What about Haskell bugs?

Adapting existing bug taxonomies to Haskell's features and community

Razvan Nistor

Supervisors: Jesper Cockx, Leonhard Applis

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: Razvan Nistor
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx Leonhard Applis , Koen Langendoen

Abstract

The classification of bugs in functional languages is an understudied area, as opposed to imperative counterparts, such as Java. This paper acts as an initial step to cover this gap into two complementary directions. First, a dataset of 142 bugs from 10 Haskell FOSS repositories have been classified according to two taxonomies from literature in order to assess how well they handle the differences in programming paradigms. Based on our bug classifications, the first taxonomy has very little variance in types of bugs, with 86% of bugs being classified in the same category. On the other hand, the second taxonomy showed more potential as the bugs were more balanced between categories, but with occasional difficulties in classification, as some bugs fitted more than one type. Second, we performed interviews with four Haskell developers about their experience with bugs and the usefulness of these taxonomies in practice. They argued that while such taxonomies can prove useful, the context in which it is being used is more important. Thus, circumstances such as the size of the project and team, and stages of development need to be taken in consideration before trying to apply any taxonomy.

1 Introduction

Bugs are an inevitable part of software development. As code becomes larger and more complex, more bugs are introduced [2]. In order for developers to fix these bugs, they first need to understand them. To aid them, different types of defect classification schemes have been proposed, each offering a unique perspective on how bugs should be dealt with.

Bug classification of software bugs is an explored field. One of the earliest bug classifications, developed by IBM in 1992, is the *Orthogonal Defect Classification* (ODC) [3]. It was developed by the quality group from IBM as a more practical tool, to bridge the increasing gap that was forming between developers and the research into software quality. By classifying defects based on their causes, split into 8 types, it tried to provide fast and effective feedback to developers. ODC represented a starting point for many researchers to develop their own defect taxonomies, such as the defect categorization proposed by Seaman et al. [6]. Seaman's taxonomy builds on the purpose of the ODC, as it was created to unify existing categorization schemes used in multiple NASA centers and aid in future development projects. Another example, given by Catolino et al., classifies bugs based on types rather than causes [2], and is meant to assist in bug understanding and triaging, but also used by the research community to further investigate bug characteristics.

Numerous studies have been done on bugs for imperative languages such as Java. These include bug datasets such as ManySStuBs4J [5], which offers around 150,000 single statement bug fixes, as well as the aforementioned taxonomy proposed by Catolino et al. which was built on bug reports from ecosystems like Mozilla, Apache, and Eclipse [2], a large part of which contain projects written in Java.

Even though a lot of effort has been made to study bugs, there seems to be a gap in the transition from imperative languages to functional ones, where little to no effort has been done in trying to understand differences between bugs in the two paradigms, or classify functional bugs. This gap is even more visible in the struggles encountered by Haskell developers when debugging. Preliminary results from Huang et al. show that developers in functional programming use similar strategies to imperative counterparts, but that some of the functional features, such as laziness, declarative syntax and abstractions, often impede them [4]. Therefore, it is important to help the Haskell community by understanding bugs in the context of functional programming. This can be done by exploring artifacts in order to find characteristics and potential classification schemes for bugs, as well as better understanding how bugs are identified, reproduced, and fixed in practice.

The goal of this paper is to identify the extent to which existing bug taxonomies can classify bugs in Haskell, and how they could be adapted to better accommodate the unique features of Haskell. The specific research questions that will be studied are:

RQ$_1$ *What are the most common types of Haskell bugs?*

RQ$_2$ *What are the limitations of existing bug taxonomies in capturing the unique features of Haskell bugs?*

RQ$_3$ *How do Haskell developers classify bugs and how is it different from the previously discussed taxonomies?*

This paper aims to help the Haskell community by making use of the existing studies about bug classifications. This is achieved by:

- The collection of a bug dataset of 142 bugs from 10 Haskell FOSS (free open-source software) projects;
- Analysing the usefulness of existing bug taxonomies in Haskell;
- Collecting qualitative data from 4 Haskell developers about their perspectives on bugs.

In this study, it was found that a large majority (86%) of the encountered bugs were classified as Program Anomaly, according to the taxonomy proposed by Catolino et al. However, other categories, such as Configuration and Performance, were found to be much less common. Because of the high number of bugs classified into a single type of bugs, this taxonomy might be considered less applicable for offering the information required to properly triage bugs. On the other hand, the taxonomy proposed by Seaman et al. provided a more balanced distribution, with the largest categories being Algorithm / Method and Logic bugs. But, this taxonomy led to difficulties in the classification of a relatively small sample of bugs, named complex bugs, which allowed for classification of the same bug into multiple types. However, the interviewees argued that for taxonomies to be useful, their purpose needs to be clear. Otherwise, the large number of possibilities in bug categorization can change the focus from offering information to limiting the time available for the bugs.

The rest of the paper is structured as follows. Section 2 describes the methodology used in the study. Section 3 presents the results, and proposes discussion points related to the findings, limitations, and improvements for the methodology. Section 4 explores ethical concerns of the study. Finally, conclusions and recommendations for future works are discussed in Section 5.

2 Methodology

Figure 1 presents the main steps of the process followed in this study. The first stage consisted of collecting bugs from Haskell repositories. Then, they were classified based on two distinct taxonomies. In the second stage, Haskell maintainers were interviewed about their perspective on bugs in practice and the usefulness of the proposed taxonomies. Finally, all the results gathered through this process are presented in Section 3.

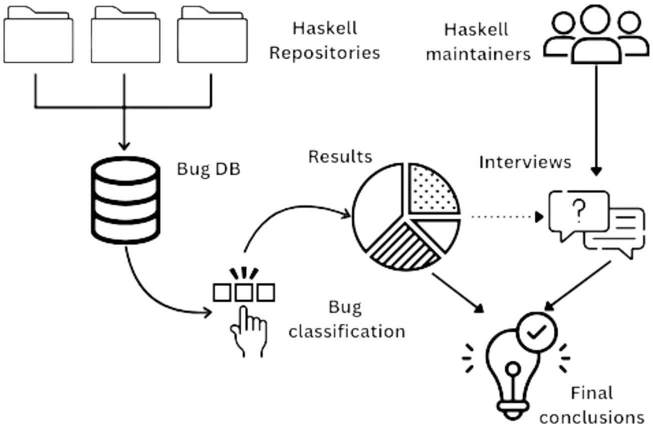

Figure 1: Overview of the workflow

Bug collection

The bug database contains 142 bugs, collected from 10 Haskell FOSS repositories. The repositories have been chosen to be as diverse as possible, to avoid bias towards any defect type. From each repository a number of 20 bugs were collected, or the maximum available.

The bugs have been taken based on the most recent closed issues with tags that contain the term “bug”. The issues were ensured to relate to commits or pull requests, which have been merged into the main branch. The code in the commits and pull requests was checked to include bug fixes for the specified bug report. Also, we ensured that the issues reported true bugs in the code, and we eliminated all duplicate reports in order to not skew the results. Also, the fix of the bug needed to include a change in at least one *.hs* file to ensure that they are bugs related to Haskell. Note that they can modify other types of files alongside the Haskell ones. Finally, the scope of this study only covers bugs in the source code, and no bugs related to test code have been considered, e.g. missing or failing tests.

Bug taxonomies

There is a large variety of taxonomies in literature, each with its own perspective on bugs. One of the most used types of classification in industry is the ODC [3]. It looks at bugs in the context of the program structure, adding them up in 8 types. This simple categorization scheme is well documented in literature and often being used as a starting point by researchers.

Seaman et al. proposed a taxonomy built on top of ODC [6], to unify the bug classification process within multiple NASA centers and projects. This classification was also used in an empirical study to test build systems done by Xia et al. [8], showing the versatility of this taxonomy outside of the context in which it was created.

The IEEE also proposed a standard for classifying software defects, meant to provide a “common vocabulary” to facilitate communication within the industry [1]. It proposes a small number of 7 defect types, where some terms are similar to the previously mentioned taxonomies, such as *Interface*, *Data*, and *Logic*.

Catolino et al. propose a different way of classifying bugs, which is argued in the paper to be usable in a “complementary manner” with ODC [2], and transitively with the other two taxonomies mentioned. It offers 9 types of bugs, including, among others, *Configuration*, *Performance*, and *Security*.

This study focuses only on two of such taxonomies, namely the one proposed by Seaman et al. [6], and the one from Catolino et al. [2]. The first taxonomy has been chosen for the adaptability of the schema, while also offering, on top of largely used ODC, the *Other* category, which can act as a flag if the rest of the categories are not able to fit the bugs found. The second taxonomy was picked because of its different perspective on characteristics of bugs, and the more project general terms it uses, e.g. *security* and *performance*. The two taxonomies, including the definitions used for each type, can be found in Appendix B.

In order to adapt these taxonomies to the scope of the project a few aspects need to be considered. Firstly, the taxonomy proposed by Seaman et al. includes defect types for requirements, source code, and test plan [6], but only the source code types have been considered for classification. Secondly, the taxonomy proposed by Catolino et al. has a special category for test code related issues [2], but, due to the scope of the study being limited to source code only, this category has not been considered.

Bug classifying

The bug dataset has been fully classified based on the two chosen taxonomies. The most relevant category from each of the taxonomies was the one chosen, but complex bugs, where a clear distinction was hard to make, have been documented. Findings from the statistical analysis of the bug classifications, alongside reflections on the usability of the taxonomies have been used to adapt the taxonomies, before asking for further opinions during the interviews. These findings are discussed in the next section. The bug dataset is available on 4TU.ResearchData at https://doi.org/10.4121/29a6a7bc-8b45-472c-bf26-710f2a2d1a3c.

Interviews

Four open source Haskell developers took part in a semi-structured interview. The lead questions can be found in

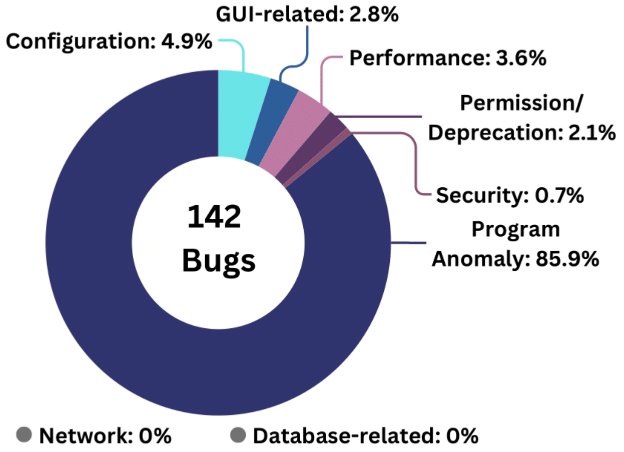

Figure 2: Percentages of bugs out of 142 bugs in each of the categories from the taxonomy proposed by Catolino et al. [2]

Appendix A. Approximately 30 people were asked to participate in the study, with a response and acceptance rate of approximately 13% . The interviews took approximately 30 minutes and were done online. They were audio recorded and transcribed. The participants were asked about how they identify, understand, classify, and fix bugs in practice, as well as what their opinion was on the proposed taxonomies. The data gathered during the interviews was used to construct a qualitative codebook, using a bottom-up approach and the findings are described in the next section. The full codebook can be found on 4TU.ResearchData at https://doi.org/10.4121/29a6a7bc-8b45-472c-bf26-710f2a2d1a3c, with an excerpt available in Appendix C.

3 Results and Discussion

This section presents the results for each of the research questions proposed in Section 1. In the latter part of this section, a discussion is presented, which also addresses threats to validity.

RQ$_1$ What are the most common types of Haskell bugs? To answer this question, two taxonomies have been used to classify bugs in Haskell FOSS repositories. This comprehensive classification of 142 bugs provides insights into the frequency of the types of bugs for both taxonomies.

Figure 2 displays the percentage of bugs, out of a total of 142, for each of the bug types proposed by Catolino et al. [2]. A predominant portion of bugs were classified as *Program Anomaly*, with 122, or about 86%, out of the 142 bugs being part of this category. The rest of the categories are rare in comparison, with the second most frequent type being *Configuration* with 7 bugs, or approximately 5%, followed by *Performance* (3.6%), *GUI-related* (2.8%), *Permission / Deprecation* (2.1%) and *Security* (0.7%). None of the bugs in the collected dataset were classified as *Network* or *Database-related*.

Figure 3, similar to the previous taxonomy, shows the percentages of bugs classified, from the same dataset, according to the taxonomy proposed by Seaman et al. [6]. This taxonomy is more balanced compared to the previous one, with the largest category, *Algorithm / Method*, only consisting of 26.8%, followed closely by the *Logic* type with 25.4%. The descending order of the rest of the types is as follows: *Data* (13.4%), *Checking* (9.9%), *External Interface* (8.4%), *Non-functional defects* (5.6%), *Internal Interface* (4.9%), *Timing / Optimization* (4.2%) and *Assignment / Initialization* (1.4%). All the bugs in the dataset have been categorized as one of the previously mentioned types, with none in the *Other* category.

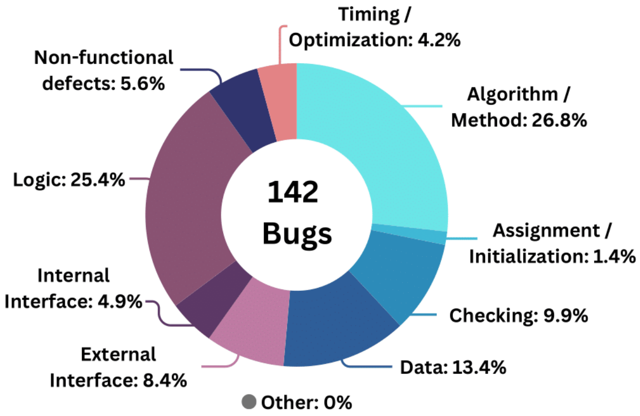

Figure 3: Percentages of bugs out of 142 bugs in each of the categories from the taxonomy proposed by Seaman et al. [6]

RQ$_2$ What are the limitations of existing bug taxonomies in capturing the unique features of Haskell bugs?

In order to understand the limitations of classifying Haskell bugs using these taxonomies, we first need to see how bugs in Haskell differ from ones in imperative languages. For the taxonomy by Catolino et al., we compare our findings with the original study, which mostly considers Java bugs [2]. In Figure 4 we show the differences per bug type. For the taxonomy by Seaman et al., because the frequency of bugs is not clearly stated in the original paper, we compare our results to the ones from Xia et al., which looks into Java, C, and C++ bugs from software build systems [8]. This comparison can be seen in Figure 5.

Both taxonomies show significant differences from the other studies. A large increase can be seen in the first taxonomy in *Program Anomaly* bugs, which are almost double in this study. All the other types are smaller in the current study, with *Configuration* and *GUI-related* showing a difference of about 10% and 15% respectively. The rest of the types, present less variations, with at most a 4% difference, with Performance having the same frequency. In the second taxonomy, *Algorithm / Method* shows a large increase of over 20%, while *Logic* increases with about 7%. A large decrease can be seen in *External Interface*, with a difference of more than 10%, and in *Assignment / Initialisation* with a decrease of 7%. The rest of the categories show a relatively small difference of less than 5%. Compared to our study, all types of bugs, from both taxonomies appeared, including *Network*, *Database-related*, and *Other*. It is important to note that the

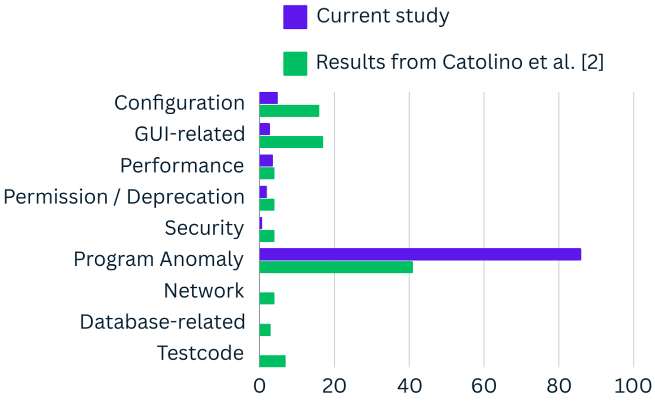

Figure 4: Differences in percentages of bug categories between current study and study by Catolino et al. [2]

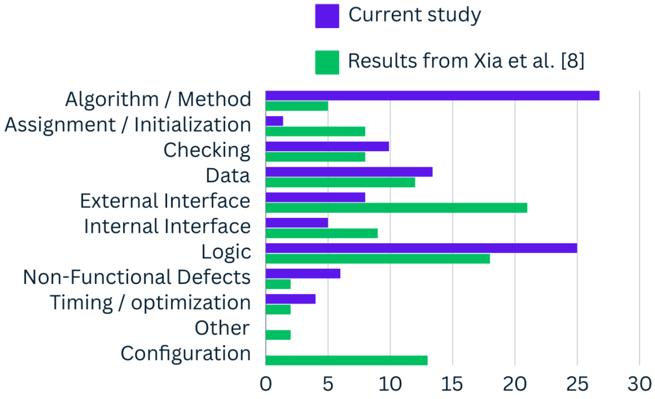

Figure 5: Differences in percentages of bug categories between current study and study by Xia et al. [8]

study from Catolino et al. also classified *Testcode* bugs [2], which was not considered in this study, and it was kept in the comparison for completeness . Similarly, Xia et al. added a class for *Configuration* bugs [8], which does not appear in the taxonomy we used.

Difficulties in the classification process, alongside the comparison presented above have revealed some limitations when using such taxonomies in Haskell.

First, there is a large portion of bugs being classified as Program Anomaly in the taxonomy proposed by Catolino et al. [2]. Similar to our findings, Program Anomaly was also found to be much more frequent in the original study, with a percentage of 41% compared to the second most common category, GUI-related issues, which accounted for 17% [2]. This large variation in the proportions of types suggests that not a big enough distinction is made within this type. Thus, not enough information would be available for developers classifying a bug in this category. As the purpose of this taxonomy is to help in bug triage, this type should be split into several, smaller types, that would each then offer more valuable insights.

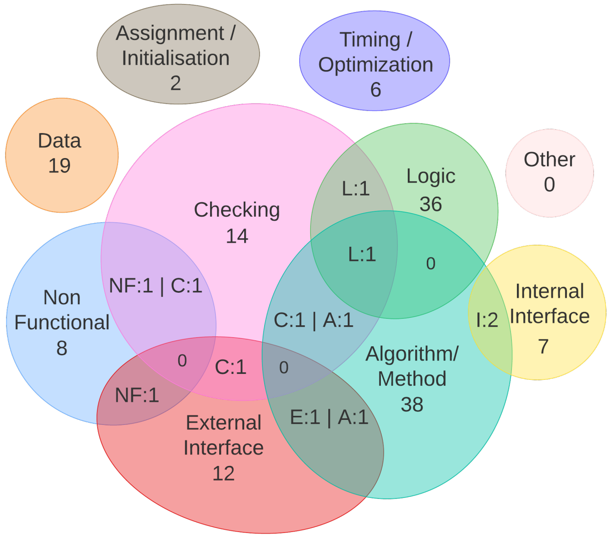

Figure 6: Venn diagram showing the number of bugs per type including the overlapping types of the complex bugs. The numbers in the intersections indicate how many bugs, out of their chosen category (indicated by the abbreviation), could be assigned to the rest of the overlapping types.

Second, some bugs exhibited characteristics that could fit into multiple types within the taxonomy proposed by Seaman et al. [6]. Because of their difficulty in classification, we named them *"complex bugs"*. There is a total of 12 complex bugs. Figure 6 is a Venn diagram showing the relations between the types of these complex bugs. As all the bugs were allowed to be classified into only one type, the most representative class had to be chosen. The diagram displays the number of bugs in each category below the category name. In each intersection, the diagram shows the number of bugs, out of their chosen category, that could also be classified into other categories within that intersection. For example, out of 8 Non-functional bugs, one of them could also be classified as Checking. Similarly, out of 14 Checking bugs, one could also fit in the Non-functional category. Even though there are 2 types of bugs that cover 7 out of the 12 bugs, namely Checking and Algorithm / Method, no clear pattern seems to emerge in the relationships between categories. Thus, no general method could be identified, and they had to be classified on a case-by-case basis.

Also, some of these types are greatly reduced in Haskell, which was also remarked by some of the interviewees. For example, the Assignment / Initialisation type is only found in 2 bugs from the dataset, which is 1.4% compared to around 8% as found by Xia et al. [8]. While this category emphasizes issues related to mutable states, in Haskell, such bugs are not possible due to the language's immutable bindings. Thus, leaving the only bugs that can fit into this category being related to wrong bindings, or to changes in function signatures.

Lastly, the explanations of the bug types were found to be slightly ambiguous and not mutually exclusive. This might be a potential cause for the existence of complex bugs.

RQ$_3$ How do Haskell developers classify bugs and how is it different from the previously discussed taxonomies? To find out how bugs are treated in practice, we performed interviews with four Haskell developers. An excerpt of the codebook created from the interviews can be found in Appendix C.

Several types of bugs were a common trend between the interviews. The most common types mentioned include performance bugs, requirements issues, logic problems, wrong understanding of the problem, interface problems, and validation issues. Most of the participants argued that the most difficult bugs to solve are memory leaks, usually related to the laziness of Haskell. P1 stated that these bugs are the "trade off" when using Haskell, as trying to inspect the state of the program is "changing the laziness behaviour", as explained by P3.

While the bugs mentioned seem to form a pattern, the interviewees could not agree on what the most frequent types of bugs encountered are. This was found to be very dependent on the projects and experience of the participants. The views were split on logic bugs, wrong understanding of the problem, problems with external interactions, and performance bugs. Also, while one interviewee stated that performance bugs are the most common type they have to fix, others mentioned that they never had to deal with issues related to this.

In practice, bug classifications seem to happen rarely, half of the interviewees agreed that they do not use any sort of categorization of bugs. The other half also stated the same at first but changed their minds throughout the discussion. One of the interviewees stated that they use priorities, e.g. safety critical, and labels with the location in the code, such as Backend and Dashboard. Also, P1 remarked that they use bug classification implicitly when discussing about bugs, for example by mentioning memory leaks.

When asked about the usability of the taxonomies, all the interviewees agreed that they need to be modified before they can prove useful. Some of the interviewees argued that the taxonomies are focused on "procedural languages" (P3), and "java-like programs" (P4), missing relevant features of strictly typed languages such as Haskell. Also, P1 argued that certain types, such as Assignment / Initialisation, are greatly reduced in Haskell. Some suggestions to improve the taxonomies include adding types, such as modeling issue or misunderstood requirements; splitting up types for better information, such as GUI being split into stylistic issues and impacting user flow; and removing types, for example, the Data type. Also, it was argued that classification is a time intensive process, which might not provide helpful for smaller teams, or for projects that have very tight deadlines.

While the taxonomies require modifications, and certain considerations, the participants found such taxonomies to be promising in practice. They argued taxonomies could prove useful for documenting, bug triage, developing steps to avoid frequent types of bugs in the future, and guiding hiring decisions.

Discussion Based on our results, the taxonomy proposed by Seaman et al. seems to be better suited when being used on Haskell bugs, while the one proposed by Catolino et al. does not make a big enough distinction between types to be helpful in bug triage. However, these findings are subject to certain threats to validity, which are discussed below.

Although these findings can mostly be attributed to the differences in the programming languages, there is the risk that the differences in results come from the different usages of the languages. However, this risk is diminished by having a diverse set of Haskell projects, that overlap in usage with projects written in imperative languages. For example, Cabal, which is one of the repositories used in this study, is a build system, similar domain as the projects considered by Xia et al. [8].

In the classification step, some adjustments had to be made to encompass some of the bugs found. None of the repositories chosen have a graphical user interface, however, there were still bugs related to the user interface. As these bugs would not match in any of the other categories presented by Catolino et al., the graphical component was considered in this study an optional requirement instead, and the bugs were classified as such.

There is no known ground truth to the classification of bugs and there are multiple factors that have hindered the process. For example, the lack of standardization in the documentation and bug reports, made understanding and classification of bugs harder. Similarly, the defect types chosen for the complex bugs could have slightly skewed the results. All assumptions made throughout this study were documented to ensure the process remains as objective and as reproducible as possible. However, certain limitations, such as limited time, relatively small dataset, and the small number of perspectives that were used in classifying, have to be acknowledged and ultimately, a larger study focusing on increasing the number of classifications is necessary.

While the comparisons in figures 4 and 5 can be representative of how the frequency of bugs changes between types, the actual numbers might be slightly different. One of the main reasons is the inclusion of the types that were not present in this study, namely *Testcode* and *Configuration*. By including them, the percentages of the other types of bugs are slightly lower than they would be otherwise. However, the results were taken as found in the studies mentioned, to ensure objectivity and completeness.

The percentage of bugs classified as Program Anomaly in this study is almost twice as large as the percentage reported by Catolino et al. The large discrepancy between the findings of the two papers might be partially attributed to the methodology of the study and the way the bugs were chosen. For example, because the focus of the study is only on Haskell bugs, the bug fixes had to touch upon at least one *.hs* file. However, by doing this, multiple bugs in a repository might be missed, such as configuration bugs which often appear in *.yml* files instead of *.hs* ones. To get a full understanding of how bugs in other parts of the code relate to the types of bugs found in Haskell, further study is required. Also, this can be a starting point to look at bugs in repositories that are constituted of other programming languages in addition to Haskell and how bugs might be correlated between the different languages.

The lack of Database-related and Network bugs might also, in addition to the reasoning given above, be attributed to the choice of repositories. Even though the repositories were picked with diversity in mind, none of the repositories chosen ended up having bugs related to these types. Due to the limited time-frame of this project, investigation into this was not possible, but further research could touch upon more types of repositories to get a better understanding of missed types of bugs. This future search is also made promising by claims made by some of the interviewees, who argued that they have come across plenty of bugs in these categories.

Similar to the bug classification, in order to identify what the most encountered type of bugs are, a larger number of developers is required. This is especially true when the frequency of bugs encountered by an individual is greatly impacted by the area of their expertise and the projects they work on. While some participants argued that Database related issues are less likely to appear in practice, others stated that they happen quite often.

For a taxonomy to prove useful to a developer, they first need to identify their needs and objectives. There is a great number of taxonomies proposed in literature, and each of them can be modified in numerous different ways to better reflect what one individual encounters in practice. As P2 stated, one needs to find “the purpose of organizing or classifying [...] otherwise there are too many options”.

Finally, this study only focused on bugs present in source code. But as mentioned by multiple interviewees, a large number of bugs are present in the modeling and requirements stages. Thus, researching a broader view of the projects would benefit the community by understanding how bugs look like throughout all the steps of development, from requirements to test code.

4 Responsible Research

To ensure the integrity of this study, multiple (ethical) aspects had to be considered during the research period.

Firstly, the data collected was ensured to be FAIR [7], to assist in the reproducibility of the study.

Findable: This is ensured by sharing the collected data on *4TU.ResearchData*.

Accessible: Similarly, by storing the data on *4TU.ResearchData* in a public way, accessible data is ensured.

Interoperable: To ensure this, the data is shared in formats that allow for easy and unrestricted usage. The bug database is stored in a CSV format, and the codebook in a PDF file.

Reusable: This is ensured by the accompanying documentation, which describes the collection of data, as well as the structures of the bug database and the codebook. For example, the bug database contains the following data: bug name, repository it was taken from, as well as a link to the bug report, the classification based on the Catolino et al. taxonomy [2], and the Seaman et al. one [6], and notes that explain the decision. The documentation also explains the types that each of the two taxonomies present. The codebook contains themes, codes, and their description as well as examples and partial fit examples taken from the interviews.

Secondly, the interviews pose ethical concerns due to higher risks for the participants. These risks have been considered and mitigated to the best of our ability. Some example risks are data breach (of potentially sensitive data), which is mainly handled by storing the data securely with access only to the study team, and re-identification, which is a risk minimized by deleting all sensitive data and ensuring that all quotes taken from the interviews are anonymized.

The interviews have been audio recorded and transcribed. Because these pose a high risk of re-identification of the participants, these were stored in a secure OneDrive file in a MS Teams channel with access only to the study team. After the transcripts were checked for correctness, the recordings were deleted to minimize the risks from a potential data breach. The transcripts were used to create an anonymized qualitative codebook, with quotes and insights, that were ensured to not contain potentially sensitive data. After the study, the transcripts were deleted for similar reasons to the deletion of the recordings. The codebook was also stored in the same secure OneDrive, Teams channel with the rest of the data. Also, all contact details have been deleted with the deletion of the rest of the data collected in the interview.

The interviewees were gathered through private networks, to minimise the risk of identification. They were informed about what was expected from them and what will happen to the collected data. They signed an informed consent form, which is stored in a private Project Drive with access only to the study team. They were informed that they could withdraw at any moment, until the creation of the codebook, and any data collected until then would be deleted. This is due to the anonymity of the codebook, as identifying the data collected from a specific participant is no longer possible. They also gave their consent to the codebook containing anonymized quotes and for it to be publicly available on 4TU.ResearchData.

The data management plan and the risks posed by the interviews, which have been described above, have been reviewed by the TU Delft Electrical Engineering, Mathematics and Computer Science faculty Data Steward, as well as approved by the Human Research Ethics Committee of TU Delft.

5 Conclusions and Future Work

This paper is meant to be a starting point in understanding how bugs differ between functional programming and the more explored imperative languages, such as Java. For this, two bug taxonomies have been used to classify 142 bugs from 10 Haskell FOSS repositories, in order to understand what types of bugs appear most in Haskell, and if such taxonomies are capable of handling functional languages while also being useful to Haskell developers. The vast majority of bugs were classified as Program Anomaly bugs, which also suggests a limitation to this taxonomy that is not capable of offering all the information necessary for developers to effectively perform bug triaging. The other taxonomy offers a more balanced classification in the types, with the majority of bugs being either classified as Algorithm / Method or as Logic. However, some bugs were identified as complex, as they could be part of multiple types. Four Haskell developers expressed their experience with bugs and offered their opinions on the usefulness of these taxonomies. While they identified some key areas where these taxonomies can be useful such as triage, documentation and management decisions, bug categorizations need to be chosen and adapted to the specific needs of the team and project.

Future work could perform bug classification on a larger scale to assess the limitations of this paper. Thus, allowing more people to classify a larger set of bugs from more projects would allow for better modeling of the bugs found in Haskell projects. Also, it would enable a better comparison with bugs from non functional languages. Additionally, it can extend the research to test code and requirements, which can create a better overview of the types of bugs in the overall projects. Future studies could also gather the perspectives of more developers to better understand the processes employed when debugging. P1 argued that it is a “hurdle” to use debugging tooling as you need experience to be able to identify the problem. Thus, research could help fit these types of tools better to the developers’ needs. Finally, all the findings from this paper and from any future work can be combined and used to improve on the taxonomies proposed to make them better fitted to the wild Haskell bugs.

6 Acknowledgements

Special thanks to the peer group that offered their support for the entirety of this Research Project. Also, we thank the interview participants who offered the experience and expertise required to complete this study.

References

[1] IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), IEEE Standard Classification for Software Anomalies.

[2] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 152:165–181, June 2019.

[3] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

[4] Ruanqianqian (Lisa) Huang, Elizaveta Pertseva, Michael Coblenz, and Sorin Lerner. How do Haskell programmers debug? *Plateau Workshop*, March 2023.

[5] Rafael-Michael Karampatsis and Charles Sutton. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577, Seoul Republic of Korea, June 2020. ACM.

[6] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 149–157, Kaiserslautern Germany, October 2008. ACM.

[7] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C ’t Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, March 2016.

[8] Xin Xia, Xiaozhen Zhou, David Lo, and Xiaoqiong Zhao. An Empirical Study of Bugs in Software Build Systems. In *2013 13th International Conference on Quality Software*, pages 200–203, Najing, China, July 2013. IEEE.

A Interview Questions

The interviews were semi-structured and mostly followed the questions below. It is important to mention that sometimes the questions were altered, or their order changed, or extra questions were asked based on the flow of the conversation.

Bugs in practice

1. What is usually the first reaction when you encounter a bug?
2. Do you have a general method for locating bugs? If not, can you walk me through how you were able to locate the last bug you encountered?
3. Do you have a general method for fixing bugs? If not, can you walk me through how you were able to fix the last bug you encountered?
4. What are the most common types of bugs you have encountered?
5. What are the types of bugs you spend the most time on?
6. What are the most tricky types of bugs you had to deal with?
7. Do you use any classification for bugs in your projects?
8. Do you ever push buggy code intentionally?

Opinions on the taxonomies

1. What category of each of the two taxonomies presented do you think will appear most often? How about the least?
2. Do you think there is a category that will not appear at all?
3. Do you think there are bugs that won't be able to be classified by these taxonomies?
4. Do you think that these taxonomies can be useful in practice?

B Bug taxonomies

Table B.1: Bug types proposed by Catolino et al. [2].

Defect Type	**Definition**
Configuration issue	Bugs concerned with building configuration files. Most are problems with: (i) external libraries that should be updated or fixed; (ii) wrong directory or file paths in xml or manifest artifacts.
Network issue	Bugs having connection or server issues, due to network problems, unexpected server shutdowns, or communication protocols that are not properly used within the source code.
Database-related issue	Bugs that report problems with the connection between the main application and a database. Note that bugs in SQL statements are also part of this category.
GUI-related issue	Bugs occurring within the Graphical User Interface (GUI) of a software project, such as stylistic errors (layouts, padding, buttons), and unexpected failures appearing to the user.
Performance issue	Bugs that report performance issues, including memory overuse, energy leaks, and methods causing endless loops.
Permission / Deprecation issue	Bugs related to: (i) the presence, modification, or removal of deprecated method calls or APIs; (ii) unused API permissions.
Security issue	Problems related to vulnerabilities in the system.
Program Anomaly issue	Bugs concerned with specific circumstances that appear when enhancing existing source code, such as exceptions, return values, issues in logic.

Table B.2: Bug types proposed by Seaman et al. [6].

Defect Type	**Definition**
Algorithm / Method	An error in the sequence or set of steps used to solve a particular problem or computation, including mistakes in computations, incorrect implementation of algorithms, or calls to an inappropriate function for the algorithm being implemented.
Assignment / Initialization	A variable or data item that is assigned a value incorrectly or is not initialized properly or where the initialization scenario is mishandled (e.g., incorrect publish or subscribe, incorrect opening of file, etc.).
Checking	Inadequate checking for potential error conditions, or an inappropriate response is specified for error conditions.
Data	Error in specifying or manipulating data items, incorrectly defined data structure, pointer or memory allocation errors, or incorrect type conversions.
External Interface	Errors in the user interface (including usability problems) or the interfaces with other systems.
Internal Interface	Errors in the interfaces between system components, including mismatched calling sequences and incorrect opening, reading, writing or closing of files and databases.
Logic	Incorrect logical conditions on if, case or loop blocks, including incorrect boundary conditions (”off by one” errors are an example) being applied, or incorrect expression (e.g., incorrect use of parentheses in a mathematical expression).
Non-Functional Defects	Includes non-compliance with standards, failure to meet non-functional requirements such as portability and performance constraints, and lack of clarity of the design or code to the reader both in the comments and the code itself.
Timing / optimization	Errors that will cause timing (e.g., potential race conditions) or performance problems (e.g., unnecessarily slow implementation of an algorithm).
Other	Anything that does not fit any of the above categories that is logged during an inspection of a design artifact or source code.

C Excerpt Codebook

Table C.1: Excerpt from the codebook created from the interviews with the Haskell developers.

Theme	Code	Explanation	Examples	Partial-fit Examples
Bugs in practice	Bug types	Types of bugs mentioned by the interviewees or used by them in practice	• Performance • Memory / Space leaks • Logic bugs • Wrong understanding of the problem • Type errors (not caught by compiler) • Parsing bugs (bugs happening when parsing data to get it in the application) • Validation issues • Regression • External Interaction • Name shadowing (accidentally used a variable from higher up and not the current state) • Requirement and environment bugs (wrong assumptions about the environment the code is running in) • Expectations of the behaviour of an API • Type Tetris (someone picked a function which seemed to fit the complicated types they needed, but it does something completely wrong) • IO runtime errors • Security relevant • Compile time errors	• Wrong variable • Wrong type • Forgot one edge case • Not working how you expect it • Wrong thing you implemented • Missing stuff • String conversion that didn't work • Something out of scope of the Haskell type system • Interactions with the other programs other file formats other interfaces and that's where it's the most easy to make mistakes • Haskell type safety erodes at the edges

Continued on next page

Theme	Code	Explanation	Examples	Partial-fit Examples
Bugs in practice	Bug classifications	Different ways of classifying a bug explained by the interviewees	• No classification system • Implicit classification • Classification by importance • Safety critical • Classification based on location of bug (e.g. server, dashboard) • Easy vs tricky bugs (based on time spent) • Treat bugs mostly with the same diligence and priority • Flexible way of combining bug tags. • Naming such as refactoring, or feature.	
Taxonomies	Suggestions	Suggestions given to improve the taxonomies, e.g. adding types.	• Adding the Misunderstanding requirements type • Splitting GUI into style related and impacting user flow. • Splitting Program Anomaly because the kind of logic error is very important • Database-related is redundant because problems with connection are actually network issues and a bug in SQL is an issue with Logic. • Separation between validation and proper program logic. • Data type is redundant as it is not big enough in Haskell • Separation of non-functional defects in the taxonomy by Seaman et al. in a similar way to how Catolino et al. does, so permissions and performance. • Adding the modeling issues type. This includes both writing the model and also implementing something that fits the model.	• Logic bugs sounds very broad • Hard to identify differences between Assignment / Initialization and Checking • Strange separation between Logic and Algorithm / Method • Taxonomies are more relevant at the later stages of a project when moving from the prototype to an actual production system • Network issues are part of Configuration issues but on the network stack.

Continued on next page

Theme	Code	Explanation	Examples	Partial-fit Examples
Taxonomies	Usefulness	How useful the interviewees perceive the taxonomies	• Only some categorizations help • Multiple bugs in a category allow for creation of steps that help avoid that category in the future • Taxonomy proposed by Catolino et al. is more useful than the one by Seaman et al. • No usefulness in programming, but it is in research. • Great for documenting • Identify areas where a developer (team) needs to learn more. • Change from gut feeling debugging to classification guided • Useful for bug triage. • Useful in larger projects and teams, but take too much time for smaller ones. • Helpful for hiring purposes. • Indicators for better tests • Identify purpose of using such a taxonomy, before choosing it. Otherwise there's just too many options to choose from and becomes unhelpful.	• People lack a vocabulary when discussing about bugs. • Taxonomies are for procedural languages, such as for Java-like programs. • Taxonomies lack elements from strictly type languages, like Haskell.