International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:15, No:8, 2021

HaskellFL: A Tool for Detecting Logical Errors in Haskell

Vanessa Vasconcelos, Mariza A. S. Bigonha

Abstract—Understanding and using the functional paradigm is a challenge for many programmers. Looking for logical errors in code may take a lot of a developer's time when a program grows in size. In order to facilitate both processes, this paper presents *HaskellFL*, a tool that uses fault localization techniques to locate a logical error in Haskell code. The Haskell subset used in this work is sufficiently expressive for those studying Functional Programming to get immediate help debugging their code and to answer questions about key concepts associated with the functional paradigm. *HaskellFL* was tested against Functional Programming assignments submitted by students enrolled at the Functional Programming class at the Federal University of Minas Gerais and against exercises from the Exercism Haskell track that are publicly available in GitHub. This work also evaluated the effectiveness of two fault localization techniques, Tarantula and Ochiai, in the Haskell context. Furthermore, the EXAM score was chosen to evaluate the tool's effectiveness, and results showed that *HaskellFL* reduced the effort needed to locate an error for all tested scenarios. The results also showed that the Ochiai method was more effective than Tarantula.

Keywords—Debug, fault localization, functional programming, Haskell.

I. Introduction

FUNCTIONAL programming is a method of program construction that emphasizes functions and their application rather than commands and execution [5].

At first sight, the functional paradigm may confuse programmers. It may be because they usually start by learning the imperative paradigm, which has no particular way of handling the state. In light of that, several difficulties may appear when programmers try to learn a new way to write code with different reasoning. If they do not address these issues early, they might use the functional language as if using an imperative one for a long time, taking no real advantage of the functional paradigm. For instance, Fig. 1 (b) exhibits an example of functional programming, a function named `fact` which calculates the factorial of a given `n`. It is written in Haskell, and it is as straightforward as the mathematical factorial definition. Fig. 1 (a) shows an implementation of `fact` in Java, which contrasts with the Haskell definition, because it demands greater knowledge of the language constructs from the developer writing it.

Additionally, functional languages are pure or impure. Pure languages being the ones not allowing side effects anywhere and impure languages being the ones allowing them. Examples of pure functional languages are Haskell and Agda. Some impure ones are Lisp, Scheme, Clojure, Standard ML, F# and OCaml. F# is integrated into the platform .NET and reaches many users [12]. OCaml stands for Objective-ML and it "is an industrial-strength programming language supporting functional, imperative and object-oriented styles" [34]. Clojure is used by *Nubank* [33]. Haskell is an excellent choice for a functional language because it has a large and active community. It also has built-in concurrency and parallelism and supports integration with other languages [46].

Vanessa Vasconcelos is with Universidade Federal de Minas Gerais, Brazil (e-mail: vanessa.cr.vasconcelos@gmail.com).

```java
public static long fact(long n){
    if (n == 0)
        return 1;
    else
        return(n * fact(n - 1));
}
```
(a)

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```
(b)

Fig. 1 Factorial function in *Java* (a) and in Haskell (b)

Other two broadly used languages that implement functional concepts are Java and Kotlin. Java 8 introduced lambda expressions and functional interfaces, which profoundly improved the language's power. Kotlin offers both functional and object-oriented concepts alongside a strong integration with Java [13]. The latter allows Kotlin to be classified as very promising language because it removes a large part of migration and integration concerns for scalable systems. Additionally, a considerable number of large companies are already adopting Kotlin.

With that said, having good knowledge of the functional paradigm is a valuable skill for developers, regardless if they work directly with a purely functional language or with any other language offering functional concepts. Also, Haskell is an excellent first functional language because it allows developers to have a clear view of functional concepts.

A. Problem Definition

Bugs are reality on software development, and while experienced programmers may know their way among several bugs, some beginners may feel discouraged by them. Compilers are able to help detecting some simpler bugs. For example, [4] conducted a study about the `javac` compiler messages for students' Java code. The top 10 student errors they found in their study are:

(i) cannot find symbol
(ii) ')' expected
(iii) ';' expected
(iv) not a statement
(v) illegal start of expression
(vi) reached end of file while parsing

(vii) illegal start of type
(viii) 'else' without 'if'
(ix) bad operand types for binary operator
(x) <identifier> expected.

Some of the messages, such as `')' expected` and `';' expected` are really effective, other such as `illegal start of expression` may be more tricky. Additionally, [42] conducted a study focusing on Haskell novice programmers and the kind of mistakes they make. Their results are:

(i) `Parenthesis mismatch`: unbalanced parenthesis characters.
(ii) `Bad scoping`: issues with `let` and `where` constructs.
(iii) `Misunderstanding do blocks`: for instance, trying to bind names in a `do` block as the final action.
(iv) `Complex constructs`: their interpreter did not support `data` and `type` definitions and users attempted to use it anyway. This was noted as a mistake.
(v) `Incorrect syntax for enumFromThenTo syntactic sugar`: there were issues with the `..` notation. The authors consider that may have been problems with their tutorial material though.

All of the errors mentioned above are automatically identified by a compiler. Becker et al. [4] also enhanced error messages in their study to test how much an improved message is able to help. Their results indicate that it does help. Nonetheless, better messages do not extinguish the errors, neither syntactic errors nor logical errors. Logical errors are the ones a compiler can not automatically catch, and for that reason, they are even harder to identify. In light of that, this paper presents *HaskellFL*, a tool that locates logical errors in functional programming assignments written in Haskell. Starting from a source code with unexpected behavior and a few test cases, some of which outputting an unexpected result, and others producing the expected output, *HaskellFL* calculates and returns a list containing the most likely expressions to be triggering this unpredictable behavior, and this list is sorted from most to less probable. This suspiciousness list is created using fault localization techniques, thus, this work also evaluates the effectiveness of two different fault localization techniques in the Haskell context.

The main reason for choosing functional programming for this work is because the functional paradigm is less spread than object-oriented concepts; consequently, the support material for learning it is also less spread. The reasons for choosing Haskell in particular pass through its expressiveness, great dealing of complex data, and the fact that it is a purely functional language that may effectively help developers to grasp the concepts present in the functional paradigm. Haskell's laziness is also an advantage for this work because it provides better visibility of the code execution.

Other data serving as motivation is [39] and [15] where they analyzed the change history of a large software project focusing on one line changes. Their results showed that 10% of the total code changes involved a single line of code, and 50% were below ten lines. The study [15] specifically found that for Haskell, localized changes are 62.7% of all changes. So, a fault localization tool may be beneficial.

Adding remarks to Haskell as the right choice for studying; there are plenty of companies using it. Enumerating few, *Facebook* uses it internally in its advertising and spam filtering internal products as well as *Google*, which published a paper about their experience [38]. *Intel* has developed a Haskell compiler as part of their research on multicore parallelism at scale [31], *Microsoft* uses it in its compilers research, and Tesla also uses it in its internal products. A list containing several companies and the respective fields in which they use Haskell may be found in Haskell Cosmos website[1], including some of the companies mentioned above are listed there.

B. Goals

This work's primary goal is to evaluate the effectiveness of two fault localization techniques in the literature, Tarantula [21] and Ochiai [1], in the context of Haskell programs, and additionally, create a tool to aid Functional Programming beginners while debugging their Haskell problems.

This tool will receive a code written in Haskell containing a yet unknown logical error and some test cases divided into two sets, one set containing tests that evoke an error and the other one containing tests that allow the code to run smoothly. With these inputs, the tool will be able to run the tests and to locate what expression is the error root cause.

The main contributions of this paper are:

(i) A tool, named *HaskellFL*, which is able to locate logical errors in Haskell code.
(ii) The implementation of two fault localization techniques: Tarantula and Ochiai.
(iii) A test suite covering the chosen Haskell grammar's subset.
(iv) The evaluation of *HaskellFL* against a test suite using EXAM score.
(v) A Haskell interpreter for a subset of Haskell 2010 grammar.

The organization of the remaining sections of this paper is as follows. Section II presents the fault localization topic, enumerating the techniques used in *HaskellFL*, showing how to evaluate these techniques, and presenting related tools. Section III introduces the `SKI` combinators. Section IV discusses the related work and the remaining gaps that motivated the present work. Section V presents the proposed solution to the identified problem, discussing the requirements and implementation of the *HaskellFL* tool for detection of logical errors in Haskell. Section VI presents the test suite and the obtained results while testing *HaskellFL* against it. Section VIII concludes this paper, presenting its contribution and suggestions for future works. It is followed by Appendix A, which exhibits the subset of Haskell 2010 grammar supported by *HaskellFL*, and the bibliography.

[1]https://haskellcosm.com/

II. Fault Localization

This section brings an overview of two fault localization approaches and it also presents methods to evaluate them. Additionally, it shows fault localization tools in the literature.

A. Spectrum-Based Fault Localization

Jones et al. [22] presented a prototype tool that uses coloring statements in the code to detect fault locations; this is classified as a spectrum-based fault localization (SBFL) technique. The authors have a piece of code with a logical error and some test cases, a few producing the expected result for a given input, and other few producing an unexpected output for a different input. While executing the code, they color the coding statements with colors inside the green, red, and yellow spectrum.

In this scenario, red means the majority of test cases that run that line fails to achieve the correct output; green means the opposite; most test cases running that line succeed in achieving the correct output. Furthermore, yellow means the test cases executing that statement are the ones that succeed sometimes and fail other times, both near to the same percentage. It is worth remembering that the colors rely upon a spectrum, which means that one statement that has 80% failing test cases and 20% successful ones running it is colored in a darker red contrasted to one with 65% failing test cases and 35% successful test cases running it. Correspondingly, the same applies to green in the reverse order. Figure 2 depicts the example displayed in [22] that demonstrates what was just explained. The `mid` function in the figure prints the central element among three elements and will be used as an example again in this paper. Moreover, in Fig. 2, `P` indicates the given test case succeeds in achieving the expected output, i.e., it passes. `F` indicates that the test case did not achieve the correct output; thus, it fails. The black bullets in the intersections between test cases and code statements show that the given test case has executed that code statement. For example, test cases `2, 1, 3`, execute lines `1, 2, 3, 6, 7, 13`, and fail.

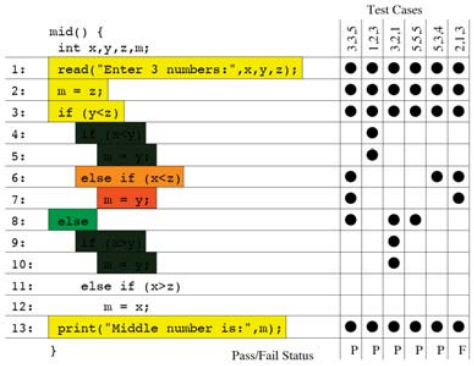

Fig. 2 Colored Code for Detecting Error Location, extracted from [22]

Lee et al. [29] adopted the same method, after adapting it to functional programs, to detect possible error locations in the process of correcting OCaml code. Their result is a set of pairs consisting of a holed program and a score for each possible error location. The lower the score, the more suspicious the expression is. Additionally, as the name suggests, the holed program contains holes in the expressions where an error may occur. It is also worth mentioning that the authors consider the size of the expression to calculate its score. They based their motivation for this on the Occam's razor principle [6], meaning they want to replace an expression as small as possible.

Table I shows two formulas from different methods - Tarantula [21], and Ochiai [1] - that were used to calculate the error localization in *HaskellFL*; both fall in the SBFL category.

TABLE I
Fault Localization Techniques' Formulas

Tarantula:	$\frac{\frac{failed(s)}{totalfailed}}{\frac{failed(s)}{totalfailed}+\frac{passed(s)}{totalpassed}}$
Ochiai:	$\frac{failed(s)}{\sqrt{totalFailed(failed(s)+passed(s))}}$

Tarantula utilizes all the standard information used by other testing tools: pass/fail information about each test case, the entities that were executed by each test case, e.g., - statements, branches, methods - and the program's source code under test [21]. Tarantula method intuition is that entities in a program primarily executed by failed test cases are more likely to be faulty than those primarily executed by passed test cases. Additionally, the method also allows some tolerance for the bug to be occasionally executed by passed test cases because they claim it often provides more effective results.

In [1], they show that for software fault diagnosis, the Ochiai similarity coefficient, known from the biology domain, outperforms several other fault localization methods. They attribute these results to the Ochiai coefficient being more sensitive to potential fault locations in failed runs than to activity in passed runs. This fact suits fine for fault localization because the execution of incorrect code does not necessarily lead to failures, while failures always involve a fault.

B. Mutation-Based Fault Localization

Another approach for finding logical errors are mutation-based fault localization (MBFL) techniques. In [35], the authors explain that this method works by introducing defects - mutants - in the program under analysis. The analysis relies on the assumption that most mutants form realistic faults, even if artificially seeded. Furthermore, it becomes possible to analyze the new code behavior against the test cases with the mutants in place. A big disadvantage of this method is that it is costly. To better measure this fact, in a technical report [36], the authors evaluated different techniques for finding faults' localization. Their data set was composed of 310 real faults and 2995 artificial faults in Java code. They took 100,000 CPU hours to get their results, mainly because of MBFL expensiveness.

In [28], the authors proposed a mutation testing tool for Haskell programs and also named mutations they consider suitable for functional programs. These mutations are:

(i) Replacing integer constant `N` with one of {`0, 1, -1, N + 1, N - 1`}.

(ii) Replacing an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another of the same class.
(iii) Negating the conditional in `if` statements.
(iv) Deleting a statement.
(v) Reordering pattern matching.
(vi) Mutation of lists and list expressions.
(vii) Type-aware function replacement.

More precisely, the first four items were originally applied for C programs, but [28] agreed they are still valid for functional programs, and they added the last three specifically for functional programming.

C. Faults Originated by Missing Code

Sometimes, bugs may be caused by the lack of an explicit expression in the program instead of an error in its expressions. Just et al. [25] cited by Pearson et al. [36] affirm that for 30% of cases, a bug fix consists of adding new code rather than changing existing code. Nonetheless, [49] states that even if a bug originates from a missing part in the code, such as an untreated corner case, fault localization techniques may still be helpful to bring attention to suspicious parts of the code by exposing possible control-flow anomalies.

Pearson et al. [36] evaluated the different fault localization techniques regarding the missing code scenario considering that the guideline for this case is to the technique to report the immediately following statement. Ideally, this should be exactly where the programmer should insert the code, and thus fault localization techniques are still able to bring awareness to the correct part of the code. Furthermore, [30] proposed a new missing code-oriented fault localization (MCFL) approach, which intuitively says that to identify a code-omission fault, the missing code site between two specific adjacent statements should be a candidate of fault localization. Such a site indicates the position of missing code in the faulty program. In other words, they consider both statements in the code and possible new code locations to calculate their suspiciousness scores.

In conclusion, bugs caused by missing code are an essential part of fault localization research. SBFL is still valid for several scenarios, including the one in this work; however, newer techniques as MCFL are improvements to the field.

D. Fault Localization Metrics

There are several literature methods for fault diagnosis in software testing. The question which arises after that is how to evaluate these different methods. Henderson [20] compiled several evaluation methods; some of them are reproduced here.

(i) **EXAM Score**. It calculates the percentage of program elements that a developer would have to inspect until finding the first fault. Formally, let `n` be the number of program elements and `r(s)` the rank of a given element for a fault localization method, the EXAM score is:

$$\frac{r(s)}{n}$$

(ii) **Tarantula Effectiveness Score (Expense)**. It calculates the percentage of program elements that do not need to be inspected to find the fault. Formally, let `n` be the number of program elements and `r(s)` the rank of a given element `s` for a fault localization method, the Expense score is:

$$\frac{n - r(s)}{n}$$

(iii) **LIL Probability Distribution**. It uses a measure of distribution divergence (Kullback-Leibler) to compute a score of how different the constructed distribution is from the "perfect" expected distribution. The advantage of the LIL framework is, it does not depend on a list of ranked statements and may be applied to non-statistical methods. Formally, let τ be a suspicious metric normalized to the `[0,1]` range of reals. Let `n` be the number of statements in the program. Let `S` be the set of statements. For all $1 \leq i \leq n$ let $s_i \in S$. The probability distribution is:

$$P_\tau(s_i) = \frac{\tau(s_i)}{\sum_{j=1}^{n} \tau(s_j)}$$

When evaluating a suspiciousness rank list, a fact to be considered is the tied scores present in it. The approach used in *HaskellFL* calculates the best and worst-case scenarios. The best-case scenario happens when among several lines with the same score, the developer starts examining them by the line containing the bug. Conversely, the worst-case scenario happens when a developer chooses to examine the line with the bug last. To exemplify, consider there is a bug in Line `2` of a four lines' program with a suspiciousness score list of `[0.5, 0.8, 0.8, 0.3]`, where the position in the array holds its score. For instance Line `1` has a score of `0.5`. In the best case scenario, a programmer would find the bug at first try, choosing to check Line 2 first, and the EXAM score for this is:

$$EXAM = \frac{1}{4} = 25\%$$

whereas for the worst-case scenario, a developer would chose to examine Line `3` before Line `2`, and the EXAM score for this scenario is:

$$EXAM = \frac{2}{4} = 50\%$$

In Section VI, results will be presented in terms of the EXAM score. This choice was made because the EXAM score is a great indicator of the effort level a developer needs to apply in order to locate a bug, and this is the key point *HaskellFL* aims to contribute.

E. Fault Localization Tools

To begin with, [43] introduced *ProFL*, a command-line fault localization tool for Prolog models. As happens for *HaskellFL*, *ProFL* takes a faulty Prolog model, a test suite for that model, and calculates which statements are most likely to

be faulty. It performs both Spectrum-Based Fault Localization and Mutation-Based Fault Localization.

Chesley et al. [8] presented *Crisp*, an Eclipse plug-in tool that allows a programmer to run regression tests after some change in her Java code. If a test fails unexpectedly, the programmer may edit parts of the code to ensure it still compiles and reruns the test focusing on the modified part. The programmer may interact with changes until finding the set originating the fault.

The work [9] uses a method that takes advantage of the information regarding method calls' sequences during program execution to calculate fault localization. It collects execution data from Java programs considering incoming method calls, i.e., how an object is used, and outgoing calls, i.e., how it is implemented. Moreover, [47] presented a cross-tabulation statistical method taking advantage of code coverage for test cases. A hypothesis test is also used to infer if execution results and each statement's coverage are dependent or independent of one another. Each statement's suspiciousness score depends on the degree of association between its coverage and the execution results.

Le et al. [27] presented *MuCheck*, a mutation testing tool for Haskell programs. The tool implements mutation operators that are specifically designed for functional programs (see Section II-B) and makes use of Haskell's type system to achieve a more relevant set of mutants. Besides that, there are other relevant works about error localization. Jose and Majumdar [24] present an algorithm for error cause localization based on a reduction to the MAX-SAT[2] problem; Ball et al. [3] show an algorithm that explores the existence of correct error traces among all the error traces pointed out by a compiler in order to localize what is causing the error, and Groce et al. [16] use distance metrics in order to better explain the error location. Distance metrics for program executions means a function `d(a,b)`, where `a` and `b` are executions of the same program, and `d(a,b)` is equal to the number of variables to which `a` and `b` assign different values.

HaskellFL differs from [43], [8] and [9] in the supported language. Additionally, [43] implemented a mutation-based algorithm that *HaskellFL* does not. The work [8] is different in the sense that their work is an interactive guide for helping to locate an error root cause. It serves as a guide for programming apprentices, similarly to Haskell, but they work as an interactive Haskell guide instead of looking for the error automatically. Furthermore, [9] implemented a different fault localization technique that was not used in *HaskellFL*. Wong et al. [47] also presented a new method which they compared and contrasted against Tarantula, which is a method present in *HaskellFL*. Le et al. [27] also used Haskell in their work, however, they implemented mutation-based fault localization while *HaskellFL* offers spectrum-based fault localization.

Finally, [49] compiled several Ph.D. and Master's Theses, techniques and tools about the fault localization topic, which makes it an excellent reference for related work. It contains the majority of the works mentioned above.

[2]MAX-SAT is the problem of determining the maximum number of clauses of a Boolean formula in conjunctive normal form, that may be made valid by an assignment of truth values to its variables.

III. SKI Combinators

A key concept used in *HaskellFL* is the `SKI` combinators [37]. `S, K, I` are supercombinators - functions with no free variables - that allows interpreting code. All expressions can be reduced to a combination of these. Listing 1 depicts the combinators as Haskell functions. Summarizing, the `S` combinator replicates one argument to two different functions. The `K` combinator is used when a value is constant regarding another value. To put it another way, in Listing 1, if `x` is constant regards `y`, `y` may be disregarded and `x` kept. And last but not least, the `I` combinator is just the identity function.

```
s f g x = f x (g x)
k x y = x
i x = x
```

Listing 1 `SKI` combinators in Haskell Notation

To better illustrate the SKI compilation algorithm, observe the following example. Given a function `\x -> x * x`, applied to the integer 10, the algorithm will follow the reduction steps in Listing 2. First, it will apply `S`, spreading the input to two instances in the body - a variable `x` and an application `(* x)` - after that, another application of `S` happens, in order to further spread the input thought the multiplication operation. Now, there is two `\x -> x` in the code, that may be reduced to supercombinator `I`. Finally, there is just multiplication, which is constant regarding `x` and may be reduced using the `K` combinator. After all the reductions, the expression `S (S (K *) I) I` is obtained, and it contains only operations that are straightforward to compute.

```
S (\x -> * x) (\x -> x) 10
S (S (\x -> *) (\x -> x)) (\x -> x) 10
S (S (\x -> *) I) (\x -> x) 10
S (S (\x -> *) I) I 10
S (S (K *) I) I 10
```

Listing 2 Example of `SKI` compilation

IV. Related Work

This section describes works related to *HaskellFL* and the process used to build it. Subsection IV-A talks about the state-of-the-art in Haskell compilers. Subsection IV-B cites several works on type errors and how to provide better feedback to them. Section IV-C mentions Haskell tutors and Section IV-D names some selected tools on automatic program repair.

A. Compilers

The most well-known and popular Haskell compiler is the Glasgow Haskell Compiler (GHC). The default compiler on the Haskell platform also includes tools to manage project building and packaging libraries. They also offer an interactive development environment, named GHCi, which may be used for incremental programming in the command line and provides handy tools for debugging. The original paper [23], which introduced the compiler, reinforces the fact the compiler is most written in Haskell, and its target language is *C*. Exemplifying its popularity, GHC is the recommended compiler on the introductory books to Haskell, "Haskell:

The Craft of Functional Programming" [44] and "Thinking Functionally with Haskell" [5].

GHC is an open-source project[3], it is on Version 9.0.1 to this date, and it is continually updated. It is an excellent tool for all the motives cited above, but there is still room to improve, as seen regarding the confusing type error feedback depicted in Fig. 3, further scrutinized in Section IV-B. Another widely known Haskell compiler is Hugs (Haskell User's Gofer System), which was the compiler reference for Haskell prior to GHC. It is no longer in development; its last release is from May 2006. Furthermore, there is a compiler named Helium created by [18], which has educational purposes and provides more detailed feedback. For example, given the `remove` function in Listing 3 with an error in Line 4, where a developer wrote `n = x` instead of `n == x`, the feedback returned by Helium, according to its creators, is the one displayed in Listing 4. It points to an error in the second equal sign and says that it may not exist another attribution after a first one. The expected input is an expression, an operator, or a constructor operator. It does not detect the most probable error cause; however, it gives better feedback showing a double attribution problem.

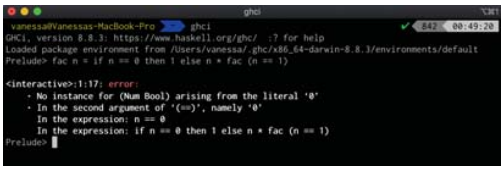

Fig. 3 Type Error pointed by GHCi

```
1 remove :: Int -> [Int] -> [Int]
2 remove n [] = []
3 remove n (x:xs)
4     | n = x = rest
5     | otherwise = x : rest
6         where rest = remove n xs
```

Listing 3 `Remove` function with an error in Haskell

```
1 (4,16): Syntax error:
2     unexpected '='
3     expecting expression, operator, constructor
4     operator,  '::',
5     '|', keyword 'where', next in block (based
6     on layout), ';'
7     or end of block (based on layout)
```

Listing 4 Feedback provided by Helium extracted from [18]

GHCi also points to a parser error on the second equal sign with no hints on how to fix it. Some other interesting remarks about Helium are that as the compiler aims to stimulate functional languages, they look for being as modular and straightforward as possible. Their code and idea are not very hard to follow, and as usual, type inference is the challenging and compelling section of their work, and their solution passes by tight constraint solving and global constraint solving.

[3]https://gitlab.haskell.org/ghc/ghc/

B. Type Errors

There are several works on compilation errors and how to provide appropriate feedback to them. To cite a few, there are [50], [7], [19], [41] and [4].

To exemplify the topic, look at the following function in Haskell to calculate the factorial of `n`:

```
1 fac n = if n == 0
2     then 1
3     else n * fac (n == 1)
```

Listing 5 Factorial function with error in Haskell

This function has a type error on Line 2, more precisely, in the expression `fac (n == 1)`, but as depicted in Fig. 3, the error accused by GHCi, the interactive development environment provided by GHC compiler, is in Line 1 when `n` is checked against `0`; this happens because, in the `else` clause, `fac` is called with a boolean parameter, binding the input to the boolean type. The comparison with `integer 0` fails on the following iteration, and this may be solved by declaring the type of the function explicitly instead of allowing the compiler to infer it. However, this will most likely confuse novice programmers, who may not be aware of this behavior.

In [50], the authors propose improving compiler error messages by looking at all possible errors as a whole and just reporting the most likely error instead of the first one encountered; the latter is how compilers handle their error messages usually. Their work uses Haskell. Fig. 4, illustrated in their paper, is used to explain their approach to the problem.

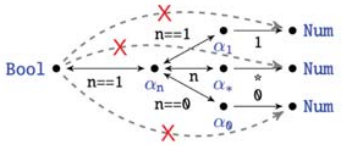

Fig. 4 Graph for Diagnosing Type Errors, extracted from [50]

A brief explanation: first, they model the set of constraints in the code as a constraint graph. Fig. 4 represents the erroneous factorial code, depicted in Listing 5. The nodes α_0, α_1, α_n and α_* represents the types of `0`, `1`, `n` and the first parameter of multiplication ($*$), respectively. The bidirectional edges mean type equality between nodes, for instance, α_n and α_0 are supposed to have the same type.

Each edge is also annotated with the expression that generates it. The direct edges represent type classes, the edge between α_1 and `Num` indicates that α_1 must be of type `Num`. The dashed edges are derived by transitivity. Furthermore, the edges are then classified as satisfiable or unsatisfiable. The red `X` means unsatisfiable. Exemplifying, `Bool`, and `Num` may not be the same.

The last pass is to use Bayesian principles, from the probability domain, to detect which edge is the most likely error source; the correct answer, in this case, is `(n == 1)`. In conclusion, it is an easy to follow method that may improve type error localization and help users in a topic that traditionally causes great confusion.

C. Haskell Tutors

This section mentions researches in building systems that offer a more driven feedback for tutoring functional programming apprentices such as [18], [14] and [17].

The tutor created by [14] is an excellent tool for Haskell and functional programming beginners. Their tutor offers incremental feedback, which means that at any point, a student feels stuck with a problem; he may ask the tutor for a hint. The tricky part is that an instructor must provide well-written solutions to the tutor and a configuration file, customizing the feedback with tips he believes would help his students to better comprehend the proposed solution and the process leading to it; this is a must to make the tutor effective. Otherwise, students may continue confused about the best path to follow to solve their problems. If more than one solution is applicable, the instructor must submit all the solutions he wants his students to know. Another remark about their work is that their tutor is a web application, making it accessible to everybody. They also use a compiler with improved error feedback, which is described in Section IV-A. Their model tracing and property-based test strategies have similarities with the strategies adopted by [29]. Their goal is to provide correct guidance that will allow the programmer to fill the holes he may have left in his code by not knowing what expression to use in a specific part of the program. To achieve that, they rely on the provided instructor's solution and in the language grammar, trying to fill the blanks with constructs available in the target functional language and with lambda calculus concepts. The latter may find equivalent expressions in the code, making it more straightforward for the tutor to interpret.

Finally, there is the project Try Haskell [10], which is worth to be mentioned. This project does not provide customized feedback about the logical errors in the code. However, it is an excellent way to start with Haskell, having a friendly and interactive tutorial about its basics.

D. Automatic Program Repair

Automatic program repair is likely the next step for research after finding code bugs automatically. This section brings up works on automatic program repair.

Lee et al. [29] created a system named *FixML* to diagnose and correct logical errors in OCaml. To do so, they need four inputs: an incorrect resolution for a program, a solution, the function name for the problem, and a file containing passing and failing test cases. As a result, *FixML* produces a repaired program consisting of the incorrect program modified to function correctly. The provided solution not necessarily follows the same structure of the program the system is trying to fix. The authors used the solution while rebuilding the code, but it is not at all a plain copy.

Kneuss et al. [26] wrote other paper on the subject to repair programs written in a Scala subset. Their process to locate a code fault starts by doing dynamic analysis using test inputs generated automatically. They combine enumeration and SMT-based techniques. Additionally, they collect traces from erroneous executions and compute common prefixes of branching decisions. On the program repair angle, they use the existing program structure as a hint to guide it. They rely on user-specified tests and automatically generated ones to localize the fault and speed up synthesis. Moreover, [45] presents a tool to find and fix bugs in Liquid Haskell. Liquid Haskell is a framework for annotating Haskell programs with refinement types, which are types decorated with predicates [45]. In their master thesis, the authors introduced a fault localization algorithm for constraint-based type systems, which searches for a minimal unsatisfiable constraint set using the type checker as guidance. To optimize the search process, they exploited the structure of Liquid Haskell constraint sets. They also presented a predicate discovery algorithm for constraint-based type systems, which allows the type checker to verify additional correct implementations.

V. Detecting Logical Errors in Haskell

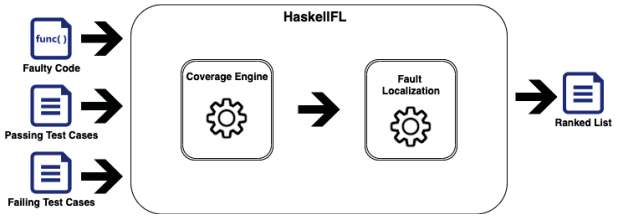

Fig. 5 *HaskellFL* architecture

Fig. 5 exhibits the architecture of the proposed solution to locate logical errors in Haskell. The tool is called *HaskellFL*.

HaskellFL expects three file paths as inputs, (i) one for the faulty Haskell code, (ii) another for the text file containing the passing test cases, and (iii) the last one for the text file with the failing test cases. Cabal, the standard package system for Haskell, was used in *HaskellFL*. Figure 6 exhibits an example of *HaskellFL* execution using this package. In other words, `cabal run` is invoked indicating the target to be executed, which is in this case *HaskellFL*, followed by the args expected by the program itself, i.e., the three files enumerated above. Optionally, it is possible to specify the technique of choice: Tarantula or Ochiai. If this information is omitted here, the program will prompt for it later. Furthermore, *HaskellFL* can also interpret the test cases, and expose their results. To do that, *HaskellFL* must be called as in Figure 7, with the code and test case paths, followed by the keyword `run` and the name of the function to be interpreted.

```
cabal run HaskellFL faulty-code.hs tests-pass.txt
tests-fail.txt [method]
```

Listing 6 *HaskellFL* execution command using Cabal

```
cabal run HaskellFL faulty-code.hs tests-pass.txt
tests-fail.txt run function-name
```

Listing 7 Command for *HaskellFL* interpreting the test cases

After having the needed inputs to run *HaskellFL*, it is necessary to obtain the code coverage for the buggy Haskell code regarding every test case separately. The count is divided between two independent sets representing the passing and the failing test cases. In possession of these two sets' data, the next

step is to feed them to the formulas exhibited in Table I to calculate the suspiciousness rank for each fault localization method.

A. HaskellFL *Tool*

This section details *HaskellFL* requirements. The first one is the Haskell 2010 grammar subset to be contemplated in *HaskellFL*. The notable parts of Haskell 2010 left out of *HaskellFL* are list comprehensions and `do` notation. However, they may be included as extensions for future work. Nevertheless, *HaskellFL* covers abstract data types, pattern matching, guards, `case`, `if-then-else`, `let` and lambda expressions, among several other features which are enough to support and guide Haskell beginners. It is important to say that the tool handles Haskell's layout rules. Moreover, the full grammar accepted by *HaskellFL* is available in full in Appendix A.

Secondly, *HaskellFL* needs to calculate the bug location using one or more fault localization techniques. To do so, it was necessary to first calculate and make available the coverage count for each statement, making note if the generated coverage corresponds to a passing or a failing test case. This feature's implementation allows additional fault localization methods to be easily added to *HaskellFL* in the future.

Thirdly, a Haskell interpreter needed to be created to obtain the code coverage map. The primary reason behind this decision was to allow the extension of *HaskellFL* in the future to repair Haskell code, as it is done for other programming languages as described in Section IV-D. Another viable extension to *HaskellFL* is to implement mutation-based fault localization techniques, also mentioned before in Section II-B.

B. Implementation

Fig. 6 exhibits *HaskellFL* high level block diagram. The diagram is divided into four main blocks representing four processes: Parser, Transformation, Interpreter, and Fault Localization.

The first block encapsulates the parsing process. The lexer was built using Alex[4] and the parser using Happy[5]. These are the Haskell equivalents for `Lex` and `YACC` respectively, and they also are the same tools used by Haskell compiler GHC. An alternative approach was to use one of the several libraries of parser combinators available such as `Parsec`[6]. The first option is restricted to LALR parsing, and the latter favors LL parsing [11]. In the end, the combo Alex and Happy was chosen because besides being more robust and offering better support for LR grammars, it also offers better visibility and control of each step, facilitating *HaskellFL* extension to repair logical errors in the future. It is also worth mentioning the BNF Converter[7], which is a powerful tool with uncomplicated

[4]https://www.haskell.org/alex/
[5]https://www.haskell.org/happy/
[6]https://wiki.haskell.org/Parsec
[7]http://bnfc.digitalgrammars.com/

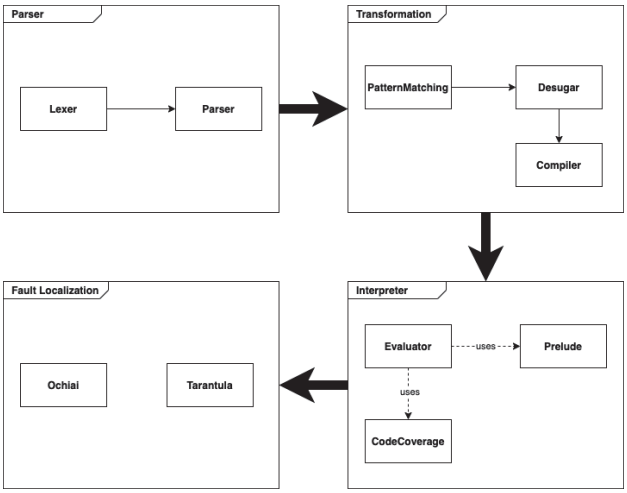

Fig. 6 HaskellFL high level block diagram

implementation, even though it is not suitable for Haskell grammar.

The parser maps tokens to a set of Haskell abstract data types. It was fundamental to *HaskellFL* to keep the expressions' lines while parsing the tokens to further calculate code coverage.

The second block is responsible for transforming the data types generated during the parsing process into `SKI` combinators. These data types are as close as the lambda calculus terms as possible. The parser already returns application, variable and lambda abstraction types, but it also returns data types corresponding to `let`, `case` and `if-then-else` expressions, that are later transformed in terms of the first three during the `desugar` phase. All the needed rules to translate a high-level functional language into lambda calculus are in Peyton Jones [37]. Additionally, before reaching the `desugar` step, pattern matching needs to be addressed. This goal was achieved by implementing the `match` function, also provided in detail in Peyton Jones [37]. Moreover, `desugar` step was also responsible for simplifying the fixed-point combinator and the other built-in functions.

`Compile` step for its turn, is responsible for transforming the desugared code into `SKI` combinators. This function implements the transformation rules presented in Section III. With that said, the second block outcome is formed by a set containing the `SKI` combinators alongside the final literals and the new local functions.

In the following block, there is the interpreter step, where `evaluator` function orchestrates calls to the previous blocks' functions. `Evaluator` extends the small prelude with the locally declared functions. This step is composed of constant exchanges between its internal blocks, represented in Fig. 6 for dashed arrows for organization purposes only. These exchanges reflect the process of getting and adding functions to the prelude and the constant execution stack updates.

And last but not least, in the fourth block, the faulty line in the Haskell code is calculated using the chosen fault localization techniques. These, for their turn, use the

coverage map exposed by the third block. This step may easily include other different coverage-based fault localization methods. Furthermore, an absence in the *HaskellFL* tool is type checking. Type checking was not implemented in *HaskellFL* because this would be an overkill for the need to obtain code coverage. However, to extend *HaskellFL* for repairing Haskell code, this is an important step. One important heuristic used to speed up finding a new bugless expression to replace a bug is to cut all candidates that are not type compliant out of the search.

C. Walkthrough

To better understand the complete process that *HaskellFL* follows, look at the example in Listing 8. A small piece of `LinkedList` module was extracted from a problem present in the test suite. This small part does not contain any bugs, but it serves to understand the whole process. The module has a generic data type `LinkedList a`, written using record syntax, and a function `fromList` that creates a `LinkedList` from a regular Haskell list.

```
module LinkedList (LinkedList, fromList) where

    data LinkedList a =
            Nul
            | LinkedList { datum :: a,
                           next :: LinkedList a }
            deriving (Eq, Show)

    fromList [] = Nul
    fromList (x:xs) = LinkedList x (fromList xs)
```

Listing 8 `LinkedList` module

Fig. 7 shows the generated AST for `LinkedList` source code after the parsing process. Some details were omitted, such as every terminal knowing its own position for better clarity in the figure, nonetheless, there are one data type declaration under the `DataDecl` set and two function bindings under `FunDecl` set. The `LinkedList DataDecl` has two constructors, `Nul` and `LinkedList`, with different arities and `fromList FunDecl` has two different bindings, one matching an empty list, i.e. `Nil`, and the other matching a non-empty list, i.e. `Cons x xs`. `MatchPat` is the label indicating the pattern matching for each specific function and `MatchBody` keeps the body of function binding for that respective pattern.

In the example, `Nul` is the `MatchBody` for the empty list, and an application of two other applications is the `MatchBody` for the non-empty list. The internal nodes representing the lambda calculus applications are displayed in yellow. Adding to the `MatchBody` for the non-empty list, the first application is of the constructor `LinkedList` to the head of its `MatchPat` and the second application is of the function `fromList` to the tail of its `MatchPat`. In light of that, it is important to restate that `MatchBody` was already transformed into a lambda calculus expression.

Once parsing is done, `match` function can be called. `match` transforms the pattern matching function bindings in a `case` expression such as the one exposed in Fig. 8. Colored in blue in the diagram, there are two concepts presented in Peyton

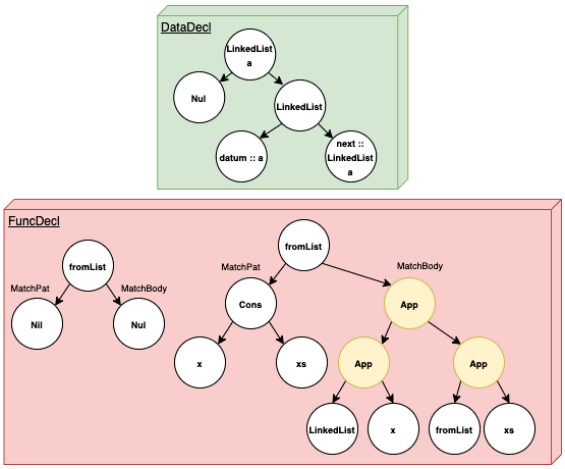

Fig. 7 LinkedList AST

Jones [37] and introduced to the code during this step. Firstly, the idea of pattern matching failing, i.e. a pattern mismatch, represented by `Fail`. Secondly, the `FatBar` operator, also represented for `[]`, which obeys the following rules: In other words, if `FatBar` operator is applied to two expressions, the result will be the first expression that is not `Fail`. It is essential to say that if the first expression fails to terminate, `[]` will also fail. Finally, the resultant `case` expression is attached to a lambda abstraction, as the body of the same, and added to the extended prelude, after being wrapped to another layer composed of the fixed-point combinator, responsible for taking care of recursion.

$$a[]b = a, \;\; if \;\; a \neq Fail$$
$$Fail[]b = b$$

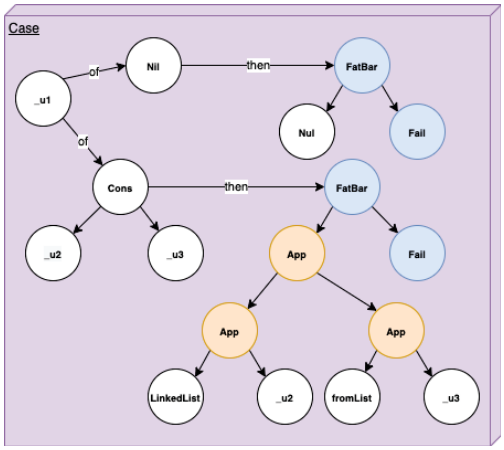

Fig. 8 Pattern Matching `case` Expression

In sum, after the process demonstrated in this section, `fromList` may be called from another function in the `LinkedList` module, or with input test cases from *HaskellFL*, such as the illustrative example shown in Listing 9 and this way it is possible to obtain `LinkedList` code coverage.

```
fromList ["UFMG", "UFV"]
fromList [False, True, True, True]
```

Listing 9 `test-cases.txt`

Concluding, *HaskellFL* is publicly available in GitHub[8], together with the test suite.

VI. Result Discussion

This section studies two different buggy versions of `mid` function displayed in Listing 10. `Mid` calculates the middle element among three given elements. In the first faulty version, `mid` has a bug in Line 2. The `if` block is `if y < z - 1` instead of `if y < z`. The second buggy version presents a bug in Line 6 where we have `then y` instead of `then x`.

```
module Main where
 mid x y z = if y < z
   then if x < y
        then y
        else if x < z
             then x
             else z
   else if x > y
        then y
        else if x > z
             then x
             else z
```

Listing 10 `Mid` function in Haskell

Table II maps the code coverage for each function call for a given input. For instance, a `mid` Version 1 call with inputs `3, 3` and `5` run lines `2, 3, 5` and `6`. The `P/F` column indicates if the output for the given input is the expected one of a non-faulty version of the code, i.e., it is a passing test case represented by `P`, or if it is an unexpected output for the specified input, i.e., it is a failing test case, represented here by `F`.

TABLE II
Code Coverage and Fault Rank for `mid`

	Test cases/Lines	1	2	3	4	5	6	7	8	9	10	11	12	P/F
Version 1	3 3 5		●	●		●	●							P
	1 2 3		●						●		●		●	F
	3 2 1		●						●	●				P
	5 5 5		●						●		●		●	P
	5 3 4		●						●	●				F
	2 1 3		●	●		●	●							P
Tarantula		0.0	**0.50**	0.0	0.0	0.0	0.0	0.0	0.67	0.67	0.67	0.0	0.67	
Ochiai		0.0	**0.58**	0.0	0.0	0.0	0.0	0.0	0.71	0.50	0.50	0.0	0.50	
Version 2	3 3 5		●	●		●	●							P
	1 2 3		●	●	●									P
	3 2 1		●						●	●				P
	5 5 5		●						●		●		●	P
	5 3 4		●	●		●		●						P
	2 1 3		●	●		●	●							F
Tarantula		0.0	0.50	0.63	0.0	0.71	**0.83**	0.0	0.0	0.0	0.0	0.0	0.0	
Ochiai		0.0	0.41	0.5	0.0	0.58	**0.71**	0.0	0.0	0.0	0.0	0.0	0.0	

Finally, under its respective labels, there are the suspiciousness scores for each statement for Tarantula and Ochiai techniques according to the formulas presented in Table I.

Describing the calculus, there are in total one failing and five passing test cases, thus `totalfailed = 1` and `totalpassed = 5`. For instance, for the fifth line from second `mid` version, there are two passing test cases covering it, as well as one failing test case, therefore, `failed(5) = 1` and `passed(5) = 2`, with this, Tarantula score for Line `5` is:

$$Tarantula(5) = \frac{\frac{failed(5)}{totalfailed}}{\frac{failed(5)}{totalfailed} + \frac{passed(5)}{totalpassed}} = \frac{1}{1+\frac{2}{5}} = \frac{5}{7} \approx 0.71$$

Similarly, the calculus for Ochiai suspiciousness score is:

$$Ochiai(5) = \frac{failed(5)}{\sqrt{totalfailed(failed(5) + passed(5))}} = \frac{1}{\sqrt{3}} \approx 0.58$$

Furthermore, the scores highlighted in bold are the ones for the statement containing the error. For Version 2, both methods assign a higher score for Line 6, finding the correct error localization. On the other hand, for Version 1, the methods do not rank the line with the error first. Ochiai assigns the highest score for Line 8, and Tarantula ranks Lines 8, 9, 10, and 12 higher than the correct error localization, that is Line 2.

Despite scores for the buggy lines in both versions of `mid` not being repeated in the results, several other statements received matching suspiciousness scores. As explained in Section II-D, one approach for this situation is to calculate the best and worst-case scenarios.

In the first case, a programmer would guess right the line containing the bug the first time while going through the list of even scores, and in the latter, a programmer would verify all the other lines with the same score before examining the buggy line.

In the final analysis, the EXAM score was calculated. This score indicates the percentage of the program that should be checked until the error location is reached. The results for `mid` function are displayed in Table III. For `mid` Version 1, a programmer would have to analyze 42% of the program if she follows Tarantula scores and 17% of it if she follows Ochiai scores. For `mid` Version 2 the results are even better. A programmer would have to analyze 8% of the program for both methods. To put it another way, in the worst of the studied scenarios, a Haskell developer would have to look for the bug in less than half the original amount of code lines before finding the bug.

TABLE III
EXAM Score for `mid` Function

	Tarantula	Ochiai
Mid Version 1	42%	17%
Mid Version 2	8%	8%

A. Test Suite

The test suite is composed of 24 problems covering Haskell's chosen subset shown in Appendix A. These problems are submissions from students in the Functional Programming class at UFMG, together with both versions of `mid` function presented in Section VI and code publicly

[8]https://github.com/VanessaCristiny/HaskellFL

available on GitHub that were submissions for Exercism's Haskell exercises track[9]. The 24 problems' source code is available in GitHub together with the source code for *HaskellFL*.

TABLE IV
TEST SUITE

Program	#Lines	#Tests	Ranking	
			Tarantula	Ochiai
mid (Version 1)	12	6	5	2
mid (Version 2)	12	6	1	1
dropWhileClone	33	10	3	1
dropWhile	10	9	1	1
break (Version 1)	27	5	1	1
break (Version 2)	27	8	1	1
toTuples	28	10	1	1
remdupsReducer	27	7	1	1
joinr	16	12	1	1
separateTuplesByType	23	7	1	1
flip	20	5	1	1
unzip	13	3	1	1
maxSumLength	8	11	1	1
binary-search-tree	55	8	2	2
grade-school	67	7	1	1
luhn	45	6	2	2
raindrops	34	8	1	1
resistor-color-duo	44	7	1	1
robot-simulator	69	9	1	1
roman-numerals	23	8	1	1
simple-linked-list	40	6	1	1
space-age	28	7	1	1
sum-of-multiples	34	7	3	1
triangle	35	8	6	5

Once the examples used to run the tests were the students' final submitted versions, they did not have errors that needed to be fixed in their majority, so the bugs were introduced in the code base, to be later detected by the fault localization techniques. The introduced bugs were not repeated.

Table IV names all the programs on the suite. As explained before, `mid` calculates the middle element among three elements. Regarding to the Functional Programming class submissions, the names chosen by the students were kept and their content is as follows:

(i) `dropWhileClone`/`dropWhile`. It drops elements while a condition is true and then stops returning the remaining elements once the condition is false.
(ii) `break`. It divides a list into a tuple of lists, breaking it at the point where a given condition is true.
(iii) `toTuples`. It transforms two lists into a list of tuples.
(iv) `remdupsReducer`. It removes the first element of a list if it is equal to the second one.
(v) `joinr`. It adds an element to the head of a list if it is not equal to the list's current head.
(vi) `separateTuplesByType`. It transforms a list of tuples into a tuple of lists, one list with all the first components and the other one with the second's tuple components.
(vii) `unzip`. Same as `separateTuplesByType`.
(viii) `flip`. It flips a function chain order.
(ix) `maxSumLength`. It calculates a tuple with three elements. The first one being the maximum between two elements, the second being their sum and the third being a given length increased by one.

In addition, the description for the problems from Exercism in their website is as follows:

(i) `binary-search-tree`. It inserts and searches for numbers in a binary search tree.
(ii) `grade-school`. It creates a roster for the school given students' names and the grade they are in.
(iii) `luhn`. It determines whether or not a given number is valid per the Luhn formula.
(iv) `raindrops`. It converts a number to a string depending on the number's factors.
(v) `resistor-color-duo`. It converts color codes, as used on resistors, to a numeric value.
(vi) `robot-simulator`. It writes a robot simulator.
(vii) `roman-numerals`. It converts natural numbers to Roman Numerals.
(viii) `simple-linked-list`. It implements a singly linked list.
(ix) `space-age`. It calculates how old someone is in terms of a given planet's solar years.
(x) `sum-of-multiples`. It finds the sum of all the multiples of a particular number up to, but not including that number itself.
(xi) `triangle`. Given three sides lengths, it determines if they can form an equilateral, isosceles or scalene triangle, or if they can not be a triangle at all.

Moreover, Table IV displays the programs' length in the test suite. This information works alongside the EXAM score results to offer a more precise dimension of the effort level needed by a programmer while locating a bug using *HaskellFL* tool. The test suite's mean program length is 30.4 lines, and the median program length is 27.5 lines.

The test cases were manually chosen and written and ranged between 3 and 12 test cases per problem, as detailed in Table IV. Even though time execution is not tracked, the test cases were balanced between passing and failing. This is based on [40], which states that the expense of fault localization for most formulas will increase with the increase of class imbalance. In light of that, this is a factor to be considered. Furthermore, Table IV also presents the faulty line rank for both Tarantula and Ochiai methods, considering the best case scenario among drawing statements for every program we tested.

1) Test Setup: To allow the tests to be interpreted by *HaskellFL*, some small code pieces in the test suite were rewrote. Fig. 11 shows in the `SumOfMultiples` module, an example of a change made in an original test in the test suite to allow its interpretation by *HaskellFL*. This example was extracted from the problems in the Exercism's *Haskell* track. *HaskellFL* still does not interpret the operator `$`, which is an operator indicating precedence among operations. Thus, this operator can be safe replaced by parenthesis without any loss in the program semantics, as shown in Fig. 12. The version using parenthesis is the one present in the test suite after a

[9]https://exercism.io/tracks/haskell/exercises

bug insertion.

```
module SumOfMultiples (sumOfMultiples) where

    import Data.List(nub)

    sumOfMultiples :: [Integer] -> Integer -> Integer
    sumOfMultiples [] limit = 0
    sumOfMultiples factors limit =
        sum $ distinctFactors factors limit

    distinctFactors :: (Integral a) => [a] -> a -> [a]
    distinctFactors [] limit = []
    distinctFactors (x:xs) limit =
        nub $
        (distinctFactors xs limit) ++
        (appendFactor x limit 1)

    appendFactor :: (Integral a) => a -> a -> a -> [a]
    appendFactor factor limit index
        | factor * index >= limit = []
        | factor == 0 = []
        | otherwise = (factor * index) :
            appendFactor factor limit (index + 1)
```

Listing 11 `SumOfMultiples` module as available on GitHub

```
module SumOfMultiples (sumOfMultiples) where

    filter :: (a -> Bool) -> [a] -> [a]
    filter _ [] = []
    filter f (x:xs)
      | f x       = x : (filter f xs)
      | otherwise = filter f xs

    nub :: (Eq a) => [a] -> [a]
    nub [] = []
    nub (x:xs) = x : nub (filter (\y -> y /= x) xs)

    sum :: [Int] -> Int
    sum [] = 0
    sum (x:xs) = x + sum xs

    appendFactor :: (Integral a) => a -> a -> a -> [a]
    appendFactor factor limit index
        | (factor * index) >= limit = []
        | factor == 0 = []
        | otherwise = (factor * index) :
            appendFactor factor limit (index + 1)

    distinctFactors :: (Integral a) => [a] -> a -> [a]
    distinctFactors [] limit = []
    distinctFactors (x:xs) limit =
        nub ((distinctFactors xs limit) ++ (appendFactor x limit 1))

    sumOfMultiples :: [Integer] -> Integer -> Integer
    sumOfMultiples [] limit = 0
    sumOfMultiples factors limit = sum (distinctFactors factors limit)
```

Listing 12 `SumOfMultiples` module equivalent to the module in Fig. 11

It is important to say that the Haskell `Prelude` module is absent from *HaskellFL*. `Prelude` is a standard Haskell module that is generally imported by default into all Haskell modules. It implements and exports several basic functions. This absence leads to the need to create `Prelude` functions when using *HaskellFL*, as may be seen in Fig. 12, with the functions `filter` and `sum`. They were imported from Haskell `Prelude` in the first `SumOfMultiples` module in Fig. 11. Similarly, `nub` function was imported from `Data.List` module in the original version of `SumOfMultiples` and it was implemented in the version used in *HaskellFL*. Several `Prelude` functions can be implemented with the constructs offered by *HaskellFL*, so this absence does not prevent the tool from being used. Additionally, the creation of a similar standard module for *HaskellFL* should not take much extra effort. Furthermore, the bugs inserted into the programs in the test suite followed the techniques mentioned in Section II-B. For instance, for the `SumOfMultiples` module shown in Fig. 12, a bug was inserted into Line `19`, in the `appendFactor` function. The operator >= was replaced by the operator >.

Finally, the passing and failing test cases were wrote for every problem present in the test suite. To illustrate the process, the passing test cases for the `sumOfMultiples` module may be seen in Fig. 13 and the failing test cases for the same module in Fig. 14. As previously mentioned, the tests were chosen trying to keep them balanced between passing and failing test cases, to cover every branch in the code.

```
sumOfMultiples [4,5] 6
sumOfMultiples [] 5
sumOfMultiples [2,5] 3
sumOfMultiples [0,2,5] 3
```

Listing 13 Passing `sumOfMultiples` test cases

```
sumOfMultiples [2,5] 2
sumOfMultiples [0,2,4] 2
sumOfMultiples [2,4] 4
```

Listing 14 Failing `sumOfMultiples` test cases

In conclusion, for `sumOfMultiples` function, the Ochiai method ranked the buggy line first, tied with several other lines, and the Tarantula method ranked it in the third position, also in a tie with other lines.

B. Results

Table V shows the EXAM score distribution for the test suite, considering best and worst-case scenarios, for Tarantula and Ochiai methods. It shows which percentage of the suite's programs is inside a specific EXAM score segment. The table is divided into segments of 5%. To demonstrate, considering the Tarantula formula and the best-case scenario, 50% of the programs secured an EXAM score between 0% and 4.9%. To put it more simply, for 50% of the programs in the suite in the best-case scenario, a student would have to examine less than 5% of his Haskell code to find the buggy line in his assignment.

TABLE V
EXAM SCORE FOR HASKELL TEST SUITE

EXAM Score	Tarantula Best	Tarantula Worst	Ochiai Best	Ochiai Worst
(0-4.9)%	58.33%	33.33%	66.67%	33.33%
(5-9.9)%	25.00%	20.83%	16.67%	20.83%
(10-14.9)%	8.33%	12.50%	12.50%	16.67%
(15-19.9)%	4.17%	8.33%	4.17%	16.67%
(20-24.9)%	0.00%	8.33%	0.00%	4.17%
(25-29.9)%	0.00%	4.17%	0.00%	0.00%
(30-34.9)%	0.00%	0.00%	0.00%	0.00%
(35-39.9)%	0.00%	0.00%	0.00%	4.17%
(40-44.9)%	4.17%	8.33%	0.00%	0.00%
(45-49.9)%	0.00%	0.00%	0.00%	0.00%
(50-54.9)%	0.00%	0.00%	0.00%	4.17%
(55-59.9)%	0.00%	4.17%	0.00%	0.00%

Fig. 9 illustrates Tarantula and Ochiai methods' effectiveness for best and worst-case scenarios. The graph is built in a way that for a given `x` value, its corresponding `y` value is the cumulative percentage of the faulty versions whose EXAM score is less than or equal to `x`, similarly to what [47] did. Table V displays the scores used to build the graph. To better exemplify, in the Ochai method in the best-case scenario, *HaskellFL* was able to find all the incorrect statements in the test suite examining less than 20% of the source code for each problem. Furthermore, Table VI presents the mean, median, and standard deviation of the calculated EXAM scores for the test suite. The highest median value is 8.7% of the source code for the Tarantula method in the worst-case scenario. The smaller median value appeared for the Ochiai method in the best-case scenario with 3.7% of the source code, and its standard deviation is equal to 4.0%, which indicates that the results are close to the median value.

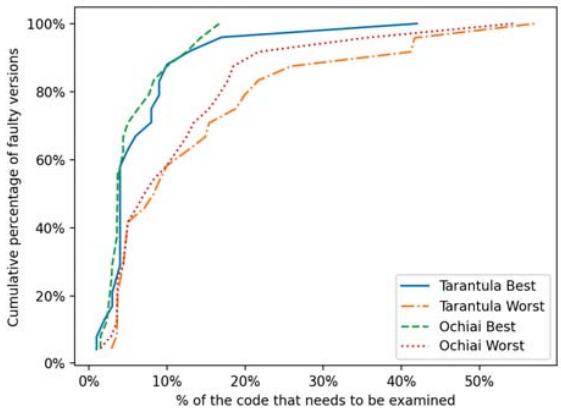

Fig. 9 Comparison between Ochiai and Tarantula methods for the test suite

TABLE VI
MEAN, MEDIAN AND STANDARD DEVIATION OF THE TEST SUITE

	Mean	Median	Standard Deviation
Tarantula Best	7.2%	4.0%	8.1%
Tarantula Worst	14.4%	8.7%	14.0%
Ochiai Best	5.5%	3.7%	4.0%
Ochiai Worst	12.0%	7.7%	11.7%

VII. Threats to Validity

This section discusses the main threats to the validity of this work and the strategies to mitigate them.

As previously mentioned, the test suite programs did not have bugs that need to be found, so the bugs were introduced into the code. To mitigate this threat, the mutants' guidelines shown in Section II-B were followed. The guidelines assume that most mutants form realistic faults, even if they are artificially seeded. Therefore, the inserted bugs represent mistakes that real students make.

Another threat is related to the passing and failing test cases used as input for the studied fault localization techniques; they were wrote by the researches conducting this work. To mitigate this threat, the number of tests between passing and failing test cases were balanced. Also, the researchers carefully and manually worked on finding test cases to cover every branch in the code, whenever this was possible, keeping the odds as fair and unbiased as possible.

Additionally, one of the goals in this work is to aid beginning students with the functional paradigm, so *HaskellFL* was tested with smaller and simpler Haskell code. With that said, there are no guarantees that the results found in this work will be applicable for larger and more complex programs.

Finally, this project used Tarantula and Ochiai fault localization techniques. There are several different techniques in the literature, for instance, Barinel [2], Op2 [32] and DStar [48]. Nevertheless, [21] conducted a study on the Siemens suite which showed that Tarantula is a more effective fault localization technique when compared to others such as set union, set intersection, nearest neighbor, and cause transition. Hence, it is a great and recognized baseline for testing. Furthermore, the Ochiai similarity coefficient-based technique is generally considered more effective than Tarantula; hence it is a great choice to measure the effectiveness of fault localization techniques in the Haskell context.

VIII. Conclusion

This paper presented *HaskellFL*, a command-line tool to locate logical errors in Haskell using spectrum-based fault localization techniques.

The results, described in Section VI, showed that *HaskellFL* located the errors for both studied methods, having to examine very few lines for the majority of the test suite. Also, Ochiai presented better results than Tarantula in terms of the EXAM Score.

A test suite suitable to the Haskell subset used in this project was also compiled. This Haskell subset is diverse and contains abstract data types that were not supported on Singer and Archibald [42]. The test suite is available together with *HaskellFL* as an open-source project.

An interpreter that supports the grammar displayed in Appendix A and Haskell's layout rules was built. Haskell's layout rules were also mentioned as a point of confusion for students in Singer and Archibald [42].

Finally, *HaskellFL* was carefully designed to allow its future extension. Potential areas for future work are:

(i) Extend the grammar to include `do` notation and list comprehensions. Two constructs that novices to Haskell may find confusing.
(ii) Implement additional spectrum-based fault localization techniques.
(iii) Implement missing code-oriented fault localization techniques.
(iv) Share *HaskellFL* with Haskell beginners and measure how much the tool is able to aid in real time.
(v) Implement techniques to repair the code.

APPENDIX A
HASKELL'S GRAMMAR SUBSET

⟨*program*⟩ ::= 'module' X 'where' '{' decls '}'
| decls

⟨*decls*⟩ ::= decl decls | d decls | decl | d

⟨*decl*⟩ ::= var '=' e 'where' '{' p_1 '=' e_1 ... p_k '=' e_k '}'
| var '=' e
| var '=' p 'where' '{' p_1 '=' e_1 ... p_k '=' e_k '}'
| var '=' p
| var '|' e_i '=' e_j ... '|' $e(i+k)$ '=' $e(j+k)$
'where' '{' p_1 '=' e_1 ... p_k '=' e_k '}'
| var '|' e_i '=' e_j ... '|' $e(i+k)$ '=' $e(j+k)$

⟨*d*⟩ ::= 'data' X '=' $C_1\ \tau_1$ ... τ_k | ... | $C_n\ \tau_1$ ... τ_k
| 'newtype' X '=' C τ

⟨*e*⟩ ::= τ | λ p '->' e | e_1 '+' e_2 | e_1 '−' e_2 | e_1 '⋆' e_2
| e_1 '\' e_2 | e_1 '^' e_2
| '['e_1']' '++' '['e_2']' | e_1 ':' '['e_1']' | e_1 '||'
e_2 | e_1 '&&' e_2 | e_1 '>' e_2
| e_1 '<' e_2 | e_1 '<=' e_2 | e_1 '>=' e_2 | e_1 '==' e_2
| e_1 '\=' e_2 | e_1 e_2
| '('e_1',' ... ',' e_k')' | '['e_1',' ... ',' e_k']' |
'if' e_1 'then' e_2 'else' e_3
| 'case' e 'of' '{' p_1 '->' e_1 ... p_k '->' e_k '}' | 'let' '{'
p_1 '=' e_1 ... p_k '=' e_k '}' 'in' e

⟨*p*⟩ ::= const | var | C p_1 ... p_k | _

⟨τ⟩ ::= Integer | Boolean | String | Char | Float | [τ] |
(τ1, ..., τ_k) | C
τ_1 ... τ_k

REFERENCES

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

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[3] Thomas Ball, Mayur Naik, and Sriram K Rajamani. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN Notices*, volume 38, pages 97–105. ACM, 2003.

[4] Brett A Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148–175, 2016.

[5] Richard Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2014.

[6] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

[7] Arthur Charguéraud. Improving type error messages in ocaml. *Electronic Proceedings in Theoretical Computer Science*, 198:80–97, Dec 2015.

[8] Ophelia C Chesley, Xiaoxia Ren, Barbara G Ryder, and Frank Tip. Crisp–a fault localization tool for java programs. In *29th International Conference on Software Engineering (ICSE'07)*, pages 775–779. IEEE, 2007.

[9] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *European conference on object-oriented programming*, pages 528–550. Springer, 2005.

[10] Chris Done. Try haskell, November 2018. Online; accessed 21-November-2018.

[11] Joao Fernandes. Generalized lr parsing in haskell. *Informal Proceedings of the Summer School on Advanced Functional Programming, students' presentation*, pages 24–37, 2004.

[12] F# Software Foundation. About f#, January 2021. Online; accessed 06-January-2021.

[13] Kotlin Foundation. Learn kotlin, January 2021. Online; accessed 06-January-2021.

[14] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100, 2017.

[15] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200. IEEE, 2014.

[16] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, 2006.

[17] Martin Handley and Graham Hutton. Improving haskell. In Michal H. Palka and Magnus O. Myreen, editors, *Trends in Functional Programming - 19th International Symposium, TFP 2018*, volume 11457 of *Lecture Notes in Computer Science*, pages 114–135, Gothenburg, Sweden, June 11-13, 2018. Springer.

[18] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings HW03: Haskell Workshop 2003*, pages 62–71, Co-Located with ICFP 2003 and PPDP 2003 Conferences) Uppsala, Sweden August, 2003. ACM.

[19] Bastiaan J Heeren. *Top quality type error messages*. Utrecht University, 2005.

[20] Tim AD Henderson. How to evaluate statistical fault localization. 2018.

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

[22] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 467–477. IEEE, 2002.

[23] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[24] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[25] René Just, Darioush Jalali, and Michael 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*, pages 437–440, 2014.

[26] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *Lecture Notes in Computer Science, 9207, 217-233. Presented at: 27th International Conference on Computer-Aided Verificationn*, pages 217–233, San Francisco, CA, USA, July 18-24, 2015. Springer.

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

[28] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mutation testing of functional programming languages. *Oregon State University, Tech. Rep*, 2014.

[29] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. Automatic diagnosis and correction of logical errors for functional programming assignments. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):158, 2018.

[30] Zijie Li, Long Zhang, Zhenyu Zhang, and Bo Jiang. Mcfl: Improving fault localization by differentiating missing code and other faults. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 943–952. IEEE, 2020.

[31] Hai Liu, Neal Glew, Leaf Petersen, and Todd A Anderson. The intel labs haskell research compiler. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.

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

[33] Redação Nubank. O que é clojure? o que isso tem a ver com o nubank?, January 2021. Online; accessed 06-January-2021.

[34] OCaml. Ocaml, November 2018. Online; accessed 21-November-2018.

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

[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating & improving fault localization techniques. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03*, 2016.

[37] Simon L Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[38] Iustin Pop. Experience report: Haskell as a reagent. In *ACM SIGPLAN International Conference on Funtional Programming*. Citeseer, 2010.

[39] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[40] Peifeng Rao, Zheng Zheng, Tsong Yueh Chen, Nan Wang, and Kaiyuan Cai. Impacts of test suite's class imbalance on spectrum-based fault localization techniques. In *2013 13th International Conference on Quality Software*, pages 260–267. IEEE, 2013.

[41] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–30, 2020.

[42] Jeremy Singer and Blair Archibald. Functional baby talk: Analysis of code fragments from novice haskell programmers. *arXiv preprint arXiv:1805.05126*, 2018.

[43] George Thompson and Allison K Sullivan. Profl: a fault localization framework for prolog. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 561–564, 2020.

[44] S. Thompson. *Haskell: The Craft of Functional Programming*. Icss Series. Addison Wesley, 2011.

[45] Anish Tondwalkar. *Finding and Fixing Bugs in Liquid Haskell*. PhD thesis, University of Virginia, 2016.

[46] Haskell Wiki. The haskell programming language, November 2018. Online; accessed 21-November-2018.

[47] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42–51. IEEE, 2008.

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

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

[50] Danfeng Zhang, Andrew C Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing haskell type errors. Technical report, Technical Report http://hdl. handle. net/1813/39907, Cornell University, 2015.