Evaluating Haskell Metrics

Looking for correlations between bug occurrences and code metrics

Nikola Dzhunov[1]
Supervisor(s): Jesper Cockx[1]**, Leonhard Applis**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Nikola Dzhunov
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Leonhard Applis, Koen Langendoen

Abstract

This study explores the use of Code Churn and Pattern Size (PSIZ) metrics to identify bug-prone areas in Haskell codebases. The primary research questions addressed are whether these metrics can effectively predict areas of code instability and potential bugs. Our contributions include a comprehensive analysis of these metrics across three large Haskell projects, examining the correlation between high metric values and documented bugs.

Our findings reveal that Code Churn is a significant indicator of potential bugs, with 'buggy' files showing markedly higher mean code churn values. However, the PSIZ metric proved ineffective in predicting bug-prone areas, as the mean and median values for both projects and 'buggy' files were similar and low. These results suggest that while Code Churn is a useful metric for identifying unstable code areas, PSIZ does not offer the same predictive value. Future research should expand the dataset and consider additional metrics to enhance the reliability of these findings.

1 Introduction

In exploring the life cycle of software artifacts, particularly within the realm of Haskell, we encounter a significant gap in understanding compared to more mainstream languages like Java. While extensive studies exist for languages such as Java [2][3][5], encompassing various aspects of the software life cycle, including release life cycles, maintenance activities, and risk management, the landscape for Haskell remains notably under-reported.

Haskell's distinctive characteristics, such as its robust compiler, have fostered a perception within the community of a fundamentally different programming experience, one less prone to bugs. However, emerging research suggests that despite these inherent strengths, developers often exhibit familiar behaviors, such as resorting to print statements [1], and may even knowingly execute code they anticipate will fail compilation.

By analyzing various Haskell metrics, we aim to shed light on several key questions:

1. Average Metric Values: What are the typical changes in the mean values of different metrics across the life cycle of a project? Understanding these changes and mean values can provide a baseline for assessing the quality and complexity of Haskell projects and functions.

2. Correlation with Bug Occurrence: Do certain Haskell metrics correlate with the occurrence of bugs or defects in software? By establishing correlations, we can potentially identify metrics that serve as early indicators of code quality issues or areas prone to bugs, thus dedicating more resources and testing to these specific areas.

Addressing these questions not only fills a crucial void in our understanding of Haskell's software life cycle but also offers valuable insights for developers, project managers, and researchers alike.

There has been a paper delving into different metrics to measure in Haskell from Chris Ryder and Simon Thompson [6]. They outline various software metrics applicable to Haskell programs. Through the examination of two case study programs, they demonstrate the potential utility of certain metrics in identifying functions with a heightened likelihood of containing errors. Consequently, these findings suggest that such functions could benefit from better testing procedures. However, they assess the metrics only on two case studies so their results cannot be generalized for all Haskell projects. Additionally, they haven't been able to look into type-based metrics.

The main research question we will be answering is "What are the mean values of Haskell Metrics in various projects and how do they correlate with bug occurrence rates?". The goal of the project is to look into these metrics for large and used projects to see if there is some correlation between bug occurrences and the metrics.

The questions that will be answered are:

- What are metrics that can be used to evaluate Haskell code?
- What are the mean and median values for these metrics for various Haskell projects?
- Is there a correlation between the values of the metrics and the chance of bugs appearing?

In section 2, we'll delve into what other researchers have discovered about the topic in Related Work, examining their findings and pinpointing areas where questions remain. Moving on to section 3, Methodology, we'll outline the steps we took to conduct our research, explaining why we chose these particular methods. Section 4, Results, will present the outcomes of the research in a straightforward manner, using figures and tables to illustrate our findings in line with the research questions. In section 5, Discussion, we'll interpret these results and reflect on the methodology used, exploring the implications of our findings. Section 6, Responsible Research, will scrutinize the ethical considerations of the study, including how we ensured the reproducibility of the methods and maintained fairness in the data practices. In section 7, Conclusions and Future Work, we'll summarize our main findings, draw conclusions, and suggest avenues for further research, highlighting key insights and areas for improvement. Finally, in section 8, Acknowledgements, we'll express gratitude to those who contributed to our research journey.

2 Related Work

There is some previous work that delves into different code metrics. Code churn, often referred to as code volatility or code instability, is a significant metric in software development that measures the frequency of code changes within a repository over time[4]. Numerous studies in the realm of software engineering have explored the implications of code churn in various programming languages and software projects. However, when it comes to Haskell-specific research, the literature on code churn is relatively sparse. While there exists a wealth of research on code churn in mainstream languages such as Java, C++, and Python[7], the same cannot be said for Haskell. Haskell's unique features, such as its strong static typing and functional programming paradigm, may lead to different patterns of code churn compared to imperative or object-oriented languages. Thus, there is a gap in understanding how code churn manifests in Haskell projects and its implications for software development practices. Moreover, this research aims to investigate not only how code churn manifests in Haskell projects but also to check if there is any correlation with occurrences of bugs, providing deeper insights into software quality and reliability.

Additionally, in their paper "Software Metrics: Measuring Haskell"[6] Chris Ryder and Simon Thompson introduced metrics for analyzing Haskell code, including a large variety of metrics - pattern metrics, distance metrics, callgraph metrics and metrics for function attributes. While these metrics offer promising avenues for understanding Haskell code quality and design, it's essential to note that their evaluations have been limited to a small number of case studies. Ryder and Thompson's work primarily examined these metrics in

the context of two relatively small Haskell programs because they needed to evaluate the metrics manually, which may not fully represent the diversity and complexity of real-world Haskell projects. As a result, their findings cannot be generalized for all Haskell projects. Additionally, this research seeks to extend their work by exploring potential correlations between some of these metrics and bug occurrences in Haskell projects, providing a more comprehensive understanding of software quality factors in functional programming contexts.

Despite the advancements in understanding code churn and the introduction of various Haskell metrics, several questions remain unanswered. Specifically, there is a need to explore code churn and these metrics in a bigger set of Haskell Projects and see if there are any repeating patterns or correlations with bugs. This research will conduct a comprehensive empirical study of these metrics in a bigger and more diverse set of Haskell projects. By analyzing a more representative sample of Haskell repositories, this study aims to provide more robust insights into the effectiveness of various metrics in assessing Haskell code quality. Furthermore, by examining the correlation between these metrics and bug occurrences, this research aims to contribute to the understanding of software quality and reliability in Haskell development.

3 Methodology

3.1 The metrics

In this section, we describe the metrics utilized in the study to analyze Haskell programs. Due to limitations in time and the available processing power, we looked only into two metrics.

3.1.1 Code Churn

The first metric that we look into is code churn. Code churn measures the frequency and extent of changes made to the code over time. It provides insights into the volatility and stability of the software system. Higher code churn may indicate areas of frequent development activity or potential code quality issues that require attention.

For evaluating the code churn on the repositories, we decided to use a Python script from Francis Lacle.

3.1.2 Pattern Size (PSIZ)

The second metric we evaluated was inspired by Chris Ryder and Simon Thompson's Pattern Size (PSIZ) metric [6]. Their metric quantifies the complexity of patterns used in pattern matching within Haskell code. It measures the size of a pattern by counting the number of components in the abstract syntax tree (AST) of the pattern. The assumption is that as pattern sizes increase, they become more complex, potentially leading to increased cognitive load for developers and impacting code readability and maintainability. Thus, the more complex the function is, the more likely it is to contain a bug.

However, counting the number of components in the AST is a little bit ambiguous as there are different definitions of how to count them, and also different libraries may build the AST from the code in a different way. Thus we settled for our version of the PSIZ metric where instead of counting the number of components of the AST for each pattern, we count

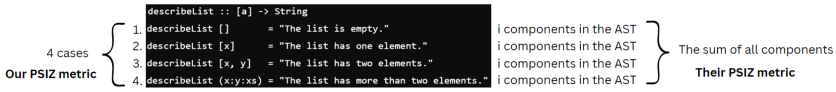

Figure 1: Example of the difference between the PSIZ metrics

the cases in the pattern matching for each function. In Figure 1 we show the differences between the two PSIZ metrics.

3.2 The projects

When picking the repositories to evaluate our metrics on, it is important that the projects have a well-documented commit history and also reporting for bugs. As we want to look at big and used projects across their whole life cycle to find some patterns, the project's commit history needs to have good documentation. Also, in order to check for correlations with bug occurrences, it is important to know at what point of the project's life cycle were the bugs introduced and fixed. Taking these requirements into account we picked three large projects on which to evaluate our metrics.

The three initial repositories are showed in Table 1:

Repository Name	**Number of Commits**	**Github Stars**
quickcheck	1174	700
hackage-server	2350	410
lens	4332	2002

Table 1: Used repositories

3.3 The plan

The steps that we plan to take to achieve the aim of our research and answer the questions we have set at the beginning are as follows:

1. Pick the metrics to evaluate on Haskell projects
2. Pick the initial projects on which to evaluate the metrics
3. Evaluate the metrics on the initial projects
4. Analyze the data and check if there can be found some correlation between bug occurrences and the metrics' values

This can also be seen in Figure 2 with connections between the different tasks.

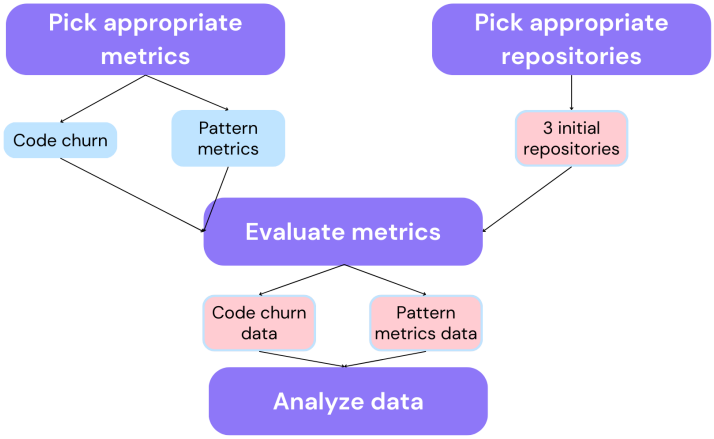

Figure 2: Diagram of the steps taken for the experiment

4 Results

In this section, we will look at the results that were achieved while evaluating the different metrics on the repositories. It is divided in two subsections - 4.1 where we look into the results from the metric Code Churn and 4.2 where we look into the results from the PSIZ metric.

4.1 Code Churn

First, we evaluate the code churn metric on the whole repositories across the entire commit history of the projects.

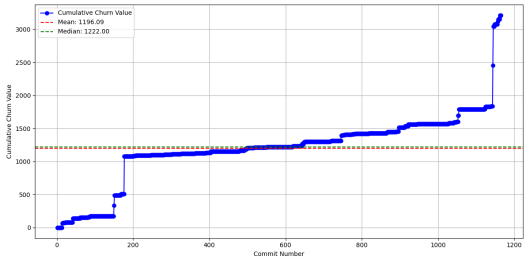

Figure 3: quickcheck cumulative

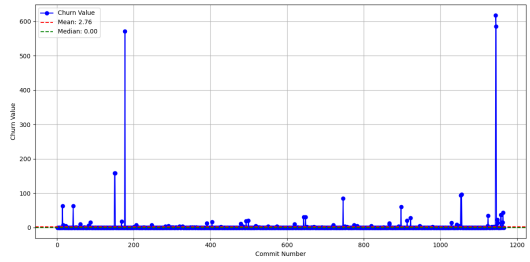

Figure 4: quickcheck non-cumulative

We display the data we achieved in two ways - cumulative and non-cumulative. In Figures 3, 5 and 7 we show the cumulative code churn values. For each commit, we display the code churn that has been added from the start of the project until now. In this way, we can see how the code churn grows over time and also it is helpful to visualize how the code churn has been changing across the project's life cycle.

We also show the non-cumulative code churn values across the whole commit history in Figures 4, 6, and 8. There we only show for each commit only the code churn value that

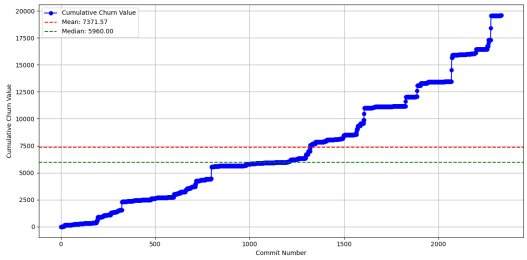

Figure 5: hackage-server cumulative

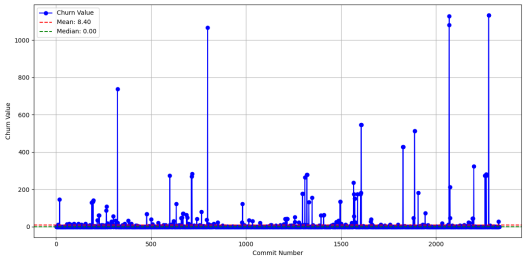

Figure 6: hackage-server non-cumulative

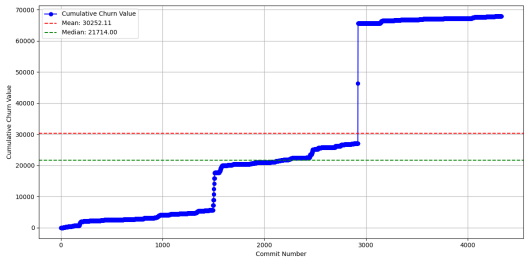

Figure 7: lens cumulative

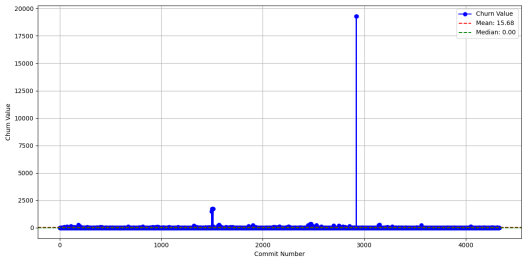

Figure 8: lens non-cumulative

this commit has added. So in the perfect scenario where we don't want any code churn, all the commits in these Figures will have 0 as values. However, that is not the case and we can see that each project has some code churn - some across the whole project's commit history, and some have big spikes where probably big changes were made.

However, this is not enough to argue about the occurrences of bugs and how code churn correlates with that. So we also searched in these repositories for 'buggy' files. By 'buggy' file we mean files in the project that have had a documented existing bug and this bug has been fixed. We did that by looking into issues that have been closed and have the bug tag, and then seeing what files were changed to fix the bug/issue.

In Table 2 we can see the mean and median values of the whole repositories and also of the 'buggy' files.

Project	**Mean**	**Median**	**File**	**File's Mean**	**File's Median**
quickcheck	2.76	0	Property	19.38	1
			Gen	16.50	0
			Arbitrary	6.62	0
hackage-server	8.40	0	Html	18.18	1
			UserDetails	101.33	61
lens	15.68	0	FieldTH	60.29	4.50
			At	96.06	6.50
			TH	75.10	0
			Cons	175.83	28.50

Table 2: Mean and Median Code Churn Values for Projects and Buggy Files

An interesting result here that needs to be taken into account is that all the mean values of the 'buggy' files are larger than the mean values of the projects there are in. Additionally, the median of all three repositories is 0, which means that most commits in the projects don't add code churn. However, 6/9 of the 'buggy' files have a median that is larger than 0, which means that most of the commits in these files add code churn to the project, so have been changing frequently in a short time.

4.2 PSIZ

First, we evaluate the PSIZ metric on the whole repositories across the entire commit history of the projects. However, unlike with code churn where we can get one value for the entire project per commit, the PSIZ metric is evaluated on single functions. So for each commit, we show the mean value of the PSIZ metric in the project (the sum of the PSIZ metric for each function that uses pattern matching divided by the total number of functions that use pattern matching) and also the median value for the PSIZ metric (the median of the PSIZ metric from all functions that use pattern matching).

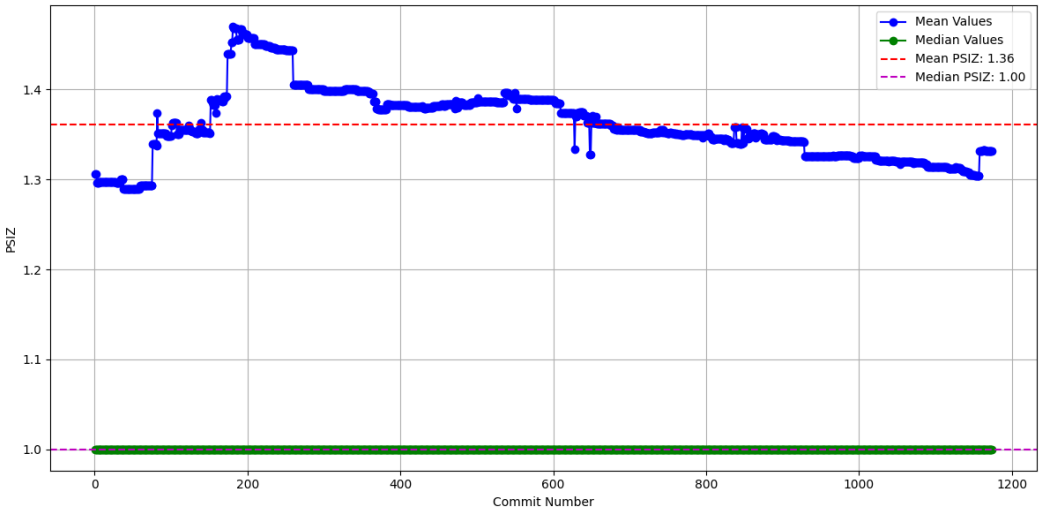

Figure 9: quickcheck PSIZ values

In Figures 9, 10 and 11 we can see how the mean and median values of the PSIZ metric change across the whole commit history of the project. Additionally, the mean of the means and the median of the medians are also shown.

However, this again is a good visualization of the metric but it is not enough to argue about the occurrences of bugs and how that correlates with the values of the metric. So again we evaluated the metric on the 'buggy' files that were described in Section 4.1 and displayed the data in Table 3.

Unlike with the code churn metric, here the results are not that drastically different. We can see that the median in all the projects is 1 which means that most of the functions that use pattern sizes have only one pattern case. From the median of the 'buggy' files, we can see that this is also the case for them and there cannot be made some distinction.

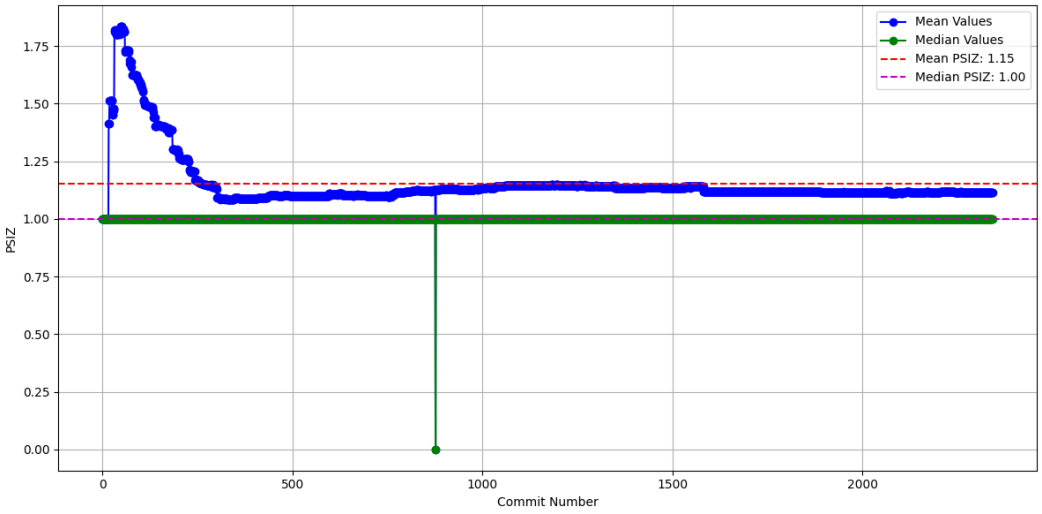

Figure 10: hackage-server PSIZ values

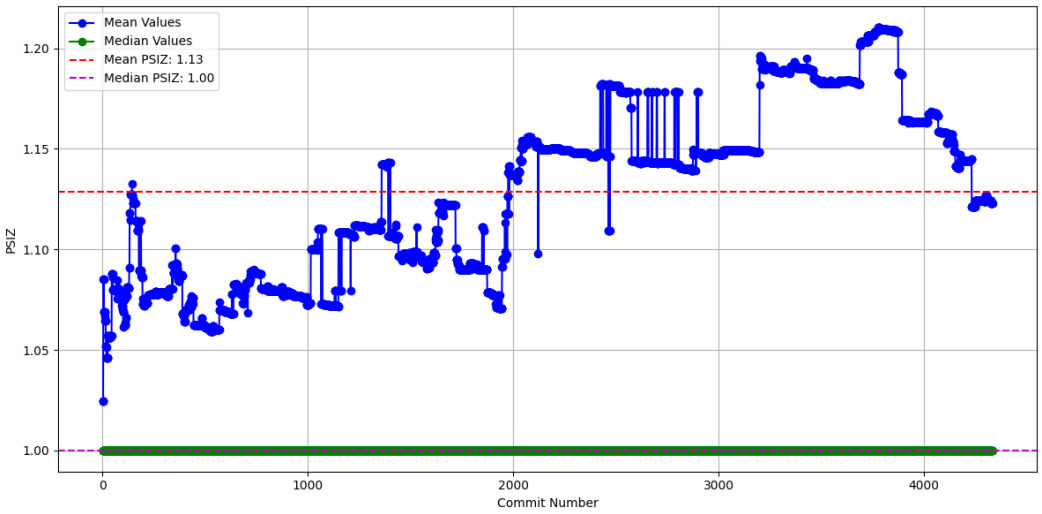

Figure 11: lens PSIZ values

Additionally, two of the files have a median of 0 which means that for most of their commits, these files don't have any functions that use pattern matching.

Additionally, we can see that the means of the 'buggy' files are also less than the mean of the repositories and overall the mean of the repositories is around 1 with a maximum of around 1.80 (can be seen from the figures).

Project	Mean	Median	File	File's Mean	File's Median
quickcheck	1.36	1	Property	0.64	1
			Gen	1.23	1
			Arbitrary	1.00	1
hackage-server	1.15	1	Html	0	0
			UserDetails	1	1
lens	1.13	1	FieldTH	1.10	1
			At	0.69	1
			TH	0.44	0
			Cons	0.89	1

Table 3: Mean and Median PSIZ Values for Projects and Buggy Files

5 Discussion

5.1 Explanation of Findings

Our study aimed to analyze the occurrences of bugs using two primary metrics: Code Churn and Pattern Size (PSIZ). The results obtained from these metrics provided insightful information about potential areas prone to bugs.

Code Churn:

The results we got when evaluating the code churn metric revealed significant insights into the occurrences of bugs in the given repositories. Only by looking at the means we can see a significant difference between the 'buggy' files' means and the projects' means. All the 'buggy' files have a higher mean (some even 10x more) which means that the files that have had documented and fixed bugs, have been changed a lot more frequently, thus having a larger code churn value. So from these results on these large and used projects, it can be seen that files that have had bugs have a larger code churn mean value than the mean of the entire project.

Another interesting result to look into is also the median. All the repositories have a code churn median of 0 which means that most commits don't bring any new code churn into the project which is good. However, if we compare that to the 'buggy' files, we see that almost all of them have a code churn median that is larger than 0, so most of their commits have introduced code churn and thus have been changed a lot more and frequently.

Lastly, we can see that the mean of the projects grows with the commit size, so the bigger the commit history, the bigger the code churn. Of course that is not always true as some projects may have commits that introduce less code churn than others, but still from looking into these famous and large projects, we can see that even projects that are used often in practice and have a lot of contributors, still have code churn and the more commits they have, the bigger the chance is that they average code churn is also bigger.

Pattern Size (PSIZ):

Unfortunately, the results here were not positive as with code churn. Looking at the median we can see that here the values for the projects are all pretty similar and close to 1. The median PSIZ values for the 'buggy' files are also close to 1. Even though all of them are smaller than the mean of the repositories, the differences aren't big enough to draw any conclusions. And after further evaluations, some of the values are so low because in some commits the files don't have any functions that use pattern matching. Thus the PSIZ value is probably not a good metric to check for occurrences of bugs as in the first place these files

and repositories don't use functions with large pattern sizes.

This is additionally proven by looking into the medians of the repositories and 'buggy' files. The medians of the repositories are all 1, so that means that for all the functions that use patter matching in the projects, they have only one case in most cases. And we can also see from the medians for the 'buggy' files that their PSIZ values are also 1 in almost all cases. So, the 'buggy' files don't really have complex pattern-matching functions and are pretty similar to the average PSIZ values of their repositories.

However, this is still a helpful result as it show that for larger projects and more used projects, there is no need to look into the pattern sizes of functions as they are really close to 1. An interesting observation is also that here we can see that for all there projects the mean PSIZ value is around 1, despite them being quite different in size and number of commits. So there may be some mean value around one that most big Haskell projects will have but further evaluation will be needed to draw such a conclusion.

5.2 Reflection on Methodology

Our methodology provided a robust framework for evaluating the selected metrics, but it also had several limitations that need to be acknowledged.

Project Selection:

We selected projects with well-documented commit histories and bug reports, which is crucial for correlating metric values with bug occurrences. However, the limited number of projects (three in total), despite being large and diverse, may not provide a comprehensive view of the broader Haskell ecosystem. Expanding the analysis to include more projects with various characteristics could help generalize the findings.

Data Analysis:

Our approach to analyzing cumulative and non-cumulative code churn provided a detailed view of code stability over time. However, the evaluation of PSIZ values at the file level, while informative, may have obscured broader trends. Trying to evaluate the PSIZ on 'buggy' functions instead of files may yield more specific results to further solidify our findings or to propose counter-examples to our results. Additionally, incorporating statistical techniques to test the significance of correlations between metrics and bug occurrences could strengthen the validity of the findings.

5.3 What does that mean

For Haskell Developers:

Our results and findings for Haskell developers mean that the code churn metric can be a really good indicator for the occurrences of bugs. So if they have a big project but not enough resources to thoroughly test it, it may be a good idea to run a code churn script or software for the whole repository and then for all the files to see the mean and median values. And then only check the files that have a really large mean and median value compared to the whole repository. Of course, this is not a foolproof method and is not guaranteed to catch all bugs but is a good and cheap way to focus your attention and resources on files that have a bigger chance to have bugs.

Similarly, it may not be useful to spend resources trying to find files with bugs using PSIZ metrics as they may be not that effective and not even have a correlation with bug occurrence rates.

For fellow Researchers:

For fellow researchers, our findings and implemented methods can be a good starting point to further evaluate these metrics or use the results to compare to other evaluations. For both the code churn and PSIZ metric, evaluation on a bigger set of projects will contribute greatly to solidifying out results and conclusions in this research.

Additionally, researchers who are also looking into evaluating metrics for Haskell code can already see that there has been research on code churn and PSIZ and focus their attention on other metrics so that the field of Haskell metrics can be reported more extensively.

6 Responsible Research

In conducting this project, which involves evaluating different coding metrics across various repositories to assess potential correlations with bug occurrence, we have prioritized responsible research practices to ensure the integrity and reliability of our findings. We have meticulously documented the methodologies used, ensuring transparency and reproducibility in our approach. Every data point was treated fairly throughout the evaluation process, and no information was omitted. The commitment to responsible data handling extends to sharing our findings, enabling others to replicate our analysis and validate conclusions. By upholding these principles, we aim to contribute robust and trustworthy insights to the field while fostering a culture of ethical research conduct.

7 Conclusions and Future Work

This research aimed to investigate the effectiveness of two specific metrics, Code Churn and Pattern Size (PSIZ), in predicting bug-prone areas within Haskell projects. The primary research questions we sought to answer were:

- What are the mean values of Haskell metrics across various projects, and how do they change over the project lifecycle?
- Do certain Haskell metrics correlate with the occurrence of bugs or defects in the software?

Our goal was to understand if these metrics could provide early indicators of code quality issues, helping developers and project managers identify areas that may require additional resources or testing.

The Code Churn metric, which measures the frequency and extent of changes made to code, proved to be a significant indicator of potential bugs. Our analysis across three large Haskell projects showed that files labeled as 'buggy' consistently exhibited higher mean code churn values compared to the overall project means. This suggests that files undergoing frequent changes are more likely to contain bugs. The median values also supported this conclusion, as the 'buggy' files had medians greater than zero, indicating frequent changes, while the project-wide medians were zero.

The PSIZ metric, inspired by the work of Chris Ryder and Simon Thompson, was less effective in predicting bug-prone areas. The PSIZ metric measures the complexity of patterns used in pattern matching within the code. Our analysis showed that the median PSIZ values for both the entire projects and the 'buggy' files were similar and close to 1, indicating no significant correlation between pattern size complexity and bug occurrences.

While Code Churn is a valuable metric for identifying unstable and bug-prone areas in Haskell code, PSIZ does not offer the same predictive value. Expanding the analysis to include more diverse projects and exploring additional metrics related to code quality could provide deeper insights into the dynamics of Haskell projects. Additionally, further research could investigate the potential combination of multiple metrics to improve predictive accuracy, ultimately contributing to the development of more reliable and maintainable software systems.

8 Acknowledgements

I would like to extend my deepest gratitude to everyone who contributed to the completion of this research.

First and foremost, I am immensely grateful to Professor Jesper Cockx for the invaluable feedback provided during the midterm poster session and on the paper draft. Their insights and guidance were instrumental in shaping the direction and quality of this research.

I would also like to thank my supervisor, Leonhard Applis, for their continuous support and guidance throughout this project. Their help during our weekly meetings and the constructive feedback provided were crucial in overcoming the various challenges encountered during this research.

Finally, I want to express my appreciation to my teammates. Their assistance, insights, and collaboration significantly contributed to the success of this project. Working with them has been a rewarding experience, and I am grateful for their support.

References

[1] Ruanqianqian (Lisa) Huang et al. "How do Haskell programmers debug?" In: *Plateau Workshop* (2023).

[2] Trupti S. Indi, Pratibha S. Yalagi, and Manisha A. Nirgude. "Use of Java Exception Stack Trace to Improve Bug Fixing Skills of Intermediate Java Learners". In: *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 2016, pp. 194–198. DOI: 10.1109/LaTiCE.2016.9.

[3] Scott Malabarba et al. "Runtime Support for Type-Safe Dynamic Java Classes". In: June 2000, pp. 337–361. ISBN: 978-3-540-67660-7. DOI: 10.1007/3-540-45102-1_17.

[4] J.C. Munson and S.G. Elbaum. "Code churn: a measure for estimating the impact of code change". In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, pp. 24–31. DOI: 10.1109/ICSM.1998.738486.

[5] Roman Roelofsen and Arne Koschel. "Evaluation of life cycle functionality of Java platform". In: (Jan. 2010).

[6] Chris Ryder and Simon Thompson. "Software Metrics: Measuring Haskell". In: *Intellect Books* (2005).

[7] Yonghee Shin et al. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". In: *IEEE Transactions on Software Engineering* 37.6 (2011), pp. 772–787. DOI: 10.1109/TSE.2010.81.