for HADDOCK and HAT.

The HADDOCK program tends to show wide ranges of values because of its parser, and this can be seen in the histograms (Appendix G) by the large spikes appearing midway through the graphs. A similar effect may be causing the wider range of values for the HAT program, which also contains a parser.

The reasons for the increased range of values of the “Width” (*c5*) measure is less clear. The histograms show that most of the values are clustered around the low end of the graphs, but there are a number of outlying values. The reason for these outlying values is unknown.

5.2.4 Typical values of distance metrics

The first observation of the mode values from Table 100 is that the distance measures typically show small values, with the largest mode value being eight. However the mean and standard deviation values show that many of the programs exhibit quite a large range of values for the metrics.

Looking at the histograms for the distance measures seems to show that spatial measures, such as “Distance by the maximum number of source lines” (*d8*) and “Distance by the maximum number of parse tree nodes” (*d11*) generally show low values, while conceptual measures such as “Distance by the maximum number of scopes” (*d2*) seem to have a more even spread of values.

This suggests that functions generally use functions that are close to them in the source code, but which may be conceptually further away. This might be a result of local declarations in `where` blocks and `let` expression. One might therefore make the following observation.

Observation 5.2.1 *The functions used by a single function, `foo`, will tend to be located close to `foo` in the source files.*

5.2.5 Typical values of miscellaneous metrics

Inspecting Table 102 shows that “Pathcount” ($m1$) values are typically quite low, with a value of one being the most common, and that the standard deviation is also typically low, as can be seen in the histograms for this measure shown in Figure 81. Thus the following observation might be made.

Observation 5.2.2 *The complexity of a function is more likely to be in its interaction with other functions than in its “local” execution path.*

This does, however, mean that functions with higher pathcount values are likely to have unusual behaviour and are therefore worth examining more closely for possible refactoring. Thus the pathcount metric may be a useful tool in combination with a visualisation technique such as a source code browser.

One exception in this selection of programs is the HAPPY program. This program has high mean and standard deviation values. Investigating the source code for this program revealed that one function has a high degree of nested local declarations, and consequently has a high pathcount value. Removing this function from the measurements produces a mean value of 1.9084 and a standard deviation value of 1.9129, which brings the HAPPY program back in line with the other programs in the selection.

The “Number of operands” ($m2$) and “Number of operators” ($m3$) metrics show mode values of 0.0 in several of the programs. This is caused by modules that re-export imported identifiers. This is because re-exported symbols are treated by the metrics as if they were defined in the importing module, but as they are imported they do not have code in the importing module to be analysed, and so they receive a value of zero. This allows for a simpler implementation of the metrics, although a more sophisticated implementation could treat re-exported identifiers differently.

The mean and standard deviation values for the operand and operator measures show that there is a reasonably small range of function sizes in the programs. The one exception to this observation is the THH program. The large range of

values in this program are caused by some large data structures representing the Haskell Prelude module, and other standard Haskell modules, which are statically constructed in the source code. Some of these data structures consist of hundreds of lines of code.

5.2.6 Summary

In the preceding sections the typical values of the various metrics have been examined by looking at the mean, mode, median and standard deviation of the values, along with the histograms of the metric values, for each of the fourteen programs chosen from the `Haskell.org` web site.

These investigations have shown that the metrics generally have values clustered around the low end of their ranges. This might be exploited by implementing a tool which places a threshold on the metric values, whereby functions with metric values greater than the threshold are indicated to the programmer for manual inspection, and hence possible refactoring.

It is also worth noting that some of the metrics seem largely unaffected by program size, such as “Pathcount” (*m1*), while others such as the semantic distance measure “Distance by the sum of the number of declarations in scope” (*d4*) are particularly sensitive to the program size.

This is important because it is easier to set thresholds for metrics that are not affected by program size because the threshold is likely to be the same across most programs, whereas thresholds for metrics which are affected by changes in program size are more likely to need tuning to the particular program being examined, although it may also be possible to combine such metrics with another metric, such as a program size measurement for instance, to nullify the effect of changes in program size.

The investigation has also shown that only a small proportion of functions are recursive and only a very small proportion are non-trivially recursive, however most programs do contain some recursion. This provides more evidence to support the observations from Section 4.3 of Chapter 4 and Section 5.1.1 of this chapter.

Examination of the distance measures has shown that typically spatial distances are low, while conceptual distances tend to have a larger range of values. One possible explanation for this is that functions might call functions that are close in the source code, but which might be further away in scope. This locality of functions may occur because functional programmers often write the functions in their program to mirror the structure of their data types.

This might imply that programmers tend to group related functions together in the source code. One obvious example of this is the grouping of functions in modules, however it may be that this principle extends to functions within a single module.

As was discussed in Section 4.2.6 of Chapter 4, it may be interesting to use this hypothesis and attempt to permute source code to produce an arrangement of the functions in the source code which in some way exhibits a “minimal” set of distances. Such an arrangement should group together related functions, and may reveal the extent to which this is already achieved by programmers. This project is left as future work.

5.3 Summary

This chapter has extended the work presented in Chapter 4 to examine a wider selection of programs. This work has shown that some of the metrics are correlated, indicating that the attributes they measure are related. For instance, the number of patterns in a function is closely related to the number of scopes in the function. These correlations are summarised in Tables 7 and 8. Furthermore, we make a number of observations from studying the metrics.

- Observation 5.1.1. The occurrence of non-trivial recursion in Haskell programs is quite unusual, and appears to be associated with complex program behaviour. However the occurrence of trivial recursion is common.
- Observation 5.1.2. The callgraphs of functions in Haskell programs generally grow uniformly in both depth and width. Short and wide, or deep and narrow callgraphs appear to be unusual.
- Observation 5.1.3. Lines in the source code of a Haskell program generally contain similar amounts of program code.
- Observation 5.1.4. The number of declarations in scope tends to increase relatively evenly with the number of scopes.
- Observation 5.1.5. Large functions tend to include a proportionally greater number of local declarations than small functions.
- Observation 5.1.6. The size and complexity of pattern expressions within a function tends to increase as function size increases.
- Observation 5.2.1. The functions used by a single function, `foo`, will tend to be located close to `foo` in the source files.
- Observation 5.2.2. The complexity of a function is more likely to be in its interaction with other functions than in its “local” execution path.

	c1	c2	c3	c4	c5	c6	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	
c1	++																		
c2		++																	
c3			++																
c4				++															
c5					++														
c6						++													
d1							++			++									
d2								++											
d3									+										
d4										++									
d5											++								
d6												+							
d7													++						
d8														++					
d9															++				
d10																++			
d11																	++		
d12																		++	
m1																			
m2																			
m3																			
p1																			
p2																			
p3																			
p4																			
p5																			
p6																			
p7																			
r1																			
r2																			
r3																			
r4																			
r5																			
r6																			

++ At least 12 out of 14 programs exhibited a correlation of at least ± 0.75
 + At least 9 out of 14 programs exhibited a correlation of at least ± 0.75
 ? At least 5 out of 14 programs exhibited a correlation of at least ± 0.75
 blank Indicates no consistent correlation

Table 7: A summary of the typical correlation between various metrics. Continued by Table 8 on Page 184. See Appendix A for a key of metric names.

	<i>m1</i>	<i>m2</i>	<i>m3</i>	<i>p1</i>	<i>p2</i>	<i>p3</i>	<i>p4</i>	<i>p5</i>	<i>p6</i>	<i>p7</i>	<i>r1</i>	<i>r2</i>	<i>r3</i>	<i>r4</i>	<i>r5</i>	<i>r6</i>
<i>c1</i>																
<i>c2</i>																
<i>c3</i>																
<i>c4</i>																
<i>c5</i>																
<i>c6</i>																
<i>d1</i>		++	++	++	+				++							
<i>d2</i>																
<i>d3</i>																
<i>d4</i>				?	?											
<i>d5</i>																
<i>d6</i>																
<i>d7</i>				?												
<i>d8</i>																
<i>d9</i>																
<i>d10</i>																
<i>d11</i>																
<i>d12</i>																
<i>m1</i>	++															
<i>m2</i>		++	++	+					+							
<i>m3</i>		++	++	+					+							
<i>p1</i>		+	+	++	++	?		?	++							
<i>p2</i>		+	+	++	++	?		?	++							
<i>p3</i>				?	?	++			?							
<i>p4</i>							++									
<i>p5</i>				?	?			++	?							
<i>p6</i>		+	+	++	++	?		?	++							
<i>p7</i>										++						
<i>r1</i>											++	++	++	++	++	++
<i>r2</i>											++	++	++	++	++	++
<i>r3</i>											++	++	++	++	++	++
<i>r4</i>											++	++	++	++	++	++
<i>r5</i>											++	++	++	++	++	++
<i>r6</i>											++	++	++	++	++	++

++ At least 12 out of 14 programs exhibited a correlation of at least ± 0.75
 + At least 9 out of 14 programs exhibited a correlation of at least ± 0.75
 ? At least 5 out of 14 programs exhibited a correlation of at least ± 0.75
 blank Indicates no consistent correlation

Table 8: A summary of the typical correlation between various metrics. Continued from Table 7 on Page 183. See Appendix A for a key of metric names.

One of the aims of investigating metrics is to determine ways to identify parts of a program that require programmer attention. To do this it is necessary to know what metric values are common, so that it is possible to spot anomalous values. By looking at the metric values from a range of programs it has been shown that most functions have metric values at the lower end of the scale, suggesting that it should be possible to set thresholds above which metric values should be considered anomalous.

This examination of the values of the metrics has also contradicted part of the commonly held notion of the “typical” programming style used by functional programmers. This notion of programming style was discussed at the beginning of this chapter, but this work has shown that it is rare for functional programmers to use recursion, and particularly that it is extremely rare for non-trivial recursive behaviour to be present.

This contradicts the belief that functional programmers often use recursion, and instead highlights that functional programmers are more likely to use abstraction mechanisms. For instance by using polymorphic higher-order functions such as `map` and `fold` to abstract much of the “control” behaviour that one might normally associate with recursion.

A general observation from the work in both this chapter and that in Chapter 4 is that the metrics can generate a vast amount of data. Because of this it can be easy to miss an anomalous metric value, or to obtain a skewed picture of the program by a few unusual values, such as in Section 5.2.5 where one function in the HAPPY program caused the mean and standard deviation of the “Pathcount” metric values to give significantly different overview of the program.

Because of this it would be desirable to have an integrated way of visualising the metric values to help highlight the pertinent aspects of the data, while reducing the clutter of the normal values. The visualisation of software and metric data is explored in Chapters 6 and 7.

Chapter 6

Software Visualisation

As software systems grow larger and legacy systems become increasingly common, the ability to explore, reverse engineer, and refactor them is becoming ever more important. To effectively re-engineer or refactor software requires a good understanding of the program being studied. Tools that automate the refactoring process can help reduce the chance of errors being introduced, but it is likely that at some point a developer will have to make manual changes, for instance to add new features. When this happens it is imperative that the developer has a clear understanding of the software system in order to ensure that changes do not introduce defects.

Software metrics can help in understanding such large pieces of software by indicating areas of the program with particular properties, however metrics on their own are not always sufficient. Metrics provide numerical values which summarise a selection of attributes of a program, but it can be hard to relate these numerical values to the program code. For instance, a metric might show that a function has a high degree of coupling, but it may not be clear just from the metric value why that occurs. One way to increase the usefulness of software metrics is to combine them with software visualisation systems, which allow other attributes of a program, such as callgraphs, to be viewed together with the metric data.

Software visualisation may be as simple as pretty printing program source code or may be a sophisticated abstract view of the software system. Such visualisation

systems may also include extra support for exploring the source code, for instance by allowing a user to perform queries, such as “highlight all the functions with a pathcount greater than X” or “highlight all the functions with pathcount less than X, but with coupling greater than Y”. Queries like this allow the user to quickly explore, and then focus on, specific properties or parts of the software.

Software visualisation can be considered from two perspectives, as described by Oudshoorn, Widjaja and Ellershaw in [74]: from the point of view of the visualisation system and from that of the user. From the system point of view, desirable properties of visualisation systems are:

- **Scalability.** Visualisation systems should be able to cope with large amounts of data, because software systems continue to grow in size.
- **Extensibility.** Visualisation systems should be flexible, being able to cope with tasks the original designer had not conceived of and be able to be customised to the users’ needs. This is also echoed by Knight and Munro [57], who acknowledge that the ability for users of the system to have a degree of control over configuration is likely to lead to greater acceptance and use. Sidarkeviciute [88] takes this further, stating that users may have their own highly individual “mental map” of the program being examined. Sidarkeviciute also points out that the potential users of program visualisation tools are most probably programmers themselves, and may therefore be capable of specifying their own visualisation systems. Additionally, programmers may be happy using a programmatic interface to specify their own visualisation systems, rather than using a graphical user interface to construct their custom visualisation systems. However, it should not be necessary for users to have to customise the system to perform common tasks. Customisation should only be needed for specialist or unusual tasks.

Although extensibility may require increased effort from implementors of visualisation systems, the effort may be rewarded by greater usability and

usefulness, and therefore greater acceptance of the system by the user community.

- **Portability.** Being able to view visualisations without being limited to a single platform can increase the usefulness and popularity of visualisation systems. A tool must be exceedingly compelling if it is to entice end users to switch between platforms.

From the perspective of the user, desirable properties of visualisation systems include:

- **Minimal disturbance to users program.** This applies to both the program performance and to the source code of the program, so that the tool works with existing source code without that code needing alteration.
- **Little or no programmer intervention.** The programmer should be able to just apply the tool to their code. Acceptable programmer intervention might include instructing the visualisation system of the location of source files that are not directly part of the program being investigated, such as library code.
- **Handle real-world problems.** This normally means being able to handle large software systems with many thousands of lines of code in multiple files, but visualisation systems should also be designed to address a real need, such as finding unreachable “dead code”, or finding parts of a program that are highly coupled and therefore inflexible.

Sidarkeviciute [88] states that most common visualisation tools are *code viewers*. Code viewers are tools which provide the user with a fixed set of graphical presentations of an input program. A selection of code viewers are described later in this chapter. Various graphical presentations may be used by code viewers, such as:

- Graph structures, such as control flow, data flow, callgraphs and module dependency graphs.

- Backward and forward slicers, which show the minimal subset of the code that affects a set of variables, known as backward slicing, or the minimal subset of code affected by a set of variables, known as forward slicing.
- Dicers, which show the minimal subset of code that can be executed when a given assertion is true.
- Dead code views, which show “unreachable” code which therefore cannot be executed. This is often useful in legacy systems where sections of code can become unreachable as changes are made to the system.

Storey, Fracchia and Mueller [91] also identify that software exploration tools have much in common with hypermedia document browsers, such as web browsers. This observation influences some of the work in Chapter 7 of this thesis. Knight [53] also highlights the importance of being able to switch between high level and low level views in order to understand how programs work, allowing a programmer to use the high level view to find interesting places in the program, and then focus in on them at a low level. The notion of switching between high and low level view is often encountered in many forms of visualisation.

In [57] Knight and Munro also describe how navigation is important to the visualisation system because it affects the ease of exploration. Navigation should be designed into the visualisation system from the start so that the necessary “signposts” are integrated into the visualisation. If it is added as an afterthought it may be difficult to create an intuitive method of navigation.

Software visualisation can also be used to help aid understand the dynamic behaviour of programs. Dynamic behaviour of a program might be captured using tracing tools such as the Hat system of Chitil, Runciman and Wallace [20]. However, this thesis concentrates on static properties of software. Therefore this chapter mainly considers visualisation systems for static data and presents some examples of work using software visualisation, both with and without metrics, to aid in the understanding of programs. The remainder of this chapter is divided into the following sections.

- Section 6.1 discusses ways to gain an overview of a system.
- Section 6.2 examines ways to investigate specific properties of a system.
- Section 6.3 examines ways in which one might visualise time series.
- Section 6.4 discusses how three dimensional visualisations can be used to overcome the limitations of two dimensional displays.
- Section 6.5 describes existing methods of visualising software systems.
- Section 6.6 summaries the chapter.

6.1 Visualisation for Gaining An Overview

The aim of software visualisation is to enhance a programmer's ability to understand a software system. Often the complexity of understanding a large software system is not caused by the low level details but instead by the high level interactions. This might occur where programmers are allocated to their own sub-systems and have detailed knowledge of those sub-systems, but at the same time they may be unable to gain a high level overview of how the system is connected.

Similar problems arise in many types of data analysis, where it is difficult to see low-level details while still being able to spot high level trends. The visualisation community has developed many ingenious methods of addressing this problem, many of which may be applicable to visualisation of software. Some of these methods will be described in the following sections.

- Section 6.1.1 presents ways of providing high level views of a software system.
- Section 6.1.2 describes a common method of simultaneously displaying both a high level and a low level view.
- Section 6.1.3 introduces some methods of introducing interaction into a visualisation system, giving the user some degree of control over the views.

- Section 6.1.4 presents a visualisation method that combines high level views, low level views and interaction to produce a more exploratory visualisation system.

6.1.1 Scaling, thumbnail views and greeking

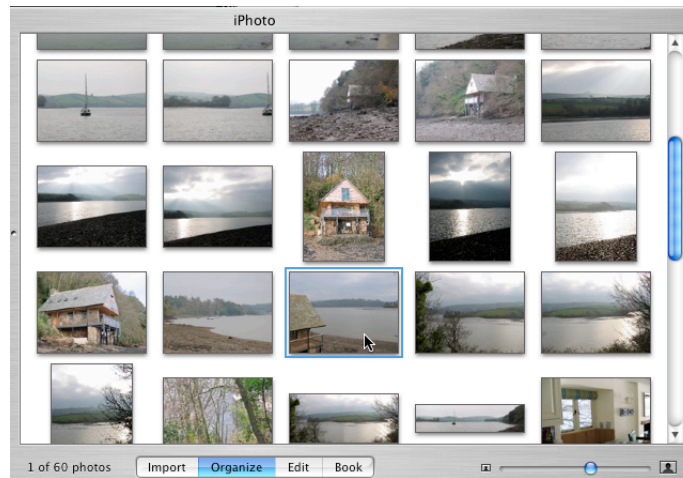
One common method used to gain an overview of data is the use of scaling. By showing components of a visualisation at a smaller scale, a greater number of components can be fitted within the bounds of a computer screen. An example of this might be the use of thumbnail views for browsing an image catalogue. The small scale thumbnail pictures are used to browse and select an image, and then a large scale view is used to view that selected thumbnail at the actual size of the image. This is illustrated in Figure 25.

However, when one considers textual data it is necessary to be aware that there is a limit to the degree to which text can be scaled before it becomes too small to be readable. *Greeking* is a method which does not attempt to display individual character glyphs at a small scale, but instead displays a single solid block of colour. This is illustrated in Figure 26. Greeking is a very effective method for seeing the “shape” of a block of text, and can be augmented by using different colours to highlight various attributes. Greeking systems like these are often called *pixel representations* in software visualisation systems, due to their use of a single screen pixel to represent a single character of a source file. Pixel representations are put to good use in systems such as SeeSoft [9] and Tarantula [29].

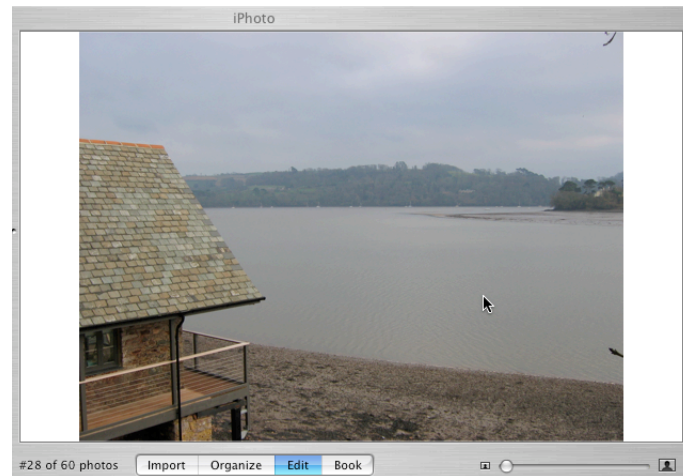
6.1.2 Focus + Context

While showing a single high level view of a data set can be very useful, in the field of software visualisation one often wishes to focus in on a particular area or aspect of the system being examined.

A common way in which this can be achieved is called *focus + context*. In



(a) Viewing images at a small scale



(b) Viewing images at a large scale

Figure 25: An example of the use of scaling in an image browser, showing both a small scale (a) and a large (b) scale view.

```

for(f=0; f < 10; f++) {
  x = doSomethingWith(f);
  y = someFunctionOf(x);
  displayResult(x,y);
}

```

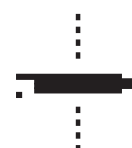


Figure 26: Example of greeked text.

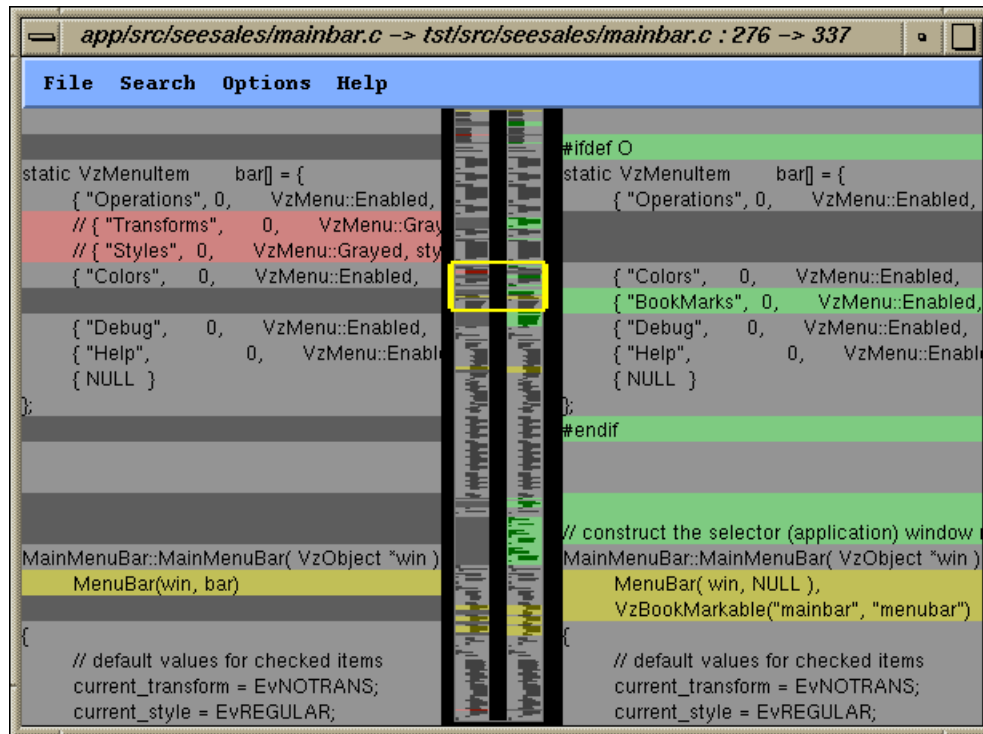


Figure 27: An example of a focus + context visualisation in SeeSoft.

such a system a high level view is used to provide the *context* and some method of selecting the *focus* of attention is provided, such as a cursor. The currently focused item is then displayed as a low level view on the same screen. Such a system allows a user to scan a high level overview for anomalies and quickly focus on them. An example of the use of a focus + context visualisation is shown in Figure 27.

Focus + Context views are particularly suited to cases where one needs to see low level details, while simultaneously being able to view the high level details, and when combined with a pixel representation view, can make an effective system to navigate the source text of a program, as can be seen in Figure 27 which illustrates part of the SeeSoft [9] system.

```

case mcg of
  Just (Just cg) -> defaultNodeDraw (255,255,255) (0,0,120) n sh mo
  _             -> defaultNodeDraw (255,255,255) (0,0,50) n sh False
)
(e mo -> -- Draw an edge
let
  (imp,exp) = generateEdgeStats tcg (deTail e) (deHead e)
  c = edgeColour imp exp
in
  defaultNodeDraw c e mo
)
(e mo -> -- Mouse over callback for a node
(e mo -> -- Mouse over callback for an edge
let
  (imp,exp) = generateEdgeStats tcg (deTail e) (deHead e)
  mv (Just x) = show x
  mv Nothing = "?"
  str =
    case s of
      (n, m) -> (deTail e) ++ " imports " ++ mv n ++ " of " ++ mv m ++
edgeMouseOver str
)
(e mo -> -- Mouse click callback for a node
let
  nodeID n
in
  case m of
    Just (Just m) -> mouseOverClick tcg mod
    _             -> return ()
)
)

```

Figure 28: Example of a fisheye lens over text.

6.1.3 Zooming and fisheye lenses

Closely related to both the focus + context and the scaling systems described previously is the notion of *zooming*. Zooming is the process of dynamically adjusting the scaling of the components in a visualisation system. Zooming differs from focus + context views because one does not see a high level (small scale) view simultaneously with a low level (large scale) view as separate items, but instead one only sees a single view at a specific scale. The most common use of zooming is in image manipulation programs, which allow the user to zoom in to or out of an image, but not to see two levels of magnification at the same time.

Zooming is a feature that allows a user to adjust the scale of a visualisation to best suit their needs for seeing enough detail without being overwhelmed, however it suffers from not simultaneously showing a high level overview. This can be addressed by incorporating zooming into a focus + context system, or by displaying multiple independent windows at different scales.

An alternative method of addressing the focus + context issue when using zooming is to use *fish-eye lenses*. A fisheye lens has varying magnification, or zoom, across its dimensions, with the greatest magnification at its centre, and reducing towards its edges. This is illustrated in Figure 28.

A fisheye system typically lets a user move the lens over a high level view, and thus the user can see both an overview and detail at the same time.

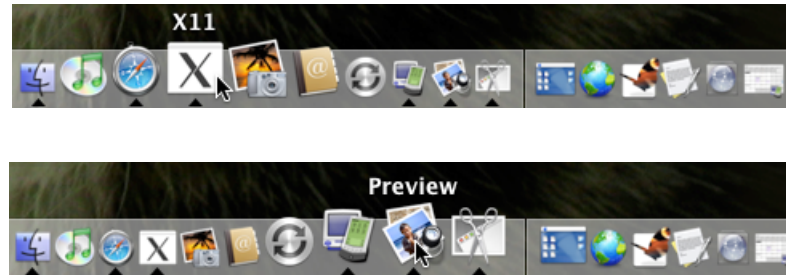


Figure 29: An example of the Mac OS X Dock system using fisheye lenses. Note the sizes of the icons change as the mouse cursor is moved.

Fisheye systems have been used in many visualisation systems, but perhaps their most recent claim to fame is its use in the magnification feature of Apple Computer’s Mac OS X Dock system [5] which is used to show the running applications in their operating system. The Mac OS X Dock is illustrated in Figure 29.

6.1.4 Perspective wall

The *perspective wall*, first implemented by Mackinlay and his co-workers [63], is a similar idea to fisheye lenses. Consider the data to be visualised as a wall. The viewer sees a central section of a wall shown at full scale, with the rest of the wall, to the left and right, tapering off into the background in such a way that the whole of the wall is still visible. The user can scroll along the wall, with the selected part of the wall always being shown in the centre of the screen at full scale. This is illustrated in Figure 30.

The perspective wall thus allows a part of the data to be focused, while the remainder of the data can be viewed from a higher level. The perspective wall is in some ways like a fisheye lens whose magnification changes in the horizontal direction, but is constant in the vertical direction.

Despite the perspective wall most commonly being arranged in a horizontal manner, scrolling left and right, there is no reason why it cannot be used in a

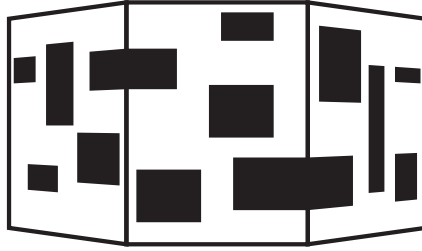


Figure 30: Example of perspective wall.

vertical orientation, for instance to scroll through source code listings.

6.2 Visualisation for Seeing Specifics: Dynamic Queries

Section 6.1 presented methods of simultaneously showing high level overviews and low level details. The attraction of such systems is that a user can quickly browse the high level view to find interesting or anomalous points on which to focus their attention. However, in practice the high level views may contain large amounts of data, much of which may be uninteresting, or which can obscure the points that one may be searching for.

One way to reduce the amount of extraneous information a user must examine is to use filtering operations prior to visualisation, however this can be quite rigid and inflexible, and can inhibit the exploration of the data. For instance, one might decide to filter out those values below a certain threshold. If subsequently the visualisation shows that the threshold is set too low or too high it is then necessary to modify and re-apply the filter and then restart the visualisation.

A more flexible approach is to incorporate the filtering into the visualisation such that the user can dynamically alter the filter and therefore adjust what is displayed on screen in real time. The notion of such *dynamic queries* was first discussed by Williamson and Shneiderman [102] in their Homefinder system, illustrated in Figure 31, and was later adopted by the FilmFinder system of Ahlberg

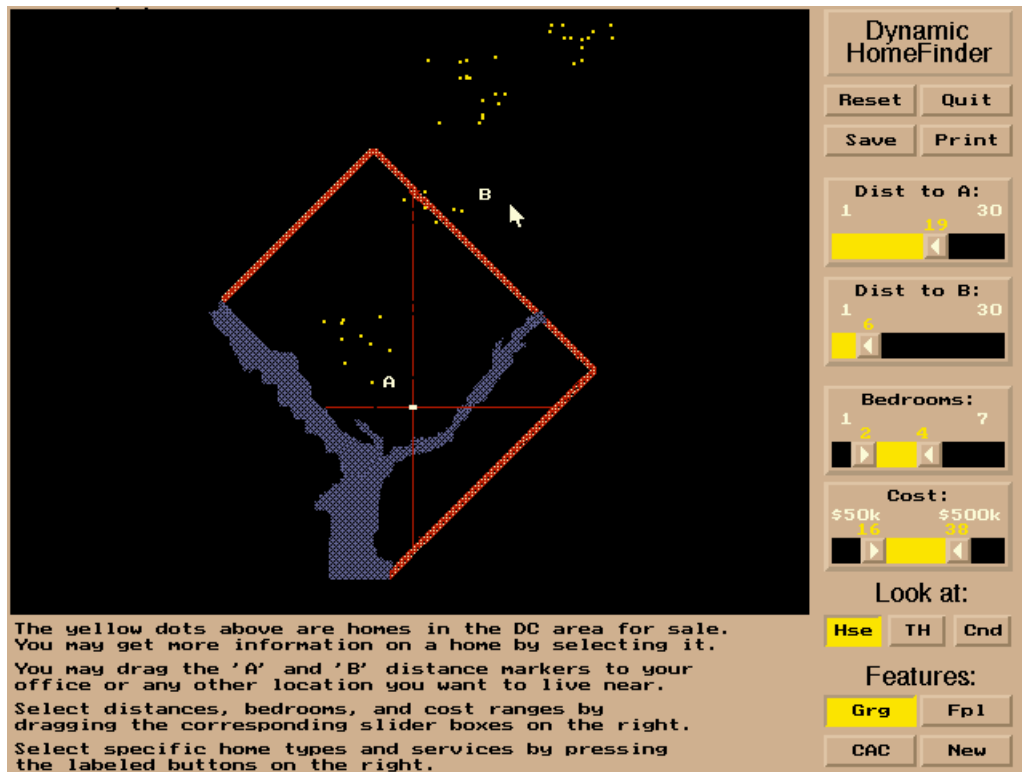


Figure 31: The Homefinder system of Williamson and Shneiderman.

and Shneiderman [3], illustrated in Figure 32.

Homefinder is a system that plots the locations of houses for sale on a map, based on a customer's dynamic query. The query, or filter, selects the houses to display, and is adjusted by means of slider controls, which are used to select the range of values of various attributes of the houses, such as price, number of bedrooms, distance to work, etc. The system automatically updates the map display as the sliders are dragged.

Such a system can provide a very powerful mechanism for a user to quickly filter out the uninteresting items and focus on the important aspects, and has been used in many data visualisation systems. Dynamic queries have also been used to good effect in software visualisation systems such as SeeSoft [9] to control the range of values in a colour scale that are displayed, for example, displaying only values in the red portion of the colour scale. SeeSoft is described in greater

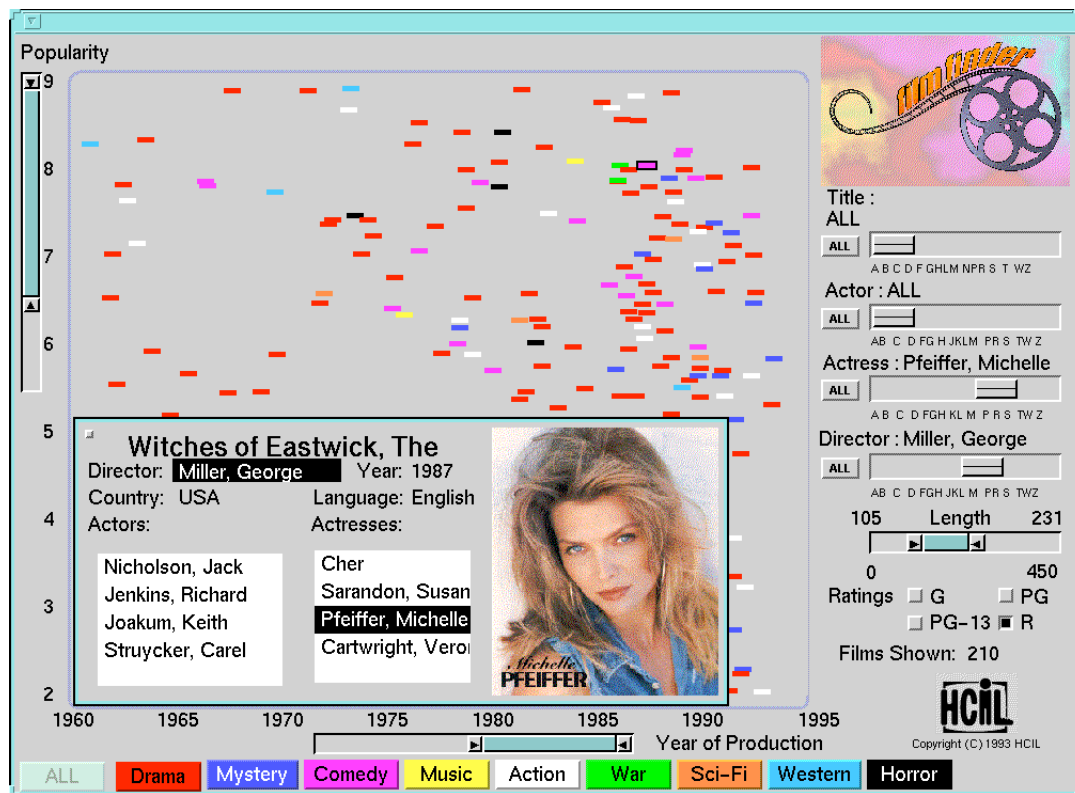


Figure 32: The FilmFinder system of Ahlberg and Shneiderman.

detail later in Section 6.5.2.

6.3 Visualising Time

So far this chapter has concentrated on general concepts of visualisation and how they relate to software visualisation, but has dealt little with specifics. One specific concept not dealt with so far is that of displaying time sequences.

It is often interesting to examine how a piece of software has changed over time, for instance, to discover how a module has changed over time and which changes made it more complex. It is not clear what constitutes the best method of incorporating such temporal ideas into a visualisation system, however there are a number of possibilities which will be discussed in this section in two parts. Section 6.3.1 examines using animation, while Section 6.3.2 presents an alternative method.

6.3.1 Animation

One obvious method of presenting time in a visualisation system is to use animation, where each time “slot” may be presented as a single frame of the animation. It is important to realise here that time may not necessarily be clock time but may be, for instance, iterations of an algorithm in algorithm animation, or bytes of memory allocated in a memory management visualisation tool, such as GCSPy [50], a tool for visualising the behaviour of garbage collection algorithms in Java virtual machines, which is illustrated in Figure 33.

Animation is particularly useful where the degree of difference between two consecutive frames of the animation is small enough that a user can still see what the changes have affected. The usefulness of animation breaks down when the amount of change is so great the user can no longer recognise common features between the frames, and therefore each frame may just as well be a different visualisation.

This can be a particular problem for “arcs and nodes” style graph displays,

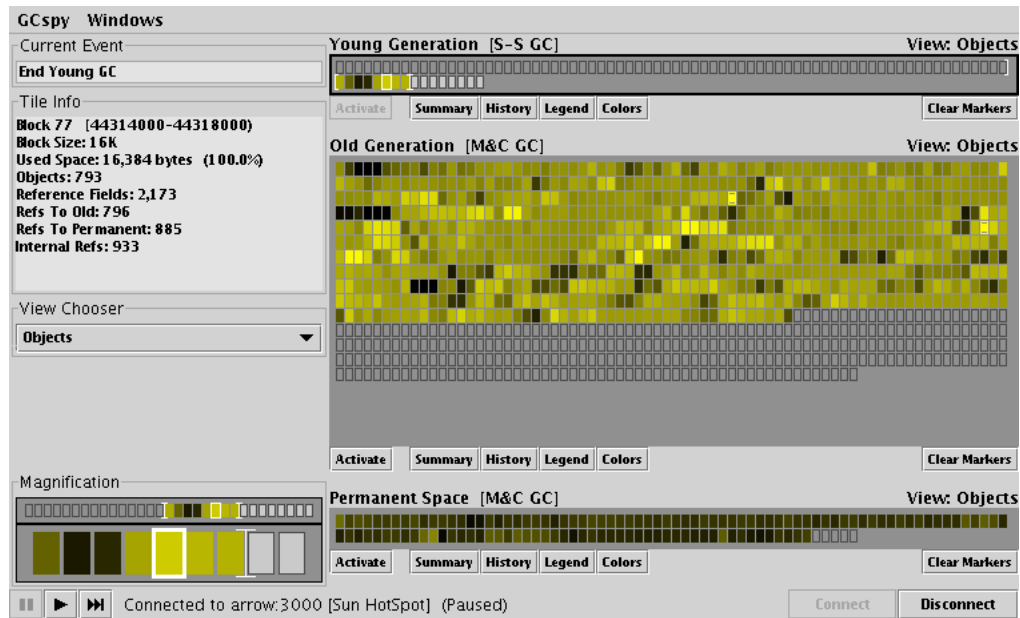


Figure 33: The GCspy system by Jones and Printezis.

such as program callgraphs, where the addition or subtraction of only a small number of nodes or arcs to the graph structure can result in a very different layout of the graph, even though the change may have little effect on the complexity of the program. For this reason, using animation to depict how the callgraph of a program changes over time may not be effective unless significant effort is applied to maintaining as much of the layout as possible between frames of the animation. Despite this problem, sometimes a large change between frames can be useful as an indication that something unusual or untoward has happened.

6.3.2 Bracketing

As was stated in the previous section, the effectiveness of animation is reduced when there is too much change between frames of the animation, and this is a particular problem for graph representations. One way in which this can be addressed is the notion of *bracketing*, introduced by Roberts in [82] and illustrated in Figure 34. Bracketing takes its inspiration from the photographic technique of exposure bracketing.

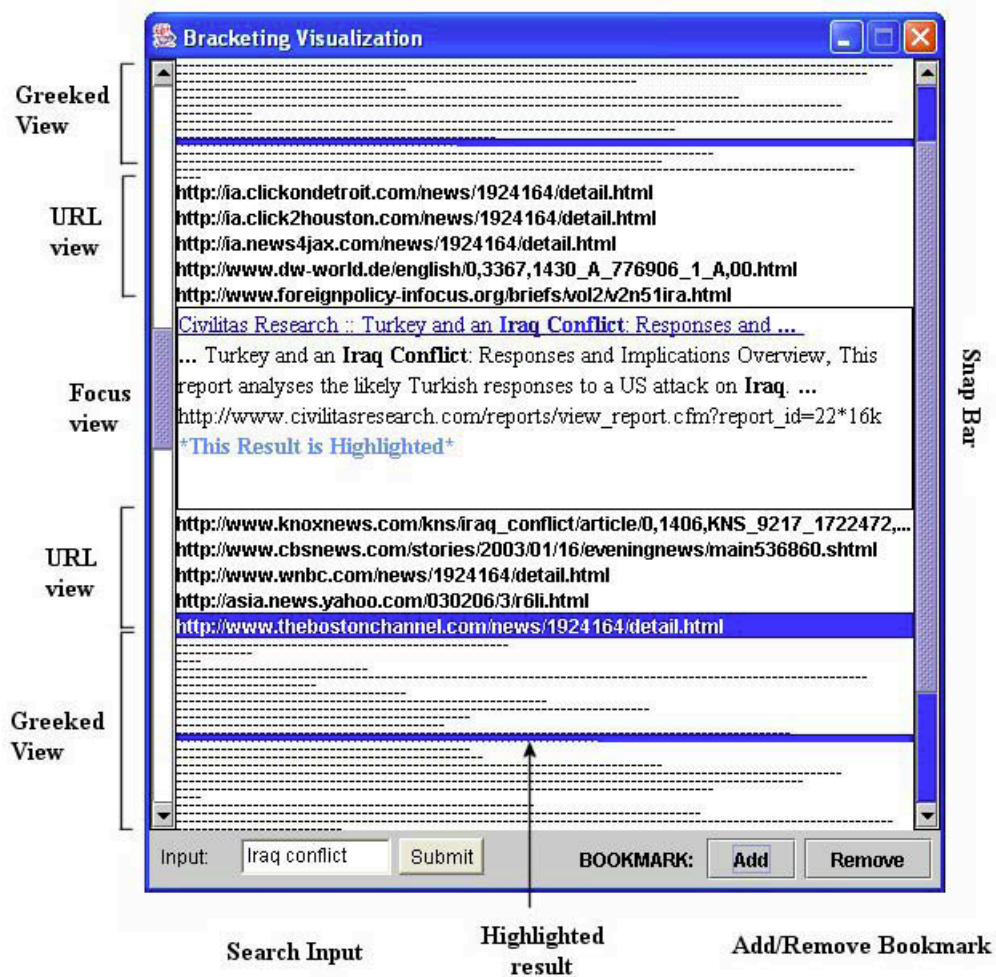


Figure 34: A bracketing visualisation by Roberts.

The basic idea of bracketing is to present a “current” view and two alternative views. For viewing time sequences, these two extra views would be the previous time slot and the next time slot. In some ways this technique is an extension of the focus + context ideas, with the “current” view providing the focus and the two extra views providing the context.

Bracketing addresses the problems of animation by allowing the user to visually compare adjacent frames of the animation in order to see what has changed. Although current bracketing systems show two views side by side for comparison, it might also be possible to overlay them. This may make it easier to spot small changes, but may make it harder to understand large changes if the display becomes “messy” and confused.

Bracketing can also be combined with dynamic queries to adjust the “width” of the bracketing, for instance by showing the Nth previous and next frames, with the width adjusted using a sliding control that the user can drag.

Bracketing is a very intuitive method of allowing a user to browse through a time sequence, and can be combined with animation to provide the positive benefits of both techniques.

6.4 The Use (and Abuse) of 3D

In [55], Knight and Munro highlight how the size and complexity of many modern and legacy software systems often show the shortcomings of graph structured visualisations, such as callgraph displays, because the cluttered and confusing displays they generate when attempting to layout the large number of elements.

While traditionally three dimensional graphics have been expensive and difficult to work with, cheap high performance hardware-accelerated three dimensional graphics has become increasingly widespread in recent years and is now available on many desktop PC’s and even laptop computers. Because of this it seems feasible that practical visualisation systems could move away from some of the constraints of two dimensional views. However, three dimensional views must

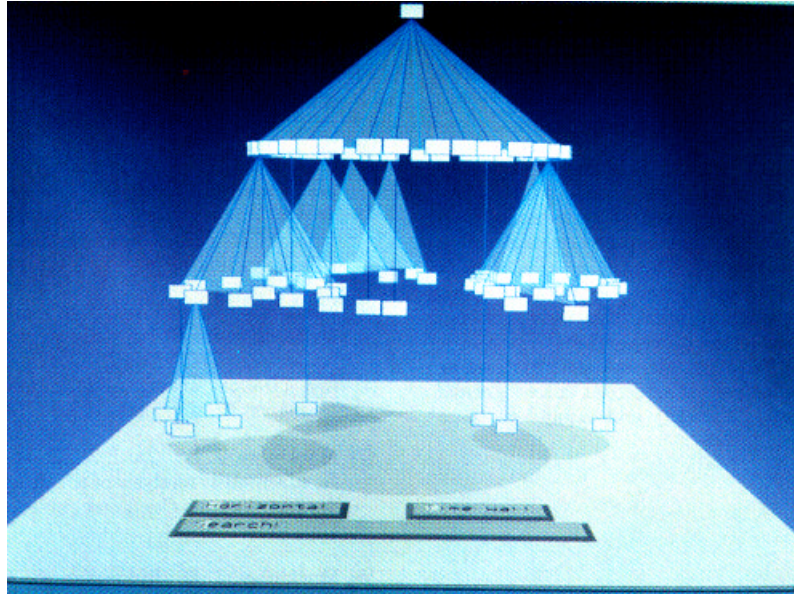


Figure 35: An example of a Cone-Tree visualisation.

still be rendered onto a two dimensional computer screen, and as such pose interesting questions about issues such as navigation and how to avoid views becoming cluttered. Systems such as VRML [43] browsers may help provide navigation facilities, but consideration must still be given to how this might be integrated into the visualisation.

While attempts have been made to transfer two dimensional graph views such as callgraphs or inheritance trees into a three dimensional space, for instance the work by Robertson, Mackinlay and Card on Cone-Trees [83] which is illustrated in Figure 35, it has been shown by Young and Munro [105] and Knight and Munro [57] that in most cases this does not produce effective and usable visualisations because the use of the third dimensional must still be rendered into a two dimensional screen and therefore tends to add clutter, rather than useful information. Instead attention has focused upon exploiting the third dimension by either producing different abstract representations, discussed below in Section 6.4.1, or generating *real world* views, examined in Section 6.4.2.

6.4.1 Abstract views

Because two dimensional graph techniques do not scale well to three dimensions, some effort has been spent on devising other abstract methods of presenting information about software.

Young and Munro [105] and Young [104] describe a visualisation based around three dimensional geometric shapes such as cubes and cylinders. Their system consists of two parts, *CallStax* and *FileVis*, which can be combined into a single visualisation system.

CallStax, illustrated in Figure 36, is used to display the calling structure of a program written in C, and so is essentially an alternative to callgraphs. *CallStax* represents calling paths through a program as stacks of coloured blocks. *FileVis*, illustrated in Figure 37, is a visualisation that shows individual source code files as platforms floating in a three dimensional space. Each function in a file is represented as a block on the platform representing the file. When viewing from a distance, blocks on a platform show two attributes of their function, the length of the function and its complexity relative to all other functions in the program. Length is represented by adjusting the height of a block, while complexity is indicated by adjusting the colour. When viewed from a short distance, blocks are modified to show extra information such as a breakdown of the lines of code, comments and blank lines in the function.

However, *CallStax* and *FileVis* are so abstract that it can sometimes be difficult to relate the visualisation to the source code. Additionally, it is still not clear that the system scales well.

A more recent visualisation technique presented by Simon, Steinbrücknet and Lewerentz in [89] is less abstract and less confusing. Their system, illustrated in Figure 38, is used to aid refactoring of software systems by moving methods and attributes between classes of a Java program, such that methods and attributes are contained in the class to which they are most related.

Their system uses a cohesion metric to measure the “distance” between the

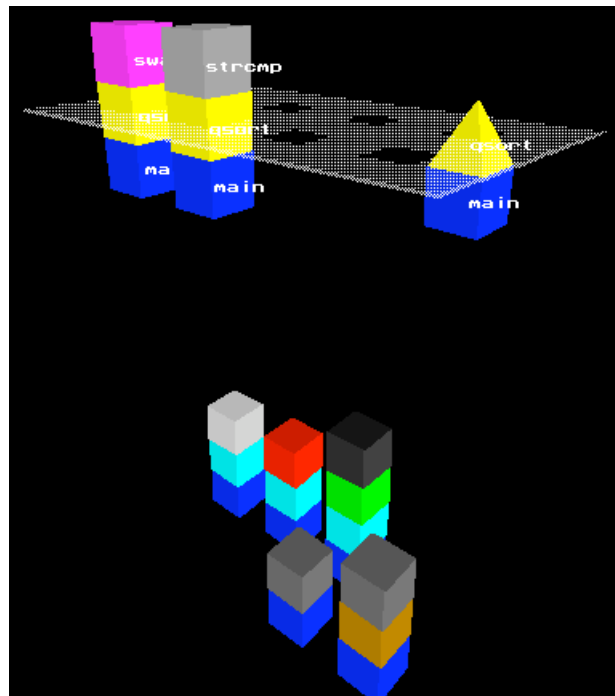


Figure 36: An example of the CallStax visualisation system.

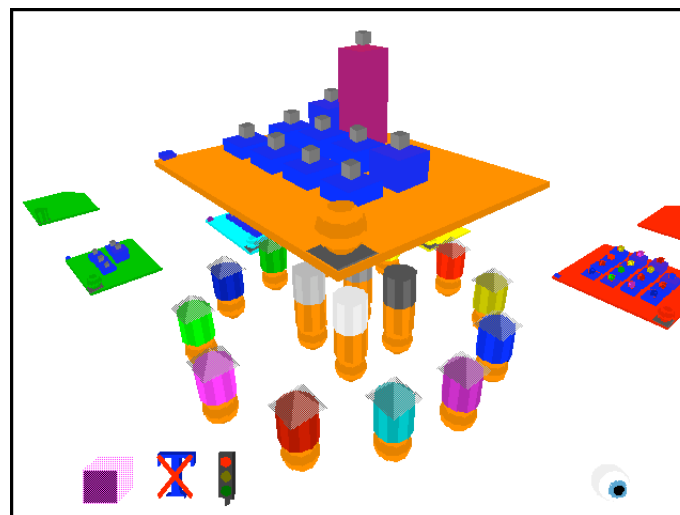


Figure 37: An example of the FileVis visualisation system, showing a file containing some low-detail function representations.

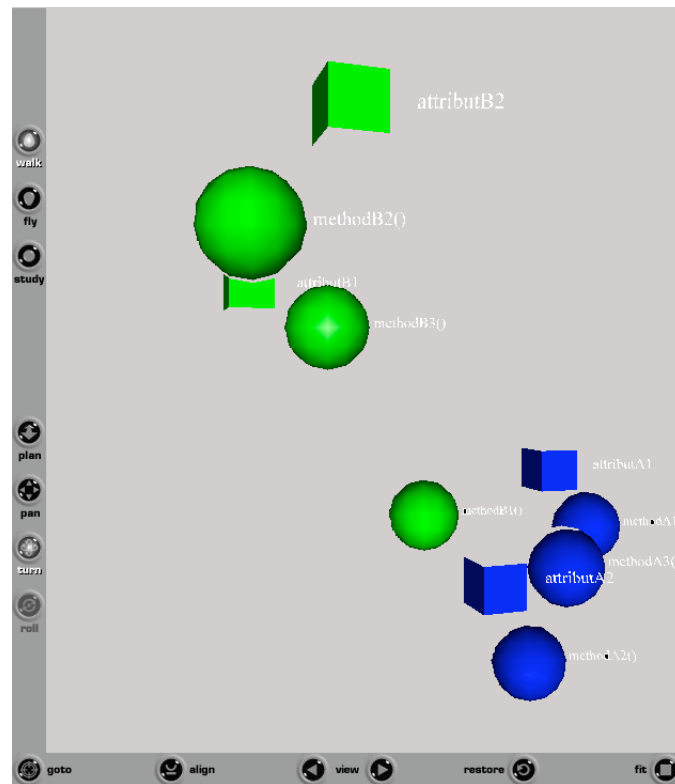


Figure 38: An illustration of visualising distance metrics by Simon, Steinbrücknet and Lewerentz.

methods and attributes of a Java program. It then uses a spring embedder to layout spheres and cubes, representing methods and attributes respectively, in a three dimensional VRML world. All elements are coloured according to the class to which they belong, for instance, all methods and attributes of class `Foo` might be green, while those of class `Bar` may be blue.

This visualisation system will show methods and attributes that are closely related as clusters of spheres and cubes. The most useful aspect of this display is that one can instantly see if a method or attribute should be moved between classes, because this will show up as a differently coloured element amongst a group of elements of a common colour. In an ideal system, in which methods and attributes which are related are grouped in common classes, the elements of the visualisation would be clustered in distinct groups, and each group would consist of elements of only a single colour.

This visualisation system can be very useful in small systems but suffers, as many software visualisation techniques do, from problems with scalability. The display can quickly become cluttered and hard to read as the number of classes, methods and attributes increases. This is avoided to some extent by allowing the user to select the classes to inspect, thereby limiting the amount of information displayed, however it is not clear if limiting the classes in this way reduces the usefulness of the system by possibly hiding interesting relationships between items in the diagram and items which have been excluded. Other methods of addressing scalability issues include the use of filtering techniques or dynamic queries, as discussed in Section 6.2. The issue of scalability is a general problem when attempting to visualise complex hierarchical data, and requires careful consideration when designing a visualisation system.

6.4.2 Real-world views

A problem with abstract views of software is that they are sometimes so abstract that it requires effort to relate the visualisation to the artifacts being examined. One way to avoid this problem is to attempt to represent the artifacts being

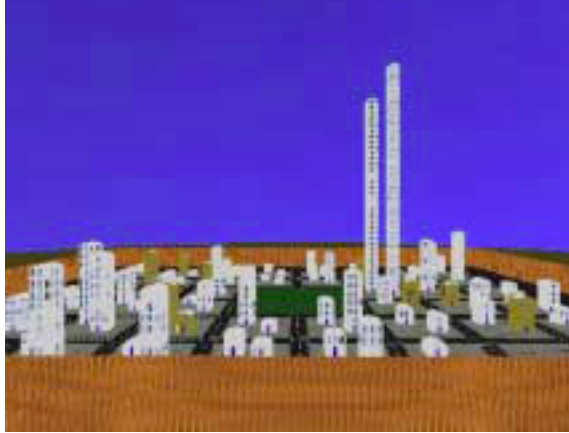


Figure 39: An example of the Software World system by Knight and Munro.

examined as a *real-world* scene.

One such example of using real-world visualisation is *Software World*, described in more detail by Knight and Munro in [54, 55, 56] and illustrated in Figure 39. Software World is a system to display Java source code as virtual worlds depicting buildings, districts, cities, and at the highest level, atlas views. The various elements of Software World represent the source code at varying levels of granularity, with buildings representing methods, in which the number of doors and windows on the buildings represent the number of local variables and parameters respectively, districts representing classes, cities representing packages of classes and atlas views showing the whole program.

Visualisation systems such as Software World show how it is possible to represent software systems in three dimensions without resorting to obscure abstract representations. Questions still remain about the scalability and the ease of navigation of such systems, but these are issues that affect all visualisation systems, and are acknowledged by Knight and Munro in [57].

6.5 Visualising Software

The previous sections of this chapter have presented some basic concepts and ideas relating to software visualisation. These concepts and ideas have not remained purely theoretical, but have been implemented in real systems. A selection of these systems are presented in the remainder of this chapter. Several of these systems also show how visualisation systems can enhance the usefulness of software measurement techniques. The rest of this section consists of the following parts.

- Section 6.5.1 discusses SeeSys, a system that displays hierarchical data in a space efficient manner.
- Section 6.5.2 explores SeeSoft, a descendent of SeeSys that introduced many visualisation techniques to software visualisation.
- Section 6.5.3 presents a system for exploring test case usage in software systems, using some of the ideas from SeeSoft.
- Section 6.5.4 presents methods for displaying large amount of data in a small area using aliasing to merge data points.
- Section 6.5.5 shows how software metrics can be incorporated into graph-style visualisations.
- Section 6.5.6 shows how metrics can be integrated into a tree visualisation in a flexible manner using fisheye lenses.
- Section 6.5.7 presents a system that allows user scripted actions to control and adjust the visualisation of metric values.

6.5.1 Space-Filling software visualisation

When applied to production-sized systems, routines for producing flow charts, function callgraphs, and structure diagrams often break down because the display

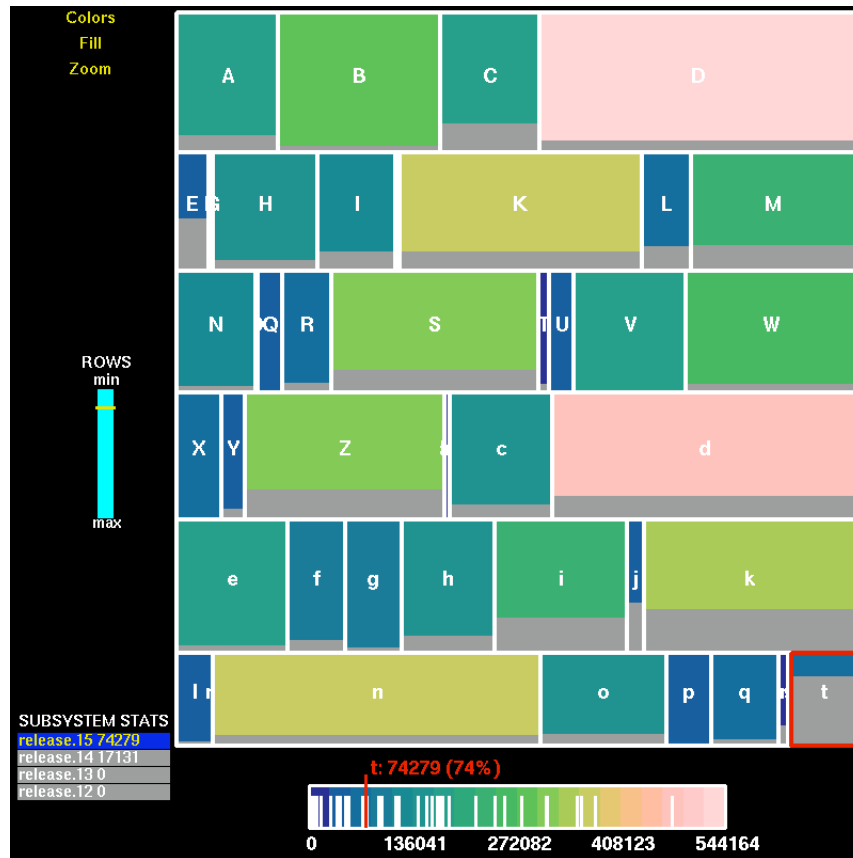


Figure 40: An illustration of SeeSys by Baker and Eick.

becomes too complicated and illegible due to the complex nature of the interactions they are modelling.

In [8] Baker and Eick developed a space-filling technique for displaying source code related software statistics, such as metrics, by visualising program source code in files, directories and subsystems. Their system, *SeeSys*, is based on the work of Johnson and Shneiderman [49] on using tree maps to show hierarchical data.

SeeSys, illustrated in Example 40, represents the entire software system as rectangles on screen, with each rectangle representing a separate sub-system. The area of each of the rectangles is proportional to some metric taken from the corresponding sub-system.

Each rectangle is further partitioned vertically, with each partition representing a directory in the sub-system, and the area of the partition being determined by a metric taken from the directory.

This system allows for a straight-forward visual comparison of directories within a sub-system because the area of each visual component is always proportional to the metric value for the corresponding software component. Additional information can be presented by vertically filling each directory rectangle with a colour.

To further aid exploration, SeeSys also supports zooming to view a single sub-system. In this view each directory rectangle is horizontally partitioned to represent the files within that directory. Once again the area of the partitions is proportional to some attribute of the corresponding file.

This technique provides a hierarchical view that immediately relates files to their directories and directories to their sub-systems, and colour filling allows outlying values to be quickly identified.

In visualising large software systems, it is often important to utilise screen real-estate efficiently. Objects placed on the screen must be large enough to convey information, but small enough to allow room for many other objects. SeeSys does this by placing rectangles next to one another in such a way that 100% of the display area is utilised. This has a small drawback when new elements are added to a program, which can cause the layout of objects in the visualisation to change. However, this is a problem with many visualisation techniques that use layout engines.

6.5.2 SeeSoft

As software grows larger, the need for some way to visualise data about such software increases. An often cited work on software visualisation is that of Ball and Eick [9]. They describe their visualisation tool, SeeSoft, and identify three basic properties of software which need to be visualised:

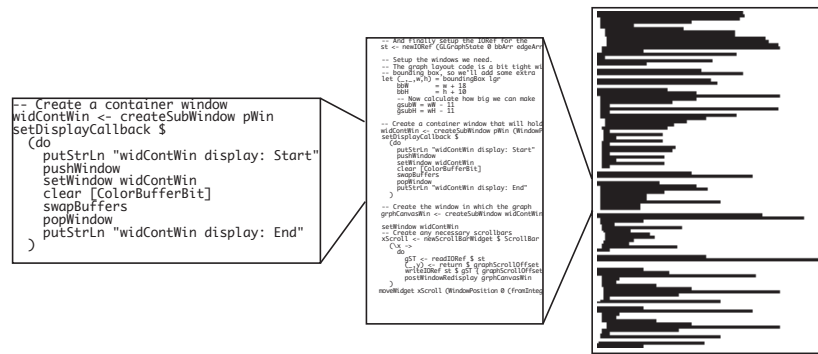


Figure 41: An illustration of the SeeSoft line representation.

- Software structure. This is generally represented using directed graphs.
- Run-time behaviour. Algorithm animation generally uses graphical representations of data structures and motion.
- The code itself. Pretty printers are basic and widely used.

The general strategy for visualisation of this data has been to decompose the program into modules and visualise each module separately. This tends to lose the bigger picture of how the modules interact. SeeSoft uses four different representations of programs:

- Line Representation. The screen is divided into three panels. Each panel displays the program text at progressively smaller scales. The left hand panel shows the program text at normal size, the middle panel shows the text at the smallest font size that is still readable, and the right hand panel shows the program text such that each line is represented by an appropriate line of pixels. In each panel, the text is colour coded. For example, if visualising code age, new code may be red, old code may be green. This three-part view is illustrated in Figure 41.
- Pixel Representation. In this representation each line of code is represented by a colour coded single pixel, or a small number of pixels, which are ordered



Figure 42: An illustration of the SeeSoft pixel representation.

left to right in rows within a column. This is illustrated in Figure 42. The ordering may be on either the line's position in the source file or the line's colour. Pixel representations are often used as a scroll bar for other visualisation system such as pretty printers. Systems that use pixel representations in this manner are described later in Chapter 7 of this thesis.

- **File Summary Representation.** This representation displays file-level statistics, and is illustrated in Figure 43. Each file is represented as a rectangle. Each rectangle may be one of four heights representing file size as measured by lines of code. Having only four heights ensures all files are visible, regardless of their size. Within each rectangle other representations, such as pixel representations, may be used to summarise the files contents.
- **Hierarchical Representation.** This representation is used to represent hierarchically stored systems. It uses a generalisation of a pie-chart called a tree-map. In a tree-map a square is divided into rectangles, with each rectangle representing a sub-system or directory. Each rectangle is proportioned to match the size of the code within that subsystem. This method was originally used by the SeeSys system, which was described earlier in Section

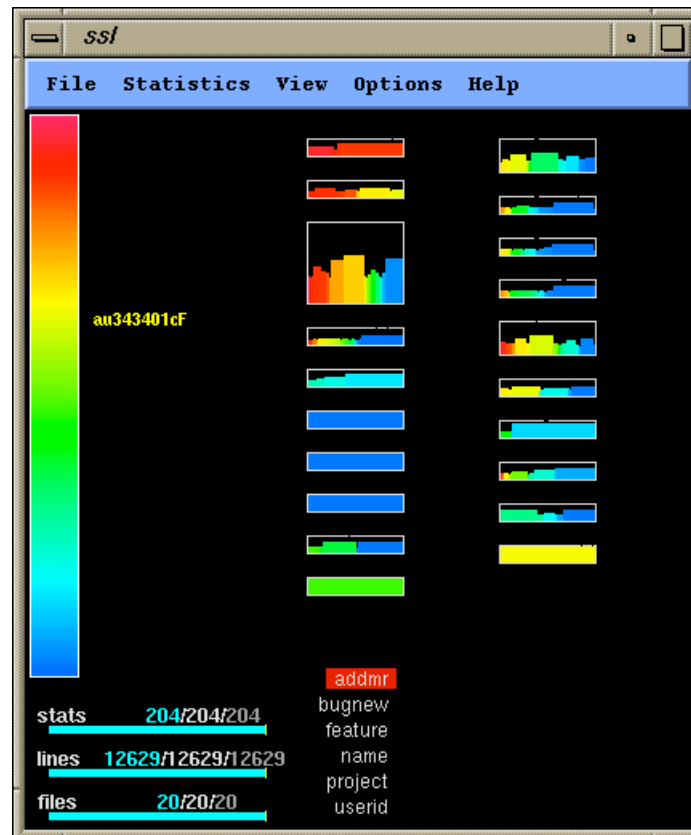


Figure 43: An example of the SeeSoft File Summary Representation.

6.5.1.

Further examples of the representations provided by SeeSoft are presented by Ball and Eick [9].

One of the problems of pixel and line views is that they can become quite busy when applied to large systems, making it harder to spot patterns within the display. To combat this a dynamic query system was used to allow the user to adjust the colour scale and turn on or off specific colour ranges, thereby reducing display clutter. The system also includes an intersection operation. This allows a user to, for example, show the bug fixing code written by a specific programmer by selecting a programmer and switching the visualisation to use a “bug fix count” metric to control the colour scale. The intersection operation allows for many interesting visualisation combinations to be achieved. A particularly interesting combination is to colour code program text by programmer. “Rainbow” files, those that contain many colours, will have been modified by many programmers suggestion that there have been many errors, or that the code is heavily coupled to other modules, and that the code may therefore benefit from re-engineering.

6.5.3 Tarantula

Eagan and his co-workers [29] have used some of the ideas from SeeSoft and the notion of dynamic queries in their Tarantula system, which is illustrated in Figure 44. Tarantula is a system that uses the line representation style used in SeeSoft to allow developers to quickly see which source lines have been executed by test cases. Source lines are colour coded to depict whether they have been executed only by passed tests, failed tests, no test or a mixture. Tarantula has several modes of operation which are described below.

- Default. In this mode, the source lines are simply shown in grey. No lines are colour coded.
- Discrete. This mode colour codes source lines in one of four colours. A line is coded grey if it is not executed by the test cases, green if it is executed

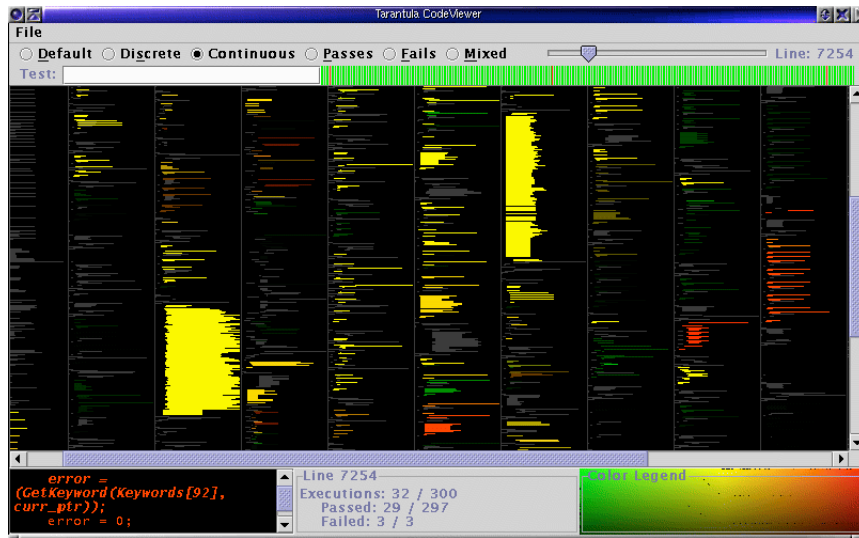


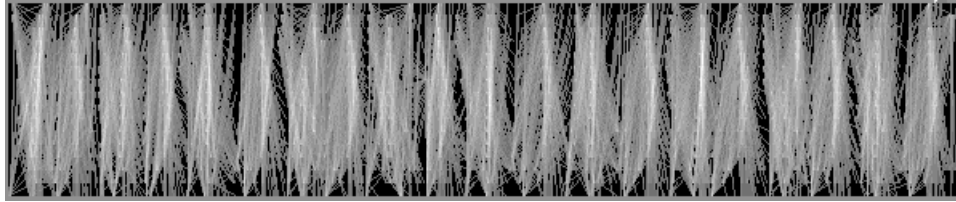
Figure 44: An example of the Tarantula system by Eagan, Harrold, Jones and Stasko.

only by test cases that were passed, red if it is executed only by test cases that failed and yellow if it is executed by a mixture of passed and failed test cases.

- Continuous. This is the most complex mode of operation. In this mode all executed lines are rendered in a spectrum from red to green and with varying brightness. The hue of a line is determined by the percentage of the tests executing the line that failed and the percentage of tests executing the line that passed. The brightness of the line is determined by the greater of the two percentages. This results in lines that have either passed all their tests or have failed all their tests being displayed at full brightness, while lines that have passed half their tests and failed the other half would be shown at half brightness.
- Passes, Fails and Mixed. In these three modes only the listed lines are shown. For instance in *Passes* mode only those lines executed in test cases that were passed are shown. In each mode, the brightness of the lines is determined by the percentage of test cases that execute the line. For example lines



(a) A graph of a neural network shown without the use of an Information Mural



(b) A graph of a neural network shown using an Information Mural

Figure 45: An example of data shown with and without the use of the information mural technique.

executed by all failed test are bright red, while lines executed by only a small percentage of failed tests are dark red.

6.5.4 Information mural

An interesting variation of the SeeSoft work is the *information mural* which is described in detail by Jerding and Stasko in [48]. Information murals provide a technique in which large data sets are represented in miniature using attributes such as greyscale shading, intensity, colour, and pixel size along with anti-aliasing. This is illustrated in Figure 45.

One problem which occurs when displaying large data sets is the limited size of a computer's screen. It is common to scale the data set down to the size of the screen, but this can result in many data points being mapped to the same screen pixel. If this occurs, some information will be lost because it will not be apparent how many data points map to the same pixel. This is called *aliasing*. Information murals attempt to solve this problem by adjusting the intensity of

pixels to indicate the number of data points mapped to the individual pixels.

Information murals can be combined with the line view from SeeSoft to provide even greater compression. SeeSoft's line representation requires one line of pixels per text line which for large files or small screens may not be available. Using an information mural allows this to be scaled into the available number of pixels.

This use of information murals is particularly useful where pixel representations are used as scroll bars in software visualisation systems. The information mural technique allows the pixel representation to be scaled to the available screen space with only minimal information loss. Information murals have also been used to enhance graph displays such as callgraphs. When there are many overlapping edges in a callgraph it can become difficult to trace the individual edges, or even to see groups of edges. Using an information mural technique to draw edges as a greyscale can show much greater detail in these hard to read areas by showing groups of edges that follow similar paths brighter than individual paths. This can be seen in Figure 45. Jerding and Stasko illustrate many such uses of information murals in [48].

The information mural does have some limitations however. Information murals use greyscale shading to indicate density. It can be difficult for humans to distinguish between fine variations of greyscale, however, when colour is used distinguishing density can be even harder, because colour information is better suited to categorised data. However, it can still be difficult to spot small amounts of one colour amongst large amounts of a different colour. For these reasons it is important to carefully choose the colours or hues that will be used, in order to maximise the clarity of the display.

6.5.5 Graph display of software

In [25] Demeyer and others show how combining simple, easy to implement software metrics with inheritance graphs can significantly help understanding of such legacy systems. Figure 46 illustrates one of their systems which represents classes as nodes and inheritance relationships as edges. They claim each node in a graph

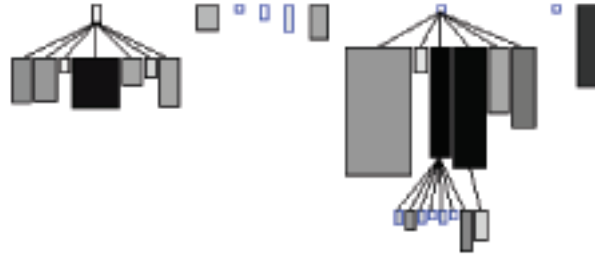


Figure 46: An example of graph display of software in which node width, height and colour are all used to indicate different attributes.

can simultaneously display up to five different metric values by adjusting the height, width, x and y positions and colour. Zooming and scaling can be used to focus on interesting parts of the graphs. In order to use the x and y position of nodes to display metric values it is necessary that there is an absolute fixed origin in the display system, and this rules out some layout systems. It is also interesting to note that they do not offer an example of a layout system in which it could be used.

Although it is not clear from their work whether adding five different attributes to each node of a graph actually increases understanding, their work does highlight that combining software metrics with visualisation techniques increases the power and usefulness of the metrics.

6.5.6 Software visualisation using C++ lenses

In [18] Cain and McCrindle present a class inheritance hierarchy browsing tool for C++ programs, utilising a lens technique. Their tool presents a class inheritance hierarchy as a tree, with the names of the classes appearing at the nodes of the tree as textual labels.

Their system, illustrated in Figure 47, then applies various “lenses” to the tree to adjust the font size of the class name labels. With no lenses applied, all the class name labels will be displayed in the same font size. By applying a lens to the tree the class name labels are displayed in a font size that is proportional to the

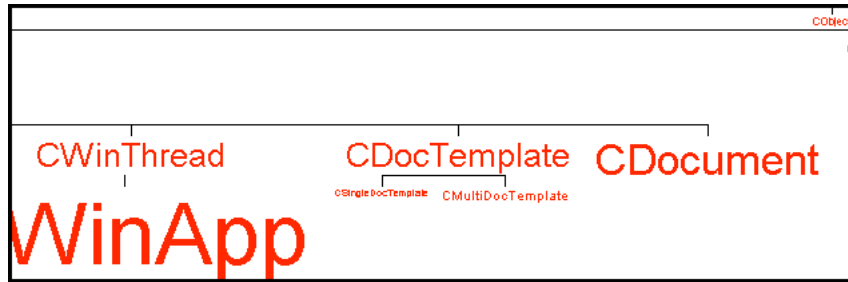


Figure 47: An illustration of the lens technique used by Cain and McCrindle.

importance of the class, as defined by the chosen lens, such that more important items are displayed in larger text than those items which are of little interest.

Lenses in this system use software metrics to quantify the importance of the various classes of the system being examined. Two such lenses are supplied with the tool.

- Reference Lens. This lens uses a metric that counts the number of references made anywhere in the program to a particular class. The most visible classes in the resultant hierarchy diagrams therefore have the most important interfaces in the program. This can allow for interesting observations about the design of a program. For instance if a base class is drawn small while its derived classes are drawn large, then the base class may not be acting as an effective interface and the dependencies are on the wrong objects.
- Uses Lens. This lens is used to answer the question “How easy will this class be to reuse in another setting?” and uses a coupling metric to rate the importance of a class. If a class is printed in a larger font, then it is tightly coupled with other system components and is therefore likely to be difficult to extract and reuse.

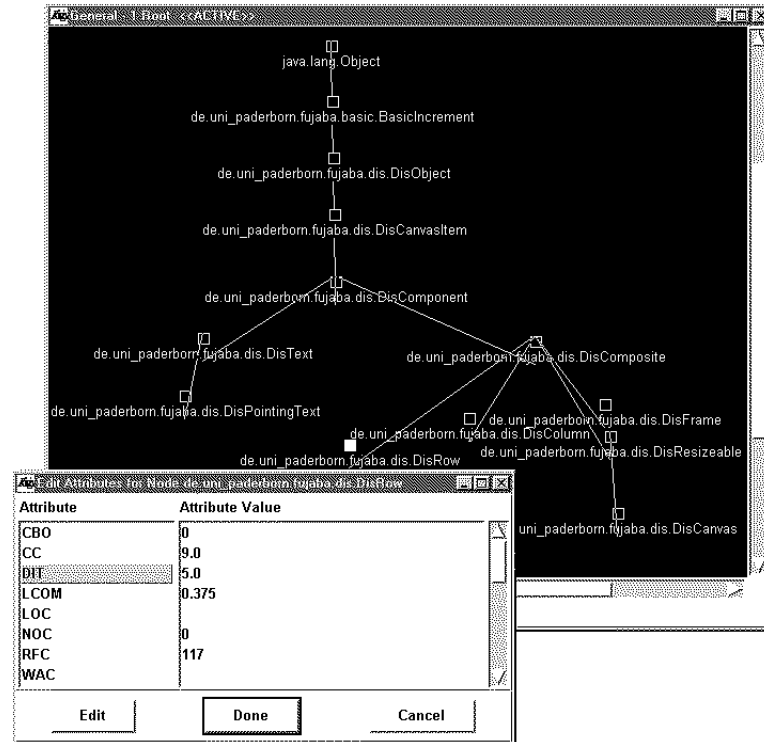


Figure 48: An illustration of the Shimba system.

6.5.7 Analysing Java software using metrics and visualisation

In [93] Syta, Yu and Muller introduce *Shimba*, a prototype reverse engineering environment for analysing Java software.

Shimba, illustrated in Figure 48, supports the exploration, visualisation and analysis of the structure of a Java program by extracting the program structure from the byte code of the program being examined. It then uses a graph model to represent information about software entities, relationships, attributes and the abstractions over them.

Queries and analyses can be encoded by the user into a script which operates on the dependency graph and the annotated object-oriented complexity measures taken from the program being examined. Scripts are a flexible and versatile method of allowing the user to customise the tool to explore and investigate the

measures in their own way, but does require the user to learn the scripting language.

Shimba extracts software artifacts such as classes, interfaces, methods, constructors, variables, and static initialisation blocks directly from the Java class files. It also extracts dependency information about these artifacts, such as implementation relationships between classes and their interfaces, containment relationships, call relationships, access relationships and assignment relationships. These artifacts are depicted visually as nodes and directed edges between nodes in a graph. Different types of nodes or edges are represented by different colours.

Software metric values are added to the display as attribute values. By default, the attribute values of nodes are not visible, but instead can be examined by selecting a node and opening a pop-up dialog for that node. This helps to avoid cluttering the display, but does not make the metric values as clear. However, this behaviour can be changed by writing the appropriate script.

6.6 Summary

This chapter has discussed the need for methods of visualising and exploring large and complex software systems, and has presented some of the methods discussed in the visualisation literature, such as pixel representations which gives a user a high level overview, and the notion of focus + context to help a user navigate and explore a system.

Using three dimensions to display visualisations has also been described in this chapter. Although an interesting area of research, it is not clear that such systems are significantly more useful than traditional two dimensional visualisations. This may partly be due to the necessity to render three dimensions on to a two dimensional computer screen.

This chapter has also presented a selection of visualisation systems, such as the often cited SeeSoft, and its precursor, SeeSys. SeeSoft shows how it is possible to display a large amount of information about software systems in a relatively small

space, and the idea of Information Murals provides a method to further compress the information, although it does highlight the necessity for very carefully choosing when to apply colour, and which colours to use.

Further tools, such as Tarantula and Shimba, show how combining visualisation systems with other program attributes, taken from test cases or metrics, for instance, can greatly simplify common tasks, such as evaluating the usefulness of test cases, or of determining when and where to apply re-engineering effort.

Many of the ideas and systems discussed in this chapter could be implemented for visualising and exploring Haskell programs. The next chapter presents some initial work in this direction.

Chapter 7

Software Visualisation for Haskell

The work in Chapter 4 presented a selection of metrics for use with Haskell programs. Metrics can generate a large amount of data for non-trivial programs and while the results can be manipulated with languages such as `perl` and UNIX tools such as `sort` and `grep`, it would be desirable to have a method of gaining an overview of the results without having to consider the actual data.

One common solution to this problem is to use visualisation techniques to graphically present the data from the metrics. Typically this might involve annotating some form of program visualisation with metric values.

The metrics that have been presented in this thesis all associate numeric measurements with individual functions, and thus all of them are suitable for use in a visualisation tool. Some of the metrics, such as those that measure attributes of patterns, capture features of the individual function and may therefore be best combined with a visualisation technique such as pixel representations. Other metrics, such as arc-to-node ratio, capture features which are associated with the interaction between functions and so may be better suited to visualisation using a callgraph, for instance.

Tools such as QuickCheck [21], an automatic testing tool, and HaRe [60], a refactoring tool, have been well received by the Haskell community. This suggests that there is a need for tools which help support an Extreme Programming style for functional languages such as Haskell. Software metrics in combination with

software visualisation can form a useful addition to this family, for instance, by acting as “drivers” for the targeted use of other tools.

As was described in Chapter 6, there are various ways in which software may be visualised. This chapter will present implementations of three basic software visualisation techniques for Haskell programs.

- Pretty printed source files
- Pixel representation of source files
- Graph representations of module hierarchy and callgraphs

Additionally this chapter presents some methods to visualise the evolution of a Haskell program over time. The rest of this chapter is divided into the following sections.

- Section 7.1 presents designs for a number of visualisation techniques which could be used to increase the usefulness of metrics for Haskell programs.
- Section 7.2 describes how the visualisation systems were implemented, and discusses some of the problems encountered in doing so.
- Section 7.3 summarises the chapter.

7.1 Designing Visualisation Systems for Haskell

When considering how one might visualise metric values taken from a Haskell program, it is important to decide what goals or tasks the visualisation is attempting to achieve. This work attempts to provide tools for the following tasks which are common when attempting to comprehend large software systems.

- Exploring a single Haskell module to find functions with unusual metric values.
- Finding modules in a program which have unusual properties.

- Browsing the dependencies between functions and modules.
- Investigating the evolution of a module through its development history.

The remainder of this section presents designs for visualisation systems that address these tasks and is divided into the following parts.

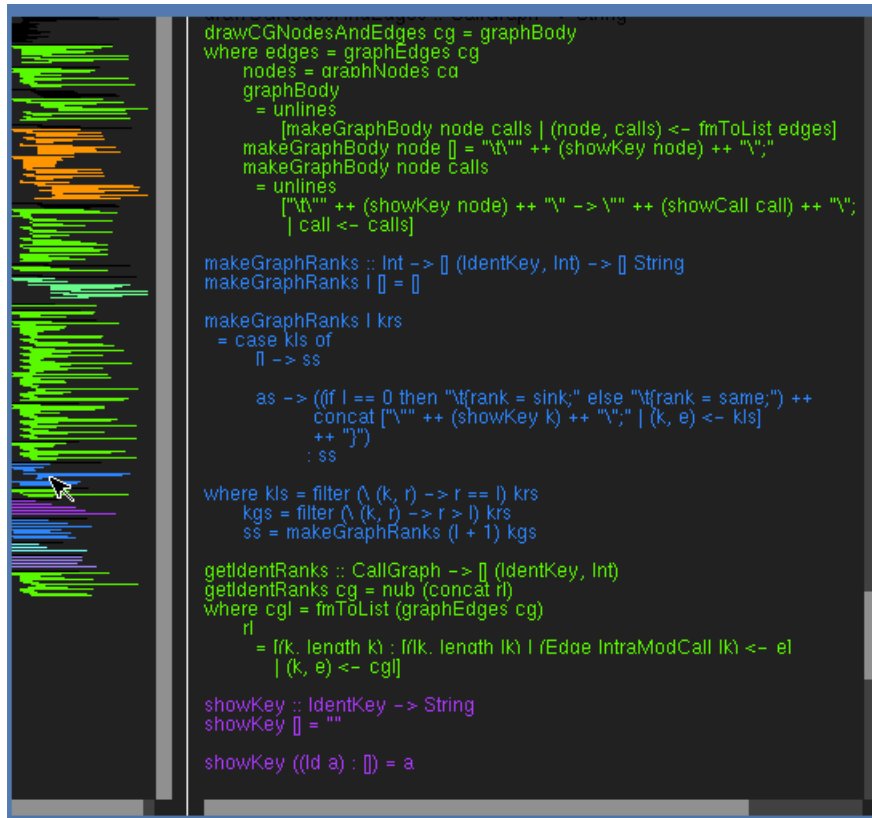
- Section 7.1.1 presents a visualisation technique for exploring a single Haskell module which uses greeking to reduce the amount of screen space a source text occupies.
- Section 7.1.2 shows how a large number of files may be examined quickly and easily using multiple pixel representation views.
- Section 7.1.3 describes a system to explore the module hierarchy and call-graph of a Haskell program.
- Section 7.1.4 discusses a method of examining how a single source file evolves and changes over time.

7.1.1 Exploring a single Haskell module

When attempting to inspect a single module of a program it can sometimes be difficult to maintain a high level overview of the module while still viewing low level details. Chapter 6 presented a widely used mechanism known as *focus + context* for addressing this issue.

The focus + context visualisation presented in this chapter uses a pixel representation of the module being explored as a scroll bar for a source code view, such that clicking on parts of the pixel representation would move the source code view to the corresponding location. Using a pixel representation rather than a real scroll bar provides the user with a degree of context, by allowing them to see the overall “shape” of the source code text.

Metric values are incorporated into the visualisation by colour coding the source code and pixel representations such that individual functions in the module being explored are coloured according to their metric value. The exact colour



```

drawCGNodesAndEdges cg = graphBody
where edges = graphEdges cg
      nodes = graphNodes cg
      graphBody
        = unlines
          [makeGraphBody node calls | (node, calls) <- fmToList edges]
      makeGraphBody node [] = "\t" ++ (showKey node) ++ "\n"
      makeGraphBody node calls
        = unlines
          ["\t" ++ (showKey node) ++ "\n -> \t" ++ (showCall call) ++ "\n";
          | call <- calls]

makeGraphRanks :: Int -> [] (IdentKey, Int) -> [] String
makeGraphRanks l [] = []

makeGraphRanks l krs
  = case kls of
      [] -> ss
    as -> ((if l == 0 then "\t(rank = sink;" else "\t(rank = same;") ++
      concat ["\t" ++ (showKey k) ++ "\n"; | (k, e) <- kls]
      ++ "\n")
      : ss

where kls = filter (\ (k, r) -> r == l) krs
      kgs = filter (\ (k, r) -> r > l) krs
      ss = makeGraphRanks (l + 1) kgs

getIdentRanks :: CallGraph -> [] (IdentKey, Int)
getIdentRanks cg = nub (concat rl)
where cgl = fmToList (graphEdges cg)
      rl
        = [(k, length k) | (k, length lk) | rEdae IntraModCall lk <- e1
          | (k, e) <- cgl]

showKey :: IdentKey -> String
showKey [] = ""

showKey ((Id a) : []) = a

```

Figure 49: Final version of the focus + context tool. The pathcount metric is used to colour code functions in this example, although it is of course possible to use other metrics instead.

scale used is not specified, and should be chosen to suit the individual metric being examined. An illustration of an implementation of this design is shown in Figure 49.

This basic visualisation could be further enhanced by the inclusion of a dynamic query system for adjusting the colour scale and for hiding common values, much like that used in SeeSoft and described in Section 6.5.2 of Chapter 6.

7.1.2 Finding modules in a program which have unusual properties

Section 7.1.1 presented a visualisation system that allows a single module to be examined. However one still needs to discover which modules, or files, in a multi-module program may need closer inspection.

One method of achieving this is to display all the modules simultaneously, thereby allowing one to visually browse the collection of modules before focusing on those of interest.

In order to display many modules on screen at once it is necessary to use techniques such as pixel representations to reduce the amount of screen space required by each file. SeeSoft, described in Section 6.5.2 of Chapter 6, uses a system of tiled pixel representations as one method of displaying multiple files.

The representation used by SeeSoft uses variable sized tiles for each source file, such that long source files have long tiles and short source files have short tiles. This allows one to immediately see the relative sizes of the files, but can complicate the layout of the tiles if the system is to fit the maximum number of files into the available screen space. One drawback of this system is that the layout of the tiles may change if the sizes of the files change, as tiles may flow from one column to the next as tiles shrink or expand. The effect is much like that occurring when adding extra words to a line in a word processor, with other words moving to different lines as the text is modified. This can make it hard to find specific files, for instance, when comparing two versions of a program.

In this work, a similar mechanism to SeeSoft is suggested, but in our design every file is represented by a tile of a fixed maximum size, although tiles smaller than the maximum size are left unchanged. This allows for a simpler and more consistent layout of tiles, with files being laid out left to right, top to bottom in alphabetical order. The fixed maximum size of the tiles means that the layout of the tiles will only change if files are added or removed, but does mean that very

small areas of colour may not be seen, due to the scaling of the pixel representations. If a file cannot be parsed¹, it is represented by a rectangle containing a red cross. Our implementation is illustrated in Figure 50.

Clicking on a rectangle highlights that tile and displays the name of the corresponding module at the bottom of the screen. This mechanism could be extended in many ways. For instance, the visualisation could display the module name when the mouse is held over a tile and could use the mouse click to start up a more focused visualisation tools on the selected module.

This browsing tool also has applications in refactoring [36], where one often wants to know how much of the program code will be modified by a refactoring such as changing the name of a function. This tool can provide an effective means of displaying the areas that will be modified by using a simple binary colour scheme. Such a display is illustrated in Figure 51, which shows all the functions which use the `map` function in red, while displaying all other functions in green.

7.1.3 Browsing the dependencies between functions and modules

When confronted with large programs written in Haskell it can be hard to obtain a coherent overview of the structure of the program. Pixel representations can be useful for exploring metric values, but do not convey the structure of a program.

Callgraphs can provide a convenient way to display structure information, particularly for functional programming languages in which the function call is such an important aspect. However a naive use of callgraphs, particularly for large programs, can suffer from “information overload” because the large number of functions and calls between them can cause a callgraph to become a jumbled mess of edges. This section presents a callgraph and module browser which provides a method of hierarchically browsing callgraphs, which attempts to minimise the

¹There are several reasons why a file in a project might not parse, but which do not indicate an error, such as the use of CPP pre-processor macros in a source file, or the use of language extensions not supported by our current parser.

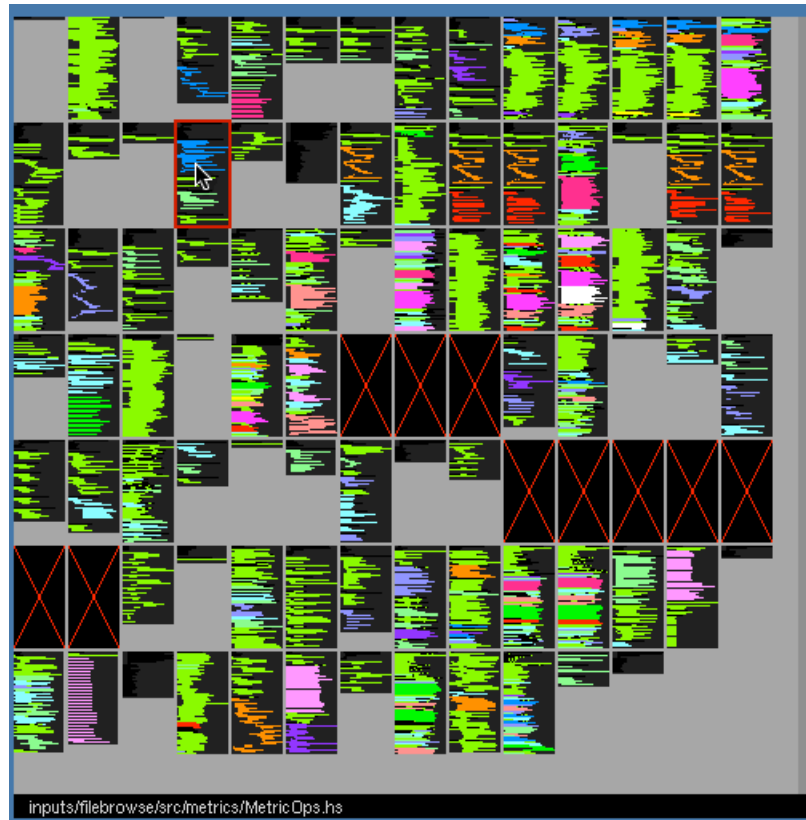


Figure 50: A tool for browsing multiple source files. This example use pathcount for colour coding the functions in the source files, although other metrics can be selected.

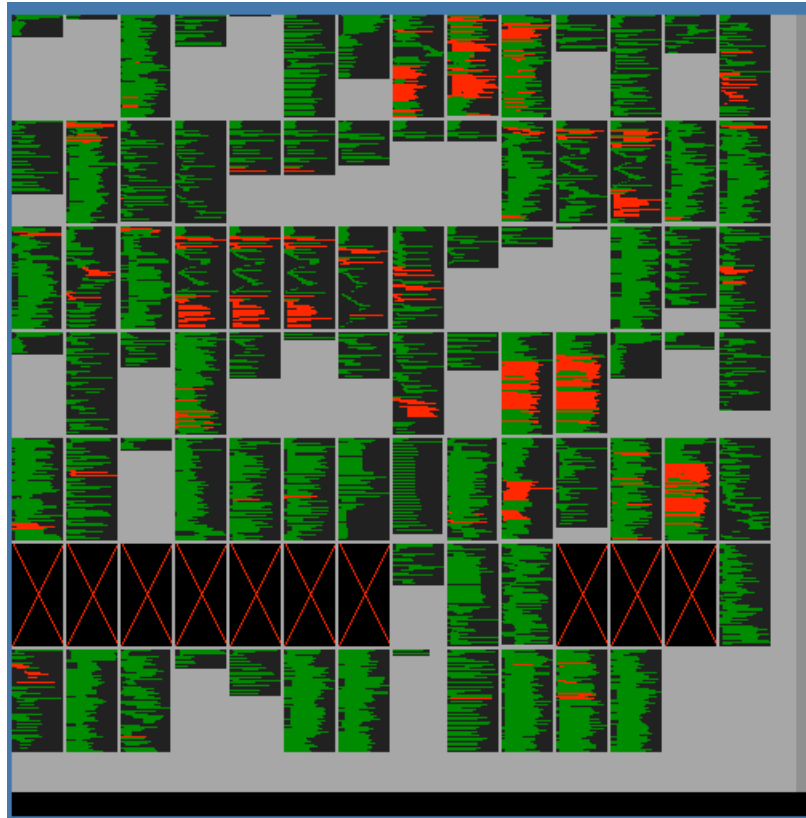


Figure 51: A file browser tool showing all functions which use the `map` function red. This is achieved using a simple binary metric to indicate usage of the `map` function.

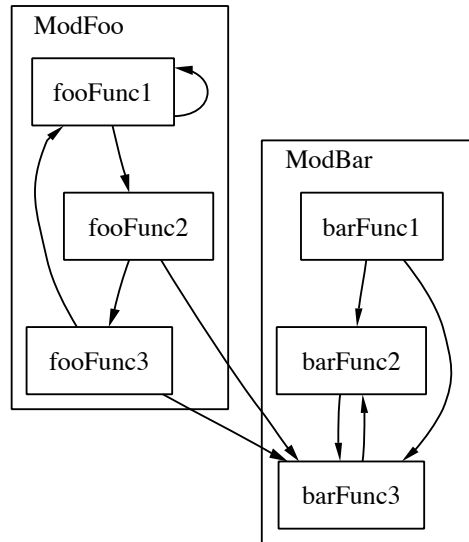


Figure 52: Example of a callgraph for two modules.

problem.

Why hierarchical browsing?

One of the first questions that comes to mind when exploring a program is “what calls what?”. A common way to answer this question is to use callgraphs, as was shown in Section 6.5 of Chapter 6 of this thesis. Callgraphs represent functions or other similar objects, such as type constructors, as nodes in a directed graph. Calls between functions are shown as edges between nodes.

Callgraphs can be expanded to include calls between functions in different modules of the program. In such a graph it makes sense to partition the graph so that the nodes that make up functions in a single module are clustered together in some way. Figure 52 shows an example of this type of callgraph.

However, for complicated or large programs this type of callgraph can quickly become impossible to read, with many parallel or crossing edges which make it difficult to trace edges accurately, as shown in Figure 53. This problem was also described in Section 6.4 of Chapter 6 of this thesis. In these circumstances it makes sense to split such “flat” callgraphs into a hierarchy of callgraphs, so that it is

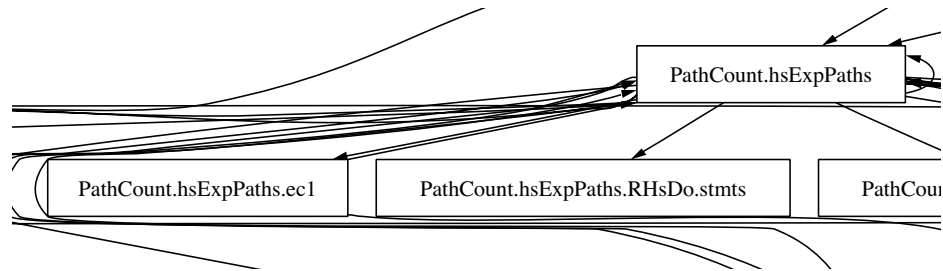


Figure 53: Part of a callgraph in which edges are hard to distinguish.

possible to obtain a top level overview while hiding some parts of the callgraph.

Dividing a callgraph

When deciding how to split a callgraph into sections it makes sense to think of there being two levels. At the top level there is the import graph which is a graph in which nodes represent modules and edges represent import statements. At a level below this there are callgraphs for individual modules. Having decided how to partition the callgraph it is necessary to decide how the irrelevant sections of the graph may be hidden from view while the relevant sections are highlighted.

One intriguing method is to have the nodes of the import graph expand to become callgraphs for their corresponding module when clicked upon. However, if one wishes to view callgraphs for multiple modules simultaneously the graph display may quickly become messy and difficult to navigate, suffering from the same problems described earlier in the previous section.

An alternate approach is to have callgraphs for modules appear in separate windows when their nodes in the import graph are clicked. This allows the callgraphs of several modules to be displayed simultaneously while leaving the import graph clear. The disadvantage of this method is that calls between functions in different modules are not shown as clearly as they may be if all the callgraphs are displayed in a single graph, as in the previous suggestion. This approach can also result in display clutter if many windows are opened simultaneously, although this is somewhat mitigated by the likelihood of the user already having their own

method of managing multiple windows, and thus not requiring them to learn a new method.

The import graph provides a useful high level overview of the structure of a program, but on its own this can be rather uninformative. To increase the usefulness of the import graph the edges can be decorated with additional information, such as the number of exported or imported symbols, or the number of times symbols from the imported module are used.

It is important not to provide too much information on the edges, otherwise important aspects may be lost amongst a mountain of routine information. For this work the following information was chosen.

- The number of symbols used from an imported module.
- The number of symbols exported from the module.
- The list of symbols involved.

However, this choice is an implementation choice and as such it should be possible for the user to specify the information they wish to be displayed, for instance, the user may wish to specify a metric to be used for the decoration.

There is a distinct possibility that the list of symbols could contain too much information for edge decoration to be viable. Because of this it may be wise to split the display in two, with the left hand side showing the import graph in which edges are decorated with labels showing the number of used symbols and the number of exported symbols, while the right hand side displays the list of symbol names when an edge was clicked.

Two problems remain with this system. Firstly, adding labels to the edges in the graph can result in a very cluttered display, and secondly, callgraphs can become hard to interpret when the layout engine places several edges close together. Solutions to both of these problems are related.

Labels on edges can be avoided by using a system of “mouse overs”, where holding the mouse cursor over an edge shows the information for that edge. This

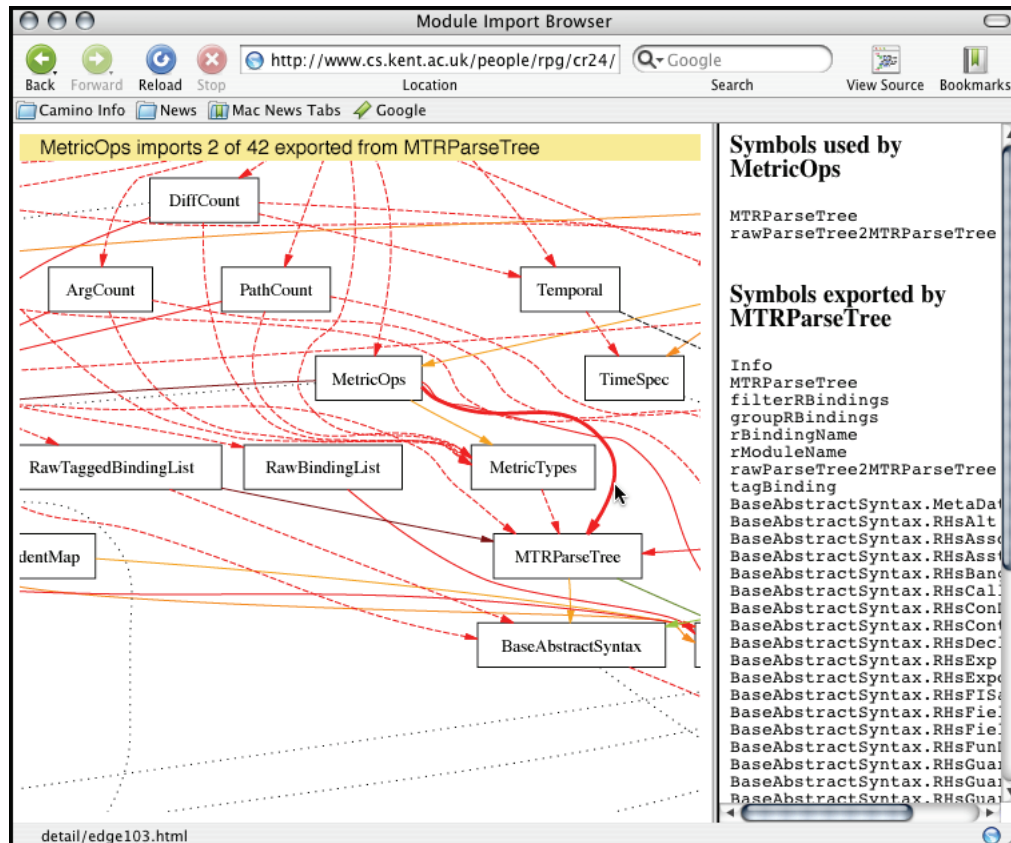


Figure 54: An illustration of mouse overs in the module and callgraph browser. Edges are colour coded according to how many symbols are imported along them.

is particularly useful if edges are colour to give a brief indication of the importance of the information they are decorated with.

Mouse overs can also be used to address the second problem, that of groups of edges being close to each other and therefore hard to trace. As well as showing information for an edge when the mouse cursor is held over it, it is also possible to redraw the edge in a thicker line style. This greatly increases the ease of following the path of the edges, particularly in the case of overlapping or grouped edges. These two uses of mouse overs greatly increase the usefulness of callgraphs.

The visualisation system outlined in this section is illustrated in Figure 54 in the Safari web browser.

7.1.4 Investigating the evolution of a module

Section 7.1.1 presented a design for a visualisation system using pixel representation views, colour coded with metric values, for exploring a single source file. This gives a tool that may prove useful for identifying places in a file which show unusual characteristics, or where one might wish to target development or testing effort. However, such a system does not easily help answer questions about the evolution of a source file.

For instance, one might discover that some part of a source file is particularly complex but it may not be clear why that is the case and so it may be useful to find out when it became complex, e.g. after feature X was added, or when function Y was refactored, etc. To answer this type of question with the pixel representation source code view described in Section 7.1.1 requires manual intervention to gather all the necessary versions of the source file, and does not allow any easy way to browse multiple versions. Because of this it is interesting to investigate an alternative solution.

If the source code of the program being explored is stored in a revision control system such as CVS [35], it is possible to programmatically extract individual versions of source files. This mechanism can form the basis of a visualisation system, but careful consideration of how to display these various versions is still needed.

One possible mechanism is to use animation to show each version of a source file as a frame of an animation. However, as was described in Section 6.3.1 of Chapter 6, this can often be unhelpful because large changes between version of a source file can make it hard to keep context between frames of the animation.

Section 6.3.2 of Chapter 6 discussed the technique of bracketing. Bracketing is a system which addresses the issue of maintaining context between frames of an animation by extending the idea of focus + context displays. Bracketing systems show the *context* in the time sequence by showing the “next” and “previous” time slots which appear either side of the *focused* time slot.

The system proposed here and illustrated in Figure 55, brings bracketing and

focus + context displays closer together by using a focus + context display for the “current” time slot, and using bracketing to supply the context in the time sequence.

The currently selected snapshot is displayed as full size text in the top left pane, and as a larger scale pixel representation in the top right pane. These two panes provide the same focus + context functionality as the pixel representation described in Section 7.1.1, allowing the user to jump to particular parts of the file by clicking in the pixel representation.

The two medium scale pixel representation panes on the lower right hand side provide the bracketing by displaying the snapshots either side of the currently selected snapshot in the sequence.

This tool allows one to explore the evolution of a source file, using the time line at the bottom of the screen to quickly identify where unusual characteristics have been introduced. One possible future addition to the system might be the ability to play the sequence of snapshots as an animation, however as was discussed in Section 6.3.1 of Chapter 6 of this thesis, if there is too much change between frames of an animation the result can be confusing.

Another feature which may be a useful addition to the system is integration with a visual diff tool, such as VDiff [101] or TkDiff [72], which displays the changes between two files. Integration could occur in several ways, for instance, one might indicate what has changed between successive snapshots by using a VDiff style display in the timeline frame at the bottom of the screen, or one might provide a mechanism to select two snapshots, which could then be passed to VDiff for display.

7.1.5 Summary

The previous sections have outlined mechanisms that can be used to visualise metrics to address four common tasks that are faced when attempting to understand large software systems. In order to test the ideas discussed in this section it is necessary to implement these systems. The implementation of these systems,

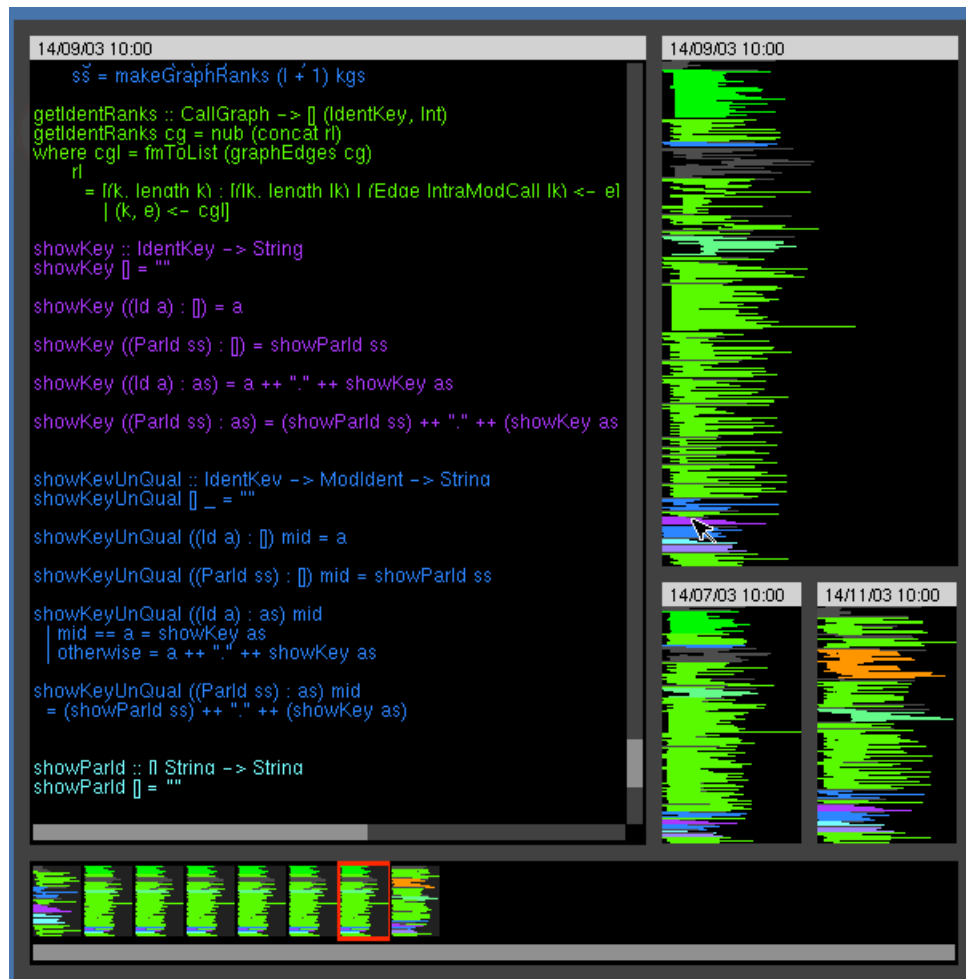


Figure 55: A browser for exploring the history of a source file. Functions are colour coded using the pathcount metric, but any of the Haskell metrics presented in this thesis can be used.

and the issues encountered while doing so, are described in the following sections.

7.2 Implementing Visualisation Systems for Haskell

When considering the implementation of software visualisation systems there are a number of issues that must be considered. These were discussed in detail in Chapter 6, but are briefly enumerated here for reference.

- Visualisation systems should be scalable to cope with large programs.
- Visualisation systems should be flexible and should not constrain a programmer to a specific method of working. Ideally a programmer would be able to customise the visualisation system to their own needs, as discussed by Oudshoorn and co-workers [74], Knight and Munro [57], and Sidarkeviciute [88].
- Visualisation systems should be portable, in order not to impose unnecessary changes or restrictions upon a programmer.

When examining how visualisation systems can be made flexible, a number of interesting possibilities arise. In [88] Sidarkeviciute makes the point that the potential users of program visualisation tools are most probably programmers themselves, and should therefore be capable of specifying their own visualisation systems.

An obvious method of leveraging the expertise of the users in order to increase the flexibility of the visualisation systems is to implement the systems as a library for a programming language. The programming interface, or API, then provides the core of a domain specific language in which the programmer can program their own visualisations. One can then provide a selection of visualisation systems, each of which would consist of little more than a veneer over the library. If the library API is well designed and documented, the users can gain tremendous flexibility to modify or combine existing visualisation systems or to create their own.

Addressing the portability of a visualisation system consists of two main issues, the language used to implement the system, and the method of displaying the results. For instance, one might use a platform independent language, such as Perl, and generate GIF or PNG images as output. Such a system would be portable, but may be of limited use due to the lack of user interaction with the output. In order to accommodate interaction in a visualisation system, it is probable that the implementation will require the use of some form of GUI toolkit to display windows, menus and such like, and so it is therefore important that this is taken into consideration when selecting an implementation language.

Storey, Fracchia and Mueller [91] identify that software exploration tools have much in common with hypermedia document browsers, such as web browsers. This raises an interesting question, if software exploration and visualisation tools have much in common with hypermedia browsers, why not implement visualisation tools within such a system?

Modern web browsers have numerous means of supporting interactive displays, such as Flash and SVG plugins, and are supported on many computing platforms. Using a web browser for the display system removes the issue of cross platform portability, and so web browser based implementations seem an ideal mechanism. However, it is important to realise that web browsers were not designed for implementing complex GUI systems, so using web browsers for visualisation tools may stretch the capabilities of the browsers.

7.2.1 An initial design

Taking the dual issues of portability and flexibility into consideration, along with the points outlined above, our initial design was to implement the visualisation systems as a library of Haskell modules, utilising web browsers as the display medium.

The choice of Haskell as the implementation language has two main advantages. Firstly, Haskell is a cross platform language and secondly, the purpose of the visualisation systems is to analyse Haskell software and so therefore the

potential users will be familiar with Haskell. Implementing the library in Haskell allows users to modify and implement their own custom visualisation systems in the language they are most familiar with.

7.2.2 Using a web browser as a display engine

The initial goal when implementing the visualisation systems presented in this chapter was to enable their cross-platform use by leveraging the functionality provided by web browsers. This was first achieved by generating HTML files and associated GIF images to produce a static web page. However, as the various visualisation systems evolved it became necessary to include more sophisticated use of interaction than could be provided by static HTML. This extra interaction was first implemented by generating SVG files. SVG [30], or *Scalable Vector Graphics*, is a format for displaying vector graphics on the web and is standardised by the W3C [97], the organisation responsible for standardising the web. SVG files contain support for interaction using scripting with JavaScript [34] and can be opened in most common web browsers with the aid of a free plugin.

Unfortunately, although SVG supports a fair degree of interaction it can be complex to implement user interface elements such as scroll bars, and can suffer performance problems when working with large numbers of elements. This was a particular problem when implementing the module browser, discussed later in Section 7.2.4, which generates graph structures containing many lines and boxes.

Because of these problems it was necessary to investigate alternative methods of implementing the visualisation systems. The ideal candidate would be a graphical user interface library for Haskell with cross-platform support. Unfortunately the available libraries at the time of writing are neither sufficiently cross-platform, nor robust enough for our purposes. It was therefore decided that HOpenGL [76], a Haskell binding to the OpenGL graphics libraries, would be used.

HOpenGL is cross-platform, robust, and is high performance. Although quite primitive in its user interface capabilities, it nonetheless provides all the support needed for the visualisation systems presented in this chapter.

The issues arising from the implementation of each of the visualisation systems presented in Section 7.1 will be discussed in the following sections.

7.2.3 Implementing a tool for exploring a single Haskell module

Section 7.1.1 presented a design for a visualisation tool for exploring a single module. The implementation uses a pixel representation of a source file as a scrollbar, providing the *context* and a pretty printed version of the same source file to provide the *focus*. Both the pixel representation and the pretty printed source code are colour coded, such that each function in the file is coloured according to its metric value.

This system initially used a web browser as a display engine by generating a selection of HTML and GIF files and using client side image maps to enable mouse clicks in the pixel representation GIF image to scroll a source code frame. This is illustrated in Figure 56 in the Safari browser for Mac OS X. A system such as this allows one to quickly examine a source file and spot areas with unusual characteristics in the pixel representation, and then to jump to those sections by clicking on them.

Later implementations of this system were implemented using HOpenGL in a modular way, such that the components of the visualisation such as the pixel representation or the source code frame, could be reused as components of other systems. The final implementation of this visualisation was illustrated in Figure 49 on page 227.

7.2.4 Implementing a tool for browsing the dependencies between functions and modules

Section 7.1.3 outlined a visualisation system for browsing the dependencies between function and modules in a Haskell program. To implement this program the

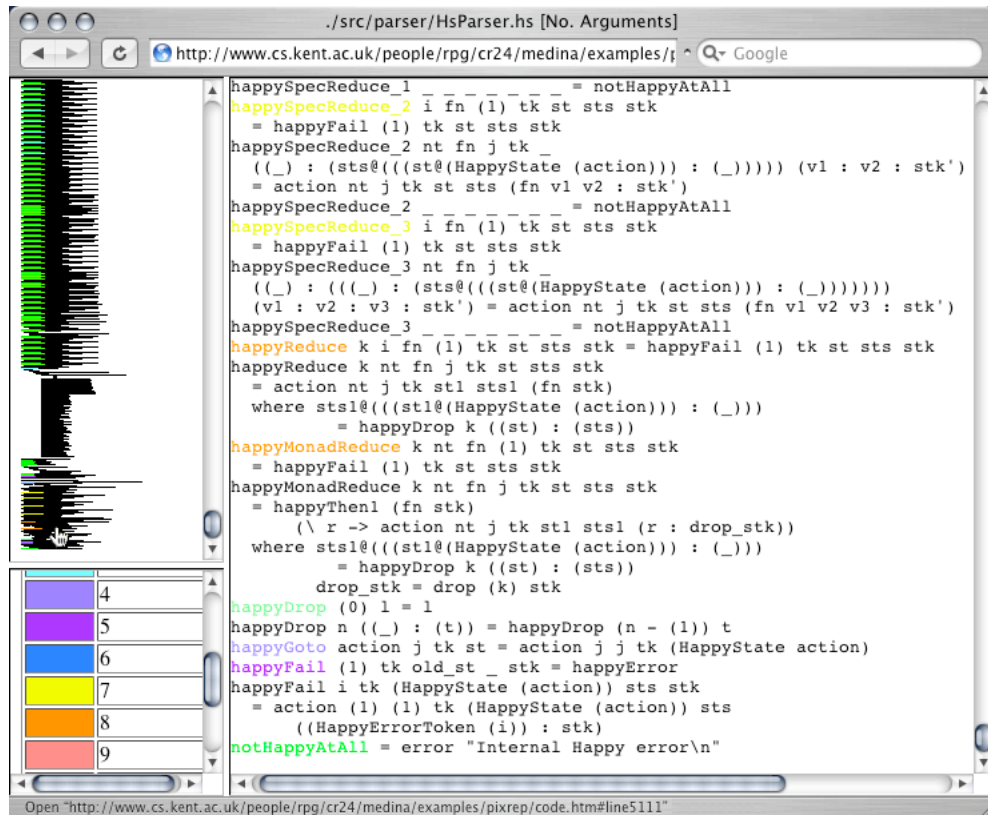


Figure 56: First version of the focus + context tool. Colour coding is performed using the pathcount metric, although other metrics could be used.

first decision to be made was the choice of layout engine for the graph. The visualisation could have used a system in which nodes that represent modules might expand or contract to show or hide the details of the modules. However, such a system can be confusing for the user, and also requires some form of dynamic graph layout algorithm because it necessarily involves modifying the layout of the import graph as nodes are expanded or collapsed. The alternative solution put forward by the design in Section 7.1.3 is to have the callgraphs for the individual modules appear separately when their corresponding nodes are clicked. This method requires only a static graph layout.

To implement this system a graph layout tool called `dot` [39] was used. `dot` is part of the GraphViz [31] suite of tools from AT&T Research Labs. `dot` reads a description of a graph in the DOT [27] language and can generate several image formats as output, as well as image map information that can be used within a web browser. The image map information lists the coordinates of areas of the graph that may be clicked upon. In this work, that is the coordinates of the edges and the coordinates of the boxes around the nodes.

To generate the callgraph browsing output the initial version of the visualisation tool produced a DOT file from the callgraph and used the `dot` tool to make a GIF format image of the import graph and to produce image map information. The tool then used the files generated by `dot` to produce a collection of HTML files, each representing the callgraph of one of the modules, which could be loaded into a web browser to explore the module hierarchy and callgraphs, as illustrated in Figure 57.

Although this initial version worked reasonably well, as the import graph became larger it became harder to navigate the graph because there was no way of zooming in or out of the import graph. One of the alternate output formats supported by `dot` is Scalable Vector Graphics or SVG [30]. The plugins to view SVG images in a web browser support zooming and scrolling operations, and SVG also has built-in support for interaction. By switching to SVG format it was no longer necessary to generate separate image map information, since this was embedded

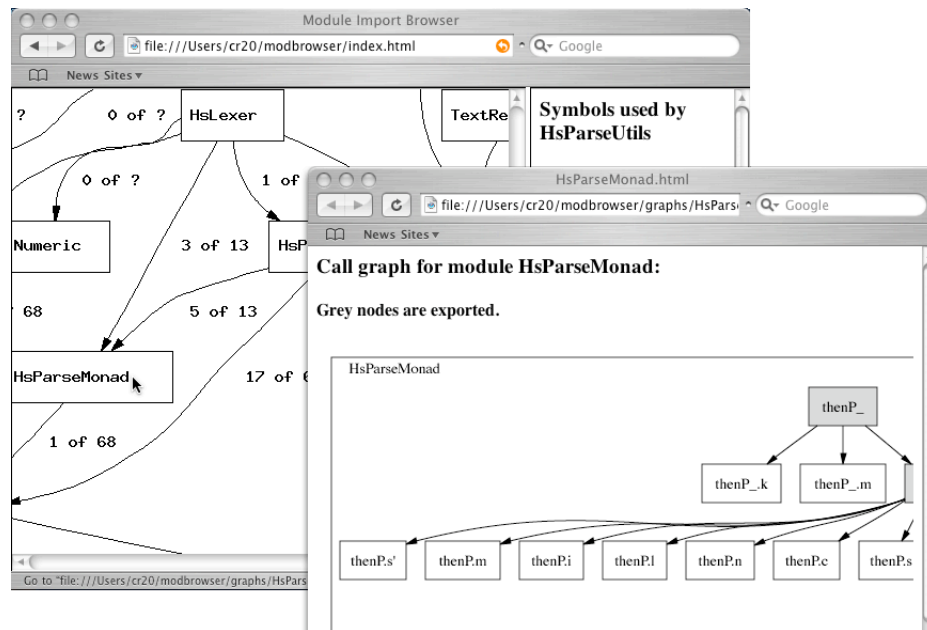


Figure 57: The first version of the module and callgraph browser.

in the SVG file by the `dot` tool.

While experimenting with import graphs that had a large number of edges it was noted that the labels attached to the edges often made the graph much more cluttered. One way to deal with this would be to remove the labels altogether, but an alternative is to have this information displayed only when the mouse is hovered over the edges. However, without the labels on the edges it is difficult to tell which edges are interesting. To combat this, the ratio of used symbols to exported symbols is used to colour code the edges, although it would, of course, also be possible to use any suitable metric value to determine the colour.

Additionally, edges connected to module nodes that have not been parsed, either due to a parse error or being unable to find the modules source file², are displayed with a dotted line and edges on which zero symbols are used are displayed as a dashed line. This is illustrated in Figure 58.

To provide additional feedback to the user about which edge is currently being

²Many Haskell libraries are distributed in binary form, and as such source code for them may not be available to the visualisation system.

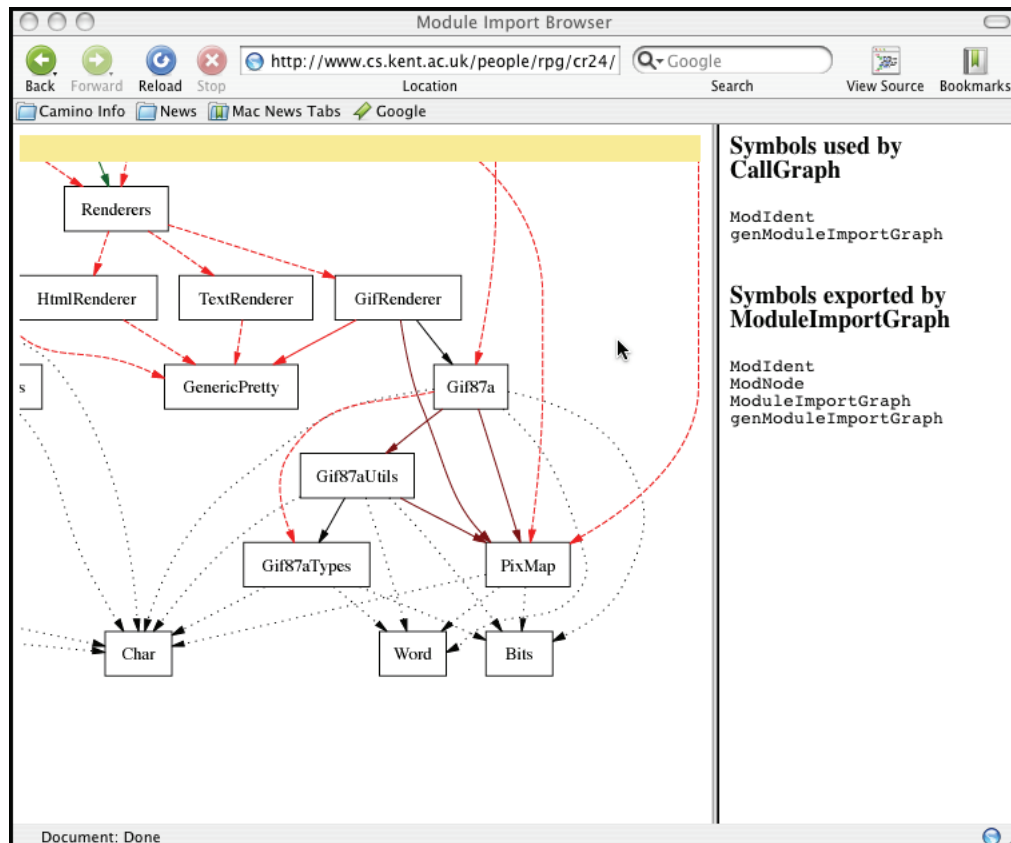


Figure 58: The module and callgraph browser implemented in SVG using colour and line styles.

inspected, edges become wider when the mouse is hovered over them. These *mouse overs* make it significantly easier to inspect parts of the graph where there are many interwoven edges. This was illustrated in Figure 54 on page 235.

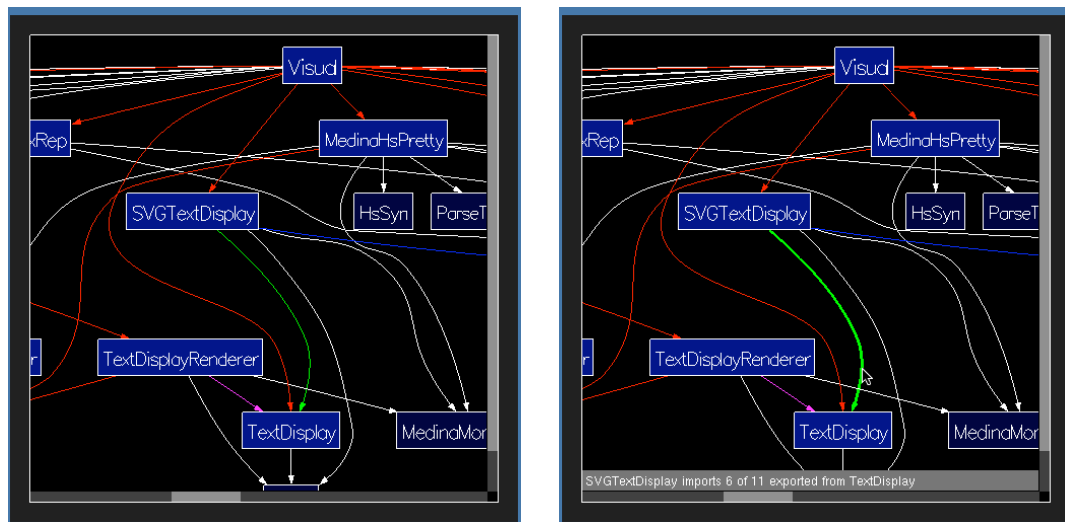
Unfortunately, the mouse overs can cause a large processing overhead when inspecting non-trivial callgraphs, resulting in significant delays between the mouse cursor being positioned over an edge and the appropriate mouse over action occurring. More unfortunately, there is no way to optimise such systems in an SVG document, because these issues are controlled by the implementor of the SVG plugin.

Because of this, and also because of the use of HOpenGL to implement the other visualisation systems described in this chapter, the module browser was reimplemented using HOpenGL which allowed considerable effort to be focused on reducing the overhead of processing mouse over actions, resulting in a more responsive user interface. The final version of the module browser is illustrated in Figures 59, 60 and 61.

When reimplementing the module browser in HOpenGL, the first issue that needed to be addressed was how to layout and draw the graphs. Previously graph layout had been performed by the `dot` tool, which generated all the required images. However, for the HOpenGL implementation all the line drawing must be performed using calls to HOpenGL, so it was necessary to find out the coordinates of the edges and nodes. Fortunately, as well as producing image formats, `dot` can produce a description of a graph, annotated with the necessary layout information. This information can then be read in via a UNIX pipe or an intermediate file and used to generate HOpenGL calls that will draw the graph.

Functions such as zooming and scrolling are all implemented using calls to the HOpenGL `transform` and `translate` functions, which on supported hardware may be performed in the graphics card giving very good performance.

The most crucial aspect in determining the user interface responsiveness was the support for mouse over actions. Delays between the mouse being placed over an edge, and the appropriate action taking place can make the system seem slow



(a) Basic import graph view

(b) Mouse over on an edge

Figure 59: The final version of the module browser. Colours for the edges are chosen based on the number of symbols imported and used along those edges. It would also be possible to use metrics to colour code nodes. The general colour scheme used, such as the background colour, etc, can be configured to the users taste and needs.

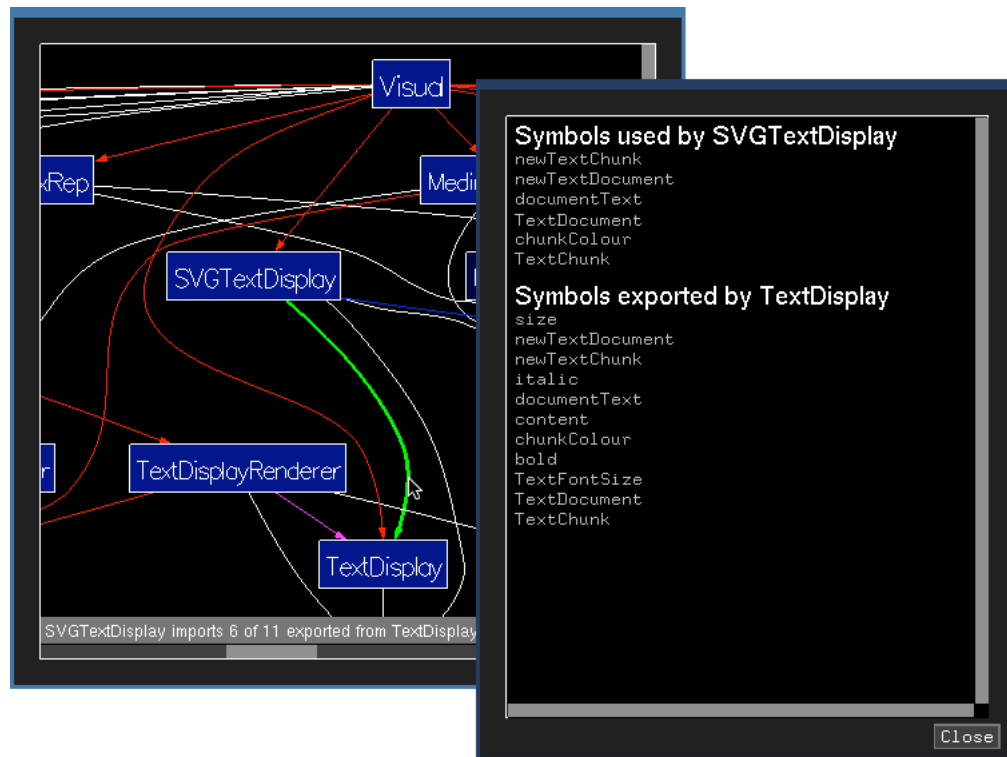


Figure 60: Popup window after clicking an edge.

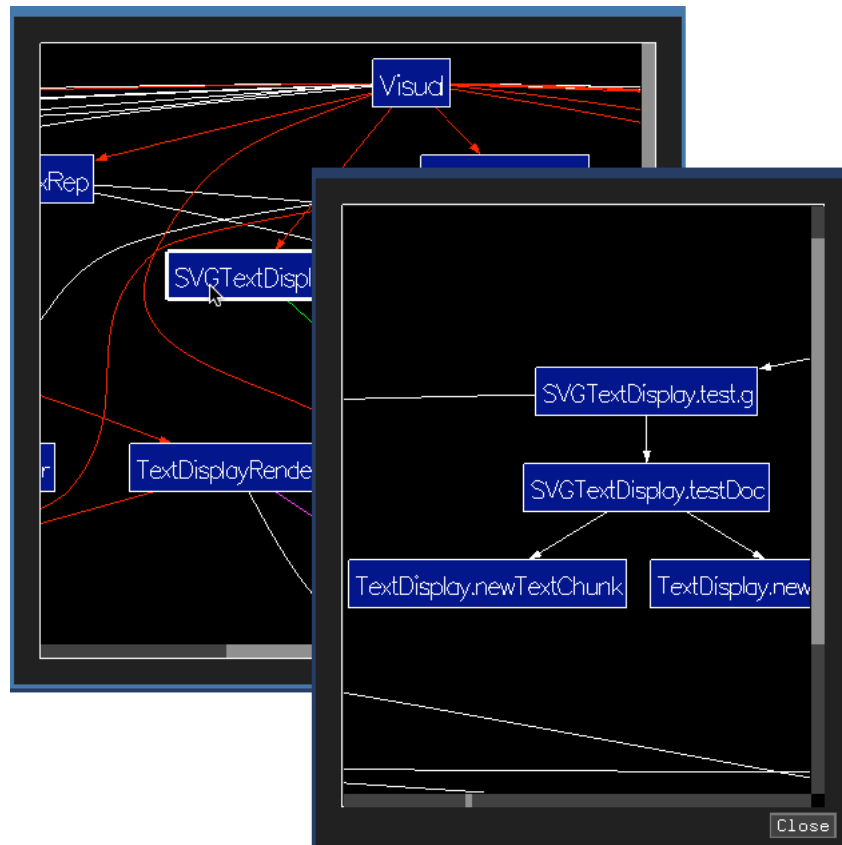


Figure 61: Popup window after clicking a module node.

and frustrating to use, and it is therefore important to reduce such delays.

HOpenGL supports mouse over actions using a simple method which involves placing the graphics pipeline into a *selection* state, in which drawing commands are not directed to the screen, but instead are remembered in a list of *selection hits* if they draw into pixels under the mouse cursor, or are otherwise ignored. The list of selection hits can then be retrieved to determine which drawing commands, and therefore which screen elements such as edges or nodes, are underneath the cursor at any particular time. This must be repeated every time the mouse cursor moves in order to update the list of selection hits.

The simplistic use of this mechanism is to redraw the entire graph each time the mouse moves, in order to determine if the mouse is over an item. This can result in unacceptable delays between the mouse moving and the appropriate action taking place. One way to optimise this method is not to draw the entire graph, but instead only the parts that are likely to be near the mouse cursor. This is achieved by calculating a bounding box for each edge and node when they are read from the graph description. When the graph is redrawn due to the mouse being moved, only those elements whose bounding box contains the mouse cursor location are drawn. This significantly reduces the number of calls to the HOpenGL library, and subsequently to the graphics card, resulting in greatly increased responsiveness of the user interface.

This type of optimisation is impossible to achieve in a format such as SVG and instead one must rely on the implementors of SVG plugins to perform similar optimisations. However, the performance problems seen with the module browser occur only because of the large number of mouse over actions required, which it appears, is an unusual requirement for SVG and other similar systems. Because of that, it appears this type of optimisation has not been applied, making formats such as SVG unsuitable for non-trivial visualisation systems.

7.2.5 Implementing a tool for investigating the evolution of a module

Section 7.1.4 discussed a visualisation system using bracketing to study the evolution of a module, which is illustrated in Figure 55. The implementation uses an interface to the commonly used revision control system CVS [35] to extract “snapshots” of the source file from the user’s CVS repository over a specified period of time. The range of dates and the time between snapshots is configurable from the command line, or alternatively it is possible to provide a list of “tags”, which are symbolic names attached to particular versions of the source file in the CVS repository. Each of these snapshots is displayed as a fixed sized miniature pixel representation at the bottom of the screen, with the currently selected snapshot displayed with a red border.

The user interface of the visualisation reuses the HOpenGL components of the pixel representation focus + context system described in Section 7.2.3. This reuse highlights the usefulness of implementing the visualisation systems as components of a library.

7.3 Summary

This chapter has outlined designs for several visualisation systems that can increase the utility of the software metrics that have been presented in previous chapters, and has shown that software visualisation tools are useful not only for exploring metric data, but also for answering questions raised during other software development activities such as refactoring or testing. There are many areas of software development that may benefit from such visualisation techniques, for instance, visualisation tools could be combined with refactoring tools such as HaRe [60], debugging tools such as Hat [20] or testing tools such as QuickCheck [21], somewhat like the Tarantula tool discussed in Section 6.5.3 of Chapter 6 of this thesis. The investigation of such combinations is left for future work.

As well as outlining designs for visualisation systems, this chapter has also presented a library containing implementations of these tools which can be combined with metrics to allow a programmer to both analyse and explore collections of files, using the file browser described in Section 7.1.2 and the callgraph and module browser described in Section 7.1.3, and individual files using the focus + context tool described in Section 7.1.1 and the bracketing tool described in Section 7.1.4.

The initial goal of implementing the visualisation tools within a web browser suffered from two main problems

- Combining elements from different visualisation tools, particularly the user interaction involving the various elements, can be complex within a web browser. This may be because browsers are not designed in general to support complex GUI interactions, although this is starting to be addressed by new standards such as XForms [28].
- The performance of complex displays, such as the callgraph and module browser tool, can be difficult or impossible to optimise, resulting in poor user interface performance in some situations.

Because of these problems the tools are currently implemented using the Haskell OpenGL binding, HOpenGL. While this offers good performance, due to OpenGL's support for hardware acceleration and low level API which allows for further optimisations, the GUI features provided are somewhat primitive. Future implementations may use one of the emerging cross platform GUI libraries, such as wxHaskell [59].

Chapter 8

Conclusions and Further Work

This thesis has discussed the twin topics of software measurement and visualisation. These topics go hand-in-hand but do not exist in isolation and are instead part of a wider software development process.

Software measurement is recognised as an important tool in imperative and object-oriented software development, which is indicated by the large body of research in those areas, and is used in practice by many companies, as was discussed in Chapter 2. However, functional programming languages such as Haskell currently lack a widely adopted software engineering process and so correspondingly there has been relatively little research into software engineering topics.

The emergence of Extreme Programming [13] as a development methodology in the object-oriented community raises interesting parallels to the ad-hoc methodologies often used to develop Haskell programs, such as the focus on very short iterations of the development cycle, on working from a functioning prototype as early as possible, on performing unit tests as early and as often as possible, and on refactoring code whenever the opportunity arises. The interest in tools such as the Haskell refactorer HaRe [60] shows that the Haskell community is receptive to tool support for such methodologies, which suggest that measurement tools may be equally valuable.

8.1 Summary of Conclusions

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 has been explored.
- Visualisation tools have been developed for further exploring the metric values in conjunction with program source code.

The work presented in this thesis, in particular the analysis of Chapters 4, 5 and 7, leads to a number of conclusions which are summarised here.

- 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. Analysing the correlation between metrics led to the following observations.
 - The occurrence of non-trivial recursion, e.g. recursion in which a function does not directly call itself, in Haskell programs is quite unusual, and is associated with complex program behaviour. However the occurrence of trivial recursion, where a function directly calls itself, is common.
 - The callgraphs of individual functions generally grow uniformly in both depth and width, rather than becoming long and thin, or short and wide.

- Large functions tend to include a greater number of local declarations than small functions. This is most likely because local declarations allow one to attach names to parts of a large, perhaps complex, function. Therefore functions which are large, but do not have many local declarations may well be difficult to understand.
 - The functions used by a single function, `foo`, will tend to be located close to `foo` in the source files.
- 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 “Outdegree” (*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. This suggests that there is no single attribute that makes a Haskell program complex, rather that the complexity is a result of a number of attributes.
 - Typically the metrics presented generate values that are distributed at the low end of their scales, e.g. $mode \leq median \leq mean$, suggesting that it should be possible to select thresholds to indicate when the various attributes are exhibiting unusual values, because any values above a threshold are very likely to indicate unusual behaviour that may warrant further investigation.
 - Software metrics can generate large amounts of data and so there is a clear need for tools that can present the interesting points of the data to the user, while hiding the bulk of the uninteresting data.

Unfortunately, analysing program change histories in order to perform statistical analysis of the relationship between metric values and the number of bug fixing or refactoring changes is time consuming.

Therefore, although we were able to observe some correlation between metric values and changes, and were able to show that typically metric values were towards the lower end of their scales, we are unable to suggest values for thresholds above which it may be advisable to re-engineer the offending code section.

Visualisation tools are an important addition to the metric tools, and as such this thesis has presented a selection of visualisation tools which can be used to explore the metric values and the Haskell programs they are taken from. These studies have highlighted the following points.

- The initial experiments, which implemented visualisation systems within a web browser, showed that web browsers are not currently suitable for implementing user interfaces of any non-trivial visualisation system because it can be difficult to program non-trivial interactive behaviour using a standards compliant mechanism. This was a disappointment because a web browser offers an excellent cross-platform interface, and presents a familiar browsing metaphor to the user.
- Some of the visualisations, in particular the callgraph and module browser system, needed significant effort to provide the user with a suitably responsive user interface. Although such optimisations do not directly affect the visualisation technique, the responsiveness of the user interface forms a large part of the overall usability and utility of such systems.
- Visualisation tools, combined with software metrics, have applications in other parts of the software engineering process. Tools such as Tarantula [29] have shown the use of visualisation tools for analysing measurements about the testing process. Work in this thesis has shown that visualisation systems such as the file browser tool, presented in Section 7.1.2 of Chapter 7, have uses in the refactoring process, for example, indicating all the sections of a program's source code that may be affected by a particular refactoring.

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

8.2 Future Work

Although this thesis has presented an examination of a selection of metrics for Haskell programs, and has shown how data from such metrics might be presented to the user using visualisation tools, there remains significant future work. This work falls loosely into two categories.

- Further validation of the metrics. This thesis has attempted to provide some validation of metrics by analysing the correlation between metric values and the occurrence of bug fixes and refactorings in the change history of a program. However, as has been discussed in Section 3.3.4 of Chapter 3, the process of analysing the change history of a program to find the bug fixing changes is extremely time consuming. Future work could dedicate greater effort to this task in order to produce a more comprehensive selection of case study programs.

An alternative validation method that is somewhat less rigorous but may be of equal interest, is to gather anecdotal evidence from large real world projects. The goal of such work would be to investigate any correlation between functions in the programs that anecdotally have caused development or maintenance problems, and the metric values taken from the program.

- Using metrics as an integral part of the refactoring process. Fowler [36] introduced the notion of *bad smells*, which are features of source code that

strongly suggest a particular refactoring should be applied. It is possible to build catalogues of refactorings and the bad smells that indicate when they should be applied. It therefore seems likely that software measurement techniques could be used to both discover sites in source code that would benefit from refactoring, and to indicate which refactorings should be applied.

Combining metrics with refactoring catalogues in this way opens up the possibility of using evolutionary computation techniques in combination with software measurement and refactoring to evolve the source code of programs to make them cleaner, more maintainable or more extensible.

As well as these specific areas, there are also more general areas where the work in this thesis could be extended. For instance, there are language features that have not been explicitly investigated in this work, such as monads and types, which may require extra metrics to adequately assign numerical measures to programs using them.

A particularly interesting area may be measures of abstraction, which might be applied to higher order or polymorphic functions. Such metrics might aid the refactoring or redesigning of large programs by indicating both parts of a program that can be reused easily, as well as indicating those parts of the program which are hard to reuse and which may therefore benefit from attention.

It would also be interesting to examine the use of dynamic software metrics for Haskell, which may produce different results than the static metrics presented in this thesis. Equally, it would be interesting to investigate whether dynamic metrics could be combined with static metrics, or with other dynamic runtime information such as the trace information provided by tools such as Hat.

The visualisation systems also offer room for improvement. One obvious improvement is to add support for dynamic queries, which would offer greater support for exploring metric values by allowing uninteresting values to be hidden by the user. This could be further extended with greater integration with the metrics. For instance by allowing the results of various metrics to be “overlaid”

within a single visualisation and using dynamic queries to adjust the visibility of each metric. Such a system may make it easier to highlight functions with unusual characteristics. Closer integration of the metrics and visualisations systems might also encourage the creation of tools to further aid the exploration of software systems by automatically selecting the most interesting metrics for a given source file.

Extending beyond metrics, there is scope for improving the tool support for various parts of the software development process. One way this might be achieved may be to gather together existing tools such as QuickCheck, HaRe, and the metrics and visualisation tools presented in this thesis, and integrate them into a sophisticated Integrated Development Environment (IDE) such as Eclipse [19] or Visual Studio, for which Haskell support is currently being added by Frenzel [38] and Marlow [65] respectively.

It is important not to underestimate the importance that software developers place on the availability of comprehensive development tools, and thus the importance of metrics and visualisation, in the wider adoption of functional programming technology.

Appendix A

Key of Metric Variable Names

- c1* Strongly connected component size
- c2* Indegree
- c3* Outdegree
- c4* Depth
- c5* Width
- c6* Arc-to-node ratio
- d1* Distance by the sum of the number of scopes
- d2* Distance by the maximum number of scopes
- d3* Distance by the average number of scopes
- d4* Distance by the sum of the number of declarations in scope
- d5* Distance by the maximum of the number of declarations in scope
- d6* Distance by the average number of declarations in scope
- d7* Distance by the sum of the number of source lines
- d8* Distance by the maximum number of source lines
- d9* Distance by the average number of source lines
- d10* Distance by the sum of the number of parse tree nodes
- d11* Distance by the maximum number of parse tree nodes
- d12* Distance by the average number of parse tree nodes

<i>m1</i>	Pathcount
<i>m2</i>	Number of operands
<i>m3</i>	Number of operators
<i>p1</i>	Number of pattern variables
<i>p2</i>	Sum of depth of patterns
<i>p3</i>	Maximum depth of patterns
<i>p4</i>	Number of overridden or overriding pattern variables
<i>p5</i>	Number of constructors in pattern
<i>p6</i>	Pattern size
<i>p7</i>	Number of wildcards in pattern
<i>r1</i>	Binary recursion
<i>r2</i>	Number of non-trivial recursive paths
<i>r3</i>	Number of trivial recursive paths
<i>r4</i>	Number of recursive paths
<i>r5</i>	Sum of lengths of recursive paths
<i>r6</i>	Product of lengths of recursive paths

Appendix B

Tables of Correlation Between Metric Values and the Number of Changes

Peg Solitaire		
Measurement	Correlation r	Significance P
Number of pattern variables ($p1$)	0.0209	0.751
Sum of depth of patterns ($p2$)	0.071	0.2805
Maximum depth of patterns ($p3$)	0.0582	0.3765
Number of overridden or overriding pattern variables ($p4$)	-0.0278	0.6729
Number of constructors in pattern ($p5$)	-0.0649	0.324
Pattern size ($p6$)	0.0682	0.2999
Number of wildcards in pattern ($p7$)	0.0324	0.6227
Refactoring		
Measurement	Correlation r	Significance P
Number of pattern variables ($p1$)	0.5927	$P < 0.0001$
Sum of depth of patterns ($p2$)	0.5692	$P < 0.0001$
Maximum depth of patterns ($p3$)	0.4208	$P < 0.0001$
Number of overridden or overriding pattern variables ($p4$)	0.3731	$P < 0.0001$
Number of constructors in pattern ($p5$)	0.3645	$P < 0.0001$
Pattern size ($p6$)	0.5423	$P < 0.0001$
Number of wildcards in pattern ($p7$)	0.3572	$P < 0.0001$

Note. The significance is tested by means of a two-tailed Student t-test.

Table 9: Measurements of pattern attributes and their correlations with change history.

Peg Solitaire Measurements	Correlation r	Significance P
Distance by the sum of the number of scopes ($d1$)	-0.0636	0.3338
Distance by the maximum number of scopes ($d2$)	0.0742	0.2593
Distance by the average number of scopes ($d3$)	0.1121	0.0878
Distance by the sum of the number of declarations in scope ($d4$)	-0.0554	0.3999
Distance by the maximum of the number of declarations in scope ($d5$)	0.0856	0.1929
Distance by the average number of declarations in scope ($d6$)	0.12	0.0675
Distance by the sum of the number of source lines ($d7$)	-0.0963	0.1428
Distance by the maximum number of source lines ($d8$)	-0.0669	0.3092
Distance by the average number of source lines ($d9$)	-0.0483	0.4631
Distance by the sum of the number of parse tree nodes ($d10$)	-0.088	0.1807
Distance by the maximum number of parse tree nodes ($d11$)	-0.0372	0.5721
Distance by the average number of parse tree nodes ($d12$)	-0.0179	0.7858
Refactoring Measurements	Correlation r	Significance P
Distance by the sum of the number of scopes ($d1$)	0.632	$P < 0.0001$
Distance by the maximum number of scopes ($d2$)	0.6006	$P < 0.0001$
Distance by the average number of scopes ($d3$)	0.4652	$P < 0.0001$
Distance by the sum of the number of declarations in scope ($d4$)	0.546	$P < 0.0001$
Distance by the maximum of the number of declarations in scope ($d5$)	0.4643	$P < 0.0001$
Distance by the average number of declarations in scope ($d6$)	0.3948	$P < 0.0001$
Distance by the sum of the number of source lines ($d7$)	0.4347	$P < 0.0001$
Distance by the maximum number of source lines ($d8$)	0.5334	$P < 0.0001$
Distance by the average number of source lines ($d9$)	0.3608	$P < 0.0001$
Distance by the sum of the number of parse tree nodes ($d10$)	0.4484	$P < 0.0001$
Distance by the maximum number of parse tree nodes ($d11$)	0.54	$P < 0.0001$
Distance by the average number of parse tree nodes ($d12$)	0.3737	$P < 0.0001$

Note. The significance is tested by means of a two-tailed Student t-test.

Table 10: Distance measures and their correlations with change history

Peg Solitaire		
Measurement	Correlation r	Significance P
Binary recursion ($r1$)	0.1119	0.0883
Number of non-trivial recursive paths ($r2$)	-0.0753	0.2523
Number of trivial recursive paths ($r3$)	0.1077	0.1010
Number of recursive paths ($r4$)	0.0428	0.5156
Sum of lengths of recursive paths ($r5$)	0.0145	0.8258
Product of lengths of recursive paths ($r6$)	-0.0349	0.5961
Refactoring		
Measurement	Correlation r	Significance P
Binary recursion ($r1$)	-0.0154	0.7213
Number of non-trivial recursive paths ($r2$)	N/A	N/A
Number of trivial recursive paths ($r3$)	-0.0154	0.7213
Number of recursive paths ($r4$)	-0.0154	0.7213
Sum of lengths of recursive paths ($r5$)	-0.0154	0.7213
Product of lengths of recursive paths ($r6$)	-0.0154	0.7213

Note. The significance is tested by means of a two-tailed Student t-test.

Table 11: Measures of recursion and their correlations with change history.

Peg Solitaire		
Measurement	Correlation r	Significance P
Strongly connected component size ($c1$)	0.3446	$P < 0.0001$
Indegree ($c2$)	-0.0905	0.1686
Outdegree ($c3$)	0.4783	$P < 0.0001$
Depth ($c4$)	-0.064	0.3307
Width ($c5$)	-0.0942	0.1518
Arc-to-node ratio ($c6$)	0.0636	0.3338
Refactoring		
Measurement	Correlation r	Significance P
Strongly connected component size ($c1$)	0.0699	0.105
Indegree ($c2$)	0.0842	0.0507
Outdegree ($c3$)	0.5723	$P < 0.0001$
Depth ($c4$)	0.4932	$P < 0.0001$
Width ($c5$)	0.3285	$P < 0.0001$
Arc-to-node ratio ($c6$)	0.4258	$P < 0.0001$

Note. The significance is tested by means of a two-tailed Student t-test.

Table 12: Callgraph measurements and their correlations with the number of changes.

Peg Solitaire		
Measurement	Correlation r	Significance P
Pathcount ($m1$)	0.0883	0.1792
Number of operands ($m2$)	0.1099	0.0942
Number of operators ($m3$)	0.1281	0.0508
Refactoring		
Measurement	Correlation r	Significance P
Pathcount ($m1$)	0.286	$P < 0.0001$
Number of operands ($m2$)	0.5795	$P < 0.0001$
Number of operators ($m3$)	0.558	$P < 0.0001$

Note. The significance is tested by means of a two-tailed Student t-test.

Table 13: Measures of miscellaneous attributes of functions and their correlations with change history.

Appendix C

Tables of Cross-correlation Between Metric Values for the Peg Solitaire and Refactoring Programs

Peg Solitaire							
	$p1$	$p2$	$p3$	$p4$	$p5$	$p6$	$p7$
$p1$	1	0.9008	0.6016	0.2844	0.2466	0.9231	0.2128
$p2$	0.9008	1	0.7231	0.2269	0.4993	0.9899	0.3804
$p3$	0.6016	0.7231	1	0.1146	0.5855	0.7305	0.2894
$p4$	0.2844	0.2269	0.1146	1	0.0143	0.2371	0.1054
$p5$	0.2466	0.4993	0.5855	0.0143	1	0.4651	0.1958
$p6$	0.9231	0.9899	0.7305	0.2371	0.4651	1	0.3466
$p7$	0.2128	0.3804	0.2894	0.1054	0.1958	0.3466	1

Refactoring							
	$p1$	$p2$	$p3$	$p4$	$p5$	$p6$	$p7$
$p1$	1	0.9591	0.5956	0.5754	0.6447	0.9447	0.5363
$p2$	0.9591	1	0.639	0.5118	0.7719	0.9922	0.7214
$p3$	0.5956	0.639	1	0.2927	0.7209	0.6598	0.4577
$p4$	0.5754	0.5118	0.2928	1	0.2302	0.486	0.2109
$p5$	0.6447	0.7719	0.7209	0.2302	1	0.7993	0.703
$p6$	0.9447	0.9922	0.6598	0.486	0.7993	1	0.7377
$p7$	0.5363	0.7214	0.4577	0.2109	0.703	0.7377	1

Table 14: Correlation matrix for pattern attributes.
See Appendix A for key.

Peg Solitaire												
	$d1$	$d2$	$d3$	$d4$	$d5$	$d6$	$d7$	$d8$	$d9$	$d10$	$d11$	$d12$
$d1$	1	0.5582	0.4083	0.8534	0.4873	0.4286	0.6131	0.448	0.2466	0.7211	0.4743	0.2838
$d2$	0.5582	1	0.8463	0.3895	0.7571	0.6114	0.4926	0.5968	0.4412	0.4684	0.5993	0.4361
$d3$	0.4083	0.8463	1	0.357	0.6639	0.7319	0.3429	0.4488	0.4615	0.3213	0.4559	0.4431
$d4$	0.8534	0.3895	0.357	1	0.5431	0.5936	0.5199	0.3695	0.2148	0.6525	0.4218	0.2716
$d5$	0.4873	0.7571	0.6639	0.5431	1	0.84	0.4586	0.5879	0.4537	0.4966	0.6462	0.5057
$d6$	0.4286	0.6114	0.7319	0.5936	0.84	1	0.3834	0.467	0.4976	0.4214	0.5201	0.5425
$d7$	0.6131	0.4926	0.3429	0.5199	0.4586	0.3834	1	0.753	0.5879	0.9572	0.7478	0.5946
$d8$	0.448	0.5968	0.4488	0.3695	0.5879	0.467	0.753	1	0.7264	0.684	0.9725	0.7055
$d9$	0.2466	0.4412	0.4615	0.2148	0.4537	0.4976	0.5879	0.7264	1	0.5469	0.741	0.9732
$d10$	0.7211	0.4684	0.3213	0.6525	0.4966	0.4214	0.9572	0.684	0.5469	1	0.7213	0.5894
$d11$	0.4743	0.5993	0.4559	0.4218	0.6462	0.5201	0.7478	0.9725	0.741	0.7213	1	0.7638
$d12$	0.2838	0.4361	0.4431	0.2716	0.5057	0.5425	0.5946	0.7055	0.9732	0.5894	0.7638	1

Refactoring												
	$d1$	$d2$	$d3$	$d4$	$d5$	$d6$	$d7$	$d8$	$d9$	$d10$	$d11$	$d12$
$d1$	1	0.6338	0.4033	0.883	0.2942	0.2148	0.6714	0.5456	0.3711	0.6886	0.5416	0.3813
$d2$	0.6338	1	0.8511	0.6365	0.7176	0.5764	0.4628	0.6004	0.4212	0.4861	0.6049	0.4468
$d3$	0.4033	0.8511	1	0.4405	0.6906	0.7254	0.2978	0.4438	0.3465	0.3107	0.4435	0.3678
$d4$	0.883	0.6365	0.4405	1	0.4559	0.405	0.7413	0.5478	0.4156	0.7523	0.5422	0.4207
$d5$	0.2942	0.7176	0.6906	0.4559	1	0.8678	0.3276	0.4317	0.3979	0.342	0.4366	0.4126
$d6$	0.2148	0.5764	0.7254	0.405	0.8678	1	0.2578	0.3178	0.3286	0.2605	0.3167	0.3342
$d7$	0.6714	0.4628	0.2978	0.7413	0.3276	0.2578	1	0.6921	0.6028	0.9958	0.6855	0.5827
$d8$	0.5456	0.6004	0.4438	0.5478	0.4317	0.3178	0.6921	1	0.753	0.7021	0.9979	0.7406
$d9$	0.3711	0.4212	0.3465	0.4156	0.3979	0.3286	0.6028	0.753	1	0.6142	0.7516	0.9891
$d10$	0.6886	0.4861	0.3107	0.7523	0.342	0.2605	0.9958	0.7021	0.6142	1	0.699	0.6006
$d11$	0.5416	0.6049	0.4435	0.5422	0.4366	0.3167	0.6855	0.9979	0.7516	0.699	1	0.7452
$d12$	0.3813	0.4468	0.3678	0.4207	0.4126	0.3342	0.5827	0.7406	0.9891	0.6006	0.7452	1

Table 15: Correlation matrix for distance attributes.

See Appendix A for key.

Peg Solitaire						
	$r1$	$r2$	$r3$	$r4$	$r5$	$r6$
$r1$	1	0.3863	0.9628	0.8183	0.7443	0.229
$r2$	0.3863	1	0.3333	0.7747	0.8666	0.8488
$r3$	0.9628	0.3333	1	0.8128	0.7265	0.2217
$r4$	0.8183	0.7747	0.8128	1	0.9869	0.6975
$r5$	0.7443	0.8666	0.7265	0.9869	1	0.7671
$r6$	0.229	0.8488	0.2217	0.6975	0.7671	1

Refactoring						
	$r1$	$r2$	$r3$	$r4$	$r5$	$r6$
$r1$	1		1	1	1	1
$r2$						
$r3$	1		1	1	1	1
$r4$	1		1	1	1	1
$r5$	1		1	1	1	1
$r6$	1		1	1	1	1

Table 16: Correlation matrix for recursion measurements.
See Appendix A for key.

Peg Solitaire						
	<i>c1</i>	<i>c2</i>	<i>c3</i>	<i>c4</i>	<i>c5</i>	<i>c6</i>
<i>c1</i>	1	-0.0275	0.3437	0.0201	-0.0235	0.1643
<i>c2</i>	-0.0275	1	-0.2512	0.0938	0.0345	-0.0196
<i>c3</i>	0.3437	-0.2512	1	-0.0745	-0.0833	0.0746
<i>c4</i>	0.0201	0.0938	-0.0745	1	0.9273	0.4799
<i>c5</i>	-0.0235	0.0345	-0.0833	0.9273	1	0.3973
<i>c6</i>	0.1643	-0.0196	0.0746	0.4799	0.3973	1

Refactoring						
	<i>c1</i>	<i>c2</i>	<i>c3</i>	<i>c4</i>	<i>c5</i>	<i>c6</i>
<i>c1</i>	1	0.3479	0.018	-0.0005	0.0047	-0.0049
<i>c2</i>	0.3479	1	0.0583	0.0185	-0.0224	0.1142
<i>c3</i>	0.018	0.0583	1	0.6331	0.4039	0.6303
<i>c4</i>	-0.0005	0.0185	0.6331	1	0.7728	0.5934
<i>c5</i>	0.0047	-0.0224	0.4039	0.7728	1	0.3191
<i>c6</i>	-0.0049	0.1142	0.6303	0.5934	0.3191	1

Table 17: Correlation matrix for callgraph measurements.
See Appendix A for key.

Peg Solitaire			
	<i>m1</i>	<i>m2</i>	<i>m3</i>
<i>m1</i>	1	-0.0896	-0.0929
<i>m2</i>	-0.0896	1	0.8963
<i>m3</i>	-0.0929	0.8963	1

Refactoring			
	<i>m1</i>	<i>m2</i>	<i>m3</i>
<i>m1</i>	1	0.3098	0.2917
<i>m2</i>	0.3098	1	0.9747
<i>m3</i>	0.2917	0.9747	1

Table 18: Correlation matrix for function measurements.
See Appendix A for key.

	<i>p2</i>	<i>p3</i>	<i>p4</i>	<i>p5</i>	<i>p7</i>	<i>d6</i>	<i>d7</i>	
<i>p2</i>	1	0.7231	0.2269	0.4993	0.3804	0.2421	0.4756	
<i>p3</i>	0.7231	1	0.1146	0.5855	0.2894	0.3296	0.2622	
<i>p4</i>	0.2269	0.1146	1	0.0143	0.1054	0.1104	0.1701	
<i>p5</i>	0.4993	0.5855	0.0143	1	0.1958	-0.0597	0.1394	
<i>p7</i>	0.3804	0.2894	0.1054	0.1958	1	0.106	0.1992	
<i>d6</i>	0.2421	0.3296	0.1104	-0.0597	0.106	1	0.3834	
<i>d7</i>	0.4756	0.2622	0.1701	0.1394	0.1992	0.3834	1	
<i>r1</i>	0.4432	0.3082	0.1636	0.059	0.0731	0.0483	0.4107	
<i>c1</i>	0.1295	0.1471	0.0804	0.0463	0.0537	0.0896	0.0064	
<i>c2</i>	0.0988	0.0553	-0.0202	0.0317	-0.035	0.0514	0.0255	
<i>c3</i>	-0.0046	-0.0079	0.0591	-0.0673	0.1254	0.1497	-0.0685	
<i>c4</i>	0.4179	0.2672	0.2089	0.0691	0.1471	0.2942	0.607	
<i>c6</i>	0.3443	0.4118	0.1079	0.0568	0.0905	0.6138	0.3708	
<i>m1</i>	0.6818	0.5632	0.0551	0.4859	0.3576	0.1101	0.4634	
<i>m2</i>	-0.123	-0.1188	-0.0397	-0.0794	-0.0191	0.1831	0.1026	
	<i>r1</i>	<i>c1</i>	<i>c2</i>	<i>c3</i>	<i>c4</i>	<i>c6</i>	<i>m1</i>	<i>m2</i>
<i>p2</i>	0.4432	0.1295	0.0988	-0.0046	0.4179	0.3443	0.6818	-0.123
<i>p3</i>	0.3082	0.1471	0.0553	-0.0079	0.2672	0.4118	0.5632	-0.1188
<i>p4</i>	0.1636	0.0804	-0.0202	0.0591	0.2089	0.1079	0.0551	-0.0397
<i>p5</i>	0.059	0.0463	0.0317	-0.0673	0.0691	0.0568	0.4859	-0.0794
<i>p7</i>	0.0731	0.0537	-0.035	0.1254	0.1471	0.0905	0.3576	-0.0191
<i>d6</i>	0.0483	0.0896	0.0514	0.1497	0.2942	0.6138	0.1101	0.1831
<i>d7</i>	0.4107	0.0064	0.0255	-0.0685	0.6070	0.3708	0.4634	0.1026
<i>r1</i>	1	0.057	-0.0407	0.0126	0.5390	0.2194	0.4138	-0.0314
<i>c1</i>	0.057	1	-0.0275	0.3437	0.0201	0.1643	0.1227	0.0092
<i>c2</i>	-0.0407	-0.0275	1	-0.2512	0.0938	-0.0196	0.0051	-0.0127
<i>c3</i>	0.0126	0.3437	-0.2512	1	-0.0745	0.0746	-0.0177	0.0348
<i>c4</i>	0.539	0.0201	0.0938	-0.0745	1	0.4799	0.3084	0.0389
<i>c6</i>	0.2194	0.1643	-0.0196	0.0746	0.4799	1	0.3213	0.039
<i>m1</i>	0.4138	0.1227	0.0051	-0.0177	0.3084	0.3213	1	-0.0896
<i>m2</i>	-0.0314	0.0092	-0.0127	0.0348	0.0389	0.039	-0.0896	1

Table 19: Correlation matrix for all measurements from the Peg Solitaire program.
See Appendix A for key.

	<i>p1</i>	<i>p3</i>	<i>p4</i>	<i>p7</i>	<i>d1</i>	<i>d2</i>	<i>r1</i>
<i>p1</i>	1	0.5956	0.5754	0.5363	0.8979	0.6562	-0.0163
<i>p3</i>	0.5956	1	0.2927	0.4577	0.3912	0.5640	0.0449
<i>p4</i>	0.5754	0.2927	1	0.2109	0.5119	0.4068	-0.0056
<i>p7</i>	0.5363	0.4577	0.2109	1	0.4441	0.3189	-0.0477
<i>d1</i>	0.8979	0.3912	0.5119	0.4441	1	0.6338	-0.0153
<i>d2</i>	0.6562	0.5640	0.4068	0.3189	0.6338	1	0.0568
<i>r1</i>	-0.0163	0.0449	-0.0056	-0.0477	-0.0153	0.0568	1
<i>c1</i>	0.2044	0.0820	-0.0123	0.3504	0.0978	0.0578	-0.0102
<i>c2</i>	0.1044	0.1583	0.0741	0.1323	0.0376	0.0828	0.0765
<i>c3</i>	0.5880	0.3779	0.3408	0.3105	0.6025	0.7541	0.1489
<i>c4</i>	0.5394	0.3821	0.3039	0.2187	0.5466	0.6932	0.0777
<i>c6</i>	0.3580	0.3736	0.1884	0.1756	0.3701	0.6387	0.0750
<i>m1</i>	0.3116	0.2777	0.2526	0.2126	0.2884	0.3247	0.3477
<i>m2</i>	0.9333	0.4776	0.5368	0.5116	0.9175	0.6095	-0.0061
	<i>c1</i>	<i>c2</i>	<i>c3</i>	<i>c4</i>	<i>c6</i>	<i>m1</i>	<i>m2</i>
<i>p1</i>	0.2044	0.1044	0.5880	0.5394	0.3580	0.3116	0.9333
<i>p3</i>	0.0820	0.1583	0.3779	0.3821	0.3736	0.2777	0.4776
<i>p4</i>	-0.0123	0.0741	0.3408	0.3039	0.1884	0.2526	0.5368
<i>p7</i>	0.3504	0.1323	0.3105	0.2187	0.1756	0.2126	0.5116
<i>d1</i>	0.0978	0.0376	0.6025	0.5466	0.3701	0.2884	0.9175
<i>d2</i>	0.0578	0.0828	0.7541	0.6932	0.6387	0.3247	0.6095
<i>r1</i>	-0.0102	0.0765	0.1489	0.0777	0.0750	0.3477	-0.0061
<i>c1</i>	1	0.3479	0.0180	-0.0005	-0.0049	-0.0168	0.1619
<i>c2</i>	0.3479	1	0.0583	0.0185	0.1142	0.0277	0.0464
<i>c3</i>	0.0180	0.0583	1	0.6331	0.6303	0.5245	0.5667
<i>c4</i>	-0.0005	0.0185	0.6331	1	0.5934	0.3158	0.5158
<i>c6</i>	-0.0049	0.1142	0.6303	0.5934	1	0.3226	0.3320
<i>m1</i>	-0.0168	0.0277	0.5245	0.3158	0.3226	1	0.3098
<i>m2</i>	0.1619	0.0464	0.5667	0.5158	0.3320	0.3098	1

Table 20: Correlation matrix for all measurements from the Refactoring program.
See Appendix A for key.

Appendix D

Tables of Regression Analysis Between Metric Values and the Number of Changes

Peg Solitaire	
R	0.1584
R^2	0.025
Significance	0.3255
Measurement	Coefficient
Sum of depth of patterns ($p2$)	0.1263
Maximum depth of patterns ($p3$)	0.107
Number of overridden or overriding pattern variables ($p4$)	-0.0731
Number of constructors in pattern ($p5$)	-0.2063
Number of wildcards in pattern ($p7$)	0.009
Refactoring	
R	0.6015
R^2	0.3618
Significance	< 0.0001
Measurement	Coefficient
Number of pattern variables ($p1$)	0.4035
Maximum depth of patterns ($p3$)	0.0856
Number of overridden or overriding pattern variables ($p4$)	0.0527
Number of wildcards in pattern ($p7$)	0.0378

Table 21: Regression analysis of measurements of pattern attributes.

Peg Solitaire	
R	0.1957
R^2	0.0383
Significance	0.0112
Measurement	Coefficient
Distance by the average number of declarations in scope ($d6$)	0.2273
Distance by the sum of the number of source lines ($d7$)	-0.2058

Refactoring	
R	0.6829
R^2	0.4663
Significance	< 0.0001
Measurement	Coefficient
Distance by the sum of the number of scopes ($d1$)	0.3581
Distance by the maximum number of scopes ($d2$)	0.2849

Table 22: Regression analysis of distance metric measurements.

R	0.1978
R^2	0.0391
Significance	0.1683
Measurement	Coefficient
Binary recursion ($r1$)	0.4136
Number of non-trivial recursive paths ($r2$)	-0.6805
Number of trivial recursive paths ($r3$)	-0.2852
Number of recursive paths ($r4$)	-0.6058
Sum of lengths of recursive paths ($r5$)	0.9592
Product of lengths of recursive paths ($r6$)	0.1898

Table 23: Regression analysis of recursion measurements from the Peg Solitaire program.

Peg Solitaire	
R	0.5176
R^2	0.268
Significance	< 0.0001
Measurement	Coefficient
Strongly connected component size ($c1$)	0.2483
Indegree ($c2$)	0.0285
Outdegree ($c3$)	0.5039
Depth ($c4$)	-0.0639
Arc-to-node ratio ($c6$)	0.0312

Refactoring	
R	0.6017
R^2	0.362
Significance	< 0.0001
Measurement	Coefficient
Strongly connected component size ($c1$)	0.0434
Indegree ($c2$)	0.0292
Outdegree ($c3$)	0.3526
Depth ($c4$)	0.1774
Arc-to-node ratio ($c6$)	0.0323

Table 24: Regression analysis of callgraph measurements.

Peg Solitaire	
R	0.1476
R^2	0.0218
Significance	0.0794
Measurement	Coefficient
Pathcount ($m1$)	0.1219
Number of operands ($m2$)	0.1463

Refactoring	
R	0.5902
R^2	0.3483
Significance	< 0.0001
Measurement	Coefficient
Pathcount ($m1$)	0.1004
Number of operands ($m2$)	0.4628

Table 25: Regression analysis of function measurements.

R	0.583
R^2	0.3399
Significance	< 0.0001
Measurement	Coefficient
Sum of depth of patterns ($p2$)	0.1516
Maximum depth of patterns ($p3$)	-0.0341
Number of overridden or overriding pattern variables ($p4$)	-0.083
Number of constructors in pattern ($p5$)	-0.1392
Number of wildcards in pattern ($p7$)	-0.0692
Distance by the average number of declarations in scope ($d6$)	0.1532
Distance by the sum of the number of source lines ($d7$)	-0.2673
Binary recursion ($r1$)	0.1691
Strongly connected component size ($c1$)	0.2314
Indegree ($c2$)	0.0127
Outdegree ($c3$)	0.4731
Depth ($c4$)	-0.0699
Arc-to-node ratio ($c6$)	-0.0774
Pathcount ($m1$)	0.1979
Number of operands ($m2$)s	0.1440

Table 26: Regression analysis of all measurements from the Peg Solitaire program.

R	0.6973
R^2	0.4863
Significance	< 0.0001
Measurement	Coefficient
Maximum depth of patterns ($p3$)	0.0558
Number of overridden or overriding pattern variables ($p4$)	0.021
Number of wildcards in pattern ($p7$)	0.0471
Distance by the sum of the number of scopes ($d1$)	0.315
Distance by the maximum number of scopes ($d2$)	0.1753
Binary recursion ($r1$)	-0.0447
Strongly connected component size ($c1$)	-0.0108
Indegree ($c2$)	0.0268
Depth ($c4$)	0.0393
Arc-to-node ratio ($c6$)	0.063
“Pathcount” ($m1$)	0.047

Table 27: Regression analysis of all measurements from the Refactoring program.

Appendix E

Tables of Cross-correlation

E.1 Tables of Cross-correlation of Recursion Metrics

	r1	r2	r3	r4	r5	r6
c1	0.766		0.766	0.766	0.766	0.766
c2	0.0696		0.0696	0.0696	0.0696	0.0696
c3	0.4967		0.4967	0.4967	0.4967	0.4967
c4	0.1437		0.1437	0.1437	0.1437	0.1437
c5	0.1388		0.1388	0.1388	0.1388	0.1388
c6	0.1442		0.1442	0.1442	0.1442	0.1442
d1	0.2918		0.2918	0.2918	0.2918	0.2918
d2	0.2135		0.2135	0.2135	0.2135	0.2135
d3	-0.0006		-0.0006	-0.0006	-0.0006	-0.0006
d4	0.2118		0.2118	0.2118	0.2118	0.2118
d5	0.1174		0.1174	0.1174	0.1174	0.1174
d6	0.0227		0.0227	0.0227	0.0227	0.0227
d7	0.0487		0.0487	0.0487	0.0487	0.0487
d8	0.0034		0.0034	0.0034	0.0034	0.0034
d9	-0.0853		-0.0853	-0.0853	-0.0853	-0.0853
d10	0.1831		0.1831	0.1831	0.1831	0.1831
d11	0.0796		0.0796	0.0796	0.0796	0.0796
d12	-0.0751		-0.0751	-0.0751	-0.0751	-0.0751
m1	0.5266		0.5266	0.5266	0.5266	0.5266
m2	0.3995		0.3995	0.3995	0.3995	0.3995
m3	0.3145		0.3145	0.3145	0.3145	0.3145
p1	0.4115		0.4115	0.4115	0.4115	0.4115
p2	0.4341		0.4341	0.4341	0.4341	0.4341
p3	0.3882		0.3882	0.3882	0.3882	0.3882
p4	0.1644		0.1644	0.1644	0.1644	0.1644
p5	0.5065		0.5065	0.5065	0.5065	0.5065
p6	0.4553		0.4553	0.4553	0.4553	0.4553
p7	0.0686		0.0686	0.0686	0.0686	0.0686
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 28: Correlation between recursion metrics and others for the CGI Library.

	r1	r2	r3	r4	r5	r6
c1	-0.0244		-0.0244	-0.0244	-0.0244	-0.0244
c2	0.0222		0.0222	0.0222	0.0222	0.0222
c3	0.372		0.372	0.372	0.372	0.372
c4	0.1819		0.1819	0.1819	0.1819	0.1819
c5	0.1529		0.1529	0.1529	0.1529	0.1529
c6	0.0775		0.0775	0.0775	0.0775	0.0775
d1	0.2569		0.2569	0.2569	0.2569	0.2569
d2	0.1058		0.1058	0.1058	0.1058	0.1058
d3	0.0457		0.0457	0.0457	0.0457	0.0457
d4	0.2394		0.2394	0.2394	0.2394	0.2394
d5	0.1536		0.1536	0.1536	0.1536	0.1536
d6	0.1282		0.1282	0.1282	0.1282	0.1282
d7	0.0994		0.0994	0.0994	0.0994	0.0994
d8	0.1881		0.1881	0.1881	0.1881	0.1881
d9	0.0225		0.0225	0.0225	0.0225	0.0225
d10	0.2986		0.2986	0.2986	0.2986	0.2986
d11	0.2603		0.2603	0.2603	0.2603	0.2603
d12	0.0899		0.0899	0.0899	0.0899	0.0899
m1	0.2948		0.2948	0.2948	0.2948	0.2948
m2	0.0417		0.0417	0.0417	0.0417	0.0417
m3	0.3214		0.3214	0.3214	0.3214	0.3214
p1	0.2678		0.2678	0.2678	0.2678	0.2678
p2	0.2841		0.2841	0.2841	0.2841	0.2841
p3	0.327		0.327	0.327	0.327	0.327
p4	-0.0412		-0.0412	-0.0412	-0.0412	-0.0412
p5	0.3048		0.3048	0.3048	0.3048	0.3048
p6	0.3257		0.3257	0.3257	0.3257	0.3257
p7	0.1816		0.1816	0.1816	0.1816	0.1816
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 29: Correlation between recursion metrics and others for the Haskell Cryptographic Library.

	r1	r2	r3	r4	r5	r6
c1	-0.0346		-0.0346	-0.0346	-0.0346	-0.0346
c2	0.2414		0.2414	0.2414	0.2414	0.2414
c3	0.3073		0.3073	0.3073	0.3073	0.3073
c4	-0.0985		-0.0985	-0.0985	-0.0985	-0.0985
c5	-0.0944		-0.0944	-0.0944	-0.0944	-0.0944
c6	0.0755		0.0755	0.0755	0.0755	0.0755
d1	0.0124		0.0124	0.0124	0.0124	0.0124
d2	0.1227		0.1227	0.1227	0.1227	0.1227
d3	0.0384		0.0384	0.0384	0.0384	0.0384
d4	0.3552		0.3552	0.3552	0.3552	0.3552
d5	0.6219		0.6219	0.6219	0.6219	0.6219
d6	0.5358		0.5358	0.5358	0.5358	0.5358
d7	0.3905		0.3905	0.3905	0.3905	0.3905
d8	0.366		0.366	0.366	0.366	0.366
d9	0.2926		0.2926	0.2926	0.2926	0.2926
d10	0.3189		0.3189	0.3189	0.3189	0.3189
d11	0.4513		0.4513	0.4513	0.4513	0.4513
d12	0.3196		0.3196	0.3196	0.3196	0.3196
m1	0.6838		0.6838	0.6838	0.6838	0.6838
m2	0.1178		0.1178	0.1178	0.1178	0.1178
m3	0.0973		0.0973	0.0973	0.0973	0.0973
p1	-0.0148		-0.0148	-0.0148	-0.0148	-0.0148
p2	0.0526		0.0526	0.0526	0.0526	0.0526
p3	0.171		0.171	0.171	0.171	0.171
p4	-0.0488		-0.0488	-0.0488	-0.0488	-0.0488
p5	-0.0282		-0.0282	-0.0282	-0.0282	-0.0282
p6	0.0496		0.0496	0.0496	0.0496	0.0496
p7	0.1192		0.1192	0.1192	0.1192	0.1192
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 30: Correlation between recursion metrics and others for the Haskell DSP Library.

	r1	r2	r3	r4	r5	r6
c1	0.1171		0.1171	0.1171	0.1171	0.1171
c2	0.0325		0.0325	0.0325	0.0325	0.0325
c3	0.5444		0.5444	0.5444	0.5444	0.5444
c4	0.1148		0.1148	0.1148	0.1148	0.1148
c5	0.1273		0.1273	0.1273	0.1273	0.1273
c6	0.0961		0.0961	0.0961	0.0961	0.0961
d1	0.4025		0.4025	0.4025	0.4025	0.4025
d2	0.269		0.269	0.269	0.269	0.269
d3	0.0007		0.0007	0.0007	0.0007	0.0007
d4	0.3366		0.3366	0.3366	0.3366	0.3366
d5	0.0984		0.0984	0.0984	0.0984	0.0984
d6	-0.0345		-0.0345	-0.0345	-0.0345	-0.0345
d7	0.1099		0.1099	0.1099	0.1099	0.1099
d8	0.0396		0.0396	0.0396	0.0396	0.0396
d9	-0.0988		-0.0988	-0.0988	-0.0988	-0.0988
d10	0.2037		0.2037	0.2037	0.2037	0.2037
d11	0.08		0.08	0.08	0.08	0.08
d12	-0.0872		-0.0872	-0.0872	-0.0872	-0.0872
m1	0.5925		0.5925	0.5925	0.5925	0.5925
m2	0.3316		0.3316	0.3316	0.3316	0.3316
m3	0.2362		0.2362	0.2362	0.2362	0.2362
p1	0.4294		0.4294	0.4294	0.4294	0.4294
p2	0.4796		0.4796	0.4796	0.4796	0.4796
p3	0.4994		0.4994	0.4994	0.4994	0.4994
p4	-0.0323		-0.0323	-0.0323	-0.0323	-0.0323
p5	0.474		0.474	0.474	0.474	0.474
p6	0.4715		0.4715	0.4715	0.4715	0.4715
p7	0.1822		0.1822	0.1822	0.1822	0.1822
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 31: Correlation between recursion metrics and others for FGL.

	r1	r2	r3	r4	r5	r6
c1	0.197	0.4984	0.1032	0.197	0.2782	0.2782
c2	0.0336	-0.0007	0.0343	0.0336	0.0319	0.0319
c3	0.3271	0.0109	0.3304	0.3271	0.3131	0.3131
c4	0.0993	0.0149	0.098	0.0993	0.0972	0.0972
c5	0.1293	0.0047	0.1305	0.1293	0.1238	0.1238
c6	0.0604	-0.0149	0.0643	0.0604	0.0547	0.0547
d1	0.1053	-0.0134	0.1097	0.1053	0.0977	0.0977
d2	0.0692	-0.0202	0.0743	0.0692	0.0622	0.0622
d3	-0.0255	-0.0203	-0.022	-0.0255	-0.028	-0.028
d4	0.0837	-0.0161	0.0882	0.0837	0.0766	0.0766
d5	0.0512	-0.0342	0.0587	0.0512	0.0424	0.0424
d6	-0.0517	-0.035	-0.0458	-0.0517	-0.0556	-0.0556
d7	0.1778	-0.0146	0.1836	0.1778	0.1664	0.1664
d8	0.222	-0.0198	0.2296	0.222	0.2076	0.2076
d9	0.1321	-0.018	0.1378	0.1321	0.1224	0.1224
d10	0.1405	-0.0136	0.1455	0.1405	0.1312	0.1312
d11	0.1543	-0.0185	0.1605	0.1543	0.1434	0.1434
d12	0.0907	-0.0179	0.0956	0.0907	0.083	0.083
m1	0.4921	0.0783	0.485	0.4921	0.4823	0.4823
m2	0.2068	0.0079	0.2087	0.2068	0.1981	0.1981
m3	0.1702	-0.0004	0.1731	0.1702	0.1618	0.1618
p1	0.3031	0.0491	0.2985	0.3031	0.2972	0.2972
p2	0.2932	0.0542	0.2875	0.2932	0.2887	0.2887
p3	0.3022	0.0446	0.2986	0.3022	0.2956	0.2956
p4	-0.0236	-0.0056	-0.0239	-0.0236	-0.0235	-0.0235
p5	0.31	-0.0203	0.3191	0.31	0.2911	0.2911
p6	0.3238	0.0406	0.3212	0.3238	0.3153	0.3153
p7	0.1948	-0.016	0.2012	0.1948	0.1824	0.1824
r1	1.0	0.18	0.9815	1.0	0.9838	0.9838
r2	0.18	1.0	-0.0117	0.18	0.3534	0.3534
r3	0.9815	-0.0117	1.0	0.9815	0.9313	0.9313
r4	1.0	0.18	0.9815	1.0	0.9838	0.9838
r5	0.9838	0.3534	0.9313	0.9838	1.0	1.0
r6	0.9838	0.3534	0.9313	0.9838	1.0	1.0

Table 32: Correlation between recursion metrics and others for the Library of Geometric Algorithms.

	r1	r2	r3	r4	r5	r6
c1	0.8689	0.1291	-0.0508	0.1286	0.1289	0.0742
c2	-0.0168	0.1407	0.0135	0.1406	0.1406	0.126
c3	-0.0153	0.5407	0.1466	0.5411	0.5408	0.0293
c4	0.8747	0.1108	-0.0065	0.1107	0.1107	0.0652
c5	0.8608	0.1275	-0.046	0.127	0.1273	0.0732
c6	0.4697	0.0601	0.0834	0.0606	0.0602	0.0347
d1	-0.0309	0.206	0.1185	0.2066	0.2062	0.0172
d2	-0.1547	0.0647	0.1598	0.0657	0.065	0.0582
d3	0.2041	0.0321	-0.0239	0.0319	0.032	-0.0172
d4	-0.168	0.2272	0.0812	0.2275	0.2273	0.0807
d5	-0.2673	0.088	-0.0312	0.0877	0.0879	0.0716
d6	-0.2671	0.0239	-0.0621	0.0235	0.0238	-0.0094
d7	0.0324	0.6352	-0.0081	0.6344	0.6349	-0.0004
d8	0.2542	0.0786	-0.075	0.078	0.0784	0.0013
d9	0.183	0.0198	-0.1054	0.0191	0.0195	-0.0231
d10	0.0305	0.6317	-0.0057	0.631	0.6314	-0.0001
d11	0.2525	0.0706	-0.0676	0.07	0.0704	0.0001
d12	0.1855	0.0143	-0.1037	0.0136	0.0141	-0.0233
m1	0.0043	0.0033	0.0408	0.0036	0.0033	0.0006
m2	0.0104	0.3925	0.1626	0.3932	0.3927	0.0336
m3	0.0012	0.5021	0.1294	0.5024	0.5022	0.0367
p1	-0.0165	0.0627	0.2424	0.0643	0.0632	0.1005
p2	0.0026	0.057	0.2077	0.0583	0.0573	0.0921
p3	0.0175	0.0797	0.1814	0.0808	0.08	0.1701
p4	0.0007	-0.0051	0.0921	-0.0045	-0.0049	-0.003
p5	0.0403	0.0017	0.1277	0.0025	0.0016	0.0139
p6	-0.0052	0.0432	0.1884	0.0444	0.0435	0.0752
p7	-0.0053	0.0147	0.0372	0.0149	0.0146	0.0348
r1	1.0	0.1212	0.2976	0.1231	0.1218	0.0711
r2	0.1212	1.0	0.1615	1.0	1.0	0.2207
r3	0.2976	0.1615	1.0	0.168	0.164	-0.0065
r4	0.1231	1.0	0.168	1.0	1.0	0.2204
r5	0.1218	1.0	0.164	1.0	1.0	0.2205
r6	0.0711	0.2207	-0.0065	0.2204	0.2205	1.0

Table 33: Correlation between recursion metrics and others for Haddock.

	r1	r2	r3	r4	r5	r6
c1	0.1591		0.1591	0.1591	0.1591	0.1591
c2	0.0016		0.0016	0.0016	0.0016	0.0016
c3	0.3785		0.3785	0.3785	0.3785	0.3785
c4	-0.0552		-0.0552	-0.0552	-0.0552	-0.0552
c5	-0.012		-0.012	-0.012	-0.012	-0.012
c6	0.1449		0.1449	0.1449	0.1449	0.1449
d1	-0.0117		-0.0117	-0.0117	-0.0117	-0.0117
d2	0.1416		0.1416	0.1416	0.1416	0.1416
d3	-0.0309		-0.0309	-0.0309	-0.0309	-0.0309
d4	-0.0168		-0.0168	-0.0168	-0.0168	-0.0168
d5	0.1364		0.1364	0.1364	0.1364	0.1364
d6	0.0144		0.0144	0.0144	0.0144	0.0144
d7	-0.0231		-0.0231	-0.0231	-0.0231	-0.0231
d8	-0.0712		-0.0712	-0.0712	-0.0712	-0.0712
d9	-0.1187		-0.1187	-0.1187	-0.1187	-0.1187
d10	-0.0215		-0.0215	-0.0215	-0.0215	-0.0215
d11	-0.0413		-0.0413	-0.0413	-0.0413	-0.0413
d12	-0.1146		-0.1146	-0.1146	-0.1146	-0.1146
m1	-0.0158		-0.0158	-0.0158	-0.0158	-0.0158
m2	0.0356		0.0356	0.0356	0.0356	0.0356
m3	0.0544		0.0544	0.0544	0.0544	0.0544
p1	0.0739		0.0739	0.0739	0.0739	0.0739
p2	0.0785		0.0785	0.0785	0.0785	0.0785
p3	0.2309		0.2309	0.2309	0.2309	0.2309
p4	-0.0317		-0.0317	-0.0317	-0.0317	-0.0317
p5	-0.0401		-0.0401	-0.0401	-0.0401	-0.0401
p6	0.075		0.075	0.075	0.075	0.075
p7	-0.0452		-0.0452	-0.0452	-0.0452	-0.0452
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 34: Correlation between recursion metrics and others for Happy.

	r1	r2	r3	r4	r5	r6
c1	0.0742	0.0431	0.0556	0.0511	0.0456	0.0267
c2	0.002	-0.0106	0.0071	-0.0095	-0.0122	-0.0033
c3	0.346	0.0976	0.3326	0.1469	0.1088	0.0503
c4	0.2266	0.2395	0.0729	0.2487	0.2443	0.1431
c5	0.1979	0.2128	0.069	0.2216	0.2173	0.1288
c6	0.1761	0.1342	0.1029	0.1487	0.1359	0.0832
d1	0.1678	0.0205	0.1846	0.0481	0.0283	0.0096
d2	0.1846	0.0599	0.1663	0.0845	0.0646	0.0331
d3	0.0422	0.0463	0.0039	0.0465	0.0469	0.0157
d4	0.1546	-0.076	0.13	0.095	0.085	0.0407
d5	0.262	0.2442	0.1145	0.2596	0.2515	0.1449
d6	0.1188	0.1001	0.0307	0.104	0.1011	0.0669
d7	0.1566	0.0708	0.1374	0.091	0.0746	0.0395
d8	0.1386	0.1384	0.0533	0.1454	0.1348	0.0742
d9	0.1226	0.2399	-0.0403	0.232	0.2316	0.1405
d10	0.1644	0.0691	0.1477	0.0909	0.0728	0.0361
d11	0.1548	0.146	0.0662	0.1549	0.1423	0.0761
d12	0.1568	0.2651	-0.0193	0.2603	0.2552	0.1492
m1	0.2293	-0.0133	0.2766	0.0284	-0.0036	-0.007
m2	0.2132	0.0055	0.2465	0.0426	0.0147	0.0007
m3	0.1965	-0.011	0.238	0.0249	-0.0018	-0.0086
p1	0.2073	-0.0094	0.2486	0.0281	-0.0012	-0.0076
p2	0.227	-0.0095	0.2724	0.0316	-0.0005	-0.0068
p3	0.2605	0.0211	0.2923	0.0649	0.0237	0.0069
p4	-0.012	-0.0121	-0.0043	-0.0126	-0.0119	-0.0074
p5	0.2248	-0.0088	0.2735	0.0324	-0.0019	-0.001
p6	0.2143	-0.0134	0.2598	0.0258	-0.0047	-0.0093
p7	0.1058	-0.0216	0.1371	-0.0008	-0.015	-0.0156
r1	1.0	0.389	0.8467	0.5136	0.4088	0.239
r2	0.389	1.0	-0.0268	0.9886	0.9864	0.6196
r3	0.8467	-0.0268	1.0	0.1239	0.0091	-0.0184
r4	0.5136	0.9886	0.1239	1.0	0.9805	0.6122
r5	0.4088	0.9864	0.0091	0.9805	1.0	0.6088
r6	0.239	0.6196	-0.0184	0.6122	0.6088	1.0

Table 35: Correlation between recursion metrics and others for Hat.

	r1	r2	r3	r4	r5	r6
c1	0.4838	0.4891	0.3451	0.5395	0.5399	0.2775
c2	0.0001	-0.0165	0.0048	-0.0045	-0.0122	-0.0104
c3	0.4144	0.0964	0.3898	0.3762	0.2428	0.0636
c4	0.275	0.2006	0.2418	0.3053	0.2746	0.1498
c5	0.1828	0.1113	0.162	0.1928	0.1583	0.0678
c6	0.2779	0.1122	0.2536	0.2701	0.1983	0.0738
d1	0.1404	0.0345	0.1299	0.1266	0.0854	0.0118
d2	0.1775	0.0876	0.1657	0.1837	0.1487	0.0562
d3	0.0637	0.0474	0.0636	0.0776	0.0729	0.0305
d4	0.1738	0.0159	0.1783	0.1576	0.0884	0.0042
d5	0.118	0.0332	0.1239	0.1209	0.0837	0.029
d6	0.0148	-0.0131	0.0252	0.0144	0.0028	-0.0089
d7	0.0568	0.0265	0.0464	0.0525	0.0455	0.0108
d8	0.0375	0.0352	0.0295	0.0428	0.0484	0.0263
d9	-0.0089	0.0298	-0.0158	0.0019	0.0269	0.0195
d10	0.1061	0.0271	0.096	0.0944	0.0659	0.0095
d11	0.0924	0.0249	0.0871	0.0857	0.0574	0.0147
d12	0.0238	0.0246	0.0201	0.0294	0.0355	0.0139
m1	0.0289	-0.0028	0.0288	0.0227	0.0091	-0.0022
m2	0.1767	0.0095	0.1724	0.1495	0.0782	0.0009
m3	0.1584	0.0219	0.1491	0.1363	0.0811	0.0076
p1	0.1987	0.0104	0.195	0.1689	0.087	-0.0001
p2	0.1677	-0.0058	0.1685	0.1384	0.0622	-0.0058
p3	0.1792	-0.0255	0.1836	0.1409	0.0508	-0.0138
p4	-0.0183	-0.0049	-0.0171	-0.0169	-0.0108	-0.0031
p5	0.1497	-0.0362	0.1582	0.1142	0.0314	-0.018
p6	0.1624	-0.0074	0.163	0.133	0.0587	-0.0068
p7	0.0172	-0.0229	0.0225	0.0072	-0.0105	-0.0127
r1	1.0	0.2667	0.9378	0.9229	0.5923	0.1696
r2	0.2667	1.0	0.0411	0.5457	0.9046	0.8061
r3	0.9378	0.0411	1.0	0.8597	0.431	0.1069
r4	0.9229	0.5457	0.8597	1.0	0.8239	0.5017
r5	0.5923	0.9046	0.431	0.8239	1.0	0.7818
r6	0.1696	0.8061	0.1069	0.5017	0.7818	1.0

Table 36: Correlation between recursion metrics and others for HaXml.

	r1	r2	r3	r4	r5	r6
c1						
c2	0.0498		0.0498	0.0498	0.0498	0.0498
c3	0.0436		0.0436	0.0436	0.0436	0.0436
c4	-0.0372		-0.0372	-0.0372	-0.0372	-0.0372
c5	-0.0317		-0.0317	-0.0317	-0.0317	-0.0317
c6	-0.0019		-0.0019	-0.0019	-0.0019	-0.0019
d1	-0.0192		-0.0192	-0.0192	-0.0192	-0.0192
d2	0.0557		0.0557	0.0557	0.0557	0.0557
d3	0.0539		0.0539	0.0539	0.0539	0.0539
d4	-0.0235		-0.0235	-0.0235	-0.0235	-0.0235
d5	0.0205		0.0205	0.0205	0.0205	0.0205
d6	-0.0144		-0.0144	-0.0144	-0.0144	-0.0144
d7	-0.0316		-0.0316	-0.0316	-0.0316	-0.0316
d8	-0.0393		-0.0393	-0.0393	-0.0393	-0.0393
d9	-0.0343		-0.0343	-0.0343	-0.0343	-0.0343
d10	-0.0316		-0.0316	-0.0316	-0.0316	-0.0316
d11	-0.0355		-0.0355	-0.0355	-0.0355	-0.0355
d12	-0.0296		-0.0296	-0.0296	-0.0296	-0.0296
m1	0.2959		0.2959	0.2959	0.2959	0.2959
m2	-0.0159		-0.0159	-0.0159	-0.0159	-0.0159
m3	0.0449		0.0449	0.0449	0.0449	0.0449
p1	-0.0094		-0.0094	-0.0094	-0.0094	-0.0094
p2	0.0807		0.0807	0.0807	0.0807	0.0807
p3	0.224		0.224	0.224	0.224	0.224
p4	-0.0105		-0.0105	-0.0105	-0.0105	-0.0105
p5	0.2943		0.2943	0.2943	0.2943	0.2943
p6	0.0794		0.0794	0.0794	0.0794	0.0794
p7	0.6287		0.6287	0.6287	0.6287	0.6287
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 37: Correlation between recursion metrics and others for HUnit.

	r1	r2	r3	r4	r5	r6
c1	0.5343		0.5343	0.5343	0.5343	0.5343
c2	0.1982		0.1982	0.1982	0.1982	0.1982
c3	0.7145		0.7145	0.7145	0.7145	0.7145
c4	0.3271		0.3271	0.3271	0.3271	0.3271
c5	0.3602		0.3602	0.3602	0.3602	0.3602
c6	0.3851		0.3851	0.3851	0.3851	0.3851
d1	0.5203		0.5203	0.5203	0.5203	0.5203
d2	0.3837		0.3837	0.3837	0.3837	0.3837
d3	0.2021		0.2021	0.2021	0.2021	0.2021
d4	0.4557		0.4557	0.4557	0.4557	0.4557
d5	0.3165		0.3165	0.3165	0.3165	0.3165
d6	0.2223		0.2223	0.2223	0.2223	0.2223
d7	0.5909		0.5909	0.5909	0.5909	0.5909
d8	0.6742		0.6742	0.6742	0.6742	0.6742
d9	0.2996		0.2996	0.2996	0.2996	0.2996
d10	0.5295		0.5295	0.5295	0.5295	0.5295
d11	0.6807		0.6807	0.6807	0.6807	0.6807
d12	0.5373		0.5373	0.5373	0.5373	0.5373
m1	0.6523		0.6523	0.6523	0.6523	0.6523
m2	0.5684		0.5684	0.5684	0.5684	0.5684
m3	0.5959		0.5959	0.5959	0.5959	0.5959
p1	0.6473		0.6473	0.6473	0.6473	0.6473
p2	0.6608		0.6608	0.6608	0.6608	0.6608
p3	0.5278		0.5278	0.5278	0.5278	0.5278
p4						
p5	0.6402		0.6402	0.6402	0.6402	0.6402
p6	0.6626		0.6626	0.6626	0.6626	0.6626
p7	0.4857		0.4857	0.4857	0.4857	0.4857
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 38: Correlation between recursion metrics and others for PCF implementation.

	r1	r2	r3	r4	r5	r6
c1	0.6523		0.6523	0.6523	0.6523	0.6523
c2	0.0772		0.0772	0.0772	0.0772	0.0772
c3	0.6568		0.6568	0.6568	0.6568	0.6568
c4	0.2419		0.2419	0.2419	0.2419	0.2419
c5	0.2567		0.2567	0.2567	0.2567	0.2567
c6	0.4494		0.4494	0.4494	0.4494	0.4494
d1	0.2395		0.2395	0.2395	0.2395	0.2395
d2	0.3328		0.3328	0.3328	0.3328	0.3328
d3	-0.1319		-0.1319	-0.1319	-0.1319	-0.1319
d4	0.2586		0.2586	0.2586	0.2586	0.2586
d5	0.4019		0.4019	0.4019	0.4019	0.4019
d6	0.0239		0.0239	0.0239	0.0239	0.0239
d7	0.2359		0.2359	0.2359	0.2359	0.2359
d8	0.2635		0.2635	0.2635	0.2635	0.2635
d9	-0.1968		-0.1968	-0.1968	-0.1968	-0.1968
d10	0.2479		0.2479	0.2479	0.2479	0.2479
d11	0.3254		0.3254	0.3254	0.3254	0.3254
d12	-0.0703		-0.0703	-0.0703	-0.0703	-0.0703
m1	0.3147		0.3147	0.3147	0.3147	0.3147
m2	0.3809		0.3809	0.3809	0.3809	0.3809
m3	0.4081		0.4081	0.4081	0.4081	0.4081
p1	0.3861		0.3861	0.3861	0.3861	0.3861
p2	0.3953		0.3953	0.3953	0.3953	0.3953
p3	0.684		0.684	0.684	0.684	0.684
p4	-0.065		-0.065	-0.065	-0.065	-0.065
p5	0.5217		0.5217	0.5217	0.5217	0.5217
p6	0.4024		0.4024	0.4024	0.4024	0.4024
p7	0.2253		0.2253	0.2253	0.2253	0.2253
r1	1.0		1.0	1.0	1.0	1.0
r2						
r3	1.0		1.0	1.0	1.0	1.0
r4	1.0		1.0	1.0	1.0	1.0
r5	1.0		1.0	1.0	1.0	1.0
r6	1.0		1.0	1.0	1.0	1.0

Table 39: Correlation between recursion metrics and others for Pretty Printer Library.

	r1	r2	r3	r4	r5	r6
c1	0.6534	0.7981	0.2445	0.7452	0.8388	0.7442
c2	-0.0053	-0.0133	0.0044	-0.0063	-0.0108	-0.0084
c3	0.2576	0.1415	0.1982	0.2433	0.207	0.1467
c4	0.3616	0.4058	0.168	0.4102	0.4429	0.3983
c5	0.3153	0.3843	0.1191	0.3598	0.4044	0.3593
c6	-0.0261	-0.003	-0.0283	-0.0224	-0.0134	-0.0074
d1	-0.0048	0.0028	-0.0102	-0.0054	-0.0013	-0.0031
d2	0.3484	0.2259	0.2338	0.3291	0.2994	0.2072
d3	0.1762	0.1568	0.0884	0.1754	0.18	0.1338
d4	-0.0134	-0.0053	-0.0127	-0.0129	-0.0097	-0.0074
d5	-0.0406	0.0006	-0.0485	-0.0344	-0.0177	-0.0092
d6	-0.0579	-0.0146	-0.0568	-0.0512	-0.0351	-0.0215
d7	-0.0112	-0.0052	-0.0097	-0.0107	-0.0085	-0.0064
d8	-0.029	0.0224	-0.0529	-0.022	0.001	0.0041
d9	-0.0966	-0.0433	-0.0851	-0.092	-0.0725	-0.054
d10	-0.0118	-0.0056	-0.0101	-0.0113	-0.009	-0.0067
d11	-0.0331	0.008	-0.0472	-0.0282	-0.0103	-0.006
d12	-0.0874	-0.0384	-0.0774	-0.083	-0.0651	-0.0483
m1	0.4532	0.043	0.4881	0.3812	0.224	0.1102
m2	-0.013	-0.0064	-0.012	-0.0132	-0.0105	-0.009
m3	-0.0148	-0.0075	-0.013	-0.0147	-0.0119	-0.0097
p1	0.636	0.2696	0.5332	0.5753	0.453	0.2901
p2	0.6439	0.1977	0.5922	0.5664	0.408	0.242
p3	0.5313	0.1162	0.5281	0.4622	0.3076	0.1772
p4						
p5	0.474	-0.0257	0.5649	0.3873	0.1886	0.0741
p6	0.6417	0.2004	0.5928	0.5687	0.4107	0.2506
p7	0.2047	0.0259	0.2067	0.1669	0.1021	0.0408
r1	1.0	0.4793	0.8508	0.953	0.7688	0.5688
r2	0.4793	1.0	-0.0244	0.6962	0.9265	0.9553
r3	0.8508	-0.0244	1.0	0.7006	0.3537	0.1499
r4	0.953	0.6962	0.7006	1.0	0.9152	0.7894
r5	0.7688	0.9265	0.3537	0.9152	1.0	0.9502
r6	0.5688	0.9553	0.1499	0.7894	0.9502	1.0

Table 40: Correlation between recursion metrics and others for Typing Haskell in Haskell.

E.2 Tables of Cross-correlation of Callgraph Metrics

	c1	c2	c3	c4	c5	c6
c1	1.0	0.0355	0.3879	0.1854	0.1714	0.1545
c2	0.0355	1.0	-0.0038	-0.1437	-0.0945	-0.119
c3	0.3879	-0.0038	1.0	0.4285	0.4083	0.6196
c4	0.1854	-0.1437	0.4285	1.0	0.9035	0.529
c5	0.1714	-0.0945	0.4083	0.9035	1.0	0.4151
c6	0.1545	-0.119	0.6196	0.529	0.4151	1.0
d1	0.2836	-0.0406	0.6261	0.3084	0.3292	0.5284
d2	0.2469	-0.1297	0.7386	0.5713	0.4803	0.7154
d3	0.0281	-0.1599	0.4995	0.4948	0.3829	0.7184
d4	0.1949	-0.056	0.631	0.2644	0.2225	0.4854
d5	0.1381	-0.0799	0.4676	0.2786	0.1324	0.3318
d6	-0.0019	-0.1076	0.3117	0.2308	0.071	0.3455
d7	0.0661	-0.0952	0.2992	0.2996	0.218	0.1901
d8	0.0224	-0.1279	0.2852	0.2841	0.161	0.2237
d9	-0.0532	-0.1366	0.1132	0.219	0.0806	0.1565
d10	0.1914	-0.0855	0.3939	0.3579	0.2855	0.2617
d11	0.0888	-0.1272	0.3521	0.3384	0.2173	0.3123
d12	-0.0396	-0.1338	0.0845	0.234	0.0999	0.1945
m1	0.2897	0.0576	0.676	0.2985	0.2455	0.3336
m2	0.3238	-0.0346	0.6638	0.3194	0.3261	0.4434
m3	0.2783	-0.0703	0.6467	0.2779	0.281	0.4822
p1	0.3535	-0.015	0.6462	0.3441	0.3175	0.3747
p2	0.329	-0.0039	0.6329	0.3167	0.3078	0.3329
p3	0.3	-0.0553	0.6567	0.3588	0.2665	0.4887
p4	0.1099	-0.0138	0.1351	0.111	0.1103	0.0917
p5	0.302	0.0402	0.5152	0.1808	0.2042	0.1287
p6	0.3436	0.0019	0.6165	0.2879	0.2825	0.3084
p7	-0.0425	0.0241	0.098	-0.0626	-0.0455	0.019
r1	0.766	0.0696	0.4967	0.1437	0.1388	0.1442
r2						
r3	0.766	0.0696	0.4967	0.1437	0.1388	0.1442
r4	0.766	0.0696	0.4967	0.1437	0.1388	0.1442
r5	0.766	0.0696	0.4967	0.1437	0.1388	0.1442
r6	0.766	0.0696	0.4967	0.1437	0.1388	0.1442

Table 41: Correlation between callgraph metrics and others for the CGI Library.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.1858	0.1051	-0.0433	-0.0531	0.0202
c2	0.1858	1.0	0.0155	-0.2039	-0.2316	0.0312
c3	0.1051	0.0155	1.0	0.3469	0.3769	0.5945
c4	-0.0433	-0.2039	0.3469	1.0	0.9567	0.2994
c5	-0.0531	-0.2316	0.3769	0.9567	1.0	0.3321
c6	0.0202	0.0312	0.5945	0.2994	0.3321	1.0
d1	-0.0146	0.0234	0.3639	0.224	0.2713	0.3619
d2	0.1107	-0.2187	0.5676	0.3969	0.4277	0.6232
d3	-0.0456	-0.3437	0.5036	0.4591	0.4313	0.6374
d4	-0.0175	0.1067	0.388	0.2245	0.2505	0.2859
d5	0.2188	-0.0187	0.6005	0.2616	0.2283	0.329
d6	-0.0367	-0.0656	0.5499	0.1697	0.1391	0.4078
d7	-0.0354	-0.1123	0.4454	0.4034	0.4916	0.2003
d8	-0.0524	-0.1829	0.4198	0.6006	0.677	0.2725
d9	-0.052	-0.2086	0.215	0.5899	0.6154	0.1912
d10	-0.0353	-0.0697	0.401	0.3955	0.4537	0.1881
d11	-0.0505	-0.1608	0.4672	0.6311	0.6705	0.2359
d12	-0.0481	-0.1991	0.296	0.6159	0.6481	0.2106
m1	0.5035	0.2436	0.5823	0.2573	0.2563	0.4538
m2	-0.0362	-0.1454	0.1328	0.0037	-0.0294	0.2638
m3	0.0791	0.004	0.4337	0.2072	0.2547	0.3028
p1	0.0729	0.0545	0.4527	0.2828	0.2999	0.2634
p2	0.1474	0.0483	0.456	0.2847	0.3042	0.253
p3	0.2228	-0.0841	0.6474	0.352	0.3776	0.4028
p4	-0.0234	0.0213	0.1775	0.0494	0.0709	0.1454
p5	-0.0244	0.0222	0.3096	0.106	0.0733	0.0688
p6	0.128	0.0797	0.4501	0.2544	0.2665	0.2167
p7	0.3635	0.1798	0.2981	0.2757	0.2476	0.0506
r1	-0.0244	0.0222	0.372	0.1819	0.1529	0.0775
r2						
r3	-0.0244	0.0222	0.372	0.1819	0.1529	0.0775
r4	-0.0244	0.0222	0.372	0.1819	0.1529	0.0775
r5	-0.0244	0.0222	0.372	0.1819	0.1529	0.0775
r6	-0.0244	0.0222	0.372	0.1819	0.1529	0.0775

Table 42: Correlation between callgraph metrics and others for the Haskell Cryptographic Library.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.0983	0.0096	-0.0095	-0.0179	-0.0253
c2	0.0983	1.0	0.1573	-0.0084	0.011	0.0484
c3	0.0096	0.1573	1.0	0.3105	0.2513	0.6725
c4	-0.0095	-0.0084	0.3105	1.0	0.9054	0.4003
c5	-0.0179	0.011	0.2513	0.9054	1.0	0.3046
c6	-0.0253	0.0484	0.6725	0.4003	0.3046	1.0
d1	-0.0198	0.0065	0.3385	0.4904	0.3351	0.3686
d2	0.0545	0.0765	0.6575	0.5458	0.3741	0.6426
d3	-0.0312	0.0158	0.6593	0.3442	0.245	0.7273
d4	-0.0366	0.0052	0.3361	0.363	0.2757	0.2964
d5	-0.02	0.0428	0.3414	0.1716	0.149	0.2475
d6	-0.0465	0.0137	0.3301	0.0879	0.0932	0.2729
d7	-0.0242	0.0127	0.2442	0.2953	0.2537	0.1527
d8	-0.0325	-0.0478	0.1931	0.2768	0.2378	0.1302
d9	-0.0335	-0.0471	0.0876	0.3003	0.2828	0.1243
d10	-0.0179	0.0392	0.2418	0.358	0.2787	0.2127
d11	-0.0256	-0.0194	0.1659	0.2465	0.2182	0.1467
d12	-0.0285	-0.0259	0.1492	0.3557	0.3242	0.1951
m1	0.0794	0.2474	0.503	0.088	0.073	0.2779
m2	-0.0003	0.0212	0.3838	0.4465	0.2805	0.4046
m3	0.0033	0.0024	0.3057	0.3282	0.1758	0.3757
p1	0.0432	0.0472	0.3873	0.4845	0.2843	0.3335
p2	0.0681	0.0837	0.3626	0.4212	0.2333	0.2723
p3	0.0817	0.2149	0.3933	0.3058	0.1965	0.283
p4	0.0925	0.001	0.0749	0.0645	0.0156	0.0065
p5	-0.0047	-0.0029	-0.0521	-0.0402	-0.033	-0.0458
p6	0.0677	0.0904	0.3572	0.4083	0.229	0.2498
p7	0.0066	0.03	0.1291	0.0249	-0.0275	-0.0226
r1	-0.0346	0.2414	0.3073	-0.0985	-0.0944	0.0755
r2						
r3	-0.0346	0.2414	0.3073	-0.0985	-0.0944	0.0755
r4	-0.0346	0.2414	0.3073	-0.0985	-0.0944	0.0755
r5	-0.0346	0.2414	0.3073	-0.0985	-0.0944	0.0755
r6	-0.0346	0.2414	0.3073	-0.0985	-0.0944	0.0755

Table 43: Correlation between callgraph metrics and others for the Haskell DSP Library.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.0295	0.2928	0.1375	0.0734	0.1705
c2	0.0295	1.0	-0.0557	-0.0981	-0.084	-0.1012
c3	0.2928	-0.0557	1.0	0.3979	0.3486	0.5367
c4	0.1375	-0.0981	0.3979	1.0	0.809	0.6287
c5	0.0734	-0.084	0.3486	0.809	1.0	0.4083
c6	0.1705	-0.1012	0.5367	0.6287	0.4083	1.0
d1	0.3304	-0.0102	0.7628	0.3344	0.3815	0.359
d2	0.1595	-0.1309	0.7388	0.4537	0.3841	0.6594
d3	-0.0112	-0.1693	0.3989	0.3046	0.1964	0.6666
d4	0.3506	-0.0194	0.7444	0.2785	0.2983	0.3645
d5	0.1255	-0.0948	0.4553	0.2215	0.0987	0.4629
d6	0.122	-0.0893	0.2202	0.0833	-0.0276	0.441
d7	0.0813	-0.0496	0.3432	0.4398	0.4136	0.2893
d8	0.0401	-0.0677	0.2661	0.4941	0.4222	0.3261
d9	-0.0744	-0.081	0.0082	0.3225	0.1507	0.2702
d10	0.1119	-0.0428	0.468	0.4376	0.4492	0.2884
d11	0.0513	-0.0742	0.3324	0.5097	0.4633	0.3393
d12	-0.079	-0.0934	0.0469	0.3139	0.1614	0.2905
m1	0.3295	0.0399	0.7439	0.2577	0.2391	0.3033
m2	0.2692	-0.0554	0.5643	0.1461	0.2148	0.2563
m3	0.2083	-0.0636	0.451	0.0998	0.1626	0.2018
p1	0.4074	-0.0183	0.7487	0.3094	0.3337	0.3132
p2	0.3901	-0.0057	0.7662	0.282	0.3242	0.2876
p3	0.2307	-0.0475	0.7372	0.2779	0.2753	0.3275
p4	0.0423	-0.0051	0.0904	0.0653	0.0312	0.1363
p5	0.1973	0.0151	0.5556	0.144	0.2159	0.1139
p6	0.3716	-0.0044	0.7375	0.2573	0.286	0.2584
p7	-0.0144	-0.0164	0.2448	0.0885	0.0772	0.0449
r1	0.1171	0.0325	0.5444	0.1148	0.1273	0.0961
r2						
r3	0.1171	0.0325	0.5444	0.1148	0.1273	0.0961
r4	0.1171	0.0325	0.5444	0.1148	0.1273	0.0961
r5	0.1171	0.0325	0.5444	0.1148	0.1273	0.0961
r6	0.1171	0.0325	0.5444	0.1148	0.1273	0.0961

Table 44: Correlation between callgraph metrics and others for FGL.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.005	0.1421	0.1405	0.0856	0.055
c2	0.005	1.0	-0.0931	-0.1216	-0.0814	-0.1098
c3	0.1421	-0.0931	1.0	0.4787	0.4628	0.6397
c4	0.1405	-0.1216	0.4787	1.0	0.7959	0.5803
c5	0.0856	-0.0814	0.4628	0.7959	1.0	0.3727
c6	0.055	-0.1098	0.6397	0.5803	0.3727	1.0
d1	0.0858	-0.0427	0.4724	0.3317	0.3584	0.3694
d2	0.0428	-0.1432	0.7097	0.5699	0.4562	0.6997
d3	0.0054	-0.157	0.5377	0.4862	0.3465	0.6577
d4	0.0622	-0.0324	0.3941	0.2454	0.2328	0.3266
d5	0.0087	-0.0905	0.5146	0.3028	0.1985	0.4707
d6	-0.0248	-0.0935	0.3764	0.2058	0.1205	0.4195
d7	0.0417	-0.0124	0.3947	0.3453	0.4002	0.2288
d8	0.0262	-0.04	0.4295	0.456	0.4705	0.284
d9	-0.0243	-0.0204	0.2227	0.3161	0.2989	0.1944
d10	0.0689	-0.0246	0.4018	0.3309	0.3809	0.2543
d11	0.0385	-0.0498	0.4386	0.4549	0.4598	0.3057
d12	-0.0153	-0.0352	0.2491	0.3254	0.3057	0.2306
m1	0.1528	-0.0184	0.5573	0.2244	0.2349	0.2539
m2	0.1159	-0.0741	0.4781	0.2842	0.3029	0.3046
m3	0.0673	-0.0796	0.3946	0.2499	0.2648	0.2705
p1	0.1536	-0.0499	0.6131	0.3707	0.347	0.3878
p2	0.1396	-0.0542	0.5636	0.3311	0.3036	0.3709
p3	0.1125	-0.0921	0.5765	0.298	0.2213	0.4388
p4	-0.0065	-0.0234	0.0686	0.1459	0.1517	0.0937
p5	0.0005	-0.0205	0.307	0.0846	0.0891	0.1319
p6	0.1207	-0.0456	0.5621	0.3116	0.2916	0.3423
p7	-0.0213	-0.0047	0.2924	0.1624	0.1539	0.1701
r1	0.197	0.0336	0.3271	0.0993	0.1293	0.0604
r2	0.4984	-0.0007	0.0109	0.0149	0.0047	-0.0149
r3	0.1032	0.0343	0.3304	0.098	0.1305	0.0643
r4	0.197	0.0336	0.3271	0.0993	0.1293	0.0604
r5	0.2782	0.0319	0.3131	0.0972	0.1238	0.0547
r6	0.2782	0.0319	0.3131	0.0972	0.1238	0.0547

Table 45: Correlation between callgraph metrics and others for the Library of Geometric Algorithms.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.1074	0.1063	0.6163	0.5649	0.2665
c2	0.1074	1.0	-0.2974	-0.1472	-0.1299	-0.3762
c3	0.1063	-0.2974	1.0	0.6948	0.7122	0.9069
c4	0.6163	-0.1472	0.6948	1.0	0.9897	0.7691
c5	0.5649	-0.1299	0.7122	0.9897	1.0	0.7353
c6	0.2665	-0.3762	0.9069	0.7691	0.7353	1.0
d1	0.2457	-0.2887	0.5133	0.643	0.6031	0.7105
d2	0.2303	-0.3545	0.7433	0.6637	0.5976	0.9151
d3	-0.0478	-0.3835	0.5423	0.1962	0.1222	0.6952
d4	0.0975	-0.3103	0.4133	0.4336	0.3935	0.6226
d5	0.2709	-0.3757	0.8024	0.6831	0.6293	0.9491
d6	-0.1339	-0.4127	0.4983	0.0722	0.0345	0.6028
d7	0.5799	-0.1061	0.4564	0.8633	0.8234	0.621
d8	0.0007	-0.099	0.4684	0.5024	0.4735	0.5087
d9	0.1698	-0.0341	0.2293	0.38	0.3108	0.3684
d10	0.496	-0.1489	0.5558	0.9065	0.8868	0.6807
d11	-0.0115	-0.1266	0.5803	0.5874	0.5826	0.5736
d12	0.2354	-0.0858	0.424	0.5436	0.4784	0.544
m1	0.1157	-0.0647	0.5468	0.3481	0.4157	0.3856
m2	0.3604	-0.2666	0.5855	0.7944	0.7722	0.7202
m3	0.2052	-0.2435	0.5586	0.7344	0.7402	0.6249
p1	0.1996	-0.2667	0.5782	0.7392	0.7322	0.6697
p2	0.265	-0.2328	0.5419	0.7551	0.7578	0.6155
p3	0.1601	-0.2927	0.8789	0.7071	0.75	0.7859
p4						
p5	0.6954	-0.1478	0.3112	0.6594	0.6097	0.5016
p6	0.203	-0.2363	0.5502	0.7304	0.7396	0.603
p7	0.0192	-0.2419	0.3922	0.5338	0.5358	0.4612
r1						
r2						
r3						
r4						
r5						
r6						

Table 46: Correlation between callgraph metrics and others for GetOpt.

	c1	c2	c3	c4	c5	c6
c1	1.0	-0.0205	-0.0846	0.9395	0.9868	0.4402
c2	-0.0205	1.0	-0.0427	-0.0424	-0.0228	-0.092
c3	-0.0846	-0.0027	1.0	0.0036	-0.057	0.2102
c4	0.9395	-0.0424	0.0036	1.0	0.9611	0.5779
c5	0.9868	-0.0228	-0.057	0.9611	1.0	0.4609
c6	0.4402	-0.092	0.2102	0.5779	0.4609	1.0
d1	-0.0759	-0.0048	0.5494	0.0621	-0.0186	0.2513
d2	-0.2338	-0.0406	0.3986	-0.0823	-0.1902	0.4096
d3	0.2218	-0.1056	0.1454	0.2929	0.2536	0.5308
d4	-0.1644	0.0396	0.2746	-0.0774	-0.1364	0.1498
d5	-0.2402	0.0538	0.1116	-0.2688	-0.2417	-0.1401
d6	-0.2359	0.0286	0.0382	-0.2928	-0.2423	-0.1821
d7	0.0476	-0.0165	0.8434	0.0534	0.0503	0.0538
d8	0.3509	-0.0932	0.0652	0.3702	0.3582	0.2784
d9	0.2754	-0.0814	0.0114	0.2686	0.2691	0.1867
d10	0.0452	-0.0183	0.8528	0.0581	0.0494	0.0669
d11	0.344	-0.0946	0.0749	0.3744	0.3536	0.3012
d12	0.2768	-0.0821	0.0103	0.2751	0.2712	0.2002
m1	-0.0146	-0.0017	0.1973	0.0578	0.0667	0.0527
m2	-0.0506	0.0	0.7999	0.0495	-0.0272	0.2089
m3	-0.0538	0.0067	0.8186	0.0323	-0.0273	0.1643
p1	-0.1259	0.0358	0.4142	0.0233	-0.0835	0.2474
p2	-0.0977	0.0423	0.3928	0.0329	-0.0608	0.2232
p3	-0.096	0.0558	0.309	0.0284	-0.0762	0.2716
p4	-0.0359	-0.0017	0.0934	0.0043	-0.0285	0.0627
p5	-0.0251	0.0121	0.3298	0.0408	-0.0071	0.1099
p6	-0.1007	0.0347	0.3695	0.0229	-0.0673	0.2124
p7	-0.0482	0.0249	0.1195	-0.0009	-0.0392	0.1067
r1	0.8689	-0.0168	-0.0153	0.8747	0.8608	0.4697
r2	0.1291	0.1407	0.5407	0.1108	0.1275	0.0601
r3	-0.0508	0.0135	0.1466	-0.0065	-0.046	0.0834
r4	0.1286	0.1406	0.5411	0.1107	0.127	0.0606
r5	0.1289	0.1406	0.5408	0.1107	0.1273	0.0602
r6	0.0742	0.126	0.0293	0.0652	0.0732	0.0347

Table 47: Correlation between callgraph metrics and others for Haddock.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.013	0.1468	0.1392	0.0576	0.0905
c2	0.013	1.0	-0.0204	-0.079	-0.0498	-0.065
c3	0.1468	-0.0204	1.0	0.2913	0.2454	0.5399
c4	0.1392	-0.079	0.2913	1.0	0.7512	0.4828
c5	0.0576	-0.0498	0.2454	0.7512	1.0	0.3052
c6	0.0905	-0.065	0.5399	0.4828	0.3052	1.0
d1	0.0061	-0.0261	0.3807	0.2589	0.3697	0.2008
d2	0.0822	-0.0443	0.6449	0.2488	0.33	0.5027
d3	-0.0252	-0.036	0.3029	0.2161	0.1394	0.5411
d4	-0.0043	-0.0189	0.2826	0.1809	0.2175	0.1603
d5	0.0355	0.0917	0.517	0.0546	0.0829	0.3295
d6	-0.0245	0.1216	0.2364	-0.0092	0.0569	0.3044
d7	-0.0065	-0.0211	0.2407	0.1823	0.1879	0.1475
d8	-0.0001	-0.0954	0.2347	0.5929	0.3845	0.3028
d9	-0.0353	-0.0876	0.0043	0.5118	0.2902	0.2078
d10	-0.0047	-0.0215	0.2513	0.185	0.1937	0.1497
d11	0.0194	-0.1009	0.2993	0.6032	0.429	0.3282
d12	-0.0322	-0.0875	0.0219	0.5074	0.3059	0.2194
m1	-0.0056	-0.0127	0.3706	0.0871	0.0663	0.0512
m2	0.0438	-0.0318	0.4614	0.2449	0.2815	0.2359
m3	0.0641	-0.039	0.5749	0.3269	0.3555	0.2954
p1	0.0604	-0.0197	0.5673	0.2651	0.2742	0.2828
p2	0.1173	-0.0317	0.6051	0.3229	0.3119	0.301
p3	0.053	-0.0452	0.5182	0.124	0.0927	0.2352
p4	-0.0148	-0.0258	0.1743	0.1842	0.0797	0.1366
p5	0.169	-0.0532	0.368	0.1338	0.149	0.147
p6	0.0982	-0.0353	0.5908	0.3028	0.2979	0.286
p7	0.0244	-0.0308	0.2913	0.2055	0.1299	0.1737
r1	0.1591	0.0016	0.3785	-0.0552	-0.012	0.1449
r2						
r3	0.1591	0.0016	0.3785	-0.0552	-0.012	0.1449
r4	0.1591	0.0016	0.3785	-0.0552	-0.012	0.1449
r5	0.1591	0.0016	0.3785	-0.0552	-0.012	0.1449
r6	0.1591	0.0016	0.3785	-0.0552	-0.012	0.1449

Table 48: Correlation between callgraph metrics and others for Happy.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.0313	0.288	0.6956	0.7969	0.3403
c2	0.0313	1.0	0.0103	-0.0111	0.004	-0.0516
c3	0.288	0.0103	1.0	0.4856	0.4447	0.4398
c4	0.6956	-0.0111	0.4856	1.0	0.9313	0.6432
c5	0.7969	0.004	0.4447	0.9313	1.0	0.4857
c6	0.3403	-0.0516	0.4398	0.6432	0.4857	1.0
d1	0.1409	0.0139	0.6746	0.2623	0.2307	0.2332
d2	0.1616	-0.0549	0.5954	0.4408	0.2949	0.616
d3	0.0741	-0.0821	0.2758	0.3322	0.1698	0.6447
d4	0.2066	0.0311	0.6561	0.3167	0.3184	0.2567
d5	0.3285	0.0282	0.4401	0.5411	0.5031	0.4672
d6	0.1782	0.0485	0.2625	0.3343	0.3079	0.3881
d7	0.2383	0.0372	0.6455	0.3	0.3152	0.189
d8	0.3337	-0.0146	0.4851	0.564	0.5379	0.3816
d9	0.1442	-0.0541	0.2127	0.3991	0.2979	0.3681
d10	0.2166	0.0347	0.6313	0.2846	0.2963	0.1824
d11	0.3239	-0.012	0.5031	0.5665	0.5382	0.3774
d12	0.1591	-0.0477	0.261	0.4292	0.3295	0.3679
m1	0.0363	-0.002	0.5311	0.0937	0.0907	0.1378
m2	0.1461	0.0048	0.6722	0.225	0.2117	0.1841
m3	0.1433	-0.0021	0.6089	0.1983	0.1936	0.1586
p1	0.1393	0.0125	0.6026	0.2163	0.1865	0.1823
p2	0.1144	0.0087	0.5527	0.1783	0.1555	0.1651
p3	0.1661	-0.062	0.5229	0.2865	0.2232	0.3757
p4	0.0349	-0.0041	0.163	0.111	0.0734	0.0782
p5	0.1141	-0.005	0.5255	0.1448	0.1471	0.1148
p6	0.1171	0.0069	0.5454	0.1763	0.1568	0.1551
p7	0.0734	-0.0065	0.3385	0.0957	0.0993	0.0816
r1	0.0742	0.002	0.346	0.2266	0.1979	0.1761
r2	0.0431	-0.0106	0.0976	0.2395	0.2128	0.1342
r3	0.0556	0.0071	0.3326	0.0729	0.069	0.1029
r4	0.0511	-0.0095	0.1469	0.2487	0.2216	0.1487
r5	0.0456	-0.0122	0.1088	0.2443	0.2173	0.1359
r6	0.0267	-0.0033	0.0503	0.1431	0.1288	0.0832

Table 49: Correlation between callgraph metrics and others for Hat.

	c1	c2	c3	c4	c5	c6
c1	1.0	-0.0162	0.2826	0.3481	0.227	0.2049
c2	-0.0162	1.0	-0.0292	-0.0942	-0.0707	-0.0775
c3	0.2826	-0.0292	1.0	0.562	0.5193	0.7098
c4	0.3481	-0.0942	0.562	1.0	0.7563	0.7014
c5	0.227	-0.0707	0.5193	0.7563	1.0	0.5218
c6	0.2049	-0.0775	0.7098	0.7014	0.5218	1.0
d1	0.1476	-0.032	0.4874	0.3068	0.2818	0.3113
d2	0.1556	-0.0852	0.7323	0.5775	0.4591	0.7999
d3	0.0497	-0.109	0.5443	0.5288	0.3979	0.7663
d4	0.1222	-0.0463	0.5621	0.3792	0.3702	0.3953
d5	0.0523	-0.0669	0.5442	0.4589	0.4216	0.5286
d6	-0.003	-0.0817	0.411	0.3734	0.3538	0.4749
d7	0.0942	-0.0408	0.3684	0.2907	0.1976	0.2122
d8	0.0558	-0.114	0.4056	0.559	0.3824	0.4468
d9	0.0279	-0.1012	0.2257	0.3575	0.1428	0.3096
d10	0.1488	-0.0445	0.4694	0.3297	0.2626	0.2659
d11	0.0847	-0.1151	0.5078	0.5665	0.4621	0.5026
d12	0.0329	-0.1062	0.3081	0.3796	0.2176	0.3664
m1	0.0045	-0.0076	0.0499	-0.0145	-0.0069	0.045
m2	0.1778	-0.0468	0.4352	0.1938	0.1787	0.1796
m3	0.1485	-0.0514	0.3532	0.1406	0.1312	0.144
p1	0.1663	-0.0147	0.4126	0.2219	0.1438	0.258
p2	0.1278	-0.0169	0.3654	0.1989	0.1235	0.2381
p3	0.07	-0.044	0.4845	0.3334	0.1519	0.4822
p4	-0.0095	-0.011	0.0153	0.0506	0.0256	0.0504
p5	0.0258	-0.0368	0.3062	0.2218	0.0936	0.225
p6	0.1285	-0.0172	0.3717	0.1985	0.1215	0.2362
p7	0.0172	-0.0259	0.2016	0.1105	0.0934	0.1718
r1	0.4838	0.0001	0.4144	0.275	0.1828	0.2779
r2	0.4891	-0.0165	0.0964	0.2006	0.1113	0.1122
r3	0.3451	0.0048	0.3898	0.2418	0.162	0.2536
r4	0.5395	-0.0045	0.3762	0.3053	0.1928	0.2701
r5	0.5399	-0.0122	0.2428	0.2746	0.1583	0.1983
r6	0.2775	-0.0104	0.0636	0.1498	0.0678	0.0738

Table 50: Correlation between callgraph metrics and others for HaXml.

	c1	c2	c3	c4	c5	c6
c1						
c2		1.0	0.1171	0.0046	-0.0523	0.2253
c3		0.1171	1.0	0.6095	0.6257	0.6353
c4		0.0046	0.6095	1.0	0.8157	0.6039
c5		-0.0523	0.6257	0.8157	1.0	0.4369
c6		0.2253	0.6353	0.6039	0.4369	1.0
d1		-0.018	0.6213	0.5564	0.5515	0.3472
d2		0.0606	0.7993	0.6979	0.6029	0.662
d3		0.1719	0.6269	0.5539	0.4465	0.8029
d4		-0.0104	0.6088	0.4468	0.4804	0.3649
d5		0.0738	0.6745	0.4204	0.4728	0.5925
d6		0.0984	0.5332	0.263	0.3212	0.5805
d7		-0.0698	0.4831	0.551	0.7601	0.3337
d8		-0.0582	0.5413	0.6472	0.7213	0.3497
d9		-0.0134	0.2967	0.3999	0.5411	0.3023
d10		-0.0693	0.4814	0.5661	0.7415	0.3347
d11		-0.0666	0.5402	0.603	0.7074	0.3413
d12		-0.0308	0.3469	0.4165	0.5499	0.3189
m1		0.247	0.5463	0.3564	0.3261	0.3878
m2		-0.0052	0.611	0.5189	0.495	0.3918
m3		0.0083	0.612	0.4864	0.448	0.3348
p1		0.0445	0.5536	0.4857	0.452	0.2813
p2		0.0436	0.4999	0.4421	0.403	0.2544
p3		0.1694	0.6586	0.5862	0.4728	0.5915
p4		-0.0241	0.2325	0.2582	0.2402	0.0632
p5		0.072	0.2886	0.2925	0.2373	0.1133
p6		0.0363	0.4949	0.4358	0.4021	0.2483
p7		-0.0053	0.2907	0.1038	0.0614	0.0839
r1		0.0498	0.0436	-0.0372	-0.0317	-0.0019
r2						
r3		0.0498	0.0436	-0.0372	-0.0317	-0.0019
r4		0.0498	0.0436	-0.0372	-0.0317	-0.0019
r5		0.0498	0.0436	-0.0372	-0.0317	-0.0019
r6		0.0498	0.0436	-0.0372	-0.0317	-0.0019

Table 51: Correlation between callgraph metrics and others for HUnit.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.2919	0.4614	0.2511	0.2257	0.3953
c2	0.2919	1.0	-0.1107	-0.1007	-0.1007	0.0575
c3	0.4614	0.1107	1.0	0.4974	0.4957	0.6409
c4	0.2511	-0.1007	0.4974	1.0	0.9849	0.7156
c5	0.2257	-0.1007	0.4957	0.9849	1.0	0.6428
c6	0.3953	0.0575	0.6409	0.7156	0.6428	1.0
d1	0.2657	0.1049	0.705	0.1285	0.1532	0.3617
d2	0.4064	0.039	0.6495	0.698	0.6486	0.9177
d3	0.2815	0.0019	0.3874	0.4098	0.3152	0.791
d4	0.1894	0.0796	0.6337	0.1041	0.1303	0.34
d5	0.2865	-0.0026	0.5565	0.7429	0.7125	0.8604
d6	0.1667	0.0069	0.4165	0.3249	0.28	0.6733
d7	0.28	0.1185	0.7682	0.1361	0.1738	0.3169
d8	0.4688	0.1653	0.7666	0.2904	0.3136	0.3913
d9	0.381	0.0991	0.2876	0.2022	0.1519	0.3833
d10	0.2447	0.1025	0.6872	0.0965	0.1289	0.2975
d11	0.4798	0.1623	0.8057	0.2807	0.3068	0.4017
d12	0.5437	0.1345	0.5897	0.3168	0.3004	0.4691
m1	0.3724	0.1474	0.8835	0.1527	0.1767	0.3476
m2	0.3016	0.0983	0.7812	0.1237	0.1483	0.347
m3	0.3842	0.1229	0.8634	0.0908	0.1061	0.3513
p1	0.3866	0.136	0.8734	0.2487	0.2694	0.4375
p2	0.3942	0.1501	0.902	0.2159	0.2379	0.4036
p3	0.4578	0.188	0.7455	0.3113	0.2679	0.7385
p4						
p5	0.4247	0.1724	0.8647	0.1442	0.1606	0.3168
p6	0.3947	0.1533	0.9013	0.2093	0.2323	0.396
p7	0.1694	0.0908	0.7617	0.1095	0.1436	0.3089
r1	0.5343	0.1982	0.7145	0.3271	0.3602	0.3851
r2						
r3	0.5343	0.1982	0.7145	0.3271	0.3602	0.3851
r4	0.5343	0.1982	0.7145	0.3271	0.3602	0.3851
r5	0.5343	0.1982	0.7145	0.3271	0.3602	0.3851
r6	0.5343	0.1982	0.7145	0.3271	0.3602	0.3851

Table 52: Correlation between callgraph metrics and others for PCF implementation.

	c1	c2	c3	c4	c5	c6
c1	1.0	0.0793	0.6377	0.5044	0.4693	0.4142
c2	0.0793	1.0	-0.181	-0.181	-0.1005	-0.2821
c3	0.6377	-0.0181	1.0	0.4503	0.4762	0.5585
c4	0.5044	-0.181	0.4503	1.0	0.9283	0.7457
c5	0.4693	-0.1005	0.4762	0.9283	1.0	0.6718
c6	0.4142	-0.2821	0.5585	0.7457	0.6718	1.0
d1	0.3015	-0.0426	0.518	0.294	0.2818	0.3528
d2	0.2855	-0.2086	0.5836	0.3537	0.2978	0.6286
d3	-0.1427	-0.2601	-0.1008	0.1453	-0.0218	0.3136
d4	0.2139	-0.0532	0.5422	0.2027	0.2129	0.3649
d5	0.2309	-0.0988	0.5821	0.0401	0.0796	0.4017
d6	-0.1367	-0.1067	0.0651	-0.1903	-0.1795	0.1748
d7	0.3445	-0.0868	0.5634	0.3754	0.3998	0.3718
d8	0.3348	-0.1735	0.4689	0.3668	0.299	0.3601
d9	-0.0433	-0.2601	-0.1188	0.0555	-0.1013	0.0455
d10	0.3296	-0.0662	0.5663	0.3669	0.4016	0.3642
d11	0.3173	-0.1247	0.5291	0.3513	0.3317	0.3947
d12	0.0821	-0.2503	0.0113	0.1544	0.0184	0.1294
m1	0.2042	-0.0093	0.3867	0.1411	0.113	0.2964
m2	0.415	-0.0249	0.6551	0.3197	0.3157	0.3883
m3	0.4076	-0.0077	0.6011	0.2697	0.272	0.3758
p1	0.4454	-0.0157	0.675	0.3322	0.3255	0.3824
p2	0.411	-0.0114	0.6434	0.2963	0.2996	0.3686
p3	0.6142	-0.0136	0.6446	0.3182	0.2771	0.4856
p4	-0.0424	-0.0347	0.0014	0.0575	0.0565	0.0784
p5	0.404	0.0311	0.6618	0.1914	0.2022	0.3142
p6	0.4241	-0.0061	0.629	0.292	0.2886	0.3645
p7	0.2409	0.0426	0.2841	0.0657	0.1545	0.1698
r1	0.6523	0.0772	0.6568	0.2419	0.2567	0.4494
r2						
r3	0.6523	0.0772	0.6568	0.2419	0.2567	0.4494
r4	0.6523	0.0772	0.6568	0.2419	0.2567	0.4494
r5	0.6523	0.0772	0.6568	0.2419	0.2567	0.4494
r6	0.6523	0.0772	0.6568	0.2419	0.2567	0.4494

Table 53: Correlation between callgraph metrics and others for Pretty Printer Library.

	c1	c2	c3	c4	c5	c6
c1	1.0	-0.0158	0.2282	0.4965	0.4766	-0.0056
c2	-0.0158	1.0	-0.0654	-0.1034	-0.0725	-0.0525
c3	0.2282	-0.0654	1.0	0.3829	0.3455	0.5855
c4	0.4965	-0.1034	0.3829	1.0	0.9007	0.0567
c5	0.4766	-0.0725	0.3455	0.9007	1.0	0.0893
c6	-0.0056	-0.0525	0.5855	0.0567	0.0893	1.0
d1	0.0082	-0.0118	0.81	0.1147	0.157	0.3796
d2	0.3425	-0.1619	0.4928	0.5963	0.4017	0.2603
d3	0.2033	-0.1624	0.4662	0.4564	0.2894	0.4143
d4	-0.0068	-0.011	0.6634	0.1224	0.1407	0.0768
d5	-0.0039	-0.1315	0.2324	0.1369	-0.022	0.0942
d6	-0.0305	-0.1151	0.2274	0.0778	-0.0509	0.1344
d7	-0.0068	-0.0106	0.5343	0.0924	0.1087	0.0378
d8	0.0357	-0.1147	0.4523	0.2973	0.2113	0.0054
d9	-0.0574	-0.1054	0.3508	0.1814	0.0849	0.0053
d10	-0.0075	-0.0102	0.5632	0.098	0.1154	0.0401
d11	0.0148	-0.0854	0.6106	0.2806	0.2381	0.0361
d12	-0.0516	-0.0905	0.4835	0.2052	0.1351	0.0304
m1	0.199	0.0073	0.2215	0.1287	0.097	0.0125
m2	-0.0044	-0.0121	0.8016	0.1055	0.1466	0.373
m3	-0.007	-0.0122	0.7933	0.1077	0.1463	0.342
p1	0.6246	-0.0015	0.34	0.4508	0.3855	0.005
p2	0.5467	-0.0026	0.3245	0.3914	0.3308	-0.003
p3	0.2829	-0.0221	0.3379	0.3902	0.2556	0.109
p4						
p5	0.2349	-0.0055	0.2091	0.1403	0.1123	-0.0184
p6	0.5564	-0.0043	0.3165	0.3832	0.3292	-0.0071
p7	0.0953	-0.0108	0.0557	0.0861	0.0668	-0.0153
r1	0.6534	-0.0053	0.2576	0.3616	0.3153	-0.0261
r2	0.7981	-0.0133	0.1415	0.4058	0.3843	-0.003
r3	0.2445	0.0044	0.1982	0.168	0.1191	-0.0283
r4	0.7452	-0.0063	0.2433	0.4102	0.3598	-0.0224
r5	0.8388	-0.0108	0.207	0.4429	0.4044	-0.0134
r6	0.7442	-0.0084	0.1467	0.3983	0.3593	-0.0074

Table 54: Correlation between callgraph metrics and others for Typing Haskell in Haskell.

E.3 Tables of Cross-correlation of Distance Metrics

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.2836	0.2469	0.0281	0.1949	0.1381	-0.0019	0.0661	0.0224	-0.0532	0.1914	0.0888	-0.0396
c2	-0.0406	-0.1297	-0.1599	-0.0566	-0.0799	-0.1076	-0.0992	-0.1279	-0.1366	-0.0855	-0.1272	-0.1338
c3	0.6261	0.7386	0.4995	0.631	0.4676	0.3117	0.2992	0.2952	0.1132	0.3939	0.3521	0.0845
c4	0.3084	0.5713	0.4948	0.2644	0.2786	0.2208	0.2996	0.2841	0.219	0.3579	0.3384	0.234
c5	0.3292	0.4803	0.3829	0.2225	0.1324	0.071	0.218	0.161	0.0806	0.2855	0.2173	0.0999
c6	0.5284	0.7154	0.7184	0.4854	0.3318	0.3455	0.1901	0.2237	0.1565	0.2617	0.3123	0.1945
d1	1.0	0.6858	0.4634	0.7562	0.2192	0.1242	0.2547	0.117	-0.0288	0.4761	0.2082	-0.0194
d2	0.6858	1.0	0.8448	0.6328	0.5712	0.4632	0.3386	0.3906	0.2905	0.4499	0.4697	0.3078
d3	0.4634	0.8448	1.0	0.4277	0.416	0.5091	0.2114	0.3172	0.3543	0.2459	0.3696	0.3725
d4	0.7562	0.6328	0.4277	1.0	0.6106	0.5075	0.3966	0.3772	0.2222	0.5246	0.4251	0.1884
d5	0.2192	0.5712	0.416	0.6106	1.0	0.8294	0.4151	0.6585	0.601	0.4282	0.6591	0.5719
d6	0.1242	0.4632	0.5091	0.5075	0.8294	1.0	0.3884	0.611	0.7238	0.3569	0.5995	0.695
d7	0.2547	0.3386	0.2114	0.3966	0.4151	0.3884	1.0	0.712	0.6318	0.9001	0.6355	0.4828
d8	0.117	0.3906	0.3172	0.3772	0.6585	0.611	0.712	1.0	0.8616	0.6162	0.8767	0.6815
d9	-0.0288	0.2905	0.3543	0.2222	0.601	0.7238	0.6318	0.8616	1.0	0.526	0.7778	0.8932
d10	0.4761	0.4499	0.2459	0.5246	0.4282	0.3569	0.9001	0.6162	0.526	1.0	0.6953	0.4855
d11	0.2082	0.4697	0.3696	0.4251	0.6591	0.5995	0.6355	0.8767	0.7778	0.6953	1.0	0.7869
d12	-0.0194	0.3078	0.3725	0.1884	0.5719	0.695	0.4828	0.6815	0.8932	0.4855	0.7869	1.0
m1	0.4639	0.4498	0.2247	0.3253	0.1993	0.0828	0.1999	0.1355	0.0148	0.3378	0.208	0.0033
m2	0.9232	0.6455	0.3683	0.7298	0.2757	0.1381	0.295	0.1699	0.0091	0.5368	0.2421	-0.0086
m3	0.8406	0.6612	0.4304	0.6947	0.3087	0.185	0.2404	0.2165	0.0506	0.3876	0.2274	0.0096
p1	0.8076	0.6505	0.3378	0.642	0.3205	0.1277	0.255	0.1725	0.0083	0.5225	0.2512	0.0037
p2	0.7682	0.5933	0.2823	0.5754	0.2413	0.0692	0.2173	0.1195	-0.0383	0.4814	0.2007	-0.0407
p3	0.3733	0.5958	0.4493	0.3481	0.3125	0.2579	0.162	0.1954	0.1175	0.2392	0.2334	0.1117
p4	0.5759	0.2552	0.0794	0.4274	0.0704	0.0123	0.1021	-0.0125	-0.0538	0.3505	0.028	-0.0413
p5	0.5449	0.2994	0.0524	0.3298	0.0404	-0.0614	0.056	-0.0186	-0.1304	0.3047	0.0743	-0.1196
p6	0.7556	0.5621	0.2508	0.5594	0.2218	0.0587	0.2028	0.1029	-0.046	0.468	0.1791	-0.05
p7	0.0177	0.0111	-0.003	-0.0512	-0.0923	-0.1037	-0.0405	-0.0525	-0.0837	-0.0389	-0.0406	-0.0902
r1	0.2918	0.2135	-0.0006	0.2118	0.1174	0.0227	0.0487	0.0034	-0.0853	0.1831	0.0796	-0.0751
r2												
r3	0.2918	0.2135	-0.0006	0.2118	0.1174	0.0227	0.0487	0.0034	-0.0853	0.1831	0.0796	-0.0751
r4	0.2918	0.2135	-0.0006	0.2118	0.1174	0.0227	0.0487	0.0034	-0.0853	0.1831	0.0796	-0.0751
r5	0.2918	0.2135	-0.0006	0.2118	0.1174	0.0227	0.0487	0.0034	-0.0853	0.1831	0.0796	-0.0751
r6	0.2918	0.2135	-0.0006	0.2118	0.1174	0.0227	0.0487	0.0034	-0.0853	0.1831	0.0796	-0.0751

Table 55: Correlation between distance metrics and others for the CGI Library.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	-0.0146	0.1107	-0.0456	-0.0175	0.2188	-0.0367	-0.0354	-0.0524	-0.052	-0.0353	-0.0505	-0.0481
c2	0.0234	-0.2187	-0.3437	-0.1067	-0.0187	-0.0656	-0.1123	-0.1829	-0.2086	-0.0697	-0.1608	-0.1991
c3	0.3639	0.5676	0.5036	0.388	0.6005	0.5499	0.3454	0.4198	0.215	0.401	0.4672	0.296
c4	0.224	0.3969	0.4591	0.2245	0.2616	0.1697	0.4034	0.6006	0.5899	0.3955	0.6311	0.6159
c5	0.2713	0.4277	0.4313	0.2505	0.2283	0.1391	0.4916	0.677	0.6154	0.4537	0.6705	0.6481
c6	0.3619	0.6232	0.6374	0.2859	0.329	0.4078	0.2003	0.2725	0.1912	0.1881	0.2359	0.2106
d1	1.0	0.6777	0.3055	0.9514	0.5344	0.3975	0.3999	0.3621	0.0362	0.5342	0.3086	0.0632
d2	0.6777	1.0	0.7705	0.638	0.7594	0.6231	0.261	0.381	0.1719	0.283	0.3618	0.1892
d3	0.3055	0.7705	1.0	0.2853	0.5303	0.719	0.1361	0.2866	0.2833	0.1369	0.2763	0.283
d4	0.9514	0.638	0.2853	1.0	0.6366	0.4944	0.3994	0.303	-0.0147	0.5484	0.304	0.025
d5	0.5344	0.7594	0.5303	0.6366	1.0	0.7879	0.2431	0.2697	-0.0572	0.3067	0.344	-0.0096
d6	0.3975	0.6231	0.719	0.4944	0.7879	1.0	0.1038	0.106	-0.0451	0.1792	0.1399	-0.0221
d7	0.3999	0.261	0.1361	0.3994	0.2431	0.1038	1.0	0.8232	0.5985	0.9303	0.7461	0.6523
d8	0.3621	0.381	0.2866	0.303	0.2697	0.106	0.8232	1.0	0.7509	0.7416	0.942	0.7912
d9	0.0362	0.1719	0.2833	-0.0147	-0.0572	-0.0451	0.5985	0.7509	1.0	0.5006	0.6472	0.981
d10	0.5342	0.283	0.1369	0.5484	0.3067	0.1792	0.9303	0.7416	0.5006	1.0	0.7148	0.5804
d11	0.3086	0.3618	0.2763	0.304	0.344	0.1399	0.7461	0.942	0.6472	0.7148	1.0	0.723
d12	0.0632	0.1892	0.283	0.025	-0.0096	-0.0221	0.6523	0.7912	0.981	0.5804	0.723	1.0
m1	0.2724	0.3888	0.2668	0.2647	0.4189	0.2802	0.2813	0.3023	0.212	0.3555	0.3302	0.2724
m2	0.2201	0.3172	0.5507	0.2347	0.2791	0.5338	0.0363	0.0156	-0.0346	0.0977	0.007	-0.0285
m3	0.9061	0.6406	0.3589	0.903	0.5361	0.4473	0.3507	0.3071	0.0526	0.541	0.3054	0.1
p1	0.8954	0.7007	0.3331	0.9382	0.6512	0.4407	0.3363	0.2987	0.0057	0.4947	0.3425	0.0581
p2	0.8698	0.6681	0.3086	0.9071	0.6358	0.4126	0.3482	0.3127	0.0157	0.503	0.3731	0.0726
p3	0.6603	0.7316	0.5339	0.6741	0.6435	0.5075	0.2255	0.3051	0.0937	0.3671	0.3819	0.1584
p4	0.6086	0.4085	0.1406	0.572	0.2499	0.1602	0.0769	0.0776	-0.0164	0.1004	0.1014	-0.0153
p5	0.5458	0.3424	0.1086	0.5227	0.2601	0.1558	0.0631	0.0157	-0.0517	0.2797	0.0201	-0.0137
p6	0.8618	0.6198	0.2709	0.9039	0.5984	0.399	0.3046	0.2586	-0.0059	0.4951	0.3233	0.0551
p7	0.2451	0.2131	0.0776	0.4318	0.4911	0.2461	0.3957	0.252	0.0106	0.3987	0.3511	0.0545
r1	0.2569	0.1058	0.0457	0.2394	0.1536	0.1282	0.0994	0.1881	0.0225	0.2986	0.2603	0.0899
r2												
r3	0.2569	0.1058	0.0457	0.2394	0.1536	0.1282	0.0994	0.1881	0.0225	0.2986	0.2603	0.0899
r4	0.2569	0.1058	0.0457	0.2394	0.1536	0.1282	0.0994	0.1881	0.0225	0.2986	0.2603	0.0899
r5	0.2569	0.1058	0.0457	0.2394	0.1536	0.1282	0.0994	0.1881	0.0225	0.2986	0.2603	0.0899
r6	0.2569	0.1058	0.0457	0.2394	0.1536	0.1282	0.0994	0.1881	0.0225	0.2986	0.2603	0.0899

Table 56: Correlation between distance metrics and others for the Haskell Cryptographic Library.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	-0.0198	0.0545	-0.0312	-0.0366	-0.02	-0.0465	-0.0242	-0.0325	-0.0335	-0.0179	-0.0256	-0.0285
c2	0.0065	0.0765	0.0158	0.0052	0.0428	0.0137	0.0127	-0.0478	-0.0471	0.0392	-0.0194	-0.0259
c3	0.3385	0.6575	0.6593	0.3361	0.3414	0.3101	0.2442	0.1931	0.0876	0.2418	0.1659	0.1492
c4	0.4904	0.5458	0.3442	0.363	0.1716	0.0879	0.2953	0.2768	0.3003	0.358	0.2465	0.3557
c5	0.3351	0.3741	0.245	0.2757	0.149	0.0932	0.2537	0.2378	0.2828	0.2787	0.2182	0.3242
c6	0.3686	0.6426	0.7273	0.2964	0.2475	0.2729	0.1527	0.1302	0.1243	0.2127	0.1467	0.1951
d1	1.0	0.6085	0.3393	0.7773	0.1913	0.1342	0.3502	0.227	0.0188	0.6449	0.1837	0.1675
d2	0.6085	1.0	0.7578	0.4741	0.3846	0.2944	0.2858	0.234	0.075	0.3543	0.1947	0.1608
d3	0.3393	0.7578	1.0	0.3082	0.3484	0.4155	0.2035	0.2147	0.1403	0.2047	0.1731	0.1865
d4	0.7773	0.4741	0.3082	1.0	0.6449	0.5937	0.6162	0.5161	0.2888	0.7675	0.5073	0.4167
d5	0.1913	0.3846	0.3484	0.6449	1.0	0.9538	0.6423	0.6387	0.4791	0.5641	0.7115	0.5326
d6	0.1342	0.2944	0.4155	0.5937	0.9538	1.0	0.5805	0.5898	0.4611	0.5033	0.657	0.5053
d7	0.3502	0.2858	0.2035	0.6162	0.6423	0.5805	1.0	0.872	0.7078	0.8825	0.8877	0.7396
d8	0.227	0.234	0.2147	0.5161	0.6387	0.5898	0.872	1.0	0.7494	0.7104	0.9236	0.7604
d9	0.0188	0.075	0.1403	0.2888	0.4791	0.4611	0.7078	0.7494	1.0	0.559	0.824	0.9633
d10	0.6449	0.3543	0.2047	0.7675	0.5641	0.5033	0.8825	0.7104	0.559	1.0	0.7745	0.6707
d11	0.1837	0.1947	0.1731	0.5073	0.7115	0.657	0.8877	0.9236	0.824	0.7745	1.0	0.8724
d12	0.1675	0.1608	0.1865	0.4167	0.5326	0.5053	0.7396	0.7604	0.9633	0.6707	0.8724	1.0
m1	0.1477	0.3475	0.311	0.4793	0.6976	0.6478	0.5669	0.4824	0.4185	0.5124	0.5732	0.4714
m2	0.9569	0.6302	0.3654	0.808	0.2974	0.2305	0.4187	0.2773	0.076	0.6937	0.2603	0.2249
m3	0.7484	0.5335	0.3222	0.5999	0.2304	0.1725	0.331	0.2188	0.0569	0.5607	0.2199	0.1789
p1	0.8426	0.7103	0.3301	0.5462	0.086	-0.0305	0.1974	0.0672	-0.0997	0.424	0.021	0.0145
p2	0.7671	0.6707	0.2714	0.4863	0.078	-0.0562	0.1567	0.0275	-0.1151	0.381	-0.0055	-0.0081
p3	0.3048	0.5991	0.3365	0.1968	0.1529	0.0389	0.0255	-0.0336	-0.0496	0.1227	-0.0116	0.0163
p4	0.1241	0.1747	0.0118	0.0632	0.0101	-0.0394	0.0184	-0.0306	-0.0511	0.051	-0.0117	-0.0331
p5	-0.0335	-0.0203	0.0137	-0.0344	-0.0281	-0.0176	-0.0319	-0.0334	-0.0315	-0.0325	-0.0325	-0.0336
p6	0.7222	0.663	0.2598	0.4488	0.0712	-0.066	0.1348	0.0066	-0.1238	0.3508	-0.0193	-0.0198
p7	0.0876	0.2369	0.0296	0.0222	0.0543	-0.064	-0.0369	-0.0836	-0.0891	0.0135	-0.0638	-0.0574
r1	0.0124	0.1227	0.0384	0.3552	0.6219	0.5358	0.3905	0.366	0.2926	0.3189	0.4513	0.3196
r2												
r3	0.0124	0.1227	0.0384	0.3552	0.6219	0.5358	0.3905	0.366	0.2926	0.3189	0.4513	0.3196
r4	0.0124	0.1227	0.0384	0.3552	0.6219	0.5358	0.3905	0.366	0.2926	0.3189	0.4513	0.3196
r5	0.0124	0.1227	0.0384	0.3552	0.6219	0.5358	0.3905	0.366	0.2926	0.3189	0.4513	0.3196
r6	0.0124	0.1227	0.0384	0.3552	0.6219	0.5358	0.3905	0.366	0.2926	0.3189	0.4513	0.3196

Table 57: Correlation between distance metrics and others for the Haskell DSP Library.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.3304	0.1595	-0.0112	0.3506	0.1255	0.122	0.0813	0.0401	-0.0744	0.1119	0.0513	-0.079
c2	-0.0102	-0.1309	-0.1693	-0.0194	-0.0948	-0.0893	-0.0496	-0.0677	-0.081	-0.0428	-0.0742	-0.0934
c3	0.7628	0.7388	0.3989	0.7444	0.4553	0.2202	0.3432	0.2661	0.0082	0.468	0.3324	0.0469
c4	0.3344	0.4537	0.3046	0.2785	0.2215	0.0833	0.4398	0.4941	0.3225	0.4376	0.5097	0.3139
c5	0.3815	0.3841	0.1964	0.2983	0.0987	-0.0276	0.4136	0.4222	0.1507	0.4492	0.4633	0.1614
c6	0.359	0.6594	0.6666	0.3645	0.4629	0.441	0.2893	0.3261	0.2702	0.2884	0.3393	0.2905
d1	1.0	0.6889	0.274	0.8309	0.2759	0.081	0.462	0.3111	-0.0644	0.5819	0.3493	-0.052
d2	0.6889	1.0	0.7237	0.6102	0.5944	0.4082	0.3823	0.3556	0.1327	0.4164	0.3726	0.1577
d3	0.274	0.7237	1.0	0.3713	0.6115	0.7211	0.2133	0.2375	0.2371	0.2032	0.2415	0.2669
d4	0.8309	0.6102	0.3713	1.0	0.5766	0.4247	0.38	0.2656	-0.0189	0.4931	0.3064	0.0069
d5	0.2759	0.5944	0.6115	0.5766	1.0	0.802	0.2092	0.2195	0.2072	0.2568	0.2584	0.2745
d6	0.081	0.4082	0.7211	0.4247	0.802	1.0	0.0795	0.1022	0.1702	0.0687	0.1016	0.1853
d7	0.462	0.3823	0.2133	0.38	0.2092	0.0795	1.0	0.8901	0.6259	0.9394	0.8755	0.5851
d8	0.3111	0.3556	0.2375	0.2656	0.2195	0.1022	0.8901	1.0	0.7484	0.8042	0.9565	0.6888
d9	-0.0644	0.1327	0.2371	-0.0189	0.2072	0.1702	0.6259	0.7484	1.0	0.5123	0.7051	0.9524
d10	0.5819	0.4164	0.2032	0.4931	0.2568	0.0687	0.9394	0.8042	0.5123	1.0	0.8669	0.5337
d11	0.3493	0.3726	0.2415	0.3064	0.2584	0.1016	0.8755	0.9565	0.7051	0.8669	1.0	0.7136
d12	-0.052	0.1577	0.2669	0.0069	0.2745	0.1853	0.5851	0.6888	0.9524	0.5337	0.7136	1.0
m1	0.633	0.4159	0.0898	0.564	0.2139	0.0301	0.2244	0.1069	-0.0893	0.3664	0.1748	-0.0718
m2	0.6522	0.4975	0.2577	0.5767	0.3115	0.1945	0.248	0.1529	-0.0299	0.3764	0.2146	0.0298
m3	0.5347	0.4459	0.244	0.475	0.2908	0.1906	0.2151	0.1294	-0.0198	0.317	0.1807	0.0375
p1	0.8069	0.6399	0.1561	0.6504	0.246	0.0012	0.2655	0.1839	-0.118	0.3665	0.2111	-0.1214
p2	0.7938	0.6143	0.1337	0.6484	0.2341	-0.0144	0.2526	0.1603	-0.1331	0.359	0.1988	-0.1315
p3	0.5919	0.5759	0.2777	0.5591	0.3006	0.1009	0.1948	0.1267	-0.0786	0.2977	0.1813	-0.0639
p4	0.197	0.1569	0.0717	0.1161	0.0257	-0.0038	0.1278	0.0426	-0.0291	0.1094	0.0359	-0.0295
p5	0.5159	0.2988	-0.0128	0.4489	0.131	-0.0624	0.1524	0.0687	-0.1313	0.2678	0.1353	-0.1158
p6	0.7575	0.5808	0.1143	0.63	0.2252	-0.0182	0.2096	0.1235	-0.1472	0.3143	0.1598	-0.1487
p7	0.1917	0.1867	0.0585	0.2121	0.1117	0.0113	-0.0294	-0.0528	-0.1227	-0.0064	-0.0465	-0.128
r1	0.4025	0.269	0.0007	0.3366	0.0984	-0.0345	0.1099	0.0396	-0.0988	0.2037	0.08	-0.0872
r2												
r3	0.4025	0.269	0.0007	0.3366	0.0984	-0.0345	0.1099	0.0396	-0.0988	0.2037	0.08	-0.0872
r4	0.4025	0.269	0.0007	0.3366	0.0984	-0.0345	0.1099	0.0396	-0.0988	0.2037	0.08	-0.0872
r5	0.4025	0.269	0.0007	0.3366	0.0984	-0.0345	0.1099	0.0396	-0.0988	0.2037	0.08	-0.0872
r6	0.4025	0.269	0.0007	0.3366	0.0984	-0.0345	0.1099	0.0396	-0.0988	0.2037	0.08	-0.0872

Table 58: Correlation between distance metrics and others for FGL.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.0858	0.0428	0.0054	0.0622	0.0087	-0.0248	0.0417	0.0262	-0.0243	0.0689	0.0385	-0.0153
c2	-0.0427	-0.1432	-0.157	-0.0324	-0.0905	-0.0935	-0.0124	-0.04	-0.0204	-0.0246	-0.0498	-0.0352
c3	0.4724	0.7097	0.5377	0.3941	0.5146	0.3764	0.3947	0.4295	0.2227	0.4018	0.4386	0.2491
c4	0.3317	0.5699	0.4862	0.2454	0.3028	0.2058	0.3453	0.456	0.3161	0.3309	0.4549	0.3254
c5	0.3584	0.4562	0.3465	0.2328	0.1985	0.1205	0.4002	0.4705	0.2989	0.3809	0.4598	0.3057
c6	0.3694	0.6997	0.6577	0.3266	0.4707	0.4195	0.2288	0.284	0.1944	0.2543	0.3057	0.2306
d1	1.0	0.6313	0.3047	0.9053	0.3269	0.1593	0.7036	0.3639	0.0782	0.8207	0.4052	0.1171
d2	0.6313	1.0	0.7607	0.5086	0.6252	0.4654	0.4221	0.4173	0.1759	0.4625	0.4552	0.2138
d3	0.3047	0.7607	1.0	0.2527	0.5175	0.6383	0.1812	0.2354	0.2202	0.1975	0.2589	0.2419
d4	0.9053	0.5086	0.2527	1.0	0.4381	0.2899	0.707	0.3173	0.0857	0.803	0.3497	0.1141
d5	0.3269	0.6252	0.5175	0.4381	1.0	0.824	0.3211	0.3373	0.2589	0.3166	0.3544	0.2869
d6	0.1593	0.4654	0.6383	0.2899	0.824	1.0	0.1262	0.1821	0.2136	0.1272	0.2059	0.241
d7	0.7036	0.4221	0.1812	0.707	0.3211	0.1262	1.0	0.71	0.3906	0.9521	0.6924	0.3834
d8	0.3639	0.4173	0.2354	0.3173	0.3373	0.1821	0.71	1.0	0.7001	0.6362	0.9765	0.6912
d9	0.0782	0.1759	0.2202	0.0857	0.2589	0.2136	0.3906	0.7001	1.0	0.3212	0.6666	0.9699
d10	0.8207	0.4625	0.1975	0.803	0.3166	0.1272	0.9521	0.6362	0.3212	1.0	0.6579	0.3485
d11	0.4052	0.4552	0.2589	0.3497	0.3544	0.2059	0.6924	0.9765	0.6666	0.6579	1.0	0.6944
d12	0.1171	0.2138	0.2419	0.1141	0.2869	0.241	0.3834	0.6912	0.9699	0.3485	0.6944	1.0
m1	0.231	0.2336	0.1136	0.229	0.2503	0.0984	0.3109	0.2969	0.1445	0.2557	0.2678	0.1319
m2	0.8834	0.5489	0.2255	0.8347	0.3079	0.1123	0.6837	0.3562	0.0663	0.7709	0.3813	0.0959
m3	0.7863	0.5159	0.2158	0.7403	0.2503	0.0709	0.6283	0.3333	0.0437	0.6894	0.3591	0.0604
p1	0.842	0.6609	0.2697	0.778	0.4002	0.1407	0.7078	0.4363	0.0871	0.7482	0.4547	0.1055
p2	0.7936	0.6417	0.2498	0.7232	0.3842	0.133	0.6399	0.4005	0.0669	0.6886	0.424	0.0875
p3	0.4157	0.5908	0.3736	0.367	0.3914	0.2376	0.312	0.2888	0.093	0.3268	0.3	0.1103
p4	0.4916	0.2701	0.0298	0.4758	0.142	0.0016	0.38	0.1642	0.0145	0.4219	0.1813	0.0169
p5	0.2841	0.2293	0.0742	0.2729	0.1768	0.0297	0.3196	0.2334	0.0833	0.3052	0.2038	0.0657
p6	0.7637	0.5986	0.2233	0.7092	0.3651	0.1174	0.6583	0.402	0.0743	0.6888	0.4166	0.0891
p7	0.3289	0.2798	0.1005	0.2635	0.1195	0.0035	0.347	0.2413	0.061	0.3033	0.2374	0.0588
r1	0.1053	0.0692	-0.0255	0.0837	0.0512	-0.0517	0.1778	0.222	0.1321	0.1405	0.1543	0.0907
r2	-0.0134	-0.0202	-0.0203	-0.0161	-0.0342	-0.035	-0.0146	-0.0198	-0.018	-0.0136	-0.0185	-0.0179
r3	0.1097	0.0743	-0.022	0.0882	0.0587	-0.0458	0.1836	0.2296	0.1378	0.1455	0.1605	0.0956
r4	0.1053	0.0692	-0.0255	0.0837	0.0512	-0.0517	0.1778	0.222	0.1321	0.1405	0.1543	0.0907
r5	0.0977	0.0622	-0.028	0.0766	0.0424	-0.0556	0.1664	0.2076	0.1224	0.1312	0.1434	0.083
r6	0.0977	0.0622	-0.028	0.0766	0.0424	-0.0556	0.1664	0.2076	0.1224	0.1312	0.1434	0.083

Table 59: Correlation between distance metrics and others for the Library of Geometric Algorithms.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.2457	0.2303	-0.0478	0.0975	0.2709	-0.1339	0.5799	0.0007	0.1698	0.496	-0.0115	0.2354
c2	-0.2887	-0.3545	-0.3835	-0.3103	-0.3757	-0.4127	-0.1061	-0.099	-0.0341	-0.1489	-0.1266	-0.0858
c3	0.5133	0.7433	0.5423	0.4133	0.8024	0.4983	0.4564	0.4684	0.2293	0.5558	0.5803	0.424
c4	0.643	0.6637	0.1962	0.4336	0.6831	0.0722	0.8633	0.5024	0.38	0.9065	0.5874	0.5436
c5	0.6031	0.5976	0.1222	0.3935	0.6293	0.0345	0.8234	0.4735	0.3108	0.8868	0.5826	0.4784
c6	0.7105	0.9151	0.6952	0.6226	0.9491	0.6028	0.621	0.5087	0.3684	0.6807	0.5736	0.544
d1	1.0	0.8352	0.2617	0.9409	0.8219	0.2488	0.6309	0.2168	0.1015	0.6999	0.252	0.2428
d2	0.8352	1.0	0.6325	0.7383	0.9809	0.4875	0.6614	0.4719	0.3687	0.6884	0.4675	0.5276
d3	0.2617	0.6325	1.0	0.3502	0.6425	0.8847	0.103	0.4325	0.4867	0.0944	0.4171	0.5361
d4	0.9409	0.7383	0.3502	1.0	0.7355	0.4044	0.3831	0.1092	0.0439	0.4603	0.1488	0.1461
d5	0.8219	0.9809	0.6425	0.7355	1.0	0.5675	0.6257	0.3717	0.2542	0.6685	0.3964	0.4293
d6	0.2488	0.4875	0.8847	0.4044	0.5675	1.0	-0.1016	0.0726	0.0827	-0.0567	0.1203	0.1391
d7	0.6309	0.6614	0.103	0.3831	0.6257	-0.1016	1.0	0.5704	0.4684	0.979	0.5594	0.5795
d8	0.2168	0.4719	0.4325	0.1092	0.3717	0.0726	0.5704	1.0	0.9112	0.5516	0.9733	0.9363
d9	0.1015	0.3687	0.4867	0.0439	0.2542	0.0827	0.4684	0.9112	1.0	0.3864	0.8326	0.9735
d10	0.6999	0.6884	0.0944	0.4603	0.6685	-0.0567	0.979	0.5516	0.3864	1.0	0.5815	0.5207
d11	0.252	0.4675	0.4171	0.1488	0.3964	0.1203	0.5594	0.9733	0.8326	0.5815	1.0	0.8829
d12	0.2428	0.5276	0.5361	0.1461	0.4293	0.1391	0.5795	0.9363	0.9735	0.5207	0.8829	1.0
m1	-0.0046	0.0338	0.2	0.0083	0.1728	0.33	0.0065	0.0375	-0.0984	0.1052	0.2035	-0.0031
m2	0.8678	0.7847	0.1281	0.6671	0.7898	0.086	0.8119	0.2523	0.065	0.8738	0.2981	0.2405
m3	0.7401	0.6624	0.0068	0.5102	0.6655	-0.0191	0.7943	0.2882	0.0205	0.8717	0.3455	0.1833
p1	0.8275	0.7397	0.0734	0.6191	0.7317	0.0336	0.7688	0.2854	0.0441	0.8495	0.3385	0.2154
p2	0.716	0.6439	-0.0021	0.4749	0.6509	-0.0358	0.8132	0.2798	0.0298	0.8817	0.3348	0.1921
p3	0.5496	0.6015	0.2687	0.4229	0.694	0.3201	0.4921	0.2506	-0.034	0.6145	0.3924	0.1634
p4												
p5	0.5531	0.5298	0.0658	0.3515	0.5655	-0.0115	0.7484	0.0689	0.0572	0.714	0.0541	0.1786
p6	0.6993	0.6284	-0.0128	0.4597	0.6341	-0.0403	0.7821	0.2809	0.0111	0.8597	0.3414	0.1742
p7	0.61	0.5507	-0.0407	0.3815	0.5299	-0.0792	0.6858	0.2469	-0.0193	0.7484	0.2693	0.1087
r1												
r2												
r3												
r4												
r5												
r6												

Table 60: Correlation between distance metrics and others for GetOpt.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	-0.0759	-0.2338	0.2218	-0.1644	-0.2402	-0.2359	0.0476	0.3509	0.2754	0.0452	0.344	0.2768
c2	-0.0048	-0.0406	-0.1056	0.0396	0.0538	0.0286	-0.0165	-0.0932	-0.0814	-0.0183	-0.0946	-0.0821
c3	0.5494	0.3986	0.1454	0.2746	0.1116	0.0382	0.8434	0.0652	0.0114	0.8528	0.0749	0.0103
c4	0.0621	-0.0823	0.2929	-0.0774	-0.2688	-0.2928	0.0534	0.3702	0.2686	0.0581	0.3744	0.2751
c5	-0.0186	-0.1902	0.2536	-0.1364	-0.2417	-0.2423	0.0503	0.3582	0.2691	0.0494	0.3536	0.2712
c6	0.2513	0.4096	0.5308	0.1498	-0.1401	-0.1821	0.0538	0.2784	0.1867	0.0669	0.3012	0.2002
d1	1.0	0.5266	0.3102	0.5753	-0.0039	-0.0406	0.2865	-0.0135	-0.0613	0.3131	0.0131	-0.0556
d2	0.5266	1.0	0.5107	0.5383	0.4231	0.2887	0.0733	0.0624	-0.0224	0.0931	0.0866	-0.0176
d3	0.3102	0.5107	1.0	0.2746	0.2023	0.2789	0.0242	0.0181	0.0514	0.0267	0.0299	0.0606
d4	0.5753	0.5383	0.2746	1.0	0.5438	0.3976	0.0911	0.0173	-0.046	0.1123	0.0402	-0.0403
d5	-0.0039	0.4231	0.2023	0.5438	1.0	0.893	0.0624	-0.0722	-0.0576	0.0633	-0.0719	-0.0618
d6	-0.0406	0.2887	0.2789	0.3976	0.893	1.0	0.0205	-0.2067	-0.1267	0.0143	-0.211	-0.1302
d7	0.2865	0.0733	0.0242	0.0911	0.0624	0.0205	1.0	0.1908	0.1664	0.9984	0.1843	0.1581
d8	-0.0135	0.0624	0.0181	0.0173	-0.0722	-0.2067	0.1908	1.0	0.9196	0.2076	0.9967	0.9157
d9	-0.0613	-0.0224	0.0514	-0.046	-0.0576	-0.1267	0.1664	0.9196	1.0	0.179	0.9151	0.9985
d10	0.3131	0.0931	0.0267	0.1123	0.0633	0.0143	0.9984	0.2076	0.179	1.0	0.2032	0.1717
d11	0.0131	0.0866	0.0299	0.0402	-0.0719	-0.211	0.1843	0.9967	0.9151	0.2032	1.0	0.9145
d12	-0.0556	-0.0176	0.0606	-0.0403	-0.0618	-0.1302	0.1581	0.9157	0.9985	0.1717	0.9145	1.0
m1	0.5105	0.2682	0.2653	0.1444	-0.0092	-0.0096	0.0064	-0.0055	-0.0197	0.0166	0.0035	-0.0172
m2	0.8121	0.4101	0.1144	0.3915	-0.0142	-0.0608	0.582	0.0104	-0.039	0.6093	0.0311	-0.0369
m3	0.6994	0.3112	0.0979	0.3266	-0.003	-0.0373	0.7337	-0.0021	-0.035	0.7541	0.012	-0.0359
p1	0.7271	0.5894	0.0929	0.3858	-0.0124	-0.1218	0.0474	0.0246	-0.0478	0.0824	0.0524	-0.0423
p2	0.6671	0.4984	0.0668	0.3477	-0.0085	-0.0971	0.0441	0.0134	-0.0502	0.0833	0.0399	-0.0444
p3	0.3001	0.5383	0.0389	0.2943	0.1152	-0.1034	0.0625	0.1159	0.028	0.081	0.1368	0.0316
p4	0.2009	0.1498	-0.0122	0.0339	-0.0323	-0.033	-0.0022	-0.036	-0.048	0.0075	-0.0183	-0.0464
p5	0.2849	0.2284	0.0276	0.1813	0.0224	-0.0173	0.0172	0.0418	0.0019	0.0307	0.0542	0.0067
p6	0.6326	0.4718	0.0476	0.316	-0.0134	-0.098	0.0452	-0.0016	-0.0591	0.087	0.0248	-0.0532
p7	0.2055	0.1586	0.0011	0.0957	-0.0412	-0.0507	-0.0052	-0.0696	-0.0782	0.0025	-0.0601	-0.0761
r1	-0.0309	-0.1547	0.2041	-0.168	-0.2673	-0.2671	0.0324	0.2542	0.183	0.0305	0.2525	0.1855
r2	0.206	0.0647	0.0321	0.2272	0.088	0.0239	0.6352	0.0786	0.0198	0.6317	0.0706	0.0143
r3	0.1185	0.1598	-0.0239	0.0812	-0.0312	-0.0621	-0.0081	-0.075	-0.1054	-0.0057	-0.0676	-0.1037
r4	0.2066	0.0657	0.0319	0.2275	0.0877	0.0235	0.6344	0.078	0.0191	0.631	0.07	0.0136
r5	0.2062	0.065	0.032	0.2273	0.0879	0.0238	0.6349	0.0784	0.0195	0.6314	0.0704	0.0141
r6	0.0172	0.0582	-0.0172	0.0807	0.0716	-0.0094	-0.0004	0.0013	-0.0231	-0.0001	0.0001	-0.0233

Table 61: Correlation between distance metrics and others for Haddock.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.0061	0.0822	-0.0252	-0.0043	0.0355	-0.0245	-0.0065	-0.0001	-0.0353	-0.0047	0.0194	-0.0322
c2	-0.0261	-0.0443	-0.036	-0.0189	0.0917	0.1216	-0.0211	-0.0954	-0.0876	-0.0215	-0.1009	-0.0875
c3	0.3807	0.6449	0.3029	0.2826	0.517	0.2364	0.2347	0.2347	0.0043	0.2513	0.2993	0.0219
c4	0.2589	0.2488	0.2161	0.1809	0.0546	-0.0092	0.1823	0.5929	0.5118	0.185	0.6032	0.5074
c5	0.3697	0.33	0.1394	0.2175	0.0829	0.0569	0.1879	0.3845	0.2902	0.1937	0.429	0.3059
c6	0.2008	0.5027	0.5411	0.1603	0.3295	0.3044	0.1475	0.3028	0.2078	0.1497	0.3282	0.2194
d1	1.0	0.4551	0.1821	0.9491	0.2608	0.23	0.9064	0.2891	0.1241	0.914	0.3281	0.1383
d2	0.4551	1.0	0.5953	0.2656	0.6281	0.4195	0.1807	0.1286	-0.117	0.1935	0.2106	-0.0928
d3	0.1821	0.5953	1.0	0.1418	0.4165	0.6198	0.123	0.1816	0.122	0.125	0.1921	0.1312
d4	0.9491	0.2656	0.1418	1.0	0.2171	0.2236	0.9803	0.2612	0.1488	0.9829	0.2829	0.1597
d5	0.2608	0.6281	0.4165	0.2171	1.0	0.6618	0.1691	0.1143	-0.042	0.1743	0.1567	-0.0233
d6	0.23	0.4195	0.6198	0.2336	0.6618	1.0	0.2015	0.1284	0.11	0.2031	0.1435	0.121
d7	0.9064	0.1807	0.123	0.9803	0.1691	0.2015	1.0	0.2962	0.2077	0.9997	0.3081	0.218
d8	0.2891	0.1286	0.1816	0.2612	0.1143	0.1284	0.2962	1.0	0.8401	0.2957	0.9834	0.8378
d9	0.1241	-0.117	0.122	0.1488	-0.042	0.11	0.2077	0.8401	1.0	0.2034	0.7928	0.9964
d10	0.914	0.1935	0.125	0.9829	0.1743	0.2031	0.9997	0.2957	0.2034	1.0	0.3095	0.2141
d11	0.3281	0.2106	0.1921	0.2829	0.1567	0.1435	0.3081	0.9834	0.7928	0.3095	1.0	0.7994
d12	0.1383	-0.0928	0.1312	0.1597	-0.0233	0.121	0.218	0.8378	0.9964	0.2141	0.7994	1.0
m1	0.2043	0.2157	0.0363	0.0856	0.075	0.0158	0.0309	0.0689	-0.0176	0.0483	0.0823	-0.0106
m2	0.9679	0.4461	0.1726	0.9534	0.3092	0.2383	0.9198	0.2928	0.1058	0.9267	0.3333	0.1209
m3	0.8692	0.5824	0.167	0.7766	0.3718	0.2056	0.722	0.3277	0.0715	0.7331	0.38	0.0892
p1	0.872	0.6128	0.2075	0.8015	0.4383	0.2487	0.744	0.2685	0.0044	0.7548	0.3257	0.0227
p2	0.7871	0.5889	0.1508	0.7082	0.4368	0.1987	0.6539	0.2971	0.0442	0.6646	0.3548	0.0615
p3	0.1649	0.3505	0.0724	0.1181	0.5108	0.0503	0.0892	0.193	0.0351	0.0939	0.2201	0.0451
p4	0.3542	0.2557	0.058	0.3568	0.1214	0.0907	0.329	0.0642	-0.005	0.3333	0.0935	0.0023
p5	0.3861	0.3248	0.0573	0.3236	0.317	0.1506	0.2989	0.2705	0.0779	0.3046	0.2829	0.0793
p6	0.7696	0.5821	0.1437	0.6888	0.4252	0.1905	0.6343	0.2948	0.0353	0.6449	0.3516	0.0524
p7	0.3499	0.2905	0.0385	0.3118	0.2477	0.0486	0.2642	0.2408	0.0442	0.2703	0.2548	0.053
r1	-0.0117	0.1416	-0.0309	-0.0168	0.1364	0.0144	-0.0231	-0.0712	-0.1187	-0.0215	-0.0413	-0.1146
r2												
r3	-0.0117	0.1416	-0.0309	-0.0168	0.1364	0.0144	-0.0231	-0.0712	-0.1187	-0.0215	-0.0413	-0.1146
r4	-0.0117	0.1416	-0.0309	-0.0168	0.1364	0.0144	-0.0231	-0.0712	-0.1187	-0.0215	-0.0413	-0.1146
r5	-0.0117	0.1416	-0.0309	-0.0168	0.1364	0.0144	-0.0231	-0.0712	-0.1187	-0.0215	-0.0413	-0.1146
r6	-0.0117	0.1416	-0.0309	-0.0168	0.1364	0.0144	-0.0231	-0.0712	-0.1187	-0.0215	-0.0413	-0.1146

Table 62: Correlation between distance metrics and others for Happy.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.1409	0.1616	0.0741	0.2066	0.3285	0.1782	0.2383	0.3337	0.1442	0.2166	0.3239	0.1591
c2	0.0139	-0.0549	-0.0821	0.0311	0.0282	0.0485	0.0372	-0.0146	-0.0541	0.0347	-0.012	-0.0477
c3	0.6746	0.5954	0.2758	0.6561	0.4401	0.2625	0.6455	0.4851	0.2127	0.6313	0.5031	0.261
c4	0.2623	0.4408	0.3322	0.3167	0.5411	0.3343	0.3	0.564	0.3991	0.2846	0.5665	0.4292
c5	0.2307	0.2949	0.1698	0.3184	0.5031	0.3079	0.3152	0.5379	0.2979	0.2963	0.5382	0.3295
c6	0.2332	0.616	0.6447	0.2567	0.4672	0.3881	0.189	0.3816	0.3681	0.1824	0.3774	0.3679
d1	1.0	0.4596	0.148	0.7583	0.2039	0.1053	0.6516	0.3143	0.105	0.6712	0.3359	0.1362
d2	0.4596	1.0	0.7282	0.3385	0.5179	0.3816	0.2274	0.3536	0.2406	0.2283	0.3617	0.2577
d3	0.148	0.7282	1.0	0.1405	0.4328	0.506	0.0608	0.2288	0.3636	0.0543	0.2141	0.333
d4	0.7583	0.3385	0.1405	1.0	0.4091	0.3127	0.7857	0.4194	0.2198	0.7799	0.4283	0.2504
d5	0.2039	0.5179	0.4328	0.4091	1.0	0.829	0.2778	0.4998	0.4209	0.2644	0.4994	0.4432
d6	0.1053	0.3816	0.506	0.3127	0.829	1.0	0.1499	0.3046	0.3039	0.1368	0.2968	0.3046
d7	0.6516	0.2274	0.0608	0.7857	0.2778	0.1499	1.0	0.4665	0.2846	0.9928	0.4744	0.3246
d8	0.3143	0.3536	0.2288	0.4194	0.4998	0.3046	0.4665	1.0	0.7221	0.4418	0.99	0.7431
d9	0.105	0.2406	0.3636	0.2198	0.4209	0.3039	0.2846	0.7221	1.0	0.2764	0.701	0.9782
d10	0.6712	0.2283	0.0543	0.7799	0.2644	0.1368	0.9928	0.4418	0.2764	1.0	0.4554	0.322
d11	0.3359	0.3617	0.2141	0.4283	0.4994	0.2968	0.4744	0.99	0.701	0.4554	1.0	0.7409
d12	0.1362	0.2577	0.333	0.2504	0.4432	0.3046	0.3246	0.7431	0.9782	0.322	0.7409	1.0
m1	0.5327	0.212	0.0127	0.3273	0.0469	0.0047	0.3018	0.12	0.0092	0.3365	0.1422	0.033
m2	0.8749	0.3746	0.0571	0.6586	0.1605	0.0632	0.6315	0.2877	0.0618	0.6559	0.3155	0.095
m3	0.8068	0.35	0.0382	0.5815	0.1221	0.0392	0.5298	0.2577	0.0115	0.5452	0.2801	0.0387
p1	0.7437	0.3774	0.0703	0.5151	0.1259	0.0382	0.5142	0.244	0.0346	0.5432	0.2735	0.0628
p2	0.7324	0.3471	0.0485	0.4867	0.1021	0.0246	0.4598	0.2084	0.0114	0.4959	0.2398	0.0388
p3	0.4142	0.5465	0.2295	0.2913	0.2276	0.0826	0.2627	0.2098	0.0254	0.2724	0.2292	0.0552
p4	0.3185	0.2147	0.0078	0.1914	0.0263	-0.0116	0.1268	0.0803	-0.0223	0.1299	0.0922	-0.015
p5	0.6117	0.2936	0.0185	0.5268	0.11	0.0177	0.5275	0.1934	0.0018	0.5364	0.2049	0.0305
p6	0.7151	0.3373	0.0434	0.4836	0.0997	0.023	0.4639	0.2064	0.0065	0.4987	0.2369	0.0336
p7	0.5232	0.2243	0.0225	0.4119	0.057	0.0141	0.3599	0.1222	-0.0298	0.3701	0.1303	-0.0112
r1	0.1678	0.1846	0.0422	0.1546	0.262	0.1188	0.1566	0.1386	0.1226	0.1644	0.1548	0.1568
r2	0.0205	0.0599	0.0463	0.076	0.2442	0.1001	0.0708	0.1384	0.2399	0.0691	0.146	0.2651
r3	0.1846	0.1663	0.0039	0.13	0.1145	0.0307	0.1374	0.0533	-0.0403	0.1477	0.0662	-0.0193
r4	0.0481	0.0845	0.0465	0.095	0.2596	0.104	0.091	0.1454	0.232	0.0909	0.1549	0.2603
r5	0.0283	0.0646	0.0469	0.085	0.2515	0.1011	0.0746	0.1348	0.2316	0.0728	0.1423	0.2552
r6	0.0096	0.0331	0.0157	0.0407	0.1449	0.0669	0.0395	0.0742	0.1405	0.0361	0.0761	0.1492

Table 63: Correlation between distance metrics and others for Hat.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.1476	0.1556	0.0497	0.1222	0.0523	-0.003	0.0942	0.0558	0.0279	0.1488	0.0847	0.0329
c2	-0.032	-0.0852	-0.109	-0.0463	-0.0669	-0.0817	-0.0408	-0.114	-0.1012	-0.0445	-0.1151	-0.1062
c3	0.4874	0.7323	0.5443	0.5621	0.5442	0.411	0.3684	0.4056	0.2257	0.4694	0.5078	0.3081
c4	0.3068	0.5775	0.5288	0.3792	0.4589	0.3734	0.2907	0.559	0.3575	0.3297	0.5665	0.3796
c5	0.2818	0.4591	0.3979	0.3702	0.4216	0.3538	0.1976	0.3824	0.1428	0.2626	0.4621	0.2176
c6	0.3113	0.7999	0.7663	0.3953	0.5286	0.4749	0.2122	0.4468	0.3096	0.2659	0.5026	0.3664
d1	1.0	0.4663	0.2262	0.7429	0.2216	0.12	0.6354	0.2499	0.1025	0.6955	0.2759	0.1093
d2	0.4663	1.0	0.8052	0.5264	0.6659	0.5242	0.2798	0.4867	0.3136	0.3475	0.5471	0.3598
d3	0.2262	0.8052	1.0	0.4043	0.6376	0.6784	0.1893	0.545	0.4813	0.2332	0.5812	0.5275
d4	0.7429	0.5264	0.4043	1.0	0.6088	0.5622	0.603	0.4113	0.3139	0.6514	0.4577	0.3602
d5	0.2216	0.6659	0.6376	0.6088	1.0	0.8669	0.268	0.5974	0.4733	0.3312	0.6657	0.5472
d6	0.12	0.5242	0.6784	0.5622	0.8669	1.0	0.1744	0.578	0.5454	0.2284	0.6397	0.6207
d7	0.6354	0.2798	0.1893	0.603	0.268	0.1744	1.0	0.4192	0.3633	0.9493	0.3621	0.3198
d8	0.2499	0.4867	0.545	0.4113	0.5974	0.578	0.4192	1.0	0.8704	0.4577	0.9266	0.8471
d9	0.1025	0.3136	0.4813	0.3139	0.4733	0.5454	0.3633	0.8704	1.0	0.3747	0.7602	0.9411
d10	0.6955	0.3475	0.2332	0.6514	0.3312	0.2284	0.9493	0.4577	0.3747	1.0	0.4621	0.3855
d11	0.2759	0.5471	0.5812	0.4577	0.6657	0.6397	0.3621	0.9266	0.7602	0.4621	1.0	0.8516
d12	0.1093	0.3598	0.5275	0.3602	0.5472	0.6207	0.3198	0.8471	0.9411	0.3855	0.8516	1.0
m1	0.0547	0.0311	0.0169	0.0621	0.0309	0.0354	0.0037	-0.0275	-0.0267	0.0207	-0.0226	-0.0258
m2	0.8692	0.3148	0.0635	0.5726	0.1055	0.0141	0.4759	0.1324	0.013	0.57	0.17	0.0217
m3	0.8025	0.2849	0.0265	0.4879	0.056	-0.025	0.2855	0.0695	-0.0453	0.3883	0.1139	-0.03
p1	0.8313	0.3818	0.1083	0.479	0.0945	-0.0111	0.3004	0.0976	-0.0366	0.4	0.142	-0.0256
p2	0.8107	0.357	0.0818	0.4609	0.0733	-0.0224	0.2878	0.0981	-0.0279	0.4012	0.1346	-0.0232
p3	0.3436	0.5536	0.3214	0.3063	0.2562	0.148	0.2208	0.2549	0.1822	0.2243	0.2169	0.1408
p4	0.6235	0.1285	0.0165	0.3073	0.0005	-0.0184	0.0581	0.0423	-0.027	0.1154	0.069	-0.0185
p5	0.5388	0.3299	0.0483	0.3336	0.0806	-0.0254	0.2638	0.1622	0.0663	0.2913	0.1373	0.0226
p6	0.8254	0.3543	0.0807	0.4746	0.0713	-0.0219	0.3085	0.0975	-0.0255	0.4101	0.1291	-0.0248
p7	0.502	0.2469	0.0667	0.2319	-0.0169	-0.047	0.1083	0.0312	-0.0682	0.1532	0.0535	-0.0622
r1	0.1404	0.1775	0.0637	0.1738	0.118	0.0148	0.0568	0.0375	-0.0089	0.1061	0.0924	0.0238
r2	0.0345	0.0876	0.0474	0.0159	0.0332	-0.0131	0.0265	0.0352	0.0298	0.0271	0.0249	0.0246
r3	0.1299	0.1657	0.0636	0.1783	0.1239	0.0252	0.0464	0.0295	-0.0158	0.096	0.0871	0.0201
r4	0.1266	0.1837	0.0776	0.1576	0.1209	0.0144	0.0525	0.0428	0.0019	0.0944	0.0857	0.0294
r5	0.0854	0.1487	0.0729	0.0884	0.0837	0.0028	0.0455	0.0484	0.0269	0.0659	0.0574	0.0355
r6	0.0118	0.0562	0.0305	0.0042	0.029	-0.0089	0.0108	0.0263	0.0195	0.0095	0.0147	0.0139

Table 64: Correlation between distance metrics and others for HaXml.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1												
c2	-0.018	0.0606	0.1719	-0.0104	0.0738	0.0984	-0.0698	-0.0582	-0.0134	-0.0693	-0.0666	-0.0308
c3	0.6213	0.7993	0.6269	0.6088	0.6745	0.5332	0.4831	0.5413	0.2967	0.4814	0.5402	0.3469
c4	0.5564	0.6979	0.5539	0.4468	0.4204	0.263	0.551	0.6472	0.3999	0.5661	0.603	0.4165
c5	0.5515	0.6029	0.4465	0.4804	0.4728	0.3212	0.7601	0.7213	0.5411	0.7415	0.7074	0.5499
c6	0.3472	0.662	0.8029	0.3649	0.5925	0.5805	0.3337	0.3497	0.3023	0.3347	0.3413	0.3189
d1	1.0	0.7376	0.4312	0.9174	0.5357	0.4319	0.5616	0.6237	0.0918	0.6073	0.6046	0.1326
d2	0.7376	1.0	0.7906	0.6289	0.7768	0.5989	0.447	0.5875	0.1988	0.4831	0.5679	0.2428
d3	0.4312	0.7906	1.0	0.4484	0.757	0.7863	0.3872	0.3836	0.3085	0.3931	0.3796	0.3328
d4	0.9174	0.6289	0.4484	1.0	0.6049	0.5822	0.61	0.5881	0.1458	0.6368	0.602	0.1868
d5	0.5357	0.7768	0.757	0.6049	1.0	0.869	0.5117	0.5432	0.3035	0.5397	0.5699	0.3638
d6	0.4319	0.5989	0.7863	0.5822	0.869	1.0	0.4276	0.3502	0.2274	0.4304	0.377	0.2664
d7	0.5616	0.447	0.3872	0.61	0.5117	0.4276	1.0	0.8026	0.7716	0.9868	0.8338	0.7919
d8	0.6237	0.5875	0.3836	0.5881	0.5432	0.3502	0.8026	1.0	0.6291	0.8184	0.9836	0.6502
d9	0.0918	0.1988	0.3085	0.1458	0.3035	0.2274	0.7716	0.6291	1.0	0.7303	0.6485	0.9821
d10	0.6073	0.4831	0.3931	0.6368	0.5397	0.4304	0.9868	0.8184	0.7303	1.0	0.8559	0.7698
d11	0.6046	0.5679	0.3796	0.602	0.5699	0.377	0.8338	0.9836	0.6485	0.8559	1.0	0.6882
d12	0.1326	0.2428	0.3328	0.1868	0.3638	0.2664	0.7919	0.6502	0.9821	0.7698	0.6882	1.0
m1	0.4382	0.444	0.3308	0.4968	0.3474	0.2748	0.2801	0.313	0.1185	0.2633	0.3363	0.1349
m2	0.8865	0.7318	0.4287	0.8041	0.5359	0.3975	0.4883	0.5291	0.097	0.5321	0.5305	0.1467
m3	0.8283	0.7212	0.3776	0.6822	0.4614	0.3045	0.3854	0.4513	0.0671	0.4157	0.4339	0.1083
p1	0.7672	0.6954	0.3294	0.5125	0.3526	0.1944	0.2522	0.4136	-0.028	0.3023	0.3588	0.0079
p2	0.6984	0.6515	0.2834	0.4256	0.2834	0.1339	0.1976	0.3371	-0.0425	0.2458	0.2859	-0.0093
p3	0.5123	0.7847	0.6026	0.3576	0.4828	0.3369	0.2116	0.372	0.0883	0.2481	0.3575	0.1232
p4	0.5441	0.3428	0.0574	0.3096	0.1365	0.034	0.1538	0.264	-0.0224	0.1856	0.1795	-0.0124
p5	0.4181	0.3895	0.0811	0.172	0.0802	-0.0346	0.0613	0.1499	-0.0673	0.0862	0.0913	-0.0497
p6	0.7023	0.6488	0.2815	0.4294	0.285	0.1346	0.1973	0.3417	-0.0461	0.2479	0.2912	-0.0124
p7	0.1561	0.276	0.1035	0.1145	0.0844	0.0362	0.0305	0.0341	-0.0411	0.0345	0.0482	-0.0299
r1	-0.0192	0.0557	0.0539	-0.0235	0.0205	-0.0144	-0.0316	-0.0393	-0.0343	-0.0316	-0.0355	-0.0296
r2												
r3	-0.0192	0.0557	0.0539	-0.0235	0.0205	-0.0144	-0.0316	-0.0393	-0.0343	-0.0316	-0.0355	-0.0296
r4	-0.0192	0.0557	0.0539	-0.0235	0.0205	-0.0144	-0.0316	-0.0393	-0.0343	-0.0316	-0.0355	-0.0296
r5	-0.0192	0.0557	0.0539	-0.0235	0.0205	-0.0144	-0.0316	-0.0393	-0.0343	-0.0316	-0.0355	-0.0296
r6	-0.0192	0.0557	0.0539	-0.0235	0.0205	-0.0144	-0.0316	-0.0393	-0.0343	-0.0316	-0.0355	-0.0296

Table 65: Correlation between distance metrics and others for HUnit.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.2657	0.4064	0.2815	0.1894	0.2865	0.1667	0.28	0.4688	0.381	0.2447	0.4798	0.5437
c2	0.1049	0.039	0.0019	0.0796	0.0026	0.0069	0.1185	0.1653	0.0991	0.1025	0.1623	0.1345
c3	0.705	0.6495	0.3874	0.6337	0.5565	0.4165	0.7666	0.2876	0.6872	0.6872	0.8057	0.5897
c4	0.1285	0.698	0.4098	0.1041	0.7429	0.3249	0.1361	0.2904	0.2022	0.0965	0.2807	0.3168
c5	0.1532	0.6486	0.3152	0.1303	0.7125	0.28	0.1738	0.3136	0.1519	0.1289	0.3068	0.3004
c6	0.3617	0.9177	0.791	0.34	0.8604	0.6733	0.3169	0.3913	0.3833	0.2975	0.4017	0.4691
d1	1.0	0.3846	0.157	0.9914	0.2794	0.3252	0.9724	0.5927	0.1965	0.9932	0.7199	0.5656
d2	0.3846	1.0	0.7959	0.3557	0.9386	0.7065	0.3535	0.443	0.2819	0.3124	0.4545	0.4325
d3	0.157	0.7959	1.0	0.1507	0.7237	0.8513	0.103	0.2325	0.3865	0.0887	0.2239	0.3729
d4	0.9914	0.3557	0.1507	1.0	0.2677	0.3436	0.9502	0.5402	0.1888	0.9858	0.6735	0.5447
d5	0.2794	0.9386	0.7237	0.2677	1.0	0.7296	0.2529	0.3625	0.2246	0.2117	0.3618	0.3404
d6	0.3252	0.7065	0.8513	0.3436	0.7296	1.0	0.2871	0.3088	0.2639	0.269	0.3157	0.3116
d7	0.9724	0.3535	0.103	0.9502	0.2529	0.2871	1.0	0.7313	0.2638	0.9816	0.8286	0.6096
d8	0.5927	0.443	0.2325	0.5402	0.3625	0.3088	0.7313	1.0	0.6599	0.6219	0.978	0.8072
d9	0.1965	0.2819	0.3865	0.1888	0.2246	0.2639	0.2638	0.6599	1.0	0.2263	0.6047	0.8411
d10	0.9932	0.3124	0.0887	0.9858	0.2117	0.269	0.9816	0.6219	0.2263	1.0	0.7458	0.5857
d11	0.7199	0.4545	0.2239	0.6735	0.3618	0.3157	0.8286	0.978	0.6047	0.7458	1.0	0.8413
d12	0.5656	0.4325	0.3729	0.5447	0.3404	0.3116	0.6096	0.8072	0.8411	0.5857	0.8413	1.0
m1	0.8151	0.4193	0.1784	0.7488	0.3196	0.316	0.8767	0.7081	0.1151	0.8091	0.7724	0.452
m2	0.977	0.3585	0.1231	0.951	0.2479	0.2782	0.9723	0.6253	0.1723	0.9759	0.7454	0.5489
m3	0.8219	0.3631	0.1385	0.7647	0.2436	0.2576	0.86	0.6538	0.1268	0.8179	0.7349	0.4684
p1	0.9358	0.4726	0.1898	0.8866	0.3613	0.3045	0.9493	0.6744	0.162	0.9255	0.7766	0.5432
p2	0.8657	0.4675	0.1929	0.8033	0.3591	0.3145	0.9113	0.7111	0.1339	0.8572	0.7891	0.493
p3	0.5599	0.7394	0.594	0.5129	0.6645	0.5597	0.5245	0.4474	0.1785	0.5001	0.4978	0.4191
p4												
p5	0.7267	0.38	0.1781	0.6467	0.2853	0.2892	0.7921	0.6637	0.0933	0.7216	0.72	0.4124
p6	0.8577	0.4613	0.1845	0.7939	0.3524	0.3061	0.9073	0.7142	0.1305	0.85	0.7898	0.4864
p7	0.7869	0.4167	0.2027	0.7586	0.3454	0.4141	0.8376	0.6604	0.1088	0.7727	0.7106	0.4026
r1	0.5203	0.3837	0.2021	0.4557	0.3165	0.2223	0.5909	0.6742	0.2996	0.5295	0.6807	0.5373
r2												
r3	0.5203	0.3837	0.2021	0.4557	0.3165	0.2223	0.5909	0.6742	0.2996	0.5295	0.6807	0.5373
r4	0.5203	0.3837	0.2021	0.4557	0.3165	0.2223	0.5909	0.6742	0.2996	0.5295	0.6807	0.5373
r5	0.5203	0.3837	0.2021	0.4557	0.3165	0.2223	0.5909	0.6742	0.2996	0.5295	0.6807	0.5373
r6	0.5203	0.3837	0.2021	0.4557	0.3165	0.2223	0.5909	0.6742	0.2996	0.5295	0.6807	0.5373

Table 66: Correlation between distance metrics and others for PCF implementation.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.3015	0.2855	-0.1427	0.2139	0.2309	-0.1367	0.3445	0.3348	-0.0433	0.3296	0.3173	0.0821
c2	-0.0426	-0.2086	-0.2601	-0.0532	-0.0988	-0.1067	-0.0868	-0.1735	-0.2601	-0.0662	-0.1247	-0.2503
c3	0.518	0.5836	-0.1008	0.5422	0.5821	0.0651	0.4689	-0.1188	0.5663	0.5291	0.0113	0.1113
c4	0.294	0.3537	0.1453	0.2027	0.0401	-0.1903	0.3754	0.3668	0.0555	0.3669	0.3513	0.1544
c5	0.2818	0.2978	-0.0218	0.2129	0.0796	-0.1795	0.3998	0.299	-0.1013	0.4016	0.3317	0.0184
c6	0.3528	0.6286	0.3136	0.3649	0.4017	0.1748	0.3718	0.3601	0.0455	0.3642	0.3547	0.1294
d1	1.0	0.7331	-0.1265	0.9375	0.4748	0.1098	0.8504	0.5717	-0.1229	0.8923	0.6598	-0.0288
d2	0.7331	1.0	0.3086	0.7367	0.7292	0.3877	0.6229	0.5555	0.0263	0.6376	0.5704	0.0916
d3	-0.1265	0.3086	1.0	-0.0734	0.1252	0.3903	-0.1114	0.099	0.4299	-0.1304	-0.0069	0.3587
d4	0.9375	0.7367	-0.0734	1.0	0.6261	0.3228	0.7812	0.5354	-0.1515	0.8263	0.6367	-0.067
d5	0.4748	0.7292	0.1252	0.6261	1.0	0.6516	0.3989	0.2944	-0.1387	0.4187	0.3315	-0.0916
d6	0.1098	0.3877	0.3903	0.3228	0.6516	1.0	0.0097	-0.006	-0.0958	0.0248	0.0124	-0.1144
d7	0.8504	0.6229	-0.1114	0.7812	0.3989	0.0097	1.0	0.7957	0.1434	0.9896	0.8472	0.2734
d8	0.5717	0.5555	0.099	0.5354	0.2944	-0.006	0.7957	1.0	0.5201	0.7533	0.966	0.6316
d9	-0.1229	0.0263	0.4299	-0.1515	-0.1387	-0.0958	0.1434	0.5201	1.0	0.048	0.3369	0.971
d10	0.8923	0.6376	-0.1304	0.8263	0.4187	0.0248	0.9896	0.7533	0.048	1.0	0.8297	0.1811
d11	0.6598	0.5704	-0.0069	0.6367	0.3315	0.0124	0.8472	0.966	0.3369	0.8297	1.0	0.4735
d12	-0.0288	0.0916	0.3587	-0.067	-0.0916	-0.1144	0.2734	0.6316	0.971	0.1811	0.4735	1.0
m1	0.8235	0.5496	-0.1214	0.8423	0.3653	0.1749	0.635	0.4203	-0.167	0.6729	0.5018	-0.1005
m2	0.927	0.7039	-0.1601	0.834	0.492	0.0223	0.8091	0.5245	-0.1662	0.8553	0.6248	-0.0516
m3	0.8523	0.6694	-0.1196	0.7678	0.4614	0.0144	0.7437	0.4892	-0.2089	0.7953	0.5941	-0.0965
p1	0.8651	0.6715	-0.161	0.7539	0.4696	-0.0202	0.7508	0.4749	-0.1702	0.7975	0.569	-0.0553
p2	0.85	0.667	-0.1559	0.7392	0.4554	-0.0263	0.733	0.4594	-0.1907	0.7836	0.5605	-0.0747
p3	0.5069	0.5858	-0.0202	0.4458	0.3675	-0.0201	0.4208	0.402	-0.1738	0.4426	0.4598	-0.0539
p4	0.3748	0.2344	-0.0479	0.2553	0.1542	-0.0655	0.346	0.1938	-0.0474	0.389	0.248	-0.0184
p5	0.6641	0.5344	-0.1561	0.5783	0.3689	-0.0488	0.5603	0.3354	-0.2246	0.6029	0.4442	-0.1196
p6	0.8403	0.6523	-0.1555	0.7246	0.4379	-0.0351	0.7182	0.4525	-0.192	0.7687	0.5519	-0.0784
p7	0.2404	0.2621	0.0155	0.2462	0.1503	-0.038	0.2323	0.1052	-0.153	0.2397	0.1435	-0.1073
r1	0.2395	0.3328	-0.1319	0.2586	0.4019	0.0239	0.2359	0.2635	-0.1968	0.2479	0.3254	-0.0703
r2												
r3	0.2395	0.3328	-0.1319	0.2586	0.4019	0.0239	0.2359	0.2635	-0.1968	0.2479	0.3254	-0.0703
r4	0.2395	0.3328	-0.1319	0.2586	0.4019	0.0239	0.2359	0.2635	-0.1968	0.2479	0.3254	-0.0703
r5	0.2395	0.3328	-0.1319	0.2586	0.4019	0.0239	0.2359	0.2635	-0.1968	0.2479	0.3254	-0.0703
r6	0.2395	0.3328	-0.1319	0.2586	0.4019	0.0239	0.2359	0.2635	-0.1968	0.2479	0.3254	-0.0703

Table 67: Correlation between distance metrics and others for Pretty Printer Library.

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	0.0082	0.3425	0.2033	-0.0068	-0.0039	-0.0305	-0.0068	0.0357	-0.0574	-0.0075	0.0148	-0.0516
c2	-0.0118	-0.1619	-0.1624	-0.011	-0.1315	-0.1151	-0.0106	-0.1147	-0.1054	-0.0102	-0.0854	-0.0905
c3	0.81	0.4928	0.4662	0.6634	0.2324	0.2274	0.5343	0.4523	0.3508	0.5632	0.6106	0.4835
c4	0.1147	0.5963	0.4564	0.1224	0.1369	0.0778	0.0924	0.2973	0.1814	0.098	0.2806	0.2052
c5	0.157	0.4017	0.2894	0.1407	-0.022	-0.0509	0.1087	0.2113	0.0849	0.1154	0.2381	0.1351
c6	0.3796	0.2603	0.4143	0.0768	0.0942	0.1344	0.0378	0.0054	0.0053	0.0401	0.0361	0.0304
d1	1.0	0.0985	0.1366	0.918	0.0425	0.0732	0.772	0.3461	0.2204	0.8145	0.647	0.4307
d2	0.0985	1.0	0.8708	0.0645	0.5252	0.4628	0.0522	0.3959	0.3159	0.0526	0.3119	0.2906
d3	0.1366	0.8708	1.0	0.077	0.6338	0.6477	0.052	0.429	0.3773	0.0545	0.3467	0.3532
d4	0.918	0.0645	0.077	1.0	0.0756	0.1066	0.8359	0.3836	0.2521	0.8857	0.7081	0.4772
d5	0.0425	0.5252	0.6338	0.0756	1.0	0.9542	0.0563	0.6725	0.6818	0.0567	0.481	0.5601
d6	0.0732	0.4628	0.6477	0.1066	0.9542	1.0	0.0721	0.605	0.6127	0.076	0.4488	0.514
d7	0.772	0.0522	0.052	0.8359	0.0563	0.0721	1.0	0.3964	0.2825	0.995	0.6634	0.4361
d8	0.3461	0.3959	0.429	0.3836	0.6725	0.605	0.3964	1.0	0.9278	0.3972	0.8974	0.8966
d9	0.2204	0.3159	0.3773	0.2521	0.6818	0.6127	0.2825	0.9278	1.0	0.2747	0.7888	0.9324
d10	0.8145	0.0526	0.0545	0.8857	0.0567	0.076	0.995	0.3972	0.2747	1.0	0.6826	0.4456
d11	0.647	0.3119	0.3467	0.7081	0.481	0.4488	0.6634	0.8974	0.7888	0.6826	1.0	0.9069
d12	0.4307	0.2906	0.3532	0.4772	0.5601	0.514	0.4361	0.8966	0.9324	0.4456	0.9069	1.0
m1	0.0021	0.2182	0.1157	-0.0058	-0.0139	-0.0263	-0.0052	-0.0069	-0.0415	-0.0056	-0.0108	-0.0383
m2	0.994	0.094	0.1398	0.9183	0.0428	0.0744	0.7628	0.3504	0.2238	0.807	0.6533	0.4381
m3	0.9947	0.0887	0.1313	0.9392	0.0479	0.0802	0.7668	0.3521	0.226	0.8143	0.6604	0.444
p1	0.0143	0.5221	0.2539	-0.0025	-0.0117	-0.0537	0.0006	0.03	-0.0747	-0.0004	0.0118	-0.0668
p2	0.009	0.4724	0.2115	-0.0057	-0.029	-0.0664	-0.0024	0.0152	-0.0794	-0.0033	-0.0002	-0.0723
p3	0.016	0.5727	0.3892	0.0026	0.1167	0.0746	0.0008	0.093	0.0229	0.0005	0.0628	0.0197
p4												
p5	-0.0081	0.1935	0.0162	-0.0131	-0.0721	-0.0905	-0.0099	-0.0279	-0.0786	-0.0105	-0.0319	-0.0739
p6	0.0072	0.4558	0.1979	-0.007	-0.0312	-0.0682	-0.0039	0.013	-0.0809	-0.0048	-0.0022	-0.0738
p7	-0.005	0.1478	0.0532	-0.0064	-0.031	-0.0424	-0.0049	-0.016	-0.0266	-0.0052	-0.0169	-0.0273
r1	-0.0048	0.3484	0.1762	-0.0134	-0.0406	-0.0579	-0.0112	-0.029	-0.0966	-0.0118	-0.0331	-0.0874
r2	0.0028	0.2259	0.1568	-0.0053	0.0006	-0.0146	-0.0052	0.0224	-0.0433	-0.0056	0.008	-0.0384
r3	-0.0102	0.2338	0.0884	-0.0127	-0.0485	-0.0568	-0.0097	-0.0529	-0.0851	-0.0101	-0.0472	-0.0774
r4	-0.0054	0.3291	0.1754	-0.0129	-0.0344	-0.0512	-0.0107	-0.022	-0.092	-0.0113	-0.0282	-0.083
r5	-0.0013	0.2994	0.18	-0.0097	-0.0177	-0.0351	-0.0085	0.001	-0.0725	-0.009	-0.0103	-0.0651
r6	-0.0031	0.2072	0.1338	-0.0074	-0.0092	-0.0215	-0.0064	0.0041	-0.054	-0.0067	-0.006	-0.0483

Table 68: Correlation between distance metrics and others for Typing Haskell in Haskell.

E.4 Tables of Cross-correlation of Pattern Metrics

	p1	p2	p3	p4	p5	p6	p7
c1	0.3535	0.329	0.3	0.1099	0.302	0.3436	-0.0425
c2	-0.015	-0.0039	-0.0553	-0.0138	0.0402	0.0019	0.0241
c3	0.6462	0.6329	0.6567	0.1351	0.5152	0.6165	0.098
c4	0.3441	0.3167	0.3588	0.111	0.1808	0.2879	-0.0626
c5	0.3175	0.3078	0.2665	0.1103	0.2042	0.2825	-0.0455
c6	0.3747	0.3329	0.4887	0.0917	0.1287	0.3084	0.019
d1	0.8076	0.7682	0.3733	0.5759	0.5449	0.7556	0.0177
d2	0.6505	0.5933	0.5958	0.2552	0.2994	0.5621	0.0111
d3	0.3378	0.2823	0.4493	0.0794	0.0524	0.2508	-0.003
d4	0.642	0.5754	0.3481	0.4274	0.3298	0.5594	-0.0512
d5	0.3205	0.2413	0.3125	0.0704	0.0404	0.2218	-0.0923
d6	0.1277	0.0692	0.2579	0.0123	-0.0614	0.0587	-0.1037
d7	0.255	0.2173	0.162	0.1021	0.056	0.2028	-0.0405
d8	0.1725	0.1195	0.1954	-0.0125	-0.0186	0.1029	-0.0525
d9	0.0083	-0.0383	0.1175	-0.0538	-0.1304	-0.046	-0.0837
d10	0.5225	0.4814	0.2392	0.3505	0.3047	0.468	-0.0389
d11	0.2512	0.2007	0.2334	0.028	0.0743	0.1791	-0.0406
d12	0.0037	-0.0407	0.1117	-0.0413	-0.1196	-0.05	-0.0902
m1	0.5895	0.6762	0.5838	0.164	0.7697	0.6708	0.3232
m2	0.9186	0.899	0.4975	0.6532	0.69	0.8937	0.0623
m3	0.7987	0.7742	0.5597	0.5551	0.4632	0.7759	0.1074
p1	1.0	0.967	0.5597	0.7295	0.69	0.9669	0.054
p2	0.967	1.0	0.629	0.6802	0.7911	0.9945	0.2256
p3	0.5597	0.629	1.0	0.1622	0.4599	0.6321	0.3221
p4	0.7295	0.6802	0.1622	1.0	0.3987	0.7037	-0.0197
p5	0.69	0.7911	0.4599	0.3987	1.0	0.7883	0.2504
p6	0.9669	0.9945	0.6321	0.7037	0.7883	1.0	0.2194
p7	0.054	0.2256	0.3221	-0.0197	0.2504	0.2194	1.0
r1	0.4115	0.4341	0.3882	0.1644	0.5065	0.4553	0.0686
r2							
r3	0.4115	0.4341	0.3882	0.1644	0.5065	0.4553	0.0686
r4	0.4115	0.4341	0.3882	0.1644	0.5065	0.4553	0.0686
r5	0.4115	0.4341	0.3882	0.1644	0.5065	0.4553	0.0686
r6	0.4115	0.4341	0.3882	0.1644	0.5065	0.4553	0.0686

Table 69: Correlation between pattern metrics and others for the CGI Library.

	p1	p2	p3	p4	p5	p6	p7
c1	0.0729	0.1474	0.2228	-0.0234	-0.0244	0.128	0.3635
c2	0.0545	0.0483	-0.0841	0.0213	0.0222	0.0797	0.1798
c3	0.4527	0.456	0.6474	0.1775	0.3096	0.4501	0.2981
c4	0.2828	0.2847	0.352	0.0494	0.106	0.2544	0.2757
c5	0.2999	0.3042	0.3776	0.0709	0.0733	0.2665	0.2476
c6	0.2634	0.253	0.4028	0.1454	0.0688	0.2167	0.0506
d1	0.8954	0.8698	0.6603	0.6086	0.5458	0.8618	0.2451
d2	0.7007	0.6681	0.7316	0.4085	0.3424	0.6198	0.2131
d3	0.3331	0.3086	0.5339	0.1406	0.1086	0.2709	0.0776
d4	0.9882	0.9071	0.6741	0.572	0.5227	0.9039	0.4318
d5	0.6512	0.6358	0.6435	0.2499	0.2601	0.5984	0.4911
d6	0.4407	0.4126	0.5075	0.1602	0.1558	0.399	0.2461
d7	0.3363	0.3482	0.2255	0.0769	0.0631	0.3046	0.3957
d8	0.2987	0.3127	0.3051	0.0776	0.0157	0.2586	0.252
d9	0.0057	0.0157	0.0937	-0.0164	-0.0517	-0.0059	0.0106
d10	0.4947	0.503	0.3671	0.1004	0.2797	0.4951	0.3987
d11	0.3425	0.3731	0.3819	0.1014	0.0201	0.3233	0.3511
d12	0.0581	0.0726	0.1584	-0.0153	-0.0137	0.0551	0.0545
m1	0.3615	0.4328	0.573	0.1699	0.1474	0.4125	0.2907
m2	0.2422	0.2349	0.3156	0.1591	0.1504	0.2348	0.0194
m3	0.9251	0.9284	0.8171	0.6286	0.5874	0.9345	0.2722
p1	1.0	0.9836	0.8276	0.6547	0.6312	0.983	0.4067
p2	0.9836	1.0	0.8661	0.6991	0.5613	0.9882	0.4328
p3	0.8276	0.8661	1.0	0.5986	0.5201	0.8573	0.3019
p4	0.6547	0.6991	0.5986	1.0	0.3596	0.6814	-0.0451
p5	0.6312	0.5613	0.5201	0.3596	1.0	0.6561	-0.047
p6	0.983	0.9882	0.8573	0.6814	0.6561	1.0	0.385
p7	0.4067	0.4328	0.3019	-0.0451	-0.047	0.385	1.0
r1	0.2678	0.2841	0.327	-0.0412	0.3048	0.3257	0.1816
r2							
r3	0.2678	0.2841	0.327	-0.0412	0.3048	0.3257	0.1816
r4	0.2678	0.2841	0.327	-0.0412	0.3048	0.3257	0.1816
r5	0.2678	0.2841	0.327	-0.0412	0.3048	0.3257	0.1816
r6	0.2678	0.2841	0.327	-0.0412	0.3048	0.3257	0.1816

Table 70: Correlation between pattern metrics and others for the Haskell Cryptographic Library.

	p1	p2	p3	p4	p5	p6	p7
c1	0.0432	0.0681	0.0817	0.0925	-0.0047	0.0677	0.0066
c2	0.0472	0.0837	0.2149	0.001	-0.0029	0.0904	0.03
c3	0.3873	0.3626	0.3933	0.0749	-0.0521	0.3572	0.1291
c4	0.4845	0.4212	0.3058	0.0645	-0.0402	0.4083	0.0249
c5	0.2843	0.2333	0.1965	0.0156	-0.033	0.229	-0.0275
c6	0.3335	0.2723	0.283	0.0065	-0.0458	0.2498	-0.0226
d1	0.8426	0.7671	0.3048	0.1241	-0.0335	0.7222	0.0876
d2	0.7103	0.6707	0.5991	0.1747	-0.0203	0.663	0.2369
d3	0.3301	0.2714	0.3365	0.018	0.0137	0.2598	0.0296
d4	0.5462	0.4863	0.1968	0.0632	-0.0344	0.4488	0.0222
d5	0.086	0.078	0.1529	0.0101	-0.0281	0.0712	0.0543
d6	-0.0305	-0.0562	0.0389	-0.0394	-0.0176	-0.066	-0.064
d7	0.1974	0.1567	0.0255	0.0184	-0.0319	0.1348	-0.0369
d8	0.0672	0.0275	-0.0336	-0.0306	-0.0334	0.0066	-0.0836
d9	-0.0997	-0.1151	-0.0496	-0.0511	-0.0315	-0.1238	-0.0891
d10	0.424	0.381	0.1227	0.051	-0.0325	0.3508	0.0135
d11	0.021	-0.0055	-0.0116	-0.0117	-0.0325	-0.0193	-0.0638
d12	0.0145	-0.0081	0.0163	-0.0331	-0.0336	-0.0198	-0.0574
m1	0.0657	0.1041	0.2382	-0.0278	-0.0179	0.1089	0.0781
m2	0.8428	0.7861	0.3612	0.1721	-0.0372	0.7485	0.1389
m3	0.6993	0.6619	0.3366	0.1485	-0.0193	0.6464	0.1831
p1	1.0	0.9679	0.5689	0.3097	-0.0285	0.9495	0.3271
p2	0.9679	1.0	0.6512	0.3131	-0.0036	0.9893	0.4864
p3	0.5689	0.6512	1.0	0.2044	0.1086	0.7158	0.452
p4	0.3097	0.3131	0.2044	1.0	-0.0115	0.3309	0.1959
p5	-0.0285	-0.0036	0.1086	-0.0115	1.0	0.0116	0.0487
p6	0.9495	0.9893	0.7158	0.3309	0.0116	1.0	0.5291
p7	0.3271	0.4864	0.452	0.1959	0.0487	0.5291	1.0
r1	-0.0148	0.0526	0.171	-0.0488	-0.0282	0.0496	0.1192
r2							
r3	-0.0148	0.0526	0.171	-0.0488	-0.0282	0.0496	0.1192
r4	-0.0148	0.0526	0.171	-0.0488	-0.0282	0.0496	0.1192
r5	-0.0148	0.0526	0.171	-0.0488	-0.0282	0.0496	0.1192
r6	-0.0148	0.0526	0.171	-0.0488	-0.0282	0.0496	0.1192

Table 71: Correlation between pattern metrics and others for the Haskell DSP Library.

	p1	p2	p3	p4	p5	p6	p7
c1	0.4074	0.3901	0.2307	0.0423	0.1973	0.3716	-0.0144
c2	-0.0183	-0.0057	-0.0475	-0.0051	0.0151	-0.0044	-0.0164
c3	0.7487	0.7662	0.7372	0.0904	0.5556	0.7375	0.2448
c4	0.3094	0.282	0.2779	0.0653	0.144	0.2573	0.0885
c5	0.3337	0.3242	0.2753	0.0312	0.2159	0.286	0.0772
c6	0.3132	0.2876	0.3275	0.1363	0.1139	0.2584	0.0449
d1	0.8069	0.7938	0.5919	0.197	0.5159	0.7575	0.1917
d2	0.6399	0.6143	0.5759	0.1569	0.2988	0.5808	0.1867
d3	0.1561	0.1337	0.2777	0.0717	-0.0128	0.1143	0.0585
d4	0.6504	0.6484	0.5591	0.1161	0.4489	0.63	0.2121
d5	0.246	0.2341	0.3006	0.0257	0.131	0.2252	0.1117
d6	0.0012	-0.0144	0.1009	-0.0038	-0.0624	-0.0182	0.0113
d7	0.2655	0.2526	0.1948	0.1278	0.1524	0.2096	-0.0294
d8	0.1839	0.1603	0.1267	0.0426	0.0687	0.1235	-0.0528
d9	-0.118	-0.1331	-0.0786	-0.0291	-0.1313	-0.1472	-0.1227
d10	0.3665	0.359	0.2977	0.1094	0.2678	0.3143	-0.0064
d11	0.2111	0.1988	0.1813	0.0359	0.1353	0.1598	-0.0465
d12	-0.1214	-0.1315	-0.0639	-0.0295	-0.1158	-0.1487	-0.128
m1	0.7027	0.7531	0.6963	0.0807	0.6954	0.7371	0.2384
m2	0.6412	0.6313	0.4916	0.1214	0.4535	0.6133	0.1063
m3	0.5375	0.5422	0.4457	0.1183	0.3236	0.5439	0.2101
p1	1.0	0.9687	0.7428	0.1579	0.6147	0.9562	0.2056
p2	0.9687	1.0	0.7944	0.1259	0.7015	0.9826	0.3155
p3	0.7428	0.7944	1.0	0.074	0.6561	0.8294	0.478
p4	0.1579	0.1259	0.074	1.0	-0.0128	0.1164	0.043
p5	0.6147	0.7015	0.6561	-0.0128	1.0	0.7087	0.2975
p6	0.9562	0.9826	0.8294	0.1164	0.7087	1.0	0.4032
p7	0.2056	0.3155	0.478	0.043	0.2975	0.4032	1.0
r1	0.4294	0.4796	0.4994	-0.0323	0.474	0.4715	0.1822
r2							
r3	0.4294	0.4796	0.4994	-0.0323	0.474	0.4715	0.1822
r4	0.4294	0.4796	0.4994	-0.0323	0.474	0.4715	0.1822
r5	0.4294	0.4796	0.4994	-0.0323	0.474	0.4715	0.1822
r6	0.4294	0.4796	0.4994	-0.0323	0.474	0.4715	0.1822

Table 72: Correlation between pattern metrics and others for FGL.

	p1	p2	p3	p4	p5	p6	p7
c1	0.1536	0.1396	0.1125	-0.0065	0.0005	0.1207	-0.0213
c2	-0.0499	-0.0542	-0.0921	-0.0234	-0.0205	-0.0456	-0.0047
c3	0.6131	0.5636	0.5765	0.0686	0.307	0.5621	0.2924
c4	0.3707	0.3311	0.298	0.1459	0.0846	0.3116	0.1624
c5	0.347	0.3036	0.2213	0.1517	0.0891	0.2916	0.1539
c6	0.3878	0.3709	0.4388	0.0937	0.1319	0.3423	0.1701
d1	0.842	0.7936	0.4157	0.4916	0.2841	0.7637	0.3289
d2	0.6609	0.6417	0.5908	0.2701	0.2293	0.5986	0.2798
d3	0.2697	0.2498	0.3736	0.0298	0.0742	0.2233	0.1005
d4	0.778	0.7232	0.367	0.4758	0.2729	0.7092	0.2635
d5	0.4002	0.3842	0.3914	0.142	0.1768	0.3651	0.1195
d6	0.1407	0.133	0.2376	0.0016	0.0297	0.1174	0.0035
d7	0.7078	0.6399	0.312	0.38	0.3196	0.6583	0.347
d8	0.4363	0.4005	0.2888	0.1642	0.2334	0.402	0.2413
d9	0.0871	0.0669	0.093	0.0145	0.0833	0.0743	0.061
d10	0.7482	0.6886	0.3268	0.4219	0.3052	0.6888	0.3033
d11	0.4547	0.424	0.3	0.1813	0.2038	0.4166	0.2374
d12	0.1055	0.0875	0.1103	0.0169	0.0657	0.0891	0.0588
m1	0.4521	0.441	0.4095	-0.0058	0.4242	0.4991	0.3787
m2	0.8576	0.819	0.4603	0.4674	0.3667	0.8104	0.3596
m3	0.8091	0.8118	0.5048	0.5198	0.3424	0.8119	0.4699
p1	1.0	0.9594	0.622	0.5217	0.4424	0.957	0.4782
p2	0.9594	1.0	0.7078	0.5414	0.5453	0.9811	0.5462
p3	0.622	0.7078	1.0	0.168	0.4697	0.7213	0.4833
p4	0.5217	0.5414	0.168	1.0	0.2449	0.5166	0.2528
p5	0.4424	0.5453	0.4697	0.2449	1.0	0.5535	0.3379
p6	0.957	0.9811	0.7213	0.5166	0.5535	1.0	0.6249
p7	0.4782	0.5462	0.4833	0.2528	0.3379	0.6249	1.0
r1	0.3031	0.2932	0.3022	-0.0236	0.31	0.3238	0.1948
r2	0.0491	0.0542	0.0446	-0.0056	-0.0203	0.0406	-0.016
r3	0.2985	0.2875	0.2986	-0.0229	0.3191	0.3212	0.2012
r4	0.3031	0.2932	0.3022	-0.0236	0.31	0.3238	0.1948
r5	0.2972	0.2887	0.2956	-0.0235	0.2911	0.3153	0.1824
r6	0.2972	0.2887	0.2956	-0.0235	0.2911	0.3153	0.1824

Table 73: Correlation between pattern metrics and others for the Library of Geometric Algorithms.

	p1	p2	p3	p4	p5	p6	p7
c1	0.1996	0.265	0.1601		0.6954	0.203	0.0192
c2	-0.2667	-0.2328	-0.2927		-0.1478	-0.2363	-0.2419
c3	0.5782	0.5419	0.8789		0.3112	0.5502	0.3922
c4	0.7392	0.7551	0.7071		0.6594	0.7304	0.5338
c5	0.7322	0.7578	0.75		0.6097	0.7396	0.5358
c6	0.6697	0.6155	0.7859		0.5016	0.603	0.4612
d1	0.8275	0.716	0.5496		0.5531	0.6993	0.61
d2	0.7397	0.6439	0.6015		0.5298	0.6284	0.5507
d3	0.0734	-0.0021	0.2687		0.0658	-0.0128	-0.0407
d4	0.6191	0.4749	0.4229		0.3515	0.4597	0.3815
d5	0.7317	0.6509	0.694		0.5655	0.6341	0.5299
d6	0.0336	-0.0358	0.3201		-0.0115	-0.0403	-0.0792
d7	0.7688	0.8132	0.4921		0.7484	0.7821	0.6858
d8	0.2854	0.2798	0.2506		0.0689	0.2809	0.2469
d9	0.0441	0.0298	-0.034		0.0572	0.0111	-0.0193
d10	0.8495	0.8817	0.6145		0.714	0.8597	0.7484
d11	0.3385	0.3348	0.3924		0.0541	0.3414	0.2693
d12	0.2154	0.1921	0.1634		0.1786	0.1742	0.1087
m1	0.0887	0.1592	0.7057		0.2168	0.1632	0.0018
m2	0.98	0.9563	0.7146		0.7767	0.9432	0.863
m3	0.982	0.9959	0.725		0.691	0.997	0.9531
p1	1.0	0.9741	0.7076		0.6731	0.9743	0.9249
p2	0.9741	1.0	0.7233		0.7384	0.9974	0.9483
p3	0.7076	0.7233	1.0		0.5356	0.7303	0.5808
p4							
p5	0.6731	0.7384	0.5356		1.0	0.6959	0.6133
p6	0.9743	0.9974	0.7303		0.6959	1.0	0.9591
p7	0.9249	0.9483	0.5808		0.6133	0.9591	1.0
r1							
r2							
r3							
r4							
r5							
r6							

Table 74: Correlation between pattern metrics and others for GetOpt.

	p1	p2	p3	p4	p5	p6	p7
c1	-0.1259	-0.0977	-0.096	-0.0359	-0.0251	-0.1007	-0.0482
c2	0.0358	0.0423	0.0558	-0.0017	0.0121	0.0347	0.0249
c3	0.4142	0.3928	0.309	0.0934	0.3298	0.3695	0.1195
c4	0.0233	0.0329	0.0284	0.0043	0.0408	0.0229	-0.0009
c5	-0.0835	-0.0608	-0.0762	-0.0285	-0.0071	-0.0673	-0.0392
c6	0.2474	0.2232	0.2716	0.0627	0.1099	0.2124	0.1067
d1	0.7271	0.6671	0.3001	0.2009	0.2284	0.6326	0.2055
d2	0.5894	0.4984	0.5383	0.1498	0.2284	0.4718	0.1586
d3	0.0929	0.0668	0.0389	-0.0122	0.0276	0.0476	0.0011
d4	0.3858	0.3477	0.2943	0.0339	0.1813	0.316	0.0957
d5	-0.0124	-0.0085	0.1152	-0.0323	0.0224	-0.0134	-0.0412
d6	-0.1218	-0.0971	-0.1034	-0.033	-0.0173	-0.098	-0.0507
d7	0.0474	0.0441	0.0625	-0.0022	0.0172	0.0452	-0.0052
d8	0.0246	0.0134	0.1159	-0.036	0.0418	-0.0016	-0.0696
d9	-0.0478	-0.0502	0.028	-0.048	0.0019	-0.0591	-0.0782
d10	0.0824	0.0833	0.081	0.0075	0.0307	0.087	0.0025
d11	0.0524	0.0399	0.1368	-0.0183	0.0542	0.0248	-0.0601
d12	-0.0423	-0.0444	0.0316	-0.0464	0.0067	-0.0532	-0.0761
m1	0.336	0.3534	0.0795	0.0339	0.2529	0.3159	0.0488
m2	0.7343	0.7524	0.3346	0.2854	0.4239	0.7377	0.2352
m3	0.5475	0.5897	0.2878	0.2042	0.2007	0.6021	0.2993
p1	1.0	0.9022	0.5679	0.3888	0.4325	0.9001	0.3051
p2	0.9022	1.0	0.5365	0.3786	0.5674	0.9747	0.3237
p3	0.5679	0.5365	1.0	0.125	0.258	0.5282	0.2232
p4	0.3888	0.3786	0.125	1.0	0.0818	0.3502	0.1344
p5	0.4325	0.5674	0.258	0.0818	1.0	0.4897	0.2475
p6	0.9001	0.9747	0.5282	0.3502	0.4897	1.0	0.4056
p7	0.3051	0.3237	0.2232	0.1344	0.2475	0.4056	1.0
r1	-0.0165	0.0026	0.0175	0.0007	0.0403	-0.0052	-0.0053
r2	0.0627	0.057	0.0797	-0.0051	0.0017	0.0432	0.0147
r3	0.2424	0.2077	0.1814	0.0921	0.1277	0.1884	0.0372
r4	0.0643	0.0583	0.0808	-0.0045	0.0025	0.0444	0.0149
r5	0.0632	0.0573	0.08	-0.0049	0.0016	0.0435	0.0146
r6	0.1005	0.0921	0.1701	-0.003	0.0139	0.0752	0.0348

Table 75: Correlation between pattern metrics and others for Haddock.

	p1	p2	p3	p4	p5	p6	p7
c1	0.0604	0.1173	0.053	-0.0148	0.169	0.0982	0.0244
c2	-0.0197	-0.0317	-0.0452	-0.0258	-0.0532	-0.0353	-0.0308
c3	0.3673	0.6051	0.5182	0.1743	0.368	0.5908	0.2913
c4	0.2651	0.3229	0.124	0.1842	0.1338	0.3028	0.2055
c5	0.2742	0.3119	0.0927	0.0797	0.149	0.2979	0.1299
c6	0.2828	0.301	0.2352	0.1366	0.147	0.286	0.1737
d1	0.872	0.7871	0.1649	0.3542	0.3861	0.7696	0.3499
d2	0.6128	0.5889	0.3505	0.2557	0.3248	0.5821	0.2905
d3	0.2075	0.1508	0.0724	0.058	0.0573	0.1437	0.0385
d4	0.8015	0.7082	0.1181	0.3568	0.3236	0.6888	0.3118
d5	0.4383	0.4368	0.5108	0.1214	0.317	0.4252	0.2477
d6	0.2487	0.1987	0.0503	0.0907	0.1506	0.1905	0.0486
d7	0.744	0.6539	0.0892	0.329	0.2989	0.6343	0.2642
d8	0.2685	0.2971	0.193	0.0642	0.2705	0.2948	0.2408
d9	0.0044	0.0442	0.0351	-0.005	0.0779	0.0353	0.0442
d10	0.7548	0.6646	0.0939	0.3333	0.3046	0.6449	0.2703
d11	0.3257	0.3548	0.2201	0.0935	0.2829	0.3516	0.2548
d12	0.0227	0.0615	0.0451	0.0023	0.0793	0.0524	0.053
m1	0.3299	0.2999	0.0696	0.0528	0.1565	0.2834	0.1815
m2	0.9274	0.8626	0.2337	0.4164	0.4455	0.8481	0.4301
m3	0.9607	0.9594	0.4021	0.4815	0.5137	0.9579	0.6197
p1	1.0	0.9455	0.3723	0.4651	0.5038	0.935	0.5712
p2	0.9455	1.0	0.5081	0.5063	0.6074	0.9916	0.6812
p3	0.3723	0.5081	1.0	0.1163	0.4147	0.5274	0.4911
p4	0.4651	0.5063	0.1163	1.0	0.2115	0.4996	0.5178
p5	0.5038	0.6074	0.4147	0.2115	1.0	0.5899	0.4478
p6	0.935	0.9916	0.5274	0.4996	0.5899	1.0	0.6865
p7	0.5712	0.6812	0.4911	0.5178	0.4478	0.6865	1.0
r1	0.0739	0.0785	0.2309	-0.0317	-0.0401	0.075	-0.0452
r2							
r3	0.0739	0.0785	0.2309	-0.0317	-0.0401	0.075	-0.0452
r4	0.0739	0.0785	0.2309	-0.0317	-0.0401	0.075	-0.0452
r5	0.0739	0.0785	0.2309	-0.0317	-0.0401	0.075	-0.0452
r6	0.0739	0.0785	0.2309	-0.0317	-0.0401	0.075	-0.0452

Table 76: Correlation between pattern metrics and others for Happy.

	p1	p2	p3	p4	p5	p6	p7
c1	0.1393	0.1144	0.1661	0.0349	0.1141	0.1171	0.0734
c2	0.0125	0.0087	-0.062	-0.0041	-0.005	0.0069	-0.0065
c3	0.6026	0.5527	0.5229	0.163	0.5255	0.5454	0.3385
c4	0.2163	0.1783	0.2865	0.111	0.1448	0.1763	0.0957
c5	0.1865	0.1555	0.2232	0.0734	0.1471	0.1568	0.0993
c6	0.1823	0.1651	0.3757	0.0782	0.1148	0.1551	0.0816
d1	0.7437	0.7324	0.4142	0.3185	0.6117	0.7151	0.5232
d2	0.3774	0.3471	0.5465	0.2147	0.2936	0.3373	0.2243
d3	0.0703	0.0485	0.2295	0.0078	0.0185	0.0434	0.0225
d4	0.5151	0.4867	0.2913	0.1914	0.5268	0.4836	0.4119
d5	0.1259	0.1021	0.2276	0.0263	0.11	0.0997	0.057
d6	0.0382	0.0246	0.0826	-0.0116	0.0177	0.023	0.0141
d7	0.5142	0.4598	0.2627	0.1268	0.5275	0.4639	0.3599
d8	0.244	0.2084	0.2098	0.0803	0.1934	0.2064	0.1222
d9	0.0346	0.0114	0.0254	-0.0223	0.0018	0.0065	-0.0298
d10	0.5432	0.4959	0.2724	0.1299	0.5364	0.4987	0.3701
d11	0.2735	0.2398	0.2292	0.0922	0.2049	0.2369	0.1303
d12	0.0628	0.0388	0.0552	-0.015	0.0305	0.0336	-0.0112
m1	0.6305	0.6488	0.3271	0.1417	0.4218	0.6357	0.3637
m2	0.8943	0.8853	0.4339	0.3009	0.5923	0.8734	0.4926
m3	0.8769	0.8937	0.4714	0.338	0.6395	0.8948	0.6449
p1	1.0	0.9765	0.4835	0.3176	0.5832	0.9763	0.4911
p2	0.9765	1.0	0.4947	0.3091	0.6079	0.993	0.5532
p3	0.4835	0.4947	1.0	0.2349	0.5427	0.5107	0.4578
p4	0.3176	0.3091	0.2349	1.0	0.2544	0.3095	0.2995
p5	0.5832	0.6079	0.5427	0.2544	1.0	0.6302	0.7376
p6	0.9763	0.993	0.5107	0.3095	0.6302	1.0	0.6045
p7	0.4911	0.5532	0.4578	0.2995	0.7376	0.6045	1.0
r1	0.2073	0.227	0.2605	-0.012	0.2248	0.2143	0.1058
r2	-0.0094	-0.0095	0.0211	-0.0121	-0.0088	-0.0134	-0.0216
r3	0.2486	0.2724	0.2923	-0.0043	0.2735	0.2598	0.1371
r4	0.0281	0.0316	0.0649	-0.0126	0.0324	0.0258	-0.0008
r5	-0.0012	-0.0005	0.0237	-0.0119	-0.0019	-0.0047	-0.015
r6	-0.0076	-0.0068	0.0069	-0.0074	-0.001	-0.0093	-0.0156

Table 77: Correlation between pattern metrics and others for Hat.

	p1	p2	p3	p4	p5	p6	p7
c1	0.1663	0.1278	0.07	-0.0095	0.0258	0.1285	0.0172
c2	-0.0147	-0.0169	-0.044	-0.011	-0.0368	-0.0172	-0.0259
c3	0.4126	0.3654	0.4845	0.0153	0.3062	0.3717	0.2016
c4	0.2219	0.1989	0.3334	0.0506	0.2218	0.1985	0.1105
c5	0.1438	0.1235	0.1519	0.0256	0.0936	0.1215	0.0934
c6	0.258	0.2381	0.4822	0.0504	0.225	0.2362	0.1718
d1	0.8313	0.8107	0.3436	0.6235	0.5388	0.8254	0.502
d2	0.3818	0.357	0.5536	0.1285	0.3299	0.3543	0.2469
d3	0.1083	0.0818	0.3214	0.0165	0.0483	0.0807	0.0667
d4	0.479	0.4609	0.3063	0.3073	0.3336	0.4746	0.2319
d5	0.0945	0.0733	0.2562	0.0005	0.0806	0.0713	-0.0169
d6	-0.0111	-0.0224	0.148	-0.0184	-0.0254	-0.0219	-0.047
d7	0.3004	0.2878	0.2208	0.0581	0.2638	0.3085	0.1083
d8	0.0976	0.0981	0.2549	0.0423	0.1622	0.0975	0.0312
d9	-0.0366	-0.0279	0.1822	-0.027	0.0663	-0.0255	-0.0682
d10	0.4	0.4012	0.2243	0.1154	0.2913	0.4101	0.1532
d11	0.142	0.1346	0.2169	0.069	0.1373	0.1291	0.0535
d12	-0.0256	-0.0232	0.1408	-0.0185	0.0226	-0.0248	-0.0622
m1	0.0672	0.1137	0.0128	-0.0026	-0.001	0.1033	0.0956
m2	0.8519	0.8388	0.3103	0.5978	0.5368	0.853	0.5148
m3	0.8526	0.8647	0.3341	0.6858	0.5675	0.8748	0.6192
p1	1.0	0.9686	0.4133	0.7714	0.6445	0.9737	0.5892
p2	0.9686	1.0	0.4627	0.7615	0.7316	0.9959	0.6579
p3	0.4133	0.4627	1.0	0.124	0.6125	0.4773	0.4962
p4	0.7714	0.7615	0.124	1.0	0.4125	0.7606	0.5002
p5	0.6445	0.7316	0.6125	0.4125	1.0	0.7249	0.5239
p6	0.9737	0.9959	0.4773	0.7606	0.7249	1.0	0.6776
p7	0.5892	0.6579	0.4962	0.5002	0.5239	0.6776	1.0
r1	0.1987	0.1677	0.1792	-0.0183	0.1497	0.1624	0.0172
r2	0.0104	-0.0058	-0.0255	-0.0049	-0.0362	-0.0074	-0.0229
r3	0.195	0.1685	0.1836	-0.0171	0.1582	0.163	0.0225
r4	0.1689	0.1384	0.1409	-0.0169	0.1142	0.133	0.0072
r5	0.087	0.0622	0.0508	-0.0108	0.0314	0.0587	-0.0105
r6	-0.0001	-0.0058	-0.0138	-0.0031	-0.018	-0.0068	-0.0127

Table 78: Correlation between pattern metrics and others for HaXml.

	p1	p2	p3	p4	p5	p6	p7
c1							
c2	0.0445	0.0436	0.1694	-0.0241	0.072	0.0363	-0.0053
c3	0.5536	0.4999	0.6586	0.2325	0.2886	0.4949	0.2907
c4	0.4857	0.4421	0.5862	0.2582	0.2925	0.4358	0.1038
c5	0.452	0.403	0.4728	0.2402	0.2373	0.4021	0.0614
c6	0.2813	0.2544	0.5915	0.0632	0.1133	0.2483	0.0839
d1	0.7672	0.6984	0.5123	0.5441	0.4181	0.7023	0.1561
d2	0.6954	0.6515	0.7847	0.3428	0.3895	0.6488	0.276
d3	0.3294	0.2834	0.6026	0.0574	0.0811	0.2815	0.1035
d4	0.5125	0.4256	0.3576	0.3096	0.172	0.4294	0.1145
d5	0.3526	0.2834	0.4828	0.1365	0.0802	0.285	0.0844
d6	0.1944	0.1339	0.3369	0.034	-0.0346	0.1346	0.0362
d7	0.2522	0.1976	0.2116	0.1538	0.0613	0.1973	0.0305
d8	0.4136	0.3371	0.372	0.264	0.1499	0.3417	0.0341
d9	-0.028	-0.0425	0.0883	-0.0224	-0.0673	-0.0461	-0.0411
d10	0.3023	0.2458	0.2481	0.1856	0.0862	0.2479	0.0345
d11	0.3588	0.2859	0.3575	0.1795	0.0913	0.2912	0.0482
d12	0.0079	-0.0093	0.1232	-0.0124	-0.0497	-0.0124	-0.0299
m1	0.2696	0.2901	0.5134	0.0138	0.3343	0.2775	0.376
m2	0.7524	0.7011	0.5587	0.4387	0.3773	0.7026	0.1374
m3	0.8406	0.8211	0.5931	0.5808	0.5504	0.8184	0.2166
p1	1.0	0.979	0.6711	0.7236	0.6591	0.9811	0.1307
p2	0.979	1.0	0.6864	0.7109	0.7399	0.9982	0.2237
p3	0.6711	0.6864	1.0	0.2631	0.5384	0.684	0.3271
p4	0.7236	0.7109	0.2631	1.0	0.7394	0.7128	0.0197
p5	0.6591	0.7399	0.5384	0.7394	1.0	0.7235	0.3965
p6	0.9811	0.9982	0.684	0.7128	0.7235	1.0	0.2246
p7	0.1307	0.2237	0.3271	0.0197	0.3965	0.2246	1.0
r1	-0.0094	0.0807	0.224	-0.0105	0.2943	0.0794	0.6287
r2							
r3	-0.0094	0.0807	0.224	-0.0105	0.2943	0.0794	0.6287
r4	-0.0094	0.0807	0.224	-0.0105	0.2943	0.0794	0.6287
r5	-0.0094	0.0807	0.224	-0.0105	0.2943	0.0794	0.6287
r6	-0.0094	0.0807	0.224	-0.0105	0.2943	0.0794	0.6287

Table 79: Correlation between pattern metrics and others for HUnit.

	p1	p2	p3	p4	p5	p6	p7
c1	0.3866	0.3942	0.4578		0.4247	0.3947	0.1694
c2	0.136	0.1501	0.188		0.1724	0.1533	0.0908
c3	0.8734	0.902	0.7455		0.8647	0.9013	0.7617
c4	0.2487	0.2159	0.3113		0.1442	0.2093	0.1095
c5	0.2694	0.2379	0.2679		0.1606	0.2323	0.1436
c6	0.4375	0.4036	0.7385		0.3168	0.396	0.3089
d1	0.9358	0.8657	0.5599		0.7267	0.8577	0.7869
d2	0.4726	0.4675	0.7394		0.38	0.4613	0.4167
d3	0.1898	0.1929	0.594		0.1781	0.1845	0.2027
d4	0.8866	0.8033	0.5129		0.6467	0.7939	0.7586
d5	0.3613	0.3591	0.6645		0.2853	0.3524	0.3454
d6	0.3045	0.3145	0.5597		0.2892	0.3061	0.4141
d7	0.9493	0.9113	0.5245		0.7921	0.9073	0.8376
d8	0.6744	0.7111	0.4474		0.6637	0.7142	0.6604
d9	0.162	0.1339	0.1785		0.0933	0.1305	0.1088
d10	0.9255	0.8572	0.5001		0.7216	0.85	0.7727
d11	0.7766	0.7891	0.4978		0.72	0.7898	0.7106
d12	0.5432	0.493	0.4191		0.4124	0.4864	0.4026
m1	0.942	0.9867	0.6755		0.9734	0.9881	0.8813
m2	0.9679	0.9224	0.5925		0.8166	0.9166	0.8084
m3	0.9166	0.9383	0.682		0.9074	0.9378	0.8256
p1	1.0	0.978	0.6682		0.8955	0.9744	0.8205
p2	0.978	1.0	0.6915		0.9584	0.9997	0.8689
p3	0.6682	0.6915	1.0		0.6845	0.6863	0.6176
p4							
p5	0.8955	0.9584	0.6845		1.0	0.9599	0.8175
p6	0.9744	0.9997	0.6863		0.9599	1.0	0.8668
p7	0.8205	0.8689	0.6176		0.8175	0.8668	1.0
r1	0.6473	0.6608	0.5278		0.6402	0.6626	0.4857
r2							
r3	0.6473	0.6608	0.5278		0.6402	0.6626	0.4857
r4	0.6473	0.6608	0.5278		0.6402	0.6626	0.4857
r5	0.6473	0.6608	0.5278		0.6402	0.6626	0.4857
r6	0.6473	0.6608	0.5278		0.6402	0.6626	0.4857

Table 80: Correlation between pattern metrics and others for PCF implementation.

	p1	p2	p3	p4	p5	p6	p7
c1	0.4454	0.411	0.6142	-0.0424	0.404	0.4241	0.2409
c2	-0.0157	-0.0114	-0.0136	-0.0347	0.0311	-0.0061	0.0426
c3	0.675	0.6434	0.6446	0.0014	0.6618	0.629	0.2841
c4	0.3322	0.2963	0.3182	0.0575	0.1914	0.292	0.0657
c5	0.3255	0.2996	0.2771	0.0565	0.2022	0.2886	0.1545
c6	0.3824	0.3686	0.4856	0.0784	0.3142	0.3645	0.1698
d1	0.8651	0.85	0.5069	0.3748	0.6641	0.8403	0.2404
d2	0.6715	0.667	0.5858	0.2344	0.5344	0.6523	0.2621
d3	-0.161	-0.1559	-0.0202	-0.0479	-0.1561	-0.1555	0.0155
d4	0.7539	0.7392	0.4458	0.2553	0.5783	0.7246	0.2462
d5	0.4696	0.4554	0.3675	0.1542	0.3689	0.4379	0.1503
d6	-0.0202	-0.0263	-0.0201	-0.0655	-0.0488	-0.0351	-0.038
d7	0.7508	0.733	0.4208	0.346	0.5603	0.7182	0.2323
d8	0.4749	0.4594	0.402	0.1938	0.3354	0.4525	0.1052
d9	-0.1702	-0.1907	-0.1738	-0.0474	-0.2246	-0.192	-0.153
d10	0.7975	0.7836	0.4426	0.389	0.6029	0.7687	0.2397
d11	0.569	0.5605	0.4598	0.248	0.4442	0.5519	0.1435
d12	-0.0553	-0.0747	-0.0539	-0.0184	-0.1196	-0.0784	-0.1073
m1	0.5776	0.5796	0.4795	-0.0085	0.5603	0.5766	0.2395
m2	0.9858	0.9778	0.6439	0.5356	0.8248	0.9734	0.2586
m3	0.9462	0.9706	0.7261	0.5923	0.8528	0.9753	0.4227
p1	1.0	0.9901	0.661	0.5636	0.8515	0.9865	0.2433
p2	0.9901	1.0	0.6977	0.5839	0.8841	0.9983	0.3276
p3	0.661	0.6977	1.0	0.1744	0.7531	0.7083	0.454
p4	0.5636	0.5839	0.1744	1.0	0.3182	0.5946	-0.039
p5	0.8515	0.8841	0.7531	0.3182	1.0	0.8899	0.4205
p6	0.9865	0.9983	0.7083	0.5946	0.8899	1.0	0.3343
p7	0.2433	0.3276	0.454	-0.039	0.4205	0.3343	1.0
r1	0.3861	0.3953	0.684	-0.065	0.5217	0.4024	0.2253
r2							
r3	0.3861	0.3953	0.684	-0.065	0.5217	0.4024	0.2253
r4	0.3861	0.3953	0.684	-0.065	0.5217	0.4024	0.2253
r5	0.3861	0.3953	0.684	-0.065	0.5217	0.4024	0.2253
r6	0.3861	0.3953	0.684	-0.065	0.5217	0.4024	0.2253

Table 81: Correlation between pattern metrics and others for Pretty Printer Library.

	p1	p2	p3	p4	p5	p6	p7
c1	0.6246	0.5467	0.2829		0.2349	0.5564	0.0953
c2	-0.0015	-0.0026	-0.0221		-0.0055	-0.0043	-0.0108
c3	0.34	0.3245	0.3379		0.2091	0.3165	0.0557
c4	0.4508	0.3914	0.3902		0.1403	0.3832	0.0861
c5	0.3855	0.3308	0.2556		0.1123	0.3292	0.0668
c6	0.005	-0.003	0.109		-0.0184	-0.0071	-0.0153
d1	0.0143	0.009	0.016		-0.0081	0.0072	-0.005
d2	0.5221	0.4724	0.5727		0.1935	0.4558	0.1478
d3	0.2539	0.2115	0.3892		0.0162	0.1979	0.0532
d4	-0.0025	-0.0057	0.0026		-0.0131	-0.007	-0.0064
d5	-0.0117	-0.029	0.1167		-0.0721	-0.0312	-0.031
d6	-0.0537	-0.0664	0.0746		-0.0905	-0.0682	-0.0424
d7	0.0006	-0.0024	0.0008		-0.0099	-0.0039	-0.0049
d8	0.03	0.0152	0.093		-0.0279	0.013	-0.016
d9	-0.0747	-0.0794	0.0229		-0.0786	-0.0809	-0.0266
d10	-0.0004	-0.0033	0.0005		-0.0105	-0.0048	-0.0052
d11	0.0118	-0.0002	0.0628		-0.0319	-0.0022	-0.0169
d12	-0.0668	-0.0723	0.0197		-0.0739	-0.0738	-0.0273
m1	0.5144	0.5895	0.5089		0.6214	0.5714	0.0751
m2	0.0037	0.0001	0.0132		-0.0097	-0.0012	-0.0079
m3	0.0001	-0.0028	0.0109		-0.0108	-0.004	-0.0047
p1	1.0	0.9744	0.6655		0.6861	0.9766	0.1333
p2	0.9744	1.0	0.7083		0.806	0.9957	0.2197
p3	0.6655	0.7083	1.0		0.6384	0.6995	0.202
p4							
p5	0.6861	0.806	0.6384		1.0	0.8015	0.1999
p6	0.9766	0.9957	0.6995		0.8015	1.0	0.2255
p7	0.1333	0.2197	0.202		0.1999	0.2255	1.0
r1	0.636	0.6439	0.5313		0.474	0.6417	0.2047
r2	0.2696	0.1977	0.1162		-0.0257	0.2004	0.0259
r3	0.5332	0.5922	0.5281		0.5649	0.5928	0.2067
r4	0.5753	0.5664	0.4622		0.3873	0.5687	0.1669
r5	0.453	0.408	0.3076		0.1886	0.4107	0.1021
r6	0.2901	0.242	0.1772		0.0741	0.2506	0.0408

Table 82: Correlation between pattern metrics and others for Typing Haskell in Haskell.

E.5 Tables of Cross-correlation of Miscellaneous Metrics

	m1	m2	m3
c1	0.2897	0.3238	0.2783
c2	0.0576	-0.0346	-0.0703
c3	0.676	0.6638	0.6467
c4	0.2985	0.3194	0.2779
c5	0.2455	0.3261	0.281
c6	0.3336	0.4434	0.4822
d1	0.4498	0.9232	0.8406
d2	0.4639	0.6455	0.6612
d3	0.2247	0.3683	0.4304
d4	0.3253	0.7298	0.6947
d5	0.1993	0.2757	0.3087
d6	0.0828	0.1381	0.185
d7	0.1999	0.295	0.2404
d8	0.1355	0.1699	0.2165
d9	0.0148	0.0091	0.0506
d10	0.3378	0.5368	0.3876
d11	0.208	0.2421	0.2274
d12	0.0033	-0.0086	0.0096
m1	1.0	0.6013	0.4761
m2	0.6013	1.0	0.8835
m3	0.4761	0.8835	1.0
p1	0.5895	0.9186	0.7987
p2	0.6762	0.899	0.7742
p3	0.5838	0.4975	0.5597
p4	0.164	0.6532	0.5551
p5	0.7697	0.69	0.4632
p6	0.6708	0.8937	0.7759
p7	0.3232	0.0623	0.1074
r1	0.5266	0.3995	0.3145
r2			
r3	0.5266	0.3995	0.3145
r4	0.5266	0.3995	0.3145
r5	0.5266	0.3995	0.3145
r6	0.5266	0.3995	0.3145

Table 83: Correlation between miscellaneous metrics and others for the CGI Library.

	m1	m2	m3
c1	0.5035	-0.0362	0.0791
c2	0.2436	-0.1454	0.004
c3	0.5823	0.1328	0.4337
c4	0.2573	0.0037	0.2072
c5	0.2563	-0.0294	0.2547
c6	0.4538	0.2638	0.3028
d1	0.2724	0.2201	0.9061
d2	0.3888	0.3172	0.6406
d3	0.2668	0.5507	0.3589
d4	0.2647	0.2347	0.903
d5	0.4189	0.2791	0.5361
d6	0.2802	0.5338	0.4473
d7	0.2813	0.0363	0.3507
d8	0.3023	0.0156	0.3071
d9	0.212	-0.0346	0.0526
d10	0.3555	0.0977	0.541
d11	0.3302	0.007	0.3054
d12	0.2724	-0.0285	0.1
m1	1.0	0.1539	0.4254
m2	0.1539	1.0	0.3584
m3	0.4254	0.3584	1.0
p1	0.3615	0.2422	0.9251
p2	0.4328	0.2349	0.9284
p3	0.573	0.3156	0.8171
p4	0.1699	0.1591	0.6286
p5	0.1474	0.1504	0.5874
p6	0.4125	0.2348	0.9345
p7	0.2907	0.0194	0.2722
r1	0.2948	0.0417	0.3214
r2			
r3	0.2948	0.0417	0.3214
r4	0.2948	0.0417	0.3214
r5	0.2948	0.0417	0.3214
r6	0.2948	0.0417	0.3214

Table 84: Correlation between miscellaneous metrics and others for the Haskell Cryptographic Library.

	m1	m2	m3
c1	0.0794	-0.0003	0.0033
c2	0.2474	0.0212	0.0024
c3	0.503	0.3838	0.3057
c4	0.088	0.4465	0.3282
c5	0.073	0.2805	0.1758
c6	0.2779	0.4046	0.3757
d1	0.1477	0.9569	0.7484
d2	0.3475	0.6302	0.5335
d3	0.311	0.3654	0.3222
d4	0.4793	0.808	0.5999
d5	0.6976	0.2974	0.2304
d6	0.6478	0.2305	0.1725
d7	0.5669	0.4187	0.331
d8	0.4824	0.2773	0.2188
d9	0.4185	0.076	0.0569
d10	0.5124	0.6937	0.5607
d11	0.5732	0.2603	0.2199
d12	0.4714	0.2249	0.1789
m1	1.0	0.2799	0.244
m2	0.2799	1.0	0.8752
m3	0.244	0.8752	1.0
p1	0.0657	0.8428	0.6993
p2	0.1041	0.7861	0.6619
p3	0.2382	0.3612	0.3366
p4	-0.0278	0.1721	0.1485
p5	-0.0179	-0.0372	-0.0193
p6	0.1089	0.7485	0.6464
p7	0.0781	0.1389	0.1831
r1	0.6838	0.1178	0.0973
r2			
r3	0.6838	0.1178	0.0973
r4	0.6838	0.1178	0.0973
r5	0.6838	0.1178	0.0973
r6	0.6838	0.1178	0.0973

Table 85: Correlation between miscellaneous metrics and others for the Haskell DSP Library.

	m1	m2	m3
c1	0.3295	0.2692	0.2083
c2	0.0399	-0.0554	-0.0636
c3	0.7439	0.5643	0.451
c4	0.2577	0.1461	0.0998
c5	0.2391	0.2148	0.1626
c6	0.3033	0.2563	0.2018
d1	0.633	0.6522	0.5347
d2	0.4159	0.4975	0.4459
d3	0.0898	0.2577	0.244
d4	0.564	0.5767	0.475
d5	0.2139	0.3115	0.2908
d6	0.0301	0.1945	0.1906
d7	0.2244	0.248	0.2151
d8	0.1069	0.1529	0.1294
d9	-0.0893	-0.0299	-0.0198
d10	0.3664	0.3764	0.317
d11	0.1748	0.2146	0.1807
d12	-0.0718	0.0298	0.0375
m1	1.0	0.5731	0.4508
m2	0.5731	1.0	0.9402
m3	0.4508	0.9402	1.0
p1	0.7027	0.6412	0.5375
p2	0.7531	0.6313	0.5422
p3	0.6963	0.4916	0.4457
p4	0.0807	0.1214	0.1183
p5	0.6954	0.4535	0.3236
p6	0.7371	0.6133	0.5439
p7	0.2384	0.1063	0.2101
r1	0.5925	0.3316	0.2362
r2			
r3	0.5925	0.3316	0.2362
r4	0.5925	0.3316	0.2362
r5	0.5925	0.3316	0.2362
r6	0.5925	0.3316	0.2362

Table 86: Correlation between miscellaneous metrics and others for FGL.

	m1	m2	m3
c1	0.1528	0.1159	0.0673
c2	-0.0184	-0.0741	-0.0796
c3	0.5573	0.4781	0.3946
c4	0.2244	0.2842	0.2499
c5	0.2349	0.3029	0.2648
c6	0.2539	0.3046	0.2705
d1	0.231	0.8834	0.7863
d2	0.2336	0.5489	0.5159
d3	0.1136	0.2255	0.2158
d4	0.229	0.8347	0.7403
d5	0.2503	0.3079	0.2503
d6	0.0984	0.1123	0.0709
d7	0.3109	0.6837	0.6283
d8	0.2969	0.3562	0.3333
d9	0.1445	0.0663	0.0437
d10	0.2557	0.7709	0.6894
d11	0.2678	0.3813	0.3591
d12	0.1319	0.0959	0.0604
m1	1.0	0.3642	0.3241
m2	0.3642	1.0	0.927
m3	0.3241	0.927	1.0
p1	0.4521	0.8576	0.8091
p2	0.441	0.819	0.8118
p3	0.4095	0.4603	0.5048
p4	-0.0058	0.4674	0.5198
p5	0.4242	0.3667	0.3424
p6	0.4991	0.8104	0.8119
p7	0.3787	0.3596	0.4699
r1	0.4921	0.2068	0.1702
r2	0.0783	0.0079	-0.0004
r3	0.485	0.2087	0.1731
r4	0.4921	0.2068	0.1702
r5	0.4823	0.1981	0.1618
r6	0.4823	0.1981	0.1618

Table 87: Correlation between miscellaneous metrics and others for the Library of Geometric Algorithms.

	m1	m2	m3
c1	0.1157	0.3604	0.2052
c2	-0.0647	-0.2666	-0.2435
c3	0.5468	0.5855	0.5586
c4	0.3481	0.7944	0.7344
c5	0.4157	0.7722	0.7402
c6	0.3856	0.7202	0.6249
d1	-0.0046	0.8678	0.7401
d2	0.0338	0.7847	0.6624
d3	0.2	0.1281	0.0068
d4	0.0083	0.6671	0.5102
d5	0.1728	0.7898	0.6655
d6	0.33	0.086	-0.0191
d7	0.0065	0.8119	0.7943
d8	0.0375	0.2523	0.2882
d9	-0.0984	0.065	0.0205
d10	0.1052	0.8738	0.8717
d11	0.2035	0.2981	0.3455
d12	-0.0031	0.2405	0.1833
m1	1.0	0.1225	0.1366
m2	0.1225	1.0	0.9548
m3	0.1366	0.9548	1.0
p1	0.0887	0.98	0.982
p2	0.1592	0.9563	0.9959
p3	0.7057	0.7146	0.725
p4			
p5	0.2168	0.7767	0.691
p6	0.1632	0.9432	0.997
p7	0.0018	0.863	0.9531
r1			
r2			
r3			
r4			
r5			
r6			

Table 88: Correlation between miscellaneous metrics and others for GetOpt.

	m1	m2	m3
c1	-0.0146	-0.0596	-0.0538
c2	-0.0017	0.0	0.0067
c3	0.1973	0.7999	0.8186
c4	0.0578	0.0495	0.0323
c5	0.0667	-0.0272	-0.0273
c6	0.0527	0.2089	0.1643
d1	0.5105	0.8121	0.6994
d2	0.2682	0.4101	0.3112
d3	0.2653	0.1144	0.0979
d4	0.1444	0.3915	0.3266
d5	-0.0092	-0.0142	-0.003
d6	-0.0096	-0.0608	-0.0373
d7	0.0064	0.582	0.7337
d8	-0.0055	0.0104	-0.0021
d9	-0.0197	-0.039	-0.035
d10	0.0166	0.6093	0.7541
d11	0.0035	0.0311	0.012
d12	-0.0172	-0.0369	-0.0359
m1	1.0	0.2832	0.2103
m2	0.2832	1.0	0.9194
m3	0.2103	0.9194	1.0
p1	0.336	0.7343	0.5475
p2	0.3534	0.7524	0.5897
p3	0.0795	0.3346	0.2878
p4	0.0339	0.2854	0.2042
p5	0.2529	0.4239	0.2007
p6	0.3159	0.7377	0.6021
p7	0.0488	0.2352	0.2993
r1	0.0043	0.0104	0.0012
r2	0.0033	0.3925	0.5021
r3	0.0408	0.1626	0.1294
r4	0.0036	0.3932	0.5024
r5	0.0033	0.3927	0.5022
r6	0.0006	0.0336	0.0367

Table 89: Correlation between miscellaneous metrics and others for Haddock.

	m1	m2	m3
c1	-0.0056	0.0438	0.0641
c2	-0.0127	-0.0318	-0.039
c3	0.3706	0.4614	0.5749
c4	0.0871	0.2449	0.3269
c5	0.0663	0.2815	0.3555
c6	0.0512	0.2359	0.2954
d1	0.2043	0.9679	0.8692
d2	0.2157	0.4461	0.5824
d3	0.0363	0.1726	0.167
d4	0.0856	0.9534	0.7766
d5	0.075	0.3092	0.3718
d6	0.0158	0.2383	0.2056
d7	0.0309	0.9198	0.722
d8	0.0689	0.2928	0.3277
d9	-0.0176	0.1058	0.0715
d10	0.0483	0.9267	0.7331
d11	0.0823	0.3333	0.38
d12	-0.0106	0.1209	0.0892
m1	1.0	0.2151	0.3414
m2	0.2151	1.0	0.9176
m3	0.3414	0.9176	1.0
p1	0.3299	0.9274	0.9607
p2	0.2999	0.8626	0.9594
p3	0.0696	0.2337	0.4021
p4	0.0528	0.4164	0.4815
p5	0.1565	0.4455	0.5137
p6	0.2834	0.8481	0.9579
p7	0.1815	0.4301	0.6197
r1	-0.0158	0.0356	0.0544
r2			
r3	-0.0158	0.0356	0.0544
r4	-0.0158	0.0356	0.0544
r5	-0.0158	0.0356	0.0544
r6	-0.0158	0.0356	0.0544

Table 90: Correlation between miscellaneous metrics and others for Happy.

	m1	m2	m3
c1	0.0363	0.1461	0.1433
c2	-0.002	0.0048	-0.0021
c3	0.5311	0.6722	0.6089
c4	0.0937	0.225	0.1983
c5	0.0907	0.2117	0.1936
c6	0.1378	0.1841	0.1586
d1	0.5327	0.8749	0.8068
d2	0.212	0.3746	0.35
d3	0.0127	0.0571	0.0382
d4	0.3273	0.6586	0.5815
d5	0.0469	0.1605	0.1221
d6	0.0047	0.0632	0.0392
d7	0.3018	0.6315	0.5298
d8	0.12	0.2877	0.2577
d9	0.0092	0.0618	0.0115
d10	0.3365	0.6559	0.5452
d11	0.1422	0.3155	0.2801
d12	0.033	0.095	0.0387
m1	1.0	0.6346	0.595
m2	0.6346	1.0	0.9499
m3	0.595	0.9499	1.0
p1	0.6305	0.8943	0.8769
p2	0.6488	0.8853	0.8937
p3	0.3271	0.4339	0.4714
p4	0.1417	0.3009	0.338
p5	0.4218	0.5923	0.6395
p6	0.6357	0.8734	0.8948
p7	0.3637	0.4926	0.6449
r1	0.2293	0.2132	0.1965
r2	-0.0133	0.0055	-0.011
r3	0.2766	0.2465	0.238
r4	0.0284	0.0426	0.0249
r5	-0.0036	0.0147	-0.0018
r6	-0.007	0.0007	-0.0086

Table 91: Correlation between miscellaneous metrics and others for Hat.

	m1	m2	m3
c1	0.0045	0.1778	0.1485
c2	-0.0076	-0.0468	-0.0514
c3	0.0499	0.4352	0.3532
c4	-0.0145	0.1938	0.1406
c5	-0.0069	0.1787	0.1312
c6	0.045	0.1796	0.144
d1	0.0547	0.8692	0.8025
d2	0.0311	0.3148	0.2849
d3	0.0169	0.0635	0.0265
d4	0.0621	0.5726	0.4879
d5	0.0309	0.1055	0.056
d6	0.0354	0.0141	-0.025
d7	0.0037	0.4759	0.2855
d8	-0.0275	0.1324	0.0695
d9	-0.0267	0.013	-0.0453
d10	0.0207	0.57	0.3883
d11	-0.0226	0.17	0.1139
d12	-0.0258	0.0217	-0.03
m1	1.0	0.1385	0.1424
m2	0.1385	1.0	0.9489
m3	0.1424	0.9489	1.0
p1	0.0672	0.8519	0.8526
p2	0.1137	0.8388	0.8647
p3	0.0128	0.3103	0.3341
p4	-0.0026	0.5978	0.6858
p5	-0.001	0.5308	0.5675
p6	0.1033	0.853	0.8748
p7	0.0956	0.5148	0.6192
r1	0.0289	0.1707	0.1584
r2	-0.0028	0.0095	0.0219
r3	0.0288	0.1724	0.1491
r4	0.0227	0.1495	0.1363
r5	0.0091	0.0782	0.0811
r6	-0.0022	0.0009	0.0076

Table 92: Correlation between miscellaneous metrics and others for HaXml.

	m1	m2	m3
c1	0.247	-0.0052	0.0083
c2	0.5463	0.611	0.612
c4	0.3564	0.5189	0.4864
c5	0.3261	0.495	0.448
c6	0.3878	0.3918	0.3348
d1	0.4382	0.8865	0.8283
d2	0.444	0.7318	0.7212
d3	0.3308	0.4287	0.3776
d4	0.4968	0.8041	0.6822
d5	0.3474	0.5359	0.4614
d6	0.2748	0.3975	0.3045
d7	0.2801	0.4883	0.3854
d8	0.313	0.5291	0.4513
d9	0.1185	0.097	0.0671
d10	0.2633	0.5321	0.4157
d11	0.3363	0.5305	0.4339
d12	0.1349	0.1467	0.1083
m1	1.0	0.4648	0.4494
m2	0.4648	1.0	0.9343
m3	0.4494	0.9343	1.0
p1	0.2696	0.7524	0.8406
p2	0.2901	0.7011	0.8211
p3	0.5134	0.5587	0.5931
p4	0.0138	0.4387	0.5868
p5	0.3343	0.3773	0.5504
p6	0.2775	0.7026	0.8184
p7	0.376	0.1374	0.2166
r1	0.2959	-0.0159	0.0449
r2			
r3	0.2959	-0.0159	0.0449
r4	0.2959	-0.0159	0.0449
r5	0.2959	-0.0159	0.0449
r6	0.2959	-0.0159	0.0449

Table 93: Correlation between miscellaneous metrics and others for HUnit.

	m1	m2	m3
c1	0.3724	0.3016	0.3842
c2	0.1474	0.0983	0.1229
c3	0.8835	0.7812	0.8634
c4	0.1527	0.1237	0.0908
c5	0.1767	0.1483	0.1061
c6	0.3476	0.347	0.3513
d1	0.8151	0.977	0.8219
d2	0.4193	0.3585	0.3631
d3	0.1784	0.1231	0.1385
d4	0.7488	0.951	0.7647
d5	0.3196	0.2479	0.2436
d6	0.316	0.2782	0.2576
d7	0.8767	0.9723	0.86
d8	0.7081	0.6253	0.6538
d9	0.1151	0.1723	0.1268
d10	0.8091	0.9759	0.8179
d11	0.7724	0.7454	0.7349
d12	0.452	0.5489	0.4684
m1	1.0	0.8845	0.9342
m2	0.8845	1.0	0.915
m3	0.9342	0.915	1.0
p1	0.942	0.9679	0.9166
p2	0.9867	0.9224	0.9383
p3	0.6755	0.5925	0.682
p4			
p5	0.9734	0.8166	0.9074
p6	0.9881	0.9166	0.9378
p7	0.8813	0.8084	0.8256
r1	0.6523	0.5684	0.5959
r2			
r3	0.6523	0.5684	0.5959
r4	0.6523	0.5684	0.5959
r5	0.6523	0.5684	0.5959
r6	0.6523	0.5684	0.5959

Table 94: Correlation between miscellaneous metrics and others for PCF implementation.

	m1	m2	m3
c1	0.2042	0.415	0.4076
c2	-0.0093	-0.0249	-0.0077
c3	0.3867	0.6551	0.6011
c4	0.1411	0.3197	0.2697
c5	0.113	0.3157	0.272
c6	0.2964	0.3883	0.3758
d1	0.8235	0.927	0.8523
d2	0.5496	0.7039	0.6694
d3	-0.1214	-0.1601	-0.1196
d4	0.8423	0.834	0.7678
d5	0.3653	0.492	0.4614
d6	0.1749	0.0223	0.0144
d7	0.635	0.8091	0.7437
d8	0.4203	0.5245	0.4892
d9	-0.167	-0.1662	-0.2089
d10	0.6729	0.8553	0.7953
d11	0.5018	0.6248	0.5941
d12	-0.1005	-0.0516	-0.0965
m1	1.0	0.6727	0.6238
m2	0.6727	1.0	0.9574
m3	0.6238	0.9574	1.0
p1	0.5776	0.9858	0.9462
p2	0.5796	0.9778	0.9706
p3	0.4795	0.6439	0.7261
p4	-0.0085	0.5356	0.5923
p5	0.5603	0.8248	0.8528
p6	0.5766	0.9734	0.9753
p7	0.2395	0.2586	0.4227
r1	0.3147	0.3809	0.4081
r2			
r3	0.3147	0.3809	0.4081
r4	0.3147	0.3809	0.4081
r5	0.3147	0.3809	0.4081
r6	0.3147	0.3809	0.4081

Table 95: Correlation between miscellaneous metrics and others for Pretty Printer Library.

	m1	m2	m3
c1	0.199	-0.0044	-0.007
c2	0.0073	-0.0121	-0.0122
c3	0.2215	0.8016	0.7933
c4	0.1287	0.1055	0.1077
c5	0.097	0.1466	0.1463
c6	0.0125	0.373	0.342
d1	0.0021	0.994	0.9947
d2	0.2182	0.094	0.0887
d3	0.1157	0.1398	0.1313
d4	-0.0058	0.9183	0.9392
d5	-0.0139	0.0428	0.0479
d6	-0.0263	0.0744	0.0802
d7	-0.0052	0.7628	0.7668
d8	-0.0069	0.3504	0.3521
d9	-0.0415	0.2238	0.226
d10	-0.0056	0.807	0.8143
d11	-0.0108	0.6533	0.6604
d12	-0.0383	0.4381	0.444
m1	1.0	0.0007	-0.0021
m2	0.0007	1.0	0.997
m3	-0.0021	0.997	1.0
p1	0.5144	0.0037	0.0001
p2	0.5895	0.0001	-0.0028
p3	0.5089	0.0132	0.0109
p4			
p5	0.6214	-0.0097	-0.0108
p6	0.5714	-0.0012	-0.004
p7	0.0751	-0.0079	-0.0047
r1	0.4532	-0.013	-0.0148
r2	0.043	-0.0064	-0.0075
r3	0.4881	-0.012	-0.013
r4	0.3812	-0.0132	-0.0147
r5	0.224	-0.0105	-0.0119
r6	0.1102	-0.009	-0.0097

Table 96: Correlation between miscellaneous metrics and others for Typing Haskell in Haskell.

Appendix F

Tables of Metric Values

		Number of pat- tern variables (<i>p1</i>)	Sum of depth of patterns (<i>p2</i>)	Maximum depth of patterns (<i>p3</i>)	Number of overridden or overriding pat- tern variables (<i>p4</i>)	Number of con- structors in pat- tern (<i>p5</i>)	Pattern size (<i>p6</i>)	Number of wild- cards in pattern (<i>p7</i>)
CGI	Mean	1.961	2.3853	0.9784	0.0476	0.2944	2.6364	0.0563
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	2.8427	3.692	0.8294	0.4581	1.0609	4.378	0.2962
DSP	Mean	4.2359	4.8356	1.2753	0.1319	0.0038	5.304	0.2505
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	2.0	2.0	1.0	0.0	0.0	2.0	0.0
	Stddev	4.8649	5.758	0.984	0.713	0.0617	6.3941	0.9542
GEOMLIB	Mean	3.5408	4.4775	1.5901	0.0697	0.4238	5.5848	0.4152
	Mode	1.0	0.0	1.0	0.0	0.0	0.0	0.0
	Median	2.0	2.0	1.0	0.0	0.0	2.0	0.0
	Stddev	5.1532	6.4419	1.4662	0.5734	0.9659	8.4785	1.2063
PCF	Mean	2.7258	4.0645	0.9032	0.0	0.6452	4.4032	0.1129
	Mode	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	6.778	10.592	1.0733	0.0	2.0168	11.7466	0.4058
CRYPTO	Mean	2.9452	3.1507	0.9726	0.0685	0.0411	3.8356	0.0685
	Mode	0.0	0.0	1.0	0.0	0.0	0.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	4.9377	5.4764	1.0724	0.3444	0.1985	7.5199	0.302
FGL	Mean	2.8974	3.3803	1.4957	0.0107	0.3034	4.594	0.3718
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	1.0	2.0	1.0	0.0	0.0	2.0	0.0
	Stddev	3.7219	4.4713	1.3083	0.1219	0.7085	6.3355	0.8738
GETOPT	Mean	7.2941	15.5882	1.8824	0.0	1.2941	19.2941	3.4706
	Mode	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Median	5.0	12.0	2.0	0.0	0.0	14.0	0.0
	Stddev	9.6514	23.0321	1.745	0.0	2.0513	29.1986	6.9037
PRETTY	Mean	5.0964	6.8313	1.506	0.1566	1.0602	7.7229	0.3253
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	2.0	2.0	1.0	0.0	0.0	2.0	0.0
	Stddev	8.6284	11.6224	1.0223	1.3124	2.026	13.4883	0.9953
HADDOCK	Mean	2.7498	3.4298	1.4252	0.0326	0.3561	4.1572	0.1872
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	5.1464	8.0982	1.4286	0.4408	2.1895	11.0895	1.4681
HAPPY	Mean	4.5192	7.6539	2.0481	0.1058	0.6322	9.3726	0.988
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	1.0	2.0	1.0	0.0	0.0	2.5	0.0
	Stddev	11.8279	15.7755	2.0865	0.8311	1.6847	20.5541	2.4382
HAT	Mean	4.3946	6.2811	1.4726	0.0797	0.7858	8.1158	0.8609
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	14.782	22.1189	1.516	0.7751	2.5456	29.3409	3.9795
HAXML	Mean	3.1271	4.8802	1.5196	0.0856	0.7188	5.6394	0.2812
	Mode	0.0	0.0	1.0	0.0	0.0	0.0	0.0
	Median	1.0	2.0	1.0	0.0	0.0	2.0	0.0
	Stddev	8.0327	13.4373	1.5257	1.5424	1.7459	15.8864	1.0796
HUNIT	Mean	2.5462	3.0084	0.9076	0.084	0.2017	3.3277	0.0504
	Mode	0.0, 1.0	0.0, 1.0	1.0	0.0	0.0	0.0, 1.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	5.3448	6.836	0.8599	0.74	0.8752	7.7385	0.2855
THIH	Mean	1.8937	2.2143	0.9967	0.0	0.1645	2.49	0.0465
	Mode	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Median	1.0	1.0	1.0	0.0	0.0	1.0	0.0
	Stddev	4.0234	5.1265	0.8684	0.0	0.7501	6.3173	0.4365

Table 97: Mean, Mode, Median and Standard Deviation values of pattern metrics.

		Binary recursion (r1)	Number of non-trivial recursive paths (r2)	Number of trivial recursive paths (r3)	Number of recursive paths (r4)	Sum of lengths of recursive paths (r5)	Product of lengths of recursive paths (r6)
CGI	Mean	0.0649	0.0	0.0649	0.0649	0.1299	0.1299
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.2464	0.0	0.2464	0.2464	0.4928	0.4928
DSP	Mean	0.1721	0.0	0.1721	0.1721	0.3442	0.3442
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.3775	0.0	0.3775	0.3775	0.7549	0.7549
GEOMLIB	Mean	0.0622	0.0021	0.0601	0.0622	0.1288	0.1288
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.2416	0.0463	0.2376	0.2416	0.508	0.508
PCF	Mean	0.129	0.0	0.129	0.129	0.2581	0.2581
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.3352	0.0	0.3352	0.3352	0.6705	0.6705
CRYPTO	Mean	0.0411	0.0	0.0411	0.0411	0.0822	0.0822
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.1985	0.0	0.1985	0.1985	0.397	0.397
FGL	Mean	0.1197	0.0	0.1197	0.1197	0.2393	0.2393
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.3246	0.0	0.3246	0.3246	0.6491	0.6491
GETOPT	Mean	0.0	0.0	0.0	0.0	0.0	0.0
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.0	0.0	0.0	0.0	0.0	0.0
PRETTY	Mean	0.2289	0.0	0.2289	0.2289	0.4578	0.4578
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.4201	0.0	0.4201	0.4201	0.8403	0.8403
HADDOCK	Mean	0.2462	1.8301	0.03	1.8601	9.1979	1442270
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.4308	26.4121	0.1765	26.4412	132.116	35478900
HAPPY	Mean	0.0962	0.0	0.0962	0.0962	0.1923	0.1923
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.2948	0.0	0.2948	0.2948	0.5896	0.5896
HAT	Mean	0.0832	0.1851	0.0611	0.2462	1.6304	5478920
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.2762	1.5793	0.2395	1.591	13.2342	76065400
HAXML	Mean	0.0978	0.0159	0.0917	0.1076	0.2958	1.8888
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.297	0.181	0.2969	0.354	1.517	33.825
HUNIT	Mean	0.0084	0.0	0.0084	0.0084	0.0168	0.0168
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.0913	0.0	0.0913	0.0913	0.1826	0.1826
THIH	Mean	0.0565	0.0233	0.0415	0.0648	0.1993	0.3239
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	0.0	0.0	0.0
	Stddev	0.2308	0.1983	0.1995	0.2778	1.0597	2.3277

Table 98: Mean, Mode, Median and Standard Deviation values of recursion metrics.

		Strongly nected component (c1)	con- com- size (c2)	Indegree (c2)	Outdegree (c3)	Depth (c4)	Width (c5)	Arc-to-node ratio (c6)
CGI	Mean	1.0476	1.6796	4.658	3.8788	18.7922	1.3599	
	Mode	1.0	0.0	0.0	4.0	1.0	0.0	
	Median	1.0	1.0	4.0	4.0	13.0	1.4706	
	Stddev	0.213	3.7372	4.39	2.6682	28.3746	0.856	
DSP	Mean	1.0057	1.13	5.8317	4.065	23.1434	2.2041	
	Mode	1.0	0.0	0.0	3.0	1.0	0.0	
	Median	1.0	1.0	6.0	3.0	12.0	2.3333	
	Stddev	0.0755	2.734	3.959	3.1815	38.8046	1.2908	
GEOMLIB	Mean	1.03	2.0761	5.8573	4.0407	22.9571	1.4208	
	Mode	1.0	0.0	0.0	2.0	1.0	0.0	
	Median	1.0	1.0	5.0	3.0	10.0	1.5357	
	Stddev	0.2763	5.3606	4.8649	2.9803	40.3149	0.8945	
PCF	Mean	1.3548	1.1774	4.0807	3.8065	22.0806	0.91	
	Mode	1.0	0.0	0.0	1.0	1.0	0.0	
	Median	1.0	0.0	3.0	2.0	3.5	1.15	
	Stddev	0.8249	2.0831	5.8152	3.7582	32.5067	0.8406	
CRYPTO	Mean	1.0274	1.7808	4.0411	4.6712	20.5068	1.967	
	Mode	1.0	1.0	2.0	2.0	1.0	2.0	
	Median	1.0	1.0	4.0	3.0	7.0	2.2	
	Stddev	0.2325	2.0423	3.3164	4.5482	27.745	1.0263	
FGL	Mean	1.1175	2.25	6.1688	4.5556	19.5812	1.5997	
	Mode	1.0	0.0	6.0	2.0	1.0	0.0	
	Median	1.0	1.0	6.0	4.0	11.0	1.7292	
	Stddev	0.5857	7.6964	4.8806	2.7455	26.4928	0.8117	
GETOPT	Mean	1.2353	2.1765	5.2941	4.2941	26.7059	1.1723	
	Mode	1.0	1.0	0.0	1.0, 2.0	1.0	0.0	
	Median	1.0	2.0	6.0	2.0	6.0	1.5	
	Stddev	0.9412	1.917	4.0112	3.9372	33.3233	0.8103	
PRETTY	Mean	1.2651	2.6386	5.8554	4.4579	29.6265	1.279	
	Mode	1.0	0.0	4.0	4.0	9.0	1.0909	
	Median	1.0	1.0	5.0	4.0	9.0	1.1875	
	Stddev	0.7458	5.8979	5.5778	2.8804	40.8977	0.5371	
HADDOCK	Mean	75.9222	2.3856	5.1907	6.6907	503.116	1.4632	
	Mode	1.0	1.0	2.0	2.0	2444.0	2.09396	
	Median	1.0	1.0	4.0	3.0	8.0	1.4	
	Stddev	150.865	9.4334	10.0403	6.9637	966.279	0.709	
HAPPY	Mean	1.0289	2.5697	5.762	4.5649	44.1875	1.3382	
	Mode	1.0	1.0	3.0	2.0	1.0	0.0	
	Median	1.0	1.0	4.5	3.0	10.0	1.3415	
	Stddev	0.2483	6.0558	5.333	3.6302	116.323	0.6887	
HAT	Mean	6.8038	2.4628	6.2899	4.2969	64.2055	1.5903	
	Mode	1.0	0.0	0.0	1.0	1.0	0.0	
	Median	1.0	1.0	4.0	3.0	6.0	1.6711	
	Stddev	29.6067	7.3198	9.5026	4.5608	178.178	1.1156	
HAXML	Mean	1.3325	3.2604	6.4536	4.3741	49.7616	1.3253	
	Mode	1.0	1.0	0.0	1.0	1.0	0.0	
	Median	1.0	1.0	5.0	4.0	14.0	1.5	
	Stddev	1.6112	10.7851	6.4532	3.4732	100.743	0.8541	
HUNIT	Mean	1.0	1.21	4.4454	2.8151	14.5378	1.4238	
	Mode	1.0	1.0	0.0	1.0	1.0	0.0	
	Median	1.0	1.0	3.0	2.0	4.0	1.8333	
	Stddev	0.0	1.4603	5.3995	2.0166	24.8027	1.1711	
THH	Mean	1.1246	2.3937	7.7375	3.8372	35.9734	2.38	
	Mode	1.0	0.0	0.0	1.0,4.0	1.0	0.0	
	Median	1.0	1.0	6.0	3.0	11.0	1.875	
	Stddev	0.7794	6.0024	11.7023	3.4737	87.4426	4.1931	

Table 99: Mean, Mode, Median and Standard Deviation values of callgraph metrics.

		Distance by the sum of the number of scopes (d1)	Distance by the maximum number of scopes (d2)	Distance by the average number of scopes (d3)	Distance by the sum of the number of declarations in scope (d4)	Distance by the maximum of the number of declarations in scope (d5)	Distance by the average number of declarations in scope (d6)
CGI	Mean	13.5455	2.4978	1.4026	47.6883	16.8139	7.2597
	Mode	0.0	0.0	2.0	0.0	0.0	0.0
	Median	5.0	3.0	2.0	17.0	7.0	3.0
	Stddev	25.5142	2.0191	1.1388	76.7027	19.8766	9.3555
DSP	Mean	69.0727	3.7323	1.7667	298.885	20.2218	10.109
	Mode	0.0	6.0	2.0	0.0	0.0	0.0
	Median	32.0	4.0	2.0	86.0	14.0	6.0
	Stddev	106.59	2.2333	1.0545	470.231	23.6458	12.6907
GEOMLIB	Mean	29.8026	3.088	1.5129	129.912	20.9742	9.2436
	Mode	0.0	4.0	2.0	0.0	0.0	0.0
	Median	8.0	3.0	2.0	40.0	16.0	6.0
	Stddev	72.2214	2.4957	1.1732	339.775	21.6627	10.9142
PCF	Mean	12.2258	1.9194	0.7097	43.6452	10.2903	3.4194
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	3.0	2.0	1.0	3.0	2.0	1.0
	Stddev	42.0941	1.9617	0.7908	168.895	11.5051	4.9008
CRYPTO	Mean	26.7671	2.8082	1.4247	81.5069	10.3425	4.5616
	Mode	0.0	0.0	2.0	0.0	0.0	0.0
	Median	6.0	3.0	2.0	16.0	6.0	2.0
	Stddev	54.4521	2.333	1.0969	172.23	11.6657	5.0123
FGL	Mean	16.2179	2.9039	1.4979	79.1154	24.4231	11.4402
	Mode	0.0	4.0	2.0	0.0	0.0	0.0
	Median	8.0	3.0	1.5	42.5	16.0	5.5
	Stddev	25.8055	2.0655	1.10264	119.253	24.4452	14.6126
GETOPT	Mean	31.4118	2.9412	1.1765	93.0	13.0	5.3529
	Mode	0.0	0.0, 3.0	2.0	0.0	0.0	0.0
	Median	19.0	3.0	1.0	62.0	15.0	7.0
	Stddev	37.2228	2.2353	0.9226	117.982	9.2291	4.3917
PRETTY	Mean	17.241	2.3615	1.2651	85.0482	32.747	7.5301
	Mode	2.0	2.0	1.0	2.0	2.0	2.0
	Median	6.0	2.0	1.0	9.0	3.0	2.0
	Stddev	36.6845	1.3041	0.6601	169.566	38.7267	11.8924
HADDOCK	Mean	17.412	2.9695	1.5956	444.306	265.872	111.686
	Mode	4.0	2.0	2.0	4.0	2.0	2.0
	Median	5.0	3.0	2.0	7.0	3.0	2.0
	Stddev	63.5327	1.8181	0.8604	1027.23	468.627	219.189
HAPPY	Mean	47.1562	2.6827	1.2909	365.584	22.6755	8.6683
	Mode	0.0	1.0	1.0	0.0	1.0	1.0
	Median	4.0	2.0	1.0	5.0	2.0	2.0
	Stddev	308.784	2.8046	0.958	3515.93	34.9214	16.6081
HAT	Mean	30.6746	2.4983	1.383	404.342	53.8097	24.4197
	Mode	0.0	0.0	2.0	0.0	0.0	0.0
	Median	4.0	2.0	2.0	6.0	2.0	2.0
	Stddev	116.948	2.1432	1.1093	1702.23	87.9073	47.4474
HAXML	Mean	27.4535	2.7249	1.3961	281.286	53.1822	22.4095
	Mode	0.0	0.0	2.0	0.0	0.0	0.0
	Median	8.0	3.0	1.0	42.0	27.0	6.0
	Stddev	80.3673	2.2028	1.1187	618.783	70.7306	32.0744
HUNIT	Mean	27.5042	2.4202	1.2689	139.303	14.5042	8.0924
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	4.0	2.0	1.0	5.0	2.0	2.0
	Stddev	64.6817	2.6106	1.2482	373.844	20.2143	13.3744
THIH	Mean	129.924	2.5066	1.5183	3270.71	81.3987	47.897
	Mode	0.0	4.0	0.0	0.0	0.0	0.0
	Median	8.0	3.0	2.0	39.5	11.0	4.5
	Stddev	1195.4	2.1844	1.3221	47018.5	114.159	74.1336

Table 100: Mean, Mode, Median and Standard Deviation values of distance metrics (Part 1 of 2).

		Distance by the sum of the num- ber of source lines ($d7$)	Distance by the maximum num- ber of source lines ($d8$)	Distance by the average number of source lines ($d9$)	Distance by the sum of the num- ber of parse tree nodes ($d10$)	Distance by the maximum num- ber of parse tree nodes ($d11$)	Distance by the average number of parse tree nodes ($d12$)
CGI	Mean	91.4978	50.4719	24.2078	917.68	491.056	243.502
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	9.0	8.0	2.0	127.0	80.0	42.0
	Stddev	207.131	83.2839	43.2679	1756.29	691.87	408.912
DSP	Mean	115.998	48.4245	7.5545	2677.07	801.379	164.736
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	16.0	4.0	1.0	485.0	155.0	62.0
	Stddev	224.127	89.0246	14.854	5038.73	1481.0	271.31
GEOMLIB	Mean	96.7425	36.22	9.1856	1938.27	639.535	176.468
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	3.0	2.0	0.0	120.0	62.0	33.0
	Stddev	299.156	78.9768	23.671	5831.79	1339.96	405.918
PCF	Mean	14.7903	3.1452	0.7258	571.9194	90.8387	27.6613
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	12.0	9.5	8.0
	Stddev	59.8161	8.1237	1.7976	2600.3784	232.2869	56.118
CRYPTO	Mean	49.8219	18.6575	6.1507	1213.548	364.17807	122.274
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	1.0	1.0	0.0	46.0	28.0	16.0
	Stddev	142.697	35.2312	13.9418	3373.9866	688.7945	250.3776
FGL	Mean	91.7628	60.3291	23.3953	1475.1389	857.5021	352.6581
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	5.0	3.0	0.0	209.5	79.0	38.0
	Stddev	196.578	114.9033	54.3042	2846.5662	1455.517	713.2242
GETOPT	Mean	36.9412	11.9412	1.0588	1341.6471	402.0588	49.4706
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	94.0	35.0	23.0
	Stddev	59.5191	22.5818	2.8588	1950.6168	672.498	80.2211
PRETTY	Mean	310.048	105.9277	47.8072	3860.3613	1097.976	430.7229
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	140.0	122.0	11.0	1108.0	1045.0	145.0
	Stddev	575.902	111.0424	60.672	8240.875	1375.1448	508.3607
HADDOCK	Mean	2108.14	906.6882	479.6211	19150.428	8301.375	4334.1724
	Mode	0.0	0.0	0.0	8.0	8.0	8.0
	Median	292.0	210.5	108.0	2922.0	2102.0	1194.0
	Stddev	16640.5	1208.7306	701.5303	129811.4	10742.999	6193.4043
HAPPY	Mean	1100.65	159.9087	84.125	11688.863	1682.6947	841.5697
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	27.0	10.0	2.0	801.0	242.5	68.5
	Stddev	11629.1	231.1287	147.8707	120231.33	2381.0564	1451.2504
HAT	Mean	738.112	113.103	52.1583	12451.8125	1743.8656	761.3242
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	8.0	5.0	1.0	245.0	100.0	42.5
	Stddev	4649.32	270.1736	96.0151	79907.45	4250.184	1458.2773
HAXML	Mean	1369.93	217.8447	123.9328	13345.854	2272.3594	1199.2201
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	34.0	26.5	5.0	722.0	431.0	109.0
	Stddev	4997.49	286.5177	190.8073	39982.402	2958.1797	1778.6891
HUNIT	Mean	101.882	35.395	9.7647	1561.1848	470.5882	143.6218
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	0.0	0.0	0.0	26.0	19.0	14.0
	Stddev	287.644	78.293	26.1761	4245.846	1088.9442	369.1778
THIH	Mean	6306.53	143.2359	72.8422	244017.2	2952.1396	1511.8323
	Mode	0.0	0.0	0.0	0.0	0.0	0.0
	Median	4.0	3.0	1.0	181.5	92.0	52.0
	Stddev	129874.4	276.8581	160.0003	4907302.0	7274.8975	3624.7903

Table 101: Mean, Mode, Median and Standard Deviation values of distance metrics (Part 2 of 2).

		Pathcount ($m1$)	Number of operands ($m2$)	Number of operators ($m3$)
CGI	Mean	1.2035	8.658	2.7619
	Mode	1.0	0.0	0.0
	Median	1.0	5.0	2.0
	Stddev	0.9657	12.5398	3.5517
DSP	Mean	1.3614	29.8298	9.7361
	Mode	1.0	0.0	1.0
	Median	1.0	18.0	6.0
	Stddev	1.2481	37.2392	13.6341
GEOMLIB	Mean	1.294	16.5418	6.0891
	Mode	1.0	0.0	1.0
	Median	1.0	9.0	3.0
	Stddev	1.0104	26.0182	9.8312
PCF	Mean	1.7258	13.0161	4.4355
	Mode	1.0	0.0	0.0
	Median	1.0	6.0	1.0
	Stddev	2.522	34.0159	8.5187
CRYPTO	Mean	1.0	39.9315	5.9041
	Mode	1.0	0.0	0.0
	Median	1.0	18.0	4.0
	Stddev	0.4682	61.5909	9.0798
FGL	Mean	1.3205	12.8526	5.5214
	Mode	1.0	0.0	0.0
	Median	1.0	7.0	4.0
	Stddev	0.9562	16.7238	7.3253
GETOPT	Mean	1.4706	29.5882	15.5294
	Mode	1.0	0.0	0.0
	Median	1.0	27.0	9.0
	Stddev	1.1437	34.9656	22.5078
PRETTY	Mean	2.2651	16.8313	5.253
	Mode	1.0	3.0	1.0
	Median	1.0	8.0	2.0
	Stddev	3.7323	27.2727	8.0238
HADDOCK	Mean	1.6027	11.3901	3.674
	Mode	1.0	3.0	1.0
	Median	1.0	4.0	1.0
	Stddev	11.9111	29.7117	11.5678
HAPPY	Mean	223.4423	22.1202	8.3245
	Mode	1.0	0.0	1.0
	Median	1.0	6.0	3.0
	Stddev	4512.9995	83.7473	19.6534
HAT	Mean	2.0437	21.6659	7.6024
	Mode	1.0	3.0	1.0
	Median	1.0	6.0	2.0
	Stddev	6.1626	66.3639	22.4257
HAXML	Mean	3.7213	19.5513	6.824
	Mode	1.0	0.0	0.0
	Median	1.0	9.0	3.0
	Stddev	49.6457	42.7052	14.862
HUNIT	Mean	1.0252	16.8067	4.8235
	Mode	1.0	0.0	0.0
	Median	1.0	5.0	2.0
	Stddev	0.6144	27.8804	8.5687
THH	Mean	1.1728	83.8787	31.9751
	Mode	1.0	0.0	0.0
	Median	1.0	10.0	4.0
	Stddev	1.4785	699.0071	284.9418

Table 102: Mean, Mode, Median and Standard Deviation values of miscellaneous metrics.

Appendix G

Histograms of Metric Values

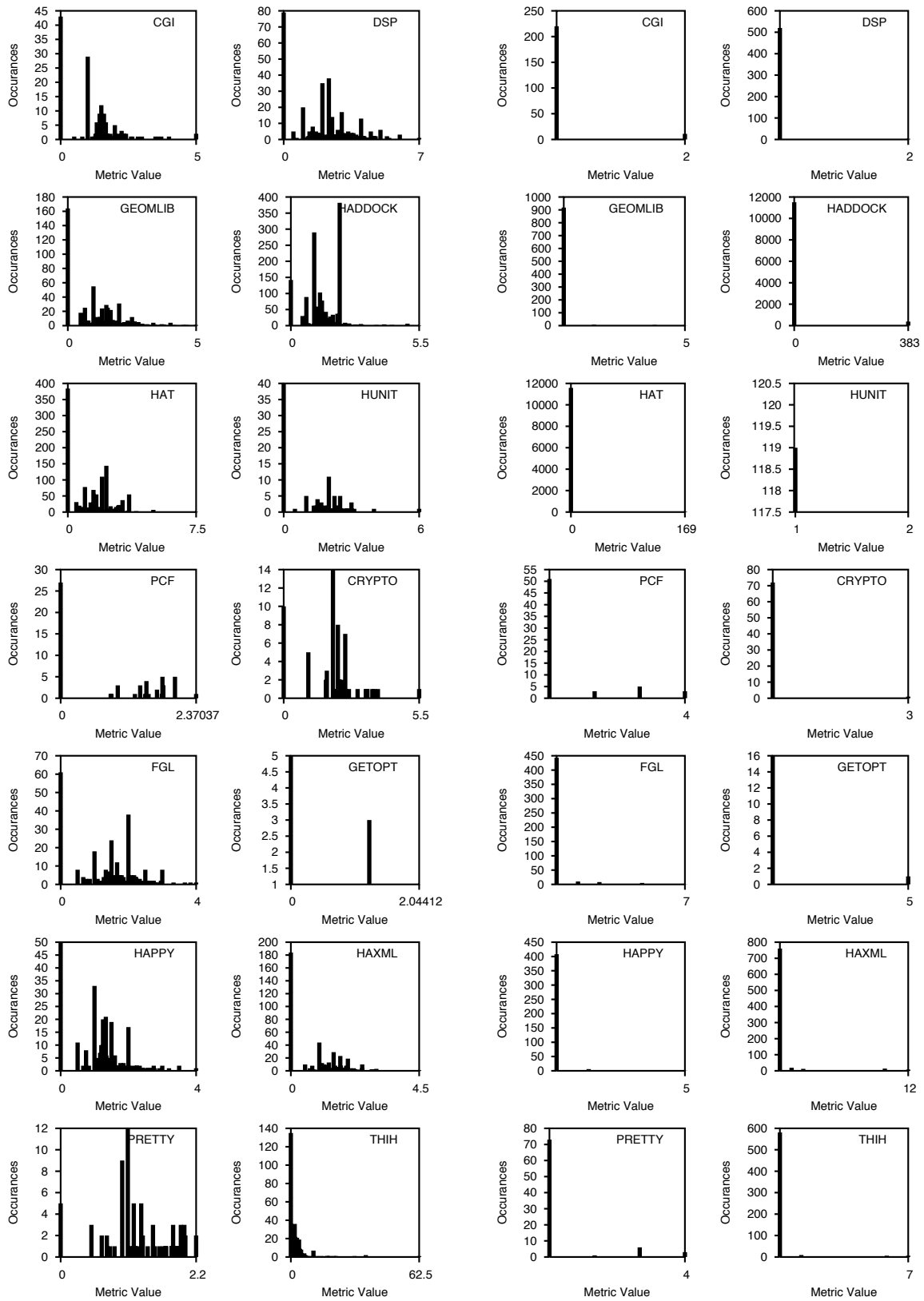


Figure 62: Histograms of “Arc-to-node ra-
tio” values.

Figure 63: Histograms of “Strongly con-
nected component size” values.

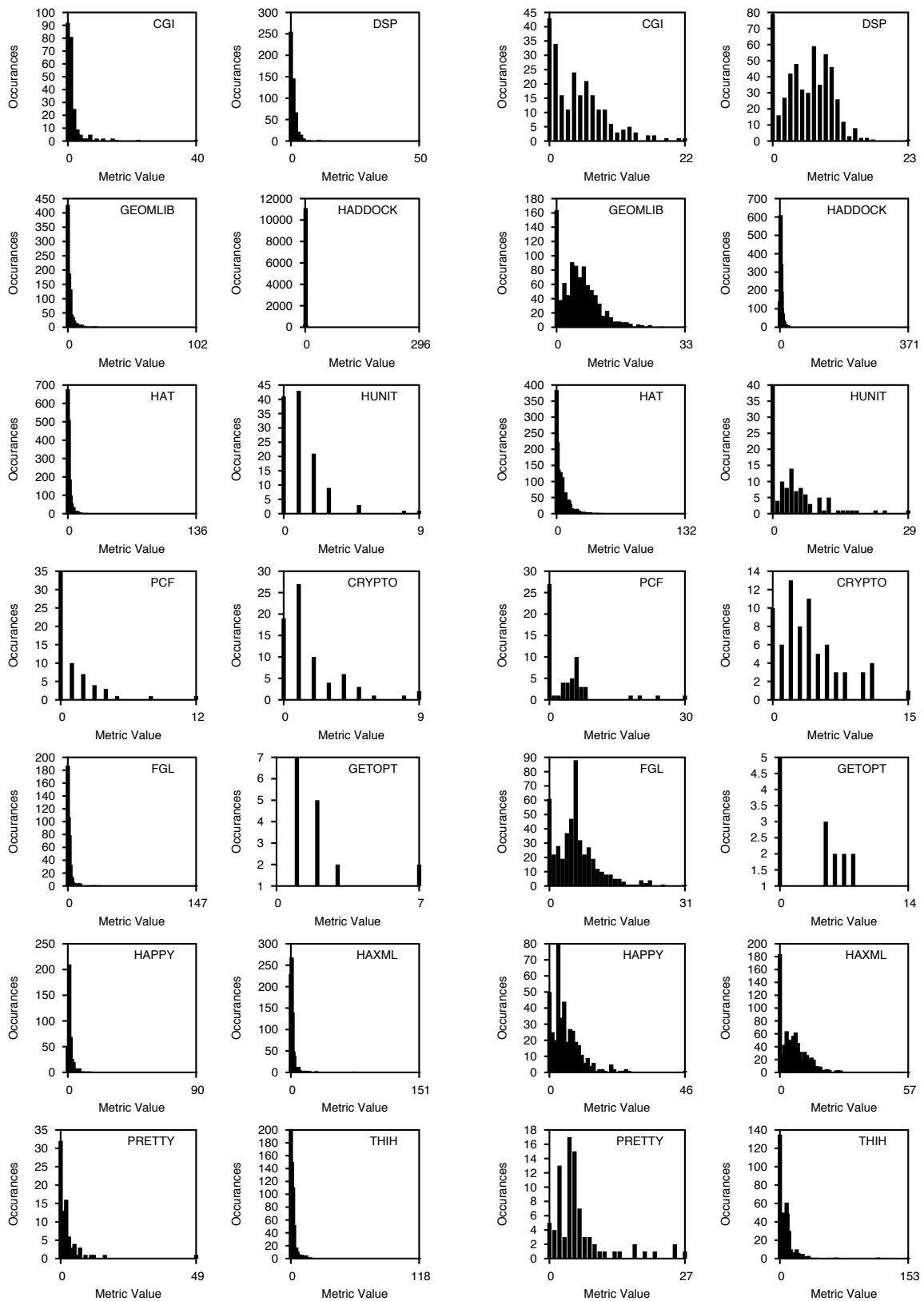


Figure 64: Histograms of “Indegree” values. Figure 65: Histograms of “Outdegree” values.

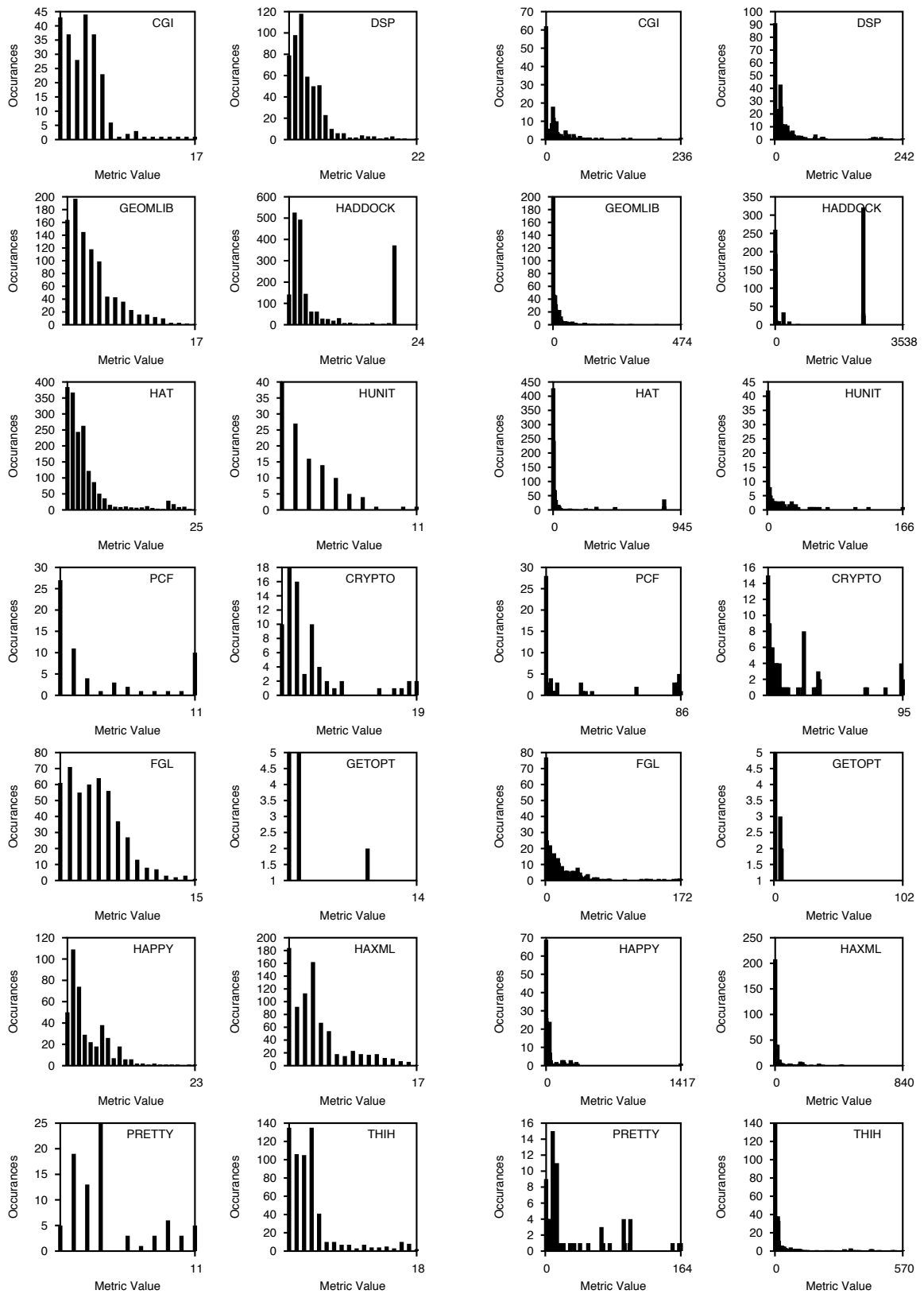


Figure 66: Histograms of “Depth” values.

Figure 67: Histograms of “Width” values.

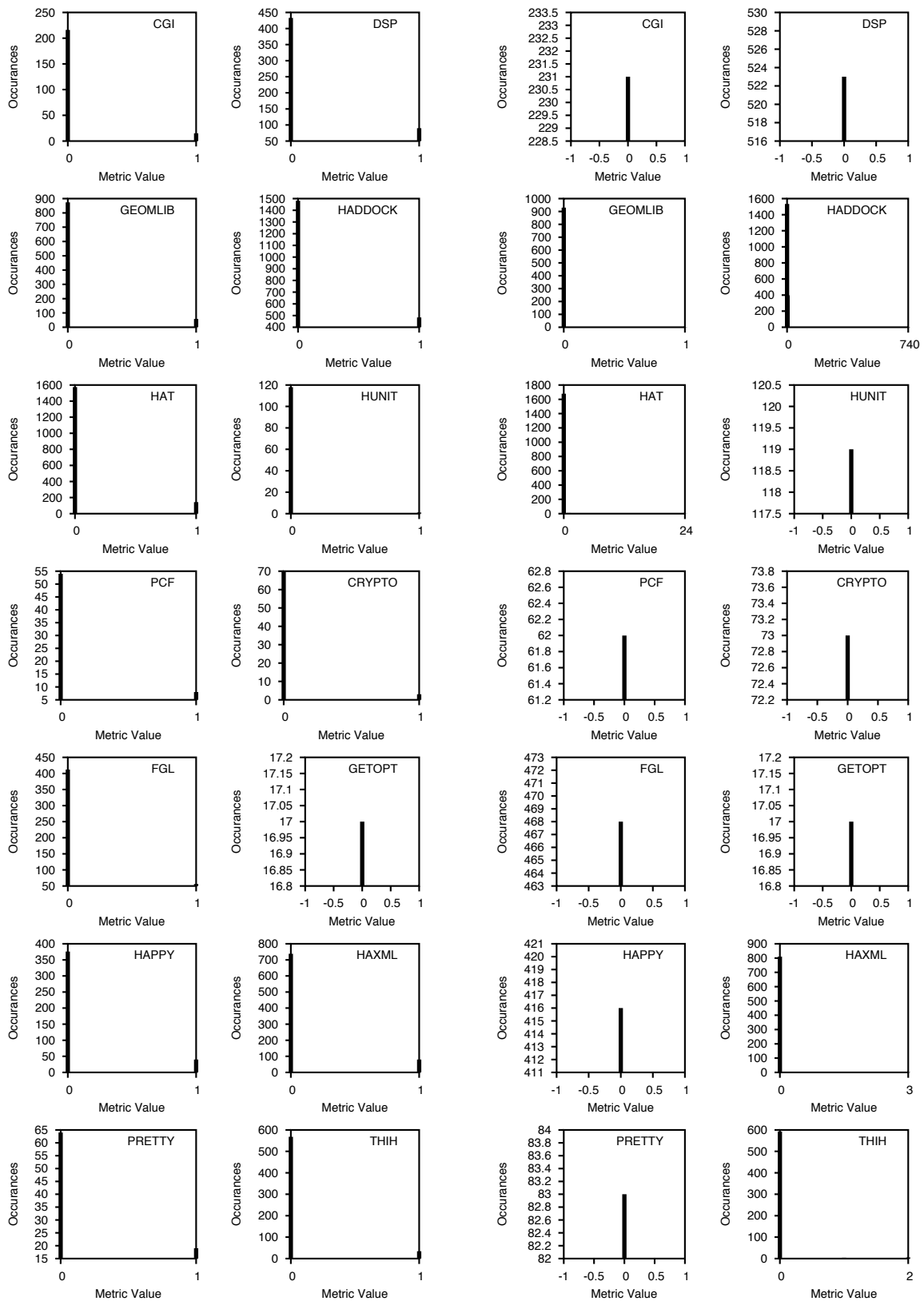


Figure 68: Histograms of “Binary recursion” Figure 69: Histograms of “Number of non-trivial recursive paths” values.

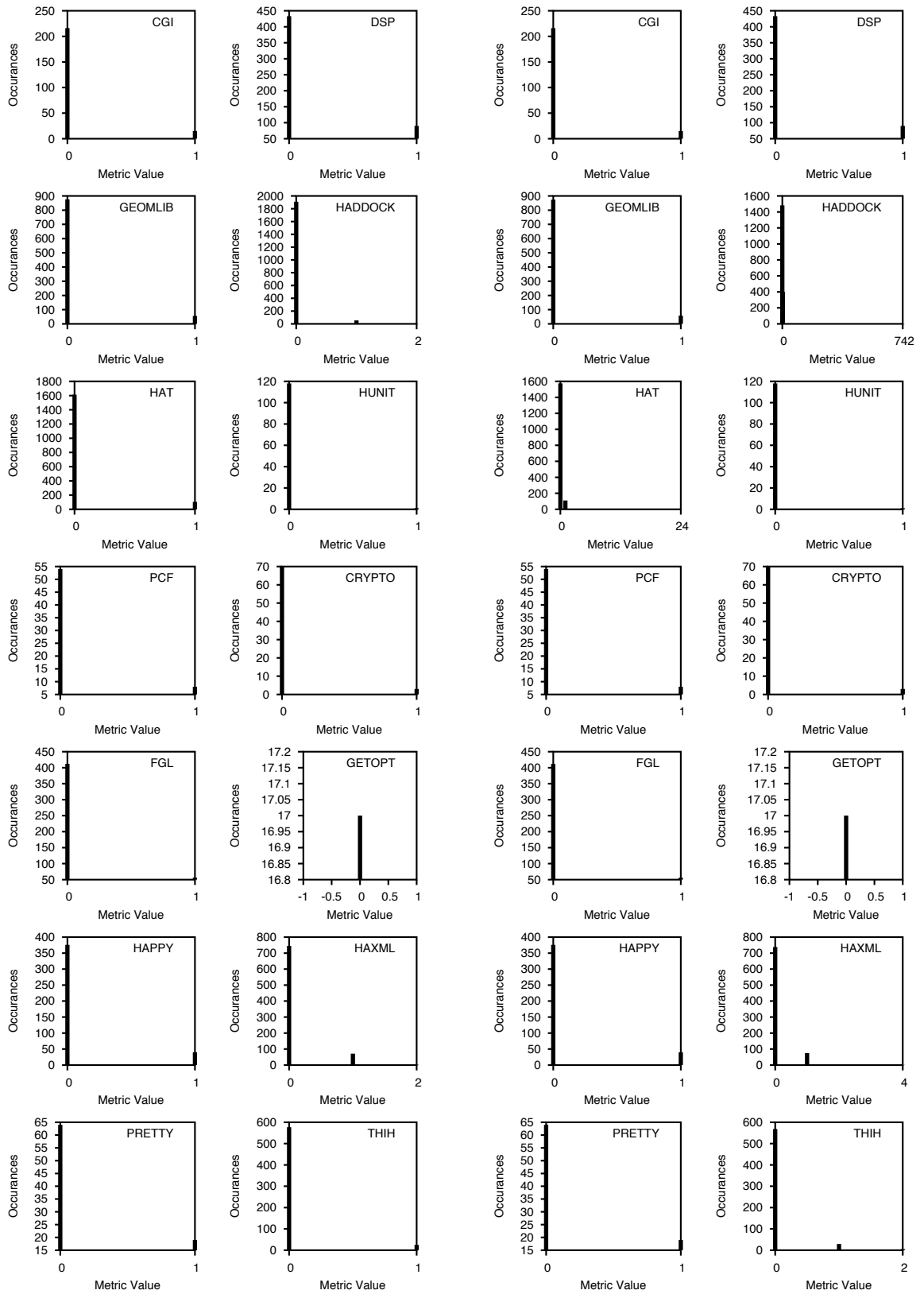


Figure 70: Histograms of “Number of trivial recursive paths” values.

Figure 71: Histograms of “Number of recursive paths” values.

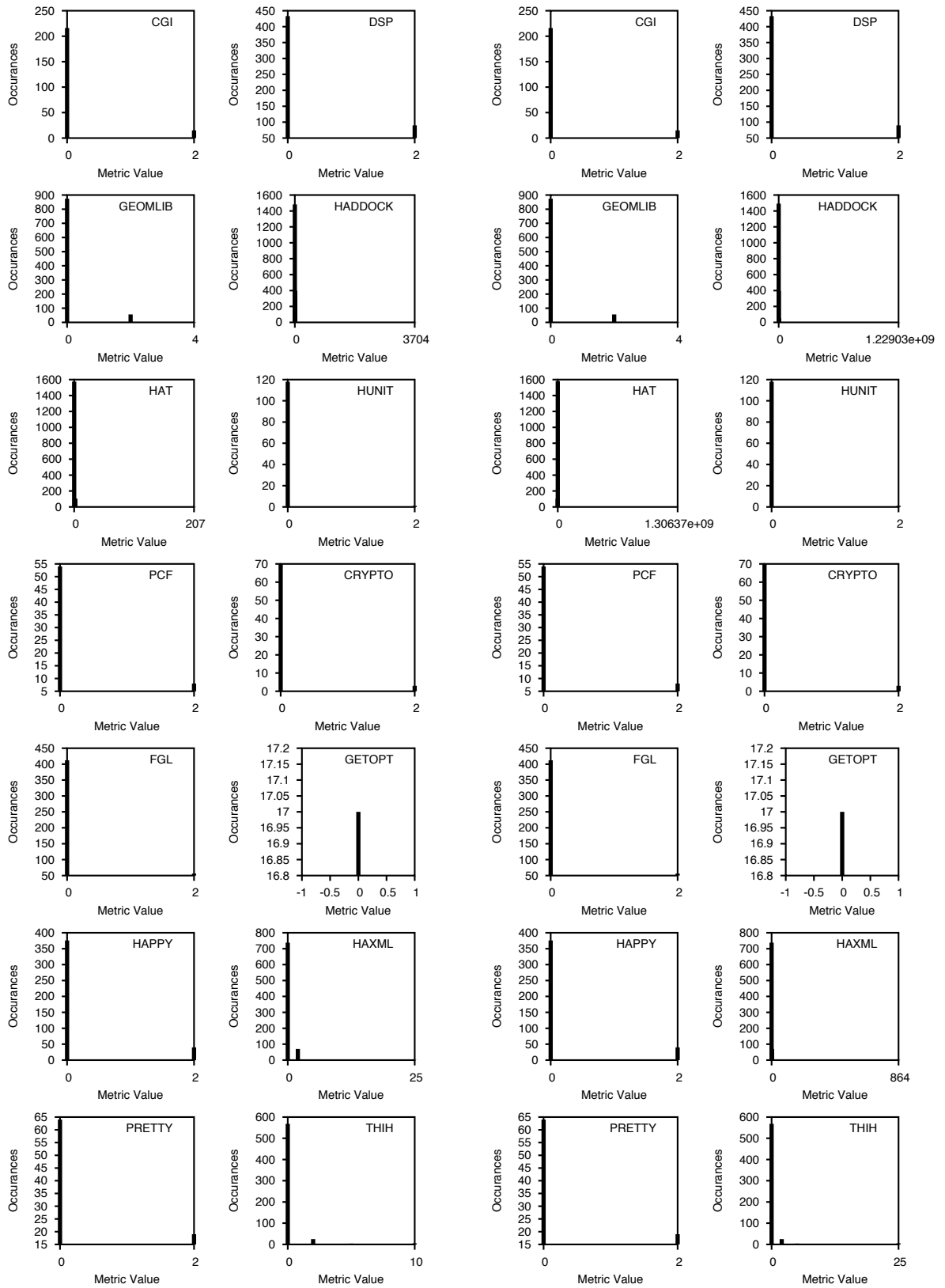


Figure 72: Histograms of “Sum of lengths of recursive paths” values. Figure 73: Histograms of “Product of lengths of recursive paths” values.

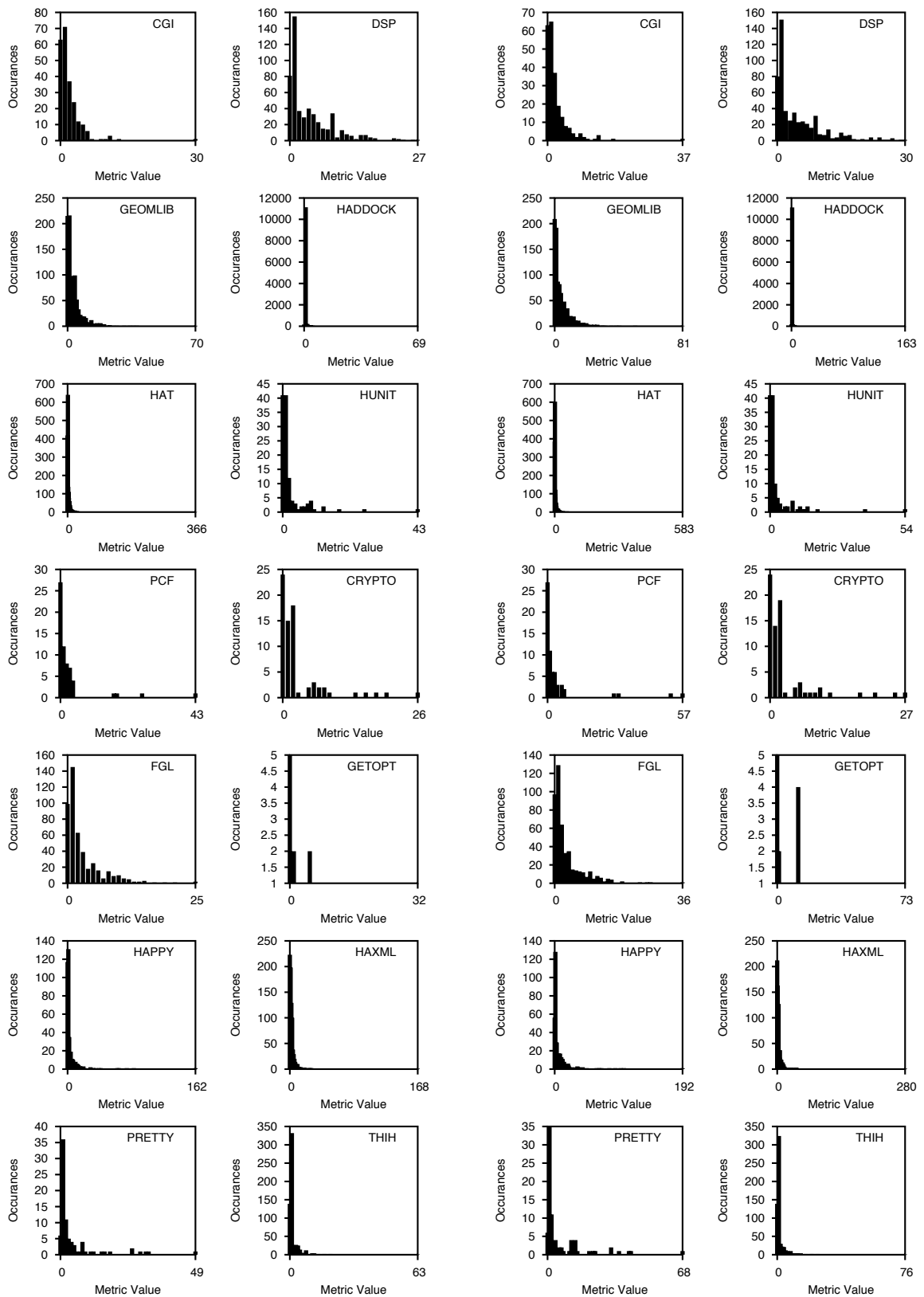


Figure 74: Histograms of “Number of pat-tern variables” values.

Figure 75: Histograms of “Sum of depth of patterns” values.

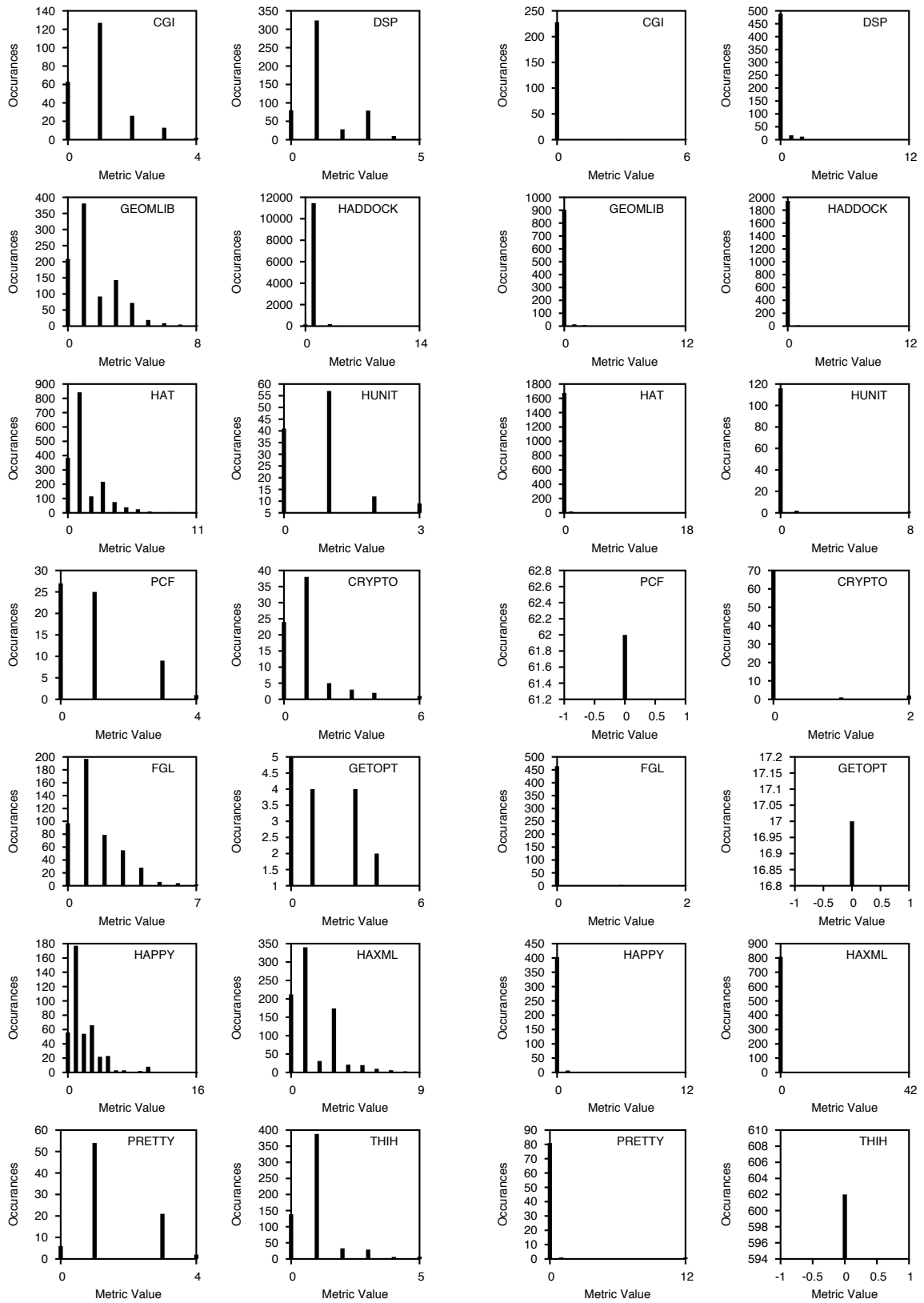


Figure 76: Histograms of “Maximum depth

Figure 77: Histograms of “Number of over-
ridden or overriding pattern variables” val-
ues.

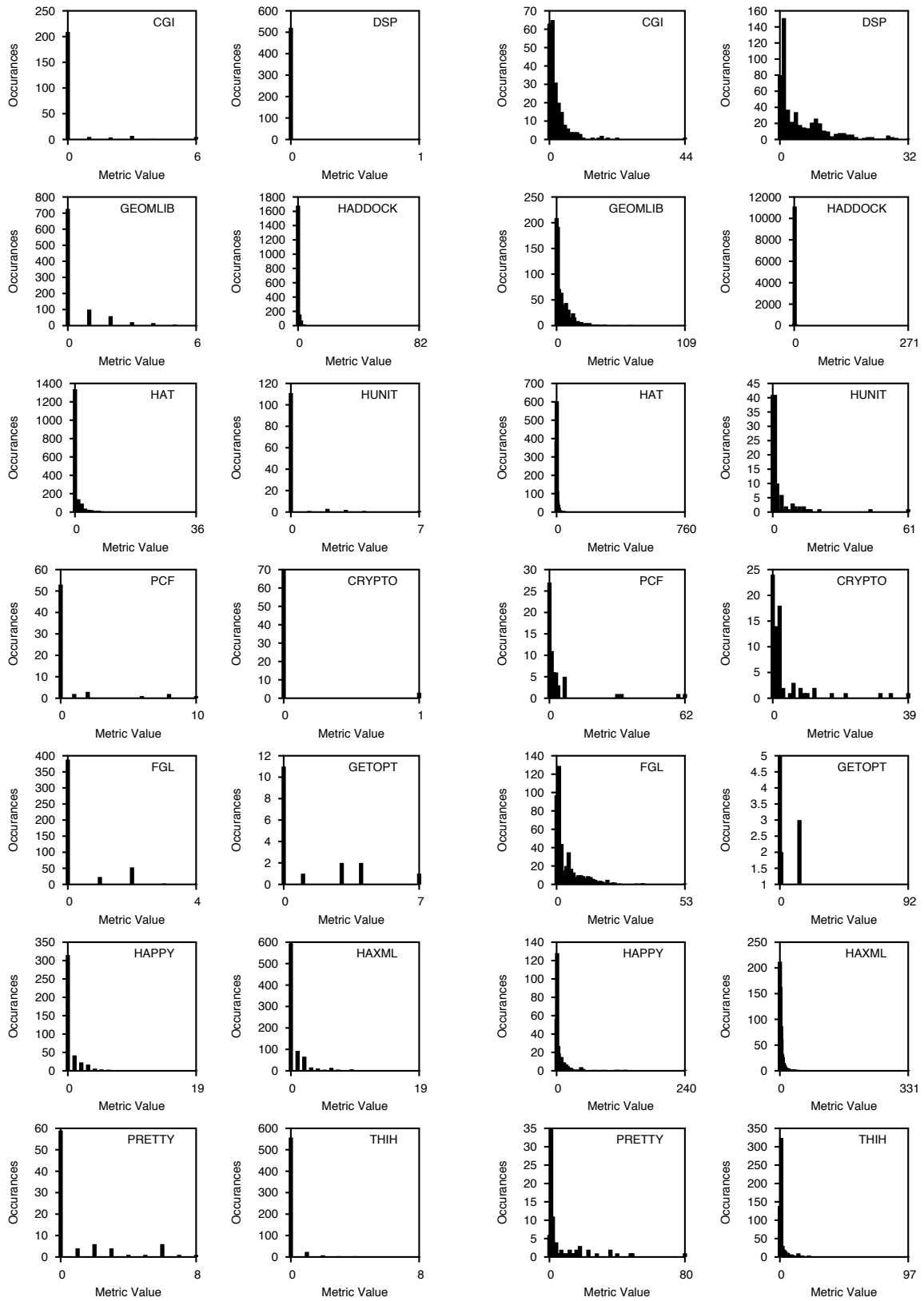


Figure 78: Histograms of “Number of con-
structors in pattern” values.

Figure 79: Histograms of “Pattern size” val-
ues.

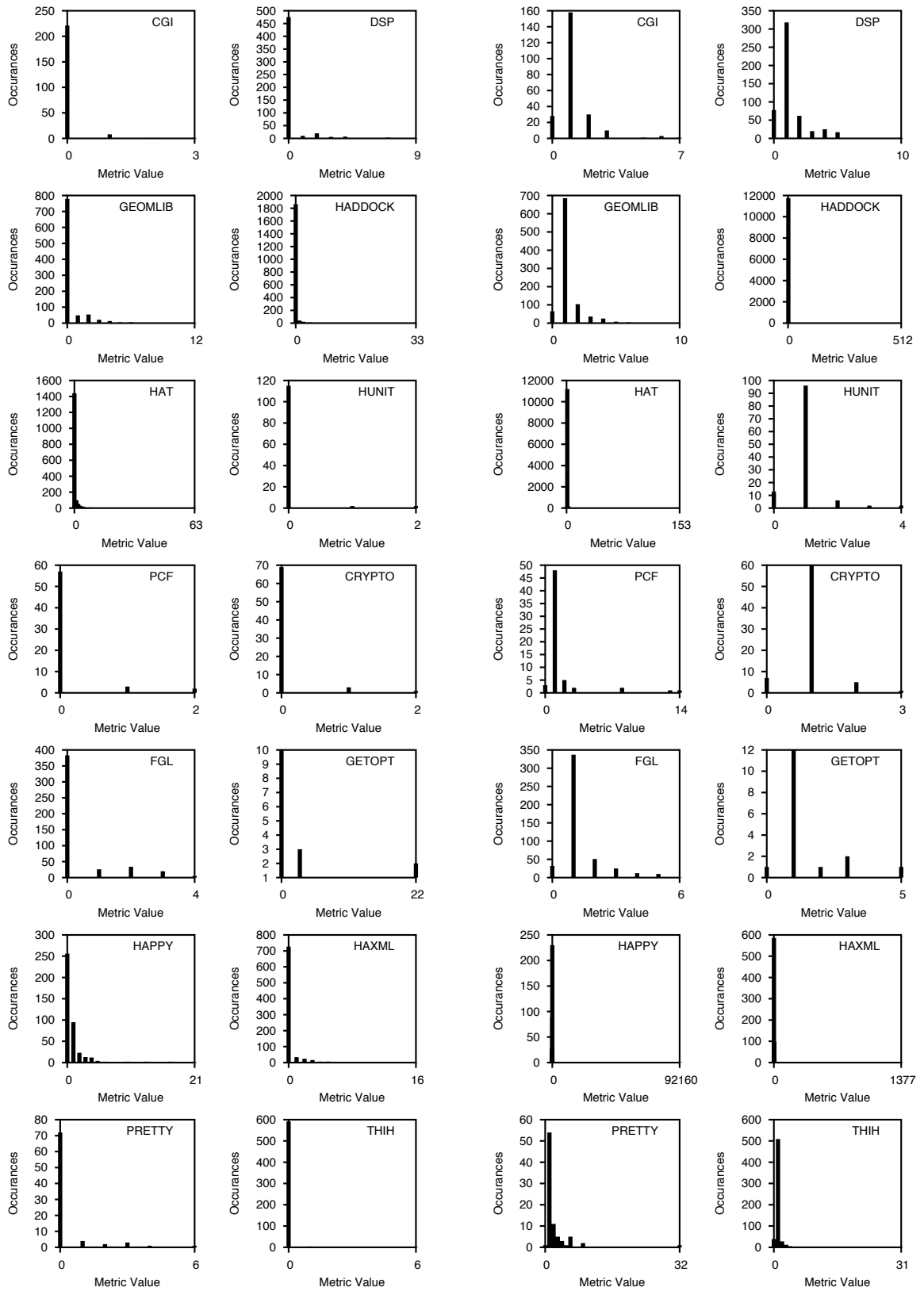


Figure 80: Histograms of “Number of wild-cards in pattern” values.

Figure 81: Histograms of “Pathcount” values.

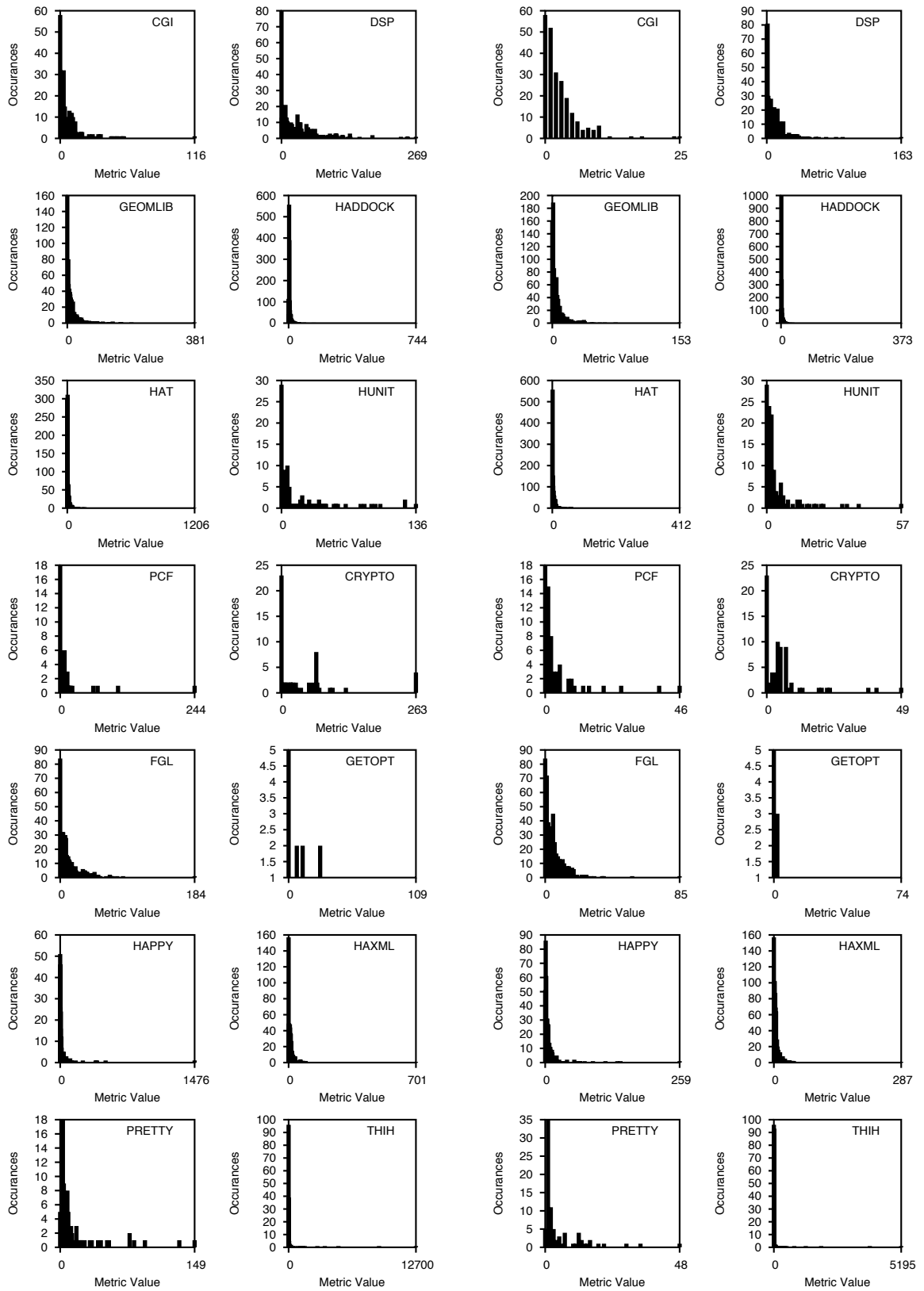


Figure 82: Histograms of “Number of operands” values.

Figure 83: Histograms of “Number of operators” values.

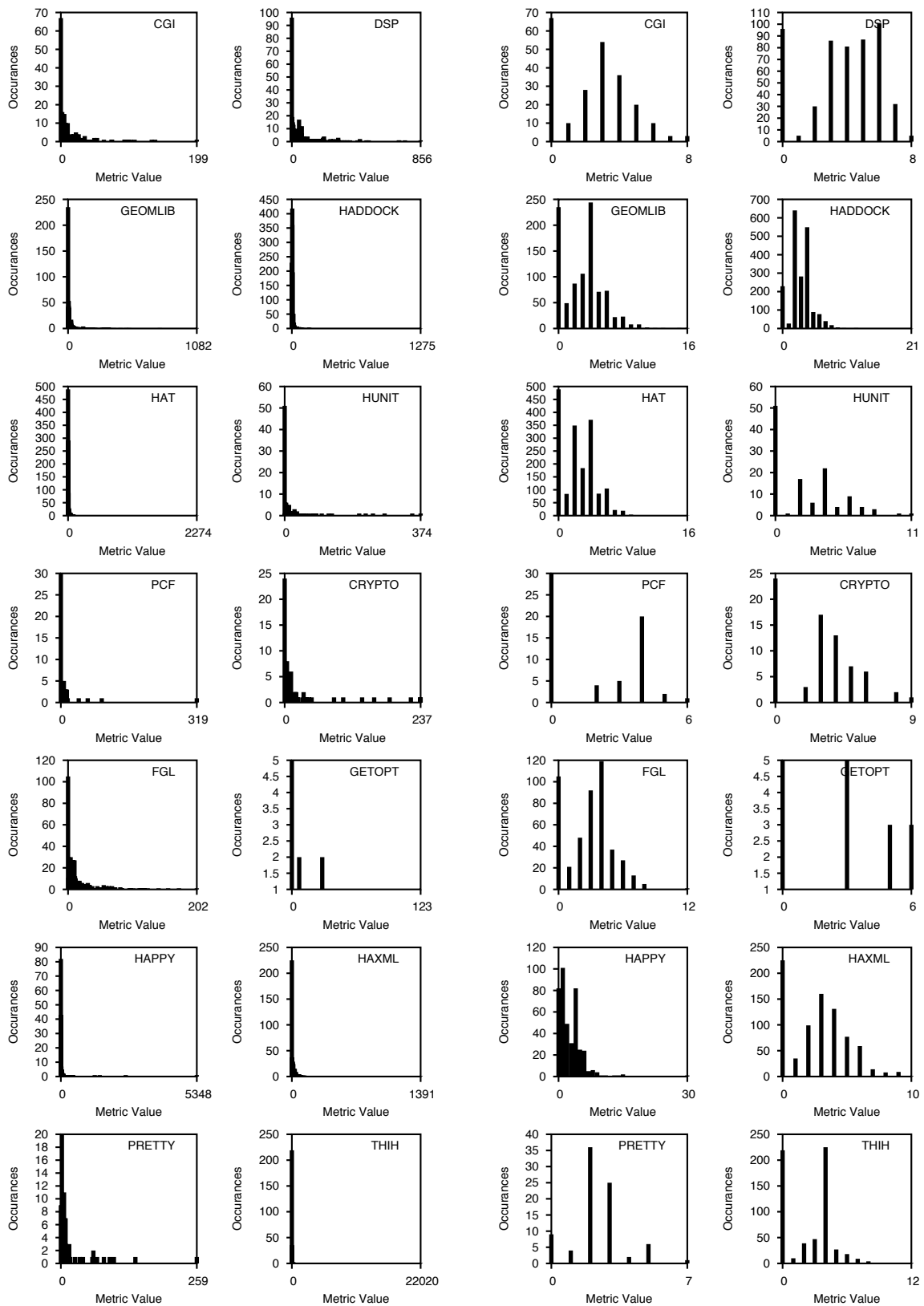


Figure 84: Histograms of “Distance by the sum of the number of scopes” values.

Figure 85: Histograms of “Distance by the maximum number of scopes” values.

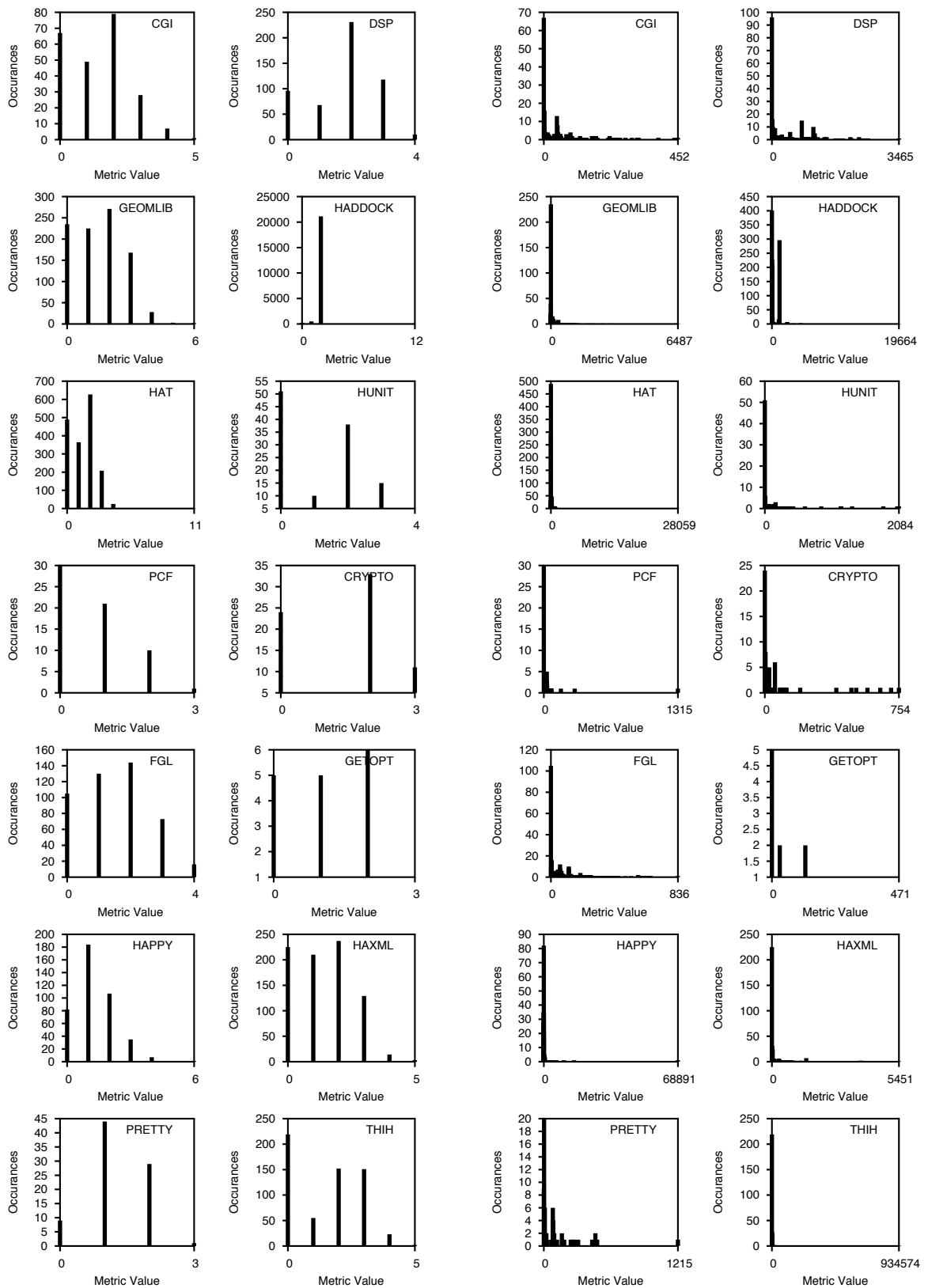


Figure 86: Histograms of “Distance by the average number of scopes” values.

Figure 87: Histograms of “Distance by the sum of the number of declarations in scope” values.

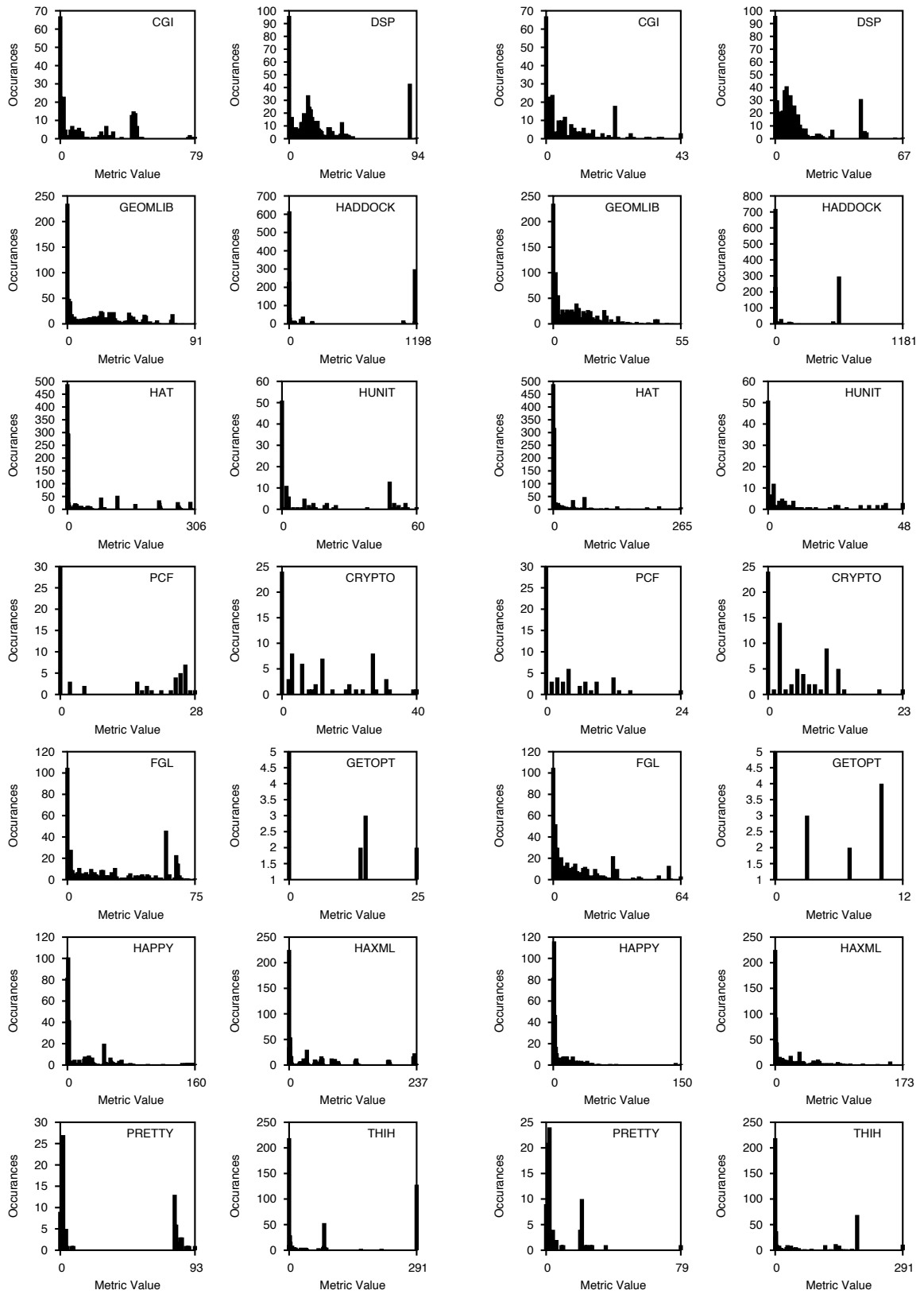


Figure 88: Histograms of “Distance by the maximum of the number of declarations in average number of declarations in scope” values. Figure 89: Histograms of “Distance by the maximum of the number of declarations in average number of declarations in scope” values.

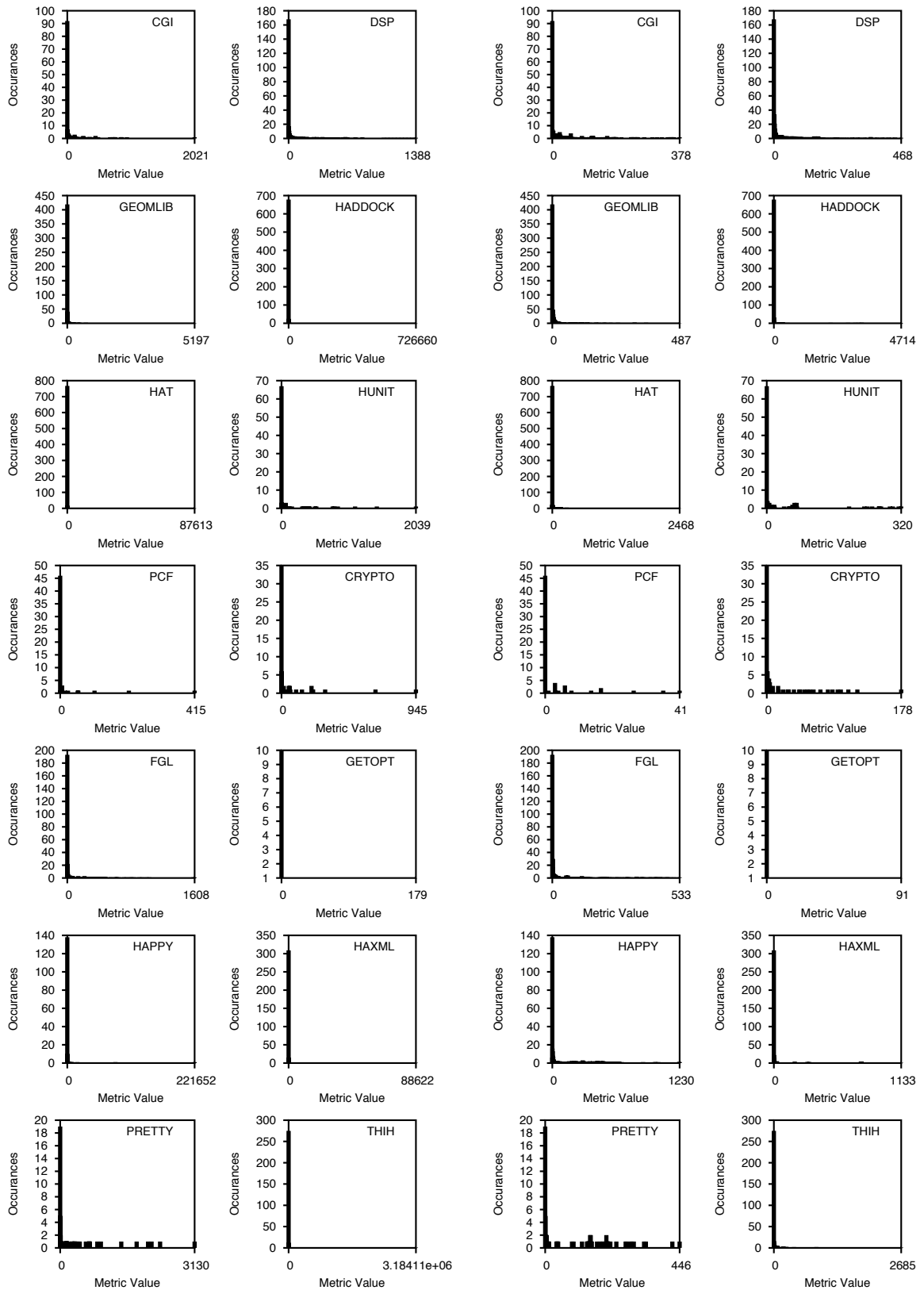


Figure 90: Histograms of “Distance by the sum of the number of source lines” values. Figure 91: Histograms of “Distance by the maximum number of source lines” values.

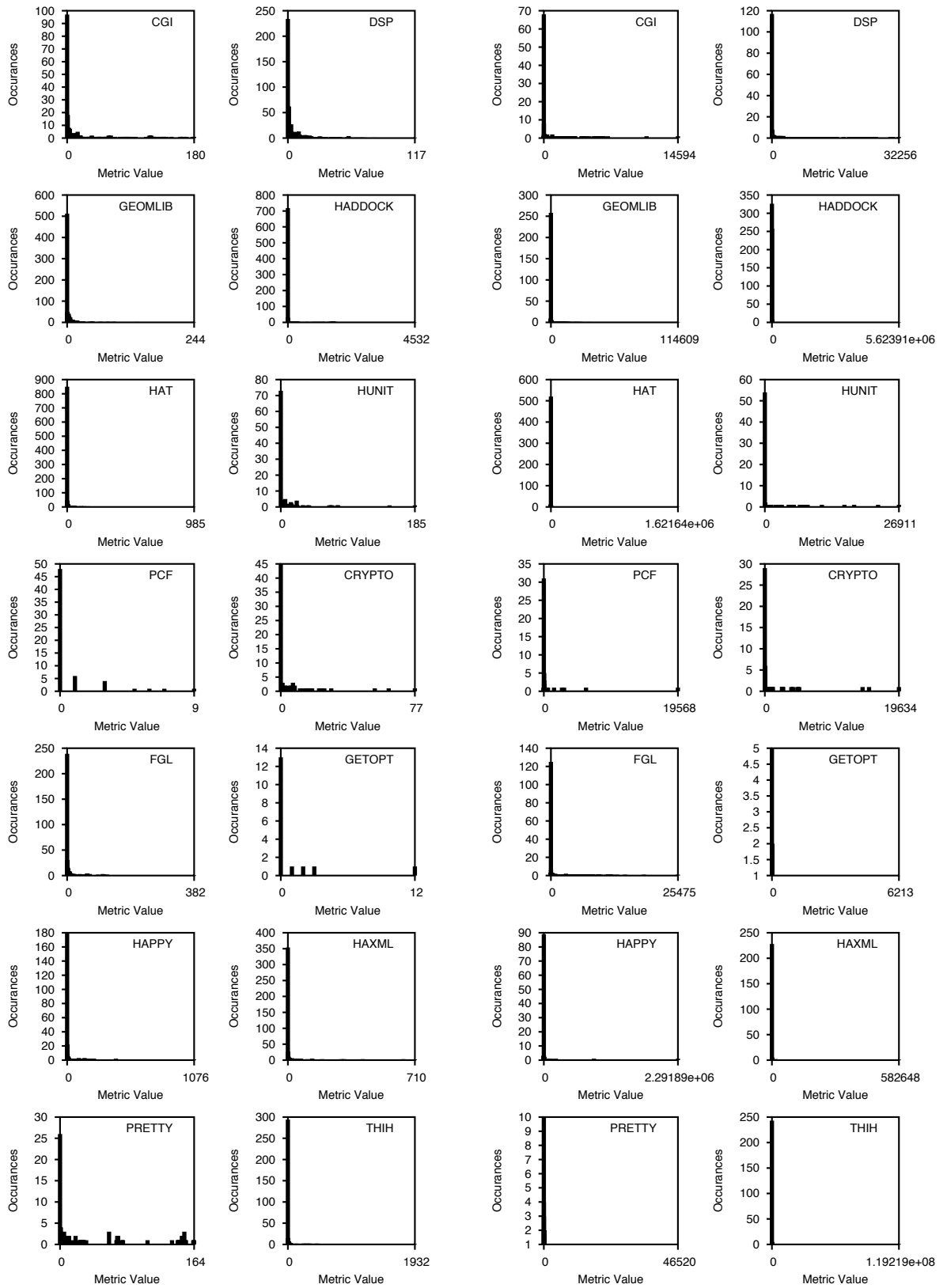


Figure 92: Histograms of “Distance by the average number of source lines” values.

Figure 93: Histograms of “Distance by the sum of the number of parse tree nodes” values.

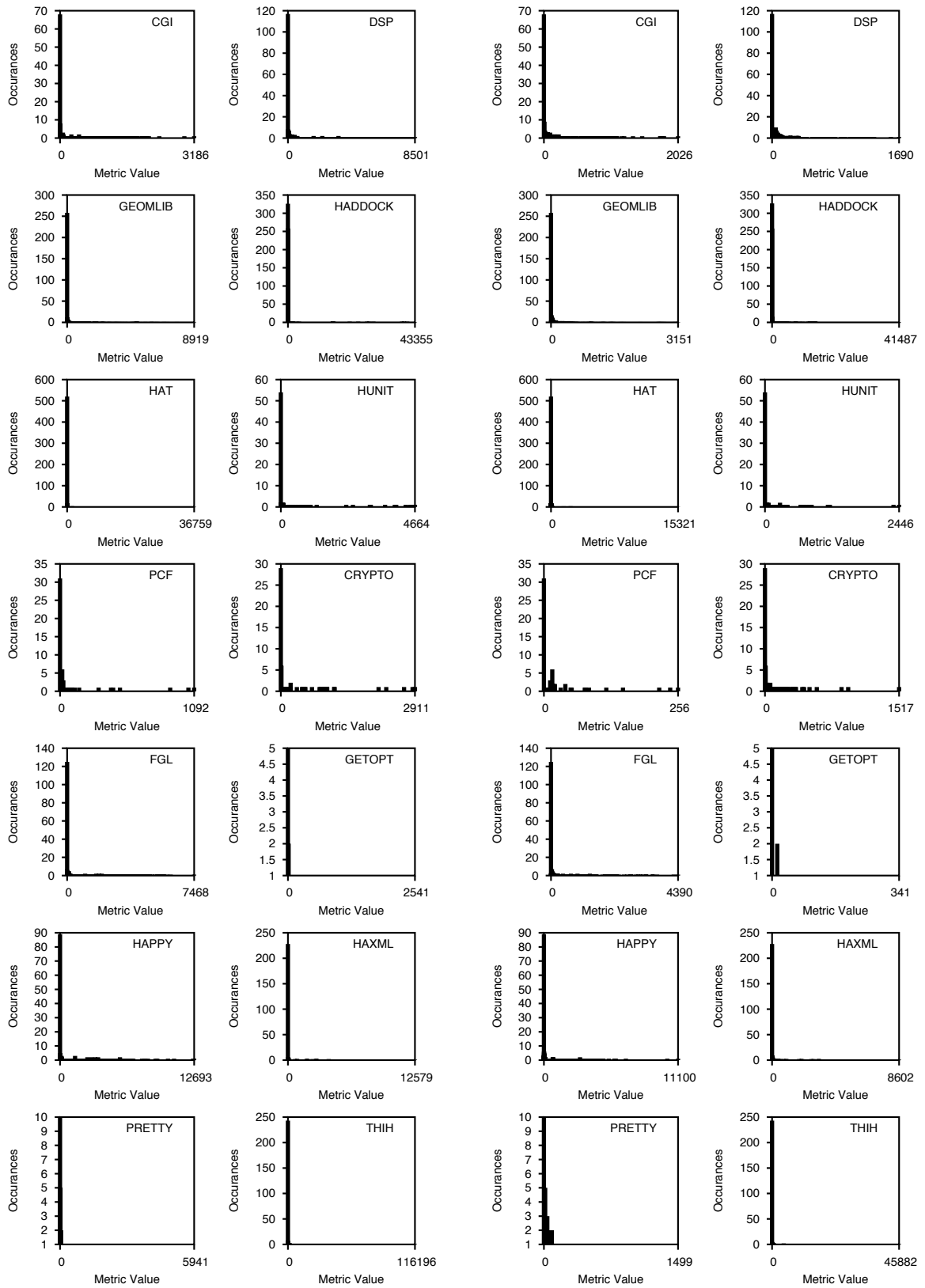


Figure 94: Histograms of “Distance by the maximum number of parse tree nodes” values.
 Figure 95: Histograms of “Distance by the average number of parse tree nodes” values.

Bibliography

- [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 Sunyé. 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.
- [25] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Françoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings of*

- WCRE'99 (6th Working Conference on Reverse Engineering)*, pages 175–187, Atlanta, Georgia, USA, October 1999. IEEE Computer Society Press.
- [26] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177, Minneapolis, Minnesota, USA, 2000. ACM Press.
- [27] A formal description of the DOT language. AT&T Labs Research, Florham Park, NJ, USA. <http://www.graphviz.org/cvs/doc/info/lang.html>, March 2003.
- [28] Micah Dubinko. *XForms Essentials*. O'Reilly & Associates, Inc., Cambridge, MA, USA, 2003.
- [29] James Eagan, Mary Jean Harrold, James A. Jones, and John Stasko. Visually encoding program test information to find faults in software. Technical Report GIT-GVU-01-09, College of Computing / GVU Center, Georgia Institute of Technology, 2001.
- [30] J. David Eisenberg. *SVG Essentials*. O'Reilly & Associates, Inc., Cambridge, MA, USA, 2002. <http://www.w3.org/TR/SVG/>.
- [31] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and Dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, Berlin, Germany, 2004.
- [32] Norman E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, London, UK, 1992.
- [33] Norman E. Fenton and R. W. Whitty. Axiomatic approach to software metrication through program decomposition. *The Computer Journal*, 29(4):330–339, 1986.

- [34] David Flanagan. *JavaScript (2nd ed.): The definitive guide*. O'Reilly & Associates, Inc., Cambridge, MA, USA, 1997.
- [35] Karl Franz Fogel and Moshe Bar. *Open Source Development with CVS*. Coriolis Group Books, Scottsdale, AZ, USA, 2001. <http://cvsbook.red-bean.com/>.
- [36] 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/>.
- [37] R.S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.
- [38] Leif Frenzel. Haskell support for the Eclipse IDE. <http://eclipsefp.sourceforge.net/>, April 2004.
- [39] Emden Gansner, Eleftherios Koutsofios, and Stephen North. User guide for the GraphViz dot tool. AT&T Labs Research, Florham Park, NJ, USA. <http://www.graphviz.org/Documentation/dotguide.pdf>, March 2003.
- [40] 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.
- [41] Thomas Hallgren. Haskell tools from the Programatica project. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 103–106, Uppsala, Sweden, 2003. ACM Press.
- [42] Maurice H. Halstead. *Elements of Software Science*. Elsevier, New York, USA, 1977.
- [43] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison-Wesley, Boston, MA, USA, 1996.

- [44] Les Hatton. *Safer C: Developing for High-Integrity and Safety-Critical Systems*. McGraw-Hill, New York, NY, USA, January 1995.
- [45] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.
- [46] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, Cambridge, UK, 2000.
- [47] C. D. Ince and S. Hekmatpour. An approach to automated software design based on product metrics. *Software Engineering Journal*, 3(2):53–56, 1988.
- [48] Dean Jerding and John Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):257–271, 1998.
- [49] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the IEEE Conference on Visualization*, pages 284–291, San Diego, CA, USA, October 1991. IEEE Computer Society Press.
- [50] Richard Jones and Tony Printezis. GCspy: An adaptable heap visualisation framework. Technical Report 5–02, University of Kent at Canterbury, March 2002.
- [51] David J. King and John Launchbury. Lazy depth-first search and linear graph algorithms in Haskell. In John T. O’Donnell and Kevin Hammond, editors, *Glasgow Workshop on Functional Programming*, pages 145–155, Ayr, Scotland, 1993. Springer-Verlag.
- [52] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

- [53] Claire Knight. System and software visualisation. In S. K. Chang, editor, *Handbook of software engineering and knowledge engineering*, volume 2, pages 82–92. World Scientific, Hackensack, NJ, USA, 2002.
- [54] Claire Knight and Malcolm Munro. Comprehension with[in] virtual environment visualisations. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 4–11, Pittsburgh, Pennsylvania, USA, 1999. IEEE Computer Society.
- [55] Claire Knight and Malcolm Munro. Visualising software - a key research area. Technical Report Computer Science Technical Report 8/99, University of Durham, Computer Science, University of Durham, UK, June 1999.
- [56] Claire Knight and Malcolm Munro. Virtual but visible software. In *Proceedings of the International Conference on Information Visualisation*, pages 198–205, London, UK, 2000. IEEE Computer Society.
- [57] Claire Knight and Malcolm Munro. Software visualisation conundrums. Technical Report Computer Science Technical Report 05/01, University of Durham, Computer Science, University of Durham, UK, July 2001.
- [58] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proceedings of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375, New Orleans, Louisiana, USA, January 2003. Springer-Verlag.
- [59] Daan Leijen. wxHaskell – A portable and concise GUI library for Haskell. In *Proceedings of ACM SIGPLAN Haskell Workshop*, Snowbird, Utah, September 2004. ACM Press.
- [60] 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.

- [61] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343. ACM Press, 1995.
- [62] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.
- [63] J. D. Mackinlay, G. G. Robertson, and S. K. Card. The Perspective Wall: Detail and context smoothly integrated. In *Proceedings of CHI'91, ACM Conference on Human Factors in Computing Systems*, pages 173–179, New Orleans, Louisiana, USA, April 1991. Addison-Wesley.
- [64] Simon Marlow. Haddock, a Haskell documentation tool. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 78–89, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.
- [65] Simon Marlow and Krasimir Angelov. Visual Studio / Haskell - Support for Haskell in Microsoft Visual Studio IDE. Microsoft Research, Cambridge, UK. <http://www.cs.kent.ac.uk/projects/refactor-fp/workshop/simonm.ppt>, April 2004.
- [66] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. Microsoft Research, Cambridge, UK. <http://www.haskell.org/ghc/>, December 2004.
- [67] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [68] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science*, 82(3):20–30, 2003.

- [69] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [70] Ivan Moore. Guru — A tool for automatic restructuring of self inheritance hierarchies. In R. Ege, editor, *Proceedings of TOOLS-USA '95, Santa Barbara, (CA), USA*, pages 267–275. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.
- [71] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the 11th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications*, pages 235–250, San Jose, California, USA, 1996. ACM Press.
- [72] Bryan Oakley and John Klassa. TkDiff: A visual comparison tool. <http://sourceforge.net/projects/tkdiff/>, July 2004.
- [73] Linda M. Ott. The early days of software metrics: Looking back after 20 years. In Austin Melton, editor, *Software Measurement*, chapter 2, pages 7–25. International Thompson Publishing, London, UK, 1996.
- [74] M. J. Oudshoorn, H. Widjaja, and S. K. Ellershaw. Aspects and taxonomy of program visualisation. In Peter D. Eades and Kang Zhang, editors, *Software Visualisation*, volume 7, pages 3–26. World Scientific, Hackensack, NJ, USA, 1996.
- [75] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford University, Stanford, CA, USA, November 1999.
- [76] Sven Panne. HOpenGL: An OpenGL/GLUT binding for Haskell. <http://www.haskell.org/HOpenGL/>, July 2004.
- [77] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability Maturity Model, Version 1.1. *IEEE Software*, 10(4):18–27, July 1993.

- [78] Robert W. Peach, editor. *The ISO 9000 Handbook*. McGraw-Hill, New York, NY, USA, 1996.
- [79] James F. Peters and Witold Pedrycz. *Software Engineering: An Engineering Approach*. John Wiley & Sons, New York, USA, 1999.
- [80] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. <http://www.haskell.org/definition/>.
- [81] Claus Reinke. GHood – graphical visualisation and animation of Haskell object observations. In Ralf Hinze, editor, *ACM SIGPLAN Haskell Workshop, Firenze, Italy*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 121–155. Elsevier Science, September 2001.
- [82] Jonathan C. Roberts. Exploratory visualization using bracketing. In *AVI '04: Proceedings of the working conference on advanced visual interfaces*, pages 188–192, Gallipoli, Italy, 2004. ACM Press.
- [83] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194, New Orleans, Louisiana, USA, 1991. ACM Press.
- [84] Dan Russell. *FAD: A Functional Analysis and Design Methodology*. PhD thesis, Computing Laboratory, University of Kent at Canterbury, Computing Laboratory, University of Kent at Canterbury, Canterbury, CT2 7NF. UK, January 2001.
- [85] Dave Schreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley, Boston, MA, USA, 1999.
- [86] Martin Shepperd. Design metrics: An empirical analysis. *Software Engineering Journal*, 5(1):3–10, 1990.

- [87] Forrest Shull, Vic Basili, Barry Boehm, A. Windsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Eighth IEEE Symposium on Software Metrics*, pages 249–258, Ottawa, Canada, June 2002. IEEE Computer Society Press.
- [88] Diana Sidarkeviciute. Program analysis and visualisation: Towards a declarative approach. *Informatica*, 8(1):153–175, 1997.
- [89] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In Pedro Sousa, editor, *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, Lisbon, Portugal, 2001. IEEE Computer Society.
- [90] Kevin W. Smith. Using metrics to reduce software defects. In *Eighth IEEE Symposium on Software Metrics, Industrial Practices Proceedings*, pages 145–155, Ottawa, Canada, June 2002. IEEE Computer Society Press.
- [91] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 17 – 28, Dearborn, Michigan, USA, 1997. IEEE Computer Society.
- [92] S.D. Swierstra, M. Schrage, and J.T. Jeuring. Combinators for layered software architectures. Technical Report UU-CS 2002-30, Department of Computer Science, Utrecht University, Department of Computer Science, Utrecht University, 2002.
- [93] Tarja Systä, Ping Yu, and Hausi Müller. Analyzing Java software by combining metrics and program visualization. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pages 199–208, Zurich, Switzerland, 2000. IEEE Computer Society.

- [94] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Boston, MA, USA, July 1996.
- [95] TkCVS, a graphical interface to CVS. <http://www.twobarleycorns.net/tkcv.html>, May 2004.
- [96] 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.
- [97] The World Wide Web Consortium (W3C): Leading the web to its full potential... <http://www.w3.org/>, January 2004.
- [98] Philip Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.
- [99] D. Wakeling. A design methodology for functional programs. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings*, volume 2196 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2001.
- [100] Elaine J. Weyuker and Alberto Avritzer. A metric to predict software scalability. In *Eighth IEEE Symposium on Software Metrics*, pages 152–158, Ottawa, Canada, June 2002. IEEE Computer Society Press.
- [101] M. C. Wheadon. VDiff: A tool for visually comparing differences between files. University of Kent, Canterbury, UK. <http://www.cs.kent.ac.uk/development/kst/descriptions/vdiff.html>, July 2004.
- [102] Christopher Williamson and Ben Shneiderman. The dynamic HomeFinder: evaluating dynamic queries in a real-estate information exploration system. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 338–346, Copenhagen, Denmark, 1992. ACM Press.

- [103] Martin R. Woodward and Zuhoor A. Al-Khanjari. Testability, fault size and the domain-to-range ratio: An eternal triangle. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 168–172, Portland, Oregon, USA, 2000. ACM Press.
- [104] P. Young. *Visualising Software in Cyberspace*. PhD thesis, University of Durham, South Road, Durham, England, October 1999.
- [105] P. Young and Malcolm Munro. Visualising software in virtual reality. In *Proceedings of the IEEE 6th International Workshop on Program Comprehension*, pages 19–26, Ischia, Italy, 1998. IEEE Computer Society.
- [106] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1979.
- [107] Horst Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter & Co., Berlin, Germany, 1991.