DETECÇÃO DE ERROS LÓGICOS EM *HASKELL*

VANESSA CRISTINY RODRIGUES VASCONCELOS

DETECÇÃO DE ERROS LÓGICOS EM ***HASKELL***

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha

Belo Horizonte

Fevereiro de 2021

VANESSA CRISTINY RODRIGUES VASCONCELOS

DETECTING LOGICAL ERRORS IN *HASKELL*

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Mariza Andrade da Silva Bigonha

Belo Horizonte

February 2021

Vasconcelos, Vanessa Cristiny Rodrigues.

V331d Detecção de erros lógicos em Haskell [manuscrito] / Vanessa Cristiny Rodrigues Vasconcelos. – 2021.
xiv, 57 f. il.

Orientadora: Mariza Andrade da Silva Bigonha.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.50-54
.
1. Computação – Teses. 2. Depuração na computação – Teses. 3. Haskell (Linguagem de programação de computador) – Teses. 4. Programação funcional (Computação) – Teses. I. Bigonha, Mariza Andrade da Silva. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Detecting Logical Errors In Haskell

VANESSA CRISTINY RODRIGUES VASCONCELOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
Profa. Mariza Andrade da Silva Bigonha - Orientadora
Departamento de Ciência da Computação - UFMG

[illegible]
Prof. Leonardo Vieira dos Santos Reis
Departamento de Ciência da Computação - UFJF

[illegible]
Prof. Vladimir Oliveira Di Iorio
Departamento de Informática - UFV

[illegible]
Prof. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 19 de Fevereiro de 2021.

Acknowledgments

I would like to thank the following people, without whom I would not have been able to complete this research, and without whom I would not have made it through my masters degree!

To my family, especially to my mother Meire and my brother Pedro who always and without a doubt support me in the projects I set as goals.

To my advisor Mariza Bigonha for always helping, advising and encouraging me all the time.

To my friends who gracefully dealt with my absences and cheered me every step in these years.

To the Graduate Program in Computer Science (PPGCC) and CNPQ for providing the means for me to achieve this goal.

To the dissertation committee, prof. Leonardo Vieira dos Santos Reis, prof. Marco Túlio Oliveira Valente and prof. Vladimir Oliveira Di Iorio for their participation at this moment.

"The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do."
(Andrew Singer)

Resumo

Compreender e utilizar o paradigma funcional é um desafio para diversos programadores. Procurar por erros lógicos em um código pode tomar muito tempo de um desenvolvedor quando o programa aumenta de tamanho.

Para facilitar ambos processos, projetamos e implementamos uma ferramenta chamada *HaskellFL*, que a partir de um código contendo um erro lógico em *Haskell* e alguns casos de teste, utiliza técnicas de localização de erro para calcular uma pontuação estatística para cada expressão executada durante a chamada de uma função. Esta pontuação representa o grau de probabilidade de cada expressão estar provocando o comportamento inesperado do programa.

O subconjunto de *Haskell* utilizado neste projeto é suficientemente expressivo para quem está estudando Programação Funcional conseguir auxílio imediato na depuração do seu código, e assim sanar dúvidas a respeito de conceitos-chave associados ao paradigma funcional. Adicionalmente, este subconjunto pode ser facilmente estendido para englobar toda a gramática de *Haskell* 2010.

Avaliamos a eficácia de duas técnicas de localização de falhas na literatura, Tarantula e Ochiai, no contexto de programas *Haskell*. Nossos resultados mostraram que o método *Ochiai* foi mais efetivo que o método *Tarantula*.

Nós testamos *HaskellFL* com trabalhos dos estudantes de Programação Funcional de uma turma da Universidade Federal de Minas Gerais, em conjunto com exercícios da trilha de *Haskell* do site *Exercism*, que foram resolvidos e estão disponíveis publicamente no GitHub.

Além disso, utilizamos a métrica *EXAM* para avaliar a eficácia da nossa ferramenta, e nossos resultados mostraram que *HaskellFL* ajudou a diminuir o esforço de um programador ao procurar por um erro em um código *Haskell* em todos os cenários testados.

Palavras-chave: *Debug*, *Haskell*, Localização de Erro, Programação Funcional.

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, we designed and implemented a tool called *HaskellFL*, which from code containing a logical error in *Haskell* and some test cases, uses a couple of fault localization techniques to calculate a statistical score for each expression executed during a function call. This score represents the degree of probability that each expression is causing the unexpected behavior of the program.

The *Haskell*'s subset used in this project is sufficiently expressive for those who are studying Functional Programming to get immediate help debugging their code, and thus answer questions about key concepts associated to the functional paradigm. In addition, this subset can be easily extended to encompass the entire *Haskell* 2010 grammar.

We evaluate the effectiveness of two fault localization techniques in the literature, Tarantula and Ochiai, in the context of *Haskell* programs. Our results showed that Ochiai method was more effective than Tarantula.

We tested *HaskellFL* against Functional Programming assignments submitted by students enrolled at the Functional Programming class at Federal University of Minas Gerais and against exercises from the Exercism *Haskell* track that are publicly available on GitHub.

Furthermore, we used the EXAM score to evaluate our tool effectiveness, and our results showed that *HaskellFL* reduced the effort needed to locate an error for all tested scenarios.

Palavras-chave: Debug, Fault Localization, Functional Programming, *Haskell*.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

According to Mitchell et al. [2003], the functional paradigm refers to programming languages in which we do most of the computation by evaluating expressions that contain functions. Another definition found in Bird [2014] is that functional programming is a method of program construction that emphasizes functions and their application rather than commands and execution. To sum up, functional programming takes advantage of mathematics functions, organizing them to write a program without side effects.

At first sight, the functional paradigm may confuse programmers; this may happen because they usually start by learning the imperative paradigm, which has no particular way of handling state. In light of that, several difficulties may appear when programmers try to learn a new way to write code with different reasoning. If these issues are not addressed early, developers may use the functional language like they are using an imperative one for a long time. Consequently, they will take no real advantage of the functional paradigm.

For instance, Figure 1.1(b) exhibits a simple example of functional programming, a function named `fact` that implements a function to calculate the factorial of a given `n`. We wrote this piece of code in *Haskell*, and it is as straightforward as the mathematical factorial definition. In Figure 1.1(a) we may see an implementation of `fact` in *Java*, which contrasting with the *Haskell* definition, 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 [F#, 2021]. *OCaml* stands for Objective-ML and it "is an industrial-strength programming language supporting functional, imper-

```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)

Figure 1.1. Factorial function in *Java* (a) and in *Haskell* (b).

ative and object-oriented styles" [OCaml, 2018]. *Clojure* is used by *Nubank* [Nubank, 2021]. *Haskell* is an excellent choice for a functional language because it has a large and active community. It has built-in concurrency and parallelism, and it supports integration with other languages [HaskellWiki, 2018a].

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* [Kotlin, 2021]. The latter allows us to classify *Kotlin* as very promising 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, we may conclude that 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 a good first functional language because it allows developers to have a clear view of functional concepts.

1.1 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 catching some simpler bugs. For example, Becker et al. [2016] conducted a study on 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, Singer and Archibald [2018] 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 errors a compiler identifies automatically. Becker et al. [2016] also enhanced error messages in their study to test how much an improved message may help. Their results indicate that it does help. Nonetheless, better messages do not extinguish the errors, and we have not yet considered logical errors. These are the errors a compiler can not automatically catch, and for that reason, they are even harder to identify.

In light of that, we projected and implemented the *HaskellFL* tool, which locates logical errors in functional programming assignments written in *Haskell*. Considering a source code with unexpected behavior and a few test cases, some of which outputting an unexpected result, while others producing the expected output, we calculate and return a list with the most likely expressions to be triggering this unpredictable behavior. After that, we ordered the list from most to less probable root cause. This suspiciousness list is created using fault localization techniques, thus, we also evaluated the effectiveness of two different techniques for the *Haskell* context.

The main reason for choosing functional programming for this project is that 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. We may also take advantage of *Haskell* laziness because this provides better visibility of the code execution.

Other data serving as motivation is the work in Purushothaman and Perry [2005] cited by Gopinath et al. [2014] 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 in Gopinath et al. [2014] 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, found in Pop [2010]. *Intel* has developed a *Haskell* compiler as part of their research on multicore parallelism at scale, which appears in Liu et al. [2013], *Microsoft* uses it in its compilers research, and Tesla also uses it in its internal products. A list containing several companies and the respective field they use *Haskell*, may be found in Haskell Cosmos website[1], inclusive some of the companies mentioned above are listed there.

1.2 Objectives

This project's primary goal is to evaluate the effectiveness of two fault localization techniques in the literature, Tarantula [Jones and Harrold, 2005] and Ochiai [Abreu et al., 2009a], 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.

To achieve our goals, we established the following milestones:

[1] `https://haskellcosm.com/`

(i) build an interpreter for a subset of the *Haskell* 2010 grammar
(ii) implement Tarantula and Ochiai fault localization techniques
(iii) create a test suite covering the *Haskell* 2010 grammar's subset chosen in (i)
(iv) implement a tool which locate logical errors in *Haskell* code, and
(v) evaluate the tool against the test suite.

1.3 Contributions

As the main results of this research project conducted by the author of this master thesis, the following contributions stand out:

(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 our test suite using the EXAM score (see Section 2.4.4).
(v) A *Haskell* interpreter for a subset of *Haskell* 2010 grammar.

1.4 Publication

An article entitled, Detecting Logical Errors in *Haskell*, accepted to be published in the proceedings of the 15th International Conference on Testing Software and Systems (ICTSS 2021).

Additionally, the *HaskellFL* tool and the test suite compiled for this work are publicly available on GitHub[2].

1.5 Dissertation Roadmap

In this chapter, we presented the problem, the objective, and the dissertation's contributions. The organization of the remaining chapters of this master's thesis is as follows. Chapter 2 details the literature overview. It presents the theoretical foundations indispensable for making this work, such as *Haskell*'s and lambda calculus' fundamental concepts, plus an overview on the fault localization topic, enumerating the techniques we use in *HaskellFL* and showing how to evaluate these techniques.

[2]`https://github.com/VanessaCristiny/HaskellFL`

Chapter 3 discusses the related work and the remaining gaps that motivated the present work. Chapter 4 presents the proposed solution to the identified problem, discussing the requirements and implementation of the *HaskellFL* tool for detection of logical errors in *Haskell*. Chapter 5 presents our test suite and the results we obtained while testing *HaskellFL* against it.

Chapter 6 concludes this dissertation, presenting our contribution and suggestions for future works. Following it, we present the bibliography and the appendices. Appendix A exhibits the subset of *Haskell* 2010 grammar supported by *HaskellFL*, while Appendix B displays two programs that are present in our test suite.

Chapter 2

Literature Overview

This chapter approaches topics that are central to this dissertation. We start with a short reminder of *Haskell*'s constructs in Section 2.1, following a brief lambda calculus concepts' revision in Section 2.2 because we relied heavily on these concepts while building our tool. Section 2.3 goes trough `SKI` combinators.

Additionally, in Section 2.4 we present an overview of error localization techniques and explain how to evaluate them. In Subsection 2.4.3 we discuss the errors originated by missing code. Section 2.5 closes this chapter.

2.1 *Haskell* Basics

Haskell was chosen for this project because it is a purely functional language that allows us to understand functional paradigm's concepts. It is also the language used in many Functional Programming courses in Brazilian universities.

GHC compiler (see Section 3.1) supports the entire *Haskell* 2010 language and a wide variety of extensions. Nonetheless, this project works with a subset of the *Haskell* 2010 language, shown in full in Appendix A, which we consider is enough to absorb critical concepts in functional programming. This section highlights these concepts.

2.1.1 Data Types

Algebraic data types are classified into: sum types and product types. Sum type happens when a type may be an instance of different constructors; for example, in Figure 2.1, `Shape` is either a `Circle` or a `Rectangle`, and this configures `Shape` as sum type. On the other hand, `Circle` and `Rectangle` constructors are samples of the product type because `Circle` is a combination of three float values and `Rectangle` is a combination of four. As seen, *Haskell* offers both flavors, which is very advantageous when coding.

```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Figure 2.1. Data type in *Haskell*.

2.1.2 Pattern matching

```haskell
data List a = NIL | CONS a (List a)

head :: List a -> a
head (CONS x xs) = x

tail :: List a -> a
tail (CONS x xs) = xs

absolute :: (Ord a, Num a) => a -> a
absolute n
  | n < 0 = -n
  | otherwise = n
```

Figure 2.2. Pattern matching in *Haskell*.

Pattern matching allows us to use algebraic data types in our code effectively. With pattern matching, we may separate the types by constructors and treat them accordingly. In Figure 2.2, we see the data type `List` and two essential functions - `head` and `tail` -, which yields the first element of a list and the list without the first element, respectively. As seen in their implementations exhibited in Figure 2.2, they may achieve that easily and efficiently by taking advantage of the data type structure.

Another form of pattern matching comes in form of guards, also shown in Figure 2.2 in the function `absolute`, which retrieves the absolute value of a given `n`. An expression in a branch with a guard is only evaluated and executed if the predicate yields `True`.

2.1.3 Laziness

Haskell is lazy. Thompson [2011] explains that to evaluate a function `f` with arguments $a_1, ..., a_k$ in *Haskell*, the first step is to substitute the expressions a_i for the correspondent variables in the definition of the function. After substitution, the function will only be evaluated if its result is needed. Figure 2.3 exhibits an example of lazy evaluation. In case `n` is greater than `0`, `y` will never be evaluated; otherwise, `x` will not; this is possible because a *Haskell* program is equivalent to a large directed graph.

```
switch :: Integer -> a -> a -> a
switch n x y
    | n > 0 = x
    | otherwise = y
```

Figure 2.3. Lazy evaluation in *Haskell*.

Another important note on lazy evaluation is that expressions are evaluated at most once, and this is also achievable because they are a graph and not specifically a tree. Figure 2.4 exemplifies the difference; if we had the expression $(5 * 3) + (5 * 3)$ in a tree, the expression `+` would point to two different nodes and calculate them individually, even though they are the same expression as showed in Figure 2.4 (a). In a graph, exhibited in (b) in the same figure, we can point both factors of expression `+` to the same node.

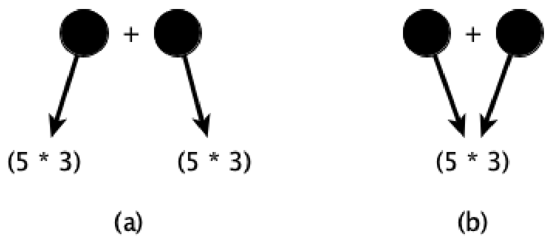


Figure 2.4. Short evaluation graph.

2.2 Lambda Calculus

This project relies heavily on the algorithms shown in Peyton Jones [1987] on compiling functional languages. The first step in the process is to transform the source code into lambda calculus expressions. This section refreshes vital concepts to this matter.

Lambda calculus is a conjunction of three terms: a variable, a lambda abstraction, and an application. These three and their recursive combinations form several expressions.

To use lambda calculus in real-world *Haskell* programs, we need to add literals as integers and booleans and `case` expressions to cite a couple of examples. These concepts compose the enriched lambda calculus and are also present in Peyton Jones [1987].

Additionally, the recursion case is fascinating because the idea of recursion is intrinsically related to naming a function, for then being able to call it from itself. However, if using Curry's Y combinator, also known as fixed-point combinator, we may have the function as an argument to a new function that is no longer recursive. Figure 2.5 shows the Y equation, and its proof of validity may be found in many references, inclusive in Peyton Jones [1987].

```
Y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

Figure 2.5. Y combinator in *Haskell* notation.

2.3 SKI combinators

Another key concept we used in *HaskellFL* is the `SKI` combinators, also cited in Peyton Jones [1987]. `S, K, I` are supercombinators - functions with no free variables - that allow us to interpret code. We are able to reduce all expressions to a combination of these. Figure 2.6 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, we may notice in Figure 2.6 that if `x` is constant regards `y`, we may disregard `y` and keep `x`. 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
```

Figure 2.6. `SKI` combinators in *Haskell* notation.

To better illustrate the SKI compilation algorithm, let us observe the following example. Given a function `\x -> x * x`, applied to the integer 10, the algorithm will follow the reduction steps in Figure 2.7. 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 through 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, we obtain the expression S (S (K *) I) I, which 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
```

Figure 2.7. Example of SKI compilation.

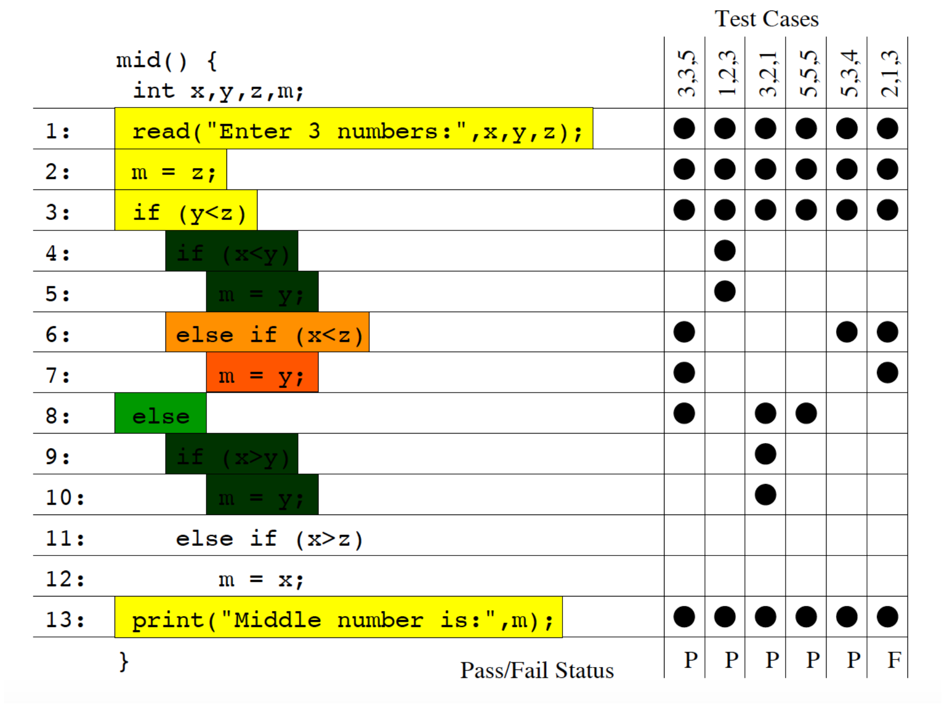
2.4 Fault Localization

Fault localization techniques aim to drive developer's attention to specific parts of the code to speed up debugging. They usually output an ordered list of suspicious program locations. There are two main fault localization approaches; they are spectrum-based fault localization and mutation-based fault localization.

2.4.1 Spectrum-Based Fault Localization

Jones et al. [2002] present a prototype tool that uses coloring statements in the code to detect fault locations, which they classify 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 produce the expected result for a given input, while the other few produce 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 2.8 depicts the example shown by Jones et al. [2002] in their paper and allows us to better demonstrate what we 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 dissertation.



	mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●	●
4:	`  if (x<y)`		●				
5:	`    m = y;`		●				
6:	`  else if (x<z)`	●				●	●
7:	`    m = y;`	●					●
8:	`else`	●		●	●		
9:	`  if (x>y)`			●			
10:	`    m = y;`			●			
11:	`  else if (x>z)`						
12:	`    m = x;`						
13:	`print("Middle number is:",m);`	●	●	●	●	●	●
	} Pass/Fail Status	P	P	P	P	P	F

Test Cases

Figure 2.8. Colored code for detecting error location, extracted from Jones et al. [2002].

Moreover, in Figure 2.8, `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 case `2, 1, 3`, execute lines `1, 2, 3, 6, 7, 13`, and fail.

Lee et al. [2018] 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, Blumer et al. [1987], meaning they want to replace an expression as small as possible.

Table 2.1 shows two formulas from different methods - Tarantula, Jones and Harrold [2005], and Ochiai, Abreu et al. [2009a] - that we used to calculate the error localization in this master thesis project; both fall in the SBFL category.

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

Table 2.1. Fault localization techniques' formulas.

According to Jones and Harrold [2005], Tarantula utilizes all the standard information other testing tools also use: 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. The 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 fault to be occasionally executed by passed test cases because they claim it often provides more effective results.

In Abreu et al. [2009a] 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 faulty code execution does not necessarily lead to failures, while failures always involve a fault.

2.4.2 Mutation-Based Fault Localization

Another approach for finding logical errors is mutation-based fault localization (MBFL) techniques. In Papadakis and Le Traon [2015], 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 made by Pearson et al. [2016], the authors evaluated different techniques for finding faults' localization. Their dataset 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 Le et al. [2014b], the authors propose a mutation testing tool for *Haskell* programs and also name 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 Le et al. [2014b] agree they are still valid for functional programs. They added the last three specifically for functional programming.

2.4.3 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. [2014] cited by Pearson et al. [2016] affirm that for 30% of cases, a bug fix consists of adding new code rather than changing existing code. Nonetheless, Wong et al. [2016] 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. [2016] 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, Li et al. [2020] 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 ours; however, newer techniques as MCFL are improvements to the field.

2.4.4 Fault Localization Metrics

As said in the previous section, 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 [2018] compiled several evaluation methods, and we reproduce some of them 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 LIL advantage framework does not depend on a list of ranked statements and may apply 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 present matching scores. The approach we took calculates the best and worst-case scenarios. We consider the best-case scenario when the developer starts examining them by the line containing the bug among the several lines with the same score. Conversely, the worst-case scenario happens when a developer chooses to examine the line with the bug last.

To exemplify, lets suppose we have a bug in Line `2` of a four lines' program and a suspiciousness score list `[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:

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

Chapter 5 will present our results in terms of the EXAM score. We chose it because we believe it is an excellent indicator of the effort level a developer needs to locate a bug, and this is the point *HaskellFL* aims to contribute. Additionally, when introducing the test suite in Section 5.2, we will also provide our problem's length, so it can work together with the EXAM score and contribute to a better understanding of our results.

2.5 Final Remarks

This chapter reviewed several literature concepts we used to build the *HaskellFL* tool. In this sense, we initially revised some essential concepts of *Haskell* language. Additionally, we went through lambda calculus notions because one step in *HaskellFL* is to transform our input into lambda calculus terms. We also explained about `SKI` combinators because transforming expressions in terms of them is a key stage in our interpreter.

Moreover, we described different types and techniques for locating faults in code, one of the central topics of this dissertation. We mentioned two different approaches, (i) spectrum-based fault localization and (ii) mutation-based fault localization, and how they work. We also presented the two techniques we implemented in *HaskellFL* that are Tarantula and Ochiai, both fitting in the spectrum-based category. Besides, we discuss how to analyze bugs caused by missing code.

Last but not least, we presented how to evaluate fault localization techniques. We showed different metrics used in the literature explaining how they work, and we also explained why we chose to use the EXAM score to measure our results. In the next chapter, we enumerate projects similar to ours, discuss, and point out what makes ours different from them.

Chapter 3

Related Work

This chapter presents works related to our *HaskellFL* tool and the process we follow to build it. Section 3.1 talks about the state-of-the-art in *Haskell* compilers. Section 3.2 enumerates tools and methods on fault diagnosis not mentioned in the previous chapter, presenting the difference of our tool regarding the existing ones with the same aim.

In addition, Section 3.3 cites several projects on type errors and how to provide better feedback to them. Section 3.4 mentions *Haskell* tutors and Section 3.5 names some selected tools on automatic program repair. Finally, Section 3.6 concludes this chapter.

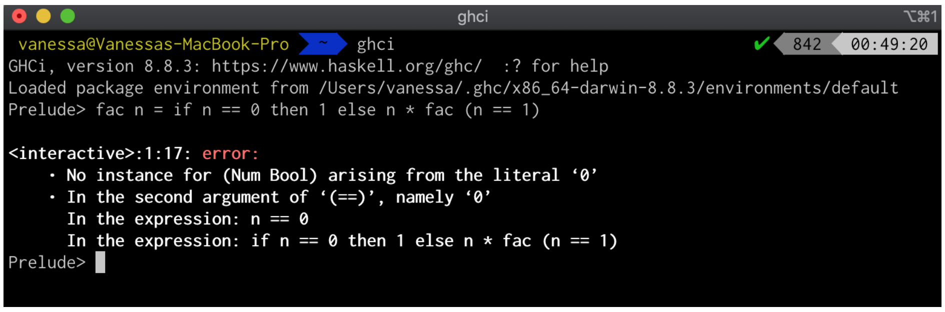
3.1 Compilers

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. We may download the *Haskell* platform on their website, HaskellWiki [2018b]. 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, which introduced the compiler, found in Jones et al. [1993], reinforces the fact the compiler is most written in *Haskell* itself, 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" by Thompson [2011] and "Thinking Functionally with *Haskell*" by Bird [2014].

GHC is an open-source project[1], it is on Version 8.10.2 to this date, and it is

[1]`https://gitlab.haskell.org/ghc/ghc/`

continually updated. It is an excellent tool for all the motives cited above. However, there is still room to improve, as we may see regarding the confusing type error feedback depicted in Figure 3.1, which we further scrutinized in Section 3.3. Another widely known *Haskell* compiler is Hugs (*Haskell* User's Gofer System), which was the compiler reference for *Haskell* prior GHC. It is no longer in development; its last release is from May 2006.



```
vanessa@Vanessas-MacBook-Pro  ~  ghci                                    ✔  842  00:49:20
GHCi, version 8.8.3: https://www.haskell.org/ghc/  :? for help
Loaded package environment from /Users/vanessa/.ghc/x86_64-darwin-8.8.3/environments/default
Prelude> fac n = if n == 0 then 1 else n * fac (n == 1)

<interactive>:1:17: error:
    • No instance for (Num Bool) arising from the literal ‘0’
    • In the second argument of ‘(==)’, namely ‘0’
      In the expression: n == 0
      In the expression: if n == 0 then 1 else n * fac (n == 1)
Prelude> 
```

Figure 3.1. Type error pointed by GHCi.

Furthermore, there is a compiler named Helium created by Heeren et al. [2003], which has educational purposes and provides more detailed feedback. For example, given the `remove` function in Figure 3.2 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 Figure 3.3. It points to an error in the second equal sign and says that we may not have another attribution after the first. 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.

```haskell
remove :: Int -> [Int] -> [Int]
remove n [] = []
remove n (x:xs)
         | n = x = rest
         | otherwise = x : rest
               where rest = remove n xs
```

Figure 3.2. `Remove` function with an error in *Haskell*.

GHCi also points to a parser error on the second equal sign with no hints 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

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

Figure 3.3. Feedback provided by Helium, extracted from Heeren et al. [2003].

is the challenging and compelling section of their work. Their solution passes by tight constraint solving and global constraint solving.

3.2 Fault Localization Tools

Section 2.4 presented some relevant works on fault diagnosis. It presented the techniques we used in *HaskellFL* and described an alternative approach to calculate fault localization that is mutation-based. This section presents other techniques and tools on error localization for several programming languages.

To begin with, Thompson and Sullivan [2020] introduced *ProFL*, a command-line fault localization tool for *Prolog* models. As happens for *HaskellFL*, *ProFL* takes a faulty *Prolog* model and 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. [2007] 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 rerun the test focusing on the modified part. The programmer may interact with changes until finding the set originating the fault.

The work in Dallmeier et al. [2005] 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 to use an object, and outgoing calls, i.e., how to implement it.

Moreover, Wong et al. [2008] presented a cross-tabulation statistical method taking advantage of code coverage for test cases. The authors used a hypothesis test to infer if execution results and each statement's coverage are dependent or independent. Each statement's suspiciousness score depends on the degree of association between its coverage and the execution results.

Le et al. [2014a] presented *MuCheck*, a mutation testing tool for *Haskell* programs.

The tool implements mutation operators that are specifically designed for functional programs (see Section 2.4.2), and makes use of *Haskell*'s type system to achieve a more relevant set of mutants.

Besides that, there are other relevant works on error localization. Jose and Majumdar [2011] present an algorithm for error cause localization based on a reduction to the MAX-SAT[2] problem; Ball et al. [2003] 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. [2006] 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.

Finally, Wong et al. [2016] compiled several Ph.D. and Master's Theses, techniques, and tools on fault localization, which makes it an excellent reference for related work. It contains the majority of the works mentioned above.

3.3 Type Errors

There are several works on compilation errors and how to provide appropriate feedback to them. To cite a few, there are Zhang et al. [2015], Charguéraud [2015], Heeren [2005], Sakkas et al. [2020] and Becker et al. [2016].

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

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

Figure 3.4. 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 Figure 3.1, 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. To solve it, it is necessary to explicitly declare the

[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.

function's type 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 Zhang et al. [2015], 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*. Figure 3.5, illustrated in their paper, is used to explain their approach to the problem.

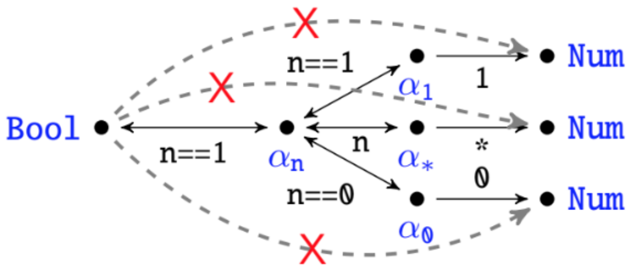


Figure 3.5. Graph for diagnosing type errors, extracted from Zhang et al. [2015].

A brief explanation: first, they model the set of constraints in the code as a constraint graph. The graph in Figure 3.5 represents the erroneous factorial code, depicted in Figure 3.4. 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 uses Bayesian principles, from probability domain, to detect which edge is the most likely error source; the correct answer is `(n == 1)`. In conclusion, it is easy to follow method that may improve type error localization and help users in a topic that traditionally causes great confusion.

3.4 Haskell Tutors

In this section, we mention researches in building systems that offer more driven feedback for tutoring functional programming apprentices, such as Heeren et al. [2003], Gerdes et al. [2017] and Handley and Hutton [2018].

The tutor created by Gerdes et al. [2017] 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. Besides that, he also needs to provide a configuration file customizing the feedback with tips that he believes would help his students to better comprehend the proposed solution and the process leading to it; these are 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 project is that their tutor is a web application, making it accessible to everybody.

They also use a compiler with improved error feedback, described in Section 3.1. Their model tracing and property-based test strategies have similarities with the strategies adopted by Lee et al. [2018]. 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, explored in Section 2.2. 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*, Done [2018], 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.

3.5 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. [2018] 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 a plain copy.

Kneuss et al. [2015] 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, Tondwalkar [2016] presents a tool to find and fix bugs in *Liquid Haskell*. As stated in Tondwalkar [2016], *Liquid Haskell* is a framework for annotating *Haskell* programs with refinement types, which are types decorated with predicates. 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.

3.6 Final Remarks

This chapter went through works that are related to any part of *HaskellFL* development. We started by citing the *Haskell* compiler GHC because it was a reference while choosing the tools to build our interpreter and handling *Haskell* layout rules.

Secondly, we enumerate works on fault localization that were not used for building *HaskellFL* directly but play a similar role for other languages that not *Haskell*. We also mentioned different methods for calculating fault location and we cited the *MuCheck* tool, provided by Le et al. [2014a], with locates bugs on *Haskell* code but following a mutant-based approach. Secondly, we enumerate works on fault localization that were not used for building *HaskellFL* directly but play a similar role for other languages that not *Haskell*. We also mentioned different methods for calculating fault location, and we cited the *MuCheck* tool, provided by Le et al. [2014a], with locates bugs on *Haskell* code but following a mutant-based approach.

Thirdly, we talked about type errors because they are traditionally a source of confusion for programmers, and they play an essential role in repairing code automatically. Furthermore, we dedicated a section to interactive tools created to tutor *Haskell* apprentices.

Our *HaskellFL* tool differs from the works in Thompson and Sullivan [2020], Chesley et al. [2007] and Dallmeier et al. [2005] in the language we support. Additionally, Thompson and Sullivan [2020] implement a mutation-based algorithm that we do not. The work in Chesley et al. [2007] is different in the sense that their work is an interactive guide for helping to locate an error root cause. Similarly, the works mentioned in Section 3.4 also serve as a guide for programming apprentices, as our work does, but they work as an interactive *Haskell* guide instead of looking for the error automatically.

Furthermore, Dallmeier et al. [2005] implements a different fault localization technique that we did not use in *HaskellFL*. Wong et al. [2008] also presented a new method which they compare and contrast against Tarantula, that is a method we studied. Le et al. [2014a] also used *Haskell* in their work, however, they implemented mutation-based fault localization while *HaskellFL* offers spectrum-based fault localization. With that said, Chapter 4 effectively presents our work.

Chapter 4

Detecting Logical Errors in *Haskell*

This chapter presents our solution to locate logical errors in *Haskell* programs. Section 4.1 presents an overview of our proposed solution and Section 4.2 explains how we built the *HaskellFL* tool from the previous mentioned solution. We detail *HaskellFL* requirements in subsection 4.2.1 and its implementation in subsection 4.2.2. To dive deeper into our implementation, subsection 4.2.2.1 uses an example to demonstrate step-by-step what *HaskellFL* does in order to locate a bug. Finally, Section 4.3 concludes this chapter.

4.1 Proposed Solution

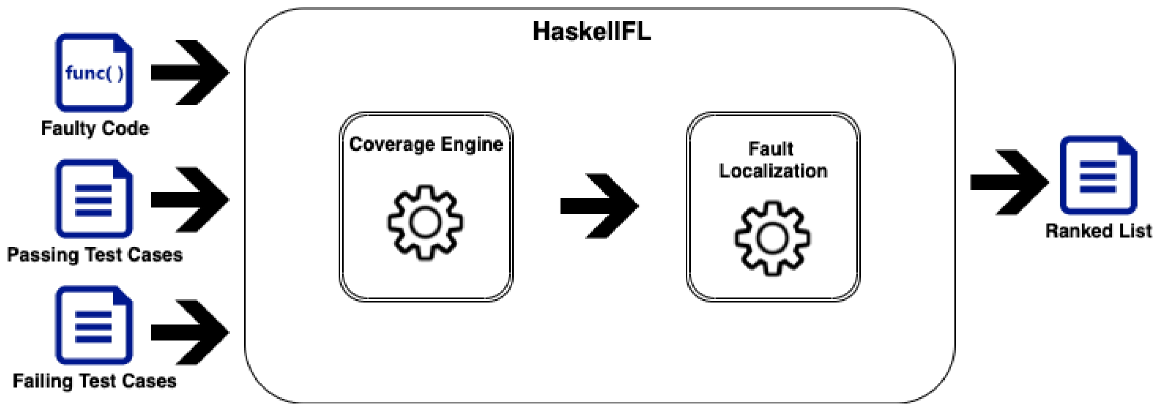


Figure 4.1. *HaskellFL* architecture.

Figure 4.1 exhibits the architecture of our proposed solution to locate logical errors in *Haskell*. We named our tool *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. We used Cabal, the standard package system for *Haskell* in *HaskellFL*. Figure 4.2 exhibits an example of *HaskellFL* execution using this package. In other words, we invoke `cabal run` indicating the target we want to execute, which is in our case *HaskellFL* followed by the args expected by the program itself, i.e., the three files enumerated above. Optionally, we may also specify the technique of choice: the Tarantula or Ochiai. If we omit this information here, we will be prompted for it later in the program.

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

Figure 4.2. *HaskellFL* execution command using Cabal.

Furthermore, *HaskellFL* can also interpret the test cases, and expose their results. To do that, we must call *HaskellFL* as in Figure 4.3, 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 run
function-name
```

Figure 4.3. Command for *HaskellFL* interpreting the test cases.

Once we have the needed inputs to run *HaskellFL*, it is necessary to obtain the code coverage for the buggy *Haskell* code regarding every test case separately. We divide the count 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 2.1, and replicated in Table 4.1 for completeness, to calculate the suspiciousness rank for each fault localization method.

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

Table 4.1. Fault localization techniques' formulas.

4.2 *HaskellFL* Tool

This section goes through *HaskellFL* details. We explain the reasoning behind our design decisions, which requirements we chose to cover and why, and what tools we used to implement *HaskellFL*. Additionally, we explain the actual process *HaskellFL* executes via a short example extracted from our test suite.

4.2.1 Requirements

Firstly, we decided the *Haskell*'s 2010 grammar subset to be contemplated by *HaskellFL*. The chosen subset is available in full in Appendix A. It is important to say that *HaskellFL* handles *Haskell*'s layout rules. The notable parts of *Haskell* 2010 we left out of our tool 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 other features which we believe are enough to support and guide *Haskell* beginners.

Secondly, we needed to calculate the fault location using one or more fault localization techniques. To do that, we calculate and make available the coverage count for each statement, making note if the generated coverage is for a passing or a failing test case. This feature's implementation allows us to insert additional fault localization methods to our tool in the future easily.

Thirdly, we chose to create our *Haskell* interpreter to obtain the code coverage map. The primary reason behind this choice is to allow the extension of *HaskellFL* in the future to repair *Haskell* code, as it is done for other programming languages as we have described in Section 3.5. Another viable extension to *HaskellFL* is to implement mutation-based fault localization techniques, also mentioned before in Section 2.4.

4.2.2 Implementation

Figure 4.4 exhibits *HaskellFL* high level block diagram. We divided the diagram into four main blocks representing four processes: Parser, Transformation, Interpreter, and Fault Localization.

The first block encapsulates the parsing process. We built the lexer using Alex[1] and the parser using Happy[2]. These are the *Haskell* equivalents for `Lex` and `YACC` respectively, and they also are the same tools used by *Haskell* compiler GHC. We

[1]`https://www.haskell.org/alex/`

[2]`https://www.haskell.org/happy/`

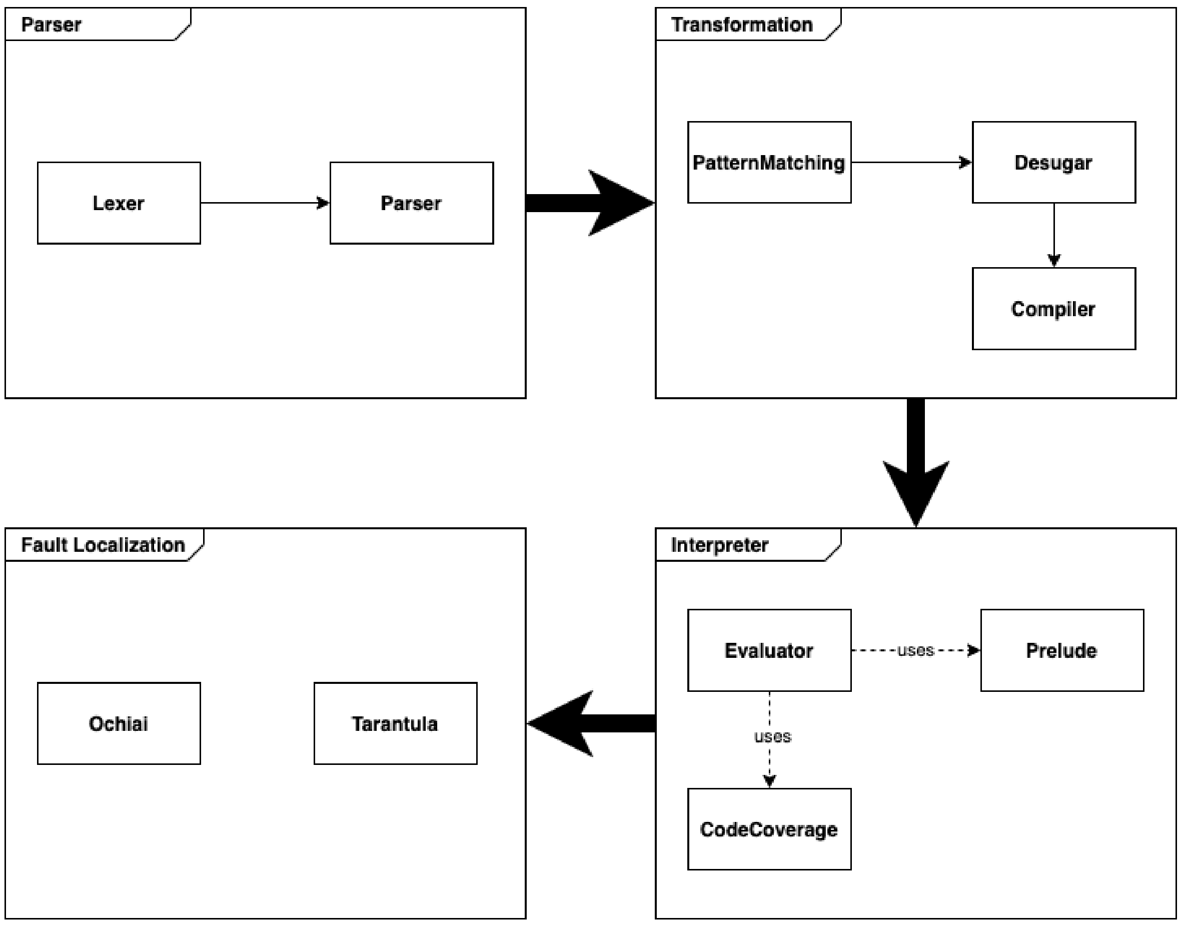


Figure 4.4. *HaskellFL* high level block diagram.

could have picked an alternative approach using one of the several libraries of parser combinators available such as `Parsec`[3]. The first option is restricted to LALR parsing, and the latter favors LL parsing, Fernandes [2004]. We chose to go with the combo Alex and Happy 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[4], which is a powerful tool with uncomplicated implementation, even though it is not suitable to *Haskell* grammar.

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

The second block is responsible for transforming the data types generated during

[3]`https://wiki.haskell.org/Parsec`
[4]`http://bnfc.digitalgrammars.com/`

the parsing process into `SKI` combinators. These data types are as close as the lambda calculus terms, mentioned in Section 2.2, 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. We obtained all the needed rules to translate a high-level functional language into lambda calculus in Peyton Jones [1987]. Additionally, before reaching the `desugar` step, we needed to handle pattern matching. We achieved that goal by implementing the `match` function, also provided in detail in Peyton Jones [1987]. Moreover, `desugar` step was also responsible for simplifying the fixed-point combinator and our other built-in functions.

`Compile` step for its turn, is the real responsible for transforming our desugared code into `SKI` combinators. This function implements the transformation rules presented in Section 2.2. With that said, we formed the second block outcome with a set containing the `SKI` combinators alongside our final literals and the new local functions.

There is the interpreter step in the following block, where the `evaluator` function orchestrates calls to the previous blocks' functions. `Evaluator` extends our small prelude with the locally declared functions. This step is composed of constant exchanges between its internal blocks, represented with dashed arrows in Figure 4.4 for organization purposes. These exchanges reflect the process of getting and adding functions to the prelude and the constant update of our execution stack.

Furthermore, in the fourth block, we were able to calculate the faulty line in the *Haskell* code using our chosen techniques, which manipulates the coverage map exposed by the third block. This step may easily include other different coverage-based fault localization methods.

An absence in *HaskellFL* tool is type checking. We did not implement a type checker for our tool for believing this would be an overkill for our 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 do not type compliant out of the search.

4.2.2.1 Walkthrough

To better understand the complete process we implemented for *HaskellFL*, let us look at the example in Figure 4.5. We extracted a small piece of `LinkedList` module that is a problem present in our test suite. This small part does not contain any bugs, but it serves to understand our 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)
```

Figure 4.5. `LinkedList` module.

Figure 4.6 shows the generated AST for `LinkedList` source code after the parsing process. We omit some details such as every terminal knowing its own position for better clarity in the figure, nonetheless, we may see that we have 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 binding and `MatchBody` keeps the function result for that respective pattern.

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

Once parsing is done, we are able to call `match` function which transforms our pattern matching function bindings in a `case` expression such as the one exposed in Figure 4.7. Colored in blue in our diagram, there are two concepts presented in Peyton Jones [1987] and introduced to our 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:

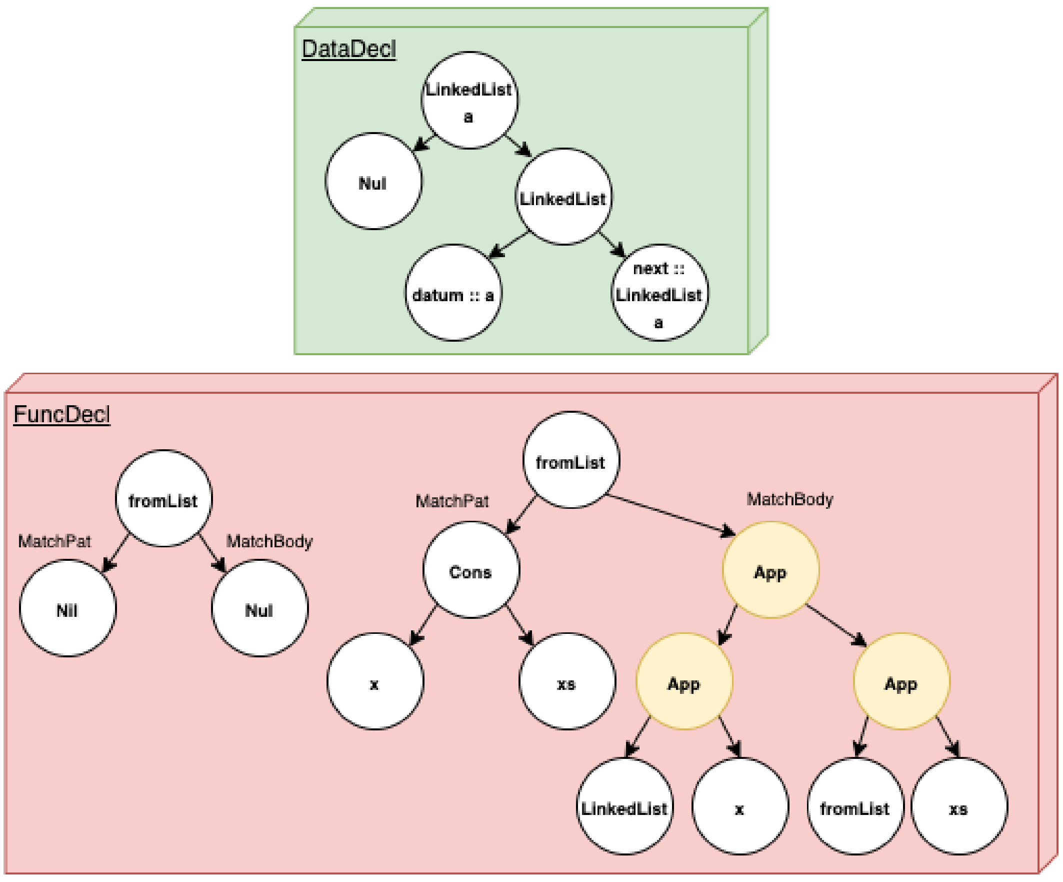


Figure 4.6. LinkedList AST.

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

In other words, if we apply the `FatBar` operator 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, we attached our resultant `case` expression to a lambda abstraction, as the body of the same, and added it to our extended prelude, after being wrapped to another layer composed of the fixed-point combinator, responsible for taking care of recursion.

In sum, after the process demonstrated in this section, we are able to call `fromList` from another function in the `LinkedList` module, or with input test cases from *HaskellFL*, such as the illustrative examples shown in Figures 4.8 and 4.9, and this way to obtain `LinkedList` code coverage.

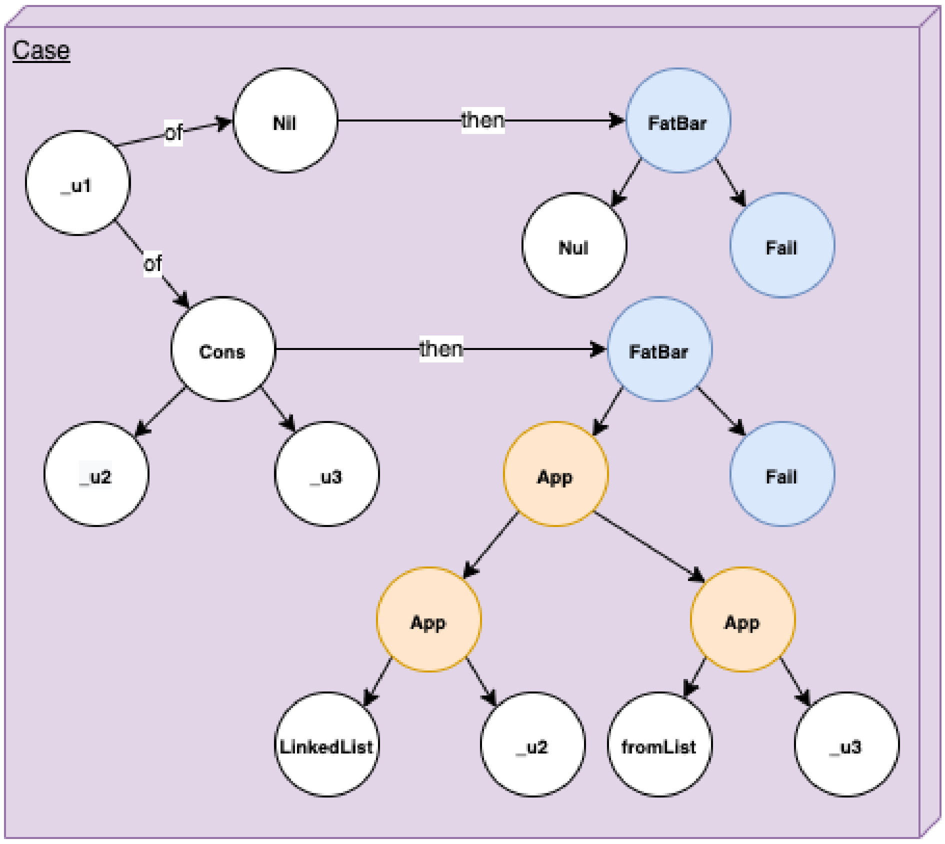


Figure 4.7. Pattern matching `case` expression.

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

Figure 4.8. `test-cases-pass.txt`

```
fromList [1, 2, 3]
fromList [1.9, 2.4, 3.8, 0]
```

Figure 4.9. `test-cases-fail.txt`

4.3 Final Remarks

This chapter showed how we implemented the *HaskellFL* tool. It started by displaying a high-level overview of our proposed solution and discussing the trade-offs of choosing which tools would better aid us in the long run. Besides, we explained how to use our tool.

Secondly, we described the steps *HaskellFL* takes to obtain code coverage with the help of an example extracted from our test suite that we will adequately introduce in Section 5.2. The tool is publicly available on GitHub[5], together with our test suite.

With that said, the next chapter will present our test suite and the quantitative results we obtained while testing *HaskellFL* against it.

[5]`https://github.com/VanessaCristiny/HaskellFL`

Chapter 5

Result Discussion

This chapter presents our test suite and the results we obtained when running *HaskellFL* against it. In Section 5.1 we introduce a case study with `mid` function in two different faulty versions, showing the coverage for the chosen test cases and the calculated scores for each statement in both methods we are studying. Section 5.2 presents the test suite we used to evaluate our work, and Section 5.3 shows our tool results tested against our suite. Section 5.4 discusses the threats to the validity of our work, and Section 5.5 concludes this chapter.

5.1 Case study

We studied two different buggy versions of `mid` function displayed in Figure 5.1. As the name suggests, `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`.

Table 5.1 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`.

Finally, we calculated the suspiciousness scores for each statement for Tarantula and Ochiai techniques according to the formulas presented in Table 4.1. We present them under their respective labels Tarantula and Ochiai.

```haskell
1  module Main where
2   mid x y z = if y < z
3     then if x < y
4           then y
5           else if x < z
6                 then x
7                 else z
8     else if x > y
9           then y
10          else if x > z
11                then x
12                else z
```

Figure 5.1. `Mid` function in *Haskell*.

	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	0.00	**0.50**	0.00	0.00	0.00	0.00	0.00	0.67	0.67	0.67	0.00	0.67	
	Ochiai	0.00	**0.58**	0.00	0.00	0.00	0.00	0.00	0.71	0.50	0.50	0.00	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	0.00	0.50	0.63	0.00	0.71	**0.83**	0.00	0.00	0.00	0.00	0.00	0.00	
	Ochiai	0.00	0.41	0.5	0.00	0.58	**0.71**	0.00	0.00	0.00	0.00	0.00	0.00	

Table 5.1. Code coverage and fault rank for `mid`.

To describe the calculus, we may take for instance the fifth line from second `mid` version. We have in total one failing and five passing test cases, thus `totalfailed = 1` and `totalpassed = 5`. Two passing test cases cover Line 5, as well as one failing test case, therefore, `failed(5) = 1` and `passed(5) = 2`, with this, we are able to calculate Tarantula score:

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

Similarly, we may calculate the Ochiai suspiciousness score:

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

Furthermore, the scores highlighted in bold are the ones for the statement containing the error. We may notice that 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, we may observe several other statements receiving matching suspiciousness scores. As explained in Section 2.4.4, one approach is to calculate the best and worst-case scenarios.

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

In the final analysis, we calculated the EXAM score introduced in Section 2.4.4. This score indicates the percentage of the program that should be checked until we reach the error location. We will better observe its application for different scenarios in Section 5.3. Nonetheless, the results for `mid` function are displayed in Table 5.2. For `mid` Version 1, we would have to analyze 42% of the program if we follow Tarantula scores and 17% of it if we follow Ochiai scores. For `mid` Version 2 the results are even better. We 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* programmer would have to look for the bug in less than half the original amount of code lines before finding it.

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

Table 5.2. EXAM score for `mid` function.

5.2 Test Suite

The examples we used to run our tests were the students' final submitted versions. Considering that they did not have errors that need to be fixed in their majority, we

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

Table 5.3. Test suite.

decided to introduce bugs in their code to be later detected by our fault localization technique. The bugs introduced were not repeated.

Table 5.3 names all the programs on our suite. As explained before, `mid` calculates the middle element among three elements. Regarding the Functional Programming class submissions, we kept the names chosen by the students, 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 functionality 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.

Besides, we have the problems from Exercism, their description in Exercism 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 5.3 displays the programs' length in our test suite. This information works alongside our EXAM score results, which will be detailed in Section 5.3, to offer a more precise dimension of the effort level needed by a programmer while locating a bug using *HaskellFL* tool. Our test suite's mean program length is 30.4 lines, and the median program length is 27.5 lines.

The test cases used to run our suite were manually chosen and written and ranged between 3 and 12 test cases per problem, as detailed in Table 5.3. Even though we are not tracking time execution in this master thesis, we tried balancing our test cases between passing and failing. We have based that on what is said in Rao et al. [2013] that the expense of fault localization for most formulas will increase with the increase of class imbalance[1]. In light of that, this a factor to be considered.

Furthermore, Table 5.3 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.

5.2.1 Test Setup

To allow our tests to be interpreted by *HaskellFL*, we rewrote some small code pieces in our test suite. An example is shown in Figure 5.2, which is a real submission from a student enrolled at the UFMG Functional Programming class. In the second line of the `dropWhile` function, the student originally wrote `dropWhile p xs@[] = []`. Similarly, in the third line, the student wrote `dropWhile p xs@(x:xs')`. Both statements use pattern matching with the notation `@`. In Line `3`, this notation allows `dropWhile` to be aware of three different values: the head of the list, kept in variable `x`, the list tail, kept in the variable `xs'` and the complete list, kept in variable `xs`. `DropWhile` is also aware of a given input function, which is kept in variable `p`. For Line `2`, `dropWhile` has access to the empty list in variable `xs` if the empty list is matched in the input. The notation `@` is not available in *HaskellFL*, however this program can be rewritten without any loss in its semantics. An example of that may be seen in Figure 5.3. The second version is the one present in our test suite, after being further modified to insert a bug, and this version is equivalent to the original definition exhibited in Figure 5.2.

```
1 dropWhile :: (a->Bool) -> [a] -> [a]
2 dropWhile p xs@[] = []
3 dropWhile p xs@(x:xs')
4           | p x = dropWhile p xs'
5           | otherwise = xs
```

Figure 5.2. `DropWhile` function submitted by a student.

[1]Class imbalance is the phenomenon of some classes having far more samples than other classes. The same applies to passing test cases being more straight-forward to find than failed test cases.

```haskell
1 dropWhile :: (a->Bool) -> [a] -> [a]
2 dropWhile p [] = []
3 dropWhile p (x:xs)
4         | p x = dropWhile p xs
5         | otherwise = (x:xs)
```

Figure 5.3. `DropWhile` function equivalent to the function in Figure 5.2.

This new instance of `dropWhile` function does not know the complete list via an exclusive variable at the beginning of the function in Line `3`, however the `otherwise` clause can have access to the complete list via the operator `:` which inserts an element at the beginning of a list. The list's head and tail are available through the variables `x` and `xs` respectively, then they may be concatenated. Likewise, the empty list knowledge is not duplicated in Line `2` but the pattern matching still works. With that said, this new `dropWhile` instance can reproduce the original `dropWhile` expected behavior, concatenating the list head and tail in the `otherwise` clause and dropping the `@` notation in Lines `2` and `3`.

Figure 5.4 shows in the `SumOfMultiples` module, another example of a change made in an original test in our 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 Figure 5.5. The version using parenthesis is the one present in our test suite after a bug insertion.

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 our own `Prelude` functions when using *HaskellFL*, as may be seen in Figure 5.5, with the functions `filter` and `sum`. They were imported from *Haskell* `Prelude` in the first `SumOfMultiples` module in Figure 5.4. 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*. We are able to implement several `Prelude` functions with the constructs offered by *HaskellFL*, so this absence does not prevent our 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 our test suite followed the techniques mentioned in Section 2.4.2. For instance, for the `SumOfMultiples` module

```
 1  module SumOfMultiples (sumOfMultiples) where
 2
 3      import Data.List(nub)
 4
 5      sumOfMultiples :: [Integer] -> Integer -> Integer
 6      sumOfMultiples [] limit = 0
 7      sumOfMultiples factors limit =
 8          sum $ distinctFactors factors limit
 9
10      distinctFactors :: (Integral a) => [a] -> a -> [a]
11      distinctFactors [] limit = []
12      distinctFactors (x:xs) limit =
13          nub $
14          (distinctFactors xs limit) ++
15          (appendFactor x limit 1)
16
17      appendFactor :: (Integral a) => a -> a -> a -> [a]
18      appendFactor factor limit index
19          | factor * index >= limit = []
20          | factor == 0 = []
21          | otherwise = (factor * index) :
22            appendFactor factor limit (index + 1)
```

Figure 5.4. `SumOfMultiples` module as available on GitHub.

shown in Figure 5.5, we inserted a bug into Line 19, in the `appendFactor` function. We replaced the operator `>=` with the operator `>`. Another example is the `dropWhile` function in Figure 5.3. For this program, we inserted a bug in Line 4, where we returned only the list tail in the `otherwise` branch, represented by `xs`, instead of the complete list.

Finally, we wrote the passing and failing test cases for every problem present in our test suite. To illustrate the process, we may see the passing test cases for the `dropWhile` function in Figure 5.6 and the failing test cases for the same function in Figure 5.7. As previously mentioned, we tried to keep the number of tests balanced between passing and failing test cases, and we also tried to cover every branch in the code. Nonetheless, this is not always possible. For instance, none of the passing test cases for the `dropWhile` function covers the `otherwise` branch, as an input executing this branch would immediately expose the bug.

Similarly, for the `sumOfMultiples` test cases, we also worked on balancing the number of tests between passing and failing test cases, as may be seen in figures 5.8 and 5.9 respectively. Contrasting with the `dropWhile` function, for `sumOfMultiples`,

```haskell
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)
```

Figure 5.5. `SumOfMultiples` module equivalent to the module in Figure 5.4.

```haskell
dropWhile (\x -> x < 10) [2,5]
dropWhile (\x -> x /= 0) []
dropWhile (\x -> False) []
dropWhile (\x -> x == 0) [0,0,0]
dropWhile (\x -> True) ["Hamilton", "Vettel"]
```

Figure 5.6. Passing `dropWhile` test cases.

the test cases cover all branches present in the code.

In conclusion, for the `dropWhile` function, both Ochiai and Tarantula techniques

```
dropWhile (\x -> x > 5) [6,7,4,3]
dropWhile (\x -> x == 5) [5,6,7]
dropWhile (\x -> x <= 6) [5,6,7]
dropWhile (\x -> False) ["Hamilton", "Vettel"]
```

Figure 5.7. Failing `dropWhile` test cases.

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

Figure 5.8. Passing `sumOfMultiples` test cases.

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

Figure 5.9. Failing `sumOfMultiples` test cases.

ranked the erroneous line first. On the other hand, for the `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.

5.3 Results

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%

Table 5.4. EXAM score for *Haskell* test suite.

Table 5.4 shows EXAM score distribution for our test suite, considering best and worst-case scenarios, for Tarantula and Ochiai methods. It shows which percentage of our test suite programs is inside a specific EXAM score segment. We divided the table into segments of 5%. To demonstrate, considering the Tarantula formula and the best-case scenario, we observe that 58.33% of the programs in our suite secured an EXAM score between 0% and 4.9%. To put it more simply, for 58.33% of the programs in our 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.

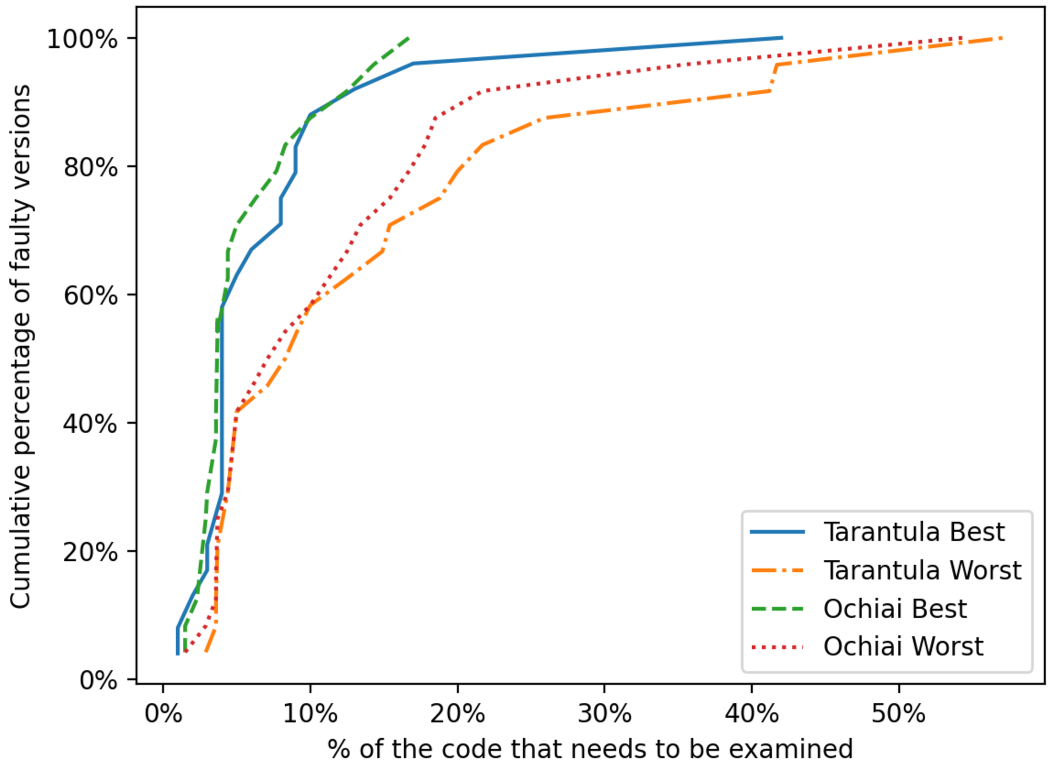


Figure 5.10. Comparison between Ochiai and Tarantula methods for our test suite.

Figure 5.10 illustrates Tarantula 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 Wong et al. [2008] did. Table 5.4 displays the scores used to build the graph.

To better exemplify, we may notice in the Ochiai method in the best-case scenario; we could find all the incorrect statements in our suite examining less than 20% of the source code for each problem.

Furthermore, in Table 5.5 we present the mean, median, and standard deviation of the calculated EXAM scores for our test suite. We notice that 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. Its standard deviation is equal to 4.0%, which indicates that the results are close to the median value.

	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%

Table 5.5. Mean, median and standard deviation of our test suite.

5.4 Threats to Validity

In this section, we discuss the main threats to the validity of our work and the strategies we took to mitigate them.

As previously mentioned, our test suite programs did not have bugs that need to be found, so we introduced them into the code. To mitigate this threat, we followed the mutants' guidelines shown in Section 2.4.2. The guidelines assume that most mutants form realistic faults, even if they are artificially seeded. Therefore, we also confidently assume that our inserted bugs represent mistakes that real students make.

Another threat is related to the passing and failing test cases used as input for the fault localization techniques we are studying; we wrote them. To mitigate this threat, we balanced the number of tests between passing and failing test cases. We carefully and manually worked on finding test cases to cover every branch in the code, whenever this was possible, to keep the odds as fair and unbiased as possible.

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

Finally, in this project, we used Tarantula and Ochiai fault localization techniques. There are several different techniques in the literature, for instance, Barinel

[Abreu et al., 2009b], Op2 [Naish et al., 2011] and DStar [Wong et al., 2013]. Nevertheless, Jones and Harrold [2005] 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 our tests. 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.

5.5 Final Remarks

In this chapter, we began with the two `mid` function versions' case study, showing its respective inputs and the code coverage for each one. Provided that, we were able to calculate both Tarantula and Ochiai suspiciousness scores for all the lines. The fault localization methods attributed the highest score for the correct line in 2 out of 4 studied scenarios. We also calculate the EXAM score for each method and scenario to measure the effort someone would have to locate a bug when following the suspiciousness scores.

Secondly, we detailed our test suite, composed of 24 *Haskell* problems, which are real Functional Programming students' submissions at UFMG, plus code available on GitHub. We mostly introduced bugs in the problems on our test suite. Subsection 5.2.1 describes the process we followed while doing so. Our suite encloses the whole grammar shown in Appendix A and Appendix B displays two programs extracted from our test suite.

Thirdly, we presented the results we got by running *HaskellFL* against our test suite. The results are in terms of the EXAM score. Ochiai method achieved better results than Tarantula. In the best-case scenario, Ochiai could locate the errors for the 24 problems examining less than 20% of the code. The less favorable scenario we studied was the worst-case scenario Tarantula, where `triangle` ranked almost 60% in the EXAM score.

Finally, we discussed the threats to the validity for our work. In the following chapter, we will conclude this dissertation.

Chapter 6

Conclusion

Bugs are a reality in software development, and while experienced programmers may know their way among several bugs, some beginners may feel discouraged by them. Considering, for instance, the functional paradigm, at first sight, it may confuse programmers; this happens because they usually start by learning the imperative paradigm, which has no particular way of handling state. Therefore, several difficulties may appear when programmers try to learn a new way to write code with different reasoning. If programmers do not address these issues early, they may use the functional language like they are using an imperative one for a long time. Consequently, they will take no real advantage of the functional paradigm. This work was conducted in this context, aiming to evaluate the effectiveness of two fault localization techniques in the literature, Tarantula [Jones and Harrold, 2005] and Ochiai [Abreu et al., 2009a], in *Haskell* programs. Additionally, create a tool to aid Functional Programming beginners while debugging their *Haskell* problems.

To achieve our goals, we conducted this dissertation as follows. Initially, we presented our motivation, defined our problem, and enumerated our objectives. After that, we approached several central topics to this dissertation. For instance, we remembered some key *Haskell* concepts addressed by *HaskellFL*, and we also refreshed some lambda calculus concepts needed to build our tool. Additionally, we explained two well-known fault localization approaches, spectrum-based and mutation-based, while explaining the two spectrum-based techniques we used for this work.

We presented several works related to the one we made for this dissertation and pointed out the differences, emphasizing the importance of our tool and why our work is relevant to the field. We enumerated different methods for calculating fault location, and the ones on fault localization that were not used for building *HaskellFL* directly but play a similar role for other languages that not *Haskell*. We also talked about type errors because they are traditionally a source of confusion for programmers, and they play an essential role in repairing code automatically. Furthermore, we mention interactive tools created to tutor *Haskell* apprentices.

We showed that the proposed *HaskellFL* tool differs from the works in Thompson and Sullivan [2020], Chesley et al. [2007] and Dallmeier et al. [2005] in the language supported. The work in Chesley et al. [2007] is different in the sense that their work is an interactive guide for helping to locate an error root cause. Similarly, the works mentioned in Section 3.4 also serve as a guide for programming apprentices, as our work does, but they work as an interactive *Haskell* guide instead of looking for the error automatically. Dallmeier et al. [2005] implements a different fault localization technique that we did not use in *HaskellFL*. Wong et al. [2008] also presented a new method that they compare and contrast against Tarantula, a method we studied and analyzed. Le et al. [2014a] also used *Haskell* in their work; however, they implemented mutation-based fault localization while *HaskellFL* offers spectrum-based fault localization.

After the description of the full process *HaskellFL* utilizes to interpret *Haskell* code and calculate the suspiciousness expressions' rank, we presented our results.

As the main results of this research project we provided *HaskelFL*. This command-line tool successfully locates logical errors in *Haskell* code using spectrum-based fault localization methods, examining very few lines in *Haskell* code for most of our test suite. Besides, we evaluated the Tarantula and Ochiai techniques, and we also evaluated the *HaskellFL* tool against our test suite using the EXAM score. In this context, Ochiai presented better results than Tarantula. Our work compiled a test suite suitable to our *Haskell* chosen subset. This set is diverse and contains abstract data types that were not supported on Singer and Archibald [2018]. This test suite, together with *HaskellFL* is available as an open-source project.

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 our tool is able to aid in real time.
(v) Implement techniques to repair the code.

Bibliography

Abreu, R., Zoeteweij, P., Golsteijn, R., and Van Gemund, A. J. (2009a). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780--1792.

Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2009b). Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88--99. IEEE.

Ball, T., Naik, M., and Rajamani, S. K. (2003). From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN Notices*, volume 38, pages 97--105. ACM.

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., and Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148--175.

Bird, R. (2014). *Thinking functionally with Haskell.* Cambridge University Press.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1987). Occam's razor. *Information processing letters*, 24(6):377--380.

Charguéraud, A. (2015). Improving type error messages in ocaml. *Electronic Proceedings in Theoretical Computer Science*, 198:80–97. ISSN 2075-2180.

Chesley, O. C., Ren, X., Ryder, B. G., and Tip, F. (2007). Crisp–a fault localization tool for java programs. In *29th International Conference on Software Engineering (ICSE'07)*, pages 775--779. IEEE.

Dallmeier, V., Lindig, C., and Zeller, A. (2005). Lightweight defect localization for java. In *European conference on object-oriented programming*, pages 528--550. Springer.

Done, C. (2018). Try haskell. `http://tryhaskell.org/`.

F# (2021). About f#. `https://fsharp.org/about/`.

Fernandes, J. (2004). Generalized lr parsing in haskell. *Informal Proceedings of the Summer School on Advanced Functional Programming, students' presentation*, pages 24--37.

Gerdes, A., Heeren, B., Jeuring, J., and van Binsbergen, L. T. (2017). Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65--100.

Gopinath, R., Jensen, C., and Groce, A. (2014). Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189--200. IEEE.

Groce, A., Chaki, S., Kroening, D., and Strichman, O. (2006). Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229--247.

Handley, M. and Hutton, G. (2018). Improving haskell. In Palka, M. H. and Myreen, M. O., 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. Springer.

HaskellWiki (2018a). Haskell. `https://wiki.haskell.org/Haskell`.

HaskellWiki (2018b). Haskell: An advanced, purely functional programming language. `https://www.haskell.org/`.

Heeren, B., Leijen, D., and van IJzendoorn, A. (2003). 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. ACM.

Heeren, B. J. (2005). *Top quality type error messages*. Utrecht University.

Henderson, T. A. (2018). How to evaluate statistical fault localization.

Jones, J. A. and Harrold, M. J. (2005). 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.

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). 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.

Jones, S. P., Hall, C., Hammond, K., Partain, W., and Wadler, P. (1993). The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93.

Jose, M. and Majumdar, R. (2011). Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437--446.

Just, R., Jalali, D., and Ernst, M. D. (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*, pages 437--440.

Kneuss, E., Koukoutos, M., and Kuncak, V. (2015). 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. Springer.

Kotlin (2021). Learn kotlin. `https://kotlinlang.org/docs/reference/`.

Le, D., Alipour, M. A., Gopinath, R., and Groce, A. (2014a). 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.

Le, D., Alipour, M. A., Gopinath, R., and Groce, A. (2014b). Mutation testing of functional programming languages. *Oregon State University, Tech. Rep.*

Lee, J., Song, D., So, S., and Oh, H. (2018). Automatic diagnosis and correction of logical errors for functional programming assignments. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):158.

Li, Z., Zhang, L., Zhang, Z., and Jiang, B. (2020). 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.

Liu, H., Glew, N., Petersen, L., and Anderson, T. A. (2013). The intel labs haskell research compiler. In *ACM SIGPLAN Notices*, volume 48, pages 105--116. ACM.

Mitchell, J. C., John, C., and Apt, K. (2003). *Concepts in programming languages.* Published by the Press Syndicate of the University of Cambridge, The Pitt Building, Trumpington Street, Cambridge, United Kingdom.

Naish, L., Lee, H. J., and Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1--32.

Nubank, R. (2021). O que é clojure? o que isso tem a ver com o nubank? `https://blog.nubank.com.br/o-que-e-clojure/`.

OCaml (2018). Ocaml is an industrial-strength programming language supporting functional, imperative and object-oriented styles. `https://ocaml.org/`.

Papadakis, M. and Le Traon, Y. (2015). Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605--628.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D., and Keller, B. (2016). Evaluating & improving fault localization techniques. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03*.

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

Pop, I. (2010). Experience report: Haskell as a reagent. In *ACM SIGPLAN International Conference on Funtional Programming*. Citeseer.

Purushothaman, R. and Perry, D. E. (2005). Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511--526.

Rao, P., Zheng, Z., Chen, T. Y., Wang, N., and Cai, K. (2013). 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.

Sakkas, G., Endres, M., Cosman, B., Weimer, W., and Jhala, R. (2020). Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16--30.

Singer, J. and Archibald, B. (2018). Functional baby talk: Analysis of code fragments from novice haskell programmers. *arXiv preprint arXiv:1805.05126*.

Thompson, G. and Sullivan, A. K. (2020). Profl: a fault localization framework for prolog. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 561--564.

Thompson, S. (2011). *Haskell: The Craft of Functional Programming*. Icss Series. Addison Wesley. ISBN 9780201882957.

Tondwalkar, A. (2016). *Finding and Fixing Bugs in Liquid Haskell.* PhD dissertation, University of Virginia.

Wong, E., Wei, T., Qi, Y., and Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42--51. IEEE.

Wong, W. E., Debroy, V., Gao, R., and Li, Y. (2013). The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290--308.

Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). aurvey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707--740.

Zhang, D., Myers, A. C., Vytiniotis, D., and Peyton-Jones, S. (2015). Diagnosing haskell type errors. Technical report, Technical Report http://hdl. handle. net/1813/39907, Cornell University.

Appendix A

Haskell's Grammar Subset

$\langle program \rangle$ ::= ‘module’ X ‘where’ ‘{’ decls ‘}’
| decls

$\langle decls \rangle$ ::= decl decls | d decls | decl | d

$\langle decl \rangle$ ::= 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 ... ‘|’ $e_(i+k)$ ‘=’ $e_(j+k)$

$\langle d \rangle$ ::= ‘data’ X ‘=’ C_1 τ_1 ... τ_k | ... | C_n τ_1 ... τ_k
| ‘newtype’ X ‘=’ C τ

$\langle e \rangle$ ::= 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

$\langle p \rangle$ ::= const | var | C p_1 ... p_k | _

$\langle \tau \rangle$ ::= Integer | Boolean | String | Char | Float | [τ] | (τ_1, ..., τ_k)
| C τ_1 ... τ_k

Appendix B

Buggy *Haskell* programs

```haskell
module SpaceAge where
    data Planet = Mercury
                | Venus
                | Earth
                | Mars
                | Jupiter
                | Saturn
                | Uranus
                | Neptune

    orbitalPeriod :: Planet -> Float
    orbitalPeriod planet =
        let
            earthYear = 31557600
        in
            case planet of
                Mercury -> 0.24084670 * earthYear
                Venus   -> 1.61519726 * earthYear -- BUG
                Earth   -> 1.00000000 * earthYear
                Mars    -> 1.88081580 * earthYear
                Jupiter -> 11.8626150 * earthYear
                Saturn  -> 29.4474980 * earthYear
                Uranus  -> 84.0168460 * earthYear
                Neptune -> 164.791320 * earthYear

    ageOn :: Planet -> Float -> Float
    ageOn planet age = age / (orbitalPeriod planet)
```

Figure B.1. Exercism's problem extracted from our test suite.

```
module Main () where
    import Prelude hiding (map)
    import Data.Char

    map f [] = []
    map f (a:b) = f a : map f b

    dropWhileClone p []           = []
    dropWhileClone p (x:xs)
        | p x         = dropWhileClone p xs
        | otherwise = x:xs

    splitWith p []       = []
    splitWith p x        = x1 : splitWith p x2
        where (x1 ,x2 ) = break p x

    isSpace s = if s == "" -- BUG
        then True
        else False

    break _ []        = ([],[])
    break p (a:x)
        | p a           = ([],a:x)
        | otherwise     = (a:x1,x2)
        where (x1,x2) = break p x

    wordsClone s = if s' == []
        then []
        else (word : wordsClone rest)
            where s' = dropWhileClone isSpace s
                  (word, rest) = break isSpace s'

    concatClone [] = []
    concatClone ([]:vs) = concatClone vs
    concatClone ((x:xs):vs) = x:concatClone (xs:vs)
```

Figure B.2. Homework extracted from our test suite.