Tool-Driven Quality Assurance for Functional Programming and Machine Learning

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op dinsdag 29 october 2024 om 15.00

door

Leonhard APPLIS

Master of Science in Computer Science,
Technische Universität Georg Simon Ohm, Duitsland,
geboren te Starnberg, Duitsland.

Dit proefschrift is goedgekeurd door de

promotor: Prof. Dr. A. van Deursen
copromotor: Dr. A. Panichella

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. A. van Deursen,	Technische Universiteit Delft
Dr. A. Panichella,	Technische Universiteit Delft
Onafhankelijke leden:	
Prof. dr. G. Keller,	Universiteit Utrecht
Prof. dr. C. Le Goues,	Carnegie Mellon University, United States of America
Prof. dr. F. Sarro,	University College London, United Kingdom
Prof. dr. K.G. Langendoen,	Technische Universiteit Delft
Dr. J. Cockx,	Technische Universiteit Delft
Prof. dr. A. Zaidman	Technische Universiteit Delft, reservelid

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords:	Software Engineering, Functional Programming, Search-based Software Engineering
Printed by:	ProefschriftMaken.nl
Cover:	Leonhard Applis
Style:	TU Delft House Style, with modifications by Moritz Beller `https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

ISBN 978-94-6510-186-6

Tradition ist nicht das Anbeten der Asche
Tradition is not to worship the ashes
sondern das Schüren des Feuers
but to stoke the fire

(frei, nach Jean Jaurés)

Contents

Summary

Finding and fixing software faults is a major part of software development and as such any improvement for such tasks is a welcome aid for developers and a worthwhile field for researchers. Like programming in general, debugging and repair need specialized tools to provide the necessary information (like the usage of runtime resources) or assure quality (e.g. with test suites). Only then, developers are able to repair faults without introducing new ones. There are also more sophisticated tools that provide stronger, more automated help to developers: Program coverage summarizes run-time behavior, fault localization helps to narrow down suspicious parts of the code and automated program repair suggests possible patches that lead to a passing test suite. On top of these approaches, large language models show promising capabilities to generate, alter and test source-code, but they have yet to be tested and hardened for their security and quality.

To enable the next generation of state of the art quality-assurance tools, this thesis investigates different techniques and their respective tools to improve their precision and correctness. To this end, we develop procedures to quantify the robustness of large language models of code to identify their weaknesses when facing metamorphic noise or statistically unlikely data. After examining quality of tools, this dissertation works towards improving existing tools and approaches in the field of functional programming, particularly for Haskell. Functional programming is a field rich of unique options such as properties, strong type-systems, side-effect free functions, but also challenges like non-strict evaluation.

Our results regarding large language models show that there are short-comings when dealing with redundant elements and that such elements can be intentionally searched for. This implies a need for further improvement of the models, to provide more consistent results for trivial changes.

The work centered around Haskell shows the value of utilizing compiler- and language-features to enhance existing techniques: Program repair can be performed with a reduced search space due to compiler-suggested elements, stack-traces and program-coverage can be enhanced by introducing an evaluation-trace and fault-localization is aided by types and expression-level granularity. While the implementation is specific, the approaches remain transferable: Any feature that is used from Haskell in this dissertation, is (or can be) implemented for Java.

In summary, this thesis touches on different topics of assuring software quality and their tools by introducing novel information. This thesis lays groundwork to improve the next generation of development-tools that utilize large language models or statically typed languages.

Samenvatting

Het opsporen en herstellen van softwarefouten is een essentieel onderdeel van softwareontwikkeling. Elke verbetering voor dergelijke taken is een welkome hulp voor ontwikkelaars en een waardevol onderzoeksveld. Net als programmeren in het algemeen, hebben debuggen en repareren gespecialiseerde tools nodig om de benodigde informatie te verstrekken (zoals het gebruik van runtime-resources) of om kwaliteit te waarborgen (bijvoorbeeld met test suites). Alleen dan kunnen ontwikkelaars fouten herstellen zonder nieuwe fouten te introduceren. Er zijn ook meer geavanceerde tools die sterkere, meer geautomatiseerde hulp bieden aan ontwikkelaars: Programmacoverage vat het runtime-gedrag samen, foutlokalisatie helpt verdachte delen van de code te beperken en geautomatiseerde programmareparatie suggereert mogelijke patches die leiden tot een geslaagde test suite. Naast deze benaderingen tonen grote taalmodellen veelbelovende mogelijkheden om broncode te genereren, aan te passen en te testen, maar ze moeten nog worden getest en versterkt voor hun beveiliging en kwaliteit.

Om de volgende generatie van state-of-the-art kwaliteitsborgingstools mogelijk te maken, onderzoekt deze scriptie verschillende technieken en hun respectieve tools om hun precisie en correctheid te verbeteren. Daartoe ontwikkelen we procedures om de robuustheid van grote taalmodellen voor code te kwantificeren en hun zwakheden te identificeren wanneer ze worden geconfronteerd met metamorfische ruis of statistisch onwaarschijnlijke data. Na het onderzoeken van de kwaliteit van tools werkt deze dissertatie aan het verbeteren van bestaande tools en benaderingen op het gebied van functioneel programmeren, met name voor Haskell. Functioneel programmeren is een veld dat rijk is aan unieke opties zoals eigenschappen, sterke type-systemen, side-effect vrije functies, maar ook uitdagingen zoals niet-strikte evaluatie.

Onze resultaten met betrekking tot grote taalmodellen laten zien dat er tekortkomingen zijn bij het omgaan met redundante elementen en dat dergelijke elementen opzettelijk kunnen worden gezocht. Dit impliceert een behoefte aan verdere verbetering van de modellen om consistentere resultaten te bieden voor triviale wijzigingen. Het werk rond Haskell toont de waarde aan van het benutten van compiler- en taalfeatures om bestaande technieken te verbeteren: Programmareparatie kan worden uitgevoerd met een verminderd zoekgebied dankzij compiler-gesuggereerde elementen, stack-traces en programmacoverage kunnen worden verbeterd door een evaluatiespoor in te voeren en foutlokalisatie wordt ondersteund door types en expressieniveau-granulariteit. Hoewel de implementatie specifiek is, blijven de benaderingen overdraagbaar: Elke feature die in deze dissertatie van Haskell wordt gebruikt, is (of kan worden) geïmplementeerd voor Java.

Samenvattend raakt deze scriptie verschillende onderwerpen aan met betrekking tot het waarborgen van softwarekwaliteit en hun tools door nieuwe informatie te introduceren. Deze scriptie legt de basis voor het verbeteren van de volgende generatie ontwikkeltools die gebruik maken van grote taalmodellen of statisch getypeerde talen.

1 Introduction

It is not exactly unique that I like programming, both as work and as a hobby, because it is a creative process and building things is fun. What made me pursue the fields of software engineering as a researcher was a certain curiosity in the patterns I saw both in me and my colleagues: When a new feature was required, it was designed, implemented, tested, integrated, crashed, and then improved. This pattern was not always followed exactly, but it repeated in both smallest unit-level elements, modules, infrastructure and even with the introduction of new team members. For me, this overarching iterative pattern is what makes software engineering more than just programming: Once you see the team members as part of a project, the project is never truly finished.

This thesis investigates two of the trends that are forming in the early 2020s: A renaissance of functional programming (and its paradigms) [1] and the maturity of machine learning that created the first generation of production-ready artificial intelligence for software engineering (AI4SE) tools [2]. Let's look at some of the trends in detail.

Functional Programming can be dated as early as 1956 and the functional language Lisp [3], and bases itself in the legacy of one of computer science's most renown figures, Alonzo Church and his λ-calculus [4]. With Churchs work as a logician, in addition to many of the issues of computer science being centered around mathematical computation, functional programming often thrives in fields that are closer to mathematics and physics [5], conceptually fitting better than other paradigms. The stateless nature of functional programs allows for an easy composition, and maybe more important, for readily testable sub-routines. Unfortunately functional programming is not widely adopted, and languages like Agda [6] and Coq [7] are effectively tied purely to academia. Other languages like Haskell, Erlang, Scala or OCaml find more real-world application, yet their use falls short compared to languages from different paradigms. The impact of functional programming is still not to be under-estimated: Rust [8], an imperative programming language, chose to adopt strong type checking similar to Haskell and extend the compiler-checks to identify memory-usage. Java and C#, both object oriented languages, opted for more functional features when it comes to data processing by promoting lambda functions in streams [9] (Java) or LINQ [10] (C#) [1]. Some domains, like large-scale data processing with SPARK

[1] LINQ looks (intentionally) like SQL, but forms a query of filters, joins, and mappings behind the scenes.

[11], have been ground-up designed around immutable data and composable functions and succeed with these principles. So even when Haskell and pure FP languages are not widely adopted, the paradigm and its benefits are (selectively) reapplied in other domains [12][1]. This is a reasonable approach - if your domain is most approachable by the concept of objects, object oriented programming (OOP) is what you want. Nevertheless, there are still benefits to using functions as first-class citizens. Languages like Scala [13][14] and F# [15] are designed to combine functional paradigms with OOP code-bases, and are well received for their interoperability [16].

The other dominating trend is the rise of production-ready *machine-learning tools*. This spans most industries, but is also prevalent in Software Engineering [17]. Specialized tools like Github's Copilot [18][19] but also (general) large language models (LLMs) like ChatGPT yield promising results for code [20], tests [21] and documentation [22]. Right after the questions of copyright [23], we must ask ourselves: Are they that good? To answer this question, we need to put them to the test - which is also the goal of this thesis. As shown later in Section 1.3, at the time of writing ChatGPT gives solid answers when facing *normal* code, but once you introduce uncommon variable names, the behavior deteriorates into a confused mess - exactly what our work on Lampion (Chapter 2) tries to assess. This resonates with some of the latest findings in literature [24][25][26].

We see that complex systems are constructed of multiple components, such as LLMs or functional elements, and quality-assurance in these fundamentals pays of as quality in the final applications. Before we target downstream tasks, we need to address issues of noise, traceability and explainability — but any progress on these matters will be inherited by task-specific models and larger software projects. One direct way to reach improvements is testing and debugging at the core-elements: The functional languages and the basic machine-learning models. This dissertation aims to add improvements to various testing and quality-assurance processes around software engineering when using models trained on code and functional programming: While there is an emerging body of research on adversarial and counterfactual examples for LLMs [27], the existence of large-scale noise and nonsense is yet to be addressed. To show the importance of this topic, we investigate and quantify the effects of redundant noise in Chapter 2 and Chapter 3. Improvements on this matter should chain into most applications that utilize these models, as issues with data-quality such as anti-patterns and smells are not limited to research-driven testing, but are also produced by developers [28].

In its second part, this thesis revolves around fault localization (Chapter 7), understanding effects of non-strict evaluation with occurring errors (Chapter 5) and automated program repair (Chapter 6) when facing different facets of Haskell bugs (Chapter 4). All of these techniques form parts of typical quality assurance, to analyze, test and repair faulty programs during software engineering. Haskell provides both special problems (e.g. issues surrounding laziness) and solutions (e.g. property-based testing), that open up opportunities for the research conducted over the course of this thesis. As many formerly exclusive functional-programming features have made it into other languages and paradigms [1], it also introduced some of the issues (e.g. memory leaks [29]). This thesis progresses on state of the art tooling, that can help both Haskell developers and potentially forms the start for applications in other languages too.

1.1 Research Questions

The preamble motivates us to improve the roots of the trends, which leads to the two primary research questions of this work:

RQ1 - Challenges in FP and AI4SE
How can we identify and fix quality issues originating from the use of lazy-functional paradigms or in the behavior of machine learning models in software engineering tasks?

RQ2 - Tooling Opportunity in FP
Given the unique attributes of the Haskell ecosystem, how can we utilize compilers, types and properties to build more reliable and efficient tools to assist quality assurance?

Due to the limitations of a single PhD, this thesis focuses on certain aspects of debugging, fault localization and program repair in the discussed domains.

Investigating these research questions will allow us to choose appropriate tools by measuring their impact and effectiveness, and to implement novel tools and approaches. Tools have been a centerpiece of development, and very likely will remain as such. It is important to see their value as the accumulated effort of developers and communities: Program coverage can only be measured with a coverage tool, performance is commonly measured with cost-centers, and many more tasks are only solved by the use of tooling. In this manner, this dissertation introduces foundations for tooling based on unique Haskell features and the behavior observed in machine learning models.

1.2 Features of Functional Programming & Haskell

Functional programming consists of a variety of languages that share some, often more mathematical, paradigms [30]. Originating from lambda calculus, most functional languages are stateless and treat functions as first class entities [12]. Not mandatory, but common, is a strong use of compositions and a *strong* compiler[2] - centered around types, a prominent goal in functional programs is to make faulty states un-representable[3] or express their possibility explicitly. To provide a simple example, most lookup functions for lists, dictionaries etc. do not return an `x` but a `Maybe x`. This way code that invokes the lookup must deal with the two possibilities of receiving `Just x` (an element) or `Nothing`. Many types of effects, alternatives to consider or special behavior are expressed by *Monads* [31], that define everyday constructs such as `Maybe`, Lists and `IO`.

The functional work presented in this thesis builds on Haskell [32]. Haskell is statically typed (see the compiler behavior above), while also providing type-inference when possible [33]. This combination leads to the proverb *"once it compiles, it works"*, as the compiler will reject a large number of implementations that are done without thought. The second important attribute of Haskell throughout this work is *laziness*, i.e. all expressions are only evaluated when required [34]. This allows for some functional pearls utilizing infinite lists that would knock out imperative programs [35], but also poses challenges for performance

[2] *strong* refers to the capabilities it hands to its users. Not only are binaries compiled, but it also provides plugins, modules and access to internal processes as well as intermediate artifacts such as ASTs

[3] See Kings Blogpost *Parse - don't validate* (`https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/`)

and debugging. It can happen that a program happily generates a lot of `thunks` (pending evaluations), that are due all at once (performance issues) or contain faulty elements (bugs) which cause errors at a different point of time than their position in the call-stack indicates. Both attributes are presented in more technical detail in their respective chapter 5.

The value of Haskell for software engineering is twofold:

First, many novelties in other languages are pioneered in the functional programming community. The use of anonymous lambda-functions and higher order functions, as well as their support in the type system as first class entities, stem from functional programming languages. It is not unreasonable to expect other features that are currently explored, e.g. dependent types [36], to make a similarly transfer into mainstream languages. The different paradigm makes it possible to explore novel features that are not easily done in e.g. Python, but might be a worthwhile longterm target. Barriers like this also regress tooling: Work by Ye et al. [37] links a compiler to a neural network and includes the compilation output (pass or failure) as part of the loss function. Despite this being an improvement of existing work on LLMs, we should consider that many code-generation tools [38] are able to create compiling code *by design*. Arguably, this is due to program-code being treated more as text than as structured data, on top of compilers not providing direct programmatic interfaces[4]. Both are matters of accessibility - and work in this thesis shows that once the compiler is also a library tools can be designed with higher quality. Instead of being restricted, tools benefit from the constraints and information provided by the compiler, offering guidance that avoids many pitfalls and sub-optimal solutions.

Second, the strictly enforced paradigms also require a certain set of habits as a side effect. As composing a complex function all at once proves difficult (both semantically and syntactically), developers are forced to separate concerns into smaller functions. Due to the stateless nature, it is also easy to write unit tests [39] or stronger quality assurance like properties [40][41] and formal verification [42]. In general, resulting Haskell code is often a bit complex, but adheres to general *best practices* [43] of programming, not limited to functional programming paradigms. This is reflected by some (imperative) domains where functional principles are chosen as standard, e.g. reactive programming [44][45] and their frameworks like React [46] or RxJava [47] in non-functional languages.

1.3 Current Machine Learning in Software Engineering

The role of machine learning is constantly growing, and with the advent of Githubs Copilot [2], ChatGPT(4) and similarly strong generative models [48], the first useful, non-specialist models became available to a broad audience. Anyone, be they excel-power-users, students or hackers can ask anything about LaTeX, Linux or Lisp.

It is important to note that this generation of tools came out during the course of this thesis, and the work around ML in this dissertation predates the latest generation of LLMs. Nevertheless, a fundamental question is shared amongst the *old* models and the new ones: How do I know it's correct?

[4]One of Javas most prominent meta-programming libraries, `Spoon`, has to re-implement every element of Javas language specification. `https://github.com/INRIA/spoon`

When developing and testing large language models, in the same fashion as normal programs, there are functional and non-functional requirements. Functional requirements for code generating models revolve around the produced artifacts, i.e. does the code compile, does it run, *look ok* (readability and minimalism to human readers), is the test suite passing, etc. Non-functional requirements cover security, performance and, as a focus in LLMs, the explainability of the process. Especially the non-functional requirements face numerous challenges, which are only amplified by the rapid progress made in generative artificial intelligence.

To provide a non-programming example, ChatGPT3 was among the first models that could reliably answer a broad range of questions with reasonable background information. Trying to address safety issues [49], OpenAI implemented checks for users that ask about sensitive topics such as suicides or terrorism [50]. After its initial success, users quickly discovered a work around: The *Do anything now* (DAN)-mode [51] could be activated, by giving ChatGPT instructions to roleplay as a comic villain, Hollywood actor or novelist. Despite OpenAIs efforts, this game of cat [52] and mouse [53] continues into ChatGPT4 [49][54].

One non-functional requirement that is explored and tested in this thesis is *robustness*, which we define as the ability to deal with noise. There is a certain amount of noisy and low-quality elements that is to be expected in code [55], and that are usually no issue for compilers, analysis tools and programmers [56]. While semantically redundant, a change in variable names can have a great impact on a LLMs performance. We see an example of changed code in Figure 1.2 for which we asked ChatGPT4[5] to generate JUnit-tests with 100% branch coverage [57].

The original example is one of the Defects4J [58] and calculates the reciprocal of a complex number. The changes are purely in naming and consist of English words.

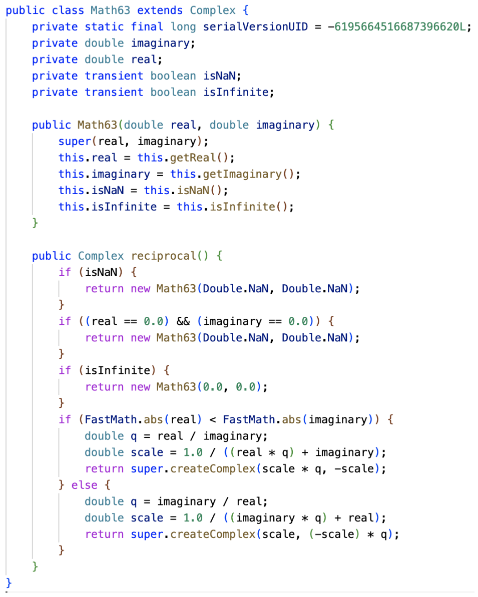
The initially generated unit tests in Figure 1.2a have some virtues and cover all branches and most of the asserts are meaningful and understandable. Most important is that ChatGPT is able to link the original code back to the `Complex`-class and utilizes methods from the parent class (`isNan()`).

The renamed output in Figure 1.2b surprises us at a variety of places. First, ChatGPT names the created objects `instance` which is arguably a loaded key-word in Java. It still captures some of the logic that the class provides, like a tuple of values (like a complex number does), but it is not able to derive the parent class like it did earlier. Lastly, instead of providing meaningful asserts, the last tests suggest a *DIY*-approach for the developer to add their own tests.

The original output is a set of solid unit -tests and the task of branch coverage is achieved. A simple renaming leads to a noticeable drop of quality in naming, and defaulting to not-null-asserts. The provided example is just one of many places where even initial probing reveals shocking shortcomings; recent work by Sallou et al. [59] identifies a set of concerning examples. The educated guess of the authors is that ChatGPT simply memorized the Defects4J bugs [58], as they are publicly hosted on GitHub and widely discussed in academic publications and forums like StackOverflow, which also (likely)[6]

[5]With ChatGPT4 as received on 1st of May 2024. We used different sessions for the different files.

[6]As of writing, OpenAI does not publicly provide information on their training data. A prominent candidate is the open-source dataset `the PILE`[60], which was at some point part of the pre-training. `The pile` explicitly

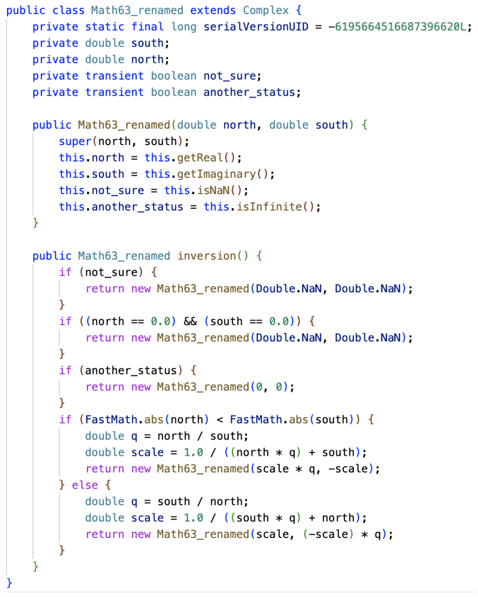


```java
public class Math63 extends Complex {
    private static final long serialVersionUID = -6195664516687396620L;
    private double imaginary;
    private double real;
    private transient boolean isNaN;
    private transient boolean isInfinite;

    public Math63(double real, double imaginary) {
        super(real, imaginary);
        this.real = this.getReal();
        this.imaginary = this.getImaginary();
        this.isNaN = this.isNaN();
        this.isInfinite = this.isInfinite();
    }

    public Complex reciprocal() {
        if (isNaN) {
            return new Math63(Double.NaN, Double.NaN);
        }
        if ((real == 0.0) && (imaginary == 0.0)) {
            return new Math63(Double.NaN, Double.NaN);
        }
        if (isInfinite) {
            return new Math63(0.0, 0.0);
        }
        if (FastMath.abs(real) < FastMath.abs(imaginary)) {
            double q = real / imaginary;
            double scale = 1.0 / ((real * q) + imaginary);
            return super.createComplex(scale * q, -scale);
        } else {
            double q = imaginary / real;
            double scale = 1.0 / ((imaginary * q) + real);
            return super.createComplex(scale, (-scale) * q);
        }
    }
}
```

(a) Original Code



```java
public class Math63_renamed extends Complex {
    private static final long serialVersionUID = -6195664516687396620L;
    private double south;
    private double north;
    private transient boolean not_sure;
    private transient boolean another_status;

    public Math63_renamed(double north, double south) {
        super(north, south);
        this.north = this.getReal();
        this.south = this.getImaginary();
        this.not_sure = this.isNaN();
        this.another_status = this.isInfinite();
    }

    public Math63_renamed inversion() {
        if (not_sure) {
            return new Math63_renamed(Double.NaN, Double.NaN);
        }
        if ((north == 0.0) && (south == 0.0)) {
            return new Math63_renamed(Double.NaN, Double.NaN);
        }
        if (another_status) {
            return new Math63_renamed(0, 0);
        }
        if (FastMath.abs(north) < FastMath.abs(south)) {
            double q = north / south;
            double scale = 1.0 / ((north * q) + south);
            return new Math63_renamed(scale * q, -scale);
        } else {
            double q = south / north;
            double scale = 1.0 / ((south * q) + north);
            return new Math63_renamed(scale, (-scale) * q);
        }
    }
}
```

(b) Renamed Code

Figure 1.1: A (known) Bug from Defects4J - original and renamed

form part of the training corpus. As part of their call to action, they suggest researchers to utilize metamorphic testing similar to Figure 1.1, a task that is explored in chapter 2. These changes make it possible to circumvent the models memorization, and reveal the quality of reasoning and deduction when facing the tasks.

ChatGPT4 is one of the most popular models and in fact has sometimes surprising capabilities. Still, even a simple set of tricks (such as renaming) makes half of the output useless - an issue we seek to better understand in this thesis.

includes GitHub, StackExchange and Arxiv.

```java
@Test
public void testReciprocalNaN() {
    Math63 complex = new Math63(Double.NaN, Double.NaN);
    Complex reciprocal = complex.reciprocal();
    assertTrue("Reciprocal should be NaN", reciprocal.isNaN());
}
@Test
public void testReciprocalZero() {
    Math63 complex = new Math63(0.0, 0.0);
    Complex reciprocal = complex.reciprocal();
    assertTrue("Reciprocal of zero should be NaN", reciprocal.isNaN());
}
@Test
public void testReciprocalInfinite() {
    Math63 complex =
        new Math63(Double.POSITIVE_INFINITY, Double.POSITIVE_INFINITY);
    Complex reciprocal = complex.reciprocal();
    assertEquals("Reciprocal of infinity should be zero",
        new Math63(0.0, 0.0), reciprocal);
}
@Test
public void testReciprocalRealLessThanImaginary() {
    Math63 complex = new Math63(1.0, 2.0);
    Complex reciprocal = complex.reciprocal();
    assertFalse("Reciprocal should not be NaN", reciprocal.isNaN());
    assertFalse("Reciprocal should not be infinite",
        reciprocal.isInfinite());
}
@Test
public void testReciprocalRealGreaterThanOrEqualToImaginary() {
    Math63 complex = new Math63(2.0, 1.0);
    Complex reciprocal = complex.reciprocal();
    assertFalse("Reciprocal should not be NaN", reciprocal.isNaN());
    assertFalse("Reciprocal should not be infinite",
        reciprocal.isInfinite());
}
```

(a) Output of original code

```java
@Test
void testInversionNotSure() {
    Math63_renamed instance = new Math63_renamed(Double.NaN, Double.NaN);
    Math63_renamed result = instance.inversion();
    assertTrue(Double.isNaN(result.north) && Double.isNaN(result.south));
}
@Test
void testInversionZeroNorthSouth() {
    Math63_renamed instance = new Math63_renamed(0.0, 0.0);
    Math63_renamed result = instance.inversion();
    assertTrue(Double.isNaN(result.north) && Double.isNaN(result.south));
}
@Test
void testInversionAnotherStatusTrue() {
    Math63_renamed instance =
        new Math63_renamed(Double.POSITIVE_INFINITY, Double.POSITIVE_INFINITY);
    Math63_renamed result = instance.inversion();
    assertEquals(0.0, result.north, 0.0);
    assertEquals(0.0, result.south, 0.0);
}
@Test
void testInversionAbsNorthLessThanAbsSouth() {
    Math63_renamed instance = new Math63_renamed(1.0, 2.0);
    Math63_renamed result = instance.inversion();
    // Use appropriate assertions to validate the result
    // This is a placeholder, adjust according to actual logic and expected results
    assertNotNull(result);
}
@Test
void testInversionAbsNorthGreaterThanOrEqualAbsSouth() {
    Math63_renamed instance = new Math63_renamed(2.0, 1.0);
    Math63_renamed result = instance.inversion();
    // Use appropriate assertions to validate the result
    // This is a placeholder, adjust according to actual logic and expected results
    assertNotNull(result);
}
```

(b) Output of renamed code

Figure 1.2: ChatGPT4 generated Unit-Tests before and after renaming

1.4 Research Outline

In the following, we take a detailed look at sub-research questions and how they relate to the chapters presented in this dissertation.

The first element we want to investigate is the quality of large language models and other models of code. Examples like the one given in figure 1.2 are easy to reproduce and occur in different downstream tasks. As emphasized in section 1.3, even simple changes can lead to big differences. We want to provide more than anecdotal evidence of this behavior: Lucky for us, as even trivial changes proof effective, we are able to automatize their generation. In the field of testing, the concept of comparing outputs from *equivalent* input is known was *metamorphic testing*. This approach to metamorphic testing for models of code embodies ***RQ1.1***:

RQ1.1

Are machine learning models that are trained on code robust to trivial, nonsensical or noisy code and if so, to what degree can we identify weaknesses in LLMs?

RQ1.1 will be investigated in **Chapter 2** and a set of experiments focused on introducing noise and observing deltas in performance. While initially successful in producing statistically significant changes, the computational power for testing was already an impacting factor, indicating even greater resource-need for an organized re-training of the models to address the identified weaknesses. We sought to improve the procedure by finding *better* counter examples *quicker*, and interpreted the generation of data as a search task. Solving search tasks for software engineering data can face two juxtaposed issues: Interpreting programs as text, and approaching the task from text-generation and search will lead to a large amount of error due to programs simply being semantically or syntactically wrong. Considering the task as *program generation* will encounter problems with search space, as most languages are turing complete. This is not a novel problem, and one technique that crystallized in the automated software engineering community as a solid starting point is *genetic search*. It strikes a balance between the potential infinite combinations of program-elements while preserving syntactic correctness. We try to investigate the merits of genetic search in ***RQ1.2***:

RQ1.2

Can genetic methods be applied to improve the quality and performance of code-mutation when testing software engineering machine learning models?

RQ1.2 is addressed within **Chapter 3**. We implement a genetic algorithm (GA) that optimizes output-deltas in regard to the code-changes for a model under test. Due to GAs, we are able to provide statistically significant changes quicker than the *apply-all* initial approach and better than a random search.

Functional programming is sometimes considered a bit complex and it is unfortunately under-represented in software engineering research. As it is a domain that often follows different paradigms, some of the *standard* approaches are not applicable. Many projects utilize properties as a form of unit testing, but these cannot *simply* be translated to a JUnit unit test. As such, we should be carefully investigating their behavior before transplanting existing techniques that utilize test suites. Other differences arise from the language specifications: With lazy evaluation being the default, there can be parts of a program that are executed in a failure, but never evaluated. Can we consider these expressions *innocent*?

These thoughts motivate ***RQ1.3***, which is also meant to provide an easy (and measured) introduction to the domain of Haskell bugs. Some of the later chapters assume a certain understanding of special language features, so we first want to introduce some common concepts and problems.

RQ1.3

What do faults in open source Haskell projects look like, and how are they fixed? Are there visible trends in errors, approaches and methods?

We answer ***RQ1.3*** in **Chapter 4** with a novel collection of documented open-source faults in Haskell Projects, in the fashion of Defects4J. As an addition for this thesis, some thought and reflection on the reoccurring problems, their tests and maintainers have been added.

One popular feature, but also source for unique issues, is Haskell's laziness. While the idea is simple, and smaller examples can be fully understood, once a complex system is in place too many parts are in motion: On top of a larger code base, the effects of compiler-optimization and runtime environments multiply the complexity, reducing the developers agency back to *trial and error*. In currently unpublished work[7] we interview Haskell maintainers and a emerging trend is that they overhaul their debugging approach once an issue is identified as a *memory leak*. Not all memory leaks are caused by laziness, but it is a potential culprit and the developers investigate such issues with tooling (dynamic analysis) plus manual efforts (*bang* patters). Not only performance, but other errors can originate from laziness as well: It is possible to face an offset of crash and evaluation, which we target in *RQ1.4*. Once this happens, a developer receives stack-traces that presents recently *called* parts, while the *last evaluated* are the points of interest.

RQ1.4

Are there unique issues arising in Haskell Faults due to its lazy evaluation, and are we able to make them visible and traceable?

For ***RQ1.4***, **Chapter 5** proposes an addition to the Haskell Program Coverage (HPC) module that captures not only recently touched, but also recently evaluated code. In case of an crash, we can see not only the last 'called' expression, but also the last evaluated (which is often the origin of the fault - un-evaluated expression cannot lead to a crash). As an example application, we report the last evaluated expression next to the stacktrace on crash and see if the fault is present in them and with which provenance. The experiments show that for some crash-types, the last evaluated expression is preferable over the stack trace, allowing developers to find the faulty location easier.

The work on *RQ1.4* closes the first part of this dissertation, which can be roughly grouped into *identifying issues*. For most developers as well as this thesis, identifying problems is not enough: We also have to fix them. As such, the second part of this thesis is devoted to assisting program repair, and *RQ2.1* tries to address the issues but

[7]Under the working title *What about Haskell Bugs? Adapting Bug Taxonomies to Haskell's Features and Community*, submitted to IFL2024 (*The 36th Symposium on Implementation and Application of Functional Languages*)

also the possibilities of Haskell that we have seen in earlier chapters. While types and properties might seem hard to master, once they are in place they can greatly aid developers, but also guide automated and semi-automated approaches. One particular approach that was brought to life (and my attention) by Matthías Páll Gissurarson are typed holes (_) which developers can utilize as temporary placeholders (comparable with Javas `NotImplementedException`). On compilation, the developer receives an error about the typed hole with its expected type. Previous work [61] implemented *hole fits* — and the developer is presented not only the required type, but possible elements in scope that fit this signature. A prominent issue in program repair is the sourcing of *donor code*, i.e. code that can be used for the repair. Again, we normally run into issues of a large search space in case of program generation, or into syntactic issues if we treat code too much as text, similar to the difficulties in *RQ1.2.* The availability of typed holes and hole fits might be the ingredient that we are missing, and we investigate it with *RQ2.1*:

RQ2.1

Can the field of program repair benefit from strongly typed language features, for example by exploiting typed holes in Haskell?

For ***RQ2.1*** we present PropR in **Chapter 6**, a novel program repair tool that uses genetic search and typed holes to repair haskell modules. By replacing suspicious expressions with a typed hole _, we can ask the compiler for suggestions of other expressions that have the correct type. The suggestions are not limited to constants or module-sourced code, but can include any expression in scope, covering the base-library and all dependencies in scope. Our experiments show that repairs are possible, and a reasonable amount of the fixes addresses the bugs (roughly 10%, similar to early findings in Astor [62], a mature program repair approach for Java).

Automated program repair is a field where many software engineering techniques go hand in hand - as such, the result is often only as good as their weakest part. For our initial implementation, we used a simple heuristic for potential faulty code, by considering all expressions that are touched by failing tests as targets for repair. This proofed feasible, but mature tools such as Astor [62] utilize the more sophisticated approach of spectrum based fault localization: Instead of filtering expressions binary, each expression is given a suspiciousness from its execution pattern in the test suite, and locations that are in more failing and less passing tests are considered better targets for replacement. In both Java and C, techniques that are centered around program-coverage and program-spectra [63] are common, and the availability of HPC in Haskell motivates *RQ2.2*:

RQ2.2

Is spectrum based fault localization (SBFL) applicable to real world Haskell programs and can type information be used to improve its performance?

In **Chapter 7** we present an extension to Haskell´s popular Tasty test framework[8] that allows to gather spectra including types, to address ***RQ2.2***. In our work we utilize the HasBugs datapoints presented in chapter 4 to localize faults, and find that for many of the faults existing SBFL techniques perform well. Some bugs had a outstanding, novel problem:

[8]`https://hackage.haskell.org/package/tasty`

The code at fault was not touched by failing tests, for example due to code-generation or as it was executed by a separate thread. For these faults normal SBFL formulas failed (as they require candidate elements to be executed by a failing test) but the type features can be used to identify some of the faulty expressions.

In summary, the research throughout this thesis aims to improve the life of a modern software developer by utilizing progress in basic elements of a modern tech-stack (machine learning models, program coverage, typed holes) and built better tools for them. The tools built throughout this thesis are of a prototypical nature, keeping realism in mind (either by using real programs, or by respecting computational boundaries). While the prototypes presented in this thesis might fall short in adaptation, they are easy to understand, re-implement and migrate.

1.5 Methodology

Throughout this thesis, common methods from the software engineering research community are applied.

To form a foundation of software engineering research, a baseline of software artifacts are required to evaluate approaches and tools against each other. This is often achieved by mining software repositories [64][65][66], either by directly evaluating a tool or producing a re-usable dataset. All of the datasets in this dissertation are sourced in that manner by extracting information from public repositories on Github: CodeBERT [67] was trained using data from the CodeSearchNET challenge [68], Defects4J [58] is a collection of well-documented bugs in Java programs drawn from Github and HasBugs in Chapter 4 was designed to follow the same approach as Defects4J. Only the data used in Chapter 5 stems from a *call-to-action* in the Haskell community, where a `nofib`-performance benchmark [69] was established for the GHC, which was adjusted by Silva [70] to produce faults. Many of the faults were open source programs that got *donated*. The realism and availability makes mining software repositories a standard approach for the field of software engineering, and the shared datasets provide a way to compare approaches. The importance of this methodology is also reflected by the conference *Mining Software Repositories* (MSR), which is collocated with the *International Conference on Software Engineering*(`ICSE`).

The general spirit of the research in this dissertation follows *Design Science* [71]: We build a tool and evaluate (and quantify) its effects. This is either done against a status quo (e.g. the fault localization in CSI-Haskell in chapter 5), or against existing approaches (like the improvements using evolutionary algorithms in chapter 3).

One way to quantify effects is statistics. A first, important differentiation when applying statistics to software engineering results is to identify whether the underlying data is normally distributed or not. An effective test for normality is the Shapiro-Wilk test [72]. When dealing with code and other derived artifacts, both normal and irregular distributions can naturally occur: As code is similar to language [73], most of the tokens and their attributes follow a normal distribution. The rare event of a bug, and the occurrence of buggy lines of code, is sparsely distributed. In the same manner, many of the bug-related tasks and metrics are also not normally distributed and require methods common for outlier detection and statistics applicable for non-normal distributions. One way to test for significance in non-normal distributions is the Wilcoxon rank-sum test [74], which asserts if two distributions (e.g. a baseline and an improved result) are significantly different. When dealing with features and individual effects, it is possible to perform a *two-way Analysis of Variance* (ANOVA) [75] which is able to determine significance of different features based on sampling perturbations. Strongly simplified, when a result can depend on features A, B and C, the ANOVA test draws random samples for each free and dependent variable and compares the resulting distributions.

The software engineering community has also developed unique metrics depending on their task. A good example is the `Top-X` metric used in recommender-systems [76], which is also applied in defect prediction [77][78] and fault localization [79][76]. Classifying a piece of code in faulty or non-faulty is a hard task, yet most classifiers will achieve good F1-scores, as almost all statements are non-faulty. As many algorithms optimize for F1 score, a different metric was necessary: Within `Top-X`, the tools (or models) predictions are listed in order of certainty, and the number of correct predictions within the first X elements are counted. In case of only a single true element, the `Top-X` can also be substituted with the `mean-reciprocal rank` (MRR). There are more task-specific metrics like the `BLEU-score` [80], derivatives like CodeBLEU [81] and many others. If applicable, they will be introduced and discussed in their respective chapters.

Some of the task-specific metrics are part of heated debates - especially the BLEU-score is under criticism [82][83][84]. This uncertainty constitutes the need for the last part of methodology: A qualitative evaluation. Statistics and metrics are very promising and necessary for large-scale evaluations, yet nothing can replace a human sanity check. As an example, many of the repairs produced by genprog [85] were not solving the relevant bug, yet they bypassed the test suite [86]. The authors of Defects4J [58] emphasized manual inspection of the bugs, fixes and tests [87] to assure quality. When applicable, results have been labeled by two authors independently, and conflicts of labels have been resolved with a short discussion. These initial labels, as well as their resolution and reason for agreement, were provided alongside the project artifacts. For large datasets, a significant subset of random elements was drawn. This lightweight qualitative assessment is based on existing program repair research[88][89].

1.6 Origin of Chapters

1. Chapter 2 is based on the short paper *Assessing robustness of ML-based program analysis tools using metamorphic program transformations* by Leonhard Applis, Annibale Panichella and Arie van Deursen, published in the *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2021). The version in this thesis is a currently unpublished extension.

2. Chapter 3 originated from a master thesis with Ruben Marang that got accepted as *Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML* by Leonhard Applis, Ruben Marang and Annibale Panichella at *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO 2023).

3. The Chapter 4 is an updated version of the dataset-paper *HasBugs - Handpicked Haskell Bugs* by Leonhard Applis and Annibale Panichella at *2023 IEEE/ACM 20th International Conference on Mining Software Repositories* (MSR 2023). There have been a few datapoints added since its original publication, and in addition a section accessing the type and themes behind the faults is added.

4. Chapter 5 was published by Matthías Páll Gissurarson and Leonhard Applis at *IFL 2023: Proceedings of the 35th Symposium on Implementation and Application of Functional Languages* as *CSI: Haskell-Tracing Lazy Evaluations in a Functional Language.*

5. Chapter 6 was published at *Proceedings of the 44th International Conference on Software Engineering* (ICSE 2021) as *Propr: property-based automatic program repair* by Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen and David Sands

6. Chapter 7 presents a submission to the *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium* (HASKELL 2024) under the title *Functional Spectra - Exploring Spectrum-Based Fault Localization in Functional Programming* by Leonhard Applis, Matthías Páll Gissurarson and Annibale Panichella. It is currently being re-iterated, as outlined in the chapters preface.

The chapters 2,3 and 4 are first-authored by me, while the works on Haskell (chapters 6, 7 and 5) share first authorship with Matthías Páll Gissurarson. For the shared works, Matthí focussed on the compiler side while my main responsibility lied in the experiments and evaluation.

1.7 Artifacts

Many of the scripts, programs and data in this thesis are available online.

1. The Lampion transformers for Python and Java are available on Github at `https://github.com/ciselab/Lampion` and a replication archived under `https://zenodo.org/records/7306931`
2. A containerized replication of CodeBERT is available on Github at `https://github.com/ciselab/CodeBert-CodeToText-Reproduction`
3. The CSI-Haskell experiments are available on Zenodo `https://zenodo.org/records/7307012`
4. The changes to HPC to trace evaluation are at `https://github.com/Tritlo/ghc/commit/62fa1edbe81d8942ce922d586d50c3f1f79ffca4` and proposed in `https://github.com/ghc-proposals/ghc-proposals/pull/539`
5. PropR is on Github `https://github.com/Tritlo/PropR` and the reproduction on Zenodo `https://zenodo.org/records/5389051`
6. HasBugs are available on Github `https://github.com/ciselab/HasBugs`, archived on Zenodo `https://zenodo.org/records/7569299` and online accessible on `https://ciselab.github.io/HasBugs/`
7. Tools and Experiments used in Chapter 3 are in `https://github.com/ciselab/Guided-MT-Code2Vec` and archived on Zenodo `https://zenodo.org/records/7307012`
8. The tasty-ingredient shown in Chapter 7 is in `https://github.com/Tritlo/TastySpectrum/` and an archived version under `https://doi.org/10.5281/zenodo.12168445`

They are also listed again in their respective chapters.

2 Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations

Summary

Metamorphic testing is a well-established testing technique successfully applied in various domains, including testing deep learning models to assess their robustness against data noise or malicious input. Currently, metamorphic testing approaches for machine learning (ML) models focus on image processing and object recognition tasks. Hence, these approaches cannot directly be applied to ML targeting program analysis tasks. In this chapter, we extend metamorphic testing approaches to ML models targeting software programs. We present Lampion a novel testing framework that applies (semantics preserving) metamorphic transformations on the test datasets. Lampion produces new code snippets that are semantically equivalent to the original test set but different in their identifiers or syntactic structure. We evaluate Lampion against CodeBERT, a state-of-the-art ML model for Code-To-Text tasks that creates documentation for given methods. Our results show that simple transformations significantly impact the target models behavior, providing additional information on the model's reasoning apart from the classic performance metric. We also prove a trend that some transformations affect Python and Java significantly differently. We further elaborate on a failed experiment attempt, and provide a post mortem advice for methodology and experiment design in the field of data-augmentation for SE ML.

Parts of this chapter have been published as *Assessing Robustness of ML-based Program Analysis Tools using Metamorphic Program Transformations* by Leonhard Applis, Annibale Panichella and Arie van Deursen in the *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2021)

2.1 Introduction

Artificial Intelligence(**AI**) has been applied to software engineering (**SE**) to address many tasks, such as fault localization [90], test case generation [91], fuzzing [92] or optimizing meta-parameters [93]. Recently, modern sequence-to-sequence deep learning models have shown promising results sparking new types of applications. Among them is the creation of code from verbatim description (*text-to-code*) [94], or the generation of documentation for source-code of previously unseen methods (*code-to-text*) [95, 96]. Yet, we argue that *it is unclear to which extent these models truly behave as intended*, apart from their reported accuracy. Applying testing strategies for machine-learning (ML) based program analysis solutions is critical — and we better do it early before we create untested ecosystems.

Recently there has been a surge in Testing ML, with goals beyond assessing accuracy (see the survey by Zhang et al. [97]). Many approaches have been taken from *classic* software testing and were adapted for ML. One example is metamorphic testing, a well-established technique that is considered a powerful approach as it addresses the *Oracle Problem*[98] in test generation. Metamorphic testing is successfully used in ML [99, 100] for image processing and object recognition. For example, image rotation is an information-preserving transformation as it alters the pixels in the image without changing its label (oracle). In computer vision, a *robust* ML model must not provide different predictions for the image altered with metamorphic transformations. Hence, quantifying the number of transformed images on which an ML model provides different answers quantifies its *robustness* against such transformations.

While extensive research has been conducted on metamorphic testing for vision computing tasks [99–101], the existing metamorphic transformations are domain-specific. Hence, they cannot be applied and do not hold for different domains and types of data. In this paper, we *extend the concept of metamorphic testing to machine learning models trained on and targeting source code.*

We define a set of transformations that alter features of code but yield the effectively equal program, such as introducing `if(true)`-conditions or +0 behind integer expressions. Using those, we modify the datapoints (programs) in order to detect differences in the models' predictions and metrics. We expect that the models are robust towards some transformations while others affect the metrics (negatively). The information gained could help to evaluate existing models, compare them to each other and provide suggestions and warnings for end-users and researchers alike. With our research and tool, we contribute to the following points:

1. A systematic approach, called Lampion , that used metamorphic transformations to quantify the robustness of ML models for code
2. Empirical evidence indicating that metamorphic transformations have a substantial effect on the performance of ML models for code
3. Identified challenges for the application of metamorphic transformations for data augmentation with sequence-2-sequence models

This paper is an extension of our preliminary study [102], which showed that even single transformations can have heavy impact on models. The added content consists of a cross-language comparison, the application and analysis of multiple transformations per datapoint (*higher order application*), a perplexity based analysis of the naturalness of altered code and an investigation of Lampion for data augmentation.

Our initial experiments on CodeBERT [94], a state-of-the-art ML model widely used in the SE literature [103–106], demonstrate the feasibility of the approach, and the type of lessons that can be learned from applying Lampion . After this initial assessment, we also want to investigate whether the tool can be utilized to act on the found issues: We conduct a second experiment reproducing the OpenVocab-Model [107], where we additionally try to improve the models robustness by augmenting the training-data.

The first experiment on CodeBERT shows that different (but not all) transformations affect models trained on `Java` and `Python` code. We also observe that various kind of transformations stimulate the models differently, and that multiple applications tend to have a stronger effect on the results. We provide a detailed insight into the variations in the BLEU scores and statistical tests to analyze the behavior of the models. There is room for debate on the discovered behavior and counter-measures — but exactly this data-backed debate is what we consider a primary contribution of our approach.

The second experiment on OpenVocab showed that data augmentation can quickly hit computational limits and other pitfalls. We discuss the experiment and dissect the problems, so that other researchers can take them into account.

The remainder of the paper is structured as follows: In Section 2.2 we summarize related work on metamorphic testing, explainability and documentation generation. Section 2.3 explains the Lampion approach and the individual transformers. The explicit research questions and their methodologies are covered in Section 2.4.1, with the results presented in Section 2.5. The issues encountered when applying Lampion to OpenVocabCodeNLM are presented in Section 2.6.4. The work is closed by a discussion in Section 2.6 and a conclusion in Section 2.7. All code artifacts, results and reproduction packages are linked in Section 2.8.

2.2 Background and Related Work

2.2.1 Metamorphic Testing

Metamorphic testing is a technique based on the concept of *metamorphic relations*, a property-based technique that exploits known equality of certain output values. Prominent examples are programs that implement mathematical functions, which are known to have metamorphic relations. For example, the sine function has a well-known metamorphic relation: $\forall x \in \mathbb{R} : sin(x) = sin(x + 2\pi)$. With such a relation, testers can easily create new test cases to verify the sine function and prove that this property holds. If one can define such *strong* properties, a common approach is property-based testing (e.g., using QuickCheck [40]) that generates test-cases utilizing the given metamorphic relations.

The earliest work on metamorphic testing can be traced back to Chen et al. [108] for test amplification. Related work in the field applied this testing technique to numerical programs (e.g., [108]), web services and applications (e.g., [109]), computer graphics (e.g., [110, 111]), embedded systems (e.g., [112]), simulation engines (e.g., [113, 114]), bioinformatic (e.g., [115]), and compilers ((e.g., [116, 117]). A complete view of metamorphic testing studies and applications can be found in the survey by Segura et al. [118].

The popularity of metamorphic testing is due to its ability to address the oracle problem, which is one of the open challenges in software testing [98]. In general, a metamorphic test uses the following pattern: Given an existing test $t(x) = y$ and a metamorphic transformation $m(x) = x'$, a new *follow-up* test can be generated as $t(x') = y$. In case the metamorphic relation $m(.)$ can generate multiple metamorphic input datapoints, it can be used to generate multiple follow-up test cases. In case the results of the follow-up test cases differ from the original ones, the metamorphic relation is *violated.* This indicates that the program under test is defective [118].

Metamorphic testing is domain-dependent: Metamorphic relations that hold for certain domains (e.g., numerical programs) may not hold in different domains (e.g., image processing). Besides, this technique cannot be applied if no metamorphic relation is known for a given program and domain. Lastly, the generated test cases might not yield additional value, for example testing every number for the sine function is unlikely to be a valuable test suite. One important pre-requisite for the effectiveness of metamorphic testing is that it should make the execution of the follow-up test case as different as possible from the original test case [118].

It is worth noting that all aforementioned studies focus on metamorphic relations with the goal of generating new test input data for existing test cases. Instead, in this paper, we aim at generating new follow-up programs to assess and validate machine learning models trained on source-code snippets. While metamorphic testing has not been applied to ML models for SE, metamorphic transformations and relations are known in software engineering and are tightly coupled to *refactoring*, program *optimization*, and linting. There are many ways to express the same code, starting from varying indentations to effectively equal loop structures or simply different comments. Another common field where metamorphic transformations on code are used extensively is optimization, where a compiler creates more efficient code using techniques like loop unrolling or function inlining [119].

When it comes to images, metamorphic transformations may change what humans recognize in images, but when it comes to source-code the metamorphic relations we define in this paper produce program-variants that are provably equivalent to the original ones, without requiring human judgment.

2.2.2 Metamorphic Testing for Machine Learning

Metamorphic testing has been applied recently to machine learning, especially to image-based object-detection tasks [99]. A metamorphic transformation on images performs *information-preserving* alternations on a given image. For example, the image of a cat might be mirrored, yet a classifier should still be able to recognize it as such. Other such transformations include (slightly) blurring, rotating, or changing hue-values of images. These operations change the values of images significantly; nevertheless, they are still easily classifiable by humans. The research on image-based models splits into two main categories, either a *pure* approach of standalone ML models [100], or the use of the models in applications and cyber-physical systems [120], both of which employ metamorphic testing.

Another use of metamorphic transformations is testing robustness of a model, by generating more datapoints in the test-set [100], which we also emphasize in our experiments. Such approaches can also be applied to generate more training-data, which can result in a more robust or precise model [101]. This idea is picked up within our second experiment, but due to a change of domain we had to re-define robustness in our own terms. Apart from testing the models performance, Xie et al. [100] showed that metamorphic testing is useful to test the ML-frameworks producing models.

Similar work to this paper stems from Rabin et Al. [121] [122] which shares the same motivation and a very similar approach. Incidentally, the authors happened to work on the same topic in parallel. While this could *hurt* the novelty, we would like to stress that it is considered best scientific evidence if two scientists independently come to the same conclusions using different methodologies [123][124]. Within their work [121] they implement various transformers for Java, and they perform an experiment [122]. The primary differences (apart from the underlying experiments) is in the evaluation, where Rabin et Al. utilize a binary-migration metric that yields more insight on the type of change. Within this paper, we use statistic tests which are not that detailed, but can be used task-agnostic. Furthermore, we try to improve the model using data augmentation, which to the best of our knowledge has not been done in a cohesive approach of testing and hardening for SE Models.

2.2.3 Testing Code2Vec Models

For testing ML-SE models Yefet et al. [125] — the authors of Code2Vec — showed that they can generate adversarial attacks on Code2Vec-based classifiers by changing variable names or introducing new variables. They generate the attacks by choosing a desired classification label and then altering an existing piece of code towards the target label using gradient ascent. Two mutations used by Yefet et al. (introduction of variables or alternation of existing names) for adversarial attack generations share some similarities with two of the metamorphic transformations we present in Section 2.3. The main difference is that Yefet et al. [125] optimize for variables and identifiers that produce an adversarial attack. In our case, we consider identifier renaming as one metamorphic relation, which is utilized to change variable names with any random name, without searching for a specific different classification/result. The approach by Yefet et al. aims at generating names of common libraries or abbreviations, while the ones created in our approach are either fully random or not associated with programming-related meaning. Finally, Yefet et al. approach is white-box while our framework treats the model as a black-box. Black-box strategies have a wider application, for example if the model is hidden behind an API or if the model is intentionally secret (when a model is bought as a service).

Another related work is from Compton et al. [126] that introduces randomization of variable-names in the training-dataset as a way of training data augmentation. Their study shows that the model trained on the augmented training-dataset achieves slightly better accuracy than the model trained on the original dataset. Variable-renaming satisfies a metamorphic relation; hence, it is also included in our work, among other transformations. Their results on data augmentation motivates our second experiment, where we try to see the impact of different transformations. Compton et al. observed the same shortcoming of some models, which is a shared motivation with our work. While Compton et al. achieve an improvement in accuracy, this does not necessarily mean an actually more robust model — which is the key objective of our work.

2.2.4 Documentation Generation

One of the tasks covered during the empirical study is the generation of program documentation. Historically, documentation-generation has been interpreted as an information-retrieval task that tries to summarize key-elements of the text [127][128]. The often rule-driven approaches look for keywords and utilized symbolic execution [129] to cover program flow and exceptions, to produce (often mechanic) summaries. Modern approaches utilize deep learning [67][130] while interpreting the summarization as a translation task — it translates from a code-sequence to a sequence of human language. These models produce good readable summaries that are evaluated via proximity to human documentations, but unfortunately humans do not exactly provide a gold standard for documentation [131][132].

Evaluating summaries is a sophisticated topic [133], as there are many attributes we want for a summary: Conciseness, readability, information and coherence are just a few attributes. Most research that does not qualify as philosophy is centered around the gold-standard that is seen as the best performance. Within documentation generation, the data is usually mined from software repositories, and existing documentation is seen as the gold standard both for training and evaluation. The common metric used for documentation generation is the BLEU-Score [80] which was initially introduced for machine-translation tasks. The BLEU-Score first calculates a unigram-precision, ignoring token-position and factors in a brevity penalty. The penalty is necessary, as many algorithms would overfit on guessing a few tokens right, without producing actual sentences. Other simple options for metrics are *n-gram-overlap* or *cosine-similarity*.

A more complex option is *ROUGE* [134], which measures n-gram co-occurrence. ROUGE tends to be stricter in evaluation (the same summaries score worse in ROUGE than in BLEU) and covers the naturalness of sentences better. We see that ROUGE has not been adapted as, despite existing literature [135][136][73], code is still treated as widely different from natural language.

2.2.5 Code Completion

The other task covered in our study is *code completion*.

The primary task of code completion is to predict a (masked) token within a sequence of code. Code completion is likely the most prominent task of machine learning for software engineering that developers use every day, and is an integral component for every IDE. The manifold approaches for code completion range from rule systems [137][138], over search based approaches [139][140] to (deep) neural networks [135][94][141][142].

Research in code completion has also identified different variations to the problem: A common differentiation is code completion in *standard,dynamic* or *maintenance* mode [135]. Standard describes the procedure as above. Dynamic refers to an online-learning approach where the files seen from the current project are taken into account for ongoing prediction - it therefore requires a test-set separated into projects. The cache/model are reset after every project. The dynamic approach is intended to match the experience a developer has when they adapts code completion to a new project. Lastly, maintenance mode describes the case that the projects' files were incorporated in the training data and it is not considered *taboo* to introduce information or context of the data into training. These modes tend to yield better metrics as they provide the model with more information. However, the authors will focus on the *standard* metric. While the other modes lean towards everyday use by developers, we would like to focus on a technical side of machine learning and focus on clean and cross-task metrics to prove stronger generalizability of our approach.

Another differentiation is based on the input sequence. Some models (especially bi-directional model like CodeBERT [94]) require the whole sequence of a statement, which is considered insufficient for *live-usage* — when creating code, outside of maintenance, one usually does not have elements to the righthand-side of the token to complete. Also it might be argued that the concept of order in natural language does not hold with the rules and scopes of programming languages.

2.3 Our Framework: Lampion

2.3.1 Overview

Figure 2.1 depicts the metamorphic testing approach, we named Lampion , and designed for testing ML models trained on source-code programs. Lampion relies on the metamorphic transformations (MTs) defined in the subsections below. Our approach consists of three main steps. First, Lampion takes as inputs a pre-trained model and a program not used during the training process (items ⑤ and ① in Figure 2.1). Then, it generates program variants (item ④) by using our MTs (item ②) and based on a given configuration file (item ③). The configuration file specifies the type of transformation to apply and the number of repetitions (order). Then, the original program and its equivalent variants are fed to the pre-trained model. Finally, Lampion compares the outcome produced by the pre-trained model for the original program (item ⑥) and its metamorphic variants (item ⑦). If there is no difference in the outcome, it means that the model is *robust* to the metamorphic transformation. Otherwise, Lampion detects some weaknesses in the pre-trained model. The type of weakness or issue is related to the type of transformation with which the model generates different results. For example, suppose the pre-trained model produces different results when introducing redundant if-statement. In that case, it is not robust to AST modifications that do not impact the program behaviors.

In addition to generating metrics for inspection and analysis, one can formulate a hard test-case given the insights gained from the Lampion approach: "The models' output should not be significantly affected by trivial if-statements". Such test-cases can provide a comparable insight on the model's behavior apart from the performance metric, especially in early stages where the model is still in conceptualization.

Alongside this paper, we provide the metamorphic transformer and the evaluation as open-source artifacts. Apart from the metamorphic transformations shown in the sections below, further six MTs are implemented and more can easily be added. In the following sections, we highlight the basics of the approach, consisting of metamorphic relations for programs and the metamorphic transformations used in the case study.

2.3.2 Metamorphic Relation for Programs

The first step is to identify *metamorphic relations* for software programs, which are the data points for ML-based SE applications. For example, software modules are data points for defect prediction models, where the goal is to predict whether the modules are defective or not. In source-code summarization or documentation, ML models take as input code snippets and produce the corresponding natural language documentation. *Metamorphic*

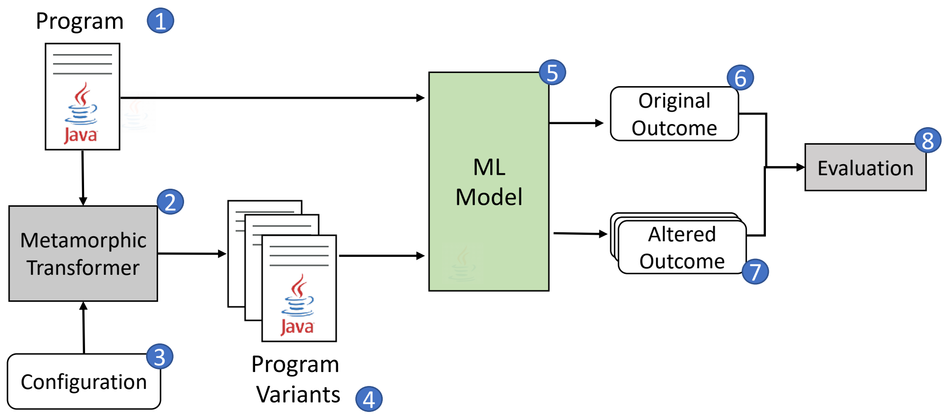


Figure 2.1: Lampion — Metamorphic Testing Framework for ML in SE Models

relations (MRs) relate multiple programs that differ in their structures (e.g., AST) but that are equivalent. As such, ML models should provide the very same output (e.g., defect prediction label) for programs that are related to one other according to an MR. Therefore, given a program P, we use MRs to generate equivalent yet different programs P'_1, ..., P'_k to test a given ML model under analysis.

Theorem 1 *Given an oracle function $O : P \rightarrow \{l_1, \dots, l_n\}$ where P is an input program and $\{l_1, \dots, l_n\}$ is the set of possible output labels (oracle); a relation $f(.)$ is metamorphic if it satisfies the following property:*

$$\forall P : O(P) \equiv O(f(P)) \tag{2.1}$$

In ML applications, the oracle function corresponds to the labels that humans would provide for a given program P. The type of label for each program (data point) is task-dependent. For example, in ML-based program documentation, the label (oracle) is the natural language description developers would write for the program P.

We identify two types of metamorphic relations for programs and that are useful to test ML models for program analysis:

MR-1: *Addition of uninformative code elements.* An uninformative code element (e.g., comments, un-used variables, un-used parameters, etc.) does not change the behavior of the target program P. As such, the label (oracle) for P and its variants with MR-1 relation remains the same. However, the different program variants are characterized by different ASTs.

MR-2: *Replace a code element with another equivalent element.* Equivalent program elements (e.g., different variable names) do not change the AST of the programs but the labels of the nodes within the AST. Using different yet equivalent elements does not change the behaviors of a program P either.

Table 2.1: Overview of metamorphic transformations for programs

Transformation	Short	Description	Estimated Effect	Variations
if-true	**MT-IF**	Wrapping a random expression in an *if(true)* statement	Structural Changes, introduction of conditions, introduction of keywords	if-false-else
add-unused-variable	**MT-UV**	Add a random unused variable	Introduction of names, introduction of types	Full random and pseudo random names (Postfix **R** & **P**), names looked up from a dictionary or the program under test
rename-entity	**MT-RE**	Rename a class, method or variable	Introduction of names, removal of known names	For variables, classes and member-types separate
lambda-identity	**MT-ID**	Wrap an expression in an identity-lambda function (including function call)	Introduction of complex structure, introduction of operators	-
delegation-method	**MT-DM**	extract an expression to a function, invoke the function instead of the method	Structural changes, change of scope for information, introduction of names	same as MT-UV
comment-alternation	**MT-CO**	Add,remove or move comments	Introduction or removal of natural language	Full or pseudo random comments generated
parameter-introduction	**MT-PI**	Introduce an unused parameter	Change of method signature, introduction of names, introduction of types	same as MT-UV
whitespace-alternation	**MT-WS**	Add or remove whitespace	Change of code-layout	-
add-neutral-element	**MT-NE**	Add the neutral element to a primitively typed expression	Change of structure, introduction of tokens	Complex equivalent transformations (e.g. replacing *true* with $0^1 == 1$)

```java
// Before
public void someMethod(){

    // Methodbody...

}
```

(a) Before

```java
// After
public void someMethod(){
    if (true) {
        // Methodbody...
    }
}
```

(b) After

Figure 2.2: Example if-true-transformation

2.3.3 Metamorphic Transformations

Given the two MRs defined above, we can define a set of *metamorphic transformations* that satisfy our MRs. A *metamorphic transformation* (MT) is a procedure that generates new programs $P'_1, \ldots, P'_k$ (*follow-up* programs) starting from an input program P and using a metamorphic relation. The mathematical definition for our metamorphic transformations breaks down into two parts: First, the oracle function must give the same output for the initial program P and the transformed program $f(P)$. Second, if P is a valid input for the ML model, then the result of the transformation function must be a valid P' for the model too. Hence, the transformation forms a *homomorphism* over the model domain.

In this paper, we choose MTs that satisfy the following properties:

1. The transformed program P' always compiles if the original program P compiles
2. The transformation can be applied any number of times
3. Different transformations are not mutually exclusive (e.g., addition and removal of comments void each other)

A list of the MTs is presented in Table 2.1. In the remainder of this section, we describe the transformations in more detail and elaborate on their purpose and how they affect the AST.

MT-IF: *Adding if-statements with tautologies.* The first transformation consists of adding trivial if conditions that are always true (tautologies). An example of transformation is shown in Figure 2.2. This transformation aims to alter the abstract syntax tree, adding nodes and keywords and increasing distances between tokens. A *good* classifier should be robust against trivial structure changes in the AST and be able to extract the relevant code patterns and features.

Note that the example in Figure 2.2 is simplified as it does not include the case where the transformed Java method does not include a return statement. For methods with a return statement, the transformation must add an else-statement returning a trivial element for the code to compile successfully.

MT-UV: *Adding unused variables.* Our second transformation adds an unused but declared variable, as shown in Figure 2.3. Many ML models for code analysis heavily depend on identifiers, class names, and datatypes. This is particularly true for Code2Vec-based classifiers, as explained in Section 2.2.3. This transformation comes in two flavors: Either the newly introduced variable has a fully random, alphanumeric name (**MT-UVR**),

```java
// Before
public void someMethod(){
    // Methodbody...

}
```

(a) Before

```java
// After
public void someMethod(){
    int raging_racoon = 3;
    // Methodbody...
}
```

(b) After

Figure 2.3: Example add-variable-transformation

```java
// Before
public void someMethod(){
    int a = 1;
    // Remaining Methodbody...
}
```

(a) Before

```java
// After
public void someMethod(){
    int a = 1 + 0;
    // Remaining Methodbody...
}
```

(b) After

Figure 2.4: Example add-neutral-element-transformation

or an English-like name (**MT-UVP**) is provided (such as *"raging_racoon"*). With these two options, we can assess whether the model is robust against unused variables in general and whether semantic information included in the names of the identifiers is relevant to the prediction. The type of the variable is chosen at random[1] from the primitive data types. Both the name and value of the introduced variable are randomly generated and depend on the type of the variable being injected.

MT-NE: *Adding neutral elements.* This transformation adds a neutral element to a given literal, such as arithmetic or conditional expressions. An example of MT-NE transformation is reported in Figure 2.4). To add a neutral element, the program under analysis needs to have at least one primitive typed expression (e.g., arithmetic operations). Hence, methods that either have no typed expressions (only method-calls without return values) or that only use complex data types, cannot be transformed.

This transformation changes the abstract syntax tree in a similar way to the MT-IF transformation, but without the use of control structures and reserved keywords. The transformation can be applied at any expression with primitive types, including conditions and return statements. Any primitive data type is supported, and if necessary, parentheses are added.

MT-RE: *Rename entities.* This transformation changes the name of entities, i.e., classes, methods, and variable names. This transformation does not impact the AST of a program P, but it changes the labels of the nodes in the AST. This transformation also comes in two flavors: (1) replacing a given name with a completely random alphanumeric string (**MT-RER**), and (2) using a pseudo-random, English-like name (**MT-REP**).

Depending on the language, there are many possible transformations. In functional languages, currying and un-currying are good examples for MTs that are not easily achieved

[1]A random seed is used for all occurring randomness.

in non-functional languages. Furthermore, there might be more options depending on the program's domain.

For a program, it is possible to change data-structures or sorting algorithms while maintaining functionality.

2.4 Empirical Study

2.4.1 Research Questions

We first want to assess whether the proposed MTs impact the performance of machine learning models. In an ideal case, ML models should not be affected by the metamorphic transformations, i.e., the model is not sensitive to changes that do not alter the code behavior. Hence, RQ1 should cover the general impact of applying one single transformation at the time, hereafter referred to as *first-order* MTs:

Research Question 1

To what extend do first-order metamorphic transformations affect the performance of ML models?

We further want to estimate the impact of applying the same transformation multiple times to a single code snippet, hereafter referred to as *higher-order* MTs. We would expect that there is a monotonic relationship between the order (i.e., number of transformations) and the changes of ML model performances. With RQ2, we want to analyze the general behavior of the model and gain insights on its robustness when the code *slowly decays*, e.g., by applying up to 10 times the MT-IF transformation. RQ2 aims to quantify the impact of applying higher-order transformations:

Research Question 2

What is the impact of higher-order MTs on the performance of ML models compared to first-order MTs?

We also want to compare the different types of transformations w.r.t the benchmark. We may expect that different transformations have different impacts on ML models. Furthermore, we aim to understand which model features are more robust, e.g., whether name-changes affect the model *more* than structural AST changes.

Research Question 3

To what extent do different types of MTs have a different impact on the performance of ML models?

An important point to consider is the *naturalness* of the produced code after transformations. There is a decent chance that the model is only dropping in metrics because the introduced tokens are out-of-vocabulary, or that the distribution is skewed. This is captured in the last RQ:

Research Question 4

How much do the MTs affect the naturalness, and can we observe relations between the naturalness of altered code and the performance of ML models?

As a follow up to these RQs, we wanted to investigate data augmentation in the field of SE tasks using Lampion . We hoped that we would either benefit on initial metric, robustness or both. Unfortunately, the experiments proved to were computationally unfeasable due to the behavior on augmented datapoints. We give a detailed analysis of the problems in Section 2.6.4.

2.4.2 Benchmark - CodeBERT

The used benchmark is *CodeBERT* by Feng et al. [94], a bimodal model trained on sequences of program language and natural language. It is publicly available in the Microsofts' reproduction suite *CodeX-GLUE* [67]. CodeBERT is trained using a pair of (1) a tokenized method and (2) the natural language (tokenized) documentation. The model is trained by masking tokens and generating possible solutions for the masked token. As a bimodal model, CodeBERT's generation capabilities can be applied to both Code-To-Text and to Text-To-Code tasks. For this paper, we focus on Code-To-Text since our metamorphic transformations apply to source programs.

CodeBERT has been trained on 6 programming languages with a total of 8.3M datapoints (code snippets) and achieves state-of-the-art results of an average BLEU4-Score of 17.65 in the CodeSearchNet-challenge [68]. The models provided by Microsoft come in two variations: (1) *cold-started*, that is without language-specific training, or (2) the model can be re-trained using language-specific data.

For this paper, the Java- and Python-specific CodeBERT models will be used. We could not reproduce the results for the cold-started model, but we retrained the model and achieved an average BLEU-Score of 17.64 on the uncleaned test set, which is very close to the results (17.65) reported by Feng et al. [94] for Java programs.

The Code-To-Text task addressed by CodeBERT is the generation of Java documentation on function-level given as input the source-code of a Java method. This performance of CodeBERT on this task is computed by comparing the original Java documentation (seen as the gold-standard) to the output (documentation) generated by the model [94]. The metric used by CodeBERT is the BLEU-Score [80], a common metric for translation tasks. For the BLEU-score, the gold-standard and the generated text are tokenized and grouped into n-grams; and these sets of n-grams are compared to each other. The BLEU-Score ranges from 1.0 (perfect translation) to 0 (not a single matching word or n-gram). There are different variants of BLEU-Score calculation, where BLEU4 promotes short translations. We use the BLEU4-Score to evaluate the research questions formulated in Section 2.4.1, following the methodology used by Feng et al. [94].

2.4.3 Methodology / Experiment Design

For the empirical study, we developed metamorphic transformers for Java and Python-Programs that work at the source-code level. In addition, we developed a set of benchmark-specific scripts that help to bridge the formats of the benchmark and the transformer. In particular, the CodeBERT model expects a `.jsonL` file, which contains the methods tokenized as well as the gold-standard tokenized; instead, our transformer requires `.java` files. The scripts help the conversion in both directions. The implementations can be found in Section 2.8.

We chose a set of transformer-parameters, such as the kind of transformation(s) applied, the number of transformations applied per code snippets, and the random seed of the transformer.

We trained the CodeBERT models as described in the official repository by Microsoft [67], using the standard-parameters given in the readme. The CodeXGLue repository provides both the instructions and scripts to train the model, as well as the training- and test-sets. The training-set has 164,923 datapoints (Java methods), the validation-set has 5,183, and both have been used as-is. The test-set has 10,955 datapoints and has been used to initially assess the model, and as starting points for the generated follow-up tests using our metamorphic transformations.

We had to remove a few datapoints from the initial test-set because they were either malformed or could not be compiled, or had encoding issues. More precisely, we had to (manually) remove 28 entries from Java and 162 from Python in total, which corresponds to 0.25% and 0.01% respectively of the test sets.

The cleaned dataset is provided for an easy reproducibility.

With these artifacts, we conduct our empirical study following the steps below:

1. The test-set is a `.jsonL` file which contains all code-snippets and their gold-standard; therefore, the first step was to extract each code-snippet and store them into `.java` / `.py` files.
2. Apply the MTs as specified in the configuration to a copy of the code-files generated in the previous step.
3. Once we obtain the follow-up code-snippets we convert them back to a `.jsonL` file, which is the input format for CodeBERT.
4. Run CodeBERT upon the `.jsonL` file from the previous step; producing a new set of source-code summaries.
5. Evaluate performance metrics (BLEU4) and similarity (Jaccard Distance) for the new summaries.

When a transformation is named *MT-A + MT-B* it means that multiple transformations have been applied. In this case, the type of MT to apply is randomly selected. Notice that the different transformations have the same probability of being selected.

To answer RQ1, we apply first-order MTs to all datapoints (methods) in the cleaned test set, resulting in a set of variant-code-snippets. We then re-calculate the BLEU4-Score for the variant-code snippets (metamorphic test cases) as well as for the original ones. This metric is a common translation metric, to compare the gold-standard documentation against the generated documentation [67]. In addition to BLEU-score, we want to look into the number of changes disregardful of their comparison with the gold standard. To this aim, we use the Jaccard-distance. In particular, let A and A' be two Java-doc-comments, generated before and after applying an MT, the Jaccard-distance is computed as follows

$$jacc\_distance(A, A') = \frac{|t(A) \cap t(A')|}{|t(A) \cup t(A')|} \tag{2.2}$$

where t is a tokenization-function. For tokenization, we used the implementation available in the CodeXGLUE repository [67].

To assess the significance of the differences in BLEU-Score achieved by the documentation generated by CodeBERT with and without metamorphic transformations, we use the Wilcoxon rank-sum test [74]. The Wilcoxon test is a non-parametric test; therefore, it does not make any assumption on data distribution. We tested beforehand whether the achieved results (e.g., BLEU scores) follow a normal distribution by applying the Shapiro-Wilkinson test [72]. According to this test, all data are not normally-distributed (p-values$<$ 0.01), thus, supporting our choice of using non-parametric tests. For both statistical tests, we used a 95% confidence level.

To better understand the impact of transformations, we grouped the code-snippets into four different categories: Setter, Getter, methods with a short gold-standard javadoc and others. Setters and Getters are methods whose gold standard contain "`set`" and "`get`" is shorter than 10 words; Short methods have less than 5 words in their gold standard that are not Getters or Setters; Others include all methods that do not belong to the previous methods. This allows us to analyze how much the BLEU-score changes for each category.

We also checked whether the type of method had an impact on the dependent variable, which is the BLEU-Score. To this aim, we used a two-way permutation test [75], which is a non-parametric equivalent of the two-way Analysis of Variance (ANOVA). We set the number of iterations of the permutation test to 10,000,000 to ensure that results did not vary over multiple executions of the procedure [75]. If the p-value is <0.05, it means the impact of the MTs on the BLEU-score significantly varies across the different types of methods. In other words, some methods are more sensitive to changes in the BLEU-Score than others, when applying MTs.

To answer RQ2, we apply the MTs multiple times per datapoint in the test-set, creating variant code-snippets for higher-order MTs. For the sake of this analysis, we applied each MT (and combination) 5 and 10 times, respectively. We recalculate the BLEU-Score for the higher-order-code-snippets and for the original code-snippet.

To assess whether higher-order MTs have a larger impact than first-order MTs, we use the Friedman test [143] and the post-hoc Nemenyi test [144]. The former test is the non-parametric equivalent to ANOVA, and it is used for assessing the statistical significance of differences between first- and higher-order MTs. The Friedman test shows whether there is a difference, but it does not specify ranks amongst them. To rank the impacts, we use the post-hoc Nemenyi test to perform a pairwise comparison. The Nemenyi test measures the difference across treatments (i.e., the order of the MT) by computing the average rank of each treatment across all datapoints in the test-set. A lower average rank means that a treatment changes the BLEU-scores more significantly.

Similar to RQ2, we answer RQ3 by grouping the existing results by type of MT. On the MT-groupings we also apply the Friedman and post-hoc Nemenyi test, to rank the MTs by their impact on the BLEU-Score. Transformations with lower ranking have a statistically more significant impact on the BLEU-Score than others.

To investigate the naturalness of RQ4 we follow an approach by Hindle et. al. [73] and fit `n-gram` models and utilize their perplexity to estimate the naturalness. For our particular experiment, we train on the training-data of CodeBERT [68] and apply our transformations to a randomly chosen subset of 50 entries of the validation set. In this fashion, we follow the same separation as in the experiments earlier - and CodeBERT cannot have seen the datapoints beforehand.

When applying the n-grams, it can happen that out-of-vocabulary tokens are encountered. The standard behavior of `ntlk` is to return an `math.inf` perplexity for such points. For reporting, we include those infinite perplexities for the median, but exclude them in the mean. The *renaming*-tranformers were only applied up to 50 times, as they run out of transformable elements.

2.5 Results

2.5.1 Performance impact of first-order MTs

Figure 2.5 shows the histogram of deltas in BLEU4-Score produced by CodeBERT before and after applying first-order MTs. On the righthandside it shows the histogram of jaccard distances (entries with changes) and on the lefthandside the resulting differences in BLEU Score. The observed changes show that many of the produced metamorphic tests fail; in other words, the model is not robust towards the first-order MTs.

Of 62970 produced documentations 13131 entries produced a non-zero delta in BLEU-Score (20.9%). The Python Model behaves flakier than the Java Model on that regard (28.3% in changes opposed to 19.3%). On average the difference in BLEU is 0.07, which resembles a difference of a few tokens.

While many entries have no deltas in BLEU-Score, there are many more methods with differences in the generated summaries before and after applying first-order MTs. This is highlighted by the results obtained with the Jaccard distance between the summaries generated before and after MTs. In total, there were Jaccard differences for 29.2% of entries. Hence, many summaries change when we apply the MTs, they just perform similarly in terms of BLEU-Scores to the gold-standard (e.g. missing the same key words).

To assess the statistical significance of the results, we run the Wilcoxon rank-sum test comparing the BLEU-Scores achieved by CodeBERT before and after applying the first-order MTs. The test revealed that the differences are statistically significant (p-value<0.01) for the code snippets with changes between pre- and post-trans-formations. [2] According to the permutation test, there is a significant interaction between the BLEU-Score of the transformed code-snippets and the method types (p-value<0.01). Applying Cliff's delta effect-size, we observed that indeed the Setters and Getters had a small effect-size (0.26 and 0.30), while short methods had a large effect size (-0.51), and other methods had a negligible effect size. This means that metamorphic transformations have a larger impact on short method-types over the others.

We also observed that on most metrics the Python Model performs less consistent - we have three primary possible explanations: ① The Python dataset is about 1/5 of the Java dataset ② the Python documentation has lower quality and/or is heavier impacted by preprocessing ③ for the Python code, some of the introduced patterns are rarer than in jung2021commitbertava (e.g. lambda-functions). We expect the dataset size to be the primary factor, however we also observed big issues with the quality of the gold-standard documentation (see Section 2.6).

[2]The changes over *all entries* are not-significant, as many elements did not change in first-order MTs.

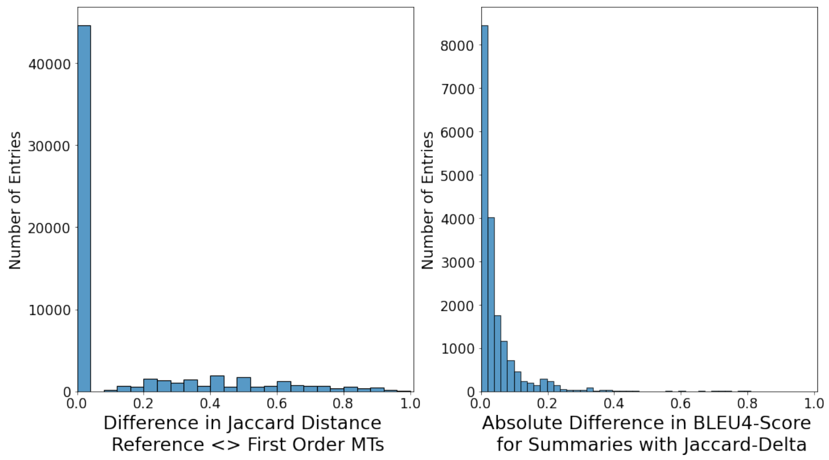


Figure 2.5: Overview of changes for first-order MTs

Summary RQ1

When applying first-order MTs, 20.9% of methods in the test set have a non-zero difference in BLEU-Score, and 29.2% have a non-zero Jaccard distance. Since the differences are statistically significant, our results suggest that CodeBERT is not robust towards first-order MTs.

2.5.2 Performance changes with higher-order MTs

The number of code snippets in the test set for which CodeBERT generates different summaries before and after applying MTs increases with higher-order MTs. In particular, 18730 out of 62970 (29.7%) code snippets have different summaries with first-order MTs, compared to 31044 (49.3%) when applying 5th-order, and 34225 (54.4%) when applying 10th-order MTs. The impact of changes for higher-order MTs are presented in Section 2.5.2 and Figure 2.6.

On average, the BLEU-Score decreases by 1.2 (absolute difference) for 10th-order MTs. A particularly strong difference is obtained when applying MT-RER+MT-UVR, which reduces the BLEU-Score down to 16.2. Putting this into perspective, Feng et al. [94] mention that RoBERTa —a model that has not been trained on source-code— achieves an average BLEU-Score of 16.47. Hence, after 10 transformations per datapoint, CodeBERT sometimes performs worse than a model that has never seen code.

A practical example can be seen in Table 2.3, which shows the Java model's behavior for a rising order of MT-IF applications. We see that after the first-order MT-IF, the generated summary becomes significantly shorter compared to the reference summary (i.e., without MTs). When the order increases to 5 and 10, the length of the summary grows. We also observe that the generated summaries vary greatly in their information with rising orders.

An overview of the effect strength is in Figure 2.7e. In short the MTs have an statistical

Table 2.2: Development of BLEU-Scores for multiple MTs by Language

	Base	MT-IF			MT-NE			MT-REP MT-UVP			MT-RER MT-UVR			MT-L		
MTs		1	5	10	1	5	10	1	5	10	1	5	10	1	5	10
Java	18.0	17.6	17.4	17.4	17.7	17.6	17.5	17.2	16.8	16.8	17.2	16.8	16.7	17.5	17.2	17.2
Python	17.0	17.1	16.7	16.3	16.5	14.6	13.6	16.5	15.0	13.9	16.4	15.1	14.1	17.3	16.8	16.6
Total	17.8	17.5	17.2	17.2	17.5	17.1	16.8	17.1	16.5	16.3	17.1	16.5	16.2	17.3	16.8	16.6

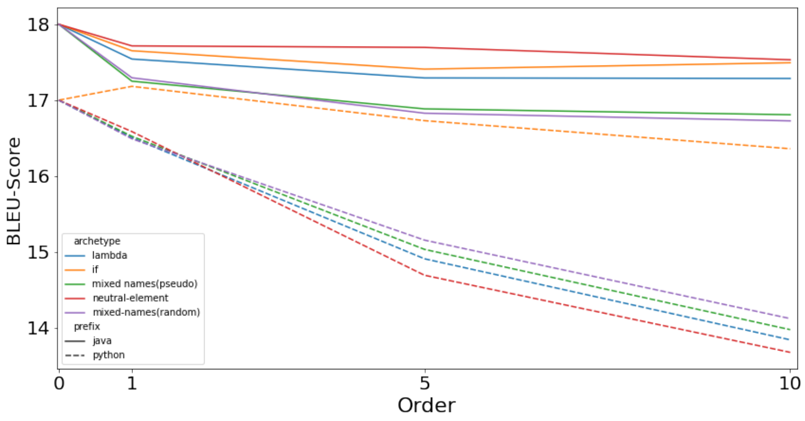


Figure 2.6: BLEU4-Scores for increasing number of metamorphic transformations (applied n-times per datapoint)

Table 2.3: example results for higher order MT-IF application (in Java)

Config.	**#Order**	**Result**	**BLEU-Score**
Gold	—	Un capitalizes the first character of the word given . It will convert i to I	1.000
No MT	—	Capitalize the first character of the first word .	0.223
MT-IF	1	Uppercase character	0.000
MT-IF	5	Uppercase character .	0.002
MT-IF	10	Capitalize the first character of a word .	0.140

Table 2.4: Example of CodeBERT output for a single method under different transformations

Configuration	#Order	Result
Gold	–	recipe for makeAFishyDecision with the numberOfFish
No MT	–	Make sure that the given number of non - empty words .
MT-IF	1	Make sure that the given number of fights in the given image .
MT-IF	5	Make sure that the given number of non - empty words should be displayed .
MT-NE	10	Make sure that the given number of fights with the given number of functions .
MT-UV + MT-RE	5	Make a fancy message .

significant impact(see above), but applying 10 MTs has a significantly bigger impact (effect strength) than applying less. This matches with our expectations - more changes in code lead to more changes in outcome. Further elaboration of the tests presented by Figure 2.7 is located in Section 2.5.3.

Summary RQ2

Higher-order MTs produce more changes and have a statistically significant higher impact on the BLEU-Score. This means that CodeBERT is less robust against higher-order MTs than towards first-order MTs.

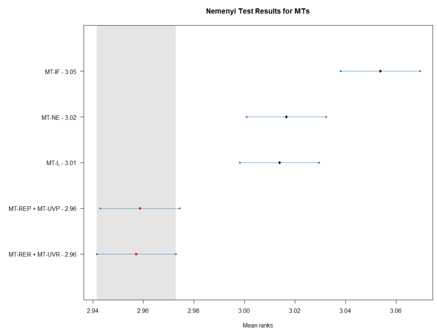
2.5.3 Comparison of Changes per Transformer

The average BLEU-Scores achieved by CodeBERT when applying different types of MTs can be seen in Section 2.5.2 and Figure 2.6. We can observe that, on average, there are strong differences per MT and per language, e.g. adding neutral elements seems to be negligible for Java models but has a high impact on Python. Changing parameter names and adding unused variables (MT-RE & MT-UV) seems to affect performance for both languages. These trends persist over higher-order MTs, i.e., when applying the same type of MT multiple times.

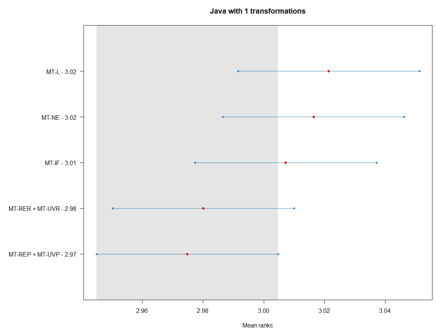
One particular example that struck out was a code snippet in the test set that contains a kids tutorial on learning Java switch-case-expressions[3]. In the example, the method performs a switch-case over an integer to display a *FancyMessageBox* with up to two fishes. A subset of the summaries generated by CodeBERT with different MTs for this example can be seen in Table 2.4.

First, we can notice that there are differences between the summary generated on the original code-snippet compared to the ones created after transformations. In two of the

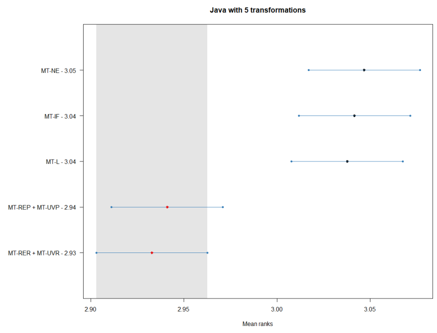
[3]Found under `https://github.com/TeachingKidsProgramming/TeachingKidsProgramming.Source.Java/blob/master/src/main/java/org/teachingkidsprogramming/recipes/completed/section07objects/WhichFish.java`



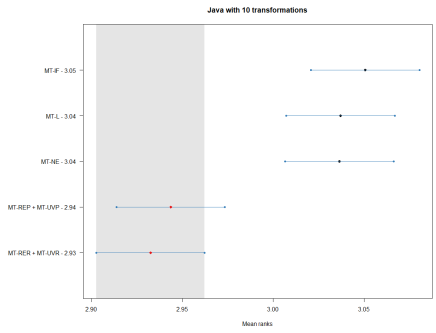
(a) MTs over all Orders



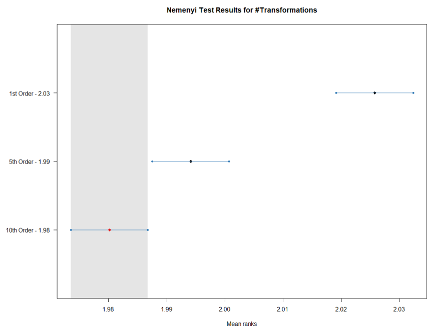
(b) Java T = 1



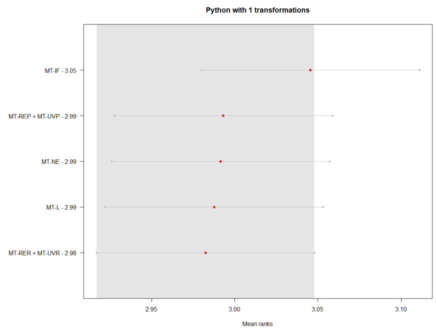
(c) Java T = 5



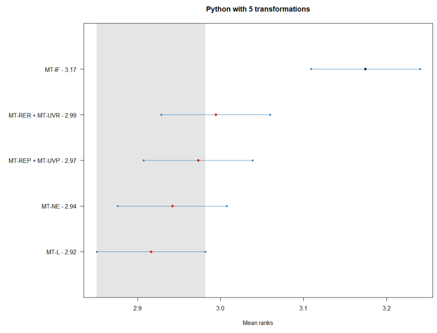
(d) Java T = 10



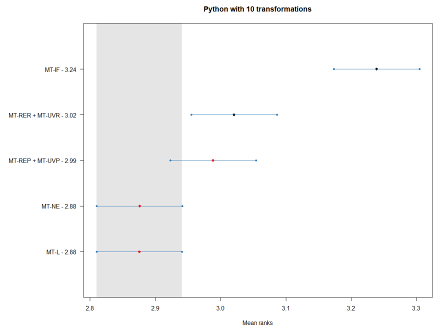
(e) Order over all MTs



(f) Python T = 1



(g) Python T = 5



(h) Python T = 10

Figure 2.7: Overview of Post-Hoc Nemenyi Results on Significance of MTs and Order

presented summaries, the word "fights" appears, despite there is no such a token in the post-MT snippets nor in the inline-comments.

For MT-UV+MT-RE, we observe substantial differences compared to the other summaries. The model generates completely different tokens, except for the word "`make`". However, it can be argued that this is actually a meaningful summary, as the code is mostly about displaying items in a *FancyMessageBox*. This summary is more accurate than those about counting non-empty words obtained with other MTs.

Another general trend we can observe is that certain keywords appear or disappear over the transformations. One example is "`non-empty`" keyword, which appears in the reference summary, disappears for MT-IF first-order, and re-appears for MT-If fifth-order. Hence, the model does not provide stable results, although all variants of the program are equivalent according to the MRs.

We applied the Friedman test and the post-hoc Nemenyi procedure to analyze the impact of the different MTs on the BLEU-Score. With a p-value<0.01, the Friedman test indicates a statistical difference across the different types of MTs. A summary of the statistical tests can be seen in Figure 2.7, which presents an overview of the Nemenyi Posthoc Ranks of different configurations; The primary observation is, that for the Java and Python different MTs achieve different rank in their effect. Noteworthy are the movements in Python — while for low transformations all MTs have an effect, with growing transformations the MT-L and MT-NE have significantly higher effect than the other MTs. For Java we find a similar movement but Renaming and Adding Variables are the dominant MTs. Figure 2.7a and Figure 2.7e show the nemenyi results before grouping by language and order. MT-RE & MT-UV are the dominant MTs overall, and the highest effect is produced by applying 10 transformations.

We further applied the two-way permutation test to assess whether different co-factors also impact the BLEU-Score achieved by CodeBERT when applying our MTs. In particular, we considered (i) type of method, (ii) type of MTs, language of the model (iii) ,length of the gold-standard[4] (iv), and (v) order of the transformations as co-factors that could play a role in decreasing the BLEU-score when applying our MTs. The achieved results indicate that the language, type of transformation and method type have significant impact on their own (p-value<0.05). From the combined multifactoral analysis, the significant combinations are {MT,number of transformations}, {MT,language} and {MT, language, number of transformations}. This confirms our results for RQ1, where we observed that getter, setter, and short methods are more affected by our MTs w.r.t. the summaries generated with CodeBert.

Outside of the proven significances, we would also point out some of the results that are not statistically significant for the outcome: the complexity of the gold-standard turned out to be un-important in the ANOVA-analysis, and the number of transformations is only relevant in regard of language and MT.

[4]we took the length of the gold-standard as a proxy of the complexity of the datapoint — short summaries are easier to guess

Summary RQ3

Different MTs have a statistically different impact on the BLEU-Scores achieve by CodeBERT. Adding unused variables (MT-UV) has the strongest impact, while adding a redundant if-statement (MT-IF) has the lowest impact.

2.5.4 Naturalness of Transformations

The achieved results for the bi-gram perplexity are shown in Figure 2.8, separately for Java and Python. Three- and Four-grams yielded comparable results, with the overall perplexities being lower but resulting in the same rankings.

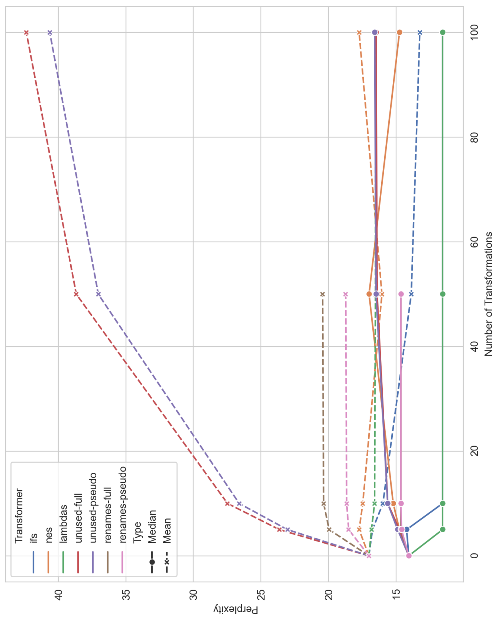
For Java, we see mostly expected trends in Figure 2.8a: The introduction of new, unused variables with random or pseudo-random names increases the perplexity. It shows a higher change for fully-random identifiers than for pseudo-random ones, which we expect to be based on the way the tokinezation is conducted. For CodeBERT, words and identifiers are split into sub-tokens, and the pseudo-random identifiers use some elements that are likely within the vocabulary. The renaming increases the perplexity and then *stagnates*, because it ran out of transformable identifiers. We also see a great disparity between the median and the mean, indicating that for most elements a low perplexity is returned, while some tokens are outliers with a high perplexity. We also see some transformers reducing the overall perplexity: Lambda- and If-Transformations produce lower mean and median perplexities. This is due to a majority of the introduced tokens being brackets, which are ubiquitous and have a low perplexity.

In Python, we observe similar trends with the exception of *neutral elements* increasing perplexity the most. On inspection, the changes consisted mostly of empty strings and zeros added, as well as brackets. An educated guess is that, for the most part, the Python code does not have the redundant +0 which leads to a high perplexity when encountering these two tokens after each other. It's important to note that the Python perplexity rises drastically more than the Java perplexity. Both have an initial average of ≈ 15, but after 10 transformations the altered Python results in a mean of 60 to 160, more than three times of it's Java counterparts.

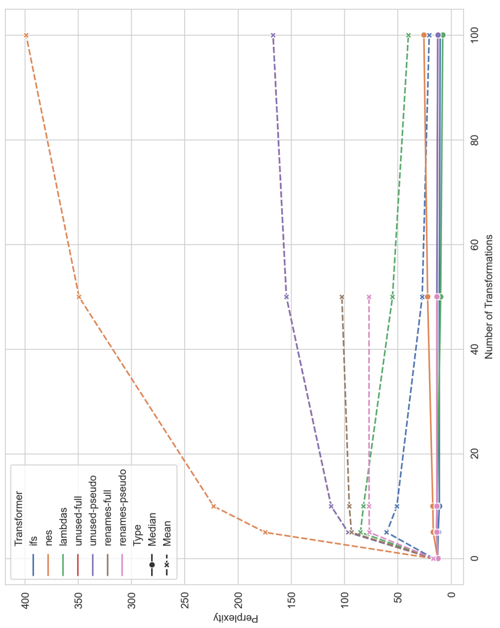
Generally speaking, the trends from Figure 2.8 are inverse to the reduction in BLEU-score seen in Figure 2.6. Transformers that produce a low change in perplexity (neutral element) for Java, also produce a low change in BLEU. The same neutral elements drastically change the perplexity for Python, and results in the biggest delta of BLEU. Based on the selected sample, it seems there is a direct correlation between the deltas of perplexity and BLEU.

Summary RQ4

Transformations that yield a higher average perplexity (i.e. introduce elements that are unnatural to the language-model) reduce the BLEU score the most.



(a) Bigram Perplexity for Java Transformations



(b) Bigram Perplexity for Python Transformations

Figure 2.8: Development of BiGram-Perplexity with higher-order Transformations

2.6 Discussion

2.6.1 Practical implications

We presented an effective approach for testing the robustness of a model towards metamorphic transformations on source-code. According to the empirical results, our approach is capable of producing significant changes in the summaries generated by CodeBERT, highlighting potential weaknesses in the model as it does not satisfy metamorphic relations. In other words, slightly different variants of the same program have vastly different results. While in this paper we focus on the Code-To-Text tasks of CodeBERT, we expect the found implications to hold true for other down-stream tasks as well. This can be considered a call-to-arms for researchers and practitioners to test machine learning models trained on source-code using metamorphic testing in addition to the traditional performance metric (e.g., accuracy). Our metamorphic transformations can be applied for other ML-based SE tasks that process source-code. However, the set of transformations can be extended and adapted to the domain and specific task under analysis. As part of future work, we try to apply the approach to other tasks such as defect-prediction, ML-based smell-detection and predictive mutation testing. Metamorphic testing can also be used to increase the size of the test-set by generating new program variants, without requiring human labeling. This could be potentially beneficial for SE tasks where labeling data is very expensive or few datapoints are available — or harmful, by introducing unrealistic or uncommon patterns that might not represent the *ground truth*.

2.6.2 Runtime Overhead

It is worth noting that generating follow-up code-snippets using our MTs incurs very negligible runtime overhead, in the order of 7 milliseconds per MT per code-snippet. However, it will result in having more datapoints in the test set, multiplying the runtime of model-inference. For CodeBERT, if one code-snippet leads to having 10 different variants using MTs, it will result in ten-times longer runtime for model-evaluation.

For our experiment, a single run of inference for the test-set took 13h, which had to be re-done for every MT configuration, resulting in another 13h wait time. While this has to be done only once for model evaluation, the time for inference still grows linearly. It might be interesting further work to filter for only promising MTs, similar to what is done for sampling in mutation testing [118] — we pursue this in Chapter 3 by introducing genetic search to find MTs that maximize the delta in performance metrics.

2.6.3 Reflection of the initial experiment

Lastly, we would like to highlight that engaging intensively with the original study like we did also suggests some improvements to the original setup. One concern is that the BLEU-Score has still very forgiving results, despite vague or uninformative summaries. For example, if both summaries just have matching words like *"the"*,*"a"*,*"an"* etc., it will still return a decent BLEU-Score.

Another issue is the dataset used, as not all of the data is up to *production* standard. We found examples of tutorials-code for kids books and code written in german. Hence, a stronger filter on the test-set should be applied, e.g., filtering out Getters, Setters and similar noise. Also, the preprocessing should be done differently, as the current Javadoc get cut off after the first @ token. As many code snippets contain (valuable) text in or after the parameters, this information should be preserved.

2.6.4 Open Challenges for Data Augmentation

We tried to apply Lampion for data augmentation, but we faced some serious computational problems. We reused the model and benchmark produced by Karampatsis et al. [107] in their work *Big code!= big vocabulary: Open-vocabulary models for source code* for code completion. They introduce the concept of Bytepair-Encoding (BPE) to SE-tasks in machine learning, achieving comparable results to SOTA approaches. OpenVocabCodeNLM was trained on different benchmarks, one is a publicly available py150k benchmark [145] consisting of 150 000 Python files mined from GitHub.

Starting from a reproduction, we used Lampion to augment the training data by copying datapoints and transforming them. The resulting dataset would have the original datapoints plus additional *noisy* datapoints that contained smells, obfuscated names or redundant structural elements. However, when we set out to perform training on the augmented dataset, we encountered a series of errors related to the memory of GPUs. For the data augmentation, we expected that doubling the data would result in a doubled training time. The general assumption was that augmentation would result in a linear growth.

This is wrong for two reasons:

1. OpenVocabCodeNLM is a sequence-learning task. Due to model architecture, the computational cost growths exponentially with the sequence length. Adding redundant elements (even after encoding) increases this sequence length, leading to exponential growth.

2. The training of OpenVocabCodeNLM tries to minimize the perplexity of all tokens in the provided sequence. The perplexity measure used in the experiment (and commonly used in NLP tasks), also growths exponentially with the sequence-length.

Additionally, many of these effects are not compensated by the BPE: The BPE is designed to cover frequently co-occurring tokens in a corpus, but many of the changes we introduce are either exotic (fully random names) or short and un-common (adding *+ 0*). For every 1 token we introduce with our transformations, 1 or more tokens are added into the final sequence after encoding. By accident, BPE amplifies the computational problems.

It is possible to re-design experiments by limiting transformations to those that do not alter token-amount too greatly. Due to the current scope of the work, this remains future research. In general, this forms a limitation for data augmentation approaches regardless of Lampion , and motivates to consider real-world constraints of the model architecture when designing the next methods. While Lampion can be applied for test-data augmentation, it should be carefully used for training-data augmentation.

The failing experiment and datasets are provided for reproduction.

2.7 Conclusion

This paper introduces metamorphic relations to test ML-models program analysis solutions. Using this technology, our objective is to gain further information on the model's behavior apart from the performance metric (e.g., accuracy).

To achieve this, we presented a generic approach (Lampion) and applied it in a case study on CodeBERT's Code-To-Text tasks. To evaluate the case study, we perform various statistical tests to prove or disprove changes in the resulting performance metric.

Our approach and framework can empower experts and laymen alike to assess the robustness of their models and provide additional tests on quality.

We tried to keep the approach ① lightweight in concept,② expendable functionality (due to plug-in MTs), ③ independent of the task (any language and quality metric).

While our initial implementation is in Java and Python, we expect that a re-implementation for any language is an easy task and the statistical analysis can be reused for most experiments.

2.8 Online Resources

The code for a sample metamorphic transformer, the grid experiment and the evaluation can be found on Github under the Lampion repository[5]. The model,cleaned test-set and post-transformation datasets can be found on Zenodo under DOI:10.5281/zenodo.6400572 . We provide CodeBERT[6] and OpenVocabCodeNLM[7] as containerized reproduction packages for other researchers.

[5]https://github.com/ciselab/Lampion
[6]https://github.com/ciselab/CodeBert-CodeToText-Reproduction
[7]https://github.com/ciselab/OpenVocabCodeNLM

3 Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML

Summary

More machine learning (ML) models are introduced to the field of Software Engineering (SE) and reached a stage of maturity to be considered for real-world use. But the real world is complex, and testing these models lacks often in explainability, feasibility and computational capacities. Existing research introduced metamorphic testing to gain additional insights and certainty about the model, by applying semantic-preserving changes to input-data while observing model-output. As this is currently done at random places, it can lead to potentially unrealistic datapoints and high computational costs. With this work, we introduce genetic search as an aid for metamorphic testing in SE ML. Exploiting the delta in output as a fitness function, the evolutionary intelligence optimizes the transformations to produce higher deltas with less changes. We perform a case study minimizing F1 and MRR for Code2Vec on a representative sample from `java-small` with both genetic and random search. Our results show that within the same amount of time, genetic search was able to achieve a decrease of 10% in F1 while random search produced 3% drop.

This chapter was published as *Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML* by Leonhard Applis, Ruben Marang and Annibale Panichella at *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO 2023).

3.1 Introduction

Producing a good model is hard. Not only is achieving good metrics often quite a challenge itself, but once entering the *real world* new problems emerge: Fairness [146], extrapolation [147], speed [148], security and robustness [97] are important non-functional requirements when it comes to machine learning (ML). And while good metrics make it into academic publications, poor non-functional qualities make the news [149]. In the realm of programming languages, ML applications have unique benefits, such as a large body of data publicly available in version control systems like GitHub or GitLab and discussion forums such as StackOverflow.

Unlike other domains, SE tasks have well-defined problems (e.g., code completion, test generation) and metrics (e.g., code coverage) that can be tackled with ML. Still, previous research shows that even in *clean* domains like software engineering problems with robustness and performance exist [102, 122, 125, 150]. How is that?

We argue that one missing piece is the lack of tools for expressing non-functional requirements as actionable tests. A meta-survey on Requirements-Engineering for ML [151] found that a considerable amount of publications are aware of non-functional requirements, but few go beyond defining the problem. Hence, we have a rich concept of requirements, but any good requirement must be expressed as a (repeatable) test.

Within this work, we target **robustness** of ML models as an exemplary non-functional requirement. Robustness expresses the ability of the model to perform reliably when facing noise and degrading data quality. Such noise in code consists of poor naming-standards, unused elements and redundant structures. As the rules for programming languages are well-defined, we can produce noise while keeping the program identical in behavior using metamorphic transformations. Metamorphic transformations utilize metamorphic relations to generate new alternative datapoints for which a ML model *should* give the same prediction/classification outcome. A *robust* model is capable of detecting redundant elements and stay mostly unaffected by variable names. Robust models do not come for free and existing research shows that Code2Vec[152] is affected by metamorphic transformations; even just renaming variables can completely change its outcome [125].

Assessing robustness as part of quality is the job of a tester, adapted for working on machine-generated models instead of code written by human developers. We argue that a non-functional requirement like robustness can be expressed using a statistical test which is explainable in layman terms. Similar approaches have been done by previous research [102, 122, 150], which are limited by *blind* (random or stacking) application of transformations. To implement the test, we use a search technique (genetic algorithms) in combination with metamorphic transformations. Introducing evolutionary intelligence ought to deal with these two crucial limitations, realism and computational efficiency. We aim to produce datapoints creating similar deltas, while requiring less computation and enabling more *realistic* datapoints. The created datapoints can be saved and re-used, to re-evaluate the model, forming an acceptance test.

The contributions of this paper can be summarized as follows:

1. Formulation of metamorphic testing as a search based problem within the SE4ML domain
2. Expand existing metamorphic testing within the domain with genetic search

3. Assess differences between random and genetic search w.r.t. generating effective metamorphic tests that affect code2vec

4. Sound statistical analysis based on multiple experiments utilizing genetic search, in particular for repeated transformations

Our results show that genetic search performs significantly better in reducing F1 score than random application of transformers. Based on the assumption that less transformations yield more realistic input-data, genetic search produces the same statistical difference in metrics with less transformations which we consider an increase in realism. Aligning with existing research, applying more transformation leads to bigger differences, which is further amplified if *the right* transformations are kept through genetic selection. An inspection of the most dominant transformation showed that especially those which add elements to the abstract syntax tree (AST) prevail, which we argue is connected to Code2Vecs' mechanisms based on AST-traversal.

3.2 Background & Related Work

3.2.1 Code2Vec & Method-name Prediction

Our experiments reuse the approach and artifacts[1] of Code2Vec [152] by Alon et al. Code2Vec is based on AST-path extraction for which code is translated into an AST and sampled into triples of `[leaf,path,leaf]`. These triples are merged into an vector-embedding per method, which shows promising results for the task of method-name prediction, and in particular their embeddings behave similar to generic nlp-embeddings. An important detail for this work is that the paths are extracted by performing random walks (default 200), limited by some further constraints (e.g. maximum width and depth). Due to the walks, the structure of the AST has paramount effect on the embeddings, as a single new leaf node doubles the amount of possible paths. Not only can the introduction of new nodes drastically change the amount of available paths, but also changes in the AST can lead to exclusion of previous valid paths. Hence, manipulation of the AST can lead to significant changes for better or worse - both important information as well as noise can be left out either by exclusion criteria or by chance.

Method-name prediction is a research area [153] where for a given method-body, a descriptive method-name is wanted. Descriptiveness is measured as F1-score by calculating sub-word token overlap of produced and actual method-names. Code2Vec outputs method-names with corresponding certainties, suitable to evaluate mean-reciprocal rank (MRR) that is based on the rank at which the correct prediction was placed. Within Code2Vec, the MRR was reported but only the F1-score was used for model training.

In addition to their work, Alon et al. provide Java-datasets and we use *java-small* for our experiments. We sample 350 of the ~6000 files, which satisfies 95% significance at 5% error rate.

[1]Note: We use a fork with minor changes due to machine dependencies `https://github.com/ciselab/code2vec`

3.2.2 Metamorphic Testing

Metamorphic testing is based on the concept of *metamorphic relations*, relations that generate new test inputs that should result in the same test outcome. Humans as well as tools can easily create new test cases based on these relations. An overview of the metamorphic testing landscape can be found in a survey by Segura et al. [118]. While metamorphic testing is not yet widely adopted to machine learning for software engineering (**ML4SE**), both transformations and relations are known in SE and are well explored for *test case generation*, *refactoring*, program *optimization*, and linting.

Metamorphic Testing for ML. Metamorphic testing has gained popularity in machine learning, particularly in image-based object-detection tasks [100][99]. These transformations on images apply *information-preserving* changes to images: The picture of a dog can be mirrored, but a model should still be able to classify it as such.

Researchers have adapted and evaluated metamorphic testing to ML models in the ML4SE domain, i.e., to models that aim to semi-automate SE tasks. Compton et al. [150] introduced obfuscation techniques for variable names and showed how `Code2Vec` models are vulnerable to variable name changes. Our underlying framework and approach share similarities, which we extend by adding search algorithms. The combination of existing research (transformers for models of code), search algorithms and statistical tests forms the unique novelty of this paper.

Yefet et al. [125] proposed a further metamorphic relation introducing unused variables in addition to variable-name obfuscation. Their study on `Code2Vec`-based classifiers showed how these simple code-snipped transformations could generate adversarial attacks that fool the model under test. While this forms a search using metamorphic transformations, our approach differs in three primary aspects: ① we use Code2Vec as a black-box model, ② we approach search as a quantitative task on multiple (many) datapoints, instead of creating single counter-examples and lastly ③ we target robustness as a quality attribute, and not security.

Cito et al. [154] generated "*counterfactual examples*" to assess the robustness of BERT-like models. Although they use different terminology, these examples are generated by applying perturbations to initial code-snippets by replacing tokens with plausible alternatives that do not alter the code's behavior; hence, these transformations are metamorphic. Their study showed how transformations could find counterexamples for BERT-like models that are in line with the rationale provided by human experts.

A more extensive list of metamorphic transformations for ML-models applied to source code has been introduced in *Lampion* by Applis et al. [102] and seen in this dissertation at chapter 2 in table 2.1. In particular, Lampion considers multiple different metamorphic transformations, which either add unused information (e.g., add unused variables, input parameters, and wrap an expression in identity-lambda functions) or replace a code element with another equivalent element (e.g., rename a class, method or variable). Their study investigated the extent to which different transformations affect the performance of CodeBERT [94].

Limitations. Despite these undisputed advances, metamorphic tests for ML models are created in ML4SE by using random sampling. In particular, given a set of possible transformations, existing approaches randomly select and apply these transformations (one or more times) until the model under test produces different test results (e.g., wrong classification). In this paper, we proposed the use of evolutionary intelligence (and evolutionary algorithms in particular) with the goal of ① leading to model miss-prediction faster via intelligent search and ② reducing the number of transformations needed to do so.

3.2.3 Genetic Algorithms

Search-based software testing (SBST) relies on search algorithms to seek for solutions to software testing problems. Since the 1990s, researchers have proposed and applied different meta-heuristics to optimize various testing problems, such as test case generation [155–157], regression testing [158, 159], and mutation testing [160]. The most applied meta-heuristics in the SBST literature include but are not limited to hill climbing [155], simulated annealing [161], and genetic algorithms [85, 157, 162]. Previous work has also shown how evolutionary intelligence can outperform random sampling (or random search) in specific testing applications [163–166], thus motivating our idea to use evolutionary testing for metamorphic testing.

Genetic algorithms (**GAs**) [167] are a group of search techniques inspired by *natural selection* and *natural evolution.* GAs evolve a pool of solutions (referred to as “individuals” or “chromosomes”) or “population”. Usually, the initial population is randomly generated and it is iteratively recombined and mutated using *crossover* and *mutation* operators. Solutions are selected for reproduction according to a fitness function that measures how good the solutions are toward solving a specific problem (e.g., generating tests that maximize coverage). Over the course of different iterations (or *generations*), this procedure of selecting, recombining, and mutating solution converges towards best-fit solutions. GAs terminate when either the optimal solution to the problem is found or when the search budget (e.g., the number of generation) is depleted. For a detailed reading on the matter, we suggest the work by Sette et al. [168].

3.3 Approach

Proving a program (or here a model) to be correct is generally unfeasible: Instead, one tests for failures. Programs are asserted for a general quality expressed through *happy-paths*, and checked for negative behavior, error recovery, and other issues through tests. In this tradition, we also design our model-test-cases failure-based: We are looking for input that makes the model perform poorly. We assume the *happy-path* is successfully covered through the performance in training and test.

Creating a single faulty data point to produce errors is an easy task, but also one that does the model’s statistical nature injustice: The issue here is that single data points likely will not be useful in producing a fix. An easy way for a model to deal with a single faulty data point would be over-fitting to avoid this particular data point, and any generalization could be mere coincidence. Hence, we zoom out a bit and consider approaches that focus on multiple data points and attributes of the dataset.

How would you go about finding these datapoints? We formulate this as a classic **search problem**:

Theorem 2 *Let $X : P \longrightarrow O$ be a pre-trained model that takes as input a program P and returns an outcome $o \in O$ (e.g., method-name prediction). Let $F = \{f_1, \cdots, f_n\}$ be a set of metamorphic transformations such that $f_i(P) \equiv P \;\; \forall f_i \in F$. Let $m : (X, P) \longrightarrow \mathbb{R}$ be a performance*

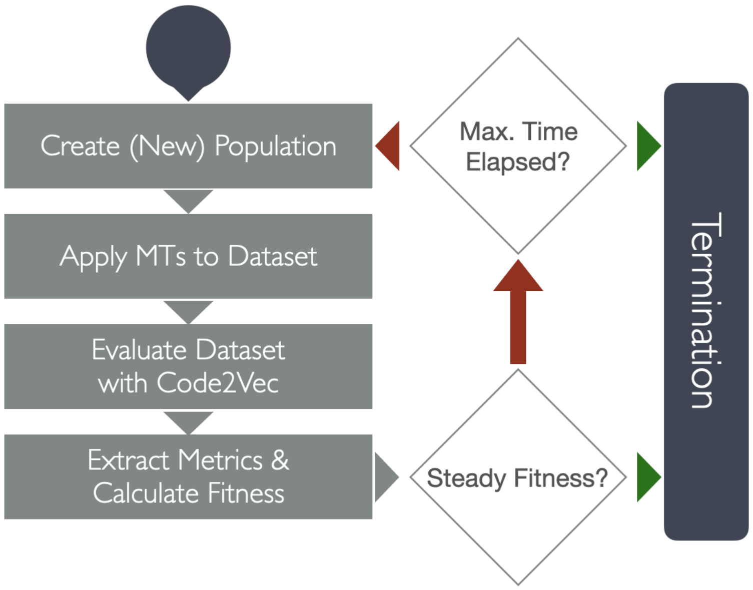


Figure 3.1: schematic Control-Flow of Guided-MT Program

metric (e.g., F1-score) computed on a pre-trained model X with input program P.
***Problem**: finding a program P' obtained by applying F to P that maximizes the differences in the performance metric m:*

$$\max \left| m(X,P') - m(X,P) \right| \ \textit{with}\ P' \equiv P \tag{3.1}$$

The formulation above can be applied to any performance metric. In this paper we focus on F1-score; our goal consists in finding a program P' that is equivalent to an initial program P (hereafter referred to as *seed*) that maximizes the difference in F1-score achieved by a model X on P and P', i.e., $\max\left|F1(X,P') - F1(X,P)\right|$. Equation (3.1) corresponds to our fitness function to optimize for metamorphic testing (our search guidance).

3.3.1 Guided Metamorphic Testing

To optimize Equation (3.1), we implemented a genetic algorithm, whose high-level workflow and its components are depicted in Figure 3.1. Within our experiments, we focused on Java programs and target `Code2Vec` as main the model under test. For `Code2Vec`, we re-use the artifacts provided by Alon et al. [152], including code, model, and dataset. To apply the metamorphic transformations, we use the Lampion framework [102] since it provides the most extensive set of metamorphic transformations for Java programs.

Encoding

As mentioned before, solutions to the problem in Section 3.3 are produced by altering a *seed* program P by applying metamorphic transformations. Instead of encoding a solution as a complete code-snippet, we only encode the changes applied to the AST of

Table 3.1: List of metamorphic transformations (MTs) for Java programs [102].

ID	Description
MT-IF	Wrapping a random expression in an if(true) statement
MT-FI	Wrapping a random expression in an if(false) else statement
MT-UV	Add a random unused variable
MT-RE	Rename a variable
MT-PR	Rename a parameter
MT-ID	Wrap an expression in an identity-lambda function (including function call)
MT-NE	Add the neutral element to a primitively typed expression

the seed image (*mask*). In particular, given the seed program P, we encode a solution (*genotype*) as a sequence of changes to the AST of P: $P' = \langle p_1, \ldots, p_k \rangle$. Each entry p in P' is a tuple $[node_i, f_j]$, where $node_i$ denotes the i-th nodes in the AST of P and f_j is the j-th metamorphic transformation applied to that AST node.

Initialization

The first step for our GA requires creating an initial population of metamorphic tests (or solutions). The initial population consists of creating N copies of the seed program P (i.e., empty mask $S = \langle \rangle$) and randomly applying one of the available metamorphic transformations at a randomly selected AST node of P. In this paper, we consider seven metamorphic transformations proposed in Lampion [102] and listed in Table 3.1.

Selection

Metamorphic tests are selected for reproduction using simple *tournament selection* with a tournament size $ts = 4$ [168]. This selection method first randomly samples four solutions from the last population and selects the solution with the best fitness function (Equation (3.1) in our case) as the winner of the tournament (parent).

Crossover

New solutions/tests are generated by recombining parent solutions selected as described in the previous subsection. In particular, we apply the multi-point (also known as scattered) crossover. This operator creates two new metamorphic tests by combining the entries of two parent solutions around multiple cut points. First we create a *crossover-mask* consisting of a binary vector with randomly generated entries, this entry determines whether a crossover at this position will take place. Each offspring is created as a copy of one of the two parents, and the transformation at index i is replaced by the gene of the other parent if the mask entry at i is $true$.

The masks' length is bounded by the shortest gene, in case of diverging length the crossover happens between the overlapping genes and the remaining genes are left untouched in the longer offspring (copy of the longer parent).

Mutation

Given a newly generated test O, we designed two types of mutation operators that in turns *add* or *delete* metamorphic transformations to O. The probability of application is configurable (for the experiments we chose 80% add and 20% shrink), the application is mutually exclusive. The chance to trigger a mutation is set to 50%, leading to (in average) one of the offsprings to be mutated.

The *add* operator iteratively adds multiple metamorphic transformations following a hyperbolic distribution. In detail it adds a (randomly selected) metamorphic transformation to O to an AST node with probability σ. Once it is added, a second (randomly selected) transformation is applied with probability σ^2, and so on until no more transformation is added. In general, at each mutation iteration n, a new transformation is added with probability σ^n. Notice that a new transformation is added if and only if the limit $L = 20$ is not reached. This non-linear operator is inspired by the *add* operator used in EvoSuite [156, 169] for unit test suite/case generation. In this paper, we set $\sigma = 2/3$ as it asymptotically applies three metamorphic transformations on average (statistical *expectation*).

The *remove* operator randomly removes one metamorphic transformation previous applied to O. This corresponds to deleting one of the entries o_j in the mask $O = \langle o_1, \ldots, o_k \rangle$, with $j \in \{1 \ldots k\}$. This operator tackles the potential *bloating effect* [170, 171], i.e., the length of the metamorphic tests might steadily increase through the generations. This *remove* operator can shorten solutions with spurious transformations that do not contribute to the fitness function throughout the generations.

Elitism

At the end of each generation, there are N parent solutions and N offspring solutions produced via the selection, crossover, and mutation. The new population for the next generation is obtained by selecting the N best solutions among parents and offspring. This survival mechanism is called *elitism* since the best solutions can survive across the generations.

Termination

The search terminates when the total search budget is reached or the fitness function cannot be further improved. We prefer running time over number of iterations as a search budget as it is considered the fairest metric to measure the cost of test generation approaches [163, 164, 166]. This is because the cost of applying the genetic operators (<10 seconds) is —in our context— negligible compared to the cost of computing the fitness function, which requires evaluating the generated solutions against the model under test (2-10 minutes for `Code2Vec`).

3.4 Methodology

3.4.1 Research Questions

Based on existing research (Section 3.2) `Code2Vec` is vulnerable to metamorphic transformations, especially centered around identifiers. Thus, we want to investigate if genetic search improves the produced effect and the speed needed to influence the model. Differences between *random search* and *evolutionary search* for a performance drop are covered in the first research question:

RQ1: Effectiveness of Search

How effective is searching for metamorphic transformations that produce a maximum drop in performance metrics (F1, MRR) of Code2Vec?

The random search adds a number of randomly chosen transformations. Primarily it is the search technique used in the existing body of research w.r.t. metamorphic testing in the ML4SE domain [102, 125, 154] and it is the current state-of-the-art. Second, random search is natural baselines when assessing search-based approaches considering its simplicity and strength. Previous studies showed that random search can outperform evolutionary algorithms in specific SE domains [172, 173].

We want to explore trade-offs between the number of transformations and their produced effect. We expect a larger number of transformations to produce bigger changes in `Code2Vec` output, but how much difference can be achieved for fixed *n* transformations? A low number of transformations keeps the code understandable, i.e., a few redundant control structures or variable names could very well be an oversight in *normal* programming. These realistic data points are the golden fleece of our second research question:

RQ2: Minimizing Number of Transformations

What are the trade-offs between keeping a low amount of transformations and producing a difference in metrics?

To position this work better in the body of existing research, we unravel the genotypes and investigate the obtained transformations. With our third research question, we aim to get an insight into the models behavior and/or data — we aim to answer a set of questions such as: *"Align our distributions of transformers with other publications?"* or *"For different metrics, do we get different transformations?"*

RQ3: Distribution of Transformations

What are the dominant transformations that lead to highest changes in performance?

Existing work shows that metamorphic transformations reduce the metrics *on average* — but individual transformed datapoints can produce a better prediction. We consider this an exotic premise: Can noise make our data better? We hope to gain insights on the topics data quality, model fitting and the stochastic nature of ML with our last research question:

RQ4: Search-Goal Inversion

Can the search be inverted, to produce an increase in performance metrics?

3.4.2 Benchmark and Dataset

`Code2Vec`, the model under test, is a neural attention-based approach that learns embeddings for programs (e.g., method-names) as continuous distributed vectors. The code embedding aims to preserve the semantics of the programs such that semantically similar methods are mapped into similar vectors. To this aim, programs are represented as path contexts, i.e., paths between nodes on the program AST.

We use the pre-trained model by Alon et al. [152], trained on more than 12M Java methods extracted from 10,072 Java projects available on GitHub. Together with the pre-trained model, the training-, validation-, and test-sets are also available in the replication package by Alon et al. [152].

For our paper, we focus on the pre-trained model and use the `java-small` test-set, which contains ~6000 Java methods. We randomly sampled 350 methods from the test-set to use a representative sample. We restricted our evaluation to a smaller set of methods in the test-set because of the large number of runs needed for a sound statistical analysis. We had to choose between using a larger sample of seed programs with few runs or a smaller sample but with enough runs to allow sound statistical analysis within a feasible wall time. We have opted for the latter based on the existing guidelines on assessing randomized algorithms in SE [174, 175], which highlight the importance of performing multiple runs for a proper assessment of search algorithms (including random search and genetic algorithms).

3.4.3 Evaluation Methods

In our experiment, we run GA and random search 10 times for each program seed in the dataset. This accounts for the random nature of the employed search algorithms. In each run, we collected the best solution (i.e., the metamorphic tests with the best fitness function value), its corresponding performance metrics, and its sequence of metamorphic transformations such solution included. We re-ran the same experiments twice, once for each fitness function in Equation (3.1) instantiated with F1-score and a second time using MRR (Mean Reciprocal Rank). In total, for RQ1, we run 350 (seed programs) × 2 (algorithms) × 10 (runs) × 2 (fitness functions) = 14,000 runs.

To answer RQ1, we compare the average (median) differences in F1-score and MRR scores achieved by the two search algorithms in the comparison. Note that achieving a lower F1-score indicates a better ability of a search algorithm to find metamorphic tests that impact `Code2Vec`. In addition to analyzing the median results, we also applied sound statistical tests as suggested by Arcuri and Briand [174, 175].

In particular, we applied the Wilcoxon rank sum test [176], with threshold p-value=0.05. We complement the Wilcoxon test with Vargha-Delaney $\hat{A}_{12}$ statistics [177], which provides a measure of the effect size (or magnitude of the difference). $\hat{A}_{12}$=0.50 indicates that two distributions in the comparison (e.g., F1-score drops achieved by GA and random search) are equivalent. For F1-score, $\hat{A}_{12} > 0.50$ indicates that GA achieves a significantly larger drop in F1-score compared to random search (i.e., GA is better). $\hat{A}_{12}$ also provides an easy-to-interpret classification of the effect size in *negligible*, *small*, *medium*, and *large* [177].

We use non-parametric tests for both significance and effect size over parametric alternatives (e.g., the paired t-test) because the data does not follow a normal distribution

[178][2].

To answer RQ2, we compare the number of transformations required by genetic algorithms and random search to achieve the same delta for `Code2Vec`. In particular, we analyze how F1-score and MRR vary when applying varying number of metamorphic transformations.

For RQ3, we performed a deeper analysis of the data collected for RQ1. We analyze the genotype (sequence of applied transformation) of the best solution produced by random search and GA in each individual run. Our goal is to determine whether certain metamorphic transformations appear more than others in the best solutions.

Finally, we re-run the experiment for RQ1 but search for improving rather than decreasing the performance metrics for *Code2Vec*, i.e., increasing the F1-score and reducing MRR. We want to understand the extent metamorphic testing could be used to increase the robustness of *Code2Vec* rather than looking for adversary examples in which it is vulnerable. Hereafter we refer to F1-min and F1-max to distinguish between the setting used in RQ1 to assess the robustness of `Code2Vec` and the one used in RQ4 to strengthen the performance of `Code2Vec`.

3.4.4 Experiment Setup

We ran the experiments utilizing CPUs on a server with an AMD EPYC 7H12 64-Core Processor. We conducted ten experiments in parallel, which lead to a total wall-time of 3 days and a computation-time of ~400h. We provide all experiments, data and model within a replication package https://doi.org/10.5281/zenodo.7306931 . The code is separately provided at https://doi.org/10.5281/zenodo.7307012 and results are available at https://doi.org/10.5281/zenodo.7307208 .

For the GA, we set a small population size of 10 individuals. This choice considered the high cost of the fitness evaluation (against the model), which can take between 2 and 10 minutes for our population size. Small population size is widely recommended in the literature for expensive fitness functions [179, 180]. For the selection operator, we used tournament selection with a tournament size of 4, which allows better exploitation and fast convergence rate [168]. The mutation rate is set to 0.50, which is relatively high, but it prevents genetic drift in case of a small population size [168]. Finally, we set the crossover rate cr=1.00, which is within the recommended range [181]. For both random search and genetic algorithm, we use the same termination criteria of 360 minutes search time. For GA, we terminate earlier if the (best) fitness has not changed for 8 continuous steady generations. Most of the experiments terminated around 4 hours due to convergence.

[2]We pre-tested the nature of the data distribution using the Shapiro-Wilk test of normality [72]

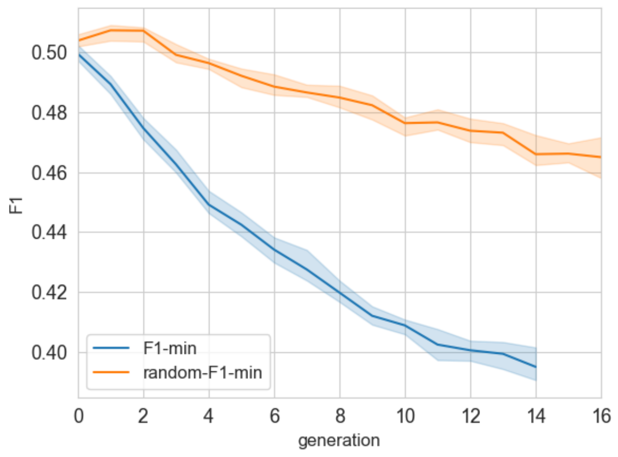


Figure 3.2: Comparison of F1 for random and genetic search

3.5 Results

3.5.1 Effectiveness of Search

Figure 3.3 shows an overview of the achieved changes in metrics for different experiment setups, with a detailed view on the primary experiments in Figure 3.2. Experiments prefixed with *random* exploit random search, while those without utilize genetic search. Most of the configurations do not achieve a visible difference of metrics within the search budget, except for `F1-min` detailed in Figure 3.2. Table 2 summarizes the statistical tests for comparison of `F1-min` and *random-F1-min*: Both experiments achieve a statistical significant difference and `F1-min` achieves higher levels of difference quicker than its random pardon. On average random search needs three generations for 1% drop in F1 while genetic search needs two. The reported Wilcoxon p-value prooves that there is a significant different distribution for the algorithms, and effect size ($\hat{A}_{12}$) shows that after one generation there is a large difference between random and genetic search.

The second biggest difference in F1 are achieved by `random-MRR-max/min`, which *simply* apply random transformations. Random search produces less movement than `F1-min`, but creates a higher delta than genetic search for MRR or `F1-max`.

Our setup seems unable to search for a change in MRR (at least as an dedicated objective). We expect this to be inherent for the model as it was trained solely for F1 and MRR was only reported for comparison with other research. Hence, while side-effects are possible, we consider that the model is *blind* towards unknown metrics — which is somewhat expected. Within the `F1-min` experiment we see an unexpected effect: While the F1-score is dropping, the MRR rises accordingly.

The counter-play of F1 and MRR seems to be related with the dataset and the properties of the metrics. In general, the method-names of the datasets are short and most are between 2-4 sub-tokens. The average prediction without any MTs tends to be slightly longer (2-6

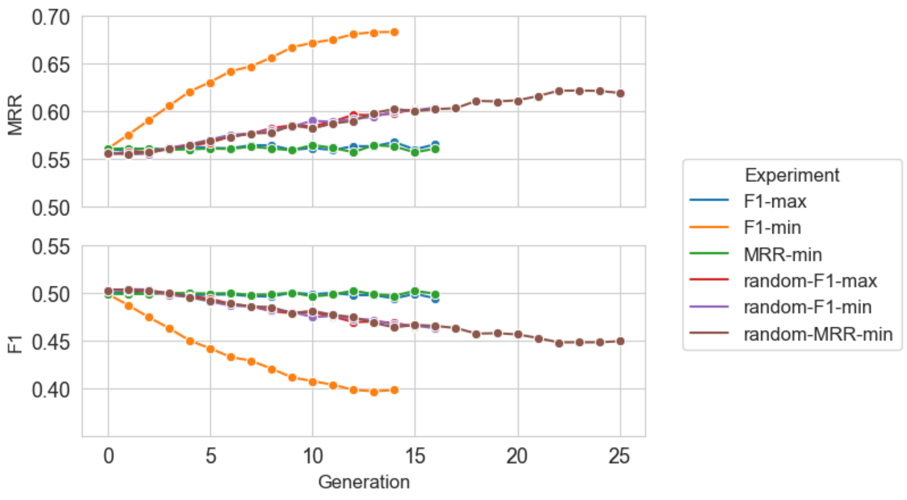


Figure 3.3: Overview of Metric-Movements

tokens). Minimizing F1 pushes the predictions to be shorter, which as a side-effect moves the predictions more towards the right token-distribution, achieving a better MRR in the process. The shorter words are *worse* for F1, as in general longer words are more forgiving. Given 5 or 6 sub-tokens, a partial overlap can produce some scores, but with 1 or 2 sub-tokens it is *"hit or miss"*.

Summary RQ1

Within 15 generations, we found datapoints resulting in a drop of up to 10% in F1-score. While the F1-Score drops for this experiment, the MRR rises respectively. Random search performs about half as good the genetic search, but is in itself significant.

3.5.2 Minimizing Number of Transformations

In terms of (co-)relations between transformations and deltas in metrics, the clear trend we found was a simple correlation between produced movement and number of transformations: More transformations produce a higher change, being nearly proportional (See Table 2 and Figure 3.2). Over the generations, initial iterations produced a higher drop in metrics and more transformations being added which eased out in later generations, however the symmetry between transformations and deltas persists.

Our intended tradeoff-analysis utilizing a weighted-sum approach failed due to this correlation: When equally weighting number of transformations and the observed delta, the fitness remained at roughly the same value producing a stale search, degrading into behavior similar to random search. Introducing more sophisticated approaches such as multi-objective optimization over different metrics is considered valuable future work, as we attribute issues solely to a simplistic fitness function.

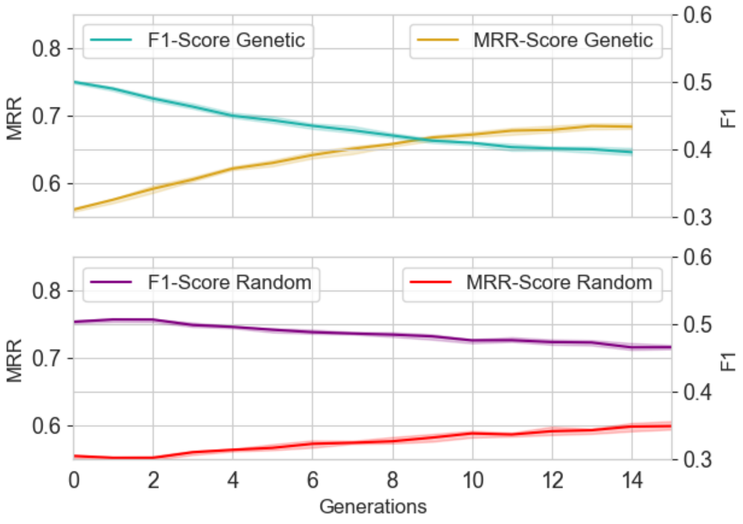


Figure 3.4: Metric-movement for random & genetic of F1-min

Summary RQ2

Movements in metrics are proportional with the amount of transformations. A weighted-sum approach to find tradeoffs failed due to this correlation.

3.5.3 Distribution of Transformations

Figure 3.5 shows the distribution of transformers over generations for minimizing the F1-score. Over the generations, it crystallizes that If-True Transformations and Lambda-Identity-Transformations seem to have the greatest effect on metrics, while renaming variables occur the least.

We attribute this to the embedding-logic as presented in section 3.2.1 as the prominent transformations alter the AST quite heavily: The added redundant condition adds a total of 6 nodes to the AST, the lambda creates additional 5. On the contrary, renaming a variable adds no node and are less represented in our results. The work by Compton et al. [150] proves that the variable names play an important role in prediction, but given our results, it seems that structural changes out-weight the changes in information, i.e. the form of the AST weights more than the content of the nodes.

The *failed* experiments (MRR-based and F1-maximization) show a near-evenly distributed composition of transformers. They behave parallel to random search, and form another piece of evidence that with the current model and approach we cannot search for these optimization goals.

Table 3.2: Statistical Tests for F1 Score and MRR

Gen	Results for F1-score					Results for MRR				
	Random	GA	p-value	$\hat{A}_{12}$		Random	GA	p-value	$\hat{A}_{12}$	
0	0.50	0.50	<0.01	0.59	small	0.56	0.56	<0.01	0.41	small
1	0.51	0.49	<0.01	0.84	large	0.55	0.57	<0.01	0.17	large
2	0.51	0.47	<0.01	0.93	large	0.55	0.59	<0.01	0.07	large
3	0.50	0.46	<0.01	0.91	large	0.56	0.60	<0.01	0.07	large
4	0.50	0.45	<0.01	0.97	large	0.56	0.62	<0.01	0.03	large
5	0.49	0.44	<0.01	0.97	large	0.57	0.63	<0.01	0.02	large
6	0.49	0.43	<0.01	0.97	large	0.57	0.64	<0.01	0.03	large
7	0.49	0.43	<0.01	0.97	large	0.57	0.65	<0.01	0.02	large
8	0.48	0.42	<0.01	0.98	large	0.58	0.66	<0.01	0.01	large
9	0.48	0.41	<0.01	0.99	large	0.58	0.67	<0.01	0.01	large
10	0.48	0.41	<0.01	0.99	large	0.59	0.67	<0.01	0.01	large
11	0.48	0.40	<0.01	0.98	large	0.59	0.68	<0.01	0.02	large
12	0.47	0.40	<0.01	0.99	large	0.59	0.68	<0.01	0.01	large
13	0.47	0.40	<0.01	0.99	large	0.59	0.68	<0.01	0.01	large
14	0.47	0.40	<0.01	0.99	large	0.60	0.68	<0.01	0.00	large

Summary RQ3

The most common transformations were `IfTrue` and `LambdaIdentity`. Transformations seen in existing research had less impact than these structural changes.

3.5.4 Inverting Search-Goals

Initially this RQ was inspired by existing research [102, 150] that found *flaky* datapoints when applying transformations: some got worse, while others got better, with the average being worse. We expected to find symmetrical behavior for maximizing and minimizing alike: If we can find datapoints that produce worse metrics once we add noise, we found a model that overfits on *clean data.* If we can find datapoints that produce better metrics once we add noise, we found a model that is still underfit.

Our experiments show that with the presented approach we cannot search for a maximization, or that the model is truly robust against the changes. Regarding the latter, we still observe the flakiness, but there is no clear *movement on average*. Further generations produce stronger oscillating results, but it cannot keep positive-changes while discarding those that decrease, simply because any change applies in both directions.

It is however strange that the approach did not *proxy* the search for MRR by the inverse optimization of F1 — after all, given Figure 3.3 they seem near correlated.

Summary RQ4

Neither maximizing F1 nor MRR was possible. The MRR noticeably increased when minimizing F1, but this is related to a specific attribute in F1 for the dataset.

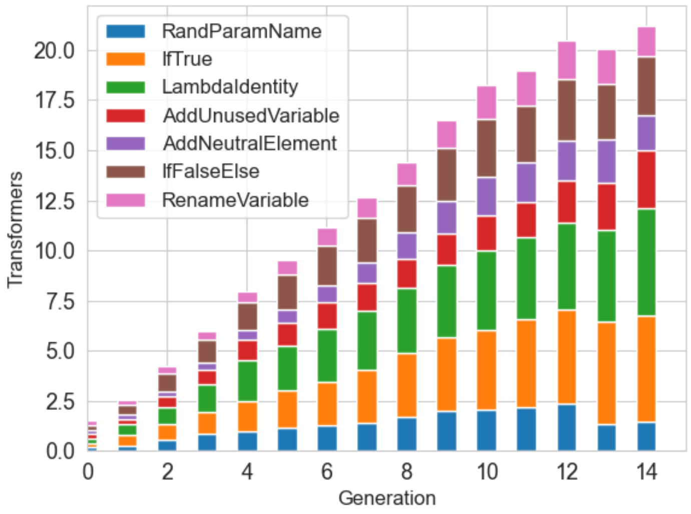


Figure 3.5: Applied Transformers for minimizing F1-score

3.6 Discussion

MRR Experiments. It is puzzling that searching for MRR did not succeed, despite `F1-min` quite successfully maximizing MRR. Why does the `MRR-max` not simply do what `F1-min` does? Looking back at the setup, the error seems to be in the `Genetic Search`. Measurement and evaluation works, and creation of datapoints that increase MRR is successful as per `F1-min`. The reasons could either be related to the feedback-loop as a whole, or be inherent to the search algorithm and its configuration.

To solve this open question, we suggest further research including a model trained on both metrics. We envision a set of models trained for MRR, F1 and in best case both, adopting the experiment from this work for each model. If we observe the same blindness towards F1 when trained on MRR, we see a connection between training metrics and search success.

Failed Maximizing. The performed experiments failed to achieve a maximization of metrics showing a behavior similar to random search. With the current configuration we apply a change to every datapoint in the test-set, but a more fine grained application is possible. In theory, a gene could be constructed consisting of `changes-per-file`. While this forms classic *future research*, we want to take a moment to dis-encourage attempts: The used dataset with 680 methods forms a representative sample of the original dataset, with an average F1-score of 0.50 Gaussian distributed. Even if every single transformation would maximize F1 for a given datapoint from 0 to 1, this transformation will at most contribute $\frac{1}{680} = 0.14\%$ towards a better F1 score. To achieve movements similar to those observed in this work, hundreds of generations are necessary[3].

[3]This is only for `java-small` – the other available datasets are magnitudes bigger

3.7 Threats to Validity

Construct validity While we assume the metamorphic tests generated by either GA or random search are equivalent to the original seed programs (due to the metamorphic relations), the resulting mutated programs might not be realistic (e.g., too many nested if conditions).

Internal validity. We selected 350 programs from the `Code2Vec` test-set using a randomised sampling that ensured diverse programs were selected considering (1) the original source project from GitHub, (2) the application domain, and (3) code characteristics (e.g., code complexity). We picked a representative sample size, however, due to implementation details, we had to re-draw the sample in some corner cases. Some elements were unsupported by our transformers, such as Java files consisting of (only) enums. The final elements are unaltered from the original dataset and are provided in the replication package. In a similar direction, some files contained multiple classes, and many of the files contained varying amounts of methods — leading to different *weight* in the fitness calculation as we apply the transformers *per class*. We consider these uncertainties to be addressed by our statistically significant sample size.

Conclusion validity. To address the randomness in the search process, we ran each search algorithm ten times on each seed program with a different random seed (for the random number generator) in each run. Besides, we applied statistical tests (i.e., the Wilcoxon rank sum test and the Vargha-Delaney statistics) following the existing guidelines on how to assess randomized algorithms [174, 175]. While the choice to re-run the algorithms multiple times reduced the number of seed programs we could consider, our analysis is statistically sound.

3.8 Conclusion

The goal of this paper is to expand existing metamorphic testing for SE4ML with an evolutionary search to save on computational costs and produce more realistic data points (w.r.t. the number of applied transformations). To that end, we implemented a Java program combining Code2Vec and a genetic algorithm using the models' metrics as fitness functions. We designed an experiment sampling a representative amount of data points from Code2Vecs' `java-small`-dataset and tried to minimize F1 and MRR with both random and genetic search.

Our results show that both random and genetic search significantly change the model metrics, with genetic search being stronger with progressing generations (determined per effect size) and leading to a total reduction of 10% in F1 for genetic and 3% in F1 for random. Minimizing MRR did not succeed with genetic search, performing similar to random search, likely due to the model being trained solely on F1 score. We found a near-proportional relation between change in metrics and applied transformations, failing our trade-off analysis experiments due to leveling the weighted-sum fitness function.

In summary, genetic search improved the existing research by proposing a more intelligent way to generate example. We consider our successes a worthwhile adoption for current approaches and our failures to be good starting points to address topics such as derived metrics and tradeoff analysis.

4
HasBugs - Handpicked Haskell Bugs

Summary

We present HasBugs, an extensible and manually-curated dataset of 34 real-world Haskell Bugs from 8 open source repositories. We provide a faulty, tested, and fixed version of each bug in our dataset with reproduction packages, description, and bug context. For technical users, the dataset is meant to either help researchers adapt techniques from other programming languages to Haskell or to provide a human-verified gold standard for tools evaluation and enable future reproducibility. We also see applicability for qualitative research, e.g., by analysis of bug lifecycles and comparison to other languages. We provide a companion website for easy access and overview under `https://ciselab.github.io/HasBugs/`.

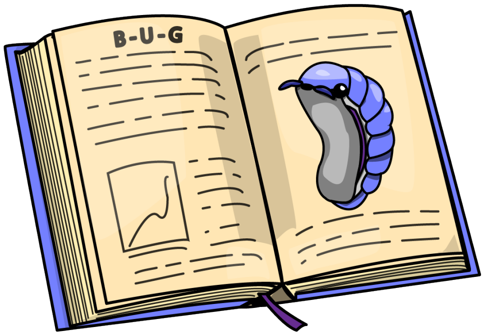


Figure 4.1: HasBugs - Detailed Haskell Bugs

This chapter has been published as *HasBugs - Handpicked Haskell Bugs* by Leonhard Applis and Annibale Panichella at *2023 IEEE/ACM 20th International Conference on Mining Software Repositories* (MSR 2023).

Preface - Positioning of HasBugs

HasBugs came to life as an idea already before the other work on Haskell - as seen with numerous examples in the introduction, data sets play a very important role for designing and evaluating software engineering tools. This has only become more prominent with machine learning tools requiring massive amounts of data, and data quality becoming a visible issue, might impacting the models heavily (as seen in the first two chapters). Having no such data for Haskell constituted a big barrier for research on my favorite programming language and started the thought of gathering bug-data. As a more personal opinion, I believe that the data will shape the tools we create with it. That is, if we have only datasets with *single location* faults, we are unlikely to see tools fixing bugs in multiple locations - after all, how would we evaluate its features? For machine learning, such issues arise by their definition.

Another issue that we see is that even good-hearted tool developers will hit a wall once in a while, and need to filter out data. When looking at Defects4J [87], it also contains datapoints for `Kotlin`[1]. The Kotlin datapoints broke with many of the other (mostly Apache) conventions, using a different test framework and utilizing compiler pragmas. This lead to Kotlin being effectively abandoned from Defects4J, and excluded by most research on the matter. The resulting tools focused on JUnit as input, and Kotlins ghost was only redeemed with the first LLMs (e.g. [182]) revisiting Defects4J. For Haskell, there is currently no overview *what is popular* and it is not quantifiable as it is for Java. More so, some metrics become debatable - a set of good properties can be worth a thousand unit tests. Does being *used more often* mean that unit tests are more popular? Are properties *more powerful* and form a stronger test suite?

HasBugs aims to open up the discussion about these topics. While limited to open source Haskell projects, we already see a variety of test tools, frameworks, features and conventions. For example, the tools seem to employ a mix of properties, unit and golden tests, where regression is usually covered by golden tests originating from the issues. This observation also leads to different research: Program repair should incorporate golden tests, while test generation might benefit from focusing on properties that lead to efficient coverage. Having HasBugs as a (small) data set also makes this *terra nova* accessible for tool designers and researchers.

Why is this before the Haskell work? This is arguably a debatable design decision. The motivation to put HasBugs here is to provide easy introduction to Haskell and the challenges it faces with bugs. The later chapters require more knowledge about specific mechanisms (e.g. typed holes and laziness) which will then be discussed in detail. The bug dataset is understandable for readers not familiar with functional programming details. Among domain-specific bugs (`HLS-3` deals with locally defined type-families in an IDE), uncommon tests (`Cabal-1` starts and compares system PIDs), there are also mundane problems such as issues with string escaping (`hledger-3`) or user-experience (`hakyll-1` was *swallowing* too many errors) One goal is that people coming from automated software engineering in Java (the first chapters) see the HasBugs as an appetizer to understand the domain and see value in its challenges without diving head-first into stack-traces of lazy evaluation.

[1]`https://kotlinlang.org/`

4.1 Introduction

Bugs are usually seen as obstacles - nuances and failures resulting from mistakes. For researchers in software engineering (SE), however, bugs are opportunities. They form the foundation for techniques such as fault localization [76][183], test generation/fuzzing [155][91], and program repair [162][184]. Observations in these fields show the age-old adage *garbage in - garbage out* applies to these domains as well: It took the community a while to realize that not all patches produced by `GenProg` [85] actually fix the program [88] - despite passing the test suite. While we can blame individuals for this, such mistakes happen and the more constructive approach is to mitigate these issues with better input. The information missing in `Defects4J` was a summary of the bug, so generated patches could not (easily) be checked by the respective researchers. The assumption *'passing CI = fixed program'* turned out to be insufficient.

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

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

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

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

The contributions can be summarized as follows:

1. 34 datapoints from 8 FOSS Haskell programs and libraries
2. Rich context information linking source code and GitHub
3. Multi-point fault-locations, test-patches and 3-stage-versions (faulty, tested & fixed)
4. Dockerfiles and pre-produced available images running builds

Similar Datasets are Defects4J [58] for Java programs, and in particular, the scripts and support created around it later[2][87]. We provide the same granularity/content of

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

bugs with the difference that our bugs exist without a failing test case first. In case of a passing CI, a failing test case is also provided separately, constituting a third version. We hope to increase the reproducibility by providing Dockerfiles for most datapoints, saving researchers from the time-intensive work of configuring necessary dependencies. While there are multi-fault locations in the newer supplements of Defects4J [185], we provide the multi-fault locations within the dataset.

Another close dataset is Bugswarm[186], which is mined from FOSS Java Projects using Travis CI. Bugs in Bugswarms are, hence, not human-evaluated. With over 3000 data points, a manual inspection is unlikely at this point. We tried to adapt ease-of-use from Bugswarm and, in particular, their very accessible website. We purposefully did not try to automate bug-mining to keep high quality and double-check every entry.

The last related dataset is Simple Stupid Bugs [187], which is automatically mined from FOSS Java Projects with single-line fixes. By far the biggest dataset, it also has the least context information, and its quality assessment is based on sampling. Furthermore they rely on SZZ [188], itself debated [189][190], for fix-localization.

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

4.2 Dataset Description

Currently, `HasBugs` contains 34 bugs from 8 repositories as shown in Table 4.1. Every bug consists of a primary `json` file that holds the unique information of the bug: the Git repository, relevant commits, PRs, descriptions, etc. A subset of the information is shown in Figure 4.3, which has been shortened for the sake of readability. With each datapoint, we provide a Dockerfile for a reproducible build, alongside the bug-asserting test isolated into a git patch. The `HasBugs-Dockerfile` exists alongside potential project-inherent Dockerfiles and performs the compilation during `build`-stages and has the `test-command` as the entrypoint. This addresses Haskell‘s long compilation times - pulling the image from a container registry[3] starts from compiled source-code immediately with running tests.

These `reference`-files are a lightweight set of information that can either be used directly or be manifested into heavier artifacts (using shell scripts). The download of repositories is automated from a data point, providing an artifact of the repository usable for static analysis. The archived projects in their three-fold states can be accessed under https://doi.org/10.5281/zenodo.7569135. On top of that, the repositories can be built inside docker containers either by using the provided `HasBugs-Dockerfiles` or by pulling pre-compiled images from the repository. Both activities are supported by shell scripts accompanying the data repository.

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

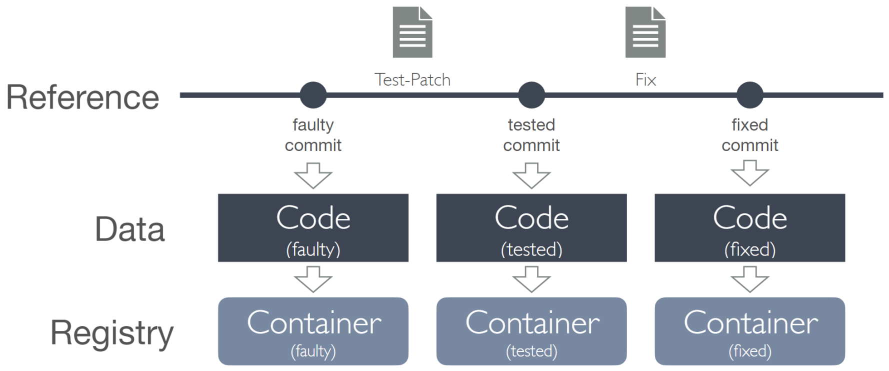


Figure 4.2: Overview of one Data-Point in HasBugs

Table 4.1: Summary of HasBugs per Repository

Repository	**Bugs**	**Stars**	**.hs-Files**	**Domain**
Cabal	6	1.5k	1.3k	Build System
Pandoc	8	27.7k	291	Document Conversion
ShellCheck	5	31.2k	24	Linter
HLS	4	2.3k	1.3k	IDE Language Server
Purescript	1	8.0k	220	Transpiler
HLedger	3	2.3k	156	Accounting
Duckling	6	4.1k	609	Entity Extraction

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

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

Figure 4.3: Example HasBugs Datapoint.json (edited for readability)

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


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

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

4.3 Data Collection and Challenges

The data collection was primarily a manual process. We started by gathering a list of high-star repositories from GitHub and Hackage[4] and filtered it for suitable FOSS licences, which resulted in a list of 44 libraries and programs.

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

From the most suitable projects, the authors decided on the initial data points by diversity: We aimed to cover *as many domains as possible*, to cover a variety of different bugs. This led, e.g., to the inclusion of Cabal (a build framework for Haskell), while Stack (another build framework) was left out. In general, many tasks covered by Haskell libraries revolve around parsing and compiling, but we consider the projects shown in Table 4.1 a good, diverse view of the mainstream applications of Haskell.

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

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

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

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

4.4 Research Opportunities

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

Technical contributions consists of tools for **fault localization**, **test generation**, **test amplification & carving** or **automated program repair** (APR). As our dataset has (often) multiple fault and fix locations, the field of multi-location fault-localization [185] could be investigated, as well as automatic analysis of which part of a patch contributes

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

to the fix. Test generation [171, 191], amplification [192], carving [193], and regression testing [158] are particularly supported as we provide the fixed version and a sample test, which cover the requirements for most common techniques in the field. Comparing the existing test can help to assess readability, coverage, and functionality. Further relevant research approaches in the field on mining software repositories (MSR) that can highly-benefits from our dataset include bug prediction, crash replication, fault localization, bug severity classification, and bug-introducing commit identification. Inspired by Sobreira et al. [87], program repair benefits from the specified *bug-reason* which supports humans in assessing automatically produced patches. The two significant challenges for APR - a stable running build as well as failing tests - are addressed by our dataset and the containers.

For **qualitative analysis**, we see good opportunities in the communication patterns of the ecosystem. The Haskell community is often perceived as slightly alien or elitist, with one of the prominent mantras being: "you don't write bugs in Haskell - once it compiles, it works". Judgments aside, our faulty versions compile and pass tests, and the presented self-admitted bugs can originate from manifold sources, for example actual implementation errors, missed requirements, or environmental factors (e.g., OS changes). We consider it fruitful to investigate the nature of the bugs and their fixes, and compare it to studies on Java [187].

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

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

4.5 Attributes and Examples of Bugs

Most of the work in this chapter focussed on the technical details of the data, but it is also worth to look at some of the datapoints and their artifacts in detail.

Testing & Testing Efforts The functional nature and encapsulation of side-effects lends itself for Haskell to be easy unit-testable. This should apply to all levels of testing, including regression tests, but does not seem to be followed. The type of test is mostly determined by the project setup, but most regression tests fall into the category of system level tests. Projects like `Pandoc` often receive a document as part of their issue, and utilize exactly the reported (minimal) document as part of a system test. The majority of cabal bugs (in `HasBugs`) are related to the console, and the tests also revolve around the packaged binary. Hledger mostly uses `Shelltest`, which are system-level golden tests. Notable exceptions to this are ShellCheck and Duckling, where bugs also directly relate to input-data. Unlike the other projects, the input-data for both are strings, making it easier to bring them to unit tests.

Why the projects utilize testing in this matter is not fully clear, but we'd like to propose some rationales: First, *translating* the faulty parts of input data takes effort, but might not produce a *better* test. Second, formulating the regression test around input data, as a system level test, verifies the behavior that users might experience. Especially for pandoc and cabal, reports are made for bugs that users experienced system-level, and it is a solid assumption that a fixed system-level test will also alleviate the issues of the user. It also allows for a direct mapping of tests, issues and commits, as many projects seem to embrace.

Bug and Fix Length The average fault touches 1.08 mean files, and has a mean length of 4.58 lines of source-code. This is the accumulated result over small, one-file-one-line patches (e.g. shellcheck-5, duckling-4), and fixes that require the removal of larger pieces of functions (e.g. cabal 5, purescript 10).

The average fix touches 1.3 mean files and introduces changes to a mean of 17.5 lines of source-code. This again is partially due to small, one line changes, but often larger pieces of code are introduced. Cabal-5, for example, removes a pattern match and re-introduces a function that handles the cases. Many of the *larger* patches in HasBugs follow this pattern, e.g. cabal-4 or pandoc-7. Most of the bugs that introduce new functionality, also introduce a new function. Sometimes, these functions are not declared on module level, but as a local function (e.g. cabal-6 introduced a local function `packageDBsToUse`).

As rough categories, the bugs fall in three categories: *Simple*, one-line changes similar to SSTUBS [194], one-line faults that introduce a function for its fix, and lastly larger changes that delete, and introduce new functions. These categories should be kept in mind when pursuing further research - e.g. a tool that cannot introduce new functions is not able to replicate a patch in the same manner as the maintainers did. For many other disciplines, like test-generation and fault-localization, single line tools are likely sufficient, given the average fault length.

Interactions in Issues and PRs Most issues report a high level of detail, often directly referring to code or providing a test-case. From the expertise expressed in the issues, it can be safely assumed that the users are (Haskell) developers too. The exception to this are Pandoc (many users report behavior with a file as example) and Duckling. Duckling has a test suite centered around examples, and it is easy to translate a report into a unit-level test.

Cabal, HLS, Pandoc, ShellCheck and Purescript provide Issue-templates, that are generally followed by the users. For many of the other issues, the users self-provide a similar approach of separating their report into *steps-to-reproduce*, *expected behavior* and their system setup. For Duckling many of the fixes are provided as pull requests, without an initial issue.

The general sentiment throughout all interactions is positive and constructive. In Hledger, reporters of bugs are explicitly appreciated. Contributors of merged Pull Requests are thanked in ShellCheck, Pandoc, HLS, Hledger, Cabal and Duckling with the notable exceptions of project-maintainers not thanking themselves. It might be the case that in *short* accepted Pull Requests there are other, less explicit ties between the actors and they might communicate on other channels.

HLS, Cabal and Pandoc have longer discussions, especially around code, from multiple (more than 2) contributors. The review often considers alternative implementations,readablity (especially conciseness) and performance.

4.6 Limitations and Future Work

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

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

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

4.7 Conclusion

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

Online Resources

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

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

5

CSI: Haskell - Tracing Lazy Evaluations in a Functional Language

Summary

In non-strict languages such as Haskell the execution of individual expressions in a program significantly deviates from the order in which they appear in the source code. This can make it difficult to find bugs related to this deviation, since the evaluation of expressions does not occur in the same order as in the source code. At the moment, Haskell errors focus on values being *produced*, whereas it is often the case that faults are due to values being *consumed*. For non-strict languages, values involved in a bug are often generated immediately prior to the evaluation of the buggy code. This creates an opportunity for *evaluation traces*, tracking recently evaluated locations (which can deviate from call-order) to help establish the origin of values involved in faults. In this paper, we describe an extension of GHC's Haskell Program Coverage with evaluation traces, recording recent evaluations in the coverage file, and reporting an evaluation trace alongside the call stack on exception. This lets us reconstruct the chain of events and locate the origin of faults. As a case study, we applied our initial implementation to the `nofib-buggy` data set and found that some runtime errors greatly benefit from trace information.

This chapter was published as *CSI: Haskell-Tracing Lazy Evaluations in a Functional Language* by Matthías Páll Gissurarson and Leonhard Applis at *IFL 2023: Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. It won the Peter Landin Prize for best paper.

5.1 Introduction

Problem and Motivation In crime scene investigation (CSI), establishing the sequence of events constituting a crime is a key technique in solving cases. While less dramatic, programs can still die: despite satisfying the compiler, even Haskell code can crash or have faulty output. When an error occurs at runtime, a common approach is investigating the reported call stack to determine where the error originated. As an example, the code in Figure 5.1 crashes with `*** Exception: Prelude.head: empty list`, and provides an error message containing the stack trace (seen in Figure 5.2). Despite the crash in `head`, the root cause of the error is based on `divs n` which results in an empty list when `n` is prime (there is an off-by-one error: `n` should be `n+1` in line 4). This is a motivating case of an error caused by the wrong data being *produced*, in contrast to errors caused by the right data being incorrectly *consumed* [1] (e.g. evaluating an `undefined` that should have been replaced). The stack trace in Figure 5.2 does not mention `divs`, and only indicates that the error stems from `head`. This lack of information makes it difficult for developers to reconstruct the events that led to the error and determine the root cause of the fault.

Without further hints, any function used (and its dependencies) is a potential suspect. This offset in the tempo of call and evaluation is not a novel discovery; in fact, a similar example to Figure 5.1 was presented by Marlow in 2007 [195].

```
1  module Main where
2
3  divs :: Int -> [Int]
4  divs n = go 2
5    where go i | i == n = []
6          go i = if d i
7            then i:(go (i+1))
8            else go (i+1)
9          d i = n `mod` i == 0
10
11 smallestDiv n = head (divs n)
12
13 main :: IO ()
14 main = print (smallestDiv 13)
```

Figure 5.1: Our running example, a generator for the divisors of a number, with an off-by-one error in the base case.

Approach To address the issue, we implement an extension to Haskell Program Coverage (HPC) built into GHC: in addition to tracking expression evaluation with *ticks*, we also emit instructions in the intermediate language to track the order of *started evaluations* and *completed evaluations*. HPC is discussed in further detail in Section 5.2. We also track the current *evaluation depth*, the number of ongoing evaluations. This allows us to reconstruct

[1] This can be translated to *blame*: is it the producer or the consumer that is wrong? Call stacks help to spot bugs in consumers, while we focus on bugs in producers for this work.

```
divs: Prelude.head: empty list
CallStack (from HasCallStack):
error, called at libraries/base/GHC/List.hs:1643:3
in base:GHC.List
errorEmptyList, called at
libraries/base/GHC/List.hs:82:11 in base:GHC.List
badHead, called at libraries/base/GHC/List.hs:78:28
in base:GHC.List
head, called at Div.hs:10:17 in main:Main
CallStack (from -prof):
Main.smallestDiv (Divs.hs:10:17-29)
Main.main (Divs.hs:13:15-28)
Main.main (Divs.hs:13:8-29)
Main.CAF (<entire-module>)
```

Figure 5.2: Error message generated by the program in Figure 5.1.

a partial *evaluation tree* an overview of completed, partial, and uncompleted evaluations of expressions, when an exception occurs (see Section 5.3.2 for details). We also track a *global trace index* that allows us to reconstruct a trace across all modules from the trace of each individual module. These *recent evaluations* are kept in a circular buffer alongside the HPC ticks and can both be inspected directly at runtime or summarized and reported on an exception alongside the call stack.

In particular, adding an evaluation trace for users is as easy as passing an additional flag during the compilation phase. It constitutes a noninvasive addition to debugging, does not require any changes to the developer's code (such as call stack annotations) and allows a better understanding of what is going on at runtime even when external libraries are being used.

This extension for tracking evaluation traces constitutes the main contribution of this work. Improved runtime errors are one low-level domain that motivates the extension and is easy to understand for broad audiences. In the future, these evaluation traces could be used for more sophisticated use cases, such as program repair or visualization (see Section 5.5).

Experimental Evaluation We apply our prototype to a subset of the `nofib-buggy` data set [70]. The data consist of a selection of `nofib` programs which GHC uses for internal validation with artificially introduced bugs (see Section 5.3.7). These bugs result in either a runtime exception (e.g. index-out-of-bounds or division by zero) or incorrect output. In our pre-processing of the data set, we

- remove all non-terminating programs, and
- add `assert` statements to those data points that return incorrect output to force a runtime exception.

This accounts for a total of 21 investigated data points. From the initial findings, we see a trend that certain exceptions benefit from trace information, depending on the exception type. The data points using `assert` usually cover the fault, but the quality of the trace is dependent on the scope of the test — unit tests are more precise, while system tests produce crowded traces with many locations irrelevant to the introduced bug. We analyze the performance overhead introduced by collecting traces, which seems stable: most data points require between 100% and 300% more compute time, depending on the length of the collected trace. Maximum memory usage increases from 20% to 120%, and the additional binary size is negligible. There is a general trend that the additional memory allocation is related to the number of modules, while the additional compute time seems to depend primarily on the total number of evaluations the program makes. As the `nofib` data set is used in the current test suite and the benchmarking of GHC, we consider it representative of performance estimates. We thus suggest collecting and reporting the evaluation on a per-exception basis. In the long-term view, we hope to support debugging for new and seasoned Haskell programmers alike, but we also see the potential for classroom use: using the data collected by HPC at runtime the evaluation tree can be partially reconstructed (up to the length of the trace) and a clearer view of non-strict evaluation presented to students. *Understanding* laziness is a big challenge for students from other programming paradigms, and visualizing (both buggy and working) program evaluation traces can be a great aid. Our experiments are shared in a replication package[2].

We utilize `nix` and shell scripts for easy replication, but we also provide the output (enhanced error messages) for lightweight investigation without additional dependencies. The contributions presented in this work can be summarized as follows:

1. a prototype implementation of a non-invasive, optional, coverage-based tracing of evaluations,
2. example tooling-improvement by reporting of evaluation traces alongside call stacks,
3. an initial investigation on the `nofib-buggy` data set, and
4. estimates of performance overhead

5.2 Background and Related Work

Thunks In non-strict languages, values are not evaluated until needed in the computation. In Haskell, this is implemented through *thunks*: instead of directly producing a value, expressions produce a *thunk* that represents that unevaluated expression. This behavior is similar to asynchronous concepts in other languages like `Promises` (JavaScript) or `Task` (C#), which are often used for side-effectful computations (e.g. network requests), while in Haskell they are used for all computation. When the value of that expression is required, the thunk is evaluated and resolved to a value. This value might be fully-evaluated if it is, e.g., an integer. But it might also be just the head of a list, with the rest of the list being another thunk. Thunks are in most cases memoized, meaning that the value is evaluated only once, and the result saved. This is then *shared* if the value is needed again at a later time, without requiring re-computation.

[2]https://doi.org/10.5281/zenodo.10090375

Program Coverage and Ticks Haskell Program Coverage (HPC) is a tool that is part of GHC and allows developers to track which expressions were evaluated during the execution of the program: whenever an expression is evaluated, it bumps a number in an array (a “tick”) [196, 197]. These numbers are unique identifiers specified in a per-module `mix`-file, which are on tick registered in a companion per-program `tix`-file. For this work, we reuse the `mix`-files and identifiers unchanged, and extend the `tix`-files with extra arrays using the identifiers. Note that HPC does not require changes to the source code, but instead operates with compilation flags that emit additional instructions to the intermediate language. As we consider this an elegant interaction, we also strive for a flag-based change to the intermediate language.

The data collected by HPC can be used to generate a report on how many times each expression was evaluated and used, for example, to check the test coverage or identify unused code.

Stack Traces & Error Messages While research on the use of stack traces is a popular topic, e.g. when applied for program slicing or crash reproduction, their dedicated value for manual debugging is not thoroughly investigated. Bettenberg et al. [198] investigate differences originating from different perspectives of bug reports. One of their findings is that developers need information that users rarely provide, of which stack traces are particularly useful.

We also recommend the work of Becker et al. [199] as a general overview of research on error messages. Their extensive meta-study covers many findings and trends from the fields of technical implementation, pedagogic use, and improvement of error messages. Among their primary findings relevant to this work are: students and programmers alike actually read error messages and stack traces [200], students are discouraged when error messages do not point toward the faults [201–203] and that cognitive load should be considered in the design and presentation of errors [204–206] (i.e., do not overwhelm the user with information). Lastly, motivating this work, they identify *localization* as one of the defining technical attributes of error messages that constitute their quality, and we aim to enable better localization.

We argue that our work contributes towards the usefulness of traces and forms a starting point for similar research on Haskell. In the absence of detailed analysis in Haskell, our objective is to provide a similar investigation to that of Schroeter et al. [207] in the coverage of bug locations through stack traces. In their work, Schroeter et al. run buggy programs with known faults and investigate the produced stack traces to determine whether and where they contain the fault. We reproduce this for the `nofib-buggy` stack traces and apply the same approach for the evaluation traces.

Stack Traces for Haskell We often take stack traces for granted, but they have only been available in a limited form until recently for Haskell. As late as 2009, Allwood et al. [208] and Marlow [195] tackled the first issues that appeared due to a mismatch between the source code and the optimized executed code. Their central contribution is to address the differences between the behavior of the stack (and stack traces) and the original program syntax, by introducing a transformation of GHC core programs into ones that simulate passing a stack around to preserve the stack trace of the executed program [208]. This was further improved by introducing the **`HasCallStack`** constraint that does not need to be simulated by the runtime system, but while this constraint can sometimes be inferred, our experiments with the `nofib-buggy` data set show that this is not often the case. Similarly, the simulated call stack adding `-prof` in conjunction with the `-fprof-auto` and `-fprof-auto-calls` flags is either

- not printed for the exceptions in `nofib-buggy`, with the output being `Main: divide by zero` or similar,
- or in the form of

  ```
  CallStack (from HasCallStack):
  assert, called at Main.lhs:78:5 in main:Main
  CallStack (from -prof):
  Main.main (Main.lhs:(75,3)-(78,59))
  Main.CAF (<entire-module>)
  ```

  indicating the `assert` and the `main`-function, without giving further clue to the location of the bug.

In the output of our running example `div` from Figure 5.2, adding the `-prof -fprof-auto -fprof-auto-calls` improves the situation slightly, indicating the `smallestDiv` function, but this improvement does not extend to the `nofib-buggy` data points. Using GHC's profiling also requires annotating the **`Prelude`**`.head` function with a **`HasCallStack`** constraint, but it is still not enough to locate the fault. Manual annotations with **`HasCallStack`** are non-optimal for programmers and in our running example extend the information on the crash, but not on the source of it.

Based on the existing research, the current state of Haskell stack traces faces two main challenges: higher-order functions and lazy evaluation. Especially when combined, these tend to disturb stack traces or produce errors that are not aligned with the reported traces. We hope that our work improves the quality of errors for lazy evaluation and enables later researchers to improve errors for higher-order functions.

Tracing Evaluations for Haskell There have been some approaches to tracing Haskell evaluations, which differ from the coverage-based technique presented in this work. Chitil et al. [209] compared three systems available in 2000: HAT [210], HOOD [211], and Freja [212]. Another system mentioned is Buddha [213].

Some approaches are conceptually different, namely Buddha and Hood require changes on a source code level. This limits their application, e.g. is it hard to extend tracing to external libraries.

A part of Freja consists of a custom Haskell compiler that covers a subset of the Haskell98 standard (e.g. excluding **`IO`**). The code that is compiled is altered in an intermediate language, and the redexes are recorded. Frejas concept is the closest to the one presented in this work. Our approach differs by ① extending existing GHC modules instead of requiring an extra compiler ② covering a trace of the last evaluations instead of *all* evaluations and ③ tracking whether an evaluation was started and or finished separately. In a similar manner, HAT is tied to nhc98 and transforms the source code outside of the compilation process, which can cause performance issues and is harder to integrate with external tools. With their dual systems of data creation and browsers, the existing tools went a step further than the contribution of CSI: Haskell. Concepts of how to use the evaluation data produced are presented in Section 5.5. We also hope that the separation of tracing and debugging helps to create additive tools in a modern Haskell landscape.

Static Methods CSI: Haskell is a *dynamic* approach, based on running the code, in contrast to *static* methods, which analyze faults without running the code. In Haskell, there is already a rich type system that allows expressing a wide variety of behavior that is checked at compile time. However, these do not capture many attributes commonly expressed with properties. One approach to lift "property-style" testing and debugging to static checking is to use *refinement types* [214]. These types of checks are integrated through a GHC plugin [215], allowing properties such as `x > 0 ==> f x > 0` to be statically checked by an SMT solver. One extension to this is *lazy counterfactual symbolic execution* [216]: When paired with refinement types such as those in Liquid Haskell, lazy counterfactual execution allows the localization of refinement type errors, revealing faults in the code to be found without need for tracing. This constitutes a heavier approach and raises the entry barrier for ecosystems (including libraries) that do not yet have a refinement type specification.

Algorithmic debugging Algorithmic debugging methods are dynamic approaches based on recording information during program execution and then asking the developer whether the intermediate statements agree with their intention [217]. In most languages, this debugging suffers from side effects, which are no concern in pure functional languages, making Haskell a prime candidate for algorithmic debugging. One tool for Haskell is HOED [217], which extends the debugger HOOD [211] with GHC's profiling information. Like HOOD, it requires users to annotate the functions that they wish to "observe" and create profiling cost centers. Based on this combination, it is possible to construct a computation tree from the collected traces for the observed expressions [217]. This rich approach provides a lot of information but differs from CSI: Haskell in a few points. First, CSI: Haskell utilizes HPC and thus coverage and does not rely on tracing and cost centers. Second, our approach is capable of capturing evaluation trees, in a similar manner to computation trees, providing information on the *actual execution of an evaluation* (that is, the state of each value within the captured tree), but do not capture the values themselves. Lastly, CSI: Haskell gathers data on the entire project and does not require manual annotations on suspicious elements of the codebase. Thus, we start with an earlier stage of debugging, where suspicious elements still need to be identified.

We consider CSI: Haskell not as a debugger, but instead a source of trace information for follow-up tools. The example presentation as evaluation traces could greatly benefit from concepts of algorithmic debugging, but lies beyond the scope of this work.

$$\dfrac{\Gamma_0 \vdash \texttt{head}^2 \Downarrow \lambda\texttt{xs. head' xs}, \Gamma_1 \qquad \dfrac{\dfrac{\Gamma_1 \vdash \texttt{divs}^4 \Downarrow \lambda\texttt{n. divs' n}, \Gamma_2 \qquad \Gamma_2 \vdash \texttt{n}^5 \Downarrow \texttt{n'}, \Gamma_3 \texttt{ where } [\![n = n']\!] \qquad \dfrac{(\ldots)^6 \qquad (\ldots)^7}{\Gamma_3 \vdash \texttt{divs' n} \Downarrow \texttt{xs'}, \Gamma_{n-1}}}{\Gamma_1 \vdash \texttt{(divs n)}^3 \Downarrow \texttt{xs'}, \Gamma_n \texttt{ where } [\![xs = xs']\!]}}{\Gamma_1 \vdash \texttt{head' xs} \Downarrow \texttt{v}, \Gamma_n}}{\Gamma_0 \vdash \texttt{(head (divs n))}^1 \Downarrow \texttt{v}, \Gamma_n}$$

Figure 5.3: Evaluation tree for `head (divs n)` in Figure 5.1. Superscripts refer to indices of expressions in the Section 5.3.3 example.

5.3 Approach

5.3.1 Evaluation Trees

The approach taken by CSI: Haskell is aimed at the collection of just enough data at runtime to reconstruct the global *evaluation tree* of a program. Lazy functional program evaluation can be viewed in terms of an evaluation tree: the evaluation of each expression requires the evaluation of its subexpressions whenever those expressions are needed to produce output [218]. For Haskell, this evaluation has been linearized using implementations of machines such as Sestoft's lazy abstract machine [219], placing evaluation trees on sound theoretical foundations and (by now) a robust amount of experience. Re-using the theoretical structures lends itself for the application of debugging too: For debugging, a tree-like view of the expressions and the order of evaluations for each component is an important part of understanding the programs and how they behave. This is especially relevant when the programs fail and throw an exception at runtime, e.g. the evaluation tree in Figure 5.3.

This tree shows how evaluation proceeds by resolving the functions to be used in the relevant context (using big-step semantics, denoted by "$\Downarrow$" for readability), which are then applied to the fully resolved value of their arguments, resulting in their final value.

5.3.2 Trace Data

To collect the data used in constructing the trace, we extend HPC, the Haskell Program Coverage built into GHC by Gill et al. [196]. HPC divides the source code into *expression boxes*, which are extracted during compilation and stored in an associated `mix` file. The code is then instrumented with additional instructions in the intermediate language (C- -) to add a *bump* to the appropriate array value when the evaluation of an expression starts (i.e. its output is demanded). At runtime, HPC maintains a module-per-module in-memory array, with one entry per expression in the original program. Whenever an expression starts to be evaluated, the corresponding array entry is incremented with the bump instruction, allowing HPC to track the coverage of programs [196]. CSI: Haskell adds three additional in-memory arrays to the runtime system, along with additional bookkeeping, the *trace array*, *evaluation depth array*, and *global index array*. An example of these for the program in Figure 5.1 is provided in Section 5.3.3.

The Trace Array

The first additional array holds the trace itself, a log of values corresponding to the expression boxes defined by HPC. This array contains an entry whenever an expression starts being evaluated, and another entry whenever an expression finishes being evaluated to the outermost constructor. Each entry represents an explicit expression in the source code, which is the same as that used for the original HPC coverage: for any single expression e, the original coverage tracks the number of times that expression is evaluated. For example, if we look at an expression e_i with $i = 5$, at the beginning of the evaluation of e, the index number 5 would be incremented in the corresponding array in the `tix`-file. With our additions, the index 5 is written in a trace array both when the expression starts to be evaluated and when it has finished evaluating. Note that since Haskell is non-strict, the evaluation of an expression might not return a fully evaluated value, but rather a weak normal form represented by a constructor whose components might further, not yet fully-evaluated thunks. To log these evaluations, an additional register is introduced, in which the (possibly partial) value of an expression is saved. The completion is then recorded and the register is returned. This allows us to log the completion of evaluations even when they would have immediately returned, at the cost of additional overhead at runtime.

The Evaluation Depth Array

The second array keeps a log of the current *evaluation depth* before the start of the evaluation of an expression and the depth before the completion of the evaluation of an expression. Using the two in conjunction, the evaluation depth and trace arrays allow us to reconstruct a partial view of the full evaluation tree of the program and determine whether an entry in the trace array corresponds to the start of evaluation or the completion of evaluation of the indicated expression. It also lets us determine which evaluations have started and not finished, allowing us to determine the current call stack in terms of expressions. This allows us to see which evaluations were started and finished in the same subtree of the *evaluation tree*, allowing us to highlight the branches of the evaluation that are "close" in the tree.

The Global Index Array

The third array tracks a global counter, associating each index in the other two arrays with a unique integer timestamp. This allows us to reconstruct a global trace across modules, by gathering the trace for each module and sorting by the global counter.

Trace Length & Circular Buffers

Keeping track of an arbitrarily long run of a program would require a trace entry for each expression evaluated. For long-running programs, this would require an excessive amount of memory. As noted in the introduction, errors usually involve *recently evaluated data*. By keeping the length of the arrays constant and introducing a modulus operation to the running index, we effectively treat them as circular buffers. This allows us to keep track of only the most recently evaluated locations at the time of an error, giving us a "window" into what the program was executing right before the error occurred. Configurable with a compiler flag, this allows users to select how much memory overhead they are willing to trade off for a longer trace. Alongside the computational impact, there is also an information trade-off: Some errors are captured only in longer traces, but unnecessarily long traces form a barrier to understanding. We investigate both trade-offs in Section 5.4.

5.3.3 Example

As an example, consider the evaluation of the expression `head (divs n)` and its evaluation tree shown in Figure 5.3. Here, E_1 corresponds to the expression superscripted with 1, that is, `head (divs n)`, E_2 to `head`, E_3 to `(divs n)`, and so on. Note that each expression has an annotation, as well as each of its subexpressions. In the interest of brevity, E_6 and E_7 are not shown, nor are any of their subexpressions. Using the annotations provided in the figure, a successful evaluation trace would be $[E_1, E_2, E_2, E_3, E_4, E_4, E_5, E_5, ..., E_3, E_1]$, with the associated evaluation depths $[0, 1, 2, 1, 2, 3, 2, 3, 2, ..., 2, 1]$. The global trace array would simply be $[1, \dots]$, since there is only one module involved. The evaluation proceeds as follows: At the start of evaluation, the evaluation depth is 0. We start by evaluating `head (divs n)`, indicated by E_1. The evaluation depth is now 1. E_1 demands evaluation of `head`, i.e. E_2. Since we started evaluating E_2 and have not finished E_1, the depth of the evaluation is now 2. The function `head` is from a library, which returns directly, indicated by E_2, and the evaluation depth decreases to 1. Now the implementation of `head`, `head'` demands its first argument, which causes evaluation of `(divs n)`, i.e. E_3, resulting in an evaluation depth of 2. This continues until E_3 completes, which in turn lets E_1 complete, and the program is fully evaluated. *However*, if E_3 results in an empty list, the evaluation $\Gamma \vdash$ `head' xs'` $\Downarrow v$ will throw an exception, aborting execution *before* logging that E_1 finished. The trace will show that E_3 was the last expression to complete evaluation before the error.

5.3.4 Persistence and Tix Upgrades

As CSI: Haskell is integrated with HPC, we also extend the `tix` file format that HPC generates to persist information between runs to include the trace and evaluation depth information.

Apart from changes to the `tix`-format, the tracking is noninvasive and requires no modification of the programs on behalf of the user. Setting the size of the trace buffers to a sufficient length can be used to generate traces across multiple runs of a program. These extended `tix` files, along with the associated `mix` files that store the expression boxes, can be parsed by external tools for further analysis and presentation. One such presentation is a SARIF file derived from the a trace, allowing further integration of Haskell traces into IDEs [220]. This has been explored with a short prototype by the authors and is feasible. With respect to the scope, we consider it future work (see Section 5.5). A non-textual presentation of the trace could be to visualize the behavior of the program as a graph, as shown in Figure 5.5.

5.3.5 Output

The additional information can be accessed via runtime reflection using the GHC-API, and consumed by external tools such as automatic program repair tools, testing frameworks, and IDEs. As one immediately accessible application, we adjust the current runtime error printing in GHC and add a message detailing the recently evaluated locations. Using the trace array in conjunction with the evaluation depth array, we generate a list of recently evaluated locations. By comparing the current evaluation depth on an error and the evaluation depth array, we determine the involved expressions whose evaluation was demanded by the expression on top of the call stack at the time of crash. We label the

```
divs: Prelude.head: empty list
CallStack (from HasCallStack):
error, called at
libraries/base/GHC/List.hs:1749:3 in base:GHC.List
errorEmptyList, called at
libraries/base/GHC/List.hs:89:11 in base:GHC.List
badHead, called at
libraries/base/GHC/List.hs:83:28 in base:GHC.List
head, called at Divs.hs:10:17 in main:Main
CallStack (from -prof):
Main.smallestDiv (Divs.hs:10:17-29)
Main.main (Divs.hs:13:15-28)
Main.main (Divs.hs:13:8-29)
Main.CAF (<entire-module>)
Recently evaluated locations:
Divs.hs:4:25-4:26  ... = []
Divs.hs:4:16-4:21  |...,i == n,...=... (was matched)
repeats (11 times in window):
Divs.hs:4:9-7:28   Main:divs>go
Divs.hs:7:21-7:28  ... = go (i+1)
Divs.hs:5:19-5:21  ...else d i
Divs.hs:8:9-8:28   Main:divs>d
Divs.hs:5:16-7:28  ... = if d i...
Divs.hs:4:16-4:21  |...,i == n,...=... (not matched)
Divs.hs:4:9-7:28   Main:divs>go
Divs.hs:3:1-8:28   Main:divs
Previous expressions:
Divs.hs:10:1-10:29  Main:smallestDiv
Divs.hs:13:1-13:29  Main:main
```

Figure 5.4: The improved error message includes a list of recently evaluated locations. The preceding number is the index of the expression in the `mix` file and is used to distinguish different expressions at a glance.

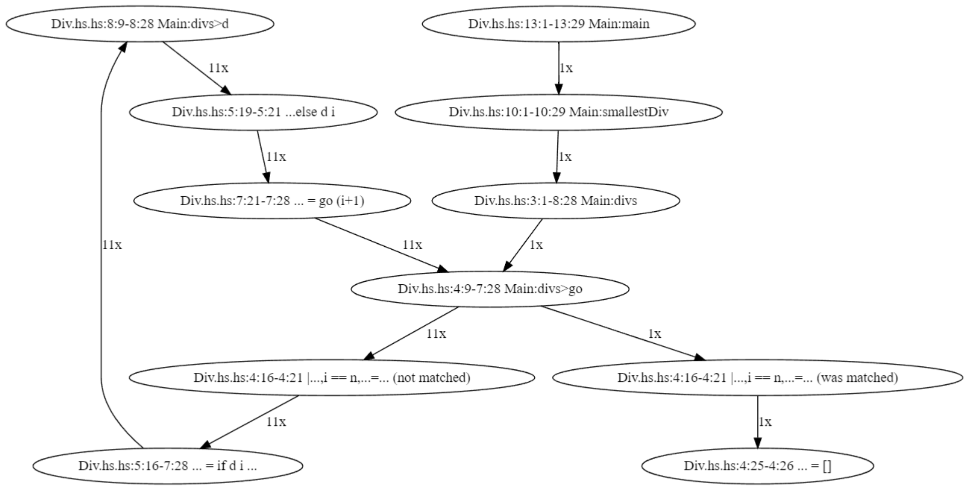

Div.hs.hs:8:9-8:28 Main:divs>d
Div.hs.hs:13:1-13:29 Main:main
11x
1x
Div.hs.hs:5:19-5:21 ...else d i
Div.hs.hs:10:1-10:29 Main:smallestDiv
11x
1x
Div.hs.hs:7:21-7:28 ... = go (i+1)
Div.hs.hs:3:1-8:28 Main:divs
11x
11x
1x
Div.hs.hs:4:9-7:28 Main:divs>go
11x
1x
Div.hs.hs:4:16-4:21 |...,i == n,...=... (not matched)
Div.hs.hs:4:16-4:21 |...,i == n,...=... (was matched)
11x
1x
Div.hs.hs:5:16-7:28 ... = if d i ...
Div.hs.hs:4:25-4:26 ... = []

Figure 5.5: A graphical representation of the trace in Figure 5.4 generated by CSI: Haskell and an external script.

rest as "previous expressions", whose evaluation was complete before the evaluation of the expression on top of the call stack started and therefore were not triggered by the expression on top of the call stack. As an example, Figure 5.4 shows the new output generated for `divs` from Figure 5.1, which shares the evaluation tree with the example above.

5.3.6 Summarization and Presentation

Since we track all evaluated expressions, the traces can become quite long. To effectively display error messages, filtering and summarizing traces is important. The summarization of traces is a rich field [221, 222], but often involves the full instrumentation of the program from the beginning to the end, while CSI: Haskell has a limited window of recently evaluated locations. To be useful as the default when printing error messages, the summarization of the traces must be done quickly and efficiently, avoiding unnecessary delay when reporting errors. The current approach in CSI: Haskell is to remove all unconditionally evaluated expressions done before the last branch. This makes the traces much shorter while keeping the relevant information about the evaluated expressions immediately preceding the error. Another summarization that CSI: Haskell implements is to merge repeated patterns that occur in loops and indicate them as repetition in the output, with the caveat that it only captures repetition in the "window" that the trace offers and may miss out on some longer patterns. This technique struggles when there is variation in the loop, such as when it is conditionally different for each iteration, e.g. cases for odd and even numbers. We aim to mitigate such variations using graph-based trace modeling and using more of the information available on the structure of the code during summarization (see Section 5.5). As described earlier, we used the evaluation depth at the time of a crash in conjunction with the tracking of the evaluation depth to segregate expressions that were evaluated at the current evaluation depth or lower and those that occurred earlier. Looking at the evaluation depth array also allows us to construct a partial notion of the call stack at the time of the crash, though some information might have been lost due to truncation in long-running programs. In this way, we can track the call stack for any program without manual annotations of `HasCallStack =>` throughout the code. Since CSI: Haskell is integrated into the compiler and runtime system itself, it can be easily applied to external Haskell libraries and dependencies simply by adding an additional flag during compilation. This helps developers trace issues that originate in external libraries and understand the interaction of their code with the library.

As for presentation, the current implementation reads the relevant locations from the source file, and displays them in a manner appropriate to their form, whether it is a branching if statement, guard or qualifier in a list comprehension or a non-branching expression. To further shrink the output, we only show non-branching expressions up to the last branching expression in the trace. This allows the focus to be on the control flow up to the point where the evaluation of a non-branching expression might have caused the error. When the total number of evaluations exceeds the window, a short statement is appended to the error message showing the total number of evaluations and suggesting to increase the trace length before rerunning. We stress that the current presentation is a prototype and outline the need for further research in Section 5.5.

5.3.7 Data

Apart from the motivating example in Figure 5.1, we draw data from the `nofib-buggy` data set [70]. In this data set, Silva introduced artificial bugs of various categories to the data points of the `nofib` benchmark [69] used in the GHC test suite.

We utilize a subset of 21 bugs summarized in Table 5.1. Our biggest exclusion criteria of the original `nofib-buggy` was the category of *non-termination*: Since our evaluation is based on crashes, non-termination does not provide the output we need. Similarly, `StackOverflowExceptions` are errors of the environment, not necessarily in the program. These exceptions come from the runtime system itself and not from the program, so such exceptions were excluded as well.

Lastly, for ease of comparison, we modified programs that merely produced incorrect results to fail with an exception using an `assert`. These assertions are constructed using the console output (`stdout`) of the correct programs. Due to the lack of annotations, the call stacks in these examined cases are all trivial and only show the call for equality in the assert, but the evaluation traces often span relevant locations in the code. We admit that the assertions based on string comparison are neither sophisticated nor best practice. In the spirit of a vertical prototype, we aimed to see *"can evaluation traces help with testing?"* Despite looking a bit ad-hoc, the insights might be as valuable as the inspection of runtime errors: a healthy project should address problems in the test suite and not at runtime. Additions to the testing toolkit may pay off earlier than post-mortem debugging tools.

Table 5.1: Overview of the `nofib-buggy` programs used

nofib-buggy / Error Type	Imaginary	Spectral	Real
Exception	paraffins digits-of-e2	sorting primetest	anna
Assert	digits-of-e1 rfib tak integrate gen_regexps bernoulli wheel-sieve1 wheel-sieve2 x2n1	chichelli fish minimax	gg parser reptile lift

5.4 Initial Results

To analyze the results, we recompiled the `nofib-buggy` data set with a fork of GHC and HPC that implements CSI: Haskell as described in Section 5.3. After obtaining crash logs, two authors looked at each log separately, deriving data and judging the merits of the new output. All code, data points, logs, and evaluations are provided in the companion package archived at `https://doi.org/10.5281/zenodo.10090375`. The remainder of this section covers the summary and highlights of the findings.

Summary & Overview Table 5.2 presents the results achieved by the `nofib-buggy` data as shown in Section 5.3.7: of the 21 data points, 13 have the location of the error appear within a trace length of 50 and 19 in traces of length 1000 visualized in Figure 5.6 and Figure 5.7.

Visible in Figure 5.7[3] is that in data points with `exceptions` appear much earlier than their assert counterparts, and most issues are covered at the top of the exceptions. For most of the data points, the displayed position in the log was quite prominent (usually within the first 10 lines).

The required trace length did not directly depend on the size of the program, but rather on the amount of thunks that the program builds up during evaluation. We can see this behavior in Figure 5.9. Naturally, the `real` data points produce a lot of thunks and evaluations due to their complexity, but some of the spectral and imaginary data points (artificially) produce large amounts of thunks (`spectral/minimax`) or evaluations (`imaginary/rfib`). For a helpful exception, it is necessary that both the start of evaluation and end of evaluation of the involved expressions be in the window of recent evaluations. But the window should be *as small as possible* - as seen in Table 5.2. For both `reptile` and `minimax` the position of the faulty statement appears later for a trace length of 1000 compared to the trace length of 50.

[3]Note the log-scale on the x-axis

Table 5.2: Summary of `nofib-buggy` results. LOC indicates the location in the output after the initial exception, and minimum trace length the shortest length in which the error location appears out of $[25, 50, 100, 500, 1000]$.

data point	Uses assert	minimum trace length	LOC 50	LOC 1000	LOC 1000 Strict
imaginary					
bernoulli	Y	50	6	6	6
digits-of-e1	Y	500	-	11	21
digits-of-e2	N	25	1	1	-
gen_regexps	Y	-	-	-	-
integrate	Y	-	-	-	-
paraffins	N	500	-	24	24
rfib	Y	500	-	4	7
tak	Y	25	4	4	2
wheel-sieve1	Y	25	2	2	-
wheel-sieve2	Y	50	7	8	-
x2n1	Y	25	2	2	2
spectral					
cichelli	Y	1000	-	36	-
fish	Y	25	3	3	1
minimax	Y	50	28	260	-
primetest	N	25	2	2	2
sorting	N	25	1	1	1
real					
anna	N	25	1	1	-
gg	Y	25	1	1	-
lift	Y	500	-	32	-
parser	Y	500	-	13	19
reptile	Y	25	29	35	94

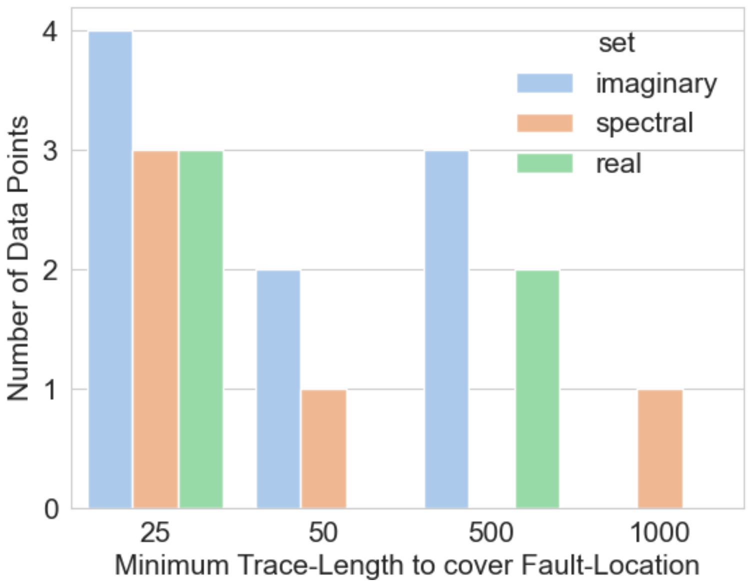


Figure 5.6: Minimum trace length to cover the error

Performance We provide a summary of the compute time used in Figure 5.8 and of the allocated (peak) memory in Figure 5.10. All reported values are derived from a set of five measured runs on a dedicated machine, dropping the highest and lowest values (outliers) and averaging the remaining three. Measurements were conducted with the Linux `/usr/bin/time` executable and the bash `time` command on a cloud-based machine with 32GB of RAM and 6 Intel Xeon E312xx @2GHz 64bit vCPUs. We also performed a set of runs with profiling turned on, using the GHC flags `-fprof`, `-fprof-auto`, and `-fprof-auto-calls`, which yielded comparable increments. As profiling introduces more side effects, we prefer to report the non-profiling numbers in this work. Profile performance measures are included in the companion package.

Figure 5.8 is a kernel density estimate plot [223] summarizing the distribution of the calculated time deltas for all data points. It presents a smooth growth of wall-clock time to increase the trace length, with the majority of data points needing between ~100% and ~300% longer. We can also observe that *outliers* move respectively, keeping their relative position throughout increasing trace lengths. In particular, the *hungriest* data point was `rfib` from the imaginary data set that needed 760% longer to finish. We take a closer look at `rfib` in the paragraph *limitations*.

The box plot in Figure 5.10 shows the memory usage and we observe a trend towards slightly higher resource need. The data points in the imaginary set allocate ~15% memory at peak use and the data points in the spectral set ~60% in the median. For the data points in the `real` set the biggest difference was found, with one data point exceeding twice the

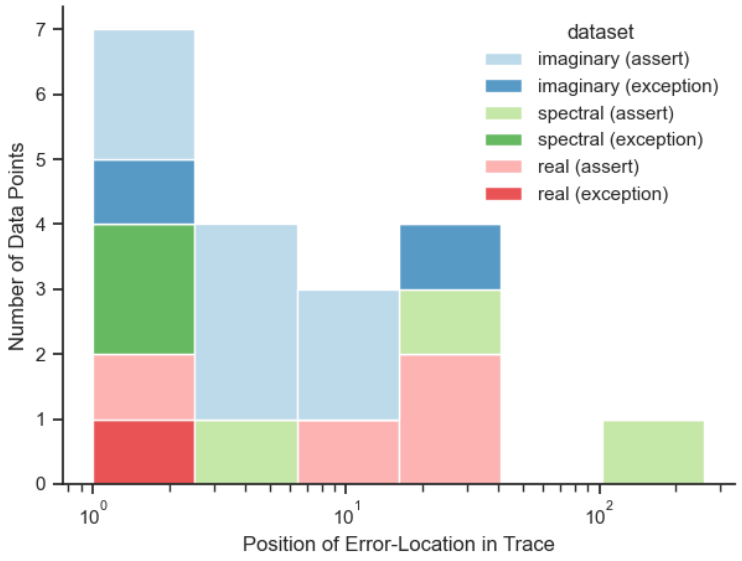


Figure 5.7: Histogram of position in the trace by data set and error type. Note that the x-axis is logarithmic.

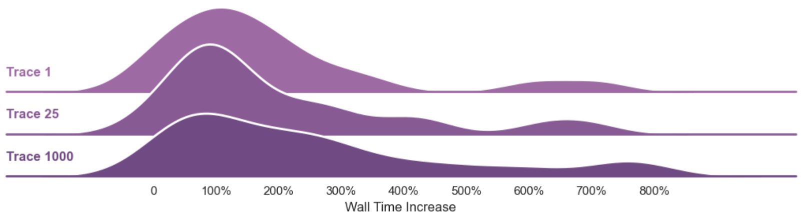


Figure 5.8: Kernel density estimate plot of increased compute time with varying trace lengths

memory usage. Due to the small amount of data points, and each data point in the `real` set being unique, we don't want to infer general assumptions about the memory usage of these programs.

Our recommendation is to investigate these individually when necessary. We also observe that memory use grows in general with the use of traces, but the size of the trace does not have a huge impact on the preliminary results. The overhead originates from data collection, and not from storing and bookkeeping.

We measured the size of the binary compiled for each program. The difference in most programs was negligible ($\leq 1M$), but we must note that the size increase can be notable for longer trace lengths[4]. For a run gathering *full-traces* (i.e., trace length set to 100k), each

[4]This is due to an in-binary representation of the tick-arrays, to address internal mechanics such as garbage

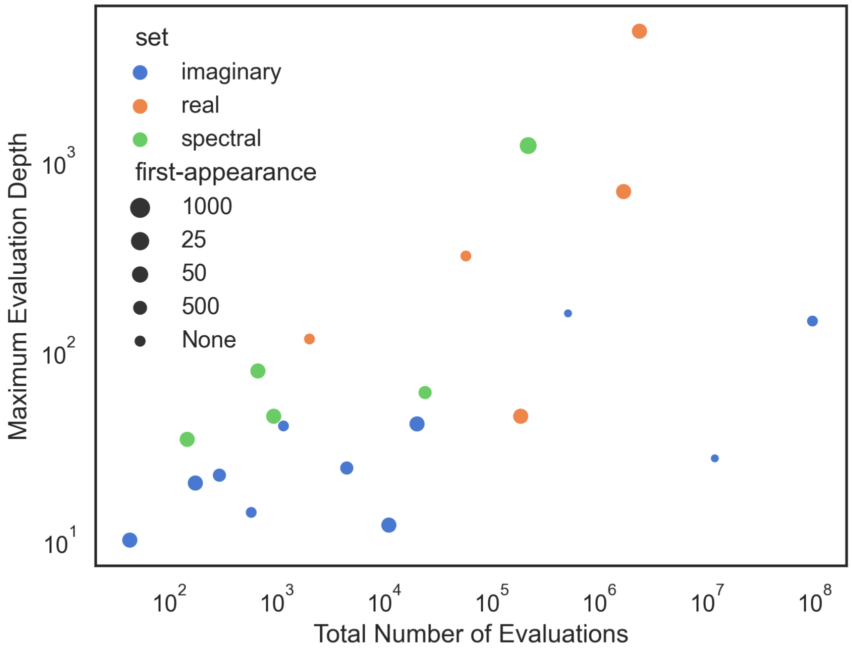


Figure 5.9: Distribution of maximum evaluation depth and total number of evaluations

binary grew between two and ten Mb.

As traces are used to locate errors, the overhead presented in this work is expected to occur during development and maintenance and will not affect production environments.

Highlights Among the best results are two data points for the `spectral` data set, `sorting` and `primetest`. The errors are division-by-zero and a non-exhaustive pattern match, respectively. These errors have little information by default, with no location or stack trace. The extended output (see `sorting` in Figure 5.11) with the trace information that CSI: Haskell adds shows the starting positions where incorrect data was produced and does so quite precisely.

The second group of promising results is demonstrated by the data points for `minimax` and `gg`: the bug introduced to minimax consists of not applying a minimax algorithm to Tic-Tac-Toe, but instead performing a *minimin*. Figure 5.12 catches this behavior by repeating the `Game:min'` function, while we would expect alternating min and max functions. This is not exactly unique to evaluation traces, but we get “`a bit of coverage for free`”. Without enhanced traces, this would also be spotted when running a HPC coverage report and seeing the uncalled max function.

Similarly `gg` from the `real` data set uses a wrong variable, leaving large parts of a **`where`** block unevaluated.

collection. For normal coverage, the addition is bounded by the modules and their expressions, while our additions can vary in length and thus grow the binary to varying degrees.

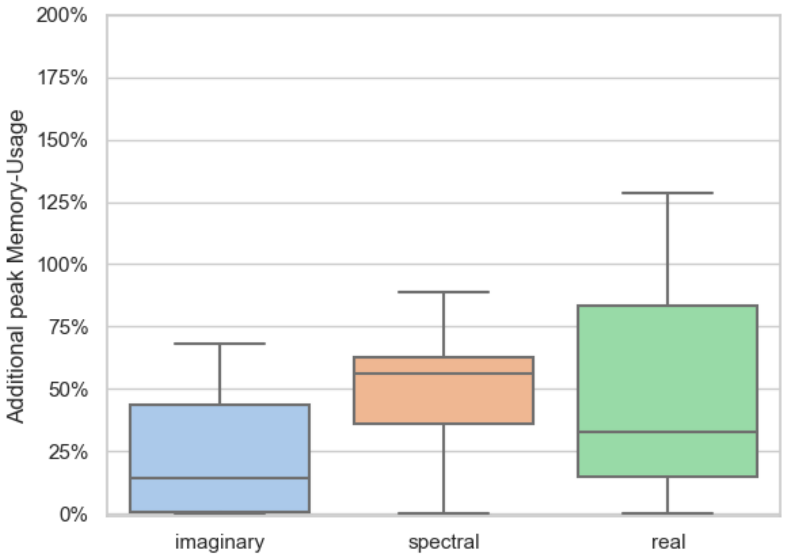


Figure 5.10: Additional memory usage per data set for a trace length of 1000

This second group of bugs can be quickly noticed using program coverage, and it is possible to get the same information from a coverage report. Unfortunately, we must admit that this is an enlightened guess – we knew what was going wrong, and thus we found patterns and clues in the traces. These bugs can be found quite easily when program coverage is visualized, and thus we hope that a visualization of traces would also yield such benefits, motivating more complex tooling.

Before we leave the highlights, we want to emphasize the possibilities of generated traces for mechanical evaluations. Some of the traces presented throughout this paper are a bit crowded or hard to understand, but nevertheless, they contain the information necessary for better fault-localization and other warnings. We see potential tooling that spots mismatches in coverage and evaluation, or that warns about potential performance issues with a lot of thunks, like we see in the `rfib` example.

Full Evaluation Record versus Suffix A thread running through this paper is the initial scenario: is it enough to determine what happened immediately before a crash in order to locate the fault? We consider the recent evaluations the *suffix* of all evaluations. Shown in Table 5.2, the errors appear in 90% of the data points examined. Furthermore, a relatively short trace of only 50 locations per module is sufficient in 62% of the cases. When

[5]Note that some of the right hand sides are missing here, due to a mismatch between the locations reported by HPC and the actual location in the file... caused by the mixing of tabs and spaces! Fixing this is beyond the scope of the prototype.

```
Main: divide by zero
Recently evaluated locations:
./Sort.hs:146:25-146:25  2
./Sort.hs:146:23-146:23  2
./Sort.hs:146:22-146:26  (2-2)
./Sort.hs:146:14-146:26  k `div` (2-2)
Previous expressions:
./Sort.hs:146:5-146:26   Sort:heapSort>div2
./Sort.hs:128:52-128:67  ... =
repeats (4 times in window):
./Sort.hs:128:5-132:84   Sort:heapSort>to_heap
Main.hs:14:36-14:43      ... =
Main.hs:13:5-22:57       Main:mangle>sort
Main.hs:10:1-22:57       Main:mangle
Main.hs:5:1-7:33         Main:main
There were 668 evaluations in total but only 86 were recorded.
Re-run again with a bigger trace length for better coverage.
```

Figure 5.11: The improved error log for `Sorting` - the first locations of the trace are the precise consumers and producers of the division-by-zero error[5].

running the program in Figure 5.13, there are **95845589** evaluations in total, of which only **500** were in the final recorded trace, which is enough to cover the faulty location. Despite losing analytical benefits of the complete trace, we are able to locate the fault while keeping only **0.0005217%** of the trace. We thus recommend capturing the only the last evaluations, with a little fine-tuning in trace lengths depending on the number of evaluations (unless necessary for follow-up analysis).

Strict vs. Lazy Behavior For comparison, we conducted experiments with the `-XStrict` language extension, in addition to the `-fno-strictness` and `-fno-full-laziness` flags to observe changes in evaluation behavior. Without the extension, the trace for each data point was identical, with or without the flags. Our initial (naive) assumption was that for strict programs consumption and production of errors would align, resulting in always perfect locations. The heavy-handed use of the `-XStrict` extension meant that some of the programs would no longer terminate, as many of them rely on laziness to be computable. This resulted in 8 data points that do not finish when strict evaluation is forced.

Among the terminating data points, we see mixed results - `fish` and `tak` perform slightly better, while some evaluations appear later than in their non-strict configuration. We attribute this to the general offset in consumption and production that is also observed in strict & imperative programming languages (e.g., in the work of Zhang et al. [224]): The distance between fault-introduction and fault-consumption also exists in Haskell, but non-strict evaluation can *shrink* the gap between fault-introduction evaluation and fault-consumption evaluation.

To paraphrase, there is always a gap between fault and error, but non-strict evaluation can bridge this gap by postponing evaluations.

```
Main: Assertion failed
CallStack (from HasCallStack):
assert, called at Main.hs:12:5 in main:Main
CallStack (from -prof):
Main.main (Main.hs:(10,1)-(12,57))
Recently evaluated locations:
./Game.hs:32:59-32:59
./Game.hs:31:30-31:30
./Game.hs:36:23-36:23   e
./Board.hs:57:53-57:56  Board:showsPrec
./Board.hs:57:53-57:56  Board:show
./Game.hs:31:27-31:31   ... =
./Game.hs:31:9-33:71    Game:best>best'
./Game.hs:32:51-32:65   ... =  s bs ss
./Game.hs:32:37-32:47   |..., s') = best,...=...
(was matched)
./Game.hs:63:15-63:18   ... = OWin
./Game.hs:63:1-68:47    Game:min'
./Board.hs:57:70-57:71  Board:(==)
./Board.hs:26:33-26:55  ... = [[r1,r2,insert p r3 x]]
./Board.hs:23:26-23:46  |...,not (empty pos board),...=...
(not matched)
./Board.hs:41:24-41:27  ... = True
./Board.hs:39:1-42:18   Board:empty'
./Board.hs:36:26-36:36  ... = empty' x r3
./Board.hs:34:1-36:36   Board:empty
./Board.hs:23:1-26:55   Board:placePiece
./Game.hs:31:9-33:71    Game:best>best'
./Game.hs:32:51-32:65   ... =  s bs ss
./Game.hs:32:37-32:47   |..., s') = best,...=...
(was matched)
./Game.hs:63:15-63:18   ... = OWin
./Game.hs:63:1-68:47    Game:min'
...
```

Figure 5.12: The improved error log for `minimax` - notice the repetition of `min'`, without the appearance of a `max'`.

Thus, laziness modulates the distance between bug occurrence and consumption. This affects our configuration for the trace lengths: for short traces, a faulty location can be covered but might have been rotated out of the current trace buffer. With long and short traces alike, there is a chance that the location is reported later in the output, missing the user's attention. An example of this is `paraffins`, where sharing is a source from which an incorrect value is evaluated long before it is used. Potentially, this can be further adjusted by introducing more laziness into programs by making other adjustments, such as explicitly disabling sharing [225].

Limitations The first limitation is represented in `rfib`, which needed a surprisingly long trace for a rather simple program (calculating Fibonacci numbers). Inspecting Figure 5.13, we observe that the `rfib` program performs a cascading recursion and postpones evaluation, with a lot of redundant re-computation producing a lot of thunks. For our current reporting, it is necessary that the trace length covers a coherent sequence (i.e., covers both creation and resolution of thunks), but this coherence is only perceived when the trace length is long enough. To mitigate this, users are presented with a message when we detect that the trace length is not long enough to cover the entire execution of the program.

We are slightly divided about this topic: On the one hand, many functional pearls utilize recursion and laziness, and thus will trigger a similar behavior for our traces. Especially for these cases, the insights in the evaluation would have great potential for learning and visualization. On the other hand, recursion of this type should usually be written in a tail call-optimized fashion (see Figure 5.14), which is less graceful but is preferable in performance and also benefits the traces introduced by this work.

```haskell
nfib :: Double -> Double
-- BUG: The following line contains a bug:
nfib n = if n < 1 then 1 else nfib (n-1) + nfib (n-2)
```

Figure 5.13: nofib-buggy's `rfib`: The code first builds up a large number of thunks using recursion before completing any evaluation, posing a challenge for evaluation traces

```haskell
fib :: Double -> Double
fib n = fib' n 0 1

fib' :: Double -> Double -> Double -> Double
fib' 0 a _ = a
fib' 1 _ b = b
fib' n a b = fib (n-1) b (a+b)
```

Figure 5.14: Alternative Fibonacci implementation that utilizes tail-call optimization

Current evaluation traces are also limited by *sharing* [218]. Consider the function in Figure 5.15: Here, the call to
`three_partitions (n-1)` is used in line 3 to generate triples of integers to partition

```
1  rads_of_size_n radicals n =
2    [(C ri rj rk)
3      |(i,j,k)  <- (three_partitions (n-1)),
4       (ri:ris) <- (remainders (radicals!i)),
5       (rj:rjs) <- (remainders
6                      (if (i==j) then (ri:ris) else radicals!j)),
7       rk      <- (if (j==k) then (rj:rjs) else radicals!k)]
```

Figure 5.15: Part of the paraffins example showcasing *sharing*

a list. There is an error in this function that causes the *k*s to be invalid out-of-bound indices for the `radicals` list. Since `i` and `j` are used in lines 4 and 5 respectively, the triplets are evaluated and then the *same result* is *shared* later when the invalid `k` is used in line 7. This means that the distance from production to consumption increases due to sharing, which means that there will be more unrelated evaluations prior to the error in the trace. This could be addressed by post-processing traces and removing those evaluations that are "unrelated" (such as those in lines 4 and 5), but this would require a richer view of which values are involved in each expression. This view could possibly be created by adding *provenance* information to the values (see Section 5.5).

We spare the reader examples for readability, but it is easy to imagine that evaluation traces are not always useful. Mainly we see that traces either don't contain relevant information, or there is a major overhead attached, and we do not expect people to work through 100+ lines of trace information. A prominent example of this issue is `minimax`, for which the fault-location is *covered*, but only in the sense that the relevant statement was touched. It is not immediately clear what to do, as the issue originates from the unused parts of the program. Providing too much information can also discourage developers from reading error messages[201]. and time spent in the wrong places is a waste and reduces trust in traces and error messages[199, 203]. Thus, another crucial improvement is to determine what criteria constitute the relevance of a trace for the problem and only present them when applicable.

Discussion Based on the limitations and highlights, our current suggestion is to show evaluation traces for certain types of exceptions. The prime candidates are index errors, failed pattern matches, and exceptions for dynamically typed values, such as those from `Data.Dynamic`. These programs showed great results without any real overhead and are a perfect point-in-case for evaluation traces.

From the data points that yield wrong results and have been investigated using assertions, we see a trend that unit-level tests provide better evaluation traces than system-level tests. In particular, the `nofib-buggy/real` data points that use a string comparison for `stdin` and `stdout` did not really benefit from the evaluation traces. We expect that lower-level tests and assertions are far more useful, especially when combined with a sound approach to testing and coverage.

We also recognize the size of the errors and sometimes *mechanic* coverage of traces - as shown in Figure 5.6, some faults require long traces to be covered, and the resulting output is bound to be verbose. We do not consider these traces to be *actionable* due to their size and the effort necessary to comprehend them. Nevertheless, we hope that the tools can pick up the verbose trace information to further filter and visualize critical elements of the code.

Currently, **`Prelude`** provides two functions `error` and `errorWithoutStackTrace`. We suggest expanding this to errors with (only) evaluation traces and a combination of stack and evaluation. The choice is left to individual exceptions as to whether evaluation traces make a worthwhile addition. Another necessary step is to provide a starting guide on how to read and use evaluation traces. Typically, people google their exceptions to find some help [199, 226], but with this newly introduced addition, that is not an option. Thus, some kind of central starting point and tutorial should accompany any changes.

5.5 Next Steps

Evaluation Asserts A potential new area is the construction of *evaluation asserts* - using the enhanced coverage information, and a known expression in the source code, one can formulate tests that check for the (full) evaluation of an expression. While this comes with some difficulties in implementation (e.g. not evaluating the expression through the assert), there are certain areas where this can support developers: One application of this is in debugging, for which developers might want to check the *state* of their variables. Although this is not exactly in the spirit of functional paradigms, existing research [227] shows that Haskell developers often fall back to imperative approaches during debugging. Furthermore, we face functions such as `foldr`, `foldr'` and co. whose results are identical, but their internal traversal strategies differ. Another application is for systems that revolve around or provide evaluation strategies such as GHC itself. It can provide capabilities to test, e.g. BangPatterns and data structures.

Study An obvious next step is a detailed study. The examples presented in this work highlight initial results but hardly represent *the real world*. Thus, the authors plan to conduct a broader study utilizing most of the `nofib-buggy real` data points and modern examples from the HasBugs data set [228]. Such a study should help to grasp how often evaluation trace information covers bugs and, if so, how long the trace should be.

Furthermore, a study is necessary to estimate the computational feasibility. Additional instrumentation always comes with a performance cost, and the exploration in `nofib-buggy` is unfortunately not representative of a complete evaluation.

Provenance of Values One problem that arises when strictness and sharing are involved is that an expression might have been evaluated long before usage, such as the k in the paraffins example (Figure 5.15). This means that many unrelated evaluations occur between the production and consumption of a value, making the trace less useful to find the source of the error. One way to address is to attach *provenance* to values, highlighting the part of a trace involved in the production of any values touched on in an error.

Environment Integration and Presentation This work presents basic steps and low-level implementation for evaluation traces, but the findings might be *diamonds in the rough.* Especially for longer traces of the `real` data points, guidance and assistance are necessary. We touched on potential tools and extensions throughout the work, which we would like to summarize:

First, summarizing and filtering traces is necessary to keep the output human-readable, especially for long traces. Solutions could cover filtering modules or limit the depth and width of the presented evaluation tree. In addition to trace data, there are opportunities to accumulate data from multiple sources (test success, program coverage, etc.) and perform program slicing [229]. This is essential to scale to large programs. Another important integration is with test and build frameworks. At the moment, traces are reported on runtime exceptions, which is arguably not the best state of a program to be in. Most of the time, software engineering utilizes tests, and therefore evaluation traces should be presented in an accessible way for test failures. We hope that in the future, Haskell developers can write unit tests and investigate their evaluation for anomalies, finding potential issues before they become problems. Lastly, we did early sketches of integrating SARIF[220] based on `tix`- and `mix`-files with a prototype. Transforming the information is quite easy and can then be picked up from other popular tools such as VSCode. Especially in light of the Haskell Language Server that also targets VSCode, we hope that representation of coverage and evaluation in the IDE can be a result of this work. Such tools should not only be based on solid data (this work), but must also meet standards and needs of developers, drastically expanding the scope by a necessary user-study or other ways to capture user-experience. Thus, this work focus‘ lies on the creation, maintenance and mapping of evaluation-traces.

Automated Fault Localization Although this work covers fault-localization as a manual task, automated fault localization is a popular research topic with often great results [230, 231]. Automated fault localization often exploits a spectrum of coverage per test to find code that is *suspiciously often* involved in failing tests. These approaches are based on the program coverage of strict languages (Java, C), and revolve around expression or statement coverage. Directly copying these approaches might not be applicable to Haskell — due to laziness, we might call expressions but not evaluate them. Thus, focusing on evaluation over coverage is necessary to build a spectrum of code that was executed, and not only touched.

Apart from adjustments necessary to reproduce existing approaches, the evaluation information can also form the basis of novel techniques: normal spectrums are *binary*, things are covered or not. With evaluation, we express the concept of full or partial evaluations and can derive a continuous spectrum.

Optimization We are aware that this is merely a prototype implementation. We hope that producing a non-invasive method for gathering and reporting information on evaluation resonates positively in the community, but know that we have made some arbitrary design decisions. In this spirit, we do not consider the implementation *done* but are looking forward to feedback on this work and towards an eventual GHC proposal.

Required trace length estimation One pain point with the current design of CSI: Haskell is that the trace length is fixed and a value must be provided by the developer. One way to address this could be to have a more dynamic trace, discarding entries not involved in the current evaluation and keeping only the parts of the trace which involve values which are currently accessible and have not been garbage collected. This would involve a much deeper integration with the runtime system and memory management, but could be vital for tracing long-running programs, keeping both relevant parts of the trace but still keeping memory requirements manageable. Another approach would be to do static analysis of the program to suggest a useful length for the trace, using the call graph and structure of expressions to approximate the required length within some order of magnitude. This would involve more advanced termination checking than feasible for this chapter, but can reduce the guesswork in finding a good length. In the interim, we suggest using a trace length of approximately 100 for smaller programs and approximately 1000 for larger ones (as suggested by our experiments on the `nofib-buggy` data set) and increase or decrease as necessary.

5.6 Conclusion

This chapter presents an initial implementation to gather evaluation traces and report them alongside current stack traces on runtime exceptions. The approach utilizes *boxes* similar to regular HPC and only requires additional flags for compilation — extending from the program even into dependencies. This novel data was used to improve the runtime exceptions reported with information on the evaluation. We ran the changes on a subset of the `nofib-buggy` data set, investigating at which point of the trace the faulty location was reported. For 19 of the 21 data points, the fault was covered in a trace of 1000 entries long, and most of the locations appeared in the first 50 lines of the trace. In general, valuable information is covered by the trace, but a current limitation is the size and verbosity of the output. Most data points required two to 2 to 3 times more runtime and about 50% more memory. Outliers in performance were based on excessive amounts of thunks and a large number of modules.

Providing evaluation traces can help to spot certain errors, especially those related to lazy evaluation. The examples provided in this paper show that evaluation traces help to establish the chain of events behind certain errors better than a plain stack trace, as due to lazy evaluation the origin of a problem and its occurrence can be offset.

6 PropR: Property-Based Automatic Program Repair

Summary

Automatic program repair (APR) regularly faces the challenge of overfitting patches — patches that pass the test suite, but do not *actually* address the problems when evaluated manually. Currently, overfit detection requires manual inspection or an oracle making quality control of APR an expensive task. With this work, we want to introduce properties in addition to unit tests for APR to address the problem of overfitting. To that end, we design and implement PropR, a program repair tool for Haskell that leverages both property-based testing (via QuickCheck) and the rich type system and synthesis offered by the Haskell compiler. We compare the repair-ratio, time-to-first-patch and overfitting-ratio when using unit tests, property-based tests, and their combination. Our results show that properties lead to quicker results and have a lower overfit ratio than unit tests. The created overfit patches provide valuable insight into the underlying problems of the program to repair (e.g., in terms of fault localization or test quality). We consider this step towards *fitter*, or at least insightful, patches a critical contribution to bring APR into developer workflows.

This chapter was published as *Propr: Property-based Automatic Program Repair* by Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen and David Sands in the *Proceedings of the 44th International Conference on Software Engineering* (ICSE 2021)

6.1 Introduction

Have you ever failed to be perfect? Don't worry, so have automatic program repair (APR) approaches. APR faces many challenges, some inherited from search-based software engineering (SBSE), like overfitting [232, 233], predictive-evaluation in search [234], and duplicate handling [235]. Other challenges are unique to the domain itself, such as deriving ingredients for a fix [236] and producing valid programs [237]. Consequently, APR has open research in all of its core aspects: search-space, search-process, and fitness-evaluation. The research community is shifting its focus towards other solutions, either leaving behind boundaries of search space using generative neural networks [238**?** , 239], or by empirical evidence that fixes are often related to dependencies, not the code itself [240, 241]. Fixes are usually validated by running against the test suite of the program, assuming that a solution that passes all tests is a valid patch. However, Le Goues et al. [242] showed that Program Repair can *overfit*, i.e., that a fix passes the test suite despite removing functionality or just bypassing single tests.

Usually, generated patches are evaluated against a unit test suite of the buggy program [85]. The fitness is defined as the number of failing tests in the suite [162], making a fitness of zero a potential fix. The problem is the quality of the tests — often not all important cases are covered, and the search finds something that passes all tests but doesn't provide all wanted functionality [232]. This is considered an *overfit* repair attempt. A particularly good example for this is the Kali approach [242], that removes random statements of a program. In a later study, Martinez et al. [89] showed that out of 20 of the repair attempts that passed the tests, only one was a real fix. One approach by Yz et al. [243] to address overfitting was to introduce tests generated with EvoSuite [244] to have a stronger test suite, reporting only an improvement in speed, not in found solutions. Unfortunately, EvoSuite introduces a new problem: If the program was faulty (which programs that we are trying to repair are), an automatically generated test suite may assert the faulty behavior and make test-based repairs unable to ever produce a correct program, despite passing the (generated) test suite. Thus, current automated test-case generation is not the be-all and end-all for overfitting in APR.

This work aims to improve APR with addressing the overfitting problem by introducing properties [245] in addition to unit tests. A software property is an attribute of a function (e.g., symmetry, idempotency, etc.) that is evaluated against randomly created instances of input data. Well-written properties often cover hundreds of (unit) tests, making them attractive candidates for fitness evaluation.

We argue that properties can be an improvement to the overfitting challenge in APR. While property-based testing frameworks exist for a range of languages, the practice is particularly natural for functional programming, and widely used in the Haskell community. Therefore, we implement a tool called PropR, which utilizes properties for Haskell-Program-Repair and evaluate the repair rates and overfitting rates for different algorithms (random search, exhaustive search, and genetic algorithms). Our fixes follow a GenProg-like approach [85] of representing patches as a set of changes to the program, with the major difference that our patch ingredients (mutations) are sourced by the Haskell compiler using a mechanism called *typed holes* [61]. A typed hole can be seen as a placeholder, for which the compiler suggests elements that produce a compiling program. As these suggestions cover all elements in scope (not only those used in the existing code), we overcome to some degree the redundancy assumption [236], i.e., the concept that patches are sourced from existing code or patterns, which is common to GenProg-like approaches.

Our results show that properties help to reduce the overfit ratio from 85% to 63% and lead to faster search results. Properties can still lead to overfitting, and the union test suite of properties and unit tests inherits both strengths and weaknesses. We therefore argue to use properties if possible, and suggest to aim for the strongest test suite regardless of the test-type. The patches from PropR can produce complex repair patterns that did not appear within the code. Even patches that are overfit can give valuable insight in the test suite or the original fault.

Our contributions can be summarized as follows:

1. Introducing the use of properties for fitness functions in automatic program repair.
2. Showing how to generate patch candidates using compiler scope, partially addressing the redundancy assumption.
3. Performing an empirical study to evaluate the improvement gained by properties with a special focus on manual inspection of generated patches to detect eventual overfitting.
4. An open source implementation of our tool PropR, enabling future research on program repair in a strongly typed functional programming context.
5. Providing the empirical study data for future research.

The remainder of the chapter is organized as follows: Section 6.2 introduces property-based testing and summarizes the related work in the fields of genetic program repair as well as background on *typed holes*, which are a key element of our patch generation method. In Section 6.3 we present the primary aspects of the repair tool and their reasoning. Section 6.4 presents the data used in the empirical study, and declares research questions and methodology. The results of the research questions are covered in Section 6.5 and discussed in Section 6.6. After the threats to validity in Section 6.7 we summarize the work in Section 6.8. The shared artifacts are described in Section 6.9.

6.2 Background and Related Work

6.2.1 Property-Based Testing

Property-based testing is a form of automated testing derived from random testing [246]. While random testing executes functions and APIs on random input to detect error states and reach high code coverage, property-based testing uses a developer defined attributes called *properties* of functions that must hold for any input of that function [245]. Random tests are performed for the given property: If an input is found for which the property returns false or fails with an error, the property is reported as *failing* along with the input as a counter example [245]. Some frameworks will additionally *shrink* the counter example using a previously supplied shrinking function to offer better insight into the root cause of the failure [245].

There are some variations on property-based testing, e.g. SmallCheck, which performs an *exhaustive test* of the property [41]. QuickCheck approximates this behavior with a

```haskell
prop_1 :: Double -> Test
prop_1 x =
sin x ~== sin (x + 2*π)

prop_2 :: Double -> Test
prop_2 x =
sin (-1*x) ~== -1*(sin x)

prop_3 :: Test
prop_3 = sin (π/2) == 1

prop_4 :: Test
prop_4 = sin 0 == 0
```

```haskell
unit_1 :: Test
unit_1 =
sin π ~== sin (3*π)

unit_2 :: Test
unit_2 = sin 0 == 0

unit_3 :: Test
unit_3 = sin (π/2) == 1

unit_4 :: Test
unit_4 =
sin (-1*π/2) == -1*(sin π/2)
```

```haskell
(~==) :: Double -> Double -> Bool
n ~== m = abs (n - m) <= 1.0e-6
```

Figure 6.1: Comparison of Properties and Unit Tests for sin

configurable number of random inputs (by default 100 random samples). Figure 6.1 provides an example comparison of properties and unit tests of a sine function. The properties require an argument **Double -> Test** and must hold for any given Double. On any single QuickCheck run, 202 tests are performed, forming a much stronger test suite for a comparable amount of code.

A remaining question is whether one cannot just reproduce these 202 tests by unit tests. For a single seed, this is doable — but it is a special strength of properties that the new tests are randomly generated on demand. We hope this addresses the problem of *overfitting* [232], as there are no *fixed* tests to fit on as long as the seed changes. Furthermore, we stress that maintaining 2 properties is easier than maintaining 200 (repetitive) unit tests.

6.2.2 Haskell, GHC & Typed Holes

Haskell Haskell is a statically typed, non-strict, purely functional programming language. Its design ensures that the presence of side effects is always visible in the type of a function, and it is typical programming practice to cleanly separate code requiring side effects from the main application logic. This facilitates a modular approach to testing in which program parts can be tested in isolation without needing to consider global state or side effects.

Haskell's rich type system and type classes allow tools such as QuickCheck [245] to efficiently test functions using properties, where the inputs are generated by QuickCheck based on a generator for a given datatype.

Valid Hole-Fits Our tool is based on using the Glasgow Haskell Compiler (*GHC*), which is widely used in both industry and academia.

GHC has many features beyond the Haskell standard, including a feature known as *typed holes* [61].

A "hole", denoted by an underscore character (_), allows a programmer to write an incomplete program, where the hole is a placeholder for currently missing code.

Using a hole in an expression generates a type error containing contextual information about the placeholder, including, most importantly, its inferred type.

In addition to contextual information, GHC suggests some *valid hole-fits* [61].

Valid hole fits are a list of identifiers in scope which could be used to fill the holes without any type errors.

As a simple example, consider the interaction with the GHC REPL shown in Figure 6.2.

```
GHCi> let degreesToRadians :: Double -> Double
degreesToRadians d = d * _ / 180

   <interactive>:4:30: error:
   • Found hole: _ :: Double
   In the expression: d * _ / 180
   Valid hole fits include
   d :: Double (bound at <interactive>:4:22)
   pi :: forall a. Floating a => a (imported from ‘Prelude’)
```

Figure 6.2: Example code with a hole and its valid hole-fits

Here the definition of `degreesToRadians` contains a hole. There are just two valid hole-fits in scope: the parameter `d` and the predefined constant `pi`. GHC can not only generate simple candidates such as variables and functions, but also *refinement* hole-fits. A refinement hole-fit is a function identifier with placeholders for its parameters. In this way GHC can be used to synthesize more complex type-correct candidate expressions through a series of refinement steps up to a given user-specified *refinement depth*. For example, setting the refinement depth to 1 will additionally provide, among others, the following hole-fits:

```
        negate ( _ :: Double)
        fromInteger ( _ :: Integer)
```

In this work we use hole fitting for program repair by removing a potentially faulty sub-expression, leaving a hole in its place, and using valid hole-fits to suggest possible patches.

Hole-Fit Plugins By default, GHC considers every identifier in scope as a potential hole-fit candidate, and returns those that have a type corresponding to the hole as hole-fits. However, users might want to add or remove candidates or run additional search using a different method or external tools. For this purpose, GHC added hole-fit plugins [197], which allows users to customize the behavior of the hole-fit search. When using GHC as a library, this also allows users to extract an internal representation of the hole-fits directly from a plugin, without having to parse the error message.

6.2.3 GenProg, Genetic Program Repair & Patch Representation

Search-based program repair centered mostly around the work of Le Goues et al. [85] in GenProg, which provided genetic search for C-program repair. One of the primary contributions was the representation of a patch as a change (addition, removal, or replacement) of existing statements. Genetic search is based around the mutation, creation and combination of *chromosomes* — the basic building bricks of genetic search. A chromosome of APR is a list of such changes rather than a full program (AST), making the approach lightweight. Utilizing changes is based on the *Redundancy Assumption* [247], i.e., assuming that the required statements for the fix already exists. The code might just use the wrong variable or miss a null-check to function properly. This assumption has been verified by Martinez et al. [236], showing that the redundancy assumption widely holds for inspected repositories. We adopted the patch-representation in our tool, but were able to weaken the redundancy assumption (see Section 6.3).

Since GenProg, much has been done in genetic program repair [248] mostly for Java. Particularly Astor [62] enabled lots of research [249–252] due to its modular approach, as well as real-world applications [253, 254]. This modularity, mostly the separation of fault localization, patch-generation and search is a valuable lesson learned by the community that we adopted in our tool. Compared to this body of research, our scientific contributions lie within the patch-generation and the search-space (see Section 6.3.1).

6.2.4 Repair of Formally Verified Programs & Program Synthesis

Another field of research dominant in functional programming is formal verification, in which mathematical methods are used to prove the correctness of programs. Due to its strengths it has been widely applied to various tasks, such as hardware-verification [255], cryptographic protocols [256] or lately smart contracts [257]. But formal verification has also been applied to the domain of program repair and synthesis [258, 259], and some languages can arguably be considered synthesizers around constraints (e.g. Prolog).

Using specification-based synthesis in combination with a SAT solver can be effective, however the accuracy is closely tied to the completeness of the post-condition constraints [260].

For Haskell, these approaches revolve around *liquid types*, which enrich Haskell's type system with logical predicates that are passed on to an SMT solver during type checking [261–264].

The existing approaches [265–267] focus primarily on the search-aspects of program synthesis due to the (infinite) search space and often perform a guided search similar to proof-systems.

The approach used in the *Lifty* [268] language is especially relevant:

Lifty is a domain-specific data-centric language in which applications can be statically and automatically verified to handle data specified as per declarative security policies, and suggest provably correct repairs when a leak of sensitive data is detected.

Their approach differs in that they target a domain-specific language and focus on type-driven repair of security policies and not general properties. Another interesting approach is the TYGAR based Hoogle+ API discovery tool, where users can specify programming tasks using either a type, a set of input-output tests, or both, and get a list of programs composed from functions in popular Haskell libraries and examples of behavior [269]. It is however focused on API discovery and not program repair, although incorporating Hoogle+ into PropR is an interesting avenue for future work. The approach by Lee et al. [270] is in many ways similar. They also operate on student data and find very valuable insights from repair and identical challenges. The approach they developed (FixML) exploits typed holes to align buggy student programs with a given instructor-program based on symbolic execution. FixML is different as it requires a gold standard, and synthesizes by type-enumeration after symbolic execution. To some degree, this is similar to our implementation of an exhaustive search. Semantics-based repair using symbolic-execution like that of *Angelix* [271] can be very effective in fixing real-world bugs, and uses symbolic expressions similarly to our typed-holes. However, there are some scalability concerns for symbolic execution, and while they can be mitigated using a carefully chosen number of suspicious expression and their derived angelic forests [271], they can also be mitigated using genetic algorithms and the more lightweight property-based analysis, motivating their usage in PropR. Compared to program synthesis, program repair is better able to take advantage of a "reasonable" baseline program from the developers.

In terms of utilizing specifications, the primary benefit of QuickCheck is the easy adoption for users, whereas formal verification comes with a high barrier of entry for most programs and requires dedicated and educated developers. To some degree we utilize formal verification due to the type-correctness-constraint that already greatly shrinks the search space — while we assert the functional correctness with tests and properties. A full formal verification-suite might produce better results, but we ease the adoption of our approach by utilizing comprehensive properties and tests.

6.3 Technical Details — PropR

To investigate the effectiveness of combining property-based tests with type-based synthesis, we implemented PropR. PropR is an automated program repair tool written in Haskell, and uses GHC as a library in conjunction with custom-written hole-fit plugins as the basis for parsing source code, synthesizing fixes, as for instrumenting and running tests. PropR also parametrizes the tests so that local definitions can be exchanged with new ones, which allows us to observe the effectiveness of a fix. To automate the repair process, PropR implements the search methods described in Section 6.3.4 to find and combine fixes for the whole program repair. An overview of the PropR test-localize-synthesize-rebind (TLSR) loop is provided in Figure 6.3. The circled numbers ⓝ in this section refer to the labels in Figure 6.3.

As a running example, imagine we had an *incorrect* implementation of a function to compute the length of a list called `len`, with properties, as seen in Figure 6.4.

6.3.1 Compiler-Driven Mutation

To repair a program, we use GHC to parse and type-check the source into GHC's internal representation of the type-annotated Haskell AST. By using GHC as a library, we can interact with GHC's rich internal representation of programs without resorting to external dependencies or modeling. We determine the tests to fix by traversing the AST for top-level

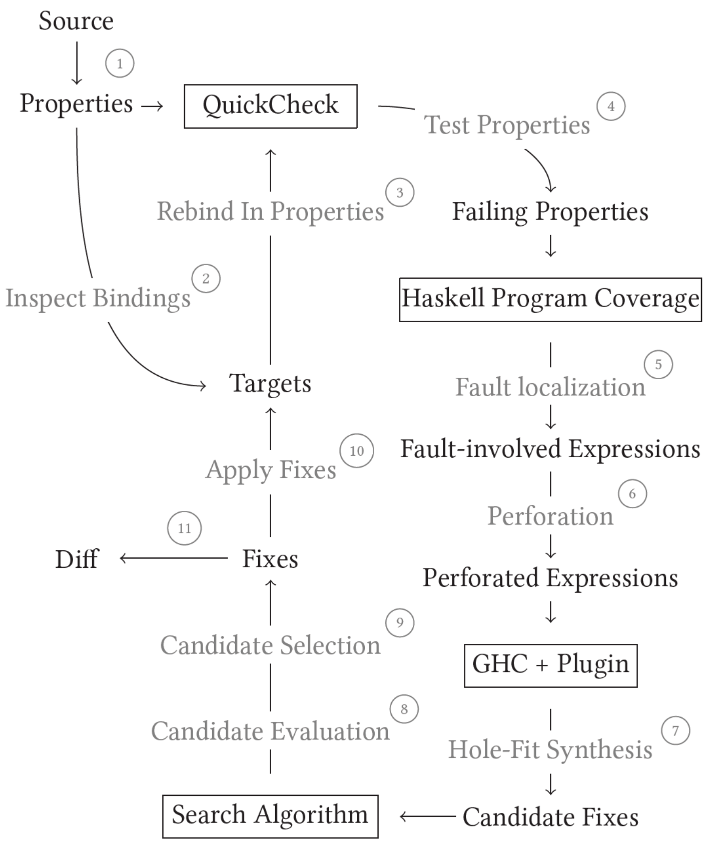


Figure 6.3: The PropR test-localize-synthesize-rebind loop

```
len :: [a] -> Int
len [] = 0
len xs = product $ map (const (1 :: Int)) xs

prop_abc :: Bool
prop_abc = len "abc" == 3

prop_dup :: [a] -> Bool
prop_dup x = len (x ++ x) == 2 * len x
```

Figure 6.4: An incorrect implementation of length. We `map` over the list and set all elements to `1 :: Int`, and take the `product` of the resulting list. This means that `len` will always return 1 for all lists. An expected fix would be to take the `sum` of the elements, which would give the length of the list.

bindings with either a type (`TestTree`) or name (`prop`) that indicates it is a test ①. We

```haskell
prop'_abc :: ([a] -> Int) -> Bool
prop'_abc f = f "abc" == 3

prop'_dup :: ([a] -> Int) -> [a] -> Bool
prop'_dup f x = f (x ++ x) == 2 * f x
```

Figure 6.5: The parametrized properties for len

```haskell
abc_prop :: Bool
abc_prop = prop'_abc length

dup_prop :: [a] -> Bool
dup_prop = prop'_dup length
```

Figure 6.6: The parametrized properties applied to a different implementation of `len`, the standard library `length`

use GHC's ability to derive data definitions for algebraic data types [197] and the Lens library [272] to generate efficient traversals of the Haskell AST. To determine the function bindings to mutate, we traverse the ASTs of the properties and find variables that refer to top-level bindings in the current module ②. We call these bindings the *targets*.

In our example, both `prop_abc` and `prop_dup` use the local top-level binding `len` in their body, so our target set will be {`len`}.

Parametrized properties To generalize over the definition of targets in the properties and tests, we create a *parametrized property* from each of the properties by changing their binding to take an additional argument for each of the *targets* in their body. This allows us to rebind (i.e., change the definition of) each of the targets by providing them as an argument to the parametrized property ③. Once the parametrized property has received all the target arguments, it now behaves like the original property, with the target bindings referring to our mutated definitions. We show the parametrized properties for the properties in Figure 6.4 in Figure 6.5.

The new properties in Figure 6.6, `abc_prop` and `double_prop` will now behave the same as the original `prop_abc` and `prop_dup`, but with every instance of `len` replaced with `length`:

```haskell
abc_prop = length "abc" == 3
double_prop x = length (x ++ x) == 2 * length x
```

This allows to create new definitions of `len` and evaluate how the properties behave with the different definitions.

Fault localization PropR uses an expression-level fault localization spectrum [230], to which we apply a binary fault localization method (touched or not touched by failing properties). A notable difference to other APR tools like Astor is that we can perform fault localization for the *mutated* targets. This enables PropR to adjust the search space once a partial repair has been found, i.e. one that passes a new subset of the properties. Since fault localization is expensive, by default we only perform it on the initial program similarly to Astor [62, 162]. GHC's *Haskell Program Coverage* (HPC) can instrument Haskell modules and get a count of how many times each expression is evaluated during execution [196]. Using QuickCheck, we find which properties are failing and generate a counterexample for each failing property ④. For properties without arguments (essentially unit tests), we do not need any additional arguments, so we can run the property as-is: the counterexample is the property itself. By applying each property to its counterexample and instrumenting the resulting program with HPC, we can see exactly which expressions in the module are evaluated in a failing execution of property ⑤. The expressions evaluated in the counterexample of the property are precisely the expressions for which a replacement would have an effect: non-evaluated expressions cannot contribute to the failing of a property. We call these the *fault-involved expressions*. These will be *all* the expressions involved in failing tests/properties, and covers every expression invoked when running counter-examples.

In our simple example, only `prop_dup` requires a counterexample, for which QuickCheck produces a simple, non-empty list, `[()]`. When we then evaluate `prop_abc` and `prop_dup [()]`, only the expressions in the non-empty branch of `len` are evaluated: the empty branch is not involved in the fault.

Perforation For the targets, we generate a version of the AST with a new typed hole in it, in a process we call *perforation*. When we perforate a target, we generate a copy of its AST for each fault-involved expression in the target, where the expression has been replaced with a typed hole ⑥. The perforated ASTs are then compiled with GHC. Since they now have a typed hole, the compilation will invoke GHC's valid hole-fit synthesis [61] ⑦. We present a few examples of the perforated versions of `len` in Figure 6.7.

```
len [] = 0
len xs = _

len [] = 0
len xs = _ $ map (const (1 :: Int)) xs

len [] = 0
len xs = product $ _

len [] = 0
len xs = product $ _ (const (1 :: Int)) xs
...
```

Figure 6.7: A few perforated versions of `len`. N.B. the empty branch is not perforated, as it is not involved in the fault

6.3.2 Fixes

A fix is represented as a map (lookup table) from *source locations* in the module to an expression representing a fix candidate. Merging two fixes is done by simply merging the two maps. Candidate fixes in PropR come in three variations, *hole-fit candidates*, *expression candidates*, and *application candidates*.

Hole-fit Candidates Using a custom hole-fit plugin, we extract the list of valid hole-fits for that hole, and now have a well-typed replacement for each expression in the target AST.

```
Found hole: _ :: [Int] -> Int
In an equation for 'len':
len xs = _ $ map (const (1 :: Int)) xs
Valid hole fits include
head :: [a] -> a
last :: [a] -> a
length :: Foldable t => t a -> Int
maximum :: (Foldable t, Ord a) => t a -> a
minimum :: (Foldable t, Ord a) => t a -> a
product :: (Foldable t, Num a) => t a -> a
sum :: (Foldable t, Num a) => t a -> a
Valid refinement hole fits include
foldl1 (_ :: Int -> Int -> Int)
...
```

Figure 6.8: Hole-fits for a perforation of `len`, where `product` has been replaced with a hole

```
{<interactive:3:10-15>: head}
{<interactive:3:10-15>: last}
{<interactive:3:10-15>: length}
...
{<interactive:3:10-15>: sum}
```

Figure 6.9: Candidate fixes derived from the valid hole-fits in Figure 6.8. The location refers to `product` in `len`

We derive hole-fit candidates directly from GHC's valid hole-fits, as seen in Figure 6.8, giving rise to the fixes in Figure 6.9. These take the form of an identifier (e.g., `sum`), or an identifier with additional holes (e.g., `foldl1 _`) for refinement fits.

Since we synthesize only well-typed programs, we cannot use refinement hole-fits directly: the resulting program would produce a typed hole error.

To use refinement hole-fits, we recursively synthesize fits for the holes in the refinement hole-fits up to a depth configurable by the user. This means that we can generate e.g., `foldl1 (+)` when the depth is set to 1, and e.g., `foldl1 (flip _)` → `foldl1` `(flip (-))` for a depth of 2, etc. By tuning the refinement level and depth, we could synthesize most Haskell programs (excepting constants). However, in practical terms, the amount of work grows exponentially with increasing depth.

To be able to find fixes that include constants (e.g., `String` or `Int`) or fixes that would otherwise require a high and deep refinement level, we search the program under repair for *expression candidates* [273]. These are injected into our custom hole-fit plugin and checked whether they fit a given hole using machinery similar to GHC's valid hole-fit synthesis, but matching the type of an expression instead of an identifier in scope. In our example, these would include `0`, `(1 :: Int)`, `(x ++ x)`, and more. For each expression candidate, we then check that all the variables referred to in the expressions are in scope, and that the expression has an appropriate type. We also look at *application candidates* of the form `(_ x)`, where `x` is some expression already in the program, and _ is filled in by GHC's valid hole-fit synthesis. This allows us to find common data transformation fixes, such as `filter (not . null)`.

Regardless of technical limitations, this approach can be considered a form of *localized program synthesis* exploited for program repair. By using valid hole-fits, we can utilize the full power GHC's type-checker when finding candidates and avoid having to model GHC's ever-growing list of language extensions. This allows us to drastically reduce the search space to well-typed programs only.

6.3.3 Checking Fixes

Once we have found a candidate fix, we need to check whether they work. We apply a fix to the program by traversing the AST, and substituting the expression found in the map with its replacement.

```
len1 [] = 0
len1 xs = head $ map (const (1 :: Int)) xs
...
len3 [] = 0
len3 xs = length $ map (const (1 :: Int)) xs
...
len7 [] = 0
len7 xs = sum $ map (const (1 :: Int)) xs
```

Figure 6.10: New targets defined by applying the fixes in Figure 6.9 to the original `len`

We do this for all targets, and obtain new targets where the locations of the holes have been replaced with fix candidates.

For the given `len` example, the fixes in Figure 6.9 give rise to the definitions shown in Figure 6.10.

We then construct a checking program that applies the parametrized properties and tests to these new target definitions and compile the result.

A simplified example of this can be seen in Figure 6.11, though we do additional work to extract the results in PropR. It might be the case that the resulting program does not compile: as our synthesis is based on the types, we might generate programs that do not

```
PropR> mapM sequence
[[quickCheck (prop'_abc len1), quickCheck (prop'_dup len1)]
,[quickCheck (prop'_abc len2), quickCheck (prop'_dup len2)]
,[quickCheck (prop'_abc len3), quickCheck (prop'_dup len3)]
...
,[quickCheck (prop'_abc len7), quickCheck (prop'_dup len7)]]
-- Evaluates to:
[[False, False],[False, False],[True, True],[False, False]
,[False, False],[False, False],[True, True]]
```

Figure 6.11: Checking our new targets from Figure 6.10

parse because of a difference in precedence (precedence is checked during renaming, *after* type-checking in GHC). We remove all those candidate fixes that do not compile, obtaining an executable that takes as an argument the property to run, and returns whether that property failed.

We run this executable in a separate process: running it in the same process might cause our own program to hang due to a loop in the check.

By running in a separate process, we can kill it after a timeout and decide that the given fix resulted in an infinite loop. After executing the program, we have three possible results: all properties succeeded; the program did not finish due to an error or timeout; or some properties failed ⑧.

In our example, we see in Figure 6.11 that `len3` and `len7` pass all the properties, meaning that replacing `product` with `length` or `sum` qualifies as a repair for the program.

6.3.4 Search

Within PropR, we implemented three different search algorithms: *random search*, *exhaustive search*, and *genetic search* ⑨.

All three algorithms share a common configuration: they all have a time budget (measured in wall clock time) after which they exit, and return the results (if any) that they've found.

For the **genetic search**, PropR implements best practices and algorithms common to other tools such as Astor [62] or EvoSuite [244]. A mutation consists of either dropping a replacement of a fix, or adding a new replacement to it. The initial population is created as picking n random mutations. The crossover randomly picks cut points within the parent chromosomes, and produces offspring by swapping the parents' genes around the cut points. We support environment-selection [167] with an elitism-rate [274] for truncation. *Elitism* means that we pick the top $x\%$ percent of the fittest candidates for the next generation, filling the remaining $(100-x)\%$ with (other) random individuals from the population. We choose random pairs from the last population as parents and perform environment selection on the parents and their offspring. Our manual sampling of repairs-in-progress on the data points showed that genetic search requires high *churn* in order to be effective: changing a single expression of the program usually failed more properties than it fixed. Hence, the resulting configurations for the experiment have a low elitism- and high mutation- and crossover-rate.

Within **random search**, we pick (up to a configurable size) evaluated holes at random and pick valid hole-fits at random with which to fill them. We then check the resulting fix and cache it. The primary reason for using random search is to show that the genetic search is an improvement over *guessing*. Nevertheless, Qi et al. [172] showed that random search sometimes can be superior to genetic search, further motivating its application. Besides, random search is a standard baseline in search-based software engineering to assess whether more "intelligent" search algorithms are needed for the problem under analysis.

For **exhaustive search**, we check each hole-fit in a breadth-first manner: first all single replacement fixes, then all two replacement fixes and so on until the search budget is exhausted. Exhaustive search is deterministic apart from inherent randomness in QuickCheck. We use exhaustive search to demonstrate the complexity of the problem, and to show that search is better than enumeration. The deterministic search pattern of exhaustive search would be ideal for a single fix problem such as our example.

The fitness for all searches is calculated as the failure ratio $\frac{\text{number of failures}}{\text{number of tests}}$, with a non-termination or errors treated as the worst fitness 1 and a fitness of 0 (all tests passing) marks a candidate patch. Such patches are removed from populations in genetic search and replaced by a new random element.

Within the test-localize-synthesize-rebind loop (Figure 6.3) we perform one generation of genetic search per loop, and after the selection of chromosomes the program is re-bound and coverage re-evaluated. The authors observed that this is a bit over-engineered for small programs — the fault localization did not greatly change when the programs had only a single failing property. As an optimization, we added a flag to skip the steps ⑤ to ⑦ in the loop to speed up the actual search. This configuration was enabled during experiments presented in Section 6.4. The exhaustive and random search do not perform any rebinding.

6.3.5 Looping and Finalizing Results

Looping If there are still failing properties after an iteration of the loop, we apply the current fixes we have found so far to the targets and enter the next iteration of the loop ⑩, repeating the process with the new targets until all properties have been fixed, or the search budget runs out.

Finalizing and Reporting Results After we have found a set of valid fixes that pass all the properties, we generate a diff for the original program based on the program bindings and the mutated targets constituting the fix ⑪. This way the resulting patches can be fed into other systems such as editors or pull requests.

```
diff --git a/<interactive> b/<interactive>
--- a/<interactive>
+++ b/<interactive>
@@ -1,2 +1,2 @@ len [] = 0
len [] = 0
-len xs = product $ map (const (1 :: Int)) xs
+len xs = length $ map (const (1 :: Int)) xs

diff --git a/<interactive> b/<interactive>
--- a/<interactive>
+++ b/<interactive>
@@ -4,2 +4,2 @@ len [] = 0
len [] = 0
-len xs = product $ map (const (1 :: Int)) xs
+len xs = sum $ map (const (1 :: Int)) xs
```

Figure 6.12: The final result of our repair for `len`

6.4 Empirical Study

6.4.1 Research Questions

Given the concepts presented in Section 6.3, research interests are twofold: How well does the typed hole synthesis perform for APR, and what is the individual contribution of properties. As within the integral approach of PropR, the effects cannot truly be dissected: The only contributions that we can separate for distinct inspection is the use of properties, under which we will investigate the patches generated by PropR.

We first want to answer whether properties add value for guiding the search. Ideally, properties should improve the repair-rate, speed and quality regardless of the approach, which we address in RQ1:

Research Question 1
To what extent does automatic program repair benefit from the use of properties?

Given that properties do have an impact (for better or worse), we want to quantify its extent on configuration and selection of search algorithms. For example, we expect that the use of properties helps with fitness and search, but will increase the time required for evaluation — this would motivate to configure the genetic search to have small but well guided populations. To elaborate this we define RQ2 as follows:

Research Question 2
How can we improve (and configure) search algorithms when used with properties?

With the last research question we want to perform a qualitative analysis on the results found. Previous research showed that *just maximizing metrics* is not sufficient. With a manual analysis we look for the issue of overfitting and try to investigate new issues and new patterns of overfitting.

Research Question 3

To what extent is overfitting in automatic program repair addressed by the use of properties?

6.4.2 Dataset

The novel dataset stems from a student course on functional programming. Within the exercise, the students had to implement a calculator that parses a term from text, calculates results and derivations. While the overall notion is that of a classroom exercise, the problem nevertheless contains real-world tasks asserted by real-world tests. The calculator itself is a classic student-exercise, but the subtask of parsing is both common and difficult, representing a valuable case for APR. In total, we collected **30 programs** that all fail at least one of **23 properties** and one of **20 unit tests**. The programs range from 150 to 700 lines of code (excluding tests) and have at least 5 top level definitions. These are *common* file-sizes for Haskell, e.g. PropR itself has an average of 200 LoC per file. The faults are localized to one of the three modules provided to PropR.

The most violated tests are either related to parsing and printing (especially of trigonometric functions, also seen in Figure 6.18) or about simplification (seen in Figure 6.13), which are core-parts of the assignment. The calculator makes a particularly good example for properties, as attributes such as commutativity, associativity etc. are easy to assert but harder to implement. Hence, we argue that the calculator-exercise makes a case for typical programs that implement properties (i.e., they are not *artificially* added for APR).

Data points were selected from the students submissions if they fulfilled the following attributes: Ⓐ it compiled Ⓑ it failed the unit test suite **and** the property-based test suite separately. An *error-producing test* is considered as a normal failure. We selected them by these criteria to draw per-data-point comparisons of properties to unit tests and their unison. We consider a separate investigation of repairing unit test failing programs versus properties failing programs and their overfitting future research.

```haskell
prop_simplify_idempotency :: Expr -> Bool
prop_simplify_idempotency e =
  simplify (simplify e) == simplify e
```

Figure 6.13: A property asserting the idempotency of simplify

The anonymized data is provided in the reproduction package.

6.4.3 Methodology / Experiment Design

To evaluate RQ1 and RQ2 we perform a grid experiment on the dataset with the parameters presented in Table 6.1. For every of the 45 configurations we make a repair attempt on every point in the dataset. The genetic search uses a single set of parameters that was determined through probing. We utilize docker and limit every container to 8 vCPUs @ 3.6ghz and 16gb RAM (the container's lifetime is exactly one data-point). Further information on the data collection can be found in the reproduction package.

Table 6.1: Parameters for Grid Experiment

parameter	inspected values
tests	Unit Tests ; Properties ; Unit Tests + Properties
search	random ; exhaustive ; genetic
termination	10 minute search-budget
seeds	5 seeds

Given this grid experiment, we collect the following values for each data point in the dataset:

1. Time to first result
2. Number of distinct results within 10 minutes
3. The fixes themselves

The search budget starts after a brief initialization, as PropR loads and instruments the program. We round the measured times to two digits as recommended by Neumann et al. and remove Type-1-Clones (identical up to whitespace) from the results [275, 276].

To answer RQ1 we check every trial whether at least one patch was found (whether it was *solved*). We then perform a Fisher exact test [277] to see if the entries originate from the same population, i.e., if they follow the same distribution. We consider results with a p-value of smaller than 0.05 as significant.

To answer RQ2 we perform a pairwise Wilcoxon-RankSum test [278] on the data points grouped by their test configuration. The Wilcoxon test is a non-parametric test and does not make any assumption on data distribution. In its pairwise application, we first compare the effect of unit tests against the effect of properties, then unit tests against combined unit tests and properties etc. We choose a significance level of 95%.

After we have seen whether properties have a significant impact on program repair, we can quantify the effect size by applying the Vargha-Delaney test [177] to the given pairs of configurations. In the Vargha-Delaney test, a value of e.g. 0.7 means that algorithm B is better than algorithm A in 70% of the cases, estimating a similar probability of dominance for future applications on similarly distributed data points. Note that a result of 0.5 does not mean there was no effect — the groups can still be significantly different without being clearly *better*.

RQ3 can (to the best of our knowledge) only be answered by human evaluation. Existing research on automatic patch-validation by Qi [279] requires an automatic test-generation framework (which is not available for Haskell) as well as a gold-standard fix to work as an oracle. They used existing git-fixes as oracles, but we expect some data points to be correct despite not matching the sample-solution. Similarly, work by Nilizadeh et al. [280] utilizes formal verification to automatically verify generated patches, but unfortunately, no specifications were available for our dataset. Instead, we perform the analysis manually, similar to [242] and [89]. As there are too many results to manually inspect, we sampled

70 fixes[1] and let two authors label them as *overfit* or *not overfit.* The authors do so based on their domain-knowledge and in accordance with a given gold-standard. On disagreement, the authors provide a short written statement before discussing and agreeing on the fix-status. The conclusion of the discussion is also documented with a short statement. The manual labels as well as the statements are shared within the replication package.

6.5 Results

The following section answers the research questions in order and presents general information gained in the study.

RQ 1 — Repair Rate In total, PropR managed to find **patches for 13 of 30 programs** of the dataset. In Table 6.2 we show the detailed results of these 13 programs. We found **228 patches** in total, with **a median of 3 patches per successful run**. A visualization of the results can be seen in Figure 6.14 and Figure 6.15.

For every entry, we performed a Fisher exact test based on the repair per seed of every test suite. The contingency tables are based on whether the specific seed found patches for the test suite. It showed that 4 of the 13 repaired entries were significantly better in producing repairs with properties (E1, E3, E4, and E14 from Table 6.2).

A *global* Fisher exact test and Wilcoxon-RankSum test showed no statistical significant difference between the test suites (p-values of 10%-20%). Whether properties are beneficial is a highly specific topic, and we expect it more to be a matter whether the bug is properly covered by the test suite. We argue that properties can produce stronger test suites than unit tests, but whether they are applicable and well implemented is ultimately up to the developers.

Figure 6.14 shows genetic search outperforming exhaustive search in any test suite configuration, and most effectively for properties.

Figure 6.15 shows the overlap of *solved* entries by test suite. It shows that four entries were uniquely solvable by using only properties and one entry was uniquely solvable by the combined test suite. All entries solved by unit tests have also been solved by the properties. This does not necessarily imply that properties are *better* — the patches can still be overfit and are to be evaluated in RQ3.

Summary RQ1

Properties do not significantly help with producing patches. In our study, properties found unique patches that unit tests did not produce. The difference between results in genetic and exhaustive search were greatest for the properties.

RQ 2 — Repair Speed We grouped the results per seed and compared the median time-to-first-result for each test suite. All two-way hypothesis-tests reported a significant p-value of less than 0.01, proving that there are significant differences in distributions.

[1]The threshold of 70 has been calculated after seeing 230 patches being generated, which is sufficient sample for a p-value of 0.05 at an error rate of 10%

Table 6.2: Number of independent runs that produced at least one patch for genetic search

Programs	E01	E02	E03	E04	E05	E07	E08	E09	E12	E13	E14	E18	E25
Units	0	1	5	5	5	5	5	5	5	0	0	0	5
Props	5	1	1	0	5	5	5	5	2	1	5	2	3
Both	0	1	4	0	1	5	5	5	3	0	0	0	3

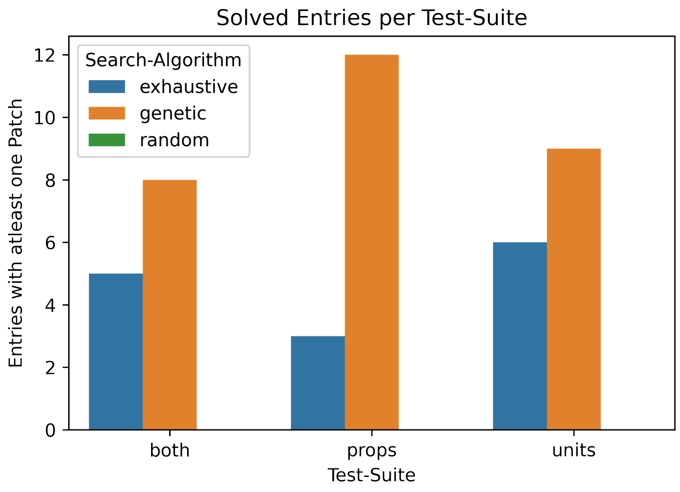


Figure 6.14: Solved Entries per Test-Suite and Algorithm

In particular, we performed a test[2] whether properties are faster than unit tests in finding patches, which was the case with a p-value of 0.02. The Vargha and Delaney effect size test showed an estimate of 0.28 which is considered a medium-effect size, showing that properties are faster than unit tests.

An overview of the time-to-first-result can be seen in Figure 6.16. We would like to stress that similar to some results of RQ3, the test suites' speed seems to behave in such a way that the slowest and hardest test determines the magnitude of search. Properties do not have a significant *overhead* by design, which is positively surprising. The cost of their execution is compensated by the speedup in search.

Summary RQ2

Genetic Search finds patches faster for properties than for unit tests. The combined test suite also yields combined search speed.

RQ 3 — Manual Inspection From the sample of 70 patches the authors agreed on 49 to be overfit and 21 to be fit. Given the overall population of 230 and an error rate of 10%, we expect 62 to 76 of total patches to be correct. This results in a total *non-overfit* rate of

[2]Wilcoxon-RankSum with *less*

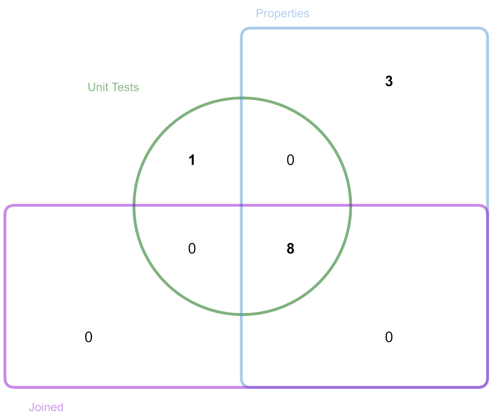


Figure 6.15: Venn-Diagram of Solved Entries per Suite

27% to 33%. In particular, patches in the sample found for unit tests were overfit in 85% of cases (19/23), but the properties were overfit in 64% of cases (21/33). The combined test suite overfit in 63% (9/14) cases.

These are not evenly distributed — some programs are only repaired overfit while others are always well fixed. Hence, we deduct that of the 13 Entries that have fixes, 3 to 4 have non-overfit repairs. This estimates an effective repair-rate of 10% or respectively 13%, which performs similar to the rates reported by Astor [89] (13%) and better than GenProg [89](1-4%). Arja [281] reports an effective repair rate of 8% which we slightly outperform.

A typical example found by manual inspection was adding space-stripping to the *addition*-case of `showExpr`, as seen in Figure 6.17.

There is a single unit test (see Figure 6.18) to assert a printed addition without spaces. Within the patch only the "+" case gets *repaired* — this is due to the precedence of the expression which is correctly picked up. Hitherto, the change in the addition actually removes all white-space and correctly passes the test. This (actually) solves the unit test as expected and is therefore arguably not truly overfitting. Nevertheless, a developer would perform the string-stripping on all cases, not only on the addition. Here we see a shortcoming of the test suite — this would have not been possible if we had a property `prop_showExpr_printNoSpaces` or if we simply had unit tests for all cases. In other data points, where the `showExpr` had a unified top-level expression (not an immediate pattern match), the repair was successful by adding top-level string-stripping. We would also like to stress the quality of the patch generated despite overfitting: It draws 4 elements (`filter`, `toLower`, `isSpace`, `(.)`) which were not in the code beforehand and applied them at the correct position.

Another issue observed were empty patches — these appeared when the QuickCheck properties exhibited inconsistent behavior. We suspect a property that tests for the idempotency of `simplify` seen in Figure 6.13, which requires a randomly generated expression. The property is meant to assert that e.g., `x * 4 * 0` gets reduced to `0` and not to `x * 0`. Whether this case (or similar ones) are tested depends on the randomly created expressions — which makes it an inconsistent test. These are issues with the test suite that were uncov-

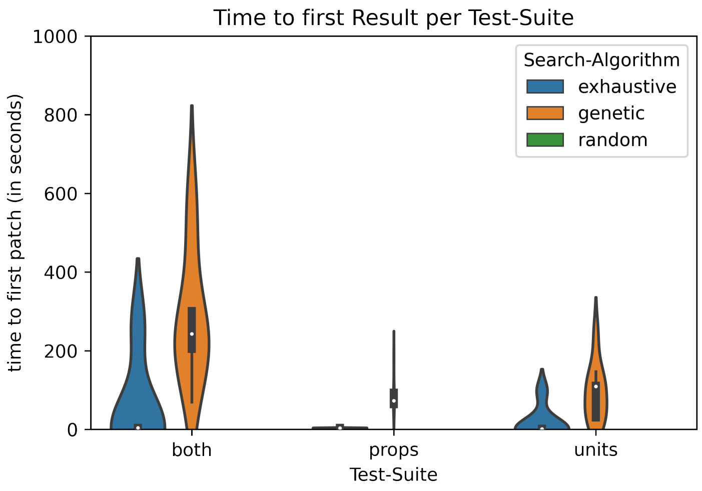


Figure 6.16: Distribution of Time to First Patch per Entry

```
diff --git a//input/expr_units.hs b//input/expr_units.hs
--- a//input/expr_units.hs
+++ b//input/expr_units.hs
@@ -59,6 +59,6 @@ showExpr (Num n) = show n
showExpr (Num n) = show n
-showExpr (Add a b) = showExpr a ++ " + " ++ showExpr b
+showExpr (Add a b) =
+ showExpr a ++ ((filter (not . isSpace)) (" + ")) ++ showExpr b
showExpr (Mul a b) = showFactor a ++ " * " ++ showFactor b
showExpr (Sin a) = "sin" ++ showFactor a
showExpr (Cos a) = "cos" ++ showFactor a
showExpr (Var c) = [c]
```

Figure 6.17: A PropR patch showing overfitting on a unit test

```
prop_unit_showBigExpr :: Bool
prop_unit_showBigExpr = strip (showExpr expr) == strip res
  where
    res = "sin (2.1 * x + 3.2) + 3.5 * x + 5.7"
    strip = filter (not . isSpace)
    arg = Expr.sin (add (mul (num 2.1) x) (num 3.2))
    expr = add (add (add  (mul (num 3.5) x)) (num 5.7)) arg
```

Figure 6.18: The unit test corresponding to the fix in Figure 6.17

ered due to the hyper-frequent evaluation. The only way to mitigate this is to provide a handful of unit tests or write a specific expression-generator used for the flaky property. We labeled empty patches to be overfit as we do not consider them proper repairs.

Summary RQ3

Adding properties reduced the overfit ratio from 85% to 63%, doubling the number of *good* patches. The resulting effective repair rate of 10% to 13% is comparable to other tools. Overfitting appeared despite the use of properties, but generally less due to an overall stronger test suite.

6.6 Discussion

Overfitting on Properties Similar to the overfitting of empty patches shown in RQ3, we had cases of patches where one or more failing properties exhibited inconsistent behavior, and an overfit patch was considered a successful patch. We observed an example that changed the simplification of multiplication to return 0 whenever a variable was in the term. This satisfies the `prop_MultWith0_Always0` property and should fail other properties such as multiplicative associativity, but (in rare cases) Quick-Check produced examples for the other properties that also evaluate to 0.

This overfitting shows that a test suite is not *better* just because it is utilizing properties. APR-fitness is still only as good as the test suite — properties help define better test suites and well-written properties positively influence APR.

Exploitable Overfitting A noticeable side effect of the tool is that if the repair overfits, it produces numerous (bad) patches, as can be seen from the number of generated proposals.

However, the repairs' output is not useless despite the overfitting: the suggested patches clearly show the shortcomings of the test suite. The proposed overfit patches help developers with fault localization and improving the test suite. In particular, as properties and unit tests are not exclusive, developers can consider a test-and-repair-driven approach, where they adjust the test suite and program iteratively assisted by the repair tool. We consider this approach attractive for class-room settings, where the programs are of lower complexity and allow for fast feedback. While we don't expect PropR to be enough to solve the tasks *for* the students, it clearly shows where the problems in the tests or code are. Exploring class-room usage is an interesting direction for future work.

Drastically Increased Search-Space Due to the novel approach to finding repair candidates, the search space drastically increased as compared to using existing expressions or statements only. This can be seen with the absence of random-search findings. Other studies showed at least some results with random search, sometimes reporting random search as most successful [172]. As we find (many) patches with exhaustive search, the problems are generally solvable with small changes. This implies that the only reason for random search to yield no results is the increased search space.

This finding motivates further investigating the genetic search and its optimization for more complex problems that do not achieve timely results with exhaustive search. We consider it worthwhile to revisit existing datasets, that were not solvable due to the redundancy assumption in most repair tools, using a typed hole approach.

Transference to Java As Java is the most prominent language for APR, it begs the question of which results can be transferred from Haskell into more mainstream approaches.

Properties are supported by JUnit-Plugins[3] and can easily be added to any common test suite and build-tool. The positive effects of properties as presented in Section 6.5 only require Java programs with sufficient properties. Unfortunately most Java-projects are not utilizing properties. Even less complex JUnit-Features, such as parameterized tests, are not widely adopted. This is in stark contrast to functional programming communities, where tools like QuickCheck are popular.

The hole-fitting repair approach cannot be easily reproduced for Java: The `JavaC`, unlike GHC, is not intended to be used as a library. Nevertheless, Java is strictly typed and the basic hole-fitting-approach can be integrated using meta-programming libraries like Spoon [282]. Many challenges remain: As Java's methods are not pure functions, they cannot be *just transplanted*. Side effects can wreak havoc and just on a technical level polymorphism, that is often only resolvable dynamically, bares huge follow-up-challenges.

But not all is lost for the JVM: Repair approaches that focus on the bytecode [283, 284], can easier adapt hole-fitting. In particular, one could imagine a tool that produces holes for bytecode and introduces the hole-fits utilizing more strict JVM Compilers such as Closure or Scala. We consider this extension a hard but valuable track for further research.

Future Work The primary research challenge we see is to combine existing approaches with the newly introduced PropR hole-fitting. A hybrid approach that could produce high churn with techniques from Astor [62] or ARJA [281] in combination with the fine-grained changes produced by PropR could solve a broader range of issues. Specific to Haskell is the need to introduce left-hand side definitions, i.e. new pattern matches or functions. These could be provided by generative neural networks [2, 285] and either be used as mutations or as an initial population of chromosomes. Representing multiple types of changes is only a matter of representation within the chromosome — the remaining search, fitness and fault localization can be kept as is.

For fault localization, we currently use *all* the expressions involved in the counter-examples. However, it should be possible to use the coverage information and the passing and failing tests for spectrum-based fault localization to narrow the fault-involved expressions further to suspicious expressions, rather than all the expressions involved in the failing test.

In terms of further evaluation, the next steps are user surveys and experiments on real world applications such as Pandoc[4] or Alex[5]. In particular, we envision a bot similar to Sorald [241] that provides patch-suggestions on failing pull-requests. We would like to ask maintainers and the public community to give feedback on the quality of repairs, and whether the suggested patches contributed to fault localization or improvements of the test suite even if not added to the code.

[3]`https://github.com/pholser/junit-quickcheck`
[4]`https://pandoc.org/`
[5]`https://www.haskell.org/alex/`

6.7 Threats to Validity

Internal Threats We addressed the randomness in our experiments by running 5 runs with different seeds according to the suggestions of Arcuri and Fraser [286]. The tool used in our experiment could contain bugs. We've published it under a FOSS-license to gain further insights and suggestions from the community. The experiment and dataset may contain mistakes, which we address by providing a reproduction package and open source the experiment and data. The package also contains notes on the data-preparation for the experiment.

External Threats The dataset is based on student data, which could be considered *artificial*. We stress that student data has been used in literature for program repair previously [248, 287–289]. A real-world study on program such as Pandoc [290] is part of future work. Pandoc, a popular Haskell document-converter, is rich in properties that test e.g., for symmetry over conversions.

6.8 Conclusion

The goal of this paper is to introduce a new automatic program repair approach based on types and compiler suggestions, in addition to utilizing properties for repair fitness and fault localization. To that end, we implemented PropR, a Haskell tool that utilizes GHC for patch-generation and can evaluate properties as well as unit tests. We provided a dataset with 30 programs and their unit tests and properties. On this dataset we performed an empirical study to compare the repair rates for different test suites and search-algorithms, and manually inspect the generated patches.

Our analysis of 230 patches show that we reach an effective repair rate of 10%-13% (comparable to other state-of-the-art tools) but have a reduced rate of overfitting (from 85% to 63% when applying properties). The novel approach for patch generation produces a greatly increased search space and promising patches on manual inspection. We observed that properties did not increase the number of programs for which patches were found, but solutions were less overfit and found faster. Overfitting based on unit tests persisted into the combined test suite. Similarly, we have observed that properties can produce cases of overfitting too.

Our results attest to the stronger utilization of language-features for patch generation to overcome the redundancy assumption, i.e., only reusing existing code. Using the compiler's information on types and scopes, the created patches are semantically correct and come in a much greater variety, which was reported as a missing feature for many APR tools. Our manual analysis motivates to use the generated patches (if not directly applicable) as guidance for fault localization or to improve the test suite.

6.9 Online Resources

PropR is available on GitHub under MIT-license at `https://github.com/Tritlo/PropR`. The reproduction package which includes the data, evaluation and a binary of PropR is available on Zenodo `https://doi.org/10.5281/zenodo.5389051`

7
Functional Spectra - Exploring Spectrum-Based Fault Localization in Functional Programming

Summary

Fault localization plays an important role in debugging, one technique thereof is *spectrum-based fault localization*, which uses tests and program coverage to produce a *spectrum* of locations involved in passing and failing tests. Despite its extensive application in Java, this technique remains underexplored within functional programming languages. This gap underscores a critical challenge: adapting spectrum-based fault localization to accommodate the unique characteristics of functional paradigms. Addressing this challenge, we evolve current spectrum-based approaches by extending the spectra with types and AST structure. We introduce a *rule-based system* tailored to capture more complex attributes of the spectrum. Spectrums are generated using an ingredient for the Tasty test framework, which allows easy adoption and reproducibility. Through an empirical study involving 11 real-world programs, we meticulously investigate the generated spectra along with the effectiveness of the rule-based system and their correlation to faults. Furthermore, we employ a set of classifiers to evaluate the potential for cross-program extrapolation of our findings. For most bugs, conventional spectrum-based formulas perform promisingly well in a functional context and are only outperformed by classifiers incorporating formulas.

This chapter presents a submission to *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium* (HASKELL 2024) under the working title *Functional Spectra - Exploring Spectrum-Based Fault Localization in Functional Programming* by Leonhard Applis, Matthías Páll Gissurarson and Annibale Panichella. It is currently being revisited and submitted to *18th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2025.* We elaborate in the preface.

Preface - Functional Programs and Spectra

This chapter is a point in time of ongoing work. The original motivation stems from PropR, where only a simple heuristic is used in place of a full-fledged fault-localization method. With PropR, we were confident that sourcing the repair from the compiler was a very fruitful technique, so improving the primitive parts in its ensemble was prioritized. We expected that employing a fine-grained approach, like spectrum-based fault localization would increase our chances of *repairing the right place*.

To bring this to life *step-by-step*, we sought out to first investigate fault localization as a isolated topic. After all, Haskell programs are different from Java, and we might have a bite too big to swallow if we bring it straight into automated program repair. To pay respect to the complexity, get the tooling incrementally in place and hopefully understand the application-domain better, we started working on a novel SBFL tool for Haskell.

One issue arose by related work from Li et al. [291], who independently worked on the topic in parallel, and even happened to have an overlap in datapoints. This forced us to extend the research scope beyond *"Applying SBFL for Haskell"* and introduce additional novel elements which form the majority of the chapter. When considering novelties, we reflected on a previous theme that stretches also over this thesis: That the compiler gives us most of the help we need. As such, we set our mind on extending the spectra by introducing compiler information, such as types and AST attributes. From the drawing board, these attributes lend themselves to arguments about faults in a schizophrenic manner: A complex type can do both, be the origin of bugs, or *protect* the developer from mistakes. Without evidence it is not clear to say if complicated types correlate with bugs.

To gather the required evidence, we opted for a *Rule*-approach: Instead of directly sorting compiler information in *good* or *bad* categories, by e.g. introducing them into a new SBFL-formula, we wanted to capture it quantifiable per expression and evaluate its merits. We implemented rules around execution-frequency, AST-attributes and types as presented later in this chapter. The goal behind this approach was two-fold: One, we hoped to find correlations between single-attributes (for example, type complexity) and faults. This itself could lead to an interesting insight on Haskell Bugs, and may open up research and debates. Second, the *good* rules can be directly combined into a expert-rule system. Dominant in reviews was a lack of clarity in results - reviewers expressed that there is no clear answer of what to do with the rules, and that the classifiers failed to do so as well. In general, there was too little distinction between rules, attributes, formulas and classifiers.

The current ongoing work on SBFL for Haskell tries to emphasize some of the learnings by introducing rule-based spectrum-filters. We still believe that the programs attributes correlate with faults, yet our first attempt did not (significantly) out-perform existing formulas while introducing high complexity. The formulas often yielded decent results, and seemed to mostly struggle with the granularity of expression level spectra. A lot of unrelated elements, or entries with an identical spectrum-entry, clock up the first x results and impact the TopX negatively. The novel found *uncovered* faults, faults that are not covered by the programs failing tests, must be solved with something beyond the spectrum-filters. Machine learning, or additional information by e.g. test generation or test carving, might be possible solutions.

7.1 Introduction

Functional programming achieved its good reputation by leading research on types and programming languages, but we believe it can also be championed in tooling and software engineering practices. It remains open *what* tooling functional programmers really want, but tooling they need. Haskell is known for adapting more niche tools than elsewhere or writing their own solutions.

According to modern developer surveys, approximately 50% of development is spent debugging, half of which is spent fixing bugs [292]. An important part of the debugging process is *fault localization*, i.e. determining *which* part of the program is at fault. For some developers, it can be enough to highlight code touched by failing tests and shade more frequently failing code darker. For maintainers, a solution in the continuous integration (CI) pipeline could be appropriate, showing the most suspicious code on failing pull requests (PRs). Lastly, for projects that heavily utilize code generation, linking identifiers and types of failing and passing code might help find faulty patterns on a broader, more abstract scale. These challenges cross most software paradigms, including functional programming [293, 294].

One way to assist the process is to introduce automated tools [295, 296], for example, spectrum-based fault localization (SBFL). A programs spectrum is created by running individual tests and collecting program coverage [63]; thus capturing different aspects of the program by branching over the test suite. Spectra have been successfully applied in imperative languages but have yet to be established in functional communities. Spectrum-Based Fault Localization is based on the premise that by comparing elements involved in failing tests and those involved in passing tests, we can deduce which location is at fault. The program spectrum captures the data required to determine which element is touched by which test through code instrumentation, running each test, and marking involved locations.

7.1.1 Example

Consider the function and properties (from QuickCheck [245]) in figure 7.1.

```
1 foldInt :: (Int -> Int -> Int) -> Int -> [Int] -> Int
2 foldInt _ z [] = 0
3 foldInt f z (x:xs) = (foldInt f z xs) `f` x
4
5 prop_sum, prop_prod, prop_diff :: [Int] -> Bool
6 prop_sum  xs = foldInt (+) xs 0 == sum xs
7 prop_prod xs = foldInt (*) xs 1 == product xs
8 prop_diff xs = foldInt (-) xs 0 == negate (sum xs)
```

Figure 7.1: A buggy program and associated properties

Here, we intended to implement `foldl`, but made a mistake: we accidentally wrote 0 instead of `z` in line 2. The `prop_sum` and `prop_diff` touch all locations in the spectrum, but `prop_prod` only touches the base case, since QuickCheck's initial test is always `[]`.

Running the properties for the program in figure 7.1, produces a spectrum similar to that in table 7.1. A standard spectrum consists of only the tests, whether they pass or fail,

Table 7.1: A spectrum for the code in figure 7.1 (numbers below locations specify the executions)

name	type	result	2:18	3:31	3:35-36	3:22-37	3:43	3:22-43	2:1-3:43
type			Int	Int -> Int -> Int	[Int]	Int	Int	Int	
identifier				f	xs		x		
sum	QC	True	100	2162	2255	2255	2255	2255	2355
prod	QC	False	1	0	0	0	0	0	1
diff	QC	True	100	2224	2319	2319	2319	2319	2419

and the locations involved in each test. We augmented spectra to also include the types of expressions and tests involved, the name of the identifier, and the number of evaluations of this location in the test. The notation $2:18$ represents line 2 column 18, and – indicates a range of characters. We see that for location $2:18$ the passing tests execute 100, which maps to the default amount of tries in QuickCheck, while the failing one aborts after the first attempt, namely `[]`. Using the standard formulas detailed in section 7.3.2 on the spectrum, we can score the locations as detailed in figure 7.2. Here, the most suspect location is indeed the underlined `0` in $2:18$: it is involved in more failing tests than other locations, apart from the definition of the entire `foldInt` that spans lines 2 and 3.

location	score
2:18	0.577
2:1-3:43	0.577
3:31	0

(a) Top 3 as per Ochiai

location	score
2:18	0.5
2:1-3:43	0.5
3:31	0

(b) Top 3 as per Tarantula

location	score
2:18	0.5
2:1-3:43	0.5
3:31	0

(c) Top 3 as per DStar 2

Figure 7.2: Results of some of the classic SBFL formulas, with the bug location in bold.

Although replacing the definition of `foldInt` is certainly an option, the *type information* in the augmented spectrum allows us to distinguish expressions from locations. Using the type information to deduce that $2:18$ is an expression, we can break the tie of suspiciousness-formulas and correctly point to `0` as the most suspicious expression in the spectrum.

However, it is not often as clear which location is at fault. If, for example, we had gotten the base case correct but had written `f x (foldInt f z xs)` in line 3, we would have accidentally implemented `foldr` instead of `foldl`.

```haskell
foldInt :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldInt _ z [] = z
foldInt f z (x:xs) = f x (foldInt f z xs)
```

Figure 7.3: The program from figure 7.1, slightly modified.

Running the properties again, this accidental `foldr` produces the spectrum in table 7.2. Here, it is not as clear which location is at fault: while `prop_sum` and `prop_prod` pass, now `prop_diff` fails and touches all except `f` in `foldInt f z xs` in line 3, since QuickCheck tests the empty list and then singleton lists. This exonerates the base case, but does not help us to distinguish the remaining locations. As seen in figure 7.4, formulas fall short in this case.

Table 7.2: A spectrum for the code in figure 7.1, with a fixed based case but `f x (foldInt f z xs)` in line 3

name	type	result	2:18	3:24	3:35	3:37	3:39-40	3:26-41	3:22-41	2:1-3:41
type identifier			Int z	Int x	Int -> Int -> Int f	Int z	[Int] xs	Int	Int	
sum	QC	True	100	2654	2560	2654	2654	2654	2654	2754
prod	QC	True	100	2604	2509	2604	2604	2604	2604	2704
diff	QC	False	7	4	0	4	4	4	4	11

location	score
2:18	0.577
3:24	0.577
3:37	0.577
3:39-40	0.577
3:26-41	0.577
3:22-41	0.577
2:1-3:41	0.577
3:35	0.0

(a) Ochiai scoring

location	score
2:18	0.5
3:24	0.5
3:37	0.5
3:39-40	0.5
3:26-41	0.5
3:22-41	0.5
2:1-3:41	0.5
3:35	0.0

(b) Tarantula scoring

location	score
2:18	0.5
3:24	0.5
3:37	0.5
3:39-40	0.5
3:26-41	0.5
3:22-41	0.5
2:1-3:41	0.5
3:35	0.0

(c) DStar 2 scoring

Figure 7.4: Classic SBFL formula results with the bug location in bold.

While this is a challenge to traditional SBFL formulas, we investigate a rule-based approach that allows us to distinguish these cases, by inspecting the AST structure, types, and identifiers. The rule-based approach is detailed further in section 7.3.2, but for this example, we could proceed as follows: we can filter out the non-expression by limiting ourselves to only those locations that have a type. In this case, we see that the columns for the remaining faulty expressions look the same, except for `2:18`. We then sort by the AST-based `rTFailFreqDiffParent` rule (see section 7.3.2), which assigns a value of 0.71 to `z` in `2:18`, 2.29 to `f x (foldInt f z xs)` in `3:22-41`, and 3 to all the others: most locations are evaluated alongside their parent, but `3:22-41` and `2:18` are not always evaluated with their parent (`2:1-3:41`). As the test is a property and properties test the base case first, a failure for the empty list would result in fewer evaluations, similar to what we saw in table 7.1. With that, we can rank `3:22-41` as the most suspicious.

The reasoning presented in this example conceptually follows a decision tree based on the rules provided in our work. Deriving and applying such trees is a promising approach to fault localization, but not always fruitful, as shown by our results (see section 7.4). This motivates us to do a detailed analysis to shed light on which attributes are important.

7.1.2 Contributions

In this paper, we construct spectra and apply popular existing suspiciousness scoring algorithms to Haskell and enrich them with some unique, novel features: Most faulty localization research addresses procedural or object-oriented language, and operates on block-level coverage or statement-level coverage, while we target a finer granularity on an expression level. We use Haskell Program Coverage (HPC) instrumentation to determine whether a location was touched during a test, but also to extract *how often* the location was evaluated. Tests are separated by test frameworks (e.g., QuickCheck, HUnit) to account for their individual nature and allow for the emergence of patterns. Lastly, Haskell's type system allows us to capture the *type*, constraints, and identifiers of locations that correspond to *expressions* within the spectrum, forming a new foundation for *type-augmented* spectra and fault localization.

We provide the tool for spectrum generation as an ingredient[1] (*ingredients* are the framework's nomenclature for modular add-ons) for the popular *Tasty* test framework[2]. Our spectra have multiple new attributes compared to existing spectra that increase their information, compared in table 7.3.

Table 7.3: Comparison between functional spectra and classical spectra

Common Spectrums	Functional Spectrums
Statement-level granularity	Expression-level granularity
Binary coverage	Counts number of evaluations
No inferred type-information	Types and constraints
No treatment of identifiers	All existing Identifiers included
No AST structure involved	AST structural relations analyzed
Difficulties with Tie-breaking	Nuanced ranking with rich information

The extensive research on Spectrum-Based Fault Localization (SBFL) includes sophisticated methods introduced over many years. However, just *transplanting* existing research might not be sufficient [291]. Issues can arise from different syntax and constructs (e.g., higher-order functions), community tooling (e.g., QuickCheck properties [245]) and computational challenges (expression-level spectra are much larger). Going straight for the best-of-breed approaches of Java or C might be overly complex and unfruitful due to the differences in paradigms. Hence, for this early stage of exploration, our suggestion is to adhere to explainable, straightforward approaches.

To explore the attributes and differences of spectra, we implement a rule-based approach that allows us to transplant the existing literature. For example, by defining rules for the **Tarantula** or **Ochiai** score. New information from analyzing AST structures or types can be implemented within the same rule framework. The targets for rules are (1) test attributes (test types, executions, frequency), (2) program attributes (abstract syntax tree structure), (3) existing SBFL formulas, and (4) type-based complexity measures (constraints, arity, order).

Note that while we are mainly interested in *expressions*, the coverage is based on *locations*. Expressions such as `let x = a in b`, are separated into the subexpressions `a`, `b`, and the whole expression `let ... in ...`, including the binding `x = a`, all of which are valid locations in the spectrum. Bindings, function definitions, and other nonexpression locations will not have an associated type but are included for completeness.

Based on the rule-results, we are able to investigate the data points and their relevant dimensions. Generally, we would like to know whether **spectra can be applied to a expression-level functional context**, and if additional sources, such as typing information and AST-based rules, can help localize faults. In particular: ⓐ Do faults generally occur in primitive types? ⓑ Do existing formulas perform well? ⓒ Does a type-based analysis of the locations involved increase accuracy? To answer these questions, we gather spectra from Haskell programs and analyze rule results, trying to find correlations of faulty expressions and their rules. To obtain a final suspiciousness ranking, we concatenate the

[1]under `https://doi.org/10.5281/zenodo.12168445`
[2]`https://github.com/UnkindPartition/tasty`

rule results into a vector and apply simple machine learning (ML) algorithms such as linear regression, decision trees, and (shallow) neural networks. These simple predictors are designed to maintain high levels of explainability, crucial for the intuitive inspection of cross-program generalization capabilities and for drawing comparisons with traditional SBFL formulas. Even when certain predictors may not be directly applicable in practical settings, their intrinsic value lies in the insights they offer. For instance, decision trees provide a transparent view into the rules most effective at isolating specific bugs, while regression models capture the correlations between rule attributes and the presence of faults. This helps to support human understanding and guides the development of future rules and models.

Upon investigation, SBFL for functional programs proved to be a nuanced problem and numerous challenges have been identified over the course of this work that adjusted the goal. Rather than striving for improved outcomes by selectively interpreting metrics or meta-tuning classifiers, our goal is to offer insights and trends encompassing both successful and unsuccessful techniques. This empowers fellow researchers to approach future challenges with a well-informed perspective.

This work investigates spectrum-based fault localization for Haskell to identify promising techniques and relevant factors. We provide an easy-to-adapt tool for practitioners and researchers to extract rich spectra. We use popular open-source projects to verify the feasibility of spectrum extraction and analyze Real-world bugs in detail by formulating rules that capture spectrum attributes. We re-implement existing SBFL formulas and investigate them for suitability, and explore simple ML algorithms with rule-based vectors. This will reduce barriers for other researchers and spark new educated approaches to spectrum-based fault localization for typed functional programs.

7.1.3 Research Questions

Our first research question centers on attributes of the faults and the programs themselves. To provide a basis for further research, it is important to know how many faults exist in the programs and what attributes distinguish them from their non-faulty counterparts. This also covers other attributes, such as the number of test failures and the frequency of their executions.

RQ1.A: Attributes of faulty data points and their spectra

What attributes significantly differentiate faulty and non-faulty expressions within spectra?

Before advancing existing research, it is worth looking at how the preceding literature performs for typed functional programs. We thus apply common spectrum-based formulas and see how well they perform without any additions. The inspected formulas are summarized in table 7.4.

RQ1.B: Effectiveness of SBFL formulas for typed functional programs

How well do existing SBFL formulas perform for the given Haskell dataset?

To close the investigation of the data, we try to spot correlations between different rules. Some are trivially connected, e.g., if the number of executed tests correlates with the frequency of execution, but we hope to see more complex and less obvious patterns. One suspicion that the authors hold is that higher-order functions combined with short identifiers lead to bugs, but this can only be verified by analyzing the data:

RQ1.C: Correlation of Spectrum-Rules
Are there significant correlations between the rules for faulty expressions?

Based on the original data investigation and rules, we apply a set of different simple classifiers and regressors to the data. While we do not hope to supersede existing research on their effectiveness, we hope that there is insight on *what are promising directions for the functional programming community?* Thus, we focus on explainable models and investigate their attributes after fitting:

RQ2.A: Attributes of simple SBFL Models
When fitted to a data point, what rules were the most important for the different models? Are there reoccurring patterns and weights?

The primary use of a model is to diagnose faults in (unseen) data, which makes debugging more effective. With RQ2.B we want to see how well the models perform on the programs that they are not fitted for, and if there are recurring patterns, successes, and challenges amongst them:

RQ2.B: Generalization of SBFL Models
How well do the fitted models perform on programs and faults outside of their training data?

In summary, this research aims to ⓐ analyze a sample of real world faults and ⓑ explore directions for predictors that perform better than existing formulas.

7.2 Background and Related Work

7.2.1 Spectrum-based Fault Localization

Spectrum-based fault localization (SBFL) was developed as a technique to cover well-testable issues related to the year 2000 problem [63], and is considered one of the most prominent due to its efficiency and effectiveness [297]. After defining a failing test that triggers the Y2k problem of an application, the program tests were executed in order, and their coverage was recorded. Under the assumption that there are (passing) tests covering expected behavior, the issue must originate in the statements that are covered by failing tests without being in passing tests.

The Y2K problem consists of straightforward fixes, and thus it is difficult to transfer the techniques developed there to more complex issues. Nevertheless, the idea of collecting per-test coverage to narrow down suspicious statements formed the core of modern SBFL: from the initial concept of intersection, many techniques emerged that use formulas [295, 298–300] to assign suspiciousness scores to different parts of the program. With differences in the details, all formulas take into account how often a given statement was touched by failing and passing tests, in addition to global attributes of the spectrum (e.g., total number of failing tests). The result of the formulas is used to produce a ranking of (all) statements and report the most suspicious locations.

Many refinements have been proposed: promising work revolves around the introduction of new AST elements and program states [301], the application of mutation [291, 302–304], meta- or machine learning approaches [305–307], or the filtering of tests and statements [308–310].

An important piece of work from which we draw is from Naish et al. [311] which discusses the mathematical attributes of spectrum-based formulas. In addition to introducing two new formulas, they prove that some formulas must result in the same ranking (equivalence classes). Within this work, we aim to implement at least one formula from each identified equivalence class. Due to our use of weights, investigating further individual formulas would be redundant.

This work draws from existing literature by constructing the spectrum in the same fashion and re-implementing existing formulas. Some research (Tarantula, Ochiai, DStar) is directly ported in our work. We hope to contribute to classical fault localization by providing spectra as `.csv` files to enable non-Haskellers to apply their methods.

7.2.2 Other Fault Localization efforts for functional programming

Fault localization in a functional setting has been explored in Liquid Haskell [312], using `refinement types`, a type system augmented with logical predicates. They collect constraints and localize faults by mapping a minimal set of atomic unsatisfiable type constraints to likely bug locations. The work relies on a more powerful type system than Haskell has, namely liquid types, which localize (and repair) errors on the type level.

The Liquid Haskell approach requires precise modeling of the expected system-behavior at the type level, which often means giving up type-inference

In this work, we target programs with existing test suites, and enable developers to get more out of previous testing efforts. Using liquid types, a form of test generation can form supplementary work similar to test generation efforts in program repair.

7.2.3 Related Work

Li et al. Comparable work on spectrum-based fault localization for Haskell originates from Li et al. [291]. They collect open source bugs and apply existing SBFL formulas on an expression-level spectrum. To improve generalizability and introduce an ML approach, the programs were also mutated to extend their data set. Although they publish the dataset which we reuse, the original code is not available. Li et al. have similar goals in introducing SBFL for Haskell, but many of the details differ. On a more fundamental level, our spectra extend previous work with unique attributes of types, tests, and identifiers. We introduce rules that extend the existing literature to capture more concepts than SBFL formulas currently can. Their approach includes data augmentation, which forms a great venue to synthesize the efforts of both works in future research.

HaskellFL HaskellFL implements the Ochiai and Tarantula algorithms for Haskell code [313]. They develop a custom compiler that compiles the program into SKI-combinators for evaluation, to determine the lines involved in a fault. As they do not integrate with

HPC or GHC, an application for real-world programs proves difficult, and no evaluation on large programs is provided.

7.3 Implementation & Experiment Setup

7.3.1 Spectrum Generation

We introduce a tasty-spectrum package which adds an *ingredient* to the Tasty test framework that captures coverage when tests are run and generates a spectrum. Tasty-Ingredients are a modular way to implement plugins for Tasty to add additional behavior around tests such as re-running, timeouts, or, in this case, data extraction.

To generate spectra, we use the instrumentation provided by Haskell Program Coverage (HPC) and programs compiled with the `-fhpc` flag. This generates `.mix` files that allow HPC to connect the indices it produces to the source locations in the modules. Our implementation also includes a *GHC source plugin*, which integrates with the compiler and extracts type and identifier information from modules during compilation and generates `.types` files.

GHC Source Plugins GHC allows users to define *source plugins*, which are run at the end of various stages of compilation, including parsing, type-checking, and renaming. These plugins allow the user to modify and interact with the source code after each stage. In the tasty-spectrum package, we define a plugin that operates at the end of the type-checking stage, where we traverse the type-checked expressions, and note their types in the `.types` file. The `.types` files are saved alongside the `.mix` files and later combined with the `.mix` information during spectrum generation.

Haskell Program Coverage (HPC) HPC instrumentation is integrated into GHC, and is based on maintaining an array that counts executions for each source location (which corresponds to expressions) in the module during runtime. Whenever an expression is evaluated, this also triggers a “bump” in the array, allowing HPC to track the number of times each expression was evaluated in the module. Outside of *standard behavior*, HPC also allows for the setting, reinitializing, and accessing this array at runtime.

Spectra are generated by running the test suite. As the code has been compiled with the `-fhpc` flag, the RTS will keep the Tix array in memory. Before running each test, we reset the HPC state. After each test, we read the current state of HPC, and track which expressions were evaluated.

After running all tests, the Tix array for each test is combined with the module structure from the Mix files and the type/identifier information from the `.types` files to produce a *type-augmented spectrum* as a `.csv` file. This `.csv` file contains the names of the tests, their results, the full path of each location involved in each test, and how often times each location was evaluated per test. The locations themselves include information about which lines of source code correspond to the location, as well as its type and identifier, if available. To compress the data, we only include locations that are involved in any of the tests, silently dropping those that have zero evaluations across the test suite. The `.csv` file containing the spectrum is available for further processing with our library or alternatively with other external tools.

7.3.2 Rules

Fault localization commonly ranks locations based on their *suspiciousness.* To achieve this, the information in the spectrum is quantified and turned into a score. This is traditionally done using formulas that depend on the number of times a location is involved in passing or failing tests, n_{e_p} and n_{e_f} respectively, and the number of total passing and failing tests, n_{t_p} and n_{t_f}.

In our analysis, we include these classic formulas, but we also quantify other elements of the augmented spectrum, aiming to find correlations with faults.

- *Test-type count* the number of tests, passing or failing, that this location is involved in.
- *SBFL-Formulas* apply existing formulas from previous literature; the rule output is the calculated value of the formula.
- *AST structure-based rules* use information based on the distance from a failing location or whether a parent or sibling was executed often.
- *Type-based rules* are based on structural analysis of the typed locations to quantify them for further downstream analysis. This analysis is done after the generation of the spectrum to reduce any dependency on specific GHC versions after the generation of the spectrum. As a downside, we work with a parsed string representation of the type and not the type as it appears in the typechecker, disabling instance resolution or further analysis beyond what the type tells us.
- *Meta-rules* operate on the results of the previous, per-module, rules and supplement the data with further analysis. These include the quantile rules and the rules that count how often types, component types, and identifiers appear in failing tests.

Table 7.4 provides an overview of the type-based rules.

We want to further motivate some of the rules presented in table 7.4. One general notion is that properties are *stronger* than regular unit tests, as they cover a wider range of input values and have logic beyond an *assert.* It makes sense to rate an expression that is in many passing properties as less suspicious. In a similar, less algorithmic viewpoint, golden tests, i.e. tests that use output comparison, are often written after users report a bug. Instead of writing unit tests to follow up, the report is used to create a failing test case. Thus, it could make sense to rate golden test failures as more suspicious, as they often capture failing behavior, contrary to properties that often test positive program paths. Lastly, we will use the test frequency and other patterns by separating the test frameworks. Golden tests often span wide ranges of code (such as system-level tests), while properties should result in many executions. Taking this into account, there is *no one test better than the others* - but there might be patterns that we only find when inspecting them separately.

AST rules are based mainly on existing research on active and algorithmic debugging [195, 217, 314]. `rFailUniqueBranch` aims to capture uniqueness in execution and amplify such patterns. If an expression is in the only failing branch of a function, that is a good indicator for investigation. Similarly, `rFailFreqDiffParent` identifies critical irregularities in program execution, such as recursion bases, pattern matches, or monadic constructs, which, although less executed, are of disproportionate significance.

Table 7.4: Overview of the rules in the rules-based system.

Rules	Description
Test-type count	
rTFail & rTPass	Total number of failing tests involving this location
rPropFail & rPropPass	Number of failing QuickCheck tests involving this location
rUnitFail & rUnitPass	Number of failing unit tests involving this location
rGoldenFail & rGoldenPass	Number of failing golden tests involving this location
rOtherTestFail & rOtherTestPass	Number of other failing tests involving this location
rTFailFreq & rTPassFreq	Sums the number of evaluations in failing and passing tests involving this location.
Formulas from SBFL literature	n_{e_p}/n_{e_f} is the number of passing/failing tests the expression is involved in, while n_{t_p}/n_{t_f} is the total number of passes/fails.
rJaccard	$\frac{n_{e_f}}{n_{e_f}+n_{t_f}+n_{e_p}}$
rHamming	$n_{e_f}+n_{t_p}$
rOptimal	$\begin{cases} -1\ if\ n_{t_f}>0 \\ n_{t_p}\ otherwise \end{cases}$
rOptimalP	$n_{e_f}-\frac{n_{e_p}}{n_{e_p}+n_{t_p}+1}$
rTarantula	$\frac{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}}{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}+\frac{n_{e_p}}{n_{e_p}+n_{t_p}}}$
rOchiai	$\frac{n_{e_f}}{\sqrt{(n_{e_f}+n_{t_f})(n_{e_f}+n_{e_p})}}$
rDStar 2 & rDStar 3	$\frac{(n_{e_f})^2}{n_{t_f}+n_{e_p}}$
rRogot1	$\frac{1}{2}(\frac{n_{e_f}}{2n_{e_f}+n_{t_f}+n_{e_p}}+\frac{n_{t_p}}{2n_{t_p}+n_{t_f}+n_{e_p}})$
AST structure-based based rules	
rASTLeaf	Counts the distance of this node to the nearest leaf
rFailUniqueBranch	How many times this location is involved in a failure that none of its sibling expressions is involved in.
rFailFreqDiffParent	Checks how many times this statement is evaluated compared to how many times its parent is evaluated. Sums up the ratios for the total.
rDistToFailure	How far this location is from a location involved in a failing test, counted by the sum of the number of links to a common parent.
Type-based formula rules	See table 7.5

With `rDistToFailure` we hope to produce a *taint* that accumulates over test failures, and failing expressions raise suspicion of nearby code.

With the group of type rules in table 7.5, we aim to proxy the complexity of an expression and its context. We expect longer types to indicate a more complex process; especially higher-order functions are a unique case of complexity that is well represented at the type level. Other attributes, such as `rTypeContraints`, can indicate a stricter environment that limits possible faults, since the types are more restrictive. `rNumSubTypeFails` aims to connect types seen in failing locations with seemingly un-connected locations — the rationale being that concepts in the program are expressed as types, and there can be a failure in the concept. Some of the rules have a different character: `rTypeArity` and `rTypePrimitives` allow us to identify correlations of faults with parts of a type and form more basis of analysis than direct, actionable suspiciousness. Nevertheless, it can be worth seeing for programs and bugs if faults occur in basic elements or complex compositions.

Unlike SBFL formulas, our novel rules are not meant to be finished ranking algorithms. Instead, our objective is to capture information and suspicious elements to combine it in later processing and analysis, mainly to allow tie-breaking, as seen in table 7.2.

Table 7.5: Rules based on the type of the expression at the given location.

Type-based formula rules	
rTypeArity & rTypeConstraints	Number of arguments and the number of constraints the function has.
rTypeArrows	Number of arrows `(->)` in the type
rTypeFunArgs	Numbers of parentheses in the type to quantify how many *function arguments* there are, and in turn whether it is a higher-order function or not.
rTypeOrder	Counts the number of type applications in the type, such as `Maybe a` or `[[a]]`
rTypePrimitives	Number of primitives, i.e. `String` or `Int`.
rTypeSubTypes	Counts the number of types in the type, i.e., unfolds all constructors and applications.
rTypeLength	Number of Characters of the Type, when represented as `String`.
rNumSubTypeFails	Number of times types which appear in this type are involved in a location involved in a failing test.

7.3.3 Data

A challenge in bringing SBFL to a Haskell setting has been the lack of available datasets for evaluation. The data source for this article comes from two recently published Haskell fault datasets, `HasBugs` [228] and `HaFla` [291]. Both datasets provide a similar granularity of faults originating from projects with known faults (based on issues and PRs) whose fault-fixing commits include a test. These tests were extracted to produce a *faulty but tested* version with a failing test suite. The faulty locations are extracted from the git-difference of the source files. We determine faulty expressions as all expressions that are completely within faulty lines.

A subset of the data was chosen to produce the spectra that met the required versions of Cabal, Tasty (>v1.0), and GHC (>= 8.6). Some other limitations excluded projects like Purescript (many of the tests run against compiled Javascript code) or Cabal (all bug-asserting tests are package-level tests outside the Tasty test suite). This results in a total of 11 programs[3] from 3 projects - **Pandoc**, **Duckling** and an **HLS**-plugin. An overview of the data points used is presented in table 7.6.

Pandoc is a document converter and, outside of language-specific tooling (GHC, Cabal, HLS, etc.), the biggest open-source Haskell project with over 50k lines of code. The general *flow* of conversion consists of three steps: a reader, an internal representation, and a writer. Most bug reports and issues are based on user-perceived misbehavior, which is commonly captured with a unit or golden test.

HLS is a joint community effort of Haskellers to provide the backbone of a modern Haskell IDE. Most of it is centered on providing a language server in typescript style for the popular Visual Studio Code. Apart from a base framework, many functions are provided as plugins to cover linting, type suggestions, suggested imports, and other features.

Duckling is an open-source Facebook project that extracts structured entities (times, dates, weights, etc.) from texts. The general business logic consists of regex-based rules that are applied in a fine-to-coarse fashion (more specific Australian-English rules supersede generic English rules). The test suite consists of a domain-specific corpus with examples and *broad* tests that run all examples within a corpus. Generally, the corpus is structured per module, which is why the duckling data points only show one test failure, despite multiple examples being added to a corpus. Duckling has more LOC than Pandoc (180k), but much of the code is template Haskell generated.

Comparison with Defects4J - Comparing the spectra between paradigms is challenging, but to approximate, we consult some data from Defects4J [58]. Although other benchmarks exist (e.g., the Siemens Dataset[315], NoFib-Buggy [70] or BugBench [316]), Defects4J matches the character of our dataset the most. Both datasets are drawn from public open-source repositories, and their faults and tests are extracted from commits. This makes for a natural state of the projects. Siemens data, for example, has (unrealistic) test coverage of 100%, while NoFib-Buggy has artificially introduced bugs. We draw our data from a public repository shared by René Just[4] that provides statistics from applying GZoltar [79] to a subset of 395 bugs from Defects4J.

The Defects4J bugs inspected have a mean SLOC[317] of 57.7k and a median of 62.5k. The mean number of tests in Defects4J is 1439, with a median of 202. Both the mean and median of failing tests per bug is two. Under the assumption that most of the SLOCs represent line-level statements, the resulting spectra will have a comparable number of elements. As the size of bugs (*Faulty LOC*) is not reported in Defects4J, we estimate it by inspecting the patches and counting the removed lines. We filter the removals for

[3]5 from HasBugs, two from HaFla, and four more data points sourced by the authors
[4]https://bitbucket.org/rjust/fault-localization-data/src/master/

lines consisting only of whitespace or only containing opening or closing brackets. The approximate faulty LOC for Defects4J is, on average, 2.56. In conclusion, the programs and bugs used in this work are comparable in size to Defects4J.

7.3.4 Experimental Setup

Within the experiments, we first set up all the programs in the same manner. Based on the fault fixing commits of a data point, we revert the source code patch while keeping the changes to the test code, observing a test failure during `cabal test`. At this stage, we also distinguish *noisy* test failures as presented in table 7.6. Noisy test failures are failures of tests that are *unrelated to the patch*, i.e., tests that fail due to some circumstance of the environment rather than the code. As the next step, the cabal file is altered to include spectrum generation and coverage, following the description in section 7.3.1 These result files form the basis of a data analysis, done in Python.

RQ1 is answered by investigating the results of their triggered rules. Many of the spectrum attributes are directly captured in rules (e.g., `rTFail` corresponds to *was touched by a failing test*), and thus facilitate the analysis of distributions and proportions.

The primary metric considered for ranking the expressions is the `Top-`X-metric [318], This originated from recommendation system research and was widely adopted for fault localization. Within `Top`X, the recommended elements are sorted by their *suspiciousness*, and the *correct* classifications (truly faulty expressions) within the first X are counted. This metric makes it easier to determine how well the rankings perform, and a reasonable X is an acceptable number of locations for developers to inspect. For this work, we considered the `Top10`, `Top50` and `Top100`, following previous literature.

Another common metric is EXAM [319], assuming that the user follows every recommendation in order until the real fault(s) are fixed. The index of the first correct fault is used to calculate the ratio of the inspected (total) program, with the exam score expressing *how many locations can be skipped when following the recommendations?* The EXAM score is proportional to the *mean reciprocal rank*, another metric commonly reported for FL. For this work, we discarded MRR and EXAM, as we work with different granularity due to our expression level spectrum: when introduced in 2003, EXAM was targeting block-level spectra, but the sheer difference in the quantity of mostly (benign) expressions would draw a highly beneficial picture of our approach. Therefore, for ranking evaluations, we focus on the `Top`X metrics [76].

RQ2 is investigated by training classifiers and regressors on the `result files`. Namely we implemented `decision trees`, `random forests`, `linear-` & `logistic regression` and `Multilayer Regressors` from SciKit [320]. At last, we considered a genetic algorithm using Pymoo [321] for an evolutionary search of regressor weights.

To separate the effects of the new rules from existing rules, we assert a total of four configurations:

1. *all* - all existing rules
2. *classic* - only pre-existing SBFL formulas
3. *original* - only the novel rules of this work

Table 7.6: Overview of the used data points

Data Point	Issue	Faulty LOC	Faulty Expressions	Total Expressions	Failing Tests	Noisy Test-Failures	Total Tests
pandoc-3be256efb	Wrongful application for 'Big Note' highlighting when converting to LaTeX. Reordering is necessary.	1	6	88k	6	0	3254
pandoc-4	Failure in converting combined code and bold highlighted text to latex.	3	12	91k	1	1	3056
pandoc-5	Miss-interpretation of code blocks when converting to ROFF MS. Requires escaping.	1	8	61k	2	6	2400
pandoc-6	Miss-converting code blocks starting with (1) into enumerations.	5	39	59k	10	13	2365
pandoc-7	Failure picking up empty cells in multi-cells when reading latex.	27	72	61k	3	7	2415
hls-2	Issue accounting for relative location "./" instead of expected "."	2	15	269	1	0	6
hls-afac9b18	HLS-Plugins can reformat code, Stylish Haskell was removing the last line of files regardless of whether they had content.	1	17	122	2	0	13
duckling - ea8a4f6d	Wrong pronomina for German million. Adjustment of Regex.	1	5	288k	1	0	364
duckling - 4cfe88ea	missing cases for combined durations such as "2 hours and 20 minutes".	18	4	260k	1	1	342
duckling - 28ddc3bf	Wrong parsing of `1.000,00` for dutch.	1	5	299k	1	0	346
duckling - 328e59eb	Missing cases for weights (and their combinators) in Portuguese language.	19	26	277k	1	1	360

4. *cherries* - handpicked subset of rules

To account for different value ranges, we re-run all experiments with `min-max-scaling`, mapping all column values between 0 and 1.

Fitting the binary classifiers (decision tree, random forest, logistic regression) targets locations to be faulty or not faulty. Regressors are trained to assign faulty locations with a suspiciousness of 1 while other locations have a suspiciousness of 0.

GA-based regression . GAs utilize a custom fitness function to optimize the ranking of the first reported faulty locations, effectively optimizing on `Top`X. For GAs, we set the population to 200 individuals and use Latin Hypercube Sampling [322] to generate the initial population. The population is then evolved trough subsequent generations, by using *binary tournament selection* [323], for selecting the solutions (regression weights) for reproduction based on their fitness. *Simulated Binary Crossover* [324] SBX is used to recombine the selected solutions, and *polynomial mutation* [325] (PM) is used to introduce diversity to the population. We opt for these genetic operators and their recommended parameters values (i.e., SBX with index $\eta_c = 30$, PM with index $\eta_m = 20$ and probability $p_m = 1/n$, with n being the number of regression weights), as they are known to be effective in solving continuous optimization problems [325]. GAs are set to run for 2000 generations or terminate early if no improvement in the fitness function is observed for 100 generations. The solution (vector of weights) in the final population with the best value of the fitness function is used as the final GA-based regression.

Regressors are evaluated on the resulting `Top`X, while for classifiers, true and false positives are evaluated. A global seed was used to account for inherent randomness.

7.4 Results

7.4.1 Attributes of Spectra

The created spectra range in size from 25Kb (HLS), 200 MB (`duckling`) to up to 500 MB (`Pandoc`). Spectrum generation is not a costly addition to the runtime of the test suite, but we emphasize that the compilation time of some projects is longer as the `-fhpc` flag for coverage is required. Upon code changes, e.g. during debugging, a recompilation is necessary to realign the `.mix`-entries.

The first important view of the data points is presented in table 7.7. The table groups the expressions into those touched by failing tests and those that are not, a common approach to reduce spectrum sizes. The rationale is that statements without failing tests are *innocent*, under the preposition that a correct test was written. When organized in this way, we see that `duckling-4cfe88ea`, `duckling-ea8a4f6d`, `duckling-1dac46a8` and `pandoc-4` do not have faults covered by the tests.

The authors double-checked the test suite, and for duckling, the correct (and expected) corpus tests were failing. On inspection, we suspect that the tests do not run against the *original source*, but against the generated code. The generated code is also faulty but is not the origin of the issue, as fixed in the commit. Some of the duckling data-points, e.g. `duckling-328e59eb` have faults covered by failing tests. The fix for `duckling-328e59eb` is more than the adjustment of a regex, and the changes to the structure are successfully tested and represented in the spectrum.

Table 7.7: Test-coverage within gathered spectra

Program	Expressions covered by failing Tests	Expressions untouched by failing tests	Faulty Expressions not covered by failing tests	Faulty Expressions covered by failing tests
hls-2	205	64	1	14
hls-afac	35	87	0	17
duckling-4cfe88ea	1791	297705	4	**0**
duckling-328e59eb	1165	275942	0	26
duckling-ea8a4f6d	2541	286195	5	**0**
duckling-28ddc3bf	2256	260307	0	5
pandoc-4	419	90669	12	**0**
pandoc-5	238	60410	0	8
pandoc-6	2175	57203	34	5
pandoc-7	2235	58839	34	38
pandoc-3be256efb	623	88149	0	6

We have different suspicions for `pandoc-4` as there are faulty locations on a *reader* that need changes in the data format. The relevant test is a `golden test` that runs with a compiled binary of `pandoc` (unlike the other `pandoc` data points) that is invoked by Tasty. Still, their coverage is not collected in the project coverage. Thus, we have a failing test suite, but the *touched* expressions originate only from noisy test failures.

The existence of faults that are *not directly covered* poses a challenge for this work and a novel aspect of fault localization. Our results come from real projects, and it would be common to adjust `duckling` tests only by providing more examples in the corpus. Although this test covers the bugs *semantically*, it does not cover the faulty code and may require new spectrum techniques. To some extent, these tests are juxtaposed to *automatically generated* tests, which cover code behavior without necessarily capturing the semantics of faults [326–328]. Due to the common usage of Haskell for domain-specific languages, parsers, and code generation tooling, we expect these types of faults to be more common in functional paradigms than in other languages. They also justify our motivation to implement rules that utilize more information outside the test status and coverage.

Other relevant attributes are that, on average, 63.7% of faults are in AST leaves, while 50.5% of expressions are leaves. For duckling, most changes were adjustments to a regex (AST-Leaf) and their wrappers (non-leaf) or required the introduction of a new rule (rule-invocation is an AST leaf, adjustments to the list of rules are non-leaves). This means mainly an even distribution of faults in leaves and non-leaves for duckling. Within Pandoc, many faults revolved around combinators and parsers, which involve many higher-order functions. In particular, the program flow in a parser monad produces many non-leaf faulty locations. The combinators (<$>, <|>, etc.) and the patterns (`many1Char`, `noneOf`, etc.) are all non-leaf nodes as they require arguments. Due to this structure, the faults in the pandoc programs are proportionally more in non-leaves than leaves, as most faulty primitives are applied multiple times and structured in a monad.

Initially, we considered that reducing the spectrum to only leaves could efficiently support algorithms without requiring an *oracle*. The reduction does not directly help differentiate faulty and non-faulty expressions, but it can significantly help performance by halving the size of the spectra. We did not filter leaves for further experiments to exploit the `rASTLeaf`-rule, which we considered a promising candidate justifying the extended runtime.

Most faulty expressions are typed. Usually, one or two faulty locations are untyped, which is a special case of ambiguity that occurs in typing: these are not *expressions*, but rather bindings, e.g. `x = a`. Here, `x` and `a` will have the same type, but the *binding* `x = a` does not have a type.

We see no striking trends in the types of faulty expressions: The most common types are primitives such as `Text` or `UInt`, which are also common in non-faulty expressions. The only exceptional types are monadic parsers in `pandoc-6` and `pandoc-7`. The use of monads and the higher-order operators involved is also a reason for the high number of faulty expressions for these data points, as they imply an increased number of function applications per line of code.

Although most expressions are typed, only a few represent an identifier. Less than half of the faulty expressions correspond to an identifier, and 4 data points do not have any faulty expressions that correspond to an identifier. The identifiers encountered match the project vocabulary (e.g., `parseMultiCell` in `pandoc-7`) and there are no trends of shorter identifiers being more faulty. In general, this was a bit unexpected since much of the existing research focuses on *off-by-one* errors [194] or issues in predicates [329], which also focus on elements with identifiers.

RQ1.A: Attributes of Spectra

Three data points (`pandoc-4`,`duckling-4cfe88ea` & `duckling-ea8a4f6d`) do not have faulty expressions covered by a failing test, despite semantically correct failing tests. This is due to projects that make use of code-generation (`duckling`) and the test suite running binaries (`pandoc`). Two-thirds of expressions are AST-leaves, whereas about half of the faults are AST-leaves. Almost all faulty expressions have a type, but identifiers are rare.

7.4.2 Existing SBFL-Formulas

Figure 7.5 shows the `Top50` results when applying existing formulas and sorting the statements by their resulting score. Although `Top10` metrics are sometimes presented in the literature, most formulas seem to struggle with expression-level granularity to produce a meaningful ranking for our programs. The `Top100` produces better results, but the trends remain the same, which is why we opted to focus on the `Top50` for this section.

As seen in figure 7.5, Ochiai is the formula that performs best with our data, followed by DStar. A critical point to note is that Ochiai is the only formula with a median `Top50` above zero, implying that the other formulas have not found faults for more than half of the data points. The details of the *best* formulas are presented in table 7.8.

We expect that Ochiai is the best performing metric as it applies the square root in its denominator, which scales better for large numbers. As we face a high number of expressions and tests, this scaling is more applicable than other formulas (DStar, Tarantula). Ochiai, DStar, and Optimal also do not use n_{t_p} (number of total passing tests), which is relatively high for most programs and disproportionate to the number of failing tests.

When inspecting table 7.8, it becomes clear that the best average scores are achieved by the strong performance of some formulas on `pandoc-6` and `HLS-afac9b18`. Our educated guess is that `pandoc-6` has a large number of failing tests that exactly distinguish the faulty from the correct cases. `HLS-afac9b18` has a much more favorable ratio of faulty expressions to expressions, and the newly added tests primarily invoke the affected faulty statements. Thus, these two data points play into the strengths of formulas due to their test quality (with respect to the project size).

The data points `duckling-4cfe88ea`, `duckling-ea8a4f6d`, `pandoc-4` and `pandoc-3be256efb` did not result in `Top50` for any of the existing formulas. Again, we suggest that this is mostly due to the test suite and its attributes highlighted in the previous subsection. Without faulty expressions that are covered by failing tests, most formulas result in a suspiciousness of 0. Furthermore, formulas that include *passing tests* also struggle with the duckling data point, since most expressions are covered by only one or a few passing tests. These few tests are *rich* as they contain multiple examples, but do not take advantage of the strengths of the considered formulas.

7

Before closing the analysis of existing SBFL formulas, we emphasize that the overall quality of the formulas is quite high. The small data points of HLS are especially well predictable with formulas, motivating applications for script-sized programs. For the versions considered of `duckling`, folding tests into a corpus in combination with code generation makes formulas inapplicable.

RQ1.B: Existing SBFL Formulas

`Ochiai` and `DStar` produce the best `Top50` results with an average of 4.5 and 4.1 errors correctly reported in the first 50 expressions. The `Top10` metric seems to be too hard for the expression-level granularity. All formulas struggle with `duckling` and `pandoc-4`, due to the faulty expressions not being touched by failing tests. This is a challenge to all spectrum-based methods, and not specific to the functional context.

7.4.3 Applicability of Rules and Correlations

Correlation to investigate the correlation, we applied the Pearson correlation coefficient after combining the spectra across projects. Total correlations are shown in figure 7.6 and a summary of the significant rules in figure 7.7. The purple connections in figure 7.7 are significant in both total and *faulty-only* data while the yellow connections are significant among the faulty data points. We also calculated the Spearman correlation and observed the same trends, with more individual correlations being significant.

Some correlations verify our assumptions that we considered trivial, e.g. that type

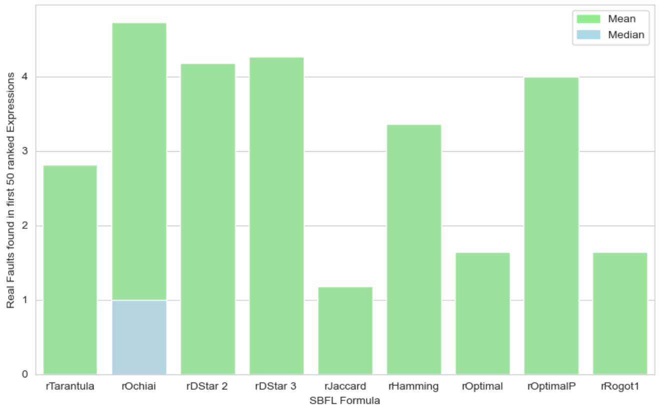


Figure 7.5: Top50 Results of SBFL Formulas

Table 7.8: Formula Top50 Results

Program	Faults	Tarantula	Ochiai	DStar 3	OptimalP
hls-2	15	2	2	2	2
hls-afac9b18	17	17	17	17	17
duckling-4cfe88ea	4	0	0	0	0
duckling-328e59eb	26	1	1	4	0
duckling-ea8a4f6d	5	0	0	0	0
duckling-28ddc3bf	5	0	0	0	0
pandoc-4	12	0	0	0	0
pandoc-5	8	8	8	0	0
pandoc-6	39	0	21	21	25
pandoc-7	72	3	3	3	0
pandoc-3be256efb	6	0	0	0	0

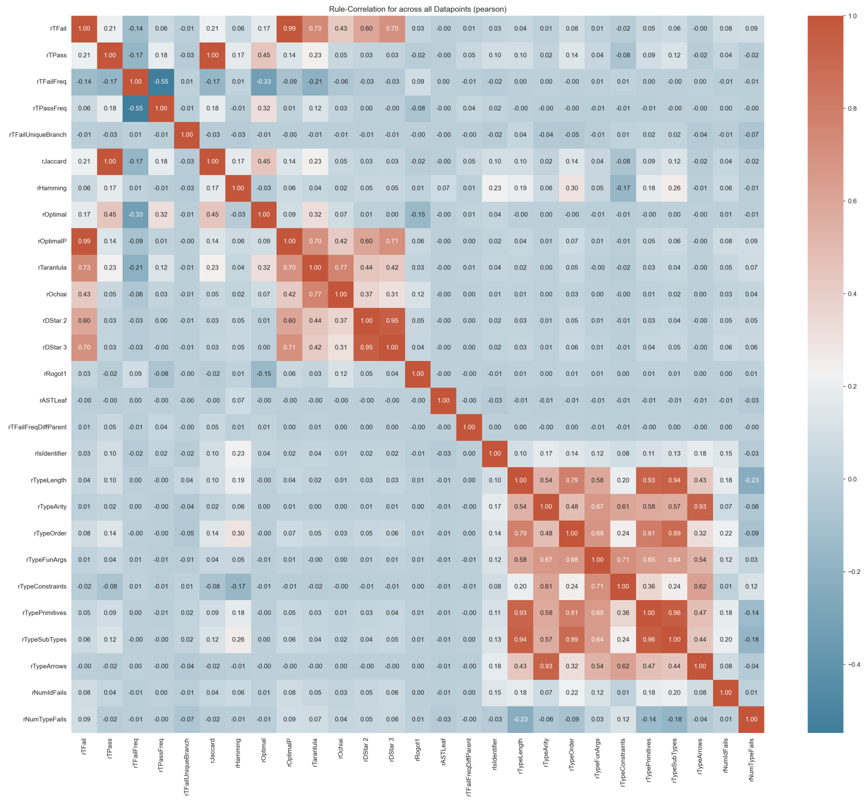


Figure 7.6: Pearson Correlation Matrix

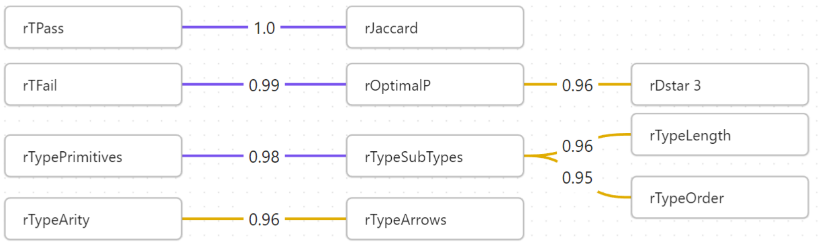


Figure 7.7: Significant Pearson Rule Correlations

lengths correlate with the number of subtypes. In general, type rules form a block in figure 7.6, since more complex types are *longer*, have a higher arity and order, and have more function applications. For most type-based rules, this relation is not statistically significantly correlated. The second block we see is the SBFT formulas from literature with Ochiai, Tarantula, DStar, and OptimalP. This is mathematically plausible, as they are proportional to n_{e_f}, the number of failing tests for this expression, in their formulas (see table 7.4). This is also expressed by the correlation with `rTFail`. `OptimalP` and `Jaccard` similarly correlate with `rTFail` and `rTPass` due to their definitions.

Most rules do not have a significant correlation with each other, and, except for the two blocks, there are no other visible trends. Although this may initially seem underwhelming, we want to stress that most rules do not correlate.

For example, `rTFail` and `rTFailFreq` do not correlate significantly within our data, implying that the execution frequency is not directly related to the number of tests. In the same way, `rTPass` and `rTPassFreq` are not correlated. This finding motivates us to investigate formulas focusing on evaluation frequency, as they seem more distinct from test failures than expected.

In general, the lack of correlation can be interpreted as a chance. The rules introduce information that is not directly correlated with existing measures, This means that they cover new aspects of complexity or unique spectrum attributes. We expect imperative programs to have similar patterns, but they can only be found as clear in functional programs. Inferred type information at the expression level is uncommon in other paradigms, and investigating correlations between types, constraints, arity and faults is out of reach for most imperative languages.

RQ1.C: Rule Correlations

Most rules do not correlate according to the Pearson coefficient. Type rules and popular SBFL formulas form (mostly non-significant) trends within the correlations. The lack of correlation implies that different rules cover different aspects of a program.

7.4.4 Attributes of SBFL Models

Logistic & Linear Regression In both logistic and linear regression for both scaled and unscaled data, the resulting weights result in significant variance, indicating overfitting. For example, many type rules differ in their polarity for logistic regression despite the rules correlating (see RQ1.C).

Decision Trees Decision trees required a class-balanced fitting using an entropy measure to produce sufficient results. A visible trend is the reproduction of the SBFL formula rankings as in figure 7.8. Given the effectiveness of Ochiai, as observed in RQ1.B, this is an understandable result.

For the larger programs (`pandoc-6` & `pandoc-7`), the trees often resulted in configurations that lean left or right with single expression branches. Tree pruning could not address this form of overfitting, as the resulting pruned trees remain with a high entropy.

Explainable conditions, such as *if it is a leaf, use Ochiai. Otherwise Tarantula*, were unfortunately not observed. The combinations that were striking to the authors are those that use one of the well-performing metrics (e.g., `DStar`), as root of the tree, and then use

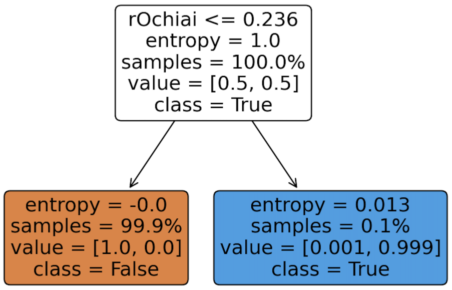


Figure 7.8: Decision Tree for Pandoc-5 (scaled data, `classic`-rules)

a more exotic rule such as `rHamming`, `rRogot1` or `rNumGoldFails`, which apply to very few locations.

The prominence of overfitting makes us wonder if decision trees can be successful for the task without combining programs. There may be too little information (persisting entropy) or an algorithmically determined overfitting. This shortcoming of decision trees is known and motivates the use of a random forest ensemble.

Genetic Algorithms A key observation is that genetic algorithms (GAs) faced convergence challenges with specific programs: `pandoc-4`, `duckling-4cfe88ea`, `duckling-ea8a4f6d`, and `duckling-28ddc3bf`, notably exhausting the maximum number of generations allocated without achieving early termination. The non-convergence co-occurs with the absence of *touched* faults. Our educated guess is that ⓐ it is hard for randomly generated weights (that is, the initial population) to produce any correct ranking, and ⓑ for the *untouched* faults the individuals who classify faults are uniquely picking single attributes and the combination skews the weights again. Individuals that rank faults are *fragile*, and mutation and combination lose beneficial attributes, stopping the genetic search to stagnation.

Random Forests & Multilayer Perceptrons The previously investigated *simple* algorithms lead to two orthogonal problems: on the one hand, models overfit the data, while, on the other hand, less detailed parameters and pruned trees could not reduce entropy.

This motivated the use of models that allow for ensembles (random forests) and multi-dimensional interpretation of the data (MLPs).

RQ2.A: Development of SBFL Models

Most models struggled with forms of overfitting. Especially linear and logistic regression, as well as decision trees, struggled with the sparse data. Genetic algorithms face issues converging for programs with untouched faults.

7.4.5 Generalizability of SBFL Models

Classifiers When investigating the classifiers (decision trees, random forests, and logistic regression), an early finding was that all three generalize better on scaled data. An overview of the transfer performance of the classifiers is shown in figure 7.9.

In figure 7.9, we see the trends in which classifiers are grouped according to their false and true positives. Logistic regression produces many true positives and false positives (≈90% false positives). Put in perspective, for many data points, a logistic regressor will give 100 faulty candidates, of which nine will be true faults. Although this is likely frustrating for developers, it can be suitable for tooling (see section 7.5.4).

The best performance with good precision was achieved by random forests using only SBFL formulas. On average, an ensemble of formula-based decision trees reports five faults, of which ≈2.5 will be true faults. This is a convincing rate for actual usage, given that the reported numbers are averages. For many programs, random forests (and decision trees) were not reporting faults as they were not certain enough. This leads to a low number of true positives, but upon author inspection, most of the actually suggested faults were either true faults or reasonably close.

Throughout the configurations, the classic SBFL formulas performed best in all classifiers. This is due to their good performance on data points with high faults for which the original formulas also performed well (`pandoc-7`). The `all-rules` and `cherries` find fewer faults and produce more false positives, but are better at predicting faults of our most troublesome data points `pandoc-4`, `duckling-4cfe88ea` and `duckling-ea8a4f6d`. Depending on the goals, the logistic regression with cherry-configuration is able to predict faults that were not touched by failing tests, especially at the cost of a high noise ratio.

Regressors Across the board, the regressors performed better on the unscaled data and primarily produced good `Top50`-scores on the data points with faults covered by failing tests. Due to poor performance, we present only examples of regressors when compared to data points that have locations touched by failing tests. Most regressors performed worse than existing formulas, with the exception of genetic algorithms. An overview of the averaged search results for `Top10` and `Top50` is given in figure 7.11 and figure 7.10.

We see that especially for the `Top10` genetic search produces much better averages than the formulas. This is only true for the mean, the absence of the median for most elements indicates that more than half of the data points did not produce any correct `Top10` suggestion. Only the `original` and `cherries` found a median of 0.5 `Top10`.

In general, we consider median results to be more relevant since the data points are unbalanced in the number of faults. A data point that performs well on the many `pandoc-7` faults will have a decent average but will not produce good results for most data points. Therefore, the median is a more reliable metric to judge the performance of fault localization, given our fault distribution.

We must stress that the averaged results only indicate the most fruitful configuration - the results varied greatly from regressor to regressor and per target data point. Thus, we want to highlight two types of well-formed searches in figure 7.13 and figure 7.12. The orange bars indicate the achieved `Top50`-score, while the blue frame indicates the maximum possible faults.

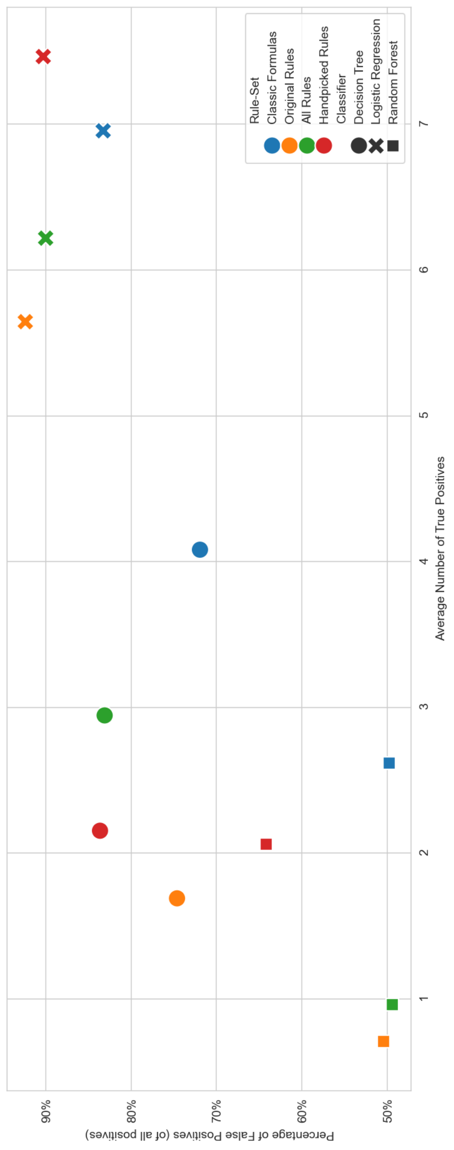


Figure 7.9: Transfer Performance of Classifiers

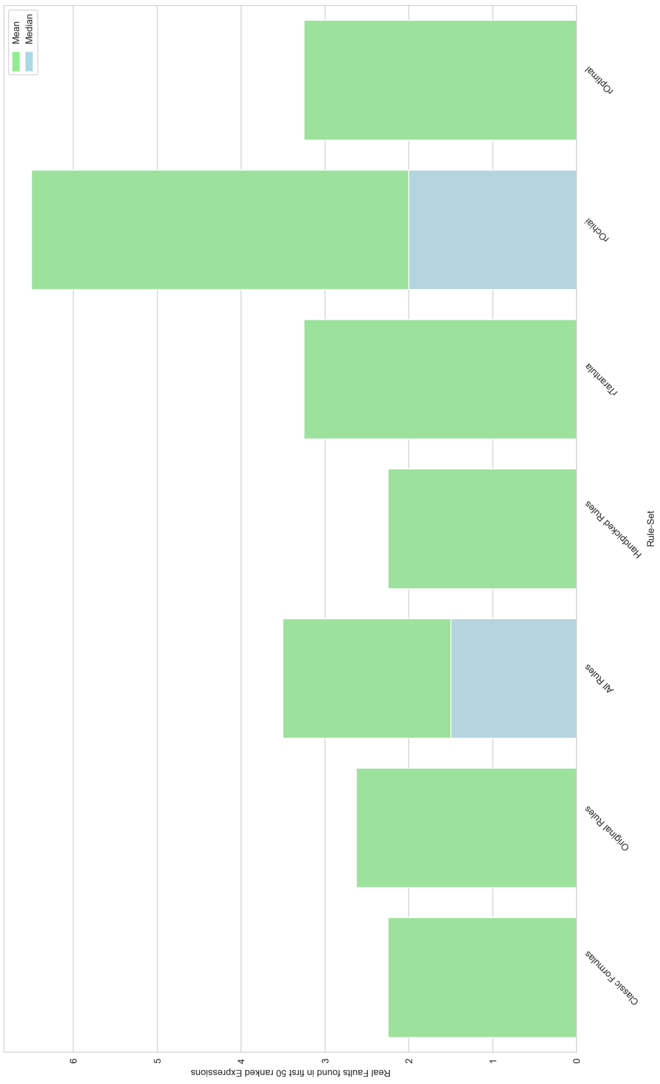


Figure 7.10: Averaged Top50-score for genetic search

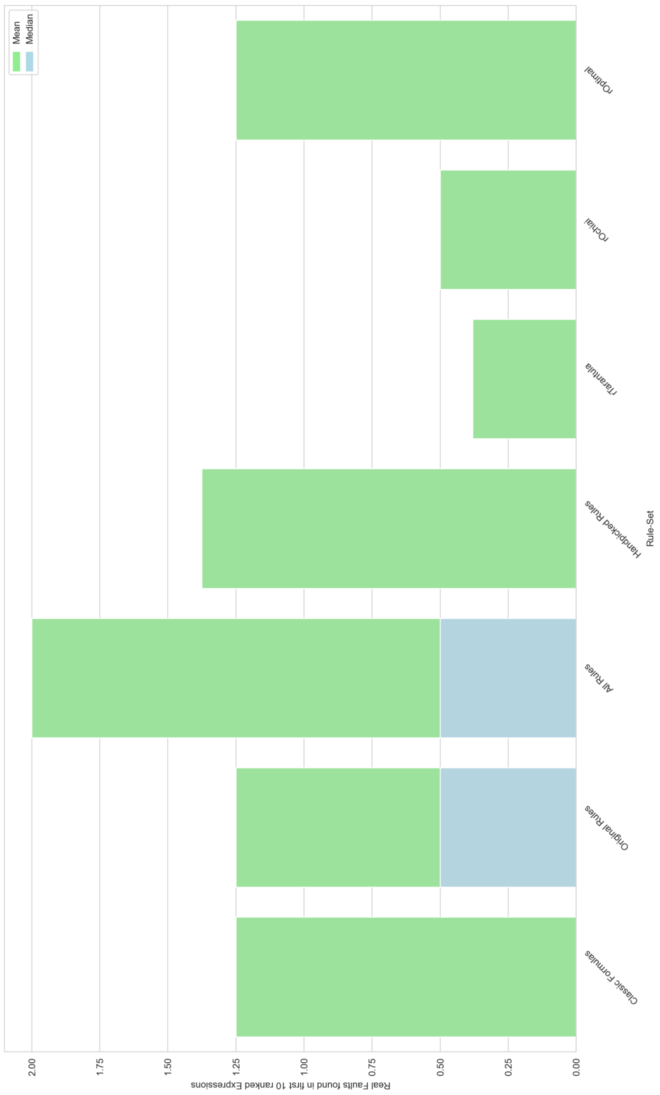


Figure 7.11: Averaged Top10-score for genetic search

Figure 7.12 shows the results when using the weights originating from the genetic search over HLS-2 using classic formulas. We observe that this configuration is well suited for a few programs and poor for others, but beats the individual formulas in mean-Top50. In general, we noticed that the *small* programs from HLS produced some of the best regressors, probably because the smaller number of entries resulted in smaller weights and less prone to overfitting.

Figure 7.13 are the results retrieved from fitting `original` rules (i.e., only rules novel from this work) on `duckling-28ddc3bf`. The resulting weights produce Top50 suggestions for almost all data points except `pandoc-6`. This model has broad generalizability across the investigated programs and is one of the drivers of the good median metrics of search-based Top50 results.

When looking for such individual results, we saw similar trends (uneven and even distributions of predictions) across all regressors, with genetic search producing the most visible trends due to the best predictions. We did not observe a specific trend for `pandoc` to infer better for `pandoc`, as most of the data points also did not share similar weights.

The best results were achieved for data points without faults executed by failing tests in which three configurations with a Top50 of 1, when fitting MLPs on `pandoc-4`, `pandoc-5` and `pandoc-3be256efb` with the original rules. Such small variations are in the realm of expected randomness and might not be worth further investigation.

RQ2.B: Applicability of Models

Classifiers performed better for scaled rules, whereas regressors had difficulties with unscaled data. Logistic regression produces high recall, while suffering from many false positives. Random forests produced a good ratio of true to false positives, but did not have a high recall. Of the regressors, only the genetic search beat the original formulas in average and median, especially performing better in Top10.

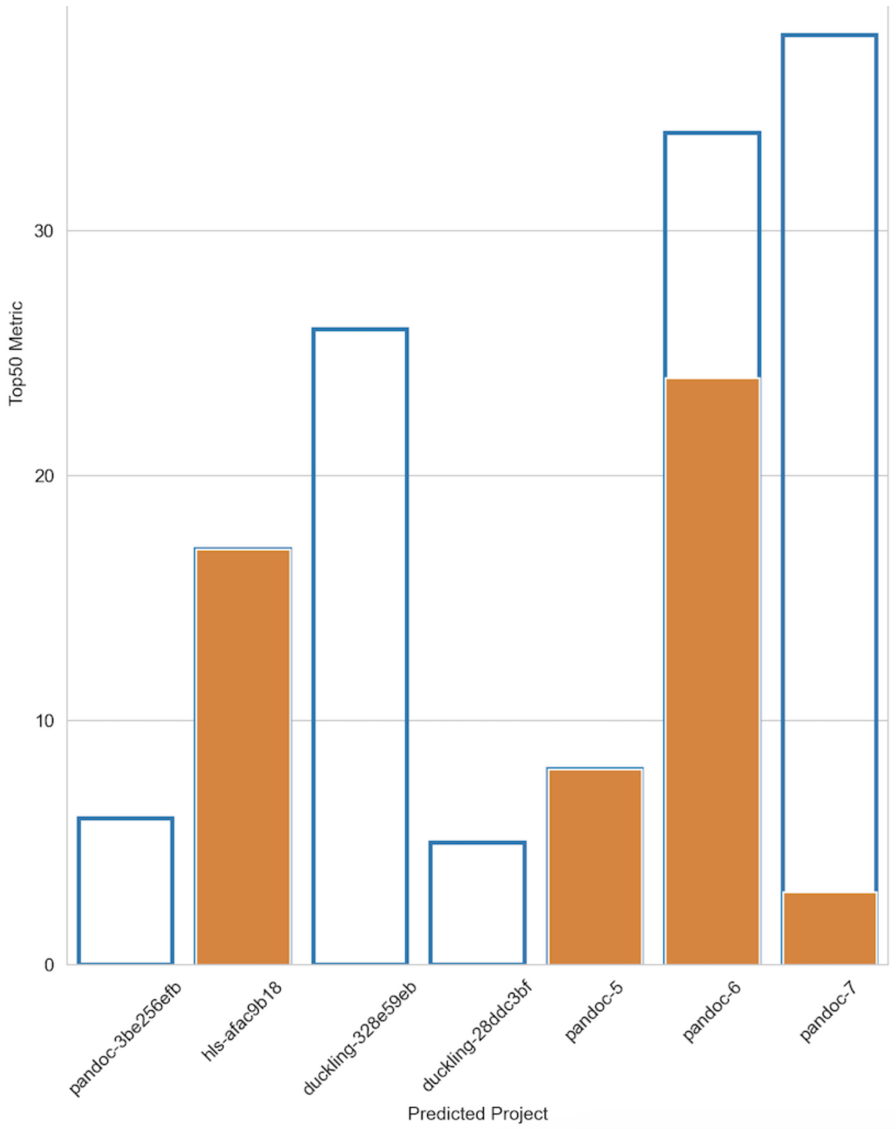


Figure 7.12: Predicting Top50 from HLS-2 with `original` rules

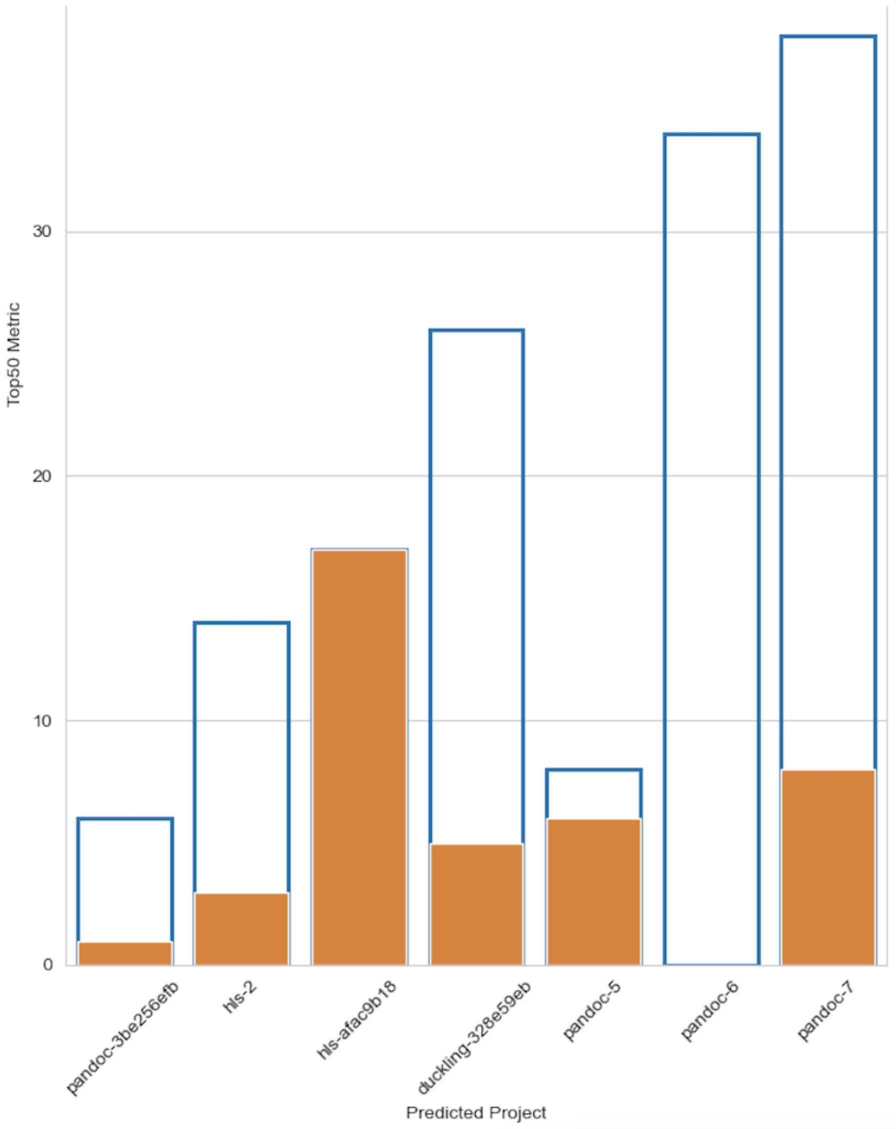


Figure 7.13: Predicting Top50 from `duckling-28ddc3bf` with `original` rules

7.5 Discussion

7.5.1 Quality of formulas

In general, the existing SBFL formulas performed well to the point that they might be used in development. The achieved rankings beat some of the existing research in Java, when the difference in granularity is taken into account: most research focuses on statements or blocks, but even the expression level seems reasonable. As formulas do not require training data and are easily applicable, they form attractive targets for Haskell tooling, e.g., suggesting points of interest on a failing PR or highlighting code of failing test runs. It might be possible to adapt existing formulas to Haskell by introducing the frequency of executions instead of binary coverage through tests. Another way to get a better result is the reduction of a spectrum, which might be achieved through filtering for AST properties or types.

However, the results of this work also show that there are unique problems with programs whose faults are not (directly) executed by failing tests. For such programs and maybe other tasks (defect prediction, test generation), novel rules based on types or AST structure can prove successful. With functional programming often used for domain-specific languages or code generation, we expect faults of this kind to be more prominent than in imperative programs.

7.5.2 Project & Test Structure

One recurring consideration throughout all results was the strong dependency on the project structure and especially on the tests.

Duckling's approach of unifying tests into a corpus of examples makes it easy for contributors and allows for a smoother execution against the generated code, while posing significant challenges for fault localization. Similarly, many contributors (or users) to Pandoc report bugs by providing examples of failing documents that are translated into a system-level regression test. This is very economical for the maintainers, but our results show that pandoc programs with unit-level tests (`pandoc-6` & `pandoc-7`) were the most approachable for all algorithms and formulas.

On the other hand, the HLS data points make use of a great degree of modularity. This is already visible, with both data points being plugins to larger systems. This separation already leads to drastically smaller spectra, and even more complicated issues (`hls-afac9b18` deleting lines on usage with other plugins) were translatable into side-effect-free unit tests. We understand that not every project can be modular to this extent, but, especially given the size, number of contributors, and changes in Pandoc and Duckling, fault localization can pay off [330].

Closing our thoughts, we would like to stress that functional programming is precisely the domain where excellent modularity can be achieved. The greater the modularity, the greater the applicability of tooling such as SBFL. For projects that have a suitable test suite, even simple SBFL formulas have immediate payoff.

7.5.3 Moving Forward on SBFL Models

The models investigated in the experiments had a variety of interesting properties, particularly their ability to produce results for data points that were not touched by failing tests. This is a unique quality and may be worth advancing in light of the fact that such faults have not been dominant throughout the fault localization literature. In most previous research, the existence of failing tests for a faulty statement has been a part of the hypothesis. Common meta-studies on errors do not mention faults of this kind [331, 332], the closest work stems from Lucia et al. [333] *"Are Faults Localizable?"*, which investigates that faults might be fixed at many places, and thus previous localizations based on patched locations might need to be revised.

From the experiments, the most promising approach is ensemble-style classifiers. Especially random forests and genetic search were able to combine the benefits of formulas while considering types and other novel rules. Given the single program fit, there is much room for improvement by combining the spectra of multiple programs to fit an ensemble.

As there is no clear trend for `pandoc` models to generalize better for `pandoc`, we think the most important attributes are determined per fault, and a training set should consist of various unique faults. A handcrafted a rule system might also be possible, as the *handpicked* rule-set emerged as a good candidate during the experiments.

7.5.4 Future Work

IDE integration We have already mentioned HLS and, in fact, used it as one of our data points. One future path would be to look at the integration of spectrum-based fault localization into IDE tools such as HLS, enabling users to get more out of their test suite than just a pass/fail. This would have to be balanced out with time constraints, since some test suites take a long time to run. However, this could be overcome by running the test suite in the background if the developer has the resources to spare. An experiment could show user needs when engaging with such tooling and measure effectivity (e.g., time improvement when using the tool to solve a set of tasks).

Innocence One way to extend this work is to introduce the notion of *innocence*. Here, we focus on the suspiciousness of a given statement, but in a typed setting, we can *verify* certain functions. This could involve functions that are *verified* using tools such as SmallCheck, where we test every possible invocation of a function of type, e.g. `Bool -> a` by applying it to both `True` and `False` and checking that the output is correct. These functions and their locations could be marked as *innocent* where we say that we *trust* this function and its associated locations.

Innocence can also span other concepts of the program, e.g., users can provide types they consider innocent (if they stem from a library), or whole modules can be declared innocent.

Automatic Program Repair One of the main challenges in automatic program repair is to locate faults to determine where to repair the program [334, 335]. In PropR, the authors use a naive fault localization algorithm that does not rank the locations, only filtering out those locations that were not involved in a failing test [335]. With a spectrum-based algorithm, repair efforts can focus on the expressions considered most suspicious, which will speed up repair/search for patches.

Error-based Fault Localization Using HPC instrumentation, we count how many times a location *started* being evaluated. However, in the presence of errors, not all evaluations will succeed. In a recent paper [336], it was shown that HPC can be extended to also record when a location finished being evaluated. By augmenting the spectra with this information, we could add a rule that tracks the difference between how often a location started being evaluated and how often it finished being evaluated. This could help localize error-based faults.

7.6 Conclusion

This paper aims to extend spectrum-based fault localization for Haskell and evaluate its applicability to real-world faults. To achieve this, we implemented a Tasty ingredient that allows the generation of spectra with expression-level granularity, including additional information on types and identifiers. Making use of the richer information, we implemented *rules* that capture the complexity of types, AST structure, or identifiers and applied them to a total of 11 real-world programs. We used the rules to investigate the attributes of spectra and to fit classifiers and regressors. Our exploration uncovered unique kinds of failures: faults that were not covered by failing tests. These failures structured the results into two groups: for most programs, the faults were covered by tests, and existing SBFL formulas performed well and were only outperformed by regression models that also make use of formulas as features. For the faults not touched by failing tests, models based on additional information (e.g., types or identifiers) were necessary to produce any correct prediction. However, these faults remain a challenging case and require further investigation. The contributions of this work hopefully open up a broader discussion of the applicability of SBFL for Haskell. The easy adoption through a plugin allows developers and researchers to experiment and provide information on user needs alongside a greater variety of projects. Further insights in addition to our initial investigation might also form a solid basis for new Haskell-specialized formulas. Especially, the novel type of failures requires an approximation that is not directly based on test failures, but exploits the project structure and types.

Why you should care about SBFL One of the big selling point of Haskell is the strong type systems and the resulting compiler feedback. But even with strong types, errors can occur (see figure 7.1) and require testing. While the compiler assists the program, tools assist the programmer. Especially within the boundaries of a strong type system in a lazy language, the rich information of types and the lack of side effects allow for better localization than imperative languages could dream of. All efforts, whether from developers, fault-localization tools, tests, or compilers, can go hand in hand to provide the best program quality with the least effort. Thanks to the ongoing efforts of the Haskell Language Server project, it is high time to introduce new software tooling for Haskell. We hope that the insights provided by our work will provide guidance when designing these tools.

8 Conclusion

This dissertation can be roughly divided in two parts. The first is testing large language models for their use in software engineering, Both will be granted a short resume and outlook, followed by a sketch how they can grow closer together.

On the robustness of code models The first topic this thesis studies in Chapter 2 is investigating the robustness of machine learning models trained for software engineering tasks. For that purpose, various models were investigated for their behavior with noisy data, and the deltas in their prediction-metric were compared. Chapter 3 improved the test-data generation, to make the process *smoother*, and bring it closer to existing work on adversarial example generation. In general, the goals were achieved, but it opened a discussion on *realism* of the transformation and the resulting files. It is debatable, but to me test-data does not need to be realistic - quite the opposite. Tests often explore the limitations of a tool and so does research. This year, a TU Delft colleague hosts a research project on *Efficiency in Compiler Architecture* [1], which investigates performance behavior of compilers. One of the target research questions is to measure the compilers runtime when facing ever larger tokens which become obviously unrealistic. This type of research helps to formulate good tests, because for a good test we also have to consider the *worst* case and sometimes *nonsense*.

Apart from these fundamental debates, CodeBERT has already been surpassed by general purpose language models. It is very likely that the next generation of tools will utilize models other than single-target models, and thus evaluating against a single metric might be incompatible. Looking back, the research around Lampion still holds some value, as it approached models as a blackbox (so the change of model is possible) and the produced artifacts form test-data, which is reusable. Thus, while the results might not be directly applicable, many of the concepts in the approach remain valuable. In fact, some ideas have been picked up by later research, as shown by Yang et al. [337] in a meta study as *Semantic Preserving Transformations*.

There is a solid amount of future research to do, but in my opinion the most valuable for better LLMs is an adaptation of the approach for training-data generation. Generating se-

[1]https://projectforum.tudelft.nl/course_editions/102/generic_projects/5152

mantically identical, but noisy data can help to address overfitting and account for (slightly) noisy input. Its not unreasonable that many people come up with very different variable names, and models should be indifferent to naming for the sake of their performance anyway. Also, semantically correct derivatives could help to bridge many issues around mining licensed code from open source repositories: There have been many (ethical) issues raised with the use of code without permission in models of profit-driven companies, but derived code could form an acceptable middle ground and provide anonymity.

Another relevant piece of work is the orchestration and testing of large language models. With the rapid progress of ChatGPT and the like, we can expect that the task of code generation will progress as well. To borrow an example from other domains, speech recognition is considered a relatively solved problem (at least for English), yet how to use it and how to make a good user experience is a completely different story. We might face similar problems in the near future, that generative models produce sufficient code, but only when they are fed with precise prompts for well-defined requirements. This is something that we can only fix with users - as we need to accommodate for their needs, skills and interests. I opened this dissertation by stating that I like programming, so I would not like an AI to take the fun parts away. It's up to us - software engineering researchers - to preserve the creative elements and automate the more mechanical parts.

Tools in Haskell, Haskell in Tools The second part of this work centered around Haskell and the possibilities to make tools utilizing types and other unique language features. In Germany we have a derogatory saying *"Alter Wein in neuen Schläuchen"* (old wine in new hoses), but contrary to proverbial wisdom many of the existing software engineering approaches were readily transferable. In Chapter 6 we re-implemented genetic program repair in the fashion of GenProg [85], with the big adaptation that our code-replacements were sourced from the compiler using typed holes [61]. Chapter 5 shows a way to enrich HPC and capture recent evaluations, which can help to reproduce and understand behavior before a crash. In Chapter 7 we transfer a *best-off* of existing spectrum based fault localization approaches and add some type-specific metrics, that enabled us to crack some of the toughest nuts we found in Chapter 4.

When using similar metrics, the Haskell tools are roughly on-par with their Java counterparts in performance. Thus, they can form a helpful addition for Haskell developers, at least if they are willing to implement the tooling. We have also demonstrated that using the strong compiler does not only open up unique solutions, but benefits the existing approaches as well: Within PropR, the typed holes automatically excluded any fix that would not compile, which results in a drastic reduction in search space. Basing the work around the compiler and core libraries also allows for a clear compatibility - if your code compiles with a (tool-supported) GHC version, the tool is also applicable.

With the initial success of software engineering approaches from Java in Haskell, a simple re-implementation and adjustment of existing approaches forms fruitful future work. I'd like to challenge this approach: Despite their academic success, program repair is not widely adopted, and neither are software spectra. Yes, they are applicable to real world programs (i.e. Defects4J), but they were not applied by the original developers. The interest remains mostly academic, and is applied to programs years after the bugs were solved. There are some applications, e.g. Metas SapFix [338], but such tailored solutions are un-achievable for most companies and users. My suggestion for future work on tooling in Haskell should focus on the users and what they want solved. To elaborate, we can look back at the work with evaluation from Chapter 5: The presented extended error message were one example, showing an offset in evaluation and call stack. Such an offset is common for lazy evaluation and might be a problem for real-world engineers. But it could be that this is not an actual problem for the developers, or that existing stack traces are sufficient. As we laid out in the future work section, there are multiple avenues where this can be useful: Either some of the existing error messages can be enhanced, but maybe it serves best in a classroom to explain laziness to students. The tools and their output should be adjusted to its final audience - anything else might remain as unused as automatic program repair. I am certain that any success with actual Haskell users will also lead to academic success.

Hand in Hand - New World Tools Both streams of research feed into improving and testing the tools available to developers, especially in the face of errors and hardship. The progress in LLMs allows creativity to consider a new generation of programming languages that for example utilize prompting, while being otherwise not human readable. Despite being an appealing futurism, this is likely not the *next* generation of tools. But it is not impossible that modern tools will change the field as much as calculators changed mathematics. If that is the case, we have to change how we engage with our tools:

In a mathematics class, we still learn the basics, even if we use calculators and solvers for real applications and problems. A key element in applied mathematics is now to understand the issue at hand, find the right model and determine correct variables. A powerful large language model, that is able to generate code and tests alike, might reduce many tasks in programming to requirements engineering and verifying tests.

We should also consider how we shape the process of building our models. There are indications that ever larger corpora improve code-generation, yet they also need more energy, more space and of course more data. Here we should keep in mind a center-piece of programming: Code is more than just text. Even if we want to build a novel AI tool, we already have very mature tools at our disposal (compilers, fault localization, versioning control, and many more) that can help us get where we want to go quicker.

It is concerning that in the AI rush we left many best-practices behind, and even derailed to producing a lot of code that could never compile nor run. The research community should reflect deeply about how this could happen, to not regress in other fields too.

With joined efforts, we can build mature tools that hone already available approaches. Steps towards this are done in this work by utilizing properties for a stronger test suite and compiler-provided (and thus valid) changes in PropR. Other emerging research also follow this venue, e.g. Ye et. al. [37] who utilize the compiler-feedback as a *post-condition* in program repair, or Zhang et al. [339] who connect LLMs with information retrieval and static code analysis. Such *pre-* and *post-conditions* around LLMs are a promising way to utilize the generative capacities in a more ordered fashion, yet there is still a lot of work to be done. Not only are there technical questions surrounding these tools (as touched in the work with Lampion), but we also have to introduce expertise from other fields like requirements engineering, security and ethics. The best time for this would have been yesterday - the second best is now.

Acknowledgments

I came across many people that I am grateful for, that changed me and hopefully I was able to change them. I have yet to see if my PhD is a change for the better, or just a change.

Some people stood out as examples - of a life full of (successful) competition and a strive for greatness. I hope that one day you meet someone that shows you your worth without comparison, and that you come to peace with who you are. I am sorry for my shortcoming to not be this person for you.

Apart from being dramatic, I also want to thank some of my friends individually:
I thank **Alireza** for being a great friend and I am very grateful for the time we spent biking, thinking about our futures - hopefully we meet again to bike and talk about our past.
I have to thank **Hanifa** for her splendid chatter, banter and the constant drama that she amused me with. I have not met many people whose live is like a movie, but I am happy to be a sidekick in the soap-opera of your life.
A special thank goes to **Matthí**. He is mentioned a bit in this work, but he has been more than just a co-author. Without him, most of my PhD would have been a very lonely journey, but thanks to him I always had a great partner in crime. In Germany we joke TEAM is an acronym: *"Toll, ein anderer machts"*.
Of course also **Benny**, who is not a certified psychologist yet, but I am looking forward to see how much you can help people, given how much help you were to me already. Your perspective on my daily affairs was invaluable, and I hope I can return the favor.
For **Annibale**, I wish you the very best. I have made many of the steps thanks to you, from the first research questions to the defense. It would have not been the same with any other supervisor. I hope you see the difficulties we had like your weights in the gym - as an opportunity to grow.
I need to thank the people at the pottery studio for making one of my favorite past-time a welcome place that I will remember for the rest of my life. Ellen for being a (sometimes) rough, but infinitely wise teacher, Liz & Lisanne for being always at my side at the wheel and Sara for cheering me up among the many mistakes it takes. Oh, and Tessa. I don't know what you are doing exactly, but I like it.

Lastly, I want to thank **Palina** for her love, compassion and the home she made for me in the last year. I am looking forward for the time to come with you.

Yours sincerely

Titles in the IPA Dissertation Series since 2021

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

Bibliography

References

[1] Zhenjiang Hu, John Hughes, and Meng Wang. How functional Programming mattered. *National Science Review*, 2(3):349–370, 2015.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large Language Models trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] John McCarthy. History of LISP. In *History of programming languages*, pages 173–185. 1978.

[4] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of mathematics*, 33(2):346–366, 1932.

[5] Paul Hudak. Conception, Evolution, and Application of functional Programming Languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[6] Ana Bove, Peter Dybjer, and Ulf Norell. A brief Overview of Agda–a functional Language with dependent Types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 73–78. Springer, 2009.

[7] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial. *Rapport Technique*, 178, 1997.

[8] Nicholas D Matsakis and Felix S Klock. The rust Language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[9] Eduardo Rosales, Andrea Rosà, Matteo Basso, Alex Villazón, Adriana Orellana, Ángel Zenteno, Jhon Rivero, and Walter Binder. Characterizing Java Streams in the Wild. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 143–152. IEEE, 2022.

[10] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling Object, Relations and XML in the .net Framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[11] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[12] John Hughes. Why functional Programming matters. *The computer journal*, 32(2):98–107, 1989.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. 2004.

[14] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima Inc, 2008.

[15] Peter Sestoft. *Programming Language Concepts.* Springer, 2017.

[16] Victor Pankratius, Felix Schmidt, and Gilda Garretón. Combining functional and imperative Programming for multicore Software: An empirical Study evaluating Scala and Java. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 123–133. IEEE, 2012.

[17] Christof Ebert and Panos Louridas. Generative AI for Software Practitioners. *IEEE Software*, 40(4):30–38, 2023.

[18] Michel Wermelinger. Using GitHub Copilot to solve simple Programming Problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 172–178, 2023.

[19] Nhan Nguyen and Sarah Nadi. An empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

[20] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving ChatGPT Prompt for Code Generation. *arXiv preprint arXiv:2305.08360*, 2023.

[21] Fardin Ahsan Sakib, Saadat Hasan Khan, and AHM Karim. Extending the Frontier of ChatGPT: Code Generation and Debugging. *arXiv preprint arXiv:2307.08260*, 2023.

[22] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. Repoagent: An llm-powered open-source Framework for Repository-level Code Documentation Generation. *arXiv preprint arXiv:2402.16667*, 2024.

[23] Robert W McGee. ChatGPT and Copyright Infringement: An Exploratory Study. *Available at SSRN 4578430*, 2023.

[24] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your Code generated by ChatGPT really correct? Rigorous Evaluation of large Language Models for Code Generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[25] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. How Secure is Code Generated by ChatGPT? In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2445–2451. IEEE, 2023.

[26] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the Robustness of Code Generation Techniques: An empirical Study on Github Copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2149–2160. IEEE, 2023.

[27] David Lo. Trustworthy and Synergistic Artificial Intelligence for Software Engineering: Vision and Roadmaps. *arXiv preprint arXiv:2309.04142*, 2023.

[28] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad Smells: A Review of current Knowledge. *Journal of Software Maintenance and Evolution: research and practice*, 23(3):179–202, 2011.

[29] Kamil Jezek and Richard Lipka. Antipatterns causing Memory Bloat: A Case Study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 306–315. IEEE, 2017.

[30] Philip Wadler. The Essence of functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

[31] Philip Wadler. Monads for functional Programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, pages 24–52. Springer, 1995.

[32] Simon Marlow et al. Haskell 2010 Language Report. 2010.

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

[34] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.

[35] Jeremy Gibbons, David Lester, and Richard Bird. Functional Pearl: Enumerating the Rationals. *Journal of Functional Programming*, 16(3):281–291, 2006.

[36] Martin Hofmann and Martin Hofmann. Syntax and Semantics of dependent Types. *Extensional Constructs in Intensional Type Theory*, pages 13–54, 1997.

[37] He Ye, Matias Martinez, and Martin Monperrus. Neural Program Repair with execution-based Backpropagation. In *Proceedings of the 44th international conference on software engineering*, pages 1506–1518, 2022.

[38] Jack Herrington. *Code Generation in Action.* Manning Publications Co., 2003.

[39] Ivan Perez and Henrik Nilsson. Testing and Debugging functional reactive Programming. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–27, 2017.

[40] John Hughes. QuickCheck Testing for Fun and Profit. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. Springer, 2007.

[41] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy Smallcheck: Automatic exhaustive Testing for small Values. *Acm sigplan notices*, 44(2):37–48, 2008.

[42] Volker Stolz and Frank Huch. Runtime Verification of concurrent Haskell Programs. *Electronic Notes in Theoretical Computer Science*, 113:201–216, 2005.

[43] Simon Thompson. *Haskell: The Craft of functional Programming*. Addison-Wesley, 2011.

[44] Zhanyong Wan and Paul Hudak. Functional reactive Programming from first Principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, 2000.

[45] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on reactive Programming. *ACM Computing Surveys (CSUR)*, 45(4):1–34, 2013.

[46] Artemij Fedosejev. *React.js Essentials*. Packt Publishing Ltd, 2015.

[47] Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava: Creating asynchronous, event-based Applications*. " O'Reilly Media, Inc.", 2016.

[48] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. Evaluating the Code Quality of AI-assisted Code Generation Tools: An empirical Study on Github Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778*, 2023.

[49] Oscar Oviedo-Trespalacios, Amy E Peden, Thomas Cole-Hunter, Arianna Costantini, Milad Haghani, JE Rod, Sage Kelly, Helma Torkamaan, Amina Tariq, James David Albert Newton, et al. The Risks of using ChatGPT to obtain common safety-related Information and Advice. *Safety science*, 167:106244, 2023.

[50] Florian Arendt, Benedikt Till, Martin Voracek, Stefanie Kirchner, Gernot Sonneck, Brigitte Naderer, Paul Pürcher, and Thomas Niederkrotenthaler. ChatGPT, Artificial Intelligence, and Suicide Prevention, 2023.

[51] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. "Do anything now": Characterizing and Evaluating in-the-Wild Jailbreak Prompts on large Language Models. *arXiv preprint arXiv:2308.03825*, 2023.

[52] Yueqi Xie, Jingwei Yi, Jiawei Shao, Justin Curl, Lingjuan Lyu, Qifeng Chen, Xing Xie, and Fangzhao Wu. Defending ChatGPT against Jailbreak Attack via Self-Reminders. *Nature Machine Intelligence*, 5(12):1486–1496, 2023.

[53] Yi Liu, Gelei Deng, Zhengzi Xu, Yuekang Li, Yaowen Zheng, Ying Zhang, Lida Zhao, Tianwei Zhang, and Yang Liu. Jailbreaking ChatGPT via Prompt Engineering: An empirical Study. *arXiv preprint arXiv:2305.13860*, 2023.

[54] Kiho Lee. ChatGPT_DAN, February 2023.

[55] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165, 2014.

[56] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Adobe Press, 2006.

[57] Takaeshi Chusho. Coverage Measure for Path Testing based on the Concept of essential Branches. *Journal of Information Processing*, 6(4):199–205, 1983.

[58] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A Database of existing Faults to enable controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[59] June Sallou, Thomas Durieux, and Annibale Panichella. Breaking the Silence: The Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 102–106, New York, NY, USA, 2024. Association for Computing Machinery.

[60] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[61] Matthías Páll Gissurarson. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 179–185, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Matias Martinez and Martin Monperrus. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*, 2016.

[63] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 432–449, 1997.

[64] Jelber Sayyad Shirabad, Timothy C Lethbridge, and Stan Matwin. Mining the Maintenance History of a legacy Software System. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 95–104. IEEE, 2003.

[65] Ahmed E Hassan. The Road ahead for Mining Software Repositories. In *2008 frontiers of software maintenance*, pages 48–57. IEEE, 2008.

[66] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python Framework for Mining Software Repositories. In *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 908–911, 2018.

[67] CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence. 2020.

[68] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet Challenge: Evaluating the State of semantic Code Search. *arXiv preprint arXiv:1909.09436*, 2019.

[69] Will Partain. The NoFib Benchmark Suite of Haskell Programs. In *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*, pages 195–202, Ayr, Scotland, 1993. Springer, Springer.

[70] Josep Silva. The Buggy Benchmarks Collection. 2007. Josep Silva self-published on his website / university.

[71] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS quarterly*, pages 75–105, 2004.

[72] Samuel Sanford Shapiro and Martin B Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.

[73] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the Naturalness of Software. *Communications of the ACM*, 59(5):122–131, 2016.

[74] Frank Wilcoxon. Individual Comparisons by ranking Methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[75] Fortunato Pesarin and Luigi Salmaso. *Permutation Tests for complex Data: Theory, Applications and Software.* John Wiley & Sons, 2010.

[76] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[77] Romi Satria Wahono. A systematic Literature Review of Software Defect Prediction. *Journal of software engineering*, 1(1):1–16, 2015.

[78] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. Progress on Approaches to Software Defect Prediction. *Iet Software*, 12(3):161–175, 2018.

[79] André Riboira and Rui Abreu. The GZoltar Project: A graphical Debugger Interface. In *International Academic and Industrial Conference on Practice and Research Techniques*, pages 215–218. Springer, 2010.

[80] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A Method for automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[81] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: A Method for automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[82] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. Re-evaluating the Role of BLEU in Machine Translation Research. In *11th conference of the european chapter of the association for computational linguistics*, pages 249–256, 2006.

[83] Ehud Reiter. A structured Review of the Validity of BLEU. *Computational Linguistics*, 44(3):393–401, 2018.

[84] Matt Post. A Call for Clarity in Reporting BLEU Scores. *arXiv preprint arXiv:1804.08771*, 2018.

[85] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic Method for automatic Software Repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[86] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of automated Program Repair: Fixing 55 out of 105 Bugs for 8 Dollar each. In *2012 34th international conference on software engineering (ICSE)*, pages 3–13. IEEE, 2012.

[87] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140, 2018.

[88] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[89] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic Repair of real Bugs in Java: A large-scale Experiment on the Defects4J Dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[90] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.

[91] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated Survey of Methodologies for automated Software Test Case Generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[92] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: A Survey. *Cybersecurity*, 1(1):1–13, 2018.

[93] Saad Shafiq, Atif Mashkoor, Christoph Mayr-Dorn, and Alexander Egyed. Machine Learning for Software Engineering: A Systematic Mapping. *arXiv preprint arXiv:2005.13299*, 2020.

[94] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained Model for Programming and natural Languages. *arXiv preprint arXiv:2002.08155*, 2020.

[95] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing Source Code using a neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[96] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. *DeepCommenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information*, page 1571–1575. Association for Computing Machinery, New York, NY, USA, 2020.

[97] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering*, 2020.

[98] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[99] Christian Murphy, Gail Kaiser, Lifeng Hu, and Leon Wu. Properties of machine learning applications for use in metamorphic testing. *20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008*, pages 867–872, 2008.

[100] Xiaoyuan Xie, Joshua W.K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating Machine Learning Classifiers by Metamorphic Testing. *Journal of Systems and Software*, 84(4):544 – 558, 2011. The Ninth International Conference on Quality Software.

[101] Muhammad Sharif, Sajjad Mohsin, Muhammad Younas Javed, and Muhammad Atif Ali. Single Image Face Recognition Using Laplacian of Gaussian and Discrete Cosine Transforms. *Int. Arab J. Inf. Technol.*, 9(6):562–570, 2012.

[102] Leonhard Applis, Annibale Panichella, and Arie van Deursen. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1377–1381. IEEE, 2021.

[103] Ehsan Mashhadi and Hadi Hemmati. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. *arXiv preprint arXiv:2103.11626*, 2021.

[104] Cong Pan, Minyan Lu, and Biao Xu. An Empirical Study on Software Defect Prediction Using CodeBERT Model. *Applied Sciences*, 11(11):4793, 2021.

[105] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical Investigation into learning Bug-fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837, 2018.

[106] Tae-Hwan Jung. CommitBERT: Commit Message Generation Using Pre-Trained Programming Language Model. *arXiv preprint arXiv:2105.14242*, 2021.

[107] Rafael Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big Code != Big Vocabulary: Open-Vocabulary Models for Source code. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20. ACM, 2020.

[108] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic Testing: A new Approach for generating next Test Cases. Technical Report Tech. Rep. HKUST-CS98-01, 1998.

[109] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. Towards a metamorphic Testing Methodology for service-oriented Software Applications. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 470–476. IEEE, 2005.

[110] Johannes Mayer and Ralph Guderlei. On random Testing of Image Processing Applications. In *2006 Sixth International Conference on Quality Software (QSIC'06)*, pages 85–92. IEEE, 2006.

[111] Ralph Guderlei and Johannes Mayer. Towards automatic Testing of imaging Software by Means of random and metamorphic Testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(06):757–781, 2007.

[112] TH Tse and Stephen S Yau. Testing context-sensitive middleware-based Software Applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 458–466. IEEE, 2004.

[113] KY Sim, WKS Pao, and C Lin. Metamorphic Testing using geometric Interrogation Technique and its Application. *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 1(2):91–95, 2005.

[114] Junhua Ding, Tong Wu, Dianxiang Wu, Jun Q Lu, and Xin-Hua Hu. Metamorphic Testing of a Monte Carlo modeling Program. In *Proceedings of the 6th international workshop on automation of software test*, pages 1–7, 2011.

[115] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative Approach for testing Bioinformatics Programs using metamorphic Testing. *BMC bioinformatics*, 10(1):1–12, 2009.

[116] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic Testing Approach for Compilers based on metamorphic Testing Technique. In *2010 Asia Pacific Software Engineering Conference*, pages 270–279. IEEE, 2010.

[117] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

[118] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A Survey on metamorphic Testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[119] Keith Cooper and Linda Torczon. *Engineering a Compiler.* Elsevier, 2011.

[120] C. N. Vasconcelos, A. Paes, and A. Montenegro. Towards Deep Learning Invariant Pedestrian Detection by Data Enrichment. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 837–841, 2016.

[121] Md Rafiqul Islam Rabin, Ke Wang, and Mohammad Amin Alipour. Testing Neural Programs. *CoRR*, 2019.

[122] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the Generalizability of Neural Program Models with Respect to semantic-preserving Program Transformations. *Information and Software Technology*, 135:106552, 2021.

[123] Karl Popper. *The Logic of scientific Discovery.* Routledge, 2005.

[124] Peter Achinstein. *The Book of Evidence.* Oxford University Press, 2001.

[125] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial Examples for Models of Code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[126] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding Java Classes with Code2vec. *Proceedings of the 17th International Conference on Mining Software Repositories*, Jun 2020.

[127] A. Van Deursen and T. Kuipers. Building Documentation Generators. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 40–49, 1999.

[128] Paul W. McBurney and Collin McMillan. Automatic Documentation Generation via Source Code Summarization of Method Context. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 279–290, New York, NY, USA, 2014. Association for Computing Machinery.

[129] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pundefined-sundefinedreanu. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 226–237, New York, NY, USA, 2008. Association for Computing Machinery.

[130] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. Code2seq: Generating Sequences from structured Representations of Code. *arXiv preprint arXiv:1808.01400*, 2018.

[131] Lionel C Briand. Software Documentation: How much is enough? In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 13–15. IEEE, 2003.

[132] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software Documentation Issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[133] Horacio Saggion, Dragomir Radev, Simone Teufel, and Wai Lam. Meta-Evaluation of Summaries in a cross-lingual Environment using content-based Metrics. In *COLING 2002: The 19th International Conference on Computational Linguistics*, 2002.

[134] Chin-Yew Lin. Rouge: A Package for automatic Evaluation of Summaries. In *Text Summarization Branches out*, pages 74–81, 2004.

[135] Vincent J. Hellendoorn and Premkumar Devanbu. Are Deep Neural Networks the Best Choice for Modeling Source Code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763–773, New York, NY, USA, 2017. Association for Computing Machinery.

[136] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[137] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[138] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent Code Completion with Bayesian Networks. *ACM Trans. Softw. Eng. Methodol.*, 25(1), dec 2015.

[139] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from Examples to improve Code Completion Systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[140] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive Source Code Completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 69–79. IEEE, 2012.

[141] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735, 2019.

[142] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token Code Completion by jointly learning from Structure and Naming Sequences. In *Proceedings of the 44th International Conference on Software Engineering*, pages 401–412, 2022.

[143] Milton Friedman. The Use of Ranks to avoid the Assumption of Normality implicit in the Analysis of Variance. *Journal of the american statistical association*, 32(200):675–701, 1937.

[144] Peter Nemenyi. Distribution-free mulitple Comparisons, PhD thesis Princeton University Princeton, 1963.

[145] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic Model for Code with Decision Trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

[146] Zhenpeng Chen, Jie M Zhang, Max Hort, Federica Sarro, and Mark Harman. Fairness Testing: A Comprehensive Survey and Analysis of Trends. *arXiv preprint arXiv:2207.10223*, 2022.

[147] Giles Hooker. *Diagnostics and Extrapolation in Machine Learning*. stanford university, 2004.

[148] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.

[149] James Zou and Londa Schiebinger. AI can be sexist and racist — it's Time to make it Fair, 2018.

[150] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding Java Classes with Code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 243–253, 2020.

[151] Khlood Ahmad, Muneera Bano, Mohamed Abdelrazek, Chetan Arora, and John Grundy. What's up with Requirements Engineering for Artificial Intelligence Systems? In *2021 IEEE 29th International Requirements Engineering Conference (RE)*, pages 1–12. IEEE, 2021.

[152] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[153] Karl Meinke and Amel Bennaceur. Machine Learning for Software Engineering: Models, Methods, and Applications. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 548–549, 2018.

[154] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. Counterfactual Explanations for Models of Code. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 125–134, 2022.

[155] Phil McMinn. Search-based Software Test Data Generation: A Survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[156] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic Test Suite Generation for object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[157] Paolo Tonella. Evolutionary Testing of Classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[158] Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[159] Shin Yoo and Mark Harman. Pareto efficient multi-objective Test Case Selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150, 2007.

[160] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sérgio Lopes de Souza. A systematic Review on search based Mutation Testing. *Information and Software Technology*, 81:19–35, 2017.

[161] Hélène Waeselynck, Pascale Thévenod-Fosse, and Olfa Abdellatif-Kaddour. Simulated Annealing applied to Test Generation: Landscape Characterization and Stopping Criteria. *Empirical Software Engineering*, 12(1):35–63, 2007.

[162] Matias Martinez and Martin Monperrus. Astor: A Program Repair Library for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, 2016.

[163] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical Evaluation of evolutionary Algorithms for Unit Test Suite Generation. *Information and Software Technology*, 104:207–235, 2018.

[164] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. A large scale empirical Comparison of State-of-the-Art search-based Test Case Generators. *Information and Software Technology*, 104:236–256, 2018.

[165] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. Single and Multi-objective Test Cases Prioritization for Self-driving Cars in Virtual Environments. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2022.

[166] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. SBST Tool Competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27. IEEE, 2021.

[167] John H Holland. *Adaptation in natural and artificial Systems: An introductory Analysis with Applications to Biology, Control, and artificial Intelligence*. MIT press, 1992.

[168] Stefan Sette and Luc Boullart. Genetic Programming: Principles and Applications. *Engineering applications of artificial intelligence*, 14(6):727–736, 2001.

[169] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating Branch Coverage as a many-objective Optimization Problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[170] Gordon Fraser and Andrea Arcuri. Evosuite: On the Challenges of Test Case Generation in the real World. In *2013 IEEE sixth international conference on software testing, verification and validation*, pages 362–369. IEEE, 2013.

[171] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated Test Case Generation as a many-objective Optimisation Problem with dynamic Selection of the Targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[172] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The Strength of random Search on automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[173] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of machine learning research*, 13(2), 2012.

[174] Andrea Arcuri and Lionel Briand. A Practical Guide for using statistical Tests to assess randomized Algorithms in Software Engineering. In *Proceedings of the 33rd international conference on software engineering*, pages 1–10, 2011.

[175] Andrea Arcuri and Lionel Briand. A Hitchhiker's Guide to statistical Tests for assessing randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[176] William Jay Conover. *Practical nonparametric Statistics*, volume 350. john wiley & sons, 1999.

[177] András Vargha and Harold D Delaney. A Critique and Improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[178] Mark D Smucker, James Allan, and Ben Carterette. A Comparison of statistical Significance Tests for Information Retrieval Evaluation. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 623–632, 2007.

[179] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Automated Repair of Feature Interaction Failures in Automated Driving Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 88–100, New York, NY, USA, 2020. Association for Computing Machinery.

[180] Tinkle Chugh, Karthik Sindhya, Jussi Hakanen, and Kaisa Miettinen. A Survey on Handling computationally expensive multiobjective Optimization Problems with evolutionary Algorithms. *Soft Computing*, 23, 2019.

[181] Helen G Cobb and John J Grefenstette. Genetic Algorithms for Tracking changing Environments. Technical report, Naval Research Lab Washington DC, 1993.

[182] Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446, 2024.

[183] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. Multiple Fault Localization of Software Programs: A Systematic Literature Review. *Information and Software Technology*, 124:106312, 2020.

[184] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical Review of Java Program Repair Tools: A large-scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 302–313, 2019.

[185] Gabin An, Juyeon Yoon, and Shin Yoo. Searching for Multi-fault Programs in Defects4J. In Una-May O'Reilly and Xavier Devroey, editors, *Search-Based Software Engineering*, pages 153–158, Cham, 2021. Springer International Publishing.

[186] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a Dataset of reproducible Failures and Fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.

[187] Rafael-Michael Karampatsis and Charles Sutton. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 573–577, New York, NY, USA, 2020. Association for Computing Machinery.

[188] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do Changes induce Fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[189] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. Reproducibility and Credibility in empirical Software Engineering: A Case Study based on a Systematic Literature Review of the Use of the SZZ Algorithm. *Information and Software Technology*, 99:164–176, 2018.

[190] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. Problems with SZZ and Features: An empirical Study of the State of Practice of Defect Prediction Data Collection. *Empirical Software Engineering*, 27(2):1–49, 2022.

[191] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial Evaluation of Unit Test Generation: Finding real Faults in a financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, 2017.

[192] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing Literature Study on Test Amplification. *Journal of Systems and Software*, 157:110398, 2019.

[193] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. Carving differential Unit Test Cases from System Test Cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–264, 2006.

[194] Balázs Mosolygó, Norbert Vándor, Gábor Antal, and Péter Hegedűs. On the Rise and Fall of simple Stupid Bugs: A Life-Cycle Analysis of SSTUBS. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 495–499. IEEE, 2021.

[195] Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. A lightweight interactive Debugger for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 13–24, 2007.

[196] Andy Gill and Colin Runciman. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, page 1–12, New York, NY, USA, 2007. Association for Computing Machinery.

[197] GHC Contributors. GHC 8.10.4 users guide, 2021.

[198] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What Makes a Good Bug Report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery.

[199] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 177–210, New York, NY, USA, 2019. Association for Computing Machinery.

[200] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. Do Developers read Compiler Error Messages? In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 575–585, Buenos Aires, Argentina, 2017. IEEE, IEEE/ACM.

[201] Peter C. Rigby and Suzanne Thompson. Study of Novice Programmers Using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '05, page 105–109, New York, NY, USA, 2005. Association for Computing Machinery.

[202] John Wrenn and Shriram Krishnamurthi. Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 134–147, New York, NY, USA, 2017. Association for Computing Machinery.

[203] V. Javier Traver. On Compiler Error Messages: What They Say and What They Mean. *Adv. in Hum.-Comp. Int.*, 2010, jan 2010.

[204] Dale Shaffer, Wendy Doube, and Juhani Tuovinen. Applying Cognitive Load Theory to Computer Science Education. In *PPIG*, volume 1, pages 333–346, Keele, UK, 2003. Citeseer, M. Petre & D. Budgen (Eds.).

[205] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. Metacognitive Difficulties faced by vovice Programmers in automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 41–50, Espoo, Finland, 2018. ACM.

[206] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 74–82, Tacoma, WA, USA, 2017. ACM.

[207] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do Stack Traces help Developers fix Bugs? In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*, pages 118–121, Cape Town, South Africa, 2010. IEEE, IEEE.

[208] Tristan O.R. Allwood, Simon Peyton Jones, and Susan Eisenbach. Finding the Needle: Stack Traces for GHC. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, page 129–140, New York, NY, USA, 2009. Association for Computing Machinery.

[209] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-View Tracing for Haskell: a New Hat. In Ralf Hinze, editor, *2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).

[210] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for Tracing. In *Symposium on Implementation and Application of Functional Languages*, pages 165–181, Madrid, Spain, 2002. Springer, Springer.

[211] Andy Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electron. Notes Theor. Comput. Sci.*, 41(1):1, 2000.

[212] Henrik Nilsson and Jan Sparud. The Evaluation Dependence Tree as a Basis for lazy functional Debugging. *Automated software engineering*, 4:121–150, 1997.

[213] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative Debugger. *Australian Computer Science Communications*, 18:401–408, 1996.

[214] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.

[215] Ranjit Jhala. LiquidHaskell is a GHC Plugin, 2020.

[216] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 411–424, New York, NY, USA, 2019. Association for Computing Machinery.

[217] Maarten Faddegon and Olaf Chitil. Algorithmic Debugging of real-world Haskell Programs: Deriving Dependencies from the Cost Centre Stack. *ACM SIGPLAN Notices*, 50(6):33–42, 2015.

[218] John Launchbury. A natural Semantics for lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, Charleston, SC, USA, 1993. ACM.

[219] Peter Sestoft. Deriving a lazy abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[220] Paul Anderson, Łucja Kot, Neil Gilmore, and David Vitek. SARIF-Enabled Tooling to Encourage Gradual Technical Debt Reduction. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 71–72, Montreal, QC, Canada, 2019. IEEE/ACM.

[221] A. Hamou-Lhadj and T. Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190, Athens, Greece, 2006. IEEE.

[222] Kunihiro Noda, Takashi Kobayashi, Tatsuya Toda, and Noritoshi Atsumi. Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 13–22, Turin, Italy, 2017. IEEE.

[223] Yen-Chi Chen. A Tutorial on Kernel Density Estimation and recent Advances. *Biostatistics & Epidemiology*, 1(1):161–187, 2017.

[224] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. Capturing Propagation of Infected Program States. ESEC/FSE '09, page 43–52, New York, NY, USA, 2009. Association for Computing Machinery.

[225] Marco Vassena, Joachim Breitner, and Alejandro Russo. Securing Concurrent Lazy Programs Against Information Leakage. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 37–52, Santa Barbara, CA, USA, 2017. IEEE.

[226] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. Towards a context-aware IDE-based meta search Engine for Recommendation about Programming Errors and Exceptions. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 194–203, Antwerp, Belgium, 2014. IEEE.

[227] Kasra Ferdowsi. Towards Human-Centered Types & Type Debugging. Plateau Workshop.

[228] Leonhard Applis and Annibale Panichella. HasBugs - Handpicked Haskell Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 223–227, Melbourne, Australia, 2023. IEEE/ACM.

[229] Mark Weiser. Program Slicing. *IEEE Transactions on software engineering*, 1(4):352–357, 1984.

[230] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, Windsor, UK, 2007. IEEE, IEEE.

[231] Tom Janssen, Rui Abreu, and Arjan JC Van Gemund. Zoltar: A Toolset for automatic Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664, Auckland, New Zealand, 2009. IEEE, IEEE.

[232] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189, 2013.

[233] Qi Xin. Towards Addressing the Patch Overfitting Problem. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 489–490, 2017.

[234] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. An Investigation of Compression Techniques to speed up Mutation Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 274–284. IEEE, 2018.

[235] Zhen Yu Ding. Patch Quality and Diversity of Invariant-Guided Search-Based Program Repair. *arXiv preprint arXiv:2003.11667*, 2020.

[236] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 492–495, New York, NY, USA, 2014. Association for Computing Machinery.

[237] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. Experience Report: How do Techniques, Programs, and Tests impact automated Program Repair? In

2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pages 194–204. IEEE, 2015.

[238] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic Neural Program Embedding for Program Repair. *arXiv preprint arXiv:1711.07163*, 2017.

[239] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation Models using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[240] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. On the Use of Dependabot Security Pull Requests. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 254–265. IEEE, 2021.

[241] Khashayar Etemadi, Nicolas Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *arXiv preprint arXiv:2103.12033*, 2021.

[242] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[243] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. *arXiv preprint arXiv:1703.00198*, 2017.

[244] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[245] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP ’00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[246] Richard Hamlet. Random Testing. *Encyclopedia of software Engineering*, 2:971–978, 1994.

[247] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current Challenges in automatic Software Repair. *Software Quality Journal*, 21(3):421–443, 2013.

[248] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 27th ACM Joint European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), 2019.

[249] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive Study of automatic Program Repair on the QuixBugs Benchmark. *Journal of Systems and Software*, 171:110825, 2021.

[250] Chadi Trad, Rawad Abou Assi, Wes Masri, and Fadi Zaraket. CFAAR: Control Flow Alteration to Assist Repair. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 208–215. IEEE, 2018.

[251] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. An empirical Analysis of the Influence of Fault Space on search-based automated Program Repair. *arXiv preprint arXiv:1707.05172*, 2017.

[252] Qi Xin and Steven P Reiss. Leveraging syntax-related Code for automated Program Repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670. IEEE, 2017.

[253] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a Program Repair Bot? Insights from the Repairnator Project. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 95–104. IEEE, 2018.

[254] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented Program Repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE, 2017.

[255] Christoph Kern and Mark R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, apr 1999.

[256] Catherine A Meadows. Formal Verification of cryptographic Protocols: A Survey. In *International Conference on the Theory and Application of Cryptology*, pages 133–150. Springer, 1994.

[257] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. PLAS '16, page 91–96, New York, NY, USA, 2016. Association for Computing Machinery.

[258] Christoph Kreitz. Program Synthesis. In *Automated Deduction—A Basis for Applications*, pages 105–134. Springer, 1998.

[259] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 313–326, New York, NY, USA, 2010. Association for Computing Machinery.

[260] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-Based Program Repair Using SAT. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[261] Ricardo Peña. An Introduction to Liquid Haskell. *arXiv preprint arXiv:1701.03320*, 2017.

[262] Patrick Redmond, Gan Shen, and Lindsey Kuper. Toward Hole-Driven Development with Liquid Haskell. *arXiv preprint arXiv:2110.04461*, 2021.

[263] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. 52(10):63–74, sep 2017.

[264] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[265] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[266] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 522–538, New York, NY, USA, 2016. Association for Computing Machinery.

[267] Susumu Katayama. MagicHaskeller: System Demonstration. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*, page 63, 2011.

[268] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid Information Flow Control. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.

[269] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[270] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[271] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 691–701, New York, NY, USA, 2016. Association for Computing Machinery.

[272] Edward Kmett. The lens Library, 2021.

[273] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 48–61, New York, NY, USA, 2005. Association for Computing Machinery.

[274] Chang Wook Ahn and R.S. Ramakrishna. Elitism-based compact genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 7(4):367–385, 2003.

[275] Geoffrey Neumann, Mark Harman, and Simon Poulding. Transformed Vargha-Delaney Effect Size. In Márcio Barros and Yvan Labiche, editors, *Search-Based Software Engineering*, pages 318–324, Cham, 2015. Springer International Publishing.

[276] Rainer Koschke. Survey of Research on Software Clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[277] Michel Raymond and Francois Rousset. An Exact Test for Population Differentiation. *Evolution*, 49(6):1280–1283, 1995.

[278] Thorsten Pohlert. The pairwise multiple Comparison of mean Ranks Package (PMCMR). *R package*, 27(2019):9, 2014.

[279] Qi Xin and Steven P. Reiss. Identifying Test-Suite-overfitted Patches through Test Case Generation. *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236, 2017.

[280] Amirfarhad Nilizadeh, Gary T. Leavens, Xuan-Bach D. Le, Corina S. Păsăreanu, and David R. Cok. Exploring True Test Overfitting in Dynamic Automated Program Repair using Formal Methods. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 229–240, 2021.

[281] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via multi-objective genetic Programming. *arXiv*, 46(10):1040–1067, 2017.

[282] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[283] Ali Ghanbari and Lingming Zhang. PraPR: Practical Program Repair via Bytecode Mutation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1118–1121, 2019.

[284] Thomas Durieux and Martin Monperrus. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, page 85–91, New York, NY, USA, 2016. Association for Computing Machinery.

[285] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified Pre-training for Program Understanding and Generation, 2021.

[286] Andrea Arcuri and Gordon Fraser. On Parameter Tuning in Search Based Software Engineering. In *International Symposium on Search Based Software Engineering*, pages 33–47. Springer, 2011.

[287] Thomas Durieux and Martin Monperrus. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Technical report, Universite Lille 1, 2016.

[288] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass Benchmarks for automated Repair of C Programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[289] Claire Le Goues, Yuriy Brun, Stephanie Forrest, and Westley Weimer. Clarifications on the Construction and Use of the ManyBugs Benchmark. *IEEE Transactions on Software Engineering*, 43(11):1089–1090, 2017.

[290] Massimiliano Dominici. An Overview of Pandoc. *TUGboat*, 35(1):44–50, 2014.

[291] Feng Li, Meng Wang, and Dan Hao. Bridging the Gap between Different Programming Paradigms in Coverage-based Fault Localization. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pages 75–84, 2022.

[292] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible Debugging Software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229, 2013.

[293] Ruanqianqian Lisa Huang, Elizaveta Pertseva, Michael Coblenz, and Sorin Lerner. How do Haskell Programmers debug? Plateau Workshop.

[294] Justin Lubin and Sarah E Chasins. How statically-typed functional Programmers write Code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[295] James A Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[296] Chris Parnin and Alessandro Orso. Are automated Debugging Techniques actually helping Programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[297] Qusay Idrees Sarhan and Árpád Beszédes. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access*, 10:10618–10639, 2022.

[298] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE ’02, page 467–477, New York, NY, USA, 2002. Association for Computing Machinery.

[299] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The DStar Method for effective Software Fault Localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[300] David Lo, Lingxiao Jiang, Aditya Budi, et al. Comprehensive Evaluation of Association Measures for Fault Localization. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[301] Zhenyu Zhang, Wing Kwong Chan, TH Tse, Bo Jiang, and Xinming Wang. Capturing Propagation of infected Program States. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 43–52, 2009.

[302] Mike Papadakis and Yves Le Traon. Effective Fault Localization via Mutation Analysis: A selective Mutation Approach. In *Proceedings of the 29th annual ACM symposium on applied computing*, pages 1293–1300, 2014.

[303] Mike Papadakis and Yves Le Traon. Metallaxis-FL: Mutation-based Fault Localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[304] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the Mutants: Mutating faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[305] Luciano C Ascari, Lucilia Y Araki, Aurora RT Pozo, and Silvia R Vergilio. Exploring Machine Learning Techniques for Fault Localization. In *2009 10th Latin American Test Workshop*, pages 1–6. IEEE, 2009.

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

[307] Meng Gao, Pengyu Li, Congcong Chen, and Yunsong Jiang. Research on Software multiple Fault Localization Method based on Machine Learning. In *MATEC web of conferences*, volume 232, page 01060. EDP Sciences, 2018.

[308] Alberto Gonzalez-Sanchez, Eric Piel, Hans-Gerhard Gross, and Arjan JC van Gemund. Prioritizing Tests for Software Fault Localization. In *2010 10th International Conference on Quality Software*, pages 42–51. IEEE, 2010.

[309] Gong Dandan, Wang Tiantian, Su Xiaohong, and Ma Peijun. A Test-suite Reduction Approach to improving Fault-Localization Effectiveness. *Computer Languages, Systems & Structures*, 39(3):95–108, 2013.

[310] László Vidács, Árpád Beszédes, Dávid Tengeri, István Siket, and Tibor Gyimóthy. Test Suite Reduction for Fault Detection and Localization: A combined Approach. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 204–213. IEEE, 2014.

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

[312] Anish Tondwalkar, Rolph Recto, Westley Weimer, and Ranjit Jhala. Finding and Fixing Bugs in Liquid Haskell. 2016.

[313] Vanessa Vasconcelos and Mariza AS Bigonha. HaskellFL: A Tool for Detecting Logical Errors in Haskell. *International Journal of Computer and Systems Engineering*, 15(8):479–493, 2021.

[314] Rafael Caballero, Adrián Riesco, and Josep Silva. A Survey of algorithmic Debugging. *ACM Computing Surveys (CSUR)*, 50(4):1–35, 2017.

[315] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow- and Control-flow-based Test Adequacy Criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994.

[316] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating Bug Detection Tools. In *Workshop on the evaluation of software defect detection tools*, volume 5. Chicago, Illinois, 2005.

[317] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC Counting Standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

[318] Ronald Fagin, Ravi Kumar, and Dakshinamurthi Sivakumar. Comparing top k Lists. *SIAM Journal on discrete mathematics*, 17(1):134–160, 2003.

[319] Manos Renieres and Steven P Reiss. Fault Localization with nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[320] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[321] J. Blank and K. Deb. pymoo: Multi-Objective Optimization in Python. *IEEE Access*, 8:89497–89509, 2020.

[322] Michael D McKay, Richard J Beckman, and William J Conover. A Comparison of three Methods for selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 42(1):55–61, 2000.

[323] Brad L Miller, David E Goldberg, et al. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex systems*, 9(3):193–212, 1995.

[324] K Deb and RB Agrawal. Simulated binary Crossover for continuous Search Space. *Complex systems*, 9(2):115–148, 1995.

[325] Kalyanmoy Deb. *Multi-objective Optimization using evolutionary Algorithms*, volume 16. John Wiley & Sons, 2001.

[326] Gordon Fraser and Andrea Arcuri. A large-scale Evaluation of automated Unit Test Generation using Evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–42, 2014.

[327] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated Unit Test Generation really help Software Testers? A controlled empirical Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):1–49, 2015.

[328] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated Unit Tests find real Faults? An empirical Study of Effectiveness and Challenges. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[329] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. SOBER: Statistical model-based Bug Localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[330] Tung Dao, Na Meng, and ThanhVu Nguyen. Triggering Modes in Spectrum-Based Multi-location Fault Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1774–1785, New York, NY, USA, 2023. Association for Computing Machinery.

[331] Dewayne E Perry and Carol S Stieg. Software Faults in Evolving a large, real-time System: A Case Study. In *European software engineering conference*, pages 48–67. Springer, 1993.

[332] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an Understanding of Bug Fix Patterns. *Empirical Software Engineering*, 14:286–315, 2009.

[333] Lucia LUCIA, Ferdian Thung, David Lo, and Lingxiao Jiang. Are Faults Localizable? 2012.

[334] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated Program Repair. *Communications of the ACM*, 62(12):56–65, 2019.

[335] Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. PropR: Property-Based Automatic Program Repair. In *The 44th IEEE/ACM In-ternational Conference on Software Engineering (ICSE)*, Pittsburgh, PA, USA, 2022. IEEE/ACM, IEEE/ACM.

[336] Matthías Páll Gissurarson and Leonhard Applis. CSI: Haskell - Tracing Lazy Evaluations in a Functional Language. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages (IFL '23)*. ACM New York, NY, USA, 2023.

[337] Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. Robustness, Security, Privacy, Explainability, Efficiency, and Usability of large Language Models for Code. *arXiv preprint arXiv:2403.07506*, 2024.

[338] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: Automated end-to-end Repair at Scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.

[339] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. *arXiv preprint arXiv:2404.05427*, 2024.