HasBugs - Handpicked Haskell Bugs

Applis, Leonhard; Panichella, Annibale

DOI
10.1109/MSR59073.2023.00040

Publication date
2023

Document Version
Final published version

Published in
Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)

2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR) | 979-8-3503-1184-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/MSR59073.2023.00040

HasBugs - Handpicked Haskell Bugs

1st Leonhard Applis
Delft University of Technology
Software Engineering Research Group (SERG)
Delft, Netherlands
L.H.Applis@Tudelft.nl

2nd Annibale Panichella
Delft University of Technology
Software Engineering Research Group (SERG)
Delft, Netherlands
A.Panichella@Tudelft.nl

***Abstract*—We present `HasBugs`, an extensible and manually-curated dataset of real-world 25 Haskell Bugs from 6 open source repositories. We provide a faulty, tested, and fixed version of each bug in our dataset with reproduction packages, description, and bug context. For technical users, the dataset is meant to either help researchers adapt techniques from other programming languages to Haskell or to provide a human-verified gold standard for tools evaluation and enable future reproducibility. We also see applicability for qualitative research, e.g., by analysis of bug lifecycles and comparison to other languages. We provide a companion website for easy access and overview under https://ciselab.github.io/HasBugs/.**

I. Introduction

Bugs are usually seen as obstacles - nuances and failures resulting from mistakes. For researchers in software engineering (SE), however, bugs are opportunities. They form the foundation for techniques such as fault localization [1] [2], test generation/fuzzing [3] [4], and program repair [5] [6]. Observations in these fields show the age-old adage *garbage in - garbage out* applies to these domains as well: It took the community a while to realize that not all patches produced by `GenProg` [7] actually fix the program [8] - despite passing the test suite. While we can blame individuals for this, such mistakes happen and the more constructive approach is to mitigate these issues with better input. The information missing in `Defects4J` was a summary of the bug, so generated patches could not (easily) be checked by the respective researchers. The assumption *'passing CI = fixed program'* turned out to be insufficient.

In this light, we present `HasBugs` - an extensible, high-quality dataset of Haskell Programs with bugs, tests, and fixes. We emphasize three key aspects in particular: (1) It provides a rich context of the bug and fixes, (2) it includes different artifacts for Software Engineering research tools and techniques, and (3) it allows easy reproduction.

We link the repository, issues, and pull requests (PRs) alongside a bug summary to capture the bugs context. This enables future researchers to verify results for their applicability within the domain - e.g., whether a generated test actually asserts against the given bug it was meant to find, or actually finds a new one. Similarly, we hope that discussions in PRs and issues help to understand implementation details; Why were things changed the way they were? Was it a *hard* or an *easy* bug?

Different research tools and techniques need different inputs. We address this by covering common artifact types for various techniques: Within the datapoint, we provide a fault location and location of the fix to clearly specify the points of interest within the patches. These locations can span multiple methods, as from our observation, it became obvious that fixes often need to be applied in connected methods. The failing test is provided in a separate patch, which helps comparing test generation tools and creates a *failing-but-tested*-version. Running this tested version produces output necessary for, e.g., fault localization and program repair. This contrasts with many other datasets - our faulty versions usually have a passing build.

Lastly, we provide reproduction by capturing the current builds in Docker-images available for download. The used Dockerfiles capture the required environments as well as commands and empower users to easily alter the code of the respective versions.

The contributions can be summarized as follows:

1) 25 datapoints from 6 FOSS Haskell programs and libraries;
2) Rich context information linking source code and GitHub;
3) Multi-point fault-locations, test-patches and 3-stage-versions (faulty, tested & fixed);
4) Dockerfiles and pre-produced available images running builds

Similar Datasets are Defects4J [9] for Java programs, and in particular, the scripts and support created

2574-3864/23/$31.00 ©2023 IEEE
DOI 10.1109/MSR59073.2023.00040

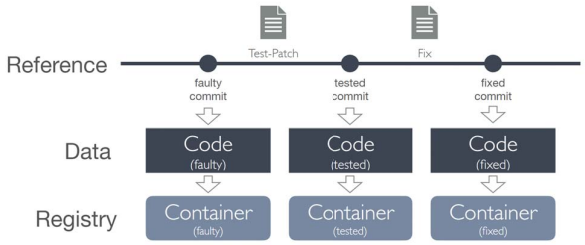

Fig. 1. Overview of one data point in HasBugs

around it later[1] [10]. We provide the same granularity/content of bugs with the difference that our bugs exist without a failing test case first. In case of a passing CI, a failing test case is also provided separately, constituting a third version. We hope to increase the reproducibility by providing Dockerfiles for most datapoints, saving researchers from the time-intensive work of configuring necessary dependencies. While there are multi-fault locations in the newer supplements of Defects4J [11], we provide the multi-fault locations within the dataset.

Another close dataset is Bugswarm [12], which is mined from FOSS Java Projects using Travis CI. Bugs in Bugswarms are, hence, not human-evaluated; with over 3000 data points, a manual inspection is unlikely at this point. We tried to adapt ease-of-use from Bugswarm and, in particular, their very accessible website. We purposefully did not try to automate bug-mining to keep high quality and double-check every entry.

The last related dataset is Simple Stupid Bugs [13], which is automatically mined from FOSS Java Projects with single-line fixes. By far the biggest dataset, it also has the least context information, and its quality assessment is based on sampling. Furthermore they rely on SZZ [14], itself debated [15] [16], for fix-localization.

In general, HasBugs is meant to be a starting point for re-implementing and progressing on software engineering algorithms in the Haskell Domain, or to be a *gold standard* in evaluation. We are aware that the size of the dataset is not suitable to train deep learning models, but such a model needs to be evaluated against high-quality human-checked data before production use. We see great potential for SE tools in functional programming, with outstanding examples like the Haskell Language Server (HLS), and want to aid the development of a broader range of tools by providing HasBugs as a resource to the community.

II. Dataset Description

Currently, HasBugs contains 25 bugs from 6 repositories as shown in Table I. Every bug consists of a primary `json` file that holds the unique information of the bug: the Git repository, relevant commits, PRs, descriptions, etc. A subset of the information is shown in Figure 2, which has been shortened for the sake of readability. With each datapoint, we provide a Dockerfile for a reproducible build, alongside the bug-asserting test isolated into a git patch. The `HasBugs-Dockerfile` exists alongside potential project-inherent Dockerfiles and performs the compilation during `build`-stages and has the `test-command` as the entrypoint. This addresses Haskell's long compilation times - pulling the image from a container registry[2] starts from compiled source-code immediately with running tests.

These `reference`-files are a lightweight set of information that can either be used directly or be manifested into heavier artifacts (using shell scripts). The download of repositories is automated from a data point, providing an artifact

Fig. 2. Example HasBugs Datapoint.json (edited for readability)

```
"id": "cabal—1",
"repositoryurl": "git@github.com:haskell/cabal",
"license": "BSD—3",
"faultcommit":  "01844...",
"fixcommit":    "55e03...",
...
"description": "Cabal starts multiple processes to build a project.
                'cabal run' termination does not terminate all child
                processes automatically as well. The solution is to use
                'withCreateProcess' rather than 'createProcess' and throw
                an asynchronous exception from the main thread when a
                termination is wanted.",
"categories": ["system—test","os","multi—threading","multi—processing"],


"relatedissues": ["https://github.com/haskell/cabal/issues/7914"],
"relatedprs": ["https://github.com/haskell/cabal/pull/7921",
               "https://github.com/haskell/cabal/pull/7757"],

"faultlocations" : [ {
    "startline": 127,
    "endline": 127,
    "file": "cabal/src/distribution/simple/program/run.hs",
    "module": "distribution.simple.program.run",
    "function": "runprograminvocation"
    },  ...  ],
"fixlocations" : [ {
        "startline": 127,
        "endline": 127,
        "file": "cabal/src/distribution/simple/program/run.hs",
        "module": "distribution.simple.program.run",
        "function": "runprograminvocation"},
    {
        "startline": 175,
        "endline": 175,
        "file": "cabal—install/main/Main.hs",
        "module": "Main",
        "function": "main"
    },
    ...
```

Repository	Bugs	Stars	.hs-Files	Domain
Cabal	6	1.5k	1.3k	Build System
Pandoc	6	27.7k	291	Document Conversion
ShellCheck	5	31.2k	24	Linter
HLS	4	2.3k	1.3k	IDE Language Server
Purescript	1	8.0k	220	Transpiler
HLedger	3	2.3k	156	Accounting

TABLE I
Summary of HasBugs per Repository

[1] https://github.com/rjust/defects4j

[2] https://github.com/orgs/ciselab/packages?repo_name=HasBugs

of the repository usable for static analysis. The archived projects in their three-fold states can be accessed under https://doi.org/10.5281/zenodo.7569135. On top of that, the repositories can be built inside docker containers either by using the provided `HasBugs-Dockerfiles` or by pulling pre-compiled images from the repository. Both activities are supported by shell scripts accompanying the data repository.

To ease access and provide a barrier-free entry, we developed a companion website: https://ciselab.github.io/HasBugs/.

The website contains a summary of our motivation and a lightweight entry to the features presented in this paper. Outside of advertisement, the website allows browsing the data points and their respective features directly without pulling the repositories and setting up your local machine. This covers the descriptions, links to GitHub, and categorical information, e.g., the license. Lastly, the website contains a more elaborate tutorial on how to approach different artifacts with concrete shell commands to run. We aim for the website to quickly enable researchers to assess whether the dataset fits their objectives and to ease adoption. For qualitative research, the website itself mitigates barriers for a less tech-savvy audience.

III. Data Collection and Challenges

The data collection was primarily a manual process. We started by gathering a list of high-star repositories from GitHub and Hackage[3] and filtered it for suitable FOSS licences, which resulted in a list of 44 libraries and programs.

These projects have been assessed in various categories by the authors, such as `quality of issues`, `linkage pr to issue`, `linkage commit to issue`, etc. A total of 7 categories were considered, forming a (subjective) score of the ease of access for each repository. This overview is provided in the `resources` of the dataset repository.

From the most suitable projects, the authors decided on the initial data points by diversity: We aimed to cover *as many domains as possible*, to cover a variety of different bugs. This led, e.g., to the inclusion of Cabal (a build framework for Haskell), while Stack (another build framework) was left out. In general, many tasks covered by Haskell libraries revolve around parsing and compiling, but we consider the projects shown in Table I a good, diverse view of the mainstream applications of Haskell.

From the chosen repositories, we looked into issues and PRs labeled `bug`, extracting faulty and fixed commits. For simplicity's sake, we consider the 'last known faulty version', i.e., the commit before the fix was applied. We limit our search to bugs of at most two years of age and Haskell framework versions above 8.10 (released march 2020) to provide an accurate snapshot of today's Haskell Project rather than a historical view. The suggested `bug-summary` and `categories` were provided by one author and evaluated by one other author. We later normed the granularity of entries in joint meetings.

While we initially considered providing our own tests for bugs, this was unnecessary: Most bug fixes in the repositories provide a test within the fix-commit, that was provided by the author of the fix. We hence extracted the test from the fix-commit and verified that it failed once applied to the faulty commit.

The provided Docker images are built using the project's documents, e.g., their READMEs, configurations, or existing Dockerfiles. The docker builds formed a unique challenge for this project, as they come with big space and computation requirements. We hence decided to diverge slightly from the repositories *intended* GHC-version that was used at the time of commit, and tried to minimize the number of our base images to utilize better caching and save disk space. As a heuristic, we bumped versions up, assuming newer GHC versions generally remove more issues than they introduce.

The list of bugs provided for the initial version of this dataset is not exhaustive for their respective repositories: Projects such as Pandoc still have *bugs left*, but we decided to focus on other repositories for a wider variety of domains.

IV. Research Opportunities

From our initial perspective of the dataset, we see two directions: (1) Technical solutions and their validation and (2) qualitative analysis of the Haskell FOSS ecosystem.

Technical contributions consists of tools for **fault localization**, **test generation**, **test amplification & carving** or **automated program repair** (APR). As our dataset has (often) multiple fault and fix locations, the field of multi-location fault-localization [11] could be investigated, as well as automatic analysis of which part of a patch contributes to the fix. Test generation [17], [18], amplification [19], carving [20], and regression testing [21] are particularly supported as we provide the fixed version and a sample test, which cover the requirements for most common techniques in the field. Comparing the existing test can help to assess readability, coverage, and functionality. Further relevant research approaches in the field on mining software repositories (MSR) that can highly-benefits from

[3]The central Haskell package archive https://hackage.haskell.org/

our dataset include bug prediction, crash replication, fault localization, bug severity classification, and bug-introducing commit identification. Inspired by Sobreira et al. [10], program repair benefits from the specified *bug-reason* which supports humans in assessing automatically produced patches. The two significant challenges for APR - a stable running build as well as failing tests - are addressed by our dataset and the containers.

For **qualitative analysis**, we see good opportunities in the communication patterns of the ecosystem. The Haskell community is often perceived as slightly alien or elitist, with one of the prominent mantras being: "you don't write bugs in Haskell - once it compiles, it works". Judgments aside, our faulty versions compile and pass tests, and the presented self-admitted bugs can originate from manifold sources, for example actual implementation errors, missed requirements, or environmental factors (e.g., OS changes). We consider it fruitful to investigate the nature of the bugs and their fixes, and compare it to studies on Java [22].

Another point of interest is the type of test: Most supplied tests were system- or integration-level tests, although they could be translated into unit tests. Why did contributors choose high-level over low-level tests? We can imagine many factors, e.g. *"being closer to the bug report"*, easier adaption of end-to-end-tests by copying existing ones, or implementation challenges for unit tests. Despite Haskell being fully functional, many functions rely on context-heavy constructs such as `monadstacks` or self-implemented data types.

The above-mentioned qualitative analysis could also help to address one of the limitations of this work: It is not imminent if we are looking at a dataset of *survivors*. While we see mostly integration- or system-level issues and their corresponding tests, we are lacking similar findings on the unit level. This could be an under-reporting of unit-level issues; maybe unit-level problems are *caught* at higher levels. On the other hand, it could be that Haskell unit-level development is outstanding and produces little errors. Findings from this could help in education by catalyzing them into *best-practices* for later stages of development.

V. Limitations and Future Work

The primary limitation for the SE community is that the size of the programs is likely insufficient for model-fitting. It might be possible to fit approaches like decision trees, however, neural networks can only be used with other techniques, e.g., transfer learning. We aim to assist these tasks by providing a dataset big enough for validation and benchmarking, as well as providing *actionable* results due to rich contextual information. Similarly, the high compilation times of Haskell programs might impact the development of tools utilizing dynamic analysis, but analog to the above, we hope to aid their evaluation.

An internal threat to validity is the data collection - we only took into account self-admitted bugs that have been made visible through either PRs or issues. This can lead to a set of biases, e.g., our over-representation of system-level tests, as the user-reported bugs are expressed as issues. Unit-tests and their bugs might be solved by developers before publishing and are hence invisible to us.

We aim to address this after publication through a community effort — we want to reach Haskell developers to verify our assumptions and gain more data points. This discussion can also shape the tools that the community needs.

VI. Conclusion

Drawing from existing datasets, `HasBugs` provides a rich data points suitable for most SE applications. We enable qualitative research by linking to social artifacts in issues and PRs, static analysis by providing code, diffs, and locations, as well as supporting dynamic approaches with containerization. Due to the limited size, we aim for `HasBugs` to become a benchmark for tool and model evaluation, as well as to provide a starting point for the next generation of Haskell SE tools.

Link-Tree

The repository is found on GitHub under https://github.com/ciselab/HasBugs and its archived form at https://doi.org/10.5281/zenodo.7569327.

The manifested datapoints are archived under https://doi.org/10.5281/zenodo.7569135 and the docker images are in the GitHub container registry.

Acknowledgment

We would like to thank our student Chris Lemaire, who contributed greatly to the collection of data and the surrounding tooling.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[2] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Information and Software Technology*, vol. 124, p. 106312, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300641

[3] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[5] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 441–444. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[6] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 302–313.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[8] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[9] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[10] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 130–140.

[11] G. An, J. Yoon, and S. Yoo, "Searching for multi-fault programs in defects4j," in *Search-Based Software Engineering*, U.-M. O'Reilly and X. Devroey, Eds. Cham: Springer International Publishing, 2021, pp. 153–158.

[12] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.

[13] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 573–577. [Online]. Available: https://doi.org/10.1145/3379597.3387491

[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[15] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.

[16] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems with szz and features: An empirical study of the state of practice of defect prediction data collection," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–49, 2022.

[17] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[18] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[19] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[20] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 253–264.

[21] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[22] B. Mosolygó, N. Vándor, G. Antal, and P. 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)*, 2021, pp. 495–499.