Bugs in Haskell Programs

What are the different stages of bugs in Haskell programs?

Amy van der Meijden[1]
Supervisors: 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: Amy van der Meijden
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Leonhard Applis, Koen Langendoen

Abstract

Various studies already exist about the lifecycle of software programs written in languages like Java, C and C++, but this is an under-reported area for the pure, functional programming language Haskell. This report explores steps in the development of Haskell programs, and particularly, of their bugs. By gaining more knowledge about a bug's development and its different stages, possible patterns or trends can be found. Using these insights, developers could get a better understanding of the development process, and optimise it. The main question of the research is formulated as **'What are the different stages of bugs in Haskell programs?'**. We consider three different stages: bug introduction, bug detection and bug fixing. To gain more knowledge about these stages, different open-source Haskell repositories, and their bugs were analysed. We found the median time between the bug introduction and detection is 381 days. The median time between the bug detection and fixing turned out to be 3 days. Most bugs were detected and fixed at similar rates regardless of their complexity, except for bugs that were detected by the same developer who introduced the bug. In this case, the time it takes to detect the bug is 30% of the global median. These results motivate developers to check their code more extensively, as they detect bugs quicker than other developers would. It also motivates developers to work with shared code ownership, as having more developers familiar with the code should reduce bug-detection time. Moreover, this study shows that single-statement bugs, which needed a simple fix, occur 2.5 times more often in code when no test is written, which argues for developers to write more test code. Lastly, we recommend repository owners to be more strict with developers working structured and precisely so that they follow Git's and the repository's standards and write test code.

1 Introduction

When writing a software program, bugs in the code are almost inevitable. Some bugs can easily be detected and fixed quickly, while the origin of some might be harder to detect. This could be, for example, because a piece of code did not appear as 'buggy' at first. It might be easy to fix the bug, yet it can take some time to detect where the bug occurred in the code. For many programming languages like Java, C, and C++, there are already various studies about the software's development process or lifecycle [21, 6, 25]. Moreover, research has been done on the lifecycle of bugs in e.g. Linux kernel [5]. If it is possible to detect bugs early in the development process, it can save the developers a significant amount of time. Research on this lifecycle rarely exists for the functional programming language Haskell, which will be the focus of this paper. The main research question is formulated as follows: **'What are the different stages of bugs in Haskell programs?'**. This question is broken down into the following sub-questions:

- **RQ1:** Which, if any, definition of bugs is most applicable for open-source Haskell repositories?
- **RQ2:** How much time or commits pass between the introduction and detection of bugs? Are tests written before the detection?
- **RQ3:** Are bugs typically detected by the same developer who introduced them?
- **RQ4:** How much time or commits pass between the detection and fixing of bugs? Are tests written after the detection?

- **RQ5:** Are the so-called Simple Stupid Bugs detected earlier than other bugs?

The results of this study provide information on the development of 'buggy' code and insights into the different stages of a bug. Possibly, patterns or trends can be found in the development process. One hypothesis is that bugs are detected earlier when they are found by the same developer who introduced them, since they know best what 'state' their code is in. Another hypothesis is that the detection of bugs takes longer when tests are not written beforehand, as a failing test is a way to detect a bug. By finding these patterns or trends, developers could gain a better understanding of the development process, and optimise it.

The first step in answering the sub-questions, is to study relevant literature on Haskell, (typical Haskell) bugs, Simple Stupid Bugs, and Bug-Introducing Commits. After the literature research, a sample of open-source Haskell projects and their commits over the last four years will be gathered. From this collection, the bug-related commits will be identified and used to retrieve the moment of introduction, detection, and fixing of bugs. Using this information, we answer sub-questions 2 to 5.

The remainder of the paper is structured as follows. In Section 2, background information for the research will be given. Topics such as Bug-Introducing Commits will be covered. In Section 3 the methodology for the study will be explained and in 4 the results will be shown and discussed. In Section 5 the ethical aspects of the research will be discussed. Lastly, in Section 6, there will be a conclusion of the research and possible future work will be discussed.

2 Background

2.1 Simple Stupid Bugs

Simple Stupid Bugs (SStuBs) are bugs of which the fixes are 'so simple that many developers would call them "stupid" upon realization' [14, p. 573]. The code compiles before and after the repair, and consequently the SStuBs are quite tedious to manually spot. In the research from Mosolygó et al., the lifecycles of SStuBs in different Java programs were studied [18]. To find the source commit (more about this in Section 2.2), the 'git blame' command was used. This returned the revision and author that last modified each file line. Mosolygó et al. found that SStuBs seem to appear more in larger chunks of code written by the same developer. This means that these mistakes come up likely due to loss of attention, rather than a lack of understanding about the code. The researchers also concluded that frequent changes do not significantly increase SStuBs' likelihood. Most SStuBs are namely added in the same commit as their neighbouring lines. In the research the lifetimes of SStuBs were also analysed; it appeared that developers who check their own code find the issues more quickly than if someone else were to look for a bug.

Eiroa-Lledo et al. tried to determine when Simple Stupid Bugs were first introduced to projects [7]. The researchers also explored the factors that drive the time it takes to fix these bugs. In the study, due to the varying size and activity per project, the metric 'Commit Cycles' is introduced to normalise time differences. This metric will also be used to express time differences in this works' methodology. The number of Commit Cycles is calculated by taking the time difference in minutes between introducing the buggy code and fixing the code and multiplying this by the 'Commit Rate'. The Commit Rate describes the number of commits a project introduces per minute. The results of the study show that the best bet to have short-lived bugs is to have the same people who created the bugs, fix them.

2.2 Bug-Introducing Commits

A Bug-Introducing Commit (BIC) is a commit that introduces a bug in a program. The SZZ algorithm was one of the first algorithms to identify BICs in a software repository [30] and the basis of other algorithms. The method of the algorithm is as follows: (1) Start with retrieving a bug from a bug report, indicating a fixed problem, (2) Extract the associated change from the version archive, which gives the location of the fix, (3) Determine the earlier change at this location. This is the fix-inducing change.

In 2020, Rodríguez-Pérez et al. distinguished two different kinds of bugs: intrinsic (the origin is a bug-introducing change in the source code management) and extrinsic bugs (the origin is a change not recorded in the source code management)[23]. A model was proposed for defining criteria to decide the first snapshot of an evolving software system that exhibits a certain bug. By evaluating four SZZ algorithms, the researchers showed that the assumption 'a bug was introduced by the lines of code that were modified to fix it' only holds for 61 to 63 per cent. Rodríguez-Pérez et al. proposes a model for identifying the first snapshot of an evolving software system that exhibits a bug. The process used by the proposed model is as follows:

- Ensure that a Control Version Exists.
- Identify the Bug-Fixing Change.
- Ensure the Perfect Fixing: If the Bug-Fixing Change is incomplete or spread over several commits, there is no perfect fixing.
- Describe whether a bug is present.
- Identify the First-Failing Change.

Izquierdo-Cortazar et al. analysed BICs of the comm-central FLOSS project and tried to find out whether the developer who introduced/seeded a bug, is the same developer who fixed the bug [8]. Using the SZZ algorithm, the previous commits for each bug-fixing change were identified. Different scenarios were created: (1) The bug is handled by the developer who introduced it, (2) The bug is handled by the developer who introduced it and other developers, (3) The bug is handled by different developers. It appeared that in general, the committer who has fixed a bug has not participated in seeding that bug.

Using two open-source projects in ArgoUML and PostgreSQL, Kim and Whitehead investigated how long it takes to fix bugs, which should indicate the software quality [15]. The bugs in the projects were identified by searching for keywords such as 'Fixed' or 'Bug', and searching for references to bug reports. The researchers concluded that the bug-fix times in buggy files range from 100 to 200 days.

Karampatsis and Sutton analysed single-statement bugs in Java [14]. In their study, a simple bug is defined as a one-line bug or a bug that falls into a small set of templates and they found on average 0.75 SStuBs per valid commit. The researchers classify a commit as bug-fixing by checking if its commit message contains at least one of the following keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw', and 'type'. We assume that the keyword 'type' is being used, for the case that an incorrect (data)type, with incorrect properties is used. In the study of Ray et al., it has been shown that this heuristic achieves 96% accuracy on a set of 300 manually verified commits [22].

3 Methodology

In this section, we will describe the methodology of this research. Based on the relevant literature research, we assume that a bug knows three different stages: bug introduction, bug detection, and bug fixing. The methodology for the remaining research is split into two parts: collecting data and processing data. The procedure of collecting data is also displayed in Figure 1.

3.1 Collecting Data

Firstly, we collect and clone different open-source Haskell repositories from GitHub [4]. The following repositories are being used: Cabal [9], Hakyll [12], Haskell Language Server [10], Hledger [26], Pandoc [13], Purescript [20], Shellcheck [16], HLint [19], and Wire-server [29]. For each repository, we obtain the total number of considered commits, the considered time frame, and the Commit Rate. The goal is to find 100 bugs in total, spread over these repositories.

Using the 'git log' command, we retrieve the commits of the repositories from the last four years. Thereafter, we pick the ones that change a Haskell file (so a '.hs' extension) and filter the Bug-Fixing Commits (BFCs) by checking if the commit message contains at least one of the following words: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw', and 'type', like used in the research of Ray et al. [22]. In addition, the commits are filtered on the words 'typo' and 'fault'. Depending on the number of commits left, we pick a sample (around 30) to reduce the number of BFCs that must be checked manually. These commits in the sample are picked arbitrarily, to make them as representative as possible for all commits found. By analysing the commits and studying relevant literature, we will answer RQ1.

For identifying the Bug-Introducing Commit (BIC) corresponding to the BFC, we use the model proposed in 2020 by Rodríguez-Pérez et al. [23]. For some BFCs, a collaborator already mentions where the bug was introduced in the corresponding Pull Request. From manually analysing the BFCs, it appeared that not all of them had a specific BIC that could be linked. We found the following reasons:

- There were multiple BICs.
- The BFC originated due to an external source being updated. This means there was no obvious bug, since the code did not behave incorrectly before.
- The BFC was a fix in a README or documentation file, e.g. a typo. This did not change the program's behaviour and thus is not counted as a bug.
- The BFC originated when an error was already thrown, but the error message was modified. This is only an improvement of the code and not a fix.
- The BFC was a cleanup or refactoring of the code. One could argue that bugs can be fixed during refactoring. However, a single BIC cannot be linked.
- The BFC was to add a new test to the code, which does not make the program behave differently. If some test code itself was changed (e.g. a test passed while it should not), it did count as a bug.

Collecting the BIC per BFC results in a JSON file such as in Appendix A, where data is redacted for readability.

After collecting the BFCs, we retrieve the corresponding Pull Request[1] (PR) in GitHub. This is done by performing the 'gh pr list' command, filtering on the PR being merged, and searching for the PR which contains the BFC's hash.

Using the body of the PR, we can retrieve the corresponding issue[2]. This is done by matching the PR body to 'KEYWORD #ISSUE-NUMBER' [3]. GitHub namely supports the following keywords to link a pull request to an issue: 'close', 'closes', 'closed', 'fix', 'fixes', 'fixed', 'resolve', 'resolves', and 'resolved'. For example, if 'closes #20' is in the PR's body, the PR is linked to issue 20. Using the issue of a Bug-Fixing Commit, we know that a bug has been detected. The creation date of the issue is used as the date of the bug detection and the creator of the issue is used as the bug detector. During the study, it appeared that for 59 Pull Requests a linked issue could not be found automatically using the Git keywords. These PRs were collected and analysed manually. In 15 of the 59 cases, there appeared to be a corresponding issue, which was not linked using the Git keywords. In the other cases, where still no issue was found, the bug detector was set to the creator of the PR and the bug detection date was set to the PR's creation date.

We examine each pair of Bug-Introducing Commit and Bug-Fixing Commit to determine whether or not any tests were present in the code, prior to or after the bug detection. If this is the case, the date of merging the test code is saved. However, during the study, linking bugs to tests proved to be difficult due to the complexity of various projects. Each repository contains multiple test files and directories, and there is no clear link between the committed code and the test code. This makes it hard to say whether a test is difficult to find, or non-existent. Instead, each BFC has been analysed to determine whether a test in the PR was either added or modified. If a test was modified in the BFC's Pull Request, we assume the test must have been present before detection. Otherwise, a test would have been written after detection, which does not catch the just-detected bug.

For each bug, we also determine whether it is a Simple Stupid Bug (SStuB). We do this by looking for single-statement bugs, keeping in mind Karampatsis and Sutton's definition: 'bugs that compile both before and after repair as such can be quite tedious to manually spot, yet their fixes are so simple that many developers would call them "stupid" upon realization' [14, p. 573]. During the research, we also classified a bug as a SStuB when a developer apologises for introducing the bug in the PR and/or issue body, or calls it a 'silly mistake'.

[1]A Pull Request is a proposal to merge a set of changes from one branch into another [2].

[2]Issues are items you can create in a repository to plan, discuss and track work [1].

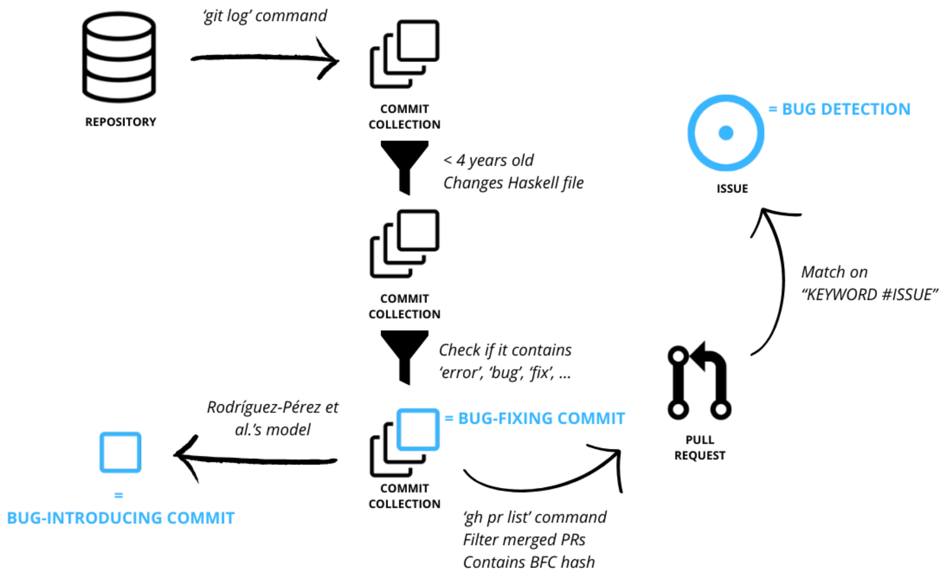

Figure 1: Schema of data collection.

3.2 Processing Data

To answer RQ2 and RQ4, we use the metric 'Commit Cycle' to express the time differences between bug introduction and detection and between detection and fixing (see Section 2.1). Using this metric, the time difference will be relative to the frequency of commits being done in the repository. This ensures that the results per repository can be compared to each other. We calculate the number of Commit Cycles by taking the difference in hours and multiplying it by the 'Commit Rate'. The Commit Rate for each repository is the total number of commits divided by the difference in hours between the first and last commit. To answer RQ2, calculating the number of Commit Cycles between the introduction and detection is done by first taking the time difference between committing the BIC and the issue's (or PR's) creation, and then multiplying this by the Commit Rate. The calculation of the number of Commit Cycles between the bug detection and fixing is done similarly, but the difference between the issue (or PR) creation and merging the BFC is taken. We answer RQ3 by checking whether the committer of the BIC is the same as the user who created the issue, and calculating for how many BICs this was the case. We answer RQ5 by first determining which BICs can be defined as a Simple Stupid Bug, and then analysing whether the number of Commit Cycles between the bug introduction and detection is less than for the non-SStuBs. Processing this data, results in a JSON file like in Appendix B. Again, data has been left out to make the file more readable.

4 Results

RQ1: Which, if any, definition of bugs is most applicable for open-source Haskell repositories?
After studying the relevant literature on how a bug is defined, we found that there is currently not a single definition of a 'bug'. Often 'bug' is used interchangeably with terms like 'defect',

‘error’, or ‘mistake’ [24, 17]. In some cases, a ‘bug’ is defined as a combination of terms. For example, in the research of Rodríguez-Pérez et al., ‘bug’ refers to both defects/faults and failures, depending on the context [23]. The term ‘bug’ is mostly used as an umbrella word because of its intuitiveness and wide usage [17], which refers to a failure of the software product’s requirements or specifications to be correctly implemented [27, 23, 11]. In this study, the following definition of a bug for open-source Haskell repositories is applied.

In a piece of code, there is a bug if at least one of these points is true. The issue linked to the BFC ...:

- ...mentions that the program produces incorrect output or makes a test fail.
- ...mentions that the program unexpectedly throws an error.
- ...had the label ‘bug’ assigned to it.
- ...mentions that a test passes, while it should not.

In the following cases, we do not say that there is a bug in the code:

- An import statement was changed because an external source got updated.
- There was a fix in a README or documentation file.
- An error was already thrown, but an error message was improved. To clarify, if a warning is given, while an error should be thrown, we do say there is a bug.

RQ2: How much time or commits pass between the introduction and detection of bugs? Are tests written before the detection?
The median of the time between the bug introduction and detection is 9154 hours, 381 days, or 332 Commit Cycles. The number of Commit Cycles per commit is also visible in Figure 2, where the dashed line is the median value. In 39 of the 100 bugs, a test (either added or modified) was found in the BFC’s PR. In 14 of the 39 cases, the test was already present, and only modified. This means that in 14 of the 100 bugs, a test was written before the detection. In one of these 14 bugs, it was a SStuB.

RQ3: Are bugs typically detected by the same developer who introduced them?
Of the 100 bugs, 24 were introduced and detected by the same developer. In the other 76 cases, the bug was detected by a different developer. When investigating the number of Commit Cycles of bugs introduced and detected by the same person, we see that the median is 100, which is 30% of the global median. This can also be seen in Figure 2.

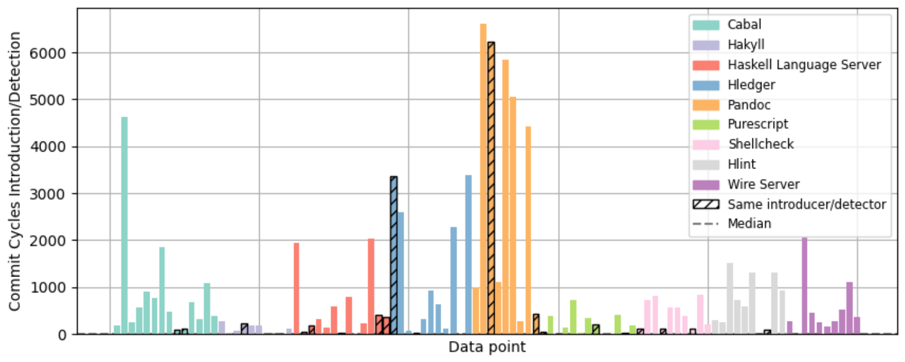

Figure 2: Commit Cycles between bug introduction and detection, same introducer/detector.

In Figure 3, the bugs are sorted on their detection time in Commit Cycles, where the horizontal line is the global median value. The vertical line indicates the point of the median value, meaning that on both sides of this line is half of the data. The bugs with the same introducer and detector are highlighted in blue. The figure shows that 19 of 24 bugs with the same introducer and detector are part of the fastest 50% of all bug detections.

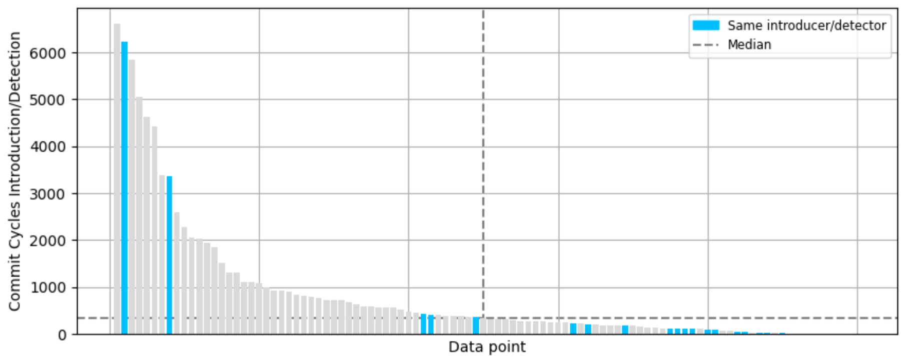

Figure 3: Commit Cycles between bug introduction and detection, same introducer/detector, sorted.

RQ4: How much time or commits pass between the detection and fixing of bugs? Are tests written after the detection?

The median of the time between the bug detection and fixing is 66 hours, 3 days, or 2 Commit Cycles. The Commit Cycles per bug are also visible in Figure 4. In the graph, the dashed line is again the median of all commits, but barely visible because of the low number.

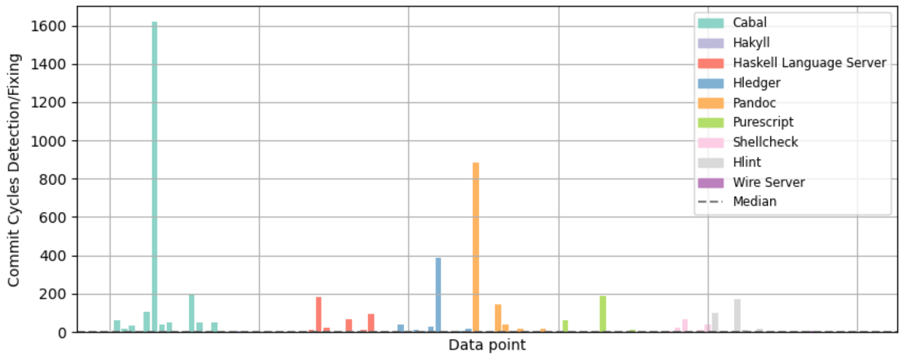

Figure 4: Commit Cycles between bug detection and fixing.

In 39 of the 100 bugs, a test (either added or modified) was found in the BFC's PR. In 25 of the 39 cases - of which 2 SStuBs - a new test was added simultaneously with the BFC, and thus after the bug detection. In 61 of the 100 cases, no test was found. The median time to fix a bug (the time difference between the bug introduction and fixing) was 10028 hours, 418 days, or 370 Commit Cycles.

RQ5: Are the so-called Simple Stupid Bugs detected earlier than other bugs? Of the 100 bugs, 14 were categorised as a Simple Stupid Bug. For these 14, the median time between the introduction and detection is 6354 hours, 265 days, or 465 Commit Cycles. This can also be seen in Figure 5.

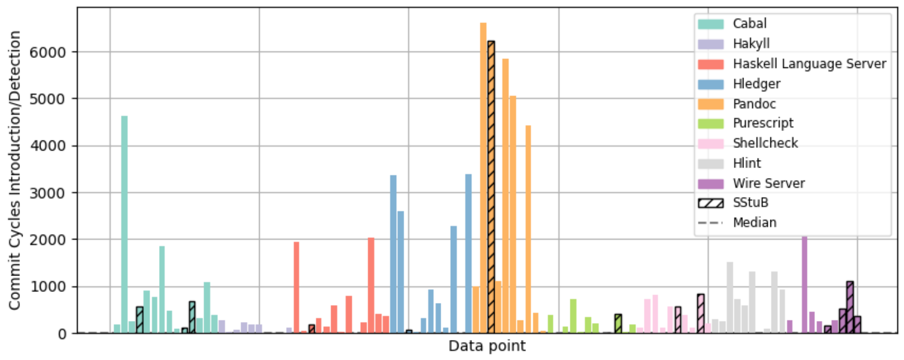

Figure 5: Commit Cycles between bug introduction and detection, SStuBs.

For 4 out of the 14 SStuBs, the person who fixed the bug was the same as the person who introduced it. For these, the median number of Commit Cycles between the bug introduction and fixing is 130. In the other 10 cases, when a different person fixed the SStuB, the median number of Commit Cycles is 550. This means that SStuBs are fixed 4.2 times faster when they are introduced and fixed by the same person, which can also be seen in Figure 6 where the SStuBs are highlighted.

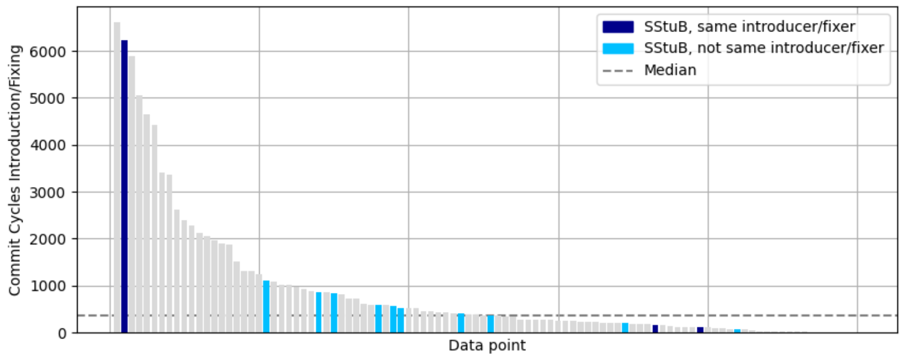

Figure 6: Commit Cycles between bug introduction and fixing, SStuBs, sorted.

Results per Repository

In Table 1, one can find the characteristics per repository (the median values per repository are given). In Figure 7, the following values are displayed per repository: the number of SStuBs, PRs without a linked issue, PRs with a test being either modified or added, bugs detected by the introducer, and bugs fixed by the introducer. These values are taken relative to the number of bugs found in that repository, indicated by the percentages.

	Cabal	Hakyll	HLS	Hledger	Pandoc
Bugs found	14	10	13	11	10
Timeframe	05/28/2020 - 05/15/2024	06/30/2020 - 06/01/2024	06/03/2020 - 06/02/2024	06/01/2020 - 06/01/2024	06/08/2020 - 06/01/2024
Total commits in time-frame	2600	172	2774	4213	4146
Commit Rate	0.0012	0.0001	0.0013	0.0020	0.0020
Commit Cycles between introduction and detection	519.16	87.18	325.10	638.32	2764.05
Commit Cycles between detection and fixing	44.10	0.42	1.63	8.10	10.82
Hours between introduction and detection	6952.81	17433.23	4118.56	5349.38	23233.17
Hours between detection and fixing	590.54	84.03	20.60	67.88	90.96

	Purescript	Shellcheck	HLint	Wire-server
Bugs found	12	10	10	10
Timeframe	06/04/2020 - 04/16/2024	06/24/2020 - 05/04/2024	06/05/2020 - 02/19/2024	06/05/2020 - 06/05/2024
Total commits in time-frame	355	412	933	3596
Commit Rate	0.0002	0.0002	0.0005	0.0017
Commit Cycles between introduction and detection	162.18	476.56	652.62	314.07
Commit Cycles between detection and fixing	1.04	3.73	1.12	0.30
Hours between introduction and detection	15338.45	39148.24	23630.05	3040.77
Hours between detection and fixing	97.95	306.70	40.44	2.87

Table 1: Characteristics per Repository.

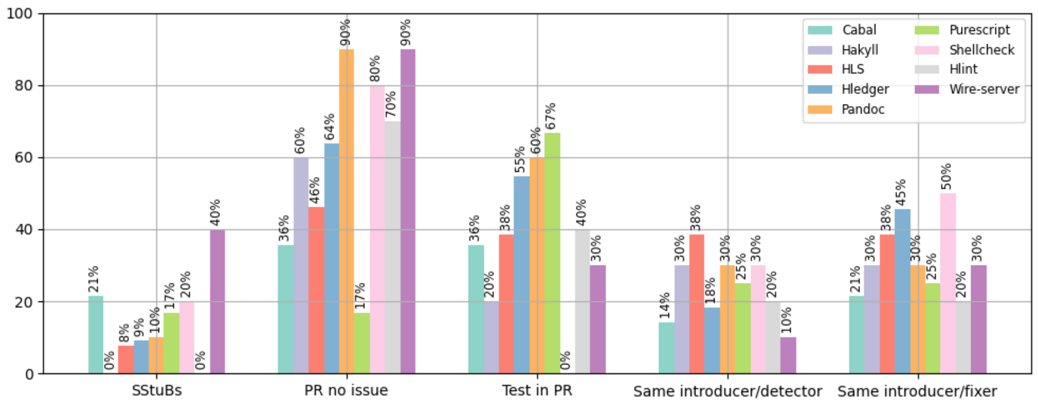

Figure 7: Values per Repository.

In this figure, one can see that, except for a small peak in Wire-server, the SStuBs are spread almost equally. Purescript seems to have a small number of PRs where no issue was linked, and a high number of PRs where a test was modified or added. Bugs that are detected and introduced by the same developer, seem to be spread equally over the repositories. This also seems to be the case for bugs that are fixed and introduced by the same developer.

Discussion

The outcomes have provided insights about Haskell bugs and their characteristics. Looking at the above plots, there is a large variety of bug-detection and bug-fix times. The large 90% confidence interval of (616.9, 1071.0) for the Commit Cycles between the bug introduction and detection, confirms this variety. The 90% confidence interval for the Commit Cycles between the bug detection and fixing is: (18.3, 81.0). These large confidence intervals are probably a result of the relatively small dataset that is used. If the dataset was expanded, the expected bug-detection or -fix time would be more accurate, and better comparable to other studies.

Despite the large variety of bug-detection and -fix times, we can still look for patterns in the data. From the 100 BFCs, there were only 14 where a test was already present, meaning tests are not typically written before the bug detection. From these 14 BFCs, 1 was to fix a SStuB. There were 61 BFCs where a test was neither added nor modified in the corresponding PR, of which 11 were to fix a SStuB. This shows that SStuBs occur 2.5 times more often when no test is written. This is presumably because developers who take time to write tests, also take more time to check their code and consequently prevent SStuBs or detect them in an early stage. Either way, this argues for developers to write test code to prevent SStuBs from being committed. We especially recommended this since more than half of the BFCs (61 of 100) did not contain any test at all.

Of the 100 bugs, 24 were introduced and detected by the same developer, meaning that bugs are not typically detected by the same developer who introduced them. When conducting a two-sample t-test on the bugs with the same introducer and detector, and the bugs with a different introducer and detector, the result is a p-value of 0.17. This does not indicate there is a significant difference between the two groups and is probably caused by the small dataset and high variance of all data. The high variance can 'mask' the differences between the groups, which makes it harder to detect a significant difference. When analysing this works' data, the median of Commit Cycles for bugs with the same introducer and detector is 30% of the global median. Moreover, we found that 19 of 24 of the bugs introduced and detected by the same person are in the smallest 50% of Commit Cycles (see Figure 3), showing that most have a relatively low detection time. So despite the result of the t-test, this shows that the detection time for bugs is lower when they are detected by the person who introduced them and is familiar with the code. In the research of Eiroa-Lledo et al., it appeared that SStuBs in Java programs are fixed 1.71 times faster when they are fixed and introduced by the same person [7]. These results argue for developers to check (and correct) their own code more extensively. Having more developers familiar with the code should also reduce bug-detection time, which motivates for working with shared code ownership.

We classified 14 of the 100 bugs as a SStuB, which is considerably low when comparing it to the research of Karampatsis and Sutton, in which was stated that 0.75 SStuBs were mined per commit in open-source Java projects [14]. The reason there are these few SStuBs, could be that Haskell is a pure and statically typed language, which means that all types are checked at compile time and that functions do not have side effects. Hence, developers do not make mistakes related to these characteristics (in contrast to other languages, like Java) and create less SStuBs. Analysing this works' SStuBs and looking at Figure 6, there does not seem to be a relation with the bug-fix time, as SStuBs seem to be equally spread over all commits.

We also showed different values and characteristics per repository. The number of Commit Cycles between detection and fixing seems to be considerably low for the Hakyll repository. This is probably caused by Hakyll's low Commit Rate, which indicates that their developers were less active. Developers being less active could also be the reason that Hakyll does not contain any SStuBs since SStuBs are more likely to occur while writing new code and adding new features. This was also shown in the research from Mosolygó et al., who found out that SStuBs seem to appear more in larger chunks of code written by the same developer [18].

From the values per repository, we also found that the Purescript repository contained most PRs where a test was included (either added or modified), and the least PRs where no issue was linked. This again shows the benefits of developers working in a structured manner, following standards and writing test code, as the Purescript repository has the second lowest number of Commit Cycles between the introduction and detection (see Figure 2 or 5).

There are still some limitations to this research. For example, from the Bug-Fixing Commits, Bug-Introducing Commits are linked manually without being checked by other researcher(s). This means that some BICs could be mislabelled. In addition, different repositories use different standards in GitHub. In some cases, the collaborators do not follow standards by not using the Git keywords for linking an issue (see Methodology) or not linking an issue at all. For this reason, it could be that some values are missing or not assigned correctly as they are reported differently per repository. In the cases where no issue was linked to the PR, the bug's detection date was set to the PR's creation date, making the bug-detection time slightly higher (since usually the issue creation is before the PR creation). This is especially the case for the Pandoc and Wire-server repository, as 90% of those PRs do not have a linked issue. The median time to fix a bug (the time difference between the bug introduction and fixing) turned out to be 418 days. This is more than the bug-fix time in the buggy files from ArgoUML and PostgreSQL, which ranges from 100 to 200 days [15].

When analysing the Simple Stupid Bugs in this research, and determining whether or not they are introduced and fixed by the same developer, there was a meagre number of 14 bugs. This means that the significance is considerably low, especially when comparing it to Eiroa-Lledo et al.'s research with 63,923 samples [7].

5 Responsible Research

In this section, the ethical aspects of the research will be discussed. Using the FAIR principles (findability, accessibility, interoperability, and reusability) [28], we explain why the research is ethically responsible.

Findability Findability means that metadata and data should be easy to find for both humans and computers. The repositories are open-source and available on GitHub, so easy to find. The repositories are also referenced in this paper.

Accessibility Accessibility means that once the user finds the required data, they need to know how they can be accessed. In this[3] repository, both the used data and code can be accessed, without need for authorisation or authentication. A description of the files is provided in the repository.

Interoperability Interoperability means that the data usually needs to be integrated with other data. In addition, the data needs to interoperate with applications or workflows for analysis, storage, and processing. The data is outputted as a JSON file, which formats the data in a human-readable way.

Reusability Reusability means that the data is reusable. To achieve this, metadata and data should be well-described so that they can be replicated and/or combined in different settings. Since the repositories are open-source and all commits, PRs, and issues contain an identifying hash, the exact data can be retrieved again and reused.

[3]`https://data.4tu.nl/private_datasets/XdMs5VuhinYLy7eZGtDB0VnQisdYT-0dYYH2lXXdfyQ`

Furthermore, one might argue that reporting about the bugs present in repositories might discredit the repositories. However, since we use an approximately equal number of bugs per repository, one cannot say that one repository is 'worse' than another because it has more bugs.

6 Conclusion and Future Work

To analyse Haskell bugs, we have conducted a study of 100 Haskell bugs from 9 different repositories. The study's goal was to gain more knowledge about the development of a bug, and its stages. Using these insights, developers could get a better understanding of the development process of Haskell programs, and optimise it. Three stages have been distinguished: bug introduction, detection, and fixing. We found that the current definition of a bug is fairly broad and we provided a way how the definition could be applied to open-source Haskell projects.

Using the metric 'Commit Cycle', the time between two commits can be measured relative to the commit frequency in the repository. After analysing the 100 bugs, we have found that the median time between the bug introduction and detection is 381 days or 332 Commit Cycles. The median time between the bug detection and fixing is 3 days or 2 Commit Cycles. In general, tests for the detected bugs are not written before detection.

We have shown that bugs are not typically introduced and detected by the same developer. However, bugs are detected earlier when the developer (who is familiar with the code) introducing them finds them, compared to when someone else detects them. For developers, this means that they should check (and correct) their code more extensively, to lower the bug-detection time. Working with shared code ownership - and thus having more developers familiar with the code - should also reduce bug-detection time.

Of the 100 bugs, 14 were classified as a Simple Stupid Bug, which is considerably low compared to open-source Java projects. There does not seem to be a relation between SStuBs and the bug-fix time, as SStuBs seem to be equally spread over all commits. This does not invalidate the recommendation for developers to check their code more extensively, as reducing the 'regular' bug-fix time, is likely to also reduce the SStuB-fix time. For reducing the number of SStuBs being committed, we recommend developers to write (more) test code, as we have shown that SStuBs occur 2.5 times more often when no test is written.

From analysing values and characteristics per repository, we have shown the benefits of developers following standards and writing test code. For repository owners, this means that we recommend being more strict with developers working structured and precisely. This could be in terms of checking their code and writing test code, but also in terms of following Git's or repository's standards.

In the future, it might be interesting to carry out further research on at which point tests are being added. Due to time constraints and lack of familiarity with the code, it could not be analysed thoroughly whether any tests were added before or after the BFC's PR. To extend the research, it might be interesting to see if the results differ from non-open-source Haskell repositories and if they differ from other languages. Lastly, the quality of the research could be improved by collecting more bugs from more repositories. Expanding the dataset would result in a more accurate bug-detection and -fix time, and make the results better comparable to other studies.

A BIC's and BFC's JSON

```json
[
    {
        "repo": "cabal",
        "total considered commits": 2600,
        "timeframe": "05/28/2020 to 05/15/2024",
        "commits": [
            {
                "BFC hash": "10de4e5bf9d2eb5fe66383aa43de829f08bc801e",
                "BIC hash": "5cb84756e042ebf00b31bf95dc7577c2a7d2c456"
            },
            {
                "BFC hash": "5b44c05da1b9a7f7a545153e0beedfa0fc085e57",
                "BIC hash": "e496b09eb3e9d15ca3380155a85c9a5c7a3587d5"
            }
        ]
    },
    {
        "repo": "hakyll",
        "total considered commits": 172,
        "timeframe": "06/30/2020 to 06/01/2024",
        "commits": [
            {
                "BFC hash": "b4c224c65218a08b9bc7207289ca90251d201b16",
                "BIC hash": "1acebf2699ecac86a2c82445eaeb11eec176be79"
            },
            {
                "BFC hash": "3b48802dd30c4d7394683f6428b73132f796e35e",
                "BIC hash": "2ea5d682cfb7631bc00c9452e5fdc1c1c15900f8"
            }
        ]
    },
    {
        "repo": "haskell-language-server",
        "total considered commits": 2774,
        "timeframe": "06/03/2020 to 06/02/2024",
        "commits": [
            {
                "BFC hash": "10b5f3bd51862175d2b63803f5dffd4bd862cae8",
                "BIC hash": "f135edb1317574a1cbfc1fd629f5f01fd911d1e7"
            },
            {
                "BFC hash": "73fdd91e5a5a6a7cabdafb453f23749693029cc5",
                "BIC hash": "5271f6312cf1b28c1d8ceee14fc12f4bae1d6c31"
            }
        ]
    },
    {
        "repo": "hledger",
        "total considered commits": 4213,
        "timeframe": "06/01/2020 to 06/01/2024",
        "commits": [
            {
                "BFC hash": "08a5f1ee78eac27beaa95ec432dfde0d51e146d4",
                "BIC hash": "07a9f119628c99284cf580b5d4da72f8ac1b8660"
            },
            {
                "BFC hash": "e035730afb6f488fceca70b0d343df5995ce4272",
                "BIC hash": "dc16451de0e7a519356653ce88b23a0176d0f202"
            }
        ]
    }
]
```

B Processed data JSON

```
[
    {
        "repo": "hlint",
        "total considered commits": 933,
        "timeframe": "06/05/2020 to 02/19/2024",
        "commit rate": 0.0004603063328254291,
        "avg commit cycles introduction/detection for repo": 698.2213495653486,
        "avg commit cycles detection/fixing for repo": 30.28655329952057,
        "avg hours introduction/detection for repo": 25281.039305555554,
        "avg hours detection/fixing for repo": 1096.6086111111113,
        "commits": [
            {
                "BIC_hash": "58b82f7dc2256e38e67b00b75abf21b2e1ba9c44",
                "BFC_hash": "e760b31702a660d6938bddbc573105d42a0280ed",
                "prnum": 1498,
                "bug introduction": "2022-02-13T17:17:57Z",
                "bug_detection": "2023-04-23T10:03:59Z",
                "bug_fixing": "2023-09-19T01:39:01Z",
                "bug introducer": "lierdakil",
                "bug detector": "yitz-zoomin",
                "bug fixer": "ulidtko",
                "same introducer/detector": false,
                "same detector/fixer": false,
                "same introducer/fixer": false,
                "is SStuB": false,
                "time introduction/detection": 10408.767222222223,
                "time detection/fixing": 3567.583888888889,
                "commit cycles introduction/detection": 287.4732881576784,
                "commit cycles detection/fixing": 98.53088741649165,
                "test": "added"
            }
        ]
    },
    {
        "repo": "wire-server",
        "total considered commits": 3596,
        "timeframe": "06/05/2020 to 06/05/2024",
        "commit rate": 0.0017214535105622925,
        "avg commit cycles introduction/detection for repo": 543.7261208244402,
        "avg commit cycles detection/fixing for repo": 0.7562833017061477,
        "avg hours introduction/detection for repo": 5264.215361111111,
        "avg hours detection/fixing for repo": 7.3221388888888885,
        "commits": [
            {
                "BIC_hash": "b06813bdaf4110214cf37891c6bd72ef3165c819",
                "BFC_hash": "bc07185d4b75637e52048ecde7f59c43f88b0705",
                "prnum": 1456,
                "bug introduction": "2021-01-05T09:41:02Z",
                "bug_detection": "2021-04-19T09:21:53Z",
                "bug_fixing": "2021-04-19T12:38:25Z",
                "bug introducer": "akshaymankar",
                "bug detector": "pcapriotti",
                "bug fixer": "pcapriotti",
                "same introducer/detector": false,
                "same detector/fixer": true,
                "same introducer/fixer": false,
                "is SStuB": false,
                "time introduction/detection": 2495.6808333333333,
                "time detection/fixing": 3.2755555555555556,
                "commit cycles introduction/detection": 257.7719119070817,
                "commit cycles detection/fixing": 0.33832299660917586,
                "test": "modified"
            }
        ]
    }
]
```

References

[1] About issues - GitHub Docs.

[2] About pull requests - GitHub Docs.

[3] Linking a pull request to an issue - GitHub Docs.

[4] GitHub: Letâs build from here, 2024.

[5] Mohamed F. Ahmed and Swapna S. Gokhale. Linux bugs: Life cycle, resolution and architectural analysis. *Information and Software Technology*, 51(11):1618–1627, 2009. Third IEEE International Workshop on Automation of Software Test (AST 2008) Eighth International Conference on Quality Software (QSIC 2008).

[6] Herbert D. Benington. Production of Large Computer Programs. *IEEE annals of the history of computing*, 5(4):350–361, 10 1983.

[7] Elia Eiroa-Lledo, Rao Hamza Ali, Gabriela Pinto, J.P. Anderson, and Erik Linstead. Large-Scale Identification and Analysis of Factors Impacting Simple Bug Resolution Times in Open Source Software Repositories. *Applied sciences*, 13(5):3150, 2 2023.

[8] Jesús M. González-Barahona, Daniel Izquierdo-Cortázar, and Andrea Capiluppi. Are developers fixing their own bugs? *International journal of open source software processes*, 3(2):23–42, 4 2011.

[9] haskell. Cabal, 2024.

[10] haskell. Haskell language server, 2024.

[11] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*, pages 392–401. IEEE, 2013.

[12] jaspervdj. Hakyll, 2024.

[13] jgm. Pandoc, 2024.

[14] 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*, MSR â20. ACM, June 2020.

[15] Sunghun Kim and E. Jr. How long did it take to fix bugs? pages 173–174, 05 2006.

[16] koalaman. Shellcheck, 2024.

[17] Martin Monperrus. Automatic software repair. *ACM computing surveys*, 51(1):1–24, 1 2018.

[18] Balázs Mosolygó, Norbert Vándor, Gábor Antal, and Péter Hegedűs. On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 495–499, 2021.

[19] ndmitchell. hlint, 2024.

[20] purescript. Purescript, 2024.

[21] Václav Rajlich and Keith Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 7 2000.

[22] Baishakhi Ray, Vincent J. Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Prémkumar Dévanbu. On the "Naturalness" of Buggy Code. *arXiv (Cornell University)*, 1 2015.

[23] Gema Rodríguez-Pérez, Gregório Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesús M. González-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical software engineering*, 25(2):1294–1340, 2 2020.

[24] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregório Robles, and Jesús M. González-Barahona. What if a bug has a different origin? *ESEMâ18, October 2018, Oulu, Finland*, 10 2018.

[25] W. W. Royce. Managing the development of large software systems. *Proceedings, IEEE Wescon*, pages 1–9, 1 1970.

[26] simonmichael. Hledger, 2024.

[27] M Thomas and H Thimbleby. Computer bugs in hospitals: An unnoticed killer, 2018.

[28] Wikipedia contributors. Fair data — Wikipedia, the free encyclopedia, 2024. [Online; accessed 3-June-2024].

[29] wire. wire-server, 2024.

[30] Jacek Åliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *Software engineering notes*, 30(4):1–5, 5 2005.