Bridging the Gap between Different Programming Paradigms in Coverage-based Fault Localization

Feng Li
MoE Key Laboratory of HCST,
School of Computer Science,
Peking University
Beijing, China
lifeng2014@pku.edu.cn

Meng Wang
Department of Computer Science,
University of Bristol
Bristol, UK
meng.wang@bristol.ac.uk

Dan Hao*
MoE Key Laboratory of HCST,
School of Computer Science,
Peking University
Beijing, China
haodan@pku.edu.cn

ABSTRACT

Fault localization is to identify faulty program elements. Among the large number of fault localization approaches in the literature, coverage-based fault localization, especially spectrum-based fault localization has been intensively studied due to its effectiveness and lightweightness. Despite the rich literature, almost all existing fault localization approaches and studies are conducted on imperative programming languages such as *Java* and *C*, leaving a gap in other programming paradigms. In this paper, we aim to study fault localization approaches for the functional programming paradigm, using *Haskell* language as a representation. We build up the first dataset on real *Haskell* projects including both real and seeded faults, which enables the research of fault localization for functional languages. With this dataset, we explore fault localization techniques for *Haskell*. In particular, as typically for SBFL approaches, we study methods for coverage collection as well as formulae for suspiciousness scores computation, and carefully adapt these two components to *Haskell* considering the language features and characteristics, resulting in a series of adaption approaches and a learning-based approach, which are evaluated on the dataset to demonstrate the promises of the direction.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

debugging, fault localization, programming paradigms, Haskell

ACM Reference Format:
Feng Li, Meng Wang, and Dan Hao. 2022. Bridging the Gap between Different Programming Paradigms in Coverage-based Fault Localization. In *13th Asia-Pacific Symposium on Internetware (Internetware 2022), June 11–12, 2022, Hohhot, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3545258.3545272

*Corresponding author

HCST: High Confidence Software Technologies

Internetware 2022, June 11–12, 2022, Hohhot, China

ACM ISBN 978-1-4503-9780-3/22/06…$15.00
https://doi.org/10.1145/3545258.3545272

1 INTRODUCTION

Fault localization (FL) [8, 32, 43, 46] aims to automatically diagnose faulty program elements (e.g., classes and methods). More specifically, fault localization techniques often leverage various static and/or dynamic program analysis information to compute suspiciousness scores (i.e., likelihood of being faulty) for each program element, which can be then ranked as candidates for repairing [12, 40, 45]. To date, researchers have proposed to leverage various information to facilitate fault localization, among which, coverage-based fault localization has been intensively studied due to its effectiveness and lightweightness for real-world applications [2, 19, 42, 49].

Spectrum-based fault localization (SBFL) [1, 19, 42] identifies faulty program elements by statistically analyzing coverage of failed and passed tests. In particular, SBFL represents coverage by the numbers of failed and passed tests covering each program element, and regards the elements covered by more failed tests and less passed tests as more suspicious. Over the years, a lot of research has gone into this [2, 27, 31, 42].

Despite this rich literature, almost all existing fault localization techniques and studies are conducted on imperative programming languages such as *Java* and *C*. There are a variety of reasons behind this phenomenon, some entirely rational and others less so. It is clear that a few imperative languages dominate software development and thus techniques based on them will have the most impact. Moreover, there is a practical concern of evaluating the research, especially when the data needed is hard to collect. Follow-on work tends to stick to the same languages to reuse the datasets.

This lack of language coverage is increasingly a concern amid the current trend of multi-language and multi-paradigm programming. In additional to the mainstream imperative languages, companies like WhatsApp, Facebook, and Standard Chartered now use functional languages in large scale. Ideas originated from functional programming such as lambda expressions, higher-order functions, list structure and generics have made their way into the design of many modern programming languages including *Java*, *C*, and *C#*. If widely used, these language features will lead to programs substantially different in structure. Therefore, even for conventional imperative languages, we can no longer safely assume the effectiveness of techniques developed solely for one language paradigm.

In this paper, we aim to diverge from the well-travelled path and study fault localization techniques for the functional programming paradigm. Specifically, we use *Haskell* [20] as a representation of the paradigm for its popularity and uncompromising functional style that is free of imperative features such as side effects and

assignment. As mentioned above, the rise of multi-paradigm languages has blurred the traditional boundaries, and paradigm has become more of a programming style rather than language distinction. Our hope is that by choosing a very "pure" language like *Haskell*, we can better isolate and contrast the differences affecting fault localization.

Having this very distinctive aim does not mean that we will start from scratch, disregarding the wealth of knowledge in this field accumulated through decades of intensive research. It is only right if we stand on the shoulders of giants, and carefully examine existing results in the new setting to identify gaps in applicability, before trying to bridge them. In this paper, we do exactly this.

As a first step, we build up the first dataset containing two types of faults (real and seeded) in *Haskell*. For real faults, we inspect the development history of popular open-source projects to handpick relevant bug-fixing issues, and construct versions of the projects representing the faulty and fixed versions of the selected issues. For seeded faults, we use a mutation analysis library [23] to generate mutants. For each faults collected, we run the real tests that come with the project to collect coverage data of each test.

The significance of the dataset is that it for the first time enables one to explore fault localization techniques for functional languages. We start with adapting existing SBFL approaches to *Haskell* and test their effectiveness. Briefly speaking, an SBFL approach includes two parts: the method of coverage collection and the formula for computing suspiciousness score; we will need to adapt both to the functional context. Neither of the adaption is trivial though. Functional programs are typically structured very differently from imperative ones, which impacts the definition of coverage criteria and collection of data. The formula for computing suspiciousness score is more abstract and not directly affected by syntactic differences. Yet we do not expect the same level of effectiveness can be achieved by simply imposing the formulae and techniques conceived and verified in the imperative setting directly to the functional setting.

Of course, it is unrealistic to expect this work alone to supersede the decades of research in the imperative setting. But we do aim to clearly identify the gaps through a series of experiments and establish a new research direction justified by evidence of improvements. To sum up, this paper makes the following contributions: (1) The first work that identifies the fault localization problem in functional programming languages and the first attempt to bridge the gap between different programming paradigms in coverage-based fault localization. (2) A dataset in *Haskell* that contains both real faults and seeded faults, which is put on our website **https://github.com/Spiridempt/HaFLa** [16]. The dataset also contains the coverage information of each test. (Section 4) (3) A series of adaption approaches of SBFL to the functional setting and a study evaluating the effectiveness of them. (Section 5) (4) A learning-based approach for establishing the relationship between coverage information and fault location. (Section 6)

2 PRELIMINARY: COVERAGE-BASED FAULT LOCALIZATION

In the literature, researchers have proposed a large number of fault localization techniques, e.g., coverage-based techniques [2, 19, 42], mutation-based techniques [30, 33], and predicate switching based techniques [48]. Among these techniques, coverage-based fault localization has been intensively studied due to its effectiveness and lightweightness for real-world systems [49]. Therefore, in this paper, we investigate coverage-based fault localization across different programming paradigms.

Coverage-based fault localization is usually proposed and evaluated in imperative programming languages, whose basic program elements are statements. Intuitively, a statement covered by more failed tests and less passed tests is more likely to be faulty. Following this intuition, a large number of spectrum-based fault localization approaches (abbreviated as SBFL) have been proposed in the literature and they are the most prominent coverage-based fault localization approaches. In particular, an SBFL approach first abstracts test coverage information on each statement e into the number of failed tests covering e (i.e., e_f) or not covering e (i.e., n_f), and the number of passed tests covering e (i.e., e_p) or not covering e (i.e., n_p), and computes suspiciousness scores (probability of being faulty) of each statement based on the number of passed/failed tests that cover it. Then SBFL outputs a ranked list of statements based on the descendent order of their suspiciousness scores.

Following this idea, based on the same spectrum tuple values (i.e., e_f, n_f, e_p, n_p), these SBFL approaches differ in their ranking formulae. Among the formulae, Ochiai [2], DStar [42], and Tarantula [19] are of the most effective ones. Moreover, these SBFL approaches are usually presented and evaluated in imperative programming languages whose granularity includes statements, methods, and branches. Furthermore, as SBFL approaches based on method coverage results in many ties, Sohn and Yoo [38] proposed to propagate the suspiciousness scores of a method based on its statements, i.e., using the maximum suspiciousness scores of all involved statements for a method, and this propagation-based approach achieves better fault localization results on the method level than the existing SBFL.

3 COVERAGE-BASED FAULT LOCALIZATION FOR HASKELL

Although the existing coverage-based fault localization techniques are not proposed explicitly for imperative programming languages, their techniques (especially the coverage information) are given based on program elements of imperative programming languages, e.g., statements and methods. Therefore, it is not clear whether these techniques can be applied to other programming languages, e.g., functional programming languages.

Recently, declarative programming, especially functional programming, became popular. Moreover, many software companies like Facebook, Twitter, and LinkedIn directly use functional programming languages like *Haskell* in their development. That is, first, *Haskell* is widely used in advanced software development. Second, *Haskell* is popular and has uncompromising functional style that is free of imperative features such as side effects and assignment. Our hope is that by choosing a very "pure" language like *Haskell*, we can better isolate and contrast the differences affecting fault localization. Third, in the literature, few work investigated *Haskell* project debugging, including fault localization. Therefore, in this paper, we take *Haskell* as an example of functional programming languages, and study coverage-based fault localization for *Haskell*.

3.1 Gaps between Different Programming Paradigms

Haskell is a purely functional programming language, which is characterized by composing of expressions and being executed through evaluating expressions. Compared with imperative programming languages like *Java* and *C*, *Haskell* has many characteristics (e.g., recursion and pattern matching, type classes, higher-order functions, lazy evaluation, and program construct), which make it hard to use coverage-based fault localization directly.

3.1.1 *Recursion and Pattern Matching.*

Functional programs are usually defined by recursion: functions call themselves in the definitions; and when the calls return, the values are used as components for constructing new return values.

This programming style results in a very different (usually more succinct) code structure compared to the imperative style with iterative loops and explicit manipulation of pointers, which may require a different level of effort in achieving high code coverage.

3.1.2 *Type Classes.*

In functional programming, a type class [17] is a sort of interface that defines some behavior. If a type is a part of a type class, that means that it supports and implements the behavior the type class describes. Take the first line in Listing 1 as an example: we want to see the type signature of function == and find it is `(Eq a) => a -> a -> Bool`. Here, we can first read the type declaration like this: the equality function takes any two values that are of the same type and returns a `Bool`. Then, the `(Eq a)` is called a class constraint, which means the type of those two values must be a member of the `Eq` class. In fact, the `Eq` type class provides an interface for testing for equality, and any type where it makes sense to test for equality between two values of that type should be a member of the `Eq` class. Another example is the second line in Listing 1, where the `elem` function uses == over a list to check whether some value we are looking for is in it.

As a special feature in functional programming, Type classes make it harder to conduct fault localization. In fact, type classes belong to the type signature and are like pre-conditions of a function, which are not expressions and cannot be evaluated during execution. At the same time, the impact of type classes is hidden to some extent, which may influence far-away code. Therefore, this adds difficulty to coverage-based fault localization.

Listing 1: Example

```
1 (==) :: (Eq a) => a -> a -> Bool
2 elem :: (Eq a) => a -> [a] -> Bool
3 insertList ks tree = foldr insert tree ks
4 mysum :: Int -> Int -> Int
5 mysum x y = x + y
6 mymax :: Int -> Int -> Int
7 mymax x y
8     | x > y = x
9     | otherwise = y
```

3.1.3 *Higher-Order Functions.*

One feature of functional programming that turns out to be influential is high-order functions, which take other functions as arguments and/or produce functions as return results. For example, `fold` is a pattern that captures structurally inductive computation, where a function is applied to each structure layer and returns the accumulated result. We can define a function `insertList` that inserts a list of elements into a tree as shown in the forth line in Listing 1. It iterates through the key list and performs the insertions one by one, and then returns the final tree with the inserted keys.

In the context of testing, the use of higher-order functions adds difficulty to fault localization. On one hand, functions as arguments make the program structure more complex, and the function calls make the program trace more complex. On the other hand, functions like `insertList` have extremely short definitions, which puts the effectiveness of coverage in doubts, as it becomes trivial to cover the program.

3.1.4 *Immutable Data and Lazy Evaluation.*

Another feature in functional programming is referential transparency or purity, which means the symbol = represents true equality instead of a destructive update in imperative programming. Consequently, there is no concept of system state in functional programming; the behavior of a function is completely determined by its definition and the arguments passed to it.

Haskell uses the call-by-need evaluation strategy (also known as laziness), which delays the evaluation of an expression until its value is needed. For example, if we only need the head of a list, then the expressions that compute the rest of the list structure will not be evaluated.

3.1.5 *Program Construct.*

A *Haskell* program is composed of functions, which consist of expressions instead of statements, which results in the difference of localization granularity between imperative and functional programming languages. The structure of a *Haskell* program is relatively simple, containing only **functions**. Functions are the most basic and important component in *Haskell* programs. Within a function, there is no statement, just **expressions**. In fact, expressions can be nested, which means one expression can contain other smaller expressions and multiple expressions can make up a bigger expression. We give an example in Listing 1 to explain the characteristics of *Haskell* and our work all through the paper. In this example, within the **function** `mysum`, there are three **expressions**: `x`, `y`, and `x + y`.

Since there is no construct in *Haskell* that is equivalent to imperative statement, there is no construct coverage in *Haskell* that is equivalent to statement coverage in imperative languages [9]. Therefore, the existing statement-coverage based fault localization techniques cannot be applied directly to *Haskell* at all, indicating some adaption is needed to bridge the gap between programming paradigms in coverage-based fault localization.

3.2 Adaption Approaches from Imperative to Functional Programming Languages

Program coverage is a snapshot for test execution, ignoring program structures, and is useful in both imperative and functional programming [9]. Therefore, we regard it as one of the bridges between imperative programming languages and *Haskell* in fault localization. More specifically, this paper targets fault localization for *Haskell* on **functions**, because functions are the basic element of a *Haskell* program. Compared with the other constructs of a

Haskell program (i.e., files and expressions), functions are semantically independent units in *Haskell* programs, while file granularity is too coarse to provide insightful information and expressions are nested so as to be too short (a token) or too long (a function).

According to Section 3.1, the existing coverage-based fault localization techniques cannot be applied directly to *Haskell* programs since no construct coverage in *Haskell* is equivalent to statement coverage in imperative languages [9]. Following the existing work [9], we map the method coverage in imperative programming languages to function coverage in *Haskell* and the statement coverage in the former to expression coverage in *Haskell*. Moreover, due to the existence of nested expressions (i.e., an expression contains sub-expressions), the expression coverage is defined as a set of expressions that are covered by a given test suite, including all sub-expressions, which is also consistent with previous work [13]. This mapping enables the adaption of coverage-based fault localization for *Haskell*, which is given in Section 3.2.1 and Section 3.2.2.

Another thing to emphasize is how to deal with different types of tests. Tests in *Haskell* consist of unit tests and property-based tests. Unit testing is a type of software testing where individual units or components of a software are tested. However, in property-based testing, instead of writing specific testing inputs and oracles, developers write the properties that a function should satisfy. Property-based testing can reduce the efforts to write specific tests and consider various corner cases. On the contrary, the testing engine can automatically generate a large quantity of tests and automatically verify them. In our work, when collecting program coverage, we regard a property as a unit test and collect the coverage after all of its generated testing inputs are run.

3.2.1 *Function Coverage based Approach.*

We first adapt the existing coverage-based fault localization techniques to *Haskell* by changing the coverage information. That is, for each function e, we compute its suspiciousness score based on the formula of a SBFL technique (e.g., Ochiai) and rank functions in the descendent order of their suspiciousness scores. This approach is called **F**unction **C**overage **B**ased **A**pproach (**FCBA**) hereafter.

3.2.2 *Propagation based Approach.*

Similarly, we adapt the propagation-based technique to *Haskell* by changing the structure coverage. In particular, for each function f consisting of expressions $e_1, e_2, ..., e_m$ (considering all sub-expressions), we compute the suspiciousness score of each expression and take a series of statistics on these scores as the suspiciousness scores of the function. In particular, the statistics used in our adaption include (1) maximum, (2) mean, (3) median, (4) the largest, 2nd largest, ... In our approach, we first directly adapt the propagation approach, which means we use the maximum score as the score of a function. This approach is called **P**ropagation **B**ased **A**pproach (**PBA**) hereafter. Then, we notice that ties often occur in coverage-based fault localization in imperative programming paradigms, which means two program elements have the same suspiciousness scores. Therefore, we use the other three statistics to break the possible ties. Moreover, to reduce the computation cost in breaking the ties, these statistics are used only when ties occur. In other words, only when two functions share the same maximum scores, their other statistics are computed to distinguish them. This approach with **T**ie **B**reaking is called **PBA_TB** hereafter.

4 BENCHMARK CONSTRUCTION

To our knowledge, there is no fault localization benchmark in *Haskell*. Therefore, we manually build the first benchmark for *Haskell*, including real projects with real faults. To enlarge the number of faults, we also add seeded faults in this benchmark. To facilitate the usage of this benchmark, we collect coverage information, including function coverage and expression coverage. This benchmark is called **HaFLa** (abbreviation of **Ha**skell **F**ault **L**ocalization D**a**taset) hereafter, whose construction takes about four man-months, and the benchmark is shared on our website [16].

4.1 Projects

The build process is similar to Defects4J [21], a dataset containing real faults in *Java*, which is widely used in software testing research [22, 24, 34]. We construct this benchmark based on popular open-source projects from GitHub. In general, we first collect *Haskell* projects on their popularity and then remove the ones with problems in building, testing, or coverage collection process. The detailed process of project selection is given below.

Step 1: We identify the Top-50 popular projects whose primary programming language is *Haskell*. Here, we use the built-in language identification logic provided by GitHub[1]. Then, we rank them according to the number of stars, which is often used as an indicator of popularity [5, 36]. The ranking date is Oct. 24, 2021.

Step 2: For each project, following the official guide, we try to build it on our server locally. We find that most *Haskell* projects take the usage of Stack[2] as the build tool, which can automate the development process just like *Java* build tools such as Maven[3], Ant[4], and Gradle[5]. Therefore, to reduce the manual efforts, we only consider *Haskell* projects that use Stack. Moreover, some projects are discarded because they are not built successfully on our server due to the dependency and requirement issues of some specific version of GHC (*Haskell* compiler), Cabal (*Haskell* build tool), or GCC (*C* compiler). There are 43 projects remaining in total.

Step 3: We then focus on the tests of remaining projects. We first exclude projects that do not have tests. Then, we have requirements for the testing framwork for each project. For example, Tasty[6] is a modern testing framework for *Haskell*. It lets developers combine unit tests, golden tests, QuickCheck/SmallCheck properties, and any other types of tests into a single test suite. For other 31 projects, as we need to collect the detailed coverage information of each test, the testing framework is supposed to support the functionality to run each test separately. For instance, Tasty provides a pattern matching feature, which can match the test description and run the selected tests. In this step, if we cannot run single test, the project is discarded (19 remaining).

Step 4: Next, we exclude projects whose coverage information cannot be collected successfully by Haskell Program Coverage (HPC) [13], which is a high-fidelity and widespread-used code coverage tool for *Haskell*. More details on coverage collection is given in Section 4.4.

[1] https://github.com/github/linguist
[2] https://www.haskellstack.org
[3] https://maven.apache.org
[4] https://ant.apache.org
[5] https://gradle.org
[6] https://hackage.haskell.org/package/tasty

That is, initially we collect 50 popular *Haskell* projects, and then we remove 7 projects due to building problems, 24 projects due to test problems, and 16 projects due to coverage collection problem. Finally, we have only three projects, namely Pandoc, Hadolint, and Duckling. In other words, a large number of projects are removed in benchmark construction, and this phenomenon may raise the attention on software engineering support in *Haskell* community.

4.2 Real Faults

For each project we collect its real faults based on the development history recorded in GitHub, and construct faulty projects by injecting these faults.

Step 1: We first collect all closed issues (reported by Oct. 30, 2020) that are labelled by "bug", "fault", or "defect" (case insensitive), which may be related to faults. Then we manually check the contents of these issues to remove the ones irrelevant to faults, and finally get a set of issues reporting faults.

Step 2: For each of these issues, we manually check its contents and the corresponding commit(s) to identify the fault reported by the issue. In particular, the relevant commit(s) contain the reported fault-fixing patch. At the same time, developers often add or modify one or more tests that could reveal the fault in relevant commit(s). However, developers may make more modification besides fault fixing in one commit, e.g., refactoring and adding features. Therefore, we manually isolate fault-fixing modification and the added/modifies test(s).

Step 3: Next, we construct faulty and fixed versions for each fault. In particular, for each fault, we construct a faulty version and its corresponding fixed version by analyzing its relevant commit(s) (mentioned in Step 2), so that the faulty version contains the code before modifying and the fixed version contains the code after modifying. At the same time, the added/modified test(s) are kept in both versions. To guarantee the correctness of manual construction, we verify the testing results on the faulty and fixed version. In particular, all tests in the fixed version are passed, indicating the correctness of the fixed version, while at least one test in the faulty version is failed, indicating the reveal of the fault. Faults that violate the above expectations are regarded unreproducible and discarded.

Similar to the project selection process (in Section 4.1), we further remove the faulty programs with problems in building, testing, or coverage collection process, and finally have 31 real faults in all 3 subjects. The detailed information is listed on our website [16]. For each fault, we report the corresponding issue number and commit SHA(s) for readers to check the fault-fixing context (e.g., bug report and discussions). We also give the number of tests and failing tests for each fault. As we can see, our dataset is very diverse: SLOC ranging from 867 to 182,171 and #Tests ranging from 71 to 2,176. The constructed diverse dataset gives us the opportunity to conduct more analysis in various cases. Additionally, compared with existing dataset (*Defects4J* in *Java*, *Siemens*, *grep*, *gzip*, *make* in *C*), the SLOC and #Tests of our dataset are as large as them.

4.3 Seeded Faults

Although we try our best to collect real faults in open-source subjects, due to the lack of mature software engineering tools in *Haskell* community, the number of collected real faults is still small. Therefore, we construct seeded faults as a supplement.

In imperative programming languages such as *Java*, people often generate seeded faults by mutation testing [18]. That is to say, certain mutation testing tools slightly modify the original program according to some pre-defined rules (mutation operators) and generate new programs (mutants) to simulate faulty programs, where the modifications are simulated faults. In our scenario, we use MuCheck [23] to generate mutants for *Haskell* programs. It does this by parsing the *Haskell* source, and replacing a few common *Haskell* functions and operators with other similar functions and operators, and running the test suite to check whether the difference has been detected.

In our experiments, we slightly modify MuCheck so that it generates mutants instead of giving a mutation score. Specifically, we use all mutation operators provided by MuCheck, including (1) re-ordering for pattern matching, e.g., exchanging the order of two patterns in a function, (2) mutation of lists and list expressions, e.g., replacing a list with the identity element (empty list), replacing `l1 ++ l2` with `l2 ++ l1`, `l1`, or `l2`, and (3) type-aware function replacement, i.e., replacing any functions with all type-equivalent functions. Further, like previous work [9], we configure MuCheck parameters by assigning lower weights to operators that mainly generate equivalent mutants such as `doMutatePatternMatches` and `doMutateValues`. The HEAD that we perform the mutation is `d5c13dd` for Pandoc and `e4e0354` for Hadolint.

To our knowledge, MuCheck is the only accessible mutation tool for *Haskell* programs, but it is not maintained anymore since 2015 and cannot be applied to many *Haskell* projects, including some *Haskell* files in our projects. Therefore, we discard those files in our dataset. Then, for each project, we generate all mutants on usable files and remove the mutants that cannot be killed by any tests, or have problems in building, testing, or coverage collection process.

Finally, after filtering, we obtain 1,014 mutants in total, consisting of 96 in Pandoc and 918 in Hadolint. Similar to real faults, our seed faults are also diverse. The SLOC is ranging from 2,911 to 73,756 with an average of 9,618, while the number of failing tests is ranging from 1 to 22 with an average of 3.17.

4.4 Coverage Collection

To know which parts of the program each test executes, we need to collect coverage information. Due to the lazy evaluation feature of *Haskell*, its coverage is relatively difficult to collect. In this paper, we use Haskell Program Coverage (HPC) [13] to collect coverage. HPC is a high-fidelity code coverage tool for *Haskell* and includes tools that instrument *Haskell* programs to record program coverage, run instrumented programs, and display the coverage information obtained.

In our HaFLa benchmark, we use HPC to collect the coverage of all functions and all expressions in a program for each test. Specifically, we instrument the source program and execute each test separately. After each execution, we use the "`hpc show`" command to get a formatted representation of coverage information and clear the current coverage for the next execution.

We notice that in functional programming languages, especially in *Haskell* community, the demand for high-quality coverage tool is

weaker than that in imperative programming languages. As a result, HPC is not maintained anymore since 2016 and the applicability of HPC is not very wide, which means it does not work in many cases. Specifically, (1) HPC sometimes cannot instrument the program because of unrecognized syntax; (2) the intrumentation details (*.mix files*) are not matched with the recorded coverage number (*.tix files*); (3) the coverage information cannot be converted to be formatted; (4) the coverage information is incorrect, e.g., some covered functions are shown to be uncovered. Because of these reasons, many subjects are discarded as we state in Section 4.1, Section 4.2, and Section 4.3.

5 STUDY OF THE PROPOSED ADAPTION APPROACHES

In this section, we conduct an experimental study on HaFLa to learn the fault localization performance in *Haskell* using approaches given by Section 3.2.

5.1 Experimental Setup

In this experimental study, we evaluate the gap-bridging approaches between *Haskell* and imperative programming languages proposed in Section 3.2. In particular, we implement three groups of adaption approaches for *Haskell* and try to answer how effective these approaches are.

Function coverage based approach (FCBA): We first directly apply coverage-based fault localization approaches (given in Section 3.2.1) with various SBFL formulae (i.e., Ochiai, DStar, Tarantula) to the HaFLa benchmark. Due to space limit, we do not present the fault localization results of all these formulae. We only present the results on the Ochiai formula and the remaining results can be found on our website [16].

Propagation based approach (PBA): We apply propagation-based approach (given in Section 3.2.2) that computes the suspiciousness scores of expressions with various SBFL formulae (i.e., Ochiai, DStar, Tarantula) to the HaFLa benchmark, and present results on Ochiai in this paper.

Propagation based approach with tie breaking (PBA_TB): As ties are widely recognized to have unignorable influence on fault localization [37, 39, 44], we further apply propagation-based approach with tie breaking strategies (given by Section 3.2.2), and record the best result of the three comparison statistics.

Similar to the existing work on fault localization [3, 4, 25, 26, 28, 38, 47], in this experiment, we use Top-N and the ranking of the faulty function as the metrics. In particular, Top-N [4, 25, 26, 28] presents how many faults (faulty functions) are successfully located within the first N functions, and the ranking of the faulty function presents how many functions are ranked before the faulty function in the ranked list of suspicious functions. Moreover, if one fault contains more than one faulty functions, we use the highest ranking because developers could start the program repairing process once given any faulty code fragment. Since there are only 31 real faults in total, in this paper we present the fault localization results on real faults in terms of ranking, while the results on seeded faults in terms of Top-N (N=1,5,10,50). The whole fault localization results on real and seeded faults in terms of these two metrics are given our website [16].

5.2 Results and Analysis

The fault localization results of the proposed adaption approaches are given by Tables 1, 2, and 3.

In Table 1, each row represents the results on one fault. The first three columns show some basic information of each fault. Specifically, Column "ID" gives the identification number of a fault, Column "#Functions" gives the total number of functions in source code, while Column "#Functions covered by failing test(s)" gives the number of functions that are covered by at least one failing test, which consist of potential faulty functions. The next four columns present the gap-bridging approaches proposed in this paper. In particular, "FCBA" represents the results on function coverage based approach, "PBA" represents the results on propagation based approach, and "PBA_TB" represents the results on propagation based approach with tie breaking. Moreover, ties often occur in the fault localization results of imperative programming languages. To learn the existence of ties in *Haskell* programs, we visualize the ties in Table 1 by showing the ranking range of a faulty function. For example, if a faulty function f has a suspiciousness score $s(f)$, and in total $h(f)$ functions have higher suspicioussness scores than f and $tie(f)$ functions have the same suspicousness scores as f, we present the ranking result by a range between $h(f) + 1$ and $h(f) + tie(f)$ in this table. In other words, $tie(f)$ shows the number of functions in a tie occurred in fault localization.

Tables 2 and 3 show the results on seeded faults, where the former presents the overall results and the latter presents the results on each project. Here we only present statistics results on real fault, and the detailed results on each fault can be found on our website [16]. If a tie occurs in a fault, we regard the fault-localization result to be within Top-N if and only if the lowest ranking in the tie is within Top-N. Under this representation, developers are able to find the fault in no more than N tries definitely.

5.2.1 Analysis of FCBA. From Table 1, in most real faults, especially on larger subjects such as Pandoc and Duckling, the fault localization results are not ideal: most rankings are not high and few rankings are within Top-3. In fact, according to previous work on fault localization in imperative programming languages, original SBFL could achieve satisfying results. For example, on *Java* real faults dataset Defects4J [21], Top-1, Top-3, Top-5 are 80, 165, and 196 out of 395, respectively. However, things are different in *Haskell*. Although we can intuitively adapt SBFL to *Haskell* with only slight modifications, the performance is not generally good.

In many cases such as P1 and P3, higher suspiciousness scores are assigned to non-faulty functions. In many cases such as P5 and P6, although the ranking of fault function is relatively high, there exist multiple functions sharing the same suspiciousness scores, which is also called "tie".

From Tables 2 and 3, similar to real faults, the performance is also not satisfying: among the total 1,014 faults, only 8 are within Top-1 and only 110 are within Top-5. Table 3 presents results on each subject. Although Hadolint is smaller than Pandoc, the Top-1 accuracy of its mutants is higher, which is contradictory to results in Table 1, where real faults in Hadolint tend to have a higher accuracy. After manual inspection, we find that the ties appear more often in Hadolint, which lead to lower Top-1 accuracy.

Table 1: Results on real faults

ID	#Functions	#Functions covered by failing test(s)	FCBA	PBA	PBA_TB	Learning
P1	5,747	261	68-69	11	11	7-8
P2	5,746	270	3-9	7-8	3	2
P3	5,537	249	34-105	4-5	4	5
P4	4,922	722	9-13	8-10	8	6
P5	4,725	827	19-55	30-57	16	16-20
P6	4,711	907	14-20	10-11	11	3-7
P7	4,691	706	33-37	9-17	10	7
P8	5,314	338	6-11	1	1	1
P9	4,691	708	13	7-10	7	5
P10	4,824	691	14	10-12	10	8
P11	4,253	801	3	1	1	1
P12	4,253	801	5-10	2-4	2	1
P13	4,229	801	10-16	3-7	3	2
P14	4,211	990	1	1	1	1-3
H1	181	13	1	1-2	1	1
H2	170	20	5-6	10	5	4
H3	170	16	5	1-7	3	3
H4	114	21	5-6	10	5	3
H5	110	18	3	1-6	1	1
H6	114	20	3	1-6	1	1
H7	114	19	7	1-5	2	2
H8	58	4	1	1	1	1
H9	57	5	1	1	1	1
H10	135	45	1	1	1	1
H11	128	42	1	1	1	1
H12	128	43	1	1	1	1
H13	128	40	2-3	3	2	2
H14	128	44	1-6	4-9	4	2
H15	188	29	9	10	6	5
D1	6,063	460	16-95	5-87	12	7
D2	5,572	471	2-126	1	1	1

Table 2: Results on seeded faults (overall)

Approach	Overall				
	Top-1	Top-5	Top-10	Top-50	Total
FCBA	8	110	196	322	1,014
PBA	27	277	570	759	1,014
PBA_TB	83	440	719	835	1,014
Learning	86	458	750	890	1,014

Table 3: Results on seeded faults (on each project)

Approach	Pandoc					Hadolint				
	Top-1	Top-5	Top-10	Top-50	Total	Top-1	Top-5	Top-10	Top-50	Total
FCBA	8	11	15	65	96	0	99	181	257	918
PBA	27	63	71	88	96	0	214	499	671	918
PBA_TB	58	71	80	91	96	25	369	639	744	918
Learning	58	75	86	92	96	28	383	664	798	918

5.2.2 Analysis of PBA. In Table 1, the performance of PBA is better than FCBA in most cases. In other cases, they achieve the same performance or PBA is even worse due to the introduced ties.

In Tables 2 and 3, PBA could largely improve the fault localization performance. Specifically, overall, 277 out of 1,014 (27.32%) faults are located within Top-5 and 27 out of 1,014 (2.66%) faults are located within Top-1. The improvement of PBA over FCBA in Top-1, Top-5, Top-10, and Top-50 is 237.5%, 151.82%, 190.82%, and 135.71%, respectively. Additionally, the results on each subject are also consistent with the overall results. The difference is that in Hadolint, PBA still cannot produce any Top-1 result, which is due to the prevalent ties.

5.2.3 Analysis of PBA_TB. In above approaches (especially PBA), ties have negative influence on fault localization performance. Therefore, we finally analyze the results of PBA_TB. In Table 1, we successfully break the ties and obtain relatively higher rankings compared with PBA. Specifically, when ties occur, in most cases, our tie-breaking approach produces satisfying results in the ties, while in some cases, our approach produces worse results.

In Tables 2 and 3, PBA_TB could largely improve the fault localization performance compared with PBA. The reason behind this is that ties seriously degrade the Top-N metrics in PBA and our tie-breaking approach can produce accurate and high-performance results. Specifically, overall, 440 out of 1,014 (43.39%) faults are located within Top-5 and 83 out of 1,014 (8.19%) faults are located within Top-1. The improvement of PBA_TB over PBA (i.e., with the usage of tie breaking strategies) in Top-1, Top-5, Top-10, and Top-50 is 207.41%, 58.84%, 26.14%, and 10.01%, respectively. Additionally, the results on each subject are also consistent with the overall results. In particular, we successfully obtain 25 Top-1 results in Hadolint, compared with previous 0 Top-1 result.

> **Finding 1:** On HaFLa, PBA_TB achieves better results than PBA, both of which are better than FCBA. Specifically, for FCBA, the results are the most unsatisfying and few faults (both real and seeded faults) are ranked within Top-5, especially on larger projects. PBA could improve the performance to some extent, and the improvement on seeded faults is larger than real faults. However, the ties have bad influence on the performance. PBA_TB could break the ties successfully and show better results.

5.3 Case Study

Here we present case studies on two real faults of Pandoc to explain the gap between *Haskell* and imperative programming languages in fault localization.

5.3.1 Analysis on P2.

In this section, we analyze the real fault P2. Since its modification involves many expressions, for ease of understanding, we present its major modification in the fixing patch as below.

```
[-] gridTableWith::(Stream s m Char,HasReaderOptions st,
                                    Monad mf,IsString s)
[+] gridTableWith::(Stream s m Char,HasReaderOptions st,
             HasLastStrPosition st, Monad mf,IsString s)
[-] gridTableHeader::(Stream s m Char,Monad mf,
                      IsString s)
[+] gridTableHeader::(Stream s m Char,Monad mf,
                      IsString s,HasLastStrPosition st)
```

The developers add a Type Class called `HasLastStrPosition` to some functions, i.e., the parameter `st` must be a member of the `HasLastStrPosition` class. As a unique feature in functional programming, Type Classes define a set of functions that can have different implementations depending on the type of data they are given. However, because this constraint is not an expression and would not be executed, coverage-based fault localization cannot deal with its related information. In fact, its impact may exist in some places far away, leading to inaccurate localization.

We then analyze it from another aspect. One feature of functional programming is higher-order functions, which take other functions as arguments and/or produce functions as return values. That is, in

Haskell, many complex function calls exist. Under this circumstance, although most of the tests in our dataset are unit tests, they are not as "clean" as unit tests in imperative programming languages. In other words, the execution of a test in *Java* often results in the execution of a few methods besides some initialization methods, while the execution of an test in *Haskell* may cause the execution of much more functions. As a result, coverage-based fault localization may not obtain satisfying performance. In this case, many innocent functions (e.g., `renderRole`, `roleAfter`, `gridTable`, `table`, `roleBefore`, `unmarkedInterpretedText` in file "pandoc/src/Text/Pandoc/Readers/RST.hs") are assigned with large suspiciousness scores, which has bad influence on the localization results.

5.3.2 *Analysis on P5.*

In this section, we analyze the real fault P5. The modification in the fixing patch and some context is shown as below. Developers move `endline` from function `inline` to function `inlineContent`. Within these two functions, the elements in the lists are functions (e.g., `note`, `whitespace`).

```
   inline = choice [note, link
[-]                , endline
                   , strong, emph, code
   inlineContent = choice [whitespace, str
[+]                       , endline
                          , smart, hyphens, escapedChar
```

However, the localization results show that functions such as `endline`, `strong`, and `emph` are assigned with higher suspiciousness scores than `inline` and `inlineContent`. That is, elements (which are also functions) within a function are assigned with higher suspiciousness scores than the function itself. The reason behind it is higher-order functions. In fact, as functions can be arguments and return values in functional programming, their coverage cannot faithfully reflect the execution details, which means that simple statistics-based formulae used by the existing SBFL approaches are far from good. At the same time, similar to the analysis in Section 5.3.1, complex function calls in *Haskell* also weaken the performance. In this case, many innocent functions (e.g., `bulletListStart`, `definitionListItem`, `symbol` in file "pandoc/src/Text/Pandoc/Readers/RST.hs") are assigned with very high suspiciousness scores accordingly.

> **Finding 2:** After inspecting some cases in our HaFLa benchmark, we find that some *Haskell* features (e.g., type classes, higher-order functions, complex function calls) do have impacts on fault localization, and although these gaps exist, our approaches could achieve good results.

6 BRIDGING THE GAP THROUGH LEARNING

According to Section 5, the simple adaption approaches proposed in Section 3.2 are not satisfactory on *Haskell*, indicating that more work is needed to bridge the gap between programming languages in fault localization. Instead of defining a new formula, we propose to learn how to compute suspiciousness scores for *Haskell* through the fault localization results of other programming languages.

6.1 Approach

The proposed learning-based fault localization approach for *Haskell* transfers the fault localization knowledge for imperative languages to *Haskell* through machine learning. For any *Haskell* program, the learning-based fault localization approach first builds a predictive model based on the fault-localization data of another programming languages (e.g., imperative languages like *Java*), and then uses this model to predict whether each function of the *Haskell* program is faulty based on the coverage information of this program. In particular, for ease of fault localization knowledge transferring, instead of using the coverage information itself, we utilize the existing fault localization knowledge for imperative languages, i.e., reusing their suspiciousness score computing formulae since they are demonstrated to be effective to some extent.

6.1.1 *Feature and Label Design.*

In the learning-based approach, we take each program element as an instance in the training and testing sets. Ideally, we use the coverage information as the features of an instance. However, to fully utilize the knowledge of existing coverage-based fault localization approaches for imperative programming languages, we use the suspiciousness scores of various SBFL formulae as the features. That is, for each program element, we compute its suspiciousness scores with various formulae, each of which is regarded as a feature for the program element. Moreover, we remove the feature whose values are small, i.e., the formulae does not contribute too much, in the process of feature selection. Additionally, if a program element is faulty, we label it 1; Otherwise, we label it 0.

6.1.2 *Learning Algorithm.*

In the learning-based approach, we use Gradient Boosting [11], an ensemble method that produces a predictive model from an ensemble of weak predictive models. Specifically, we use GradientBoostingRegressor [35] to construct our predictive model, which can predict the probability of an element to be faulty. Then, elements in a subject can be ranked according to their probabilities. Moreover, in implementation, we set parameters as follows: the learning rate is 0.05, the loss is ls, and the criterion is friedman_mse.

Through the learnt predictive model on imperative languages, we can predict whether each function is faulty and produce a ranked list of suspicious functions based on their probability of being faulty, which can be viewed as the final suspiciousness scores computed by the learning-based approach. That is, the proposed learning-based fault localization approach builds the connection between coverage information and suspiciousness scores via another formula.

6.2 Evaluation

We evaluate whether the learning-based approach further improves fault localization results, compared with the adaption approaches. In particular, we study the performance of the learning-based approach for *Haskell* benchmark HaFLa by using the fault localization results on Defects4J [21], a large-scale benchmark in *Java*.

6.2.1 *Setup.*

Because we cannot obtain enough data in *Haskell*, in our experiment, we use method-level *Java* data as training data and observe the results in HaFLa benchmark. We use Defects4J to generate training data. Defects4J [21] is a famous real-world fault dataset in *Java*, containing hundreds of reproducible real faults on real-world projects. This dataset currently has two version: an original version (i.e., V1.2.0) and a recently released version (i.e., V2.0.0) [14] with

extra faults. To learning from a larger dataset, we use its latest version in this paper. Detailed information of this benchmark can be found on our website [16].

For each project of Defects4J, we perform on-the-fly bytecode instrumentation using ASM [7] and Java Agent[7] to collect the coverage information. Then, we detail the feature as follows. For each method in *Java*, we keep its three suspiciousness scores (i.e., Ochiai, DStar, Tarantula) and the largest 9 suspiciousness scores for its statements inside, forming a feature of length 12 in total; For each function in *Haskell*, we keep its three suspiciousness scores (i.e., Ochiai, DStar, Tarantula) and the largest 9 suspiciousness scores for its expressions inside, forming a feature of length 12 in total. To keep important information while avoiding redundancy, we only select finite suspiciousness scores. If a method/function contains less than 3 statements/expressions, which means we cannot get 12 values, the remaining positions are filled with 0.

6.2.2 *Results and Analysis.*

The results of the learning-based approach are presented in the last column of Table 1, the last rows of Table 2 and Table 3. Table 1 shows the results on real faults. From the table, the performance of the proposed learning-based approach is better than the other approaches except for P3, P5, and P14. Moreover, on these faults, the fault localization results of the proposed learning-based approach is very close to the best results. This observation is as expected since the learning-based approach takes as input most valuable information generated by the compared adaption approaches. However, the learning-based approach does not perform the best on three real faults, indicating that the connection between coverage information and suspiciousness score calculation is more complex than expected. That is, exploring good learning-based approaches is a promising direction for future research.

According to Tables 2 and 3, our learning-based approach outperforms all the other approaches in seeded faults. In other words, our learning-based approach performs better in seeded faults than real faults, indicating that the connection between coverage information and suspiciousness score calculation on seeded faults may be somehow easier to learn. Specifically, overall, 458 out of 1,014 (45.17%) faults are located within Top-5 and 86 out of 1,014 (8.48%) faults are located within Top-1. The improvement of the learning-based approach over PBA_TB in Top-5, Top-10, and Top-50 is 4.10%, 4.31%, and 6.59%, respectively. Additionally, we get consistent conclusion from the results on each subject.

> **Finding 3:** Our learning technique uses *Java* data for training and *Haskell* data for testing. The results show that learning further bridges the gaps between different programming paradigms and produces promising results.

7 THREATS TO VALIDITY

The internal threats to validity lie in the implementation of the approaches and the experiment scripts. To reduce this threat, the authors of this paper review the code and scripts, construct the benchmark, and implement and evaluate the proposed approaches by using existing tools [7, 13, 23, 35]. The external threats to validity lie in the projects, faults, and test cases. As the first piece of fault localization work on *Haskell*, we manually build a benchmark consisting of three real *Haskell* projects with test cases (involving 31 real faults and 1,014 mutation faults). Due to the lack of sufficient and general tool support for *Haskell*, we spent almost four man-months on this benchmark construction. Since this benchmark is not yet large enough, in the future we will add more real projects. The construct threats to validity lie in the measurement. To reduce this threat, we choose widely-used metrics [26, 38, 47] and more metrics like Mean First Rank and Mean Average Rank [30] will be used in the future.

[7] https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

8 RELATED WORK

Fault Localization [8, 32, 43, 46] aims to diagnose faulty program elements automatically and has been extensively studied to facilitate software debugging. Fault localization techniques often leverage various static and/or dynamic program analysis information to compute suspiciousness scores (i.e., probability of being faulty) for each program element. Program elements are then ranked in the descending order of their suspiciousness scores, based on which manual fault-fixing or automated program repair [12, 40, 45] can further be applied. Despite this rich literature, most existing techniques and studies are conducted on imperative programming languages. Consequently, this paper is the first work that focuses on the fault localization problem in functional programming and tries to bridge the gaps between functional and imperative programming.

Functional Programming Testing is rarely investigated although testing is important in the software development of functional programming [41]. QuickCheck [10] is a testing tool that facilitates developers write property-based tests. Braquehais and Runciman [6] propose a tool called FitSpec that provides automated assistance in the task of refining sets of test properties for *Haskell* functions. Grieco et al. [15] propose a fuzzer that leverages QuickCheck-style random test generation to automatically test programs that manipulate common file formats by fuzzing. Mista et al. [29] propose some probabilistic formulae and design heuristics capable of automatically adjusting probabilities in order to synthesize generators in QuickCheck. Besides property-based testing, Le et al. [23] present MuCheck, a mutation testing tool for *Haskell* programs which implements mutation operators that are specifically designed for functional programs. To our best knowledge, there is no existing work targeting fault localization in *Haskell*. Therefore, we are the first to attempt to solve it and our work could encourage more future work on functional programming testing.

9 CONCLUSION

In this work, we present the first work that identifies the fault localization problem in functional programming languages and make the first attempt to bridge the gap between different programming paradigms in coverage-based fault localization. We build the first fault localization benchmark in *Haskell* called HaFLa. Then, we explore a series of adaption and learning approaches of SBFL and conduct a study to evaluate the effectiveness of them. The results show that the adaption is necessary and useful, and the learning approach shows the promises of the direction.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 61872008.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[3] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 177–188.

[4] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 907–918.

[5] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[6] Rudy Braquehais and Colin Runciman. 2016. FitSpec: refining property sets for functional testing. In *Proceedings of the 9th International Symposium on Haskell*. 1–12.

[7] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).

[8] Bruno Castro, Alexandre Perez, and Rui Abreu. 2019. Pangolin: an SFL-based toolset for feature localization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1130–1133.

[9] Yufeng Cheng, Meng Wang, Yingfei Xiong, Dan Hao, and Lu Zhang. 2016. Empirical evaluation of test coverage for functional programs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 255–265.

[10] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[11] Jerome H Friedman. 2002. Stochastic gradient boosting. *Computational statistics & data analysis* 38, 4 (2002), 367–378.

[12] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[13] Andy Gill and Colin Runciman. 2007. Haskell program coverage. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. 1–12.

[14] Greg4cr. 2022. Defects4J – version 2.0. https://github.com/rjust/defects4j.

[15] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. *ACM SIGPLAN Notices* 51, 12 (2016), 13–20.

[16] HaFLa. 2022. HaFLa Homepage. https://github.com/Spiridempt/HaFLa.

[17] Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 2 (1996), 109–138.

[18] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[19] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[20] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

[21] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.

[22] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[23] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. Mucheck: An extensible tool for mutation testing of haskell programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 429–432.

[24] Feng Li, Jianyi Zhou, Yinzhu Li, Dan Hao, and Lu Zhang. 2021. AGA: An Accelerated Greedy Additional Algorithm for Test Case Prioritization. *IEEE Transactions on Software Engineering* (2021).

[25] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 169–180.

[26] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[27] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *Acm Sigplan Notices* 40, 6 (2005), 15–26.

[28] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[29] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for quickcheck generators. *ACM SIGPLAN Notices* 53, 7 (2018), 1–13.

[30] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[31] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[32] Frolin S Ocariza Jr, Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. 2016. Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability* 26, 1 (2016), 69–88.

[33] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[34] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[35] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[36] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.

[37] Deuslirio Silva-Junior, Plinio S Leitao-Junior, Altino Dantas, Celso G Camilo-Junior, and Rachel Harrison. 2020. Data-flow-based evolutionary fault localization. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 1963–1970.

[38] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[39] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* (2019).

[40] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1–11.

[41] Manfred Widera. 2006. Why Testing Matters in Functional Programming. In *7th Symposium on Trends in Functional Programming, University of Nottingham, TFP*.

[42] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[43] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[44] Xiaofeng Xu, Vidroha Debroy, W Eric Wong, and Donghui Guo. 2011. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21, 06 (2011), 803–827.

[45] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* 46, 10 (2018), 1040–1067.

[46] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312.

[47] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.

[48] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. 272–281.

[49] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* (2019).