Thesis for The Degree of Doctor of Philosophy

Synthesis and Repair for Functional Programming:
A Type- and Test-Driven Approach

Matthías Páll Gissurarson

CHALMERS

Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2024

Synthesis and Repair for Functional Programming:
A Type- and Test-Driven Approach

Matthías Páll Gissurarson

ISBN 978-91-8103-079-2
Doktorsavhandlingar vid Chalmers tekniska högskola.
Ny serie ISSN 0346-718X.
ISSN 0346-718X

Technical Report No 5537

Department of Computer Science & Engineering
Division of Computing Science
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2024.

“Deyr fé,
deyja frændur,
deyr sjálfu ið sama
en orðstír deyr aldregi
hveim er sér góðan getur.”
- Hávamál.

Abstract

Modern programs in languages like Haskell include a lot of information beyond what is required for compilation. This includes unit tests, property-based tests, and type annotations more specific than those necessary to resolve ambiguity. This additional specification is usually only used for post-compilation verification by running the tests to verify that the code-as-written matches the specification the types and properties provide.

In this thesis, we explore ways of going beyond verification, and how this additional information can aid the developer during development. This can be done in multiple ways, for example, by helping the programmer write an implementation that matches the specification, by helping them track down the source of a bug in the implementation, and automatically repairing an implementation that does not match the specification.

In the first part, I explore the integration of program synthesis into GHC compiler error messages using typed-hole suggestions to aid completion of partial programs during development. In the second part, we present PropR, an automatic repair tool. PropR is based on type-driven synthesis, guided by property-based testing and fault localization in conjunction with genetic algorithms. A rich specification is required for these approaches to be effective. This motivates the third part of this thesis, where we present Spectacular, a specification synthesis tool. Spectacular uses ECTA-based synthesis to automatically infer properties of programs, letting us bootstrap specifications from previous versions. In the fourth and fifth part of this thesis, we present the lightweight trace-based and spectrum-based fault localization tools CSI: Haskell and TastySpectrum respectively, and explore how we can localize program faults and find likely sources of a bug.

Keywords: Compilers, Types, Tests, Program Synthesis, Program Repair

Acknowledgments

My sincerest thanks to my supervisor, David Sands, for getting me through some very tough times, and getting this WASP project where I could continue working on typed-holes and synthesis despite earlier setbacks in my PhD.

Special thanks to my co-supervisors, Alejandro Russo and Martin Monperrus, for their input on my work and many good times at various conferences and visits! And thanks to my examiner John Hughes for setting the bar high, pushing me to do even better.

And thanks to my PhD committee for their feedback on this thesis! Especially Nadia Polikarpova, whose opposition as my licentiate opponent drove me to a higher evaluation standard.

I want to thank the PhD crew for a lot of fun times; the old crew (Nachi, Agustín, Carlos, Alejandro, Elisabet, Abhiroop, Jeremy, Irene, Fabian, Mohammad, Ivan, Benjamin), as well as the new one (Hanna, Robert, Prabhat, Henrik, Lorenzo, Piero, Victor, Luque, Wincent, Katya).

Huge thanks to my co-author of many papers Leonhard Applis, without whom this thesis would be a lot shorter. Turns out bringing functional programming to software engineers and software engineering to functional programmers is in hot demand!

Thanks to Matthew Sottile for many Zoom calls and a lot of advice throughout. Many times, I've been stumped by something but helpfully pointed in the right direction by him over Zoom. Thanks!

Throughout my PhD, I've been quite active on Twitter (now X). Thanks to all my mutuals and their advice throughout the years! I've also been active in the Haskell community. I want to especially thank the GHC team. None of this this would have been possible without their work throughout the years.

To my friends and family back in Iceland, for all their support in getting me to where I am today: Mamma, Pabbi, Elísabet, Amma og Afi heitinn, Björn, Konni, Hákon, Óli, Styrmir, Birkir, Magnús Karl, Gabríela, Ragnheiður, Þeódóra, Gunnlaugur, Haukur, Bjarki, og Magnús Pálsson. Takk fyrir mig! Án ykkar hefði ég aldrei nennt þessu.

Last but not least, I want to thank my wife **Anandi** and the Rajans for their love and support over the years!

This research was supported by a grant from the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation.

Contents

1

Introduction

1.1 Motivation and Overview

Motivation

When developers write programs, they do so with specific goals in mind and some idea of how to achieve these goals. Traditionally, they communicate these goals and ideas to the computer using text in the form of source code, including the term–level implementation, and, in languages like Haskell, a type–level specification that lives alongside the implementation. They often model the intended behavior by providing type annotations alongside their functions and variables, as well as a suite of tests that demonstrates the intended runtime behavior of the program. The packaged source code and tests taken as a whole then provide much more information to the compiler than merely the implementation. Half of the time spent programming is spent on debugging [6], which means that developers are working on almost complete programs. As the programs are almost complete, there are usually some tests available (at least for the bug that is being fixed), and the types involved have stabilized. This places a lot of constraints on the possible valid implementations of the program, which we can use to synthesize fixes to suggest to the developer and guide them towards a correct solution. With a sufficiently rich specification, we can even automatically repair an incorrect implementation.

However, in modern programming practice, this information is used in a yes–or–no manner: Does the program type check? Does it pass the test suite? In this thesis, I show how to go beyond the yes–or–no use case and make better use of the information already present in the source package for:

- Program synthesis using valid hole–fits in GHC,
- Automatic program repair with PropR,
- Test suite bootstrapping and discovery using Spectacular,
- Trace–based fault localization using CSI: Haskell, and finally
- Spectrum–based fault localization using TastySpectrum.

Overview

Program synthesis is very computationally heavy, making it intractable to synthesize large programs. A more focused approach is required. But how should we direct that focus?

Typed–Holes In this thesis, we make heavy use of *typed–holes*. The programmer specifies a typed–hole during development, usually using an underscore (_), as seen in figure 1.1.

```
minimumOrBound :: [Int] -> Int
minimumOrBound [] = _
minimumOrBound (x:xs) = min x (minimumOrBound xs)
```

Figure 1.1: An example of a program with a hole in it.

These typed–holes allow us to focus our synthesis efforts on that particular part of the program to fill the hole. By integrating with the compiler (GHC) and its constraint–based type checker, we can come up with (*synthesize*) identifiers and expressions such that we can replace the hole and the resulting program would be valid, i.e., *well–typed*. An example can be seen in figure 1.2. Synthesizing these valid–hole fits for typed–holes is explored in–depth in the first paper of this thesis [11].

```
• Found hole: _ :: Int
• In an equation for ‘minimumOrBound’: minimumOrBound [] = _
• Relevant bindings include
    minimum :: [Int] -> Int (bound at f.hs:4:1)
  Valid hole fits include
    maxBound :: forall a. Bounded a => a
      with maxBound @Int
      (imported from ‘Prelude’ at f.hs:1:1-31
       (and originally defined in ‘GHC.Enum’))
    minBound :: forall a. Bounded a => a
      with minBound @Int
      (imported from ‘Prelude’ at f.hs:1:1-31
       (and originally defined in ‘GHC.Enum’))
```

Figure 1.2: A program with a hole in it and the GHC error.

Automatic Program Repair It is not enough for a program to be merely type–correct. As we see in figure 1.3, even if the types are correct, the program can still be wrong. In more advanced type systems, like that of Agda, correctness can be fully specified in the types. However, in weaker type systems like that of Haskell, we have to resort to runtime verification in the form of a test suite.

```haskell
minimumOrBound :: [Int] -> Int
minimumOrBound [] = minBound -- BUG: Should be maxBound
minimumOrBound (x:xs) = min x (minimumOrBound xs)
```

Figure 1.3: An incorrect implementation

By adding a test suite, the programmer sees that the program is not correct: it always returns `minBound`!

```haskell
prop_unit :: Bool
prop_unit = minimumOrBound [2,1,3] == 1

prop_is_min :: [Int] -> Bool
prop_is_min xs = let m = minimumOrBound xs
                 in null (filter (< m) xs)
```

Figure 1.4: A minimal test suite for our example

The programmer could, of course, realize their mistake and change `minBound` to `maxBound`, which is the right solution and passes the test suite.

The next step is to automate this process: instead of having the programmer manually choose which parts of the program to focus the synthesis and repair on, we can run the tests and pick likely candidates to target.

By replacing parts of the code with holes, synthesizing valid–hole fits for those holes, and re–running the test suite to see if we are closer to a solution, we should eventually be able to repair any program. This can scale to programs that require multiple fixes, by using genetic algorithms to combine multiple solutions that partially repair a program into one solution that passes the entire test suite. This approach is explored in detail in [17].

```
minimumOrBound :: [Int] -> Int
minimumOrBound [] = maxBound
minimumOrBound (x:xs) = min x (minimumOrBound xs)
```

Figure 1.5: The correct implementation

Synthesizing Specifications This approach relies on us being able to synthesize good hole–fit candidates and, moreover, the availability of a good test suite. This is not always the case, especially with older programs. In the third paper [18] in this thesis, we explore how we can use a recent synthesis technique based on equality–constrained tree–automata (ECTAs) to efficiently synthesize multi–term Haskell expressions directly from the types as seen by the compiler, and how we can synthesize a specification using equivalence classes for large programs.

Fault Localization Being able to synthesize good candidates is not enough; we have to be able to effectively determine where to target our synthesis. The next two papers in this thesis [15, 16] focus on improving fault localization for functional programs. In the fourth paper, CSI: Haskell, we extend the compiler to add low–level tracing of Haskell programs, and to capture the suffix of that trace. In the case of infinite loops or errors, the suffix of the trace allows us to determine which parts were *recently evaluated* and which parts were evaluated earlier and less likely to be the cause of the error or loop. Similarly, when the bug is caused by invalid data being consumed, the fact that they are likely to be recently evaluated in a lazy language like Haskell allows us to more quickly localize the fault. By focusing on recent locations, we could speed up program repair considerably.

The fifth paper on functional spectrums implements spectrum collection and spectrum–based fault localization for the popular Haskell testing framework Tasty. A spectrum is essentially a matrix of tests, the locations each of them touches and whether they failed or not. Plugging these into various

formulas, we can quantify the *suspiciousness* of each location, which indicates how strongly we believe it to be the cause of the fault. In the future work section, we describe how we can combine the previous work into an improved version of PropR.

1.2 Background and Related Work

For a better understanding of the work in this thesis and its context, we must elaborate on the components involved and the related work in the field. Specifically, we:

- Introduce the Haskell programming language and the Glasgow Haskell Compiler (GHC) with which our explorations have been conducted,
- give a brief overview of program synthesis and the specific techniques we use to synthesize fixes,
- explain property–based testing that allows us to verify our synthesis,
- have look at automatic program repair and genetic programming that allows us to scale program repair beyond single fixes,
- Examine the equality–constrained tree automata (ECTAs) that allow us to efficiently synthesize multi–term Haskell expressions, and finally,
- fault localization using spectrum–based methods and program tracing.

1.2.1 Haskell

Our explorations are conducted in the functional programming language Haskell, which sports a strong type system with rich type–inference and non–strict evaluation by default. This means that analysis can often be done on an expression–by–expression basis, without having to consider side effects. It also allows us to trace programs and closely observe the data flow. The strong type system and type–inference means that the information that the user provides can be further extrapolated, and the popular property–based testing framework QuickCheck (see section 1.2.6) pushes this even further, allowing users to write *properties* that are extrapolated into tests that cover many more cases than a comparable number of unit tests.

```
Prelude> (_ "hello, world") :: [String]
<interactive>:1:2: error:
  • Found hole: _ :: [Char] -> [String]
  • In the expression: _
    In the expression: (_ "hello, world") :: [String]
    In an equation for ‘it’: it = (_ "hello, world") :: [String]
  • Relevant bindings include
      it :: [String] (bound at <interactive>:1:1)
```

Figure 1.6: An example of a typed–hole error message in GHCi 8.10.6.

1.2.2 Glasgow Haskell Compiler (GHC)

The Glasgow Haskell Compiler (GHC) is a state–of–the–art, industrial–strength compiler for Haskell, widely used in academia and industry. GHC has a few features that are particularly relevant to our exploration:

- GHC has support for typed–holes (see section 1.2.3), which we can use to direct our efforts and query the compiler for relevant information,
- GHC has a compiler plugin infrastructure that allows you to intervene at certain stages of compilation (such as after desugaring, or during type checking) and inject your own behavior, making it particularly suitable for experimentation, as you can modify parts of the compilation pipeline without digging into the compiler's internals, and
- GHC is easy to extend, as I did with my initial valid hole–fit suggestions (presented in the first paper of this thesis [11]) and the subsequent hole–fit plugins (see section 1.2.3). These were initially implemented by me as a compiler fork and eventually integrated into the official compiler release versions 8.6 and 8.10, respectively.
- GHC has built–in program coverage (HPC) that allows us to instrument any library or package to collect traces and spectrums.

1.2.3 Typed–Holes

A typed–hole is a *hole* in the context of a program, with a type and its constraints inferred by the compiler as if the hole were a free variable. Inspired by a similar feature in Agda, a minimal implementation of typed–holes was initially added to GHC in version 7.8 [12]. An example of the typed–hole in `(_ "hello, world") :: [String]` can be seen in figure 1.6.

Finding Valid Hole–Fits

Valid hole–fits were inspired by typed–hole suggestions in the PureScript compiler, but a similar automatic proof–search was available earlier in Agda as the *auto* command [12].

```
Valid hole fits include
  lines :: String -> [String]
  words :: String -> [String]
  repeat :: forall a. a -> [a]
    with repeat @String
  return :: forall (m :: * -> *) a. Monad m => a -> m a
    with return @[] @String
  fail :: forall (m :: * -> *) a. MonadFail m => String -> m a
    with fail @[] @String
  pure :: forall (f :: * -> *) a. Applicative f => a -> f a
    with pure @[] @String
  (Some hole fits suppressed; ...)
```

Figure 1.7: An example of valid hole–fits in GHCi, continued from the output in figure 1.6. Presented without imports

As detailed in the first paper of this thesis and in my master's thesis [11, 12], valid hole–fits are found by constructing an appropriate equality type for each of the candidate hole–fits and invoking GHC's type checker. The candidate hole–fits are drawn from the global environment (imports, top–level functions, etc.) or the local context (such as function arguments or locally let– or where–bound variables). In the hole in figure 1.7, the type of the candidate hole–fits are, e.g., the types of the valid hole–fits, `String -> [String]` and `forall a. a -> [a]`, but also the types of other non–valid candidates, such as the type of `otherwise :: Bool`, the type of `[] :: forall a. [a]`, the type of `map :: (a -> b) -> [a] -> [b]`, etc. We feed the type checker with each of the equality types, as well as context of the hole, and any relevant constraints[1], and ask the solver to solve the equality. If a solution is possible, then there is a way to unify the type–variables in the type of the hole and the type of the candidate hole–fit so that the types match (e.g., setting `a` to `String` in `forall a. a -> [a]` to get `String -> [String]`), and the candidate hole–fit is then a valid–hole fit. An overview of the process of finding valid hole–fits is shown in figure 1.8.

[1] As an example of relevant constraints, the hole in `(show _)` will get the type `a` where `a` is an unbound type–variable and the relevant constraints is the set {`Show a`}.

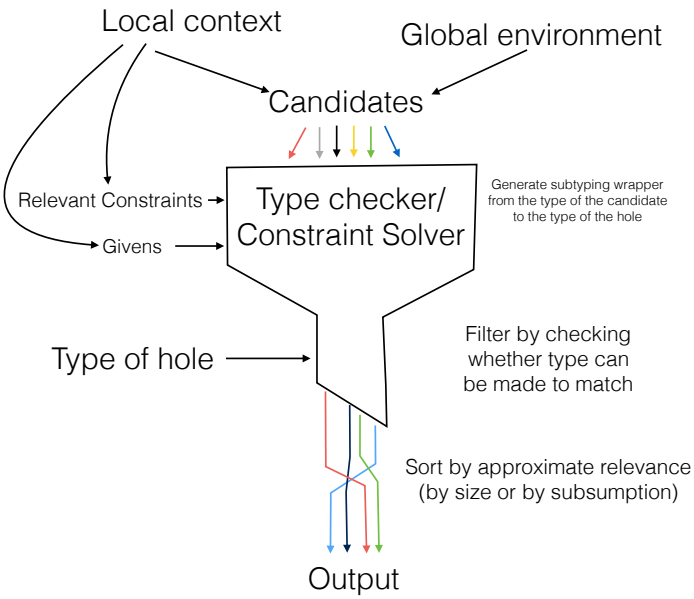


Figure 1.8: An overview of how valid hole–fit suggestions are found [12].

Refinement Hole–Fits

Of special interest are *refinement hole–fits*, an extension of valid hole–fits not found in PureScript [11]. For refinement hole–fits, we allow the candidate to have more arguments than the hole, where the number of additional arguments, n, is defined as the *refinement level*. This allows us to find fits like `foldr (_a :: Int -> Int -> Int) (_b :: Int)` for the hole `_ :: [Int] -> Int`, where `_a :: Int -> Int -> Int` and `_b :: Int` are two new holes (with the refinement level is 2). An example of refinement hole–fits for the hole in figure 1.6 can be seen in figure 1.9.

Refinement hole–fits are particularly useful for synthesis, since we can recursively fill the additional holes, allowing us to synthesize sophisticated expressions as hole–fits. Valid hole–fits and refinement hole–fits are detailed in the first paper of this thesis and in my master's thesis [11, 12].

Hole–Fit Plugins

Hole–fit plugins are an extension of GHC's plugin infrastructure that allows plugin authors to customize the behavior of valid hole–fits by manipulating what candidates get checked for validity and which of those hole–fits found to be valid are shown to users [13].

```
Valid refinement hole fits include
  iterate (_ :: String -> String)
    where iterate :: forall a. (a -> a) -> a -> [a]
    with iterate @String
  replicate (_ :: Int)
    where replicate :: forall a. Int -> a -> [a]
    with replicate @String
  mapM (_ :: Char -> [Char])
    where mapM :: forall (t :: * -> *) (m :: * -> *) a b.
                  (Traversable t, Monad m) =>
                  (a -> m b) -> t a -> m (t b)
    with mapM @[] @[] @Char @Char
  traverse (_ :: Char -> [Char])
    where traverse :: forall (t :: * -> *) (f :: * -> *) a b.
                      (Traversable t, Applicative f) =>
                      (a -> f b) -> t a -> f (t b)
    with traverse @[] @[] @Char @Char
  map (_ :: Char -> String)
    where map :: forall a b. (a -> b) -> [a] -> [b]
    with map @Char @String
  scanl (_ :: String -> Char -> String) (_ :: [Char])
    where scanl :: forall b a. (b -> a -> b) -> b -> [a] -> [b]
    with scanl @String @Char
  (Some refinement hole fits suppressed; ...)
```

Figure 1.9: An example of refinement hole–fits in GHCi, with the refinement level set to 2. Continued from the output in figure 1.7. Presented without imports.

This enables us to filter out candidates from modules and modify the order in which the fits are returned, allowing for more sophisticated heuristics. It also allows us to modify the synthesis on a per–hole basis, for instance, by writing a plugin that allows us to inject expressions mined from the context as candidate hole–fits for program repair. An overview of hole–fit–plugins is shown in figure 1.10[2].

Hole–fit plugins also provide an ideal way to inspect and integrate hole–fit–based synthesis into other tools. Using hole–fit plugins, we can extract information about the context of a given hole, such as the type, any local identifiers (such as function arguments), and global identifiers (i.e., imports, top–level bindings) available for synthesis.
In particular, since the hole–fit plugins run in the type checking phase of

[2]Presented as part of the Haskell Implementors' Workshop and the ICFP student research competition in 2019 [13].

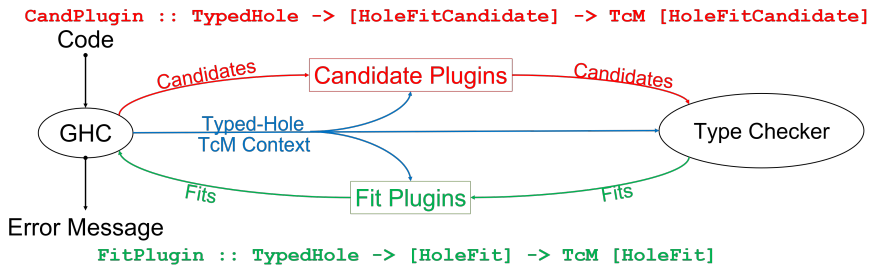


Figure 1.10: An overview of typed–hole plugins [13].

GHC's compilation pipeline, we have access to the types of these identifiers, local type–variables, implications (i.e., constraints such as `Show a =>`, and crucially, access to the GHC constraint solver. This can be very helpful for synthesis, as explored in [17, 18].

Limitations

The way typed–holes are implemented in GHC poses some limitations. Currently, they are based on the "unknown identifier" functionality and only generate an error message. However, recent efforts in, for example, Hazel have shown that by tightly integrating them into the compiler and the programming environment, they can enable a whole different style of programming [29]. We can recover some of this functionality using IDE–plugins such as the Haskell Language Server (HLS), allowing programmers to interactively choose valid hole–fits for typed–holes in their IDE, but it sorely lacks the rich contextual information present in languages such as Hazel.

1.2.4 Program Synthesis

Program synthesis is the generation of code based on a high–level specification of how that program should behave [37]. As there is an infinite number of programs, restricting the search space is key to practical program synthesis. One way to restrict the search space is to use input–output examples, such as FlashFill [19]. Using only input–output examples can be limiting, but works well when the target language is domain–specific: this shrinks the search space by reducing the possible programs that can be written in a language. Another way to efficiently synthesize programs is by focusing on parts of the program, such as in *sketching*, where users write a high–level sketch of a program but leave holes for synthesis of low–level details [37].

Type–directed synthesis is especially powerful since there are a lot more ill–typed programs than well–typed ones, and type–errors can be detected

very early [33]. How well type–directed synthesis can perform depends on the expressiveness of the type system. For example, expressive type systems such as the refinement types used in SynQuid allow developers to decorate types with predicates from a decidable logic, meaning they can more precisely specify which programs are valid, which improves the program synthesis [33]. However, more expressiveness in the type system comes at the expense of type–inference. In Haskell, the type of most programs can be inferred without the developers having to provide type annotations. Type–directed synthesis has a long history in Haskell, such as the type–based Djinn synthesizer, which can synthesize Haskell expressions based on the type [4]. More recent Haskell–based synthesizers include Hoogle+, which uses type–guided abstract refinement (TYGAR) to find programs composed from functions in Haskell libraries based on type and input–output examples [20], and Hectare, which uses an ECTA–based technique to synthesize functions from the Haskell prelude [22]. Haskell also has some integrated program synthesis–like features, such as the *deriving* mechanism that can automatically generate instances for functions like `(==)` and `show` [25]. This has later been extended with the GHC *deriving via* extension, which allows you to derive *via* other instances and gives greater control over how the resulting instance behaves [5]. However, these only work for type–class instances.

Typed–hole directed synthesis

Typed–hole directed synthesis is a combination of using contextual information as in sketching and type information to restrict the search space to only those programs that satisfy the type, such as the one used in Perelman et al. for partial expression completion in C# [32] and Myth [31] by Osera et al.

More recent work includes Smyth [24] by Lubin et al., which uses *live bidirectional evaluation* to propagate input-output examples generated from assertions to guide the search, taking it beyond the type–only directed synthesis in this work. In Smyth, they use a language that supports the *live evaluation* of code, which includes typed–holes by producing results that are either values or a "paused" expression that can resume evaluation when the necessary holes are filled, a la Omar et al. [24, 29]. A key innovation is supporting *live unevaluation*, which allows results to be checked against examples to compute constraints that, if satisfied, ensure that the result eventually produces a value that satisfies the examples [24]. By eagerly checking incomplete programs for counterexamples using constraint propagation, the synthesizer can eagerly discard programs that can never satisfy the examples [27]. A similar approach has been implemented in Scrybe by Mulleners et al., which interleaves refinements and guesses and allows arbitrary functions to

be used as refinement steps [27]. Further work by Mulleners et al. on the realizability of polymorphic programs introduces a technique to determine whether a solution to a given synthesis problem exists [28].

However, all of these synthesizers work on small examples. It is unclear if the approach scales to large code bases or can support all Haskell features. One limitation of example propagation is the exponential growth of constraint sizes [27]. The term enumeration approach taken in this work has the advantage that we can place typed–holes anywhere in a program, enumerate terms that satisfy the type, fill the holes, and use the existing GHC toolchain to efficiently recompile and check the results with respect to existing test–suites. Integration with the GHC type–checker allows us to enumerate and check all possible terms, even those using more advanced language features. This allows us to do automated program repair even on large code bases, without having to consider constraint sizes or language feature interactions.

1.2.5 Automatic Program Repair

A practical application of program synthesis is automated program repair, where we fix bugs in programs according to their specification. There are already some examples of type–directed program repair, such as Lifty, which uses the SynQuid refinement type–based technique to repair security policy violations in a domain specific language [34]. In the second paper of this thesis, we investigate the use of type–directed synthesis for automated program repair. We implemented PropR, a genetic search–based program repair tool that combines the typed–hole directed synthesis from my first paper with property–based specifications to automatically repair Haskell programs [17].

Genetic Program Repair

Genetic program repair is a successful generate–and–validate–based approach to automated program repair based on genetic search [23, 26]. The approach is exemplified by GenProg, a statement–based automatic program repair for C–programs, which uses unit tests to determine the locations of faults and the validity of fixes [23]. The quality of a fix is evaluated based on how many unit tests they pass, and two fixes are combined into a new fix by combining partial fixes into a new fix, preferring well–performing fixes to low–performing fixes [23]. For some programs, this approach can find fixes that eliminate the bug found by the tests [23]. Current state–of–the–art program repair tools, such as Astor, have been based on the same approach, but mainly target Java [26]. A genetic approach allows us to focus on find-

ing simple partial fixes and combining them, meaning we can do repair on a per–fault basis rather than having to consider the whole program.

LLM–based Program Repair

Large Language Models (LLMs) have surged in popularity with the release of GPT3 and ChatGPT in 2022. They have changed the landscape of synthesis and repair, with many tools [7, 9, 30] based on training and fine–tuning of LLMs to synthesize code. Traditional program synthesis techniques struggle with synthesizing large programs, as the search space is too big and the problem under–specified. We now have tools such as GitHub's Copilot [9], ChatGPT [30], and Codex [7], which can generate AI–powered code suggestions [10]. These tools allow programmers to generate massive amounts of code from short prompts describing what they want.

However, in this thesis, we have not used any LLM–based techniques. Had they been available in 2018, we certainly would have explored LLMs for valid hole–fits, and indeed, a neural network–based approach was suggested when the original typed–hole paper was presented. Training a neural network on available Haskell code could have been an avenue for ranking valid–hole fits, so that the more relevant suggestion would have been listed earlier than less relevant ones. However, LLM and neural network–based approaches have their own challenges. One big challenge is one of distribution and usability: is it feasible to ship a binary blob of weights along with the compiler simply to provide code suggestions? Is it viable to build GPU acceleration of token generation in GHC itself? Are users willing to connect to a cloud service for such suggestions? The existence of services such as Copilot [9] seems to suggest so, but this is more in the domain of the IDE than the compiler.

Another challenge is validating AI–powered code suggestions, and determining whether the generated code satisfies the intent of the programmer [10]. In this thesis, we explore how to perform *valid* program repair and synthesis, i.e., given a specification in the form of types and tests, we synthesize programs that are guaranteed to satisfy the specification and tests. I believe this will be an important tool in the toolbox for LLM–powered program synthesis: The LLM synthesizes an approximate solution, and then apply tools like PropR and Spectacular (as described in this thesis) to *repair* the generated programs to synthesize one that fully satisfies the specification.

1.2.6 Property–Based Testing

Property–based testing frameworks such as QuickCheck [8] allow users to specify properties that functions must satisfy and can be viewed as an intuitive way of specifying what constraints should hold for the program. These properties are tested by generating arbitrary data based on the type of the property and verifying that the property holds. This allows one property to be the equivalent of hundreds or thousands of unit tests and using shrinking to generate a small counterexample when a property does not hold. These counterexamples can then be used in conjunction with program coverage to localize the error by noting which expressions were involved in the evaluation leading to the failure of the property. We use properties and their counterexamples in the second paper of this thesis to guide automated program repair [17]. In [18], we use an ECTA–based technique to synthesize expressions and partition them into equivalence classes to efficiently synthesize properties for large modules.

ECTA–based Synthesis

Equality–Constrained Tree–Automata is a recently introduced synthesis technique that was applied to Haskell to synthesize programs from the functions in the Haskell prelude [22]. *Equality–constrained tree automata* (ECTAs) [22] are a new data structure for representing and enumerating a large space of terms with constraints between sub–terms. We go into more detail on how they work in the introduction in chapter 4. In chapter 4 , we use ECTAs to synthesize Haskell expressions that correspond to a value that can be compared for equality when applied to some arguments. In this way, we can automatically discover expressions that have the same value: a property.

Fault Localization

Fault localization is a technique that takes a buggy program and tries to determine which part of the program causes the bug, based on static and dynamic analysis. In this thesis, we employ two dynamic fault localization methods, tracing and spectrum analysis. We do not explore static methods, since the types of faults detected by static methods are covered in part by the expressive type system in Haskell.

Spectrum–Based Fault Localization

Spectrum–based fault localization (SBFL) is considered one of the most prominent fault localization techniques due to its efficiency and effectiveness [35].

With SBFL, we assume the existence of a test suite of both passing and failing tests and assume that the tests cover the expected behavior. By *instrumenting* the code using a coverage tool such as HPC, we can run the test suite and note the locations involved in each test and whether that test passes or fails. We use a heuristic called *suspiciousness*, saying that locations that are more frequently involved in failing tests than in passing tests are more suspicious. Using various formulas based on the spectrum, we can assign a suspiciousness score to each location in the program based on the number of times it is evaluated in passing and failing tests, respectively, as well as the total number of passing and failing tests in the program.

In the PropR paper [17], we use a naive version of SBFL where we only note the locations that are involved in a failing test and do not consider passing tests or the total number of tests. We improve this fault localization in the functional spectrums paper [16], where we introduce the TastySpectrum library that allows developers to add spectrum generation and analysis to their test suites. Based on HPC, we add a pass to the test–runner framework, allowing the spectrum to be collected when the test suite is run. The library implements spectrum analysis using traditional formula–based methods, as well as novel rules based on the types and AST structure of the program. By using a formula–based approach, we can more effectively target program repair by prioritizing parts of the program that involved in failing tests and avoiding parts common to all tests.

Program Tracing

Program tracing is a technique based on instrumenting a program and capturing which part of the program are evaluated when the program is run (program coverage) and, in particular, in *which order* the expressions in the program are evaluated (tracing). This sometimes includes the values involved.

In [15], we introduce CSI: Haskell, which extends GHC's built–in Haskell Program Coverage (HPC) to add runtime tracing of Haskell programs. By collecting a *suffix* of the trace, we can capture most of the information related to fault localization, with minimal overhead.

1.3 Future Work

In this thesis, we present PropR, an automatic repair tool for small Haskell programs. It works well on small programs that are only an identifier or two from being correct. However, there is still a long way to go to make it a practical program repair tool for larger programs and packages with thousands of lines of code [16].

In this section, we provide a blueprint for how the elements presented in this thesis can be integrated and combined into a single tool that could scale much better than the current approach taken by PropR. This tool is tentatively named PrIM: PropR Improved.

1.3.1 PrIM: PropR Improved

As a step toward practical automatic program repair, we envision an improved version of PropR that draws on the ground work laid out in this thesis. In particular, for more practical automatic program repair, multiple components are required:

- We must scale the synthesis to not be limited to single identifiers,
- we must be able to repair error–based and non–terminating faults,
- we must more accurately pinpoint parts of the program that should be targeted for repair, and finally,
- for unit–test–based test suites, we must to enrich them with property–based tests to better capture which parts are at fault and which parts are not.

Integrating ECTA–based synthesis.

In the current design of PropR, valid hole–fits are generated using a hole–fit plugin that uses both the valid hole–fits as suggested by GHC, as well as expressions extracted from the module being repaired. While refinement hole–fits allow us to iteratively synthesize multi–identifier expressions, it is slow in practice. However, we can build on our work from Spectacular [18] (presented in chapter 4 in this thesis), which uses a hole–fit plugin to query GHC for the local and global context of a hole and constructs an ECTA using identifiers from said context. The ECTA is then used to synthesize expressions that match the type of the hole [18]. By integrating ECTA–based synthesis into the hole–fit synthesis step (⑦ in figure 1.11) we can make more

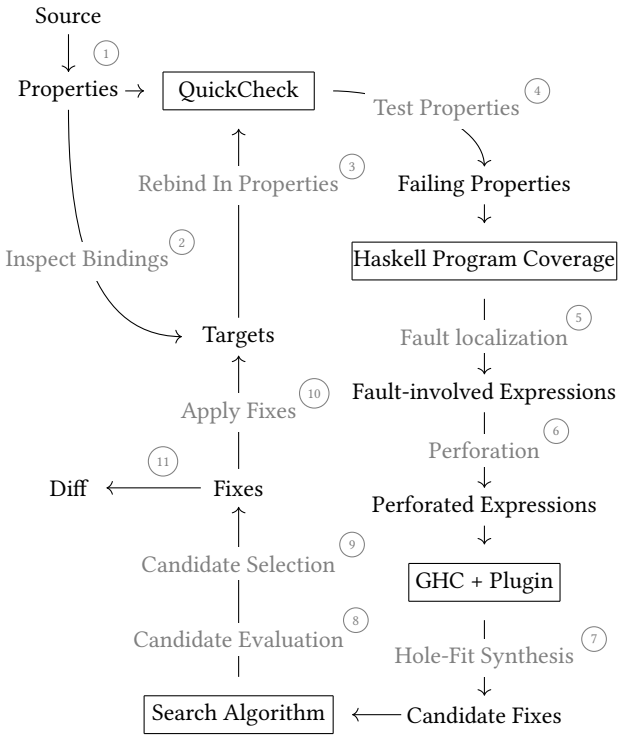


Figure 1.11: The PropR test–localize–synthesize–rebind loop

multi–identifier candidates available. This would make the repair process faster, both due to the higher–quality candidates, and removing the need to use the slower refinement–hole fit process to synthesize bigger expressions, relegating those to the ECTA instead.

Using the equivalence class–based bucketing in Spectacular, we can ensure that we only test unique candidates. Instead of evaluating both `[] :: [Int]` and `tail [1] :: [Int]`, we can choose the simplest one, `[]`, and discard the others. Although less useful for small expressions, this shrinks the search space for larger expressions.

Integrating Fault Localization.

In step (5) of the PropR repair loop, PropR runs the program and naively localizes the fault to *any* location touched by a failing test. Although guaranteed to produce a set of locations that contains the fault, we can do better. To more accurately pinpoint parts of the program that should be targeted, we must improve fault localization and introduce more sophisticated heuristics.

```haskell
gcd' :: Int -> Int -> Int
gcd' 0 b = gcd' 0 b
gcd' a b | b == 0 = a
gcd' a b =
  if (a > b)
    then gcd' (a - b) b
    else gcd' a (b - a)
```

Figure 1.12: A gcd function that loops indefinitely due to a buggy base case. Rewritten in Haskell from a GenProg example [23].

CSI: Haskell Error–based and non–terminating faults can be difficult to repair, since the program halts in an unfinished state and does not produce any output apart from the error. However, being able to capture what was being evaluated right before the error occurred and observe the control–flow of non–terminating loops lets us more effectively handle these types of faults. Using a version of GHC that implements CSI: Haskell, we can produce a trace of the code involved in the failing test. This can be especially useful for faults such as the non–termination in `gcd'` in figure 1.12, where the first base case is incorrect. Instead of returning `b` in the `gcd' 0 b` case, it instead loops without doing any work.

```
Ex: Killed
CallStack (from HasCallStack):
  error, called at Ex.hs:19:72 in main:Main
Recently evaluated locations:
  Ex.hs:14:78-14:85  "Killed"
  Ex.hs:14:71-14:86  (error "Killed")
 Previous expressions:
  repeats (60 times in window):
    Ex.hs:4:1-9:23   Main:gcd'
    Ex.hs:4:12-4:19  ... = gcd' 0 b
 There were 38347886 evaluations in total but only 250 were recorded.
 Re-run again with a bigger trace length for better coverage.
```

Figure 1.13: The trace generated from running `gcd' 0 55`, with signal handler to kill the program and produce an error.

As captured by the suffix of the trace, the base case loop is clear: as seen in figure 1.13 it is the whole trace! Using this information to guide the search to focus on the base case, we could repair `gcd'` much more quickly than we are able to today.

As shown in chapter 5 in this thesis, the source of a fault is often in the top 250 locations in the trace [15]. This is spurred by the fact that Haskell is lazy, meaning that incorrect values involved in an error are often produced right before the error occurs, allowing us to observe the production of said values in the trace [15]. By using the location in the trace as a base for a heuristic for which locations we target first, we would speed up the automatic repair.

Functional Spectrums While the naive approach of considering every location involved in a fault as a likely culprit is guaranteed to work, it means that the search space can become very large. Using the techniques described in chapter 6 in this thesis to generate a spectrum from the test suite in figure 1.14, we can more accurately pinpoint the fault. This spectrum captures that while the first test case does not involve the faulty base cases passes, the test case that *only* involves the base case fails.

```
prop_1 = gcd' 1071 1029 == 21
prop_2 = gcd' 0 55 == 55
```

Figure 1.14: Tests involving `gcd'` function in figure 1.12

Table 1.1: A spectrum from running the test suite in figure 1.14, with a timeout of 0.5 seconds.

name	result	Ex.hs:4:17	4:19	4:12-19	5:12	5:17	5:21	...
prop_1	True	0	0	0	28	28	1	...
prop_2	False	209562382	209562382	209562382	0	0	0	...

As seen in table 1.1, it is clear that the fault lies in 4:12–19. Using classical spectrum–based fault localization algorithms, this would assign a high suspiciousness score to the faulty base case, and guide the search towards the faulty expression faster than otherwise.

Improved Repair of Under–Specified Programs.

It is often the case that a module or part of a module is under–specified but is involved in a failure. Even if there is some specification in the form of unit tests, we can potentially get better coverage by generating property–based tests, which would improve the spectrums that we generate. If we have an older version of the source code or a reference implementation that does not have the bug, we can use Spectacular [18] to synthesize a specification of the correct version. The synthesized specification can then be used for fault localization and repair of the later, buggy version.

Evaluation

If we had all these components in place, I believe that we could scale PropR to work on much larger programs. One critique of the PropR paper was that the student dataset used to evaluate was not representative of actual programs. Originally chosen due to the availability of a comprehensive test suite and of data points close to a correct implementation, it did not accurately reflect real–world Haskell code. Since then, the HasBugs dataset by Applis et al. has become available [3].

Similar to the Defects4J dataset for Java [21], the HasBugs dataset includes data points from Haskell projects such as HLS, Cabal, and Pandoc, and includes descriptions of the bugs, fault locations, locations where fixes were applied, and tests that cover the bugs. By using the HasBugs dataset instead of the student dataset, we could more accurately evaluate the effectiveness of the PropR approach on real–world, large–scale Haskell programs.

Another dataset that could be useful is the Nofib–Buggy dataset [36]. Nofib–buggy consists of programs from the nofib test suite used to benchmark and regression test GHC, but with bugs intentionally introduced. This dataset is useful for evaluating fault localization tools; however, the programs do not include an extensive test suite, but rather a simplistic suite consisting of a scripted unit test with no properties. By writing a test suite we could use nofib–buggy to evaluate automatic program repair tools, and evaluate the effectiveness of the Spectacular approach, by bootstrapping a test suite from the non–buggy version of the program as a reference implementation.

1.3.2 Re–Thinking Compiler Design

In this thesis, I have already shown how program repair can work for small functional programs. However, scaling this up to larger programs is a challenge. This has prompted some ideas for improvements to the Haskell toolchain to better enable tools such as those described in this thesis.

Infrastructure

A big part of the problem boils down to infrastructure: Most production code is not written as one large module but spread out over multiple modules and multiple packages. Compiling and recompiling these after changes are made takes a lot of time, and running the test suite takes even longer. The approach we have taken to program repair relies on rapid turnaround in order to be able to check each guess before moving on to the next. This can be parallelized and run in multiple processes, but the resources and time required to repair large programs at scale are generally not accessible to your standard

programmer. Better support for *incremental compilation* would greatly improve this situation, allowing us to only recompile the parts of the module that changed and only rerunning the tests impacted by the change. The approach taken by Unison [2] is a promising step in this direction. In Unison, functions are compiled and stored in a database, with references to other functions stored as indexes to entries in this database. This allows Unison to avoid recompilation of the whole package and instead only recompile the function that was changed [2].

Loss of Context

A common detriment to program synthesis is the loss of context. When synthesizing, you want to have as much context as possible, as synthesis is essentially a function of a context to a guess.

Compilers like GHC tend to work on the notion of building up context during each compilation pass, only to erase most of that context once the compilation pass has finished. This means that synthesizers have to redo a lot of type checking and context building done by the compiler, or, as I have done in this thesis, integrating the synthesis into the compiler pass itself. This comes at a cost: any time you want to re–run the synthesis, you have to start compilation all over again to access the context, and threading the previous context throughout the compilation passes in case it is useful for later validation and synthesis.

If we could preserve and make each context available to external tools, that would greatly improve the synthesis and repair process. One way of doing so would be to take snapshots at certain points of the compilation that could be used to restart compilation or access the context at that point. Another way would be to parameterize compilation over a certain location allowing us to modify that location and only recompile the changed parts.

1.4 Conclusion

By bringing together all the components of fault localization, targeted synthesis, and efficient ECTA–based synthesis into a practical program repair loop like PropR, we have shown that the additional specification found in test suites and rich typing information available in languages like Haskell can be used to go beyond just verification and that we can use this information to aid programmers during development.

1.5 Thesis structure

This thesis builds on the work from my licentiate (mid–term) thesis [14], and contains some overlap in the first three chapters. The first chapter (introduction) has been reworked and extended with future work and a relevant new background section. The second chapter (suggesting valid–hole fits) is unchanged and represents the published version of the paper. The third chapter has been updated to reflect the final camera–ready version of the paper.

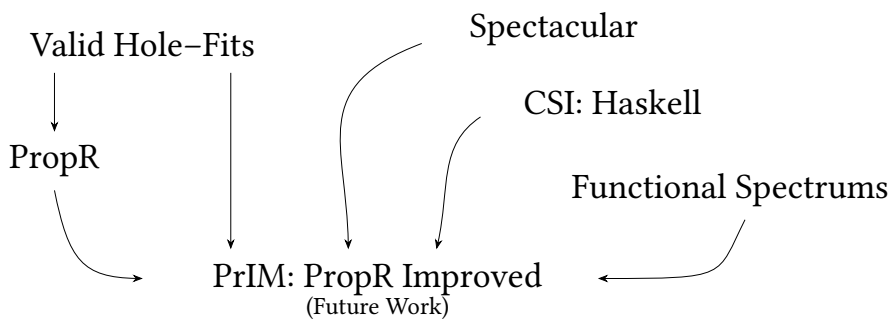


Figure 1.15: Thesis Structure Overview

Paper 1:
Suggesting Valid Hole Fits for Typed–Holes (Experience Report) [11]

by **Matthías Páll Gissurarson**

Suggesting Valid Hole Fits documents the implementation and design of the synthesis of valid hole–fits as they initially appeared in GHC. Of particular interest is the sorting of hole–fits by "relevance", using either the simplistic number of type constructors (the "size" of the type) heuristic, the more advanced subsumption sorting, where more "specific" types are treated as more "relevant" than more general types, and refinement hole–fits that are valid hole–fits that introduce additional holes to be filled.

Statement of contributions Single authored

Appeared in: Haskell Symposium 2018 (**Haskell '18**)

Paper 2:
PropR: Property-Based Automatic Program Repair [17]

by **Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands**

In the PropR paper, we introduce PropR, a tool to automatically repair Haskell programs using a combination of typed–hole synthesis to repair program expressions with well–typed replacements and using QuickCheck properties to verify the repair. We use GHC's Haskell program coverage functionality to figure out which expressions are involved in a fault based on QuickCheck generated counterexamples to failing properties, a typed–hole valid hole–fit plugin to generate well–typed replacements as fixes for said expressions, and a genetic algorithm to select and combine fixes based on QuickCheck property results after applying a fix.

Statement of contributions I was the main driver behind the paper in conjunction with Leonhard Applis, who was the joint first–author. I implemented the synthesis and repair as well as writing the technical section of the paper and parts of the introduction, whereas Leonhard focused on the genetic repair algorithm and the experimental verification.

Appeared in: International Conference on Software Engineering 2022 (**ICSE '22**) – Technical Track

Paper 3:
Spectacular: Finding Laws from 25 Trillion Programs [18]

by **Matthías Páll Gissurarson, Diego Roque, and James Koppel**

Spectacular is a new tool for automatically discovering candidate laws for use in property–based testing. Incorporating many of the ideas from QuickSpec, but using the recently developed technique of ECTAs, Spectacular can explore vastly larger, fully polymorphic program spaces efficiently.

Statement of contributions I wrote the implementation of Spectacular and evaluation, while Diego Roque provided some initial exploration of the problem and James Koppel provided guidance on the use of ECTAs.

Appeared in: International Conference on Software Testing 2023 (**ICST '23**) – Research Track

Paper 4:
CSI: Haskell – Tracing Lazy Evaluations in a Functional Language [15]

by **Matthías Páll Gissurarson and Leonhard Applis**

In CSI: Haskell, we extended the Haskell Program Coverage implementation in GHC to enable runtime tracing of Haskell Programs. In the paper, we focus on the suffix of such traces and investigate how effective at pointing to faulty locations in the nofib–buggy dataset they are.

Statement of contributions Both authors contributed equally to the paper. I forked GHC and HPC and did the initial idea and design of the problem and assisted with the analysis on the nofib–buggy data set.

Appeared in: Symposium on Implementation and Application of Functional Languages 2023 (**IFL '23**), and was awarded the *Peter Landin Prize* for the best paper presented at the symposium as selected by the program committee [1].

Paper 5:
Functional Spectrums: Exploring Spectrum–Based Fault Localization in Functional Programming [16]

by **Leonhard Applis, Matthías Páll Gissurarson, and Annibale Panichella**

In Functional Spectrums, we implemented an additional pass to the Tasty test framework and associated GHC plugin to create typed–augmented spectrums for fault localization for functional programs. In the paper, we investigate how effective this approach is for the HasBugs data set.

Statement of contributions Both first authors contributed equally to the paper. I implemented the ingredient and library that extracts the spectrums, as well as the GHC plugin for extracting typing information and most of the rule–based system for quantifying the various values involved.

Manuscript

Bibliography

[1] *IFL '23: Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*, New York, NY, USA, 2023. Association for Computing Machinery.

[2] The Unison team . The Unsion Language . Website, 2024. `https://www.unison-lang.org/docs/`.

[3] L. Applis and A. 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.

[4] L. Augustsson. The Djinn package, 2014.

[5] B. Blöndal, A. Löh, and R. Scott. Deriving Via: Or, how to turn handwritten instances into an anti-pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, pages 55–67, 2018.

[6] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating Large Language Models Trained on Code, 2021.

[8] K. Claessen and J. 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, pages 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[9] T. Dohmke. GitHub Copilot is generally available to all developers. *The GitHub Blog (blog), June*, 21, 2022.

[10] K. Ferdowsi, R. L. Huang, M. B. James, N. Polikarpova, and S. Lerner. Validating AI-Generated Code with Live Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA, 2024. Association for Computing Machinery.

[11] M. P. Gissurarson. Suggesting Valid Hole Fits for Typed–Holes (Experience Report). In *ACM SIGPLAN International Symposium on Haskell*, pages 179–185, 2018. This is the first paper in this thesis, and is included in chapter 2.

[12] M. P. Gissurarson. Suggesting Valid Hole Fits for Typed-Holes in Haskell. Master's thesis, Chalmers University of Technology, 2018.

[13] M. P. Gissurarson. Hole Fit Plugins for GHC . Poster presented at the ICFP Student Research Competition, 2019. Available at `https://mpg.is/papers/gissurarson2019hole.pdf`.

[14] M. P. Gissurarson. The Hole Story: Type-Directed Synthesis and Repair, 2022. Licentiate Thesis, Chalmers University of Technology.

[15] M. P. Gissurarson and L. Applis. CSI: Haskell – Tracing Lazy Evaluations in a Functional Language. IFL, 2023. This is the fourth paper in this thesis, and is included as chapter 5.

[16] M. P. Gissurarson, L. Applis, and A. Panichella. Functional Spectrums: Exploring Spectrum–Based Fault Localization in Functional Programming. Manuscript, 2024. This is the fifth paper in this thesis, and is included as chapter 6.

[17] M. P. Gissurarson, L. Applis, A. Panichella, A. van Deursen, and D. Sands. PropR: Property-Based Automatic Program Repair. ACM 44th International Conference on Software Engineering (ICSE), 2022. This is the second paper in this thesis, and is included as chapter 3.

[18] M. P. Gissurarson, D. Roque, and J. Koppel. Spectacular: Finding Laws from 25 Trillion Programs. In *ICST*, page 13, 2023. This is the third paper in this thesis, and is included in chapter 4.

[19] S. Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.*, 46(1):317–330, Jan 2011.

[20] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov 2020.

[21] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[22] J. Koppel, Z. Guo, et al. Searching Entangled Program Spaces. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2022.

[23] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[24] J. Lubin, N. Collins, C. Omar, and R. Chugh. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.*, 4(ICFP), Aug 2020.

[25] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. *SIGPLAN Not.*, 45(11):37–48, sep 2010.

[26] M. Martinez and M. Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, 151:65–80, 2019.

[27] N. Mulleners, J. Jeuring, and B. Heeren. Program Synthesis Using Example Propagation. In *Practical Aspects of Declarative Languages*, pages 20–36, Cham, 2023. Springer Nature Switzerland.

[28] N. Mulleners, J. Jeuring, and B. Heeren. Example-Based Reasoning about the Realizability of Polymorphic Programs. volume 8, New York, NY, USA, Aug 2024. Association for Computing Machinery.

[29] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[30] OpenAI et al. GPT-4 technical report, 2024.

[31] P.-M. Osera and S. Zdancewic. Type-and-Example-Directed Program Synthesis. *SIGPLAN Not.*, 50(6):619–630, Jun 2015.

[32] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI '12, Proc. the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–286. ACM, 2012.

[33] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. Association for Computing Machinery.

[34] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama. Liquid Information Flow Control. *Proc. ACM Program. Lang.*, 4(ICFP), Aug 2020.

[35] Q. I. Sarhan and A. Beszedes. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access*, 10:10618–10639, 2022.

[36] J. Silva. The Buggy Benchmarks Collection, 2007. Josep Silva self-published on his website / university.

[37] A. Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

2

Suggesting Valid Hole Fits for Typed–Holes (Experience Report)

Matthías Páll Gissurarson

*Haskell Symposium 2018 (**Haskell '18**)*

Abstract. Type systems allow programmers to communicate a partial specification of their program to the compiler using types, which can then be used to check that the implementation matches the specification. But can the types be used to aid programmers during development? In this experience report I describe the design and implementation of my lightweight and practical extension to the typed-holes of GHC that improves user experience by adding a list of *valid hole fits* and *refinement hole fits* to the error message of typed-holes. By leveraging the type checker, these fits are selected from identifiers in scope such that if the hole is substituted with a valid hole fit, the resulting expression is guaranteed to type check.

```
Found hole: _ :: [Int] -> Int
In the expression: _ :: [Int] -> Int
In an equation for ‘it’: it = _ :: [Int] -> Int
Relevant bindings include
  it :: [Int] -> Int (bound at <interactive>:4:1)
Valid hole fits include
  head :: forall a. [a] -> a
  last :: forall a. [a] -> a
  length :: forall (t :: * -> *) a. Foldable t => t a -> Int
  maximum :: forall (t :: * -> *) a.
             (Foldable t, Ord a) => t a -> a
  minimum :: forall (t :: * -> *) a.
             (Foldable t, Ord a) => t a -> a
  product :: forall (t :: * -> *) a.
             (Foldable t, Num a) => t a -> a
  (Some hole fits suppressed; use
   -fmax-valid-hole-fits=N or -fno-max-valid-hole-fits)
Valid refinement hole fits include
  foldl1 (_ :: Int -> Int -> Int)
    where foldl1 :: forall (t :: * -> *) a. Foldable t =>
                    (a -> a -> a) -> t a -> a
  foldr1 (_ :: Int -> Int -> Int)
    where foldr1 :: forall (t :: * -> *) a. Foldable t =>
                    (a -> a -> a) -> t a -> a
  foldl (_ :: Int -> Int -> Int) (_ :: Int)
    where foldl :: forall (t :: * -> *) b a. Foldable t =>
                   (b -> a -> b) -> b -> t a -> b
  foldr (_ :: Int -> Int -> Int) (_ :: Int)
    where foldr :: forall (t :: * -> *) a b. Foldable t =>
                   (a -> b -> b) -> b -> t a -> b
  ($) (_ :: [Int] -> Int)
    where ($) :: forall a b. (a -> b) -> a -> b
  const (_ :: Int)
    where const :: forall a b. a -> b -> a
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N
   or -fno-max-refinement-hole-fits)
```

Figure 2.1: Typed-hole error message extended with hole fits.

2.1 Introduction

When writing documentation for libraries, the Haskell community often goes the route of having descriptive function names and clear types that leverage type synonyms in order to push much of the documentation to the type-level. As developers program in Haskell, they often use a style of programming called *Type-Driven Development*. They write out the input and output types of functions before writing the functions themselves [3]. A consequence of this approach is that the compiler has a lot of type information that is only used during type checking. Can we make better use of the extra information and type-level documentation and improve user experience? According to the GitHub survey [5], user experience is the third most important factor when choosing open source software, after stability and security, and thus an important consideration.

We can leverage the richness of type information in library documentation along with users' type signatures by extending typed-hole error messages with a list of *valid hole fits* and *refinement hole fits*. These allow users to find relevant functions and constants when a typed-hole is encountered:

Valid hole fits and refinement hole fits can be used to effectively aid development in many scenarios by allowing users to view and search type-level documentation directly, thus improving the user experience.

Note: in the interest of reducing noise in the output in this report, I have opted to show only the fits themselves, and not the type application nor provenance of the fit as displayed in the ouput by default. The amount of detail in the output is controlled by flags; the format used here is achieved by setting the `-funclutter-valid-hole-fits` flag. An example of the full default output can be seen in figure 2.2.

```
product :: forall (t :: * -> *) a.
           (Foldable t, Num a) => t a -> a
  with product @[] @Int
  (imported from ‘Prelude’
   (and originally defined in ‘Data.Foldable’))
```

Figure 2.2: The full output for a fit for `_ :: [Int] -> Int`.

2.1.1 Contributions

In this experience report, I do the following:

- Describe *valid hole fits* and *refinement hole fits* as I have implemented them in GHC. Valid hole fits allow users to tap in to the extra type

information available during compilation or interactively using GHCi, while refinement hole fits extend valid hole fits beyond identifiers to find functions that need additional arguments.

- Provide a detailed explanation of how I have implemented valid hole fits and refinement hole fits in GHC, and how I solved technical hurdles along the way.

- Show the usefulness of hole fits in case studies on an introductory exercise and when using the `lens` library.

- Finally, I present an application of valid hole fits to libraries using type-in-type to annotate functions with non-functional properties, and show an example.

2.1.2 Background

Typed-Holes in GHC were introduced in version *7.8* and implemented by Simon Peyton Jones, Sean Leather and Thijs Alkemade [7]. Inspired by a similar feature in Agda, typed-holes allow a user of GHC to have "holes" in their code, using an underscore (`_`) in place of an expression. When GHC encounters a typed-hole, it generates an error with information about that hole, such as its location, the (possibly inferred) type of the hole and relevant local bindings [4]. Typed-holes can also be given names by appending characters, e.g. `_a` and `_b`, to allow users to distinguish between holes.

Valid Hole Fits: We use the type information available in typed-holes to make them more useful for programmers, by extending the typed-hole error message with a list of *valid hole fits*. Valid hole fits are expressions which the hole can be replaced with directly, and the resulting expression will type check. An example of valid hole fits can be seen in figure 2.1.

Refinement Hole Fits: It is often the case that a single identifier is not enough to implement the desired function, such as when writing the `product` function (`foldr (*) 1`). To suggest useful hole fits for these cases, we introduce *refinement hole fits*. Refinement hole fits are valid hole fits that have one or more additional holes in them. The number of additional holes is controlled by the refinement level, set via `-frefinement-level-hole-fits`. A refinement level of N means that hole fits with up to N additional holes in them will be considered. An example of refinement hole fits can be seen in figure 2.1, in which the refinement level is 2.

2.2 Case Studies

To show that valid hole fits and refinement hole fits can be used to effectively aid development, we consider two cases, an introductory programming exercise where we use the `Prelude` and an advanced case using the `lens` library.

2.2.1 Exercise from Programming in Haskell

To study how the valid hole fits perform when used by beginners, I looked at an example from Graham Hutton's introductory text, Programming in Haskell [9]. In exercise 4.8.1, students are asked to implement `halve :: [a] -> ([a],[a])`, which should split a list of even length into two halves. With refinement hole fits enabled, we can query GHCi by writing:

```
Prelude> _ :: [a] -> ([a], [a])
```

In response, GHCi will then generate a typed-hole error, including a list of valid refinement hole fits:

```
Valid refinement hole fits include
  break (_ :: a1 -> Bool)
    where break :: forall a.
                   (a -> Bool) -> [a] -> ([a], [a])
  span (_ :: a1 -> Bool)
    where span :: forall a.
                  (a -> Bool) -> [a] -> ([a], [a])
  splitAt (_ :: Int)
    where splitAt :: forall a. Int -> [a] -> ([a], [a])
  mapM (_ :: a1 -> ([a1], a1))
    where mapM :: forall (t :: * -> *) (m :: * -> *) a b.
                  (Traversable t, Monad m) =>
                  (a -> m b) -> t a -> m (t b)
  traverse (_ :: a1 -> ([a1], a1))
    where traverse :: forall (t :: * -> *) (f :: * -> *) a b.
                      (Traversable t, Applicative f) =>
                      (a -> f b) -> t a -> f (t b)
  const (_ :: ([a1], [a1]))
    where const :: forall a b. a -> b -> a
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N
   or -fno-max-refinement-hole-fits)
```

One of the suggested fits is the `splitAt (_ :: Int)` refinement, and given that the task is to *split* a list, this seems like a good fit. In this way, the student can discover the `splitAt` function from the prelude, and a correct

solution (`halve xs = splitAt (length xs `div` 2) xs`) is easy to find using refinement hole fits.

2.2.2 The Lens Library

In the `lens` library [10], the functions can be hard to find with Hoogle (see section 2.5), due to the library's extensive use of type synonyms. As an example, consider the following:

```
import Control.Lens
import Control.Monad.State

newtype T = T { _v :: Int }

val :: Lens' T Int
val f (T i) = T <$> f i

updT :: T -> T
updT t = t &~ do
    _ val (1 :: Int)
```

For the hole in the above, the typed-hole message includes:

```
Found hole:
  _ :: ((Int -> f0 Int) -> T -> f0 T) -> Int -> State T a0
```

where `f0` and `a0` are ambiguous type variables. Searching for this type signature in Hoogle (version 5.0.17) yields no results from the lens library.

When valid hole fits are available, GHC will output the following list of valid hole fits:

```
Valid hole fits include
  (#=) :: forall s (m :: * -> *) a b. MonadState s m =>
          ALens s s a b -> b -> m ()
  (<#=) :: forall s (m :: * -> *) a b. MonadState s m =>
           ALens s s a b -> b -> m b
  (<*=) :: forall s (m :: * -> *) a. (MonadState s m,
           Num a) => LensLike' ((,) a) s a -> a -> m a
  (<+=) :: forall s (m :: * -> *) a. (MonadState s m,
           Num a) => LensLike' ((,) a) s a -> a -> m a
  (<-=) :: forall s (m :: * -> *) a. (MonadState s m,
           Num a) => LensLike' ((,) a) s a -> a -> m a
  (<<*=) :: forall s (m :: * -> *) a. (MonadState s m,
            Num a) => LensLike' ((,) a) s a -> a -> m a
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N
   or -fno-max-refinement-hole-fits)
```

Though the names of the functions are opaque, we see that integrating the valid hole fits into the typed-holes and integrating with the type checker itself is a clear win, allowing us to find a mulitude of relevant functions from `lens`.

2.3 Implementation

The valid hole fit suggestions for typed-holes are implemented as an extension to the error reporting mechanism of GHC, and are only generated during error reporting of holes. This means that we can emphasize utility rather than performance, as any overhead will only be incurred when the program would in any case fail due to an error.

2.3.1 Inputs & Outputs

The entry into the valid hole fit search is the function called `findValidHoleFits` in the **`TcHoleErrors`** module [1]:

```
findValidHoleFits :: TidyEnv -- Type env for zonking
                  -> [Implication] -- Enclosing implics
                                   -- containing givens
                  -> [Ct] -- Unsolved simple constraints
                          -- in the implic for the hole.
                  -> Ct -- The hole constraint itself
                  -> TcM (TidyEnv, SDoc)
```

This function takes the hole constraint that caused the error, the unsolved simple constraints that were in the same set of wanted constraints as the hole constraint, and the list of implications which that set was nested in. The tidy type environment at that point of error reporting is also passed to the function, and used later for *zonking* [2]. To zonk, we use

```
zonkTidyTcType :: TidyEnv -> TcType -> TcM (TidyEnv, TcType)
```

from **`TcMType`**, which uses the tidy type environment to ensure that the resulting types are consistent with the rest of the error message and other error messages. The function returns the (possibly) updated tidy type environment and the message containing the valid hole fits.

[1] Available in GHC HEAD at:
`http://git.haskell.org/ghc.git/blob/refs/heads/master:/compiler/typecheck/TcHoleErrors.hs`

[2] In the context of GHC, *zonking* is when a type is traversed and mutable type variables are replaced with the real types they dereference to.

2.3.2 Relevant Constraints

The unsolved simple constraints are constraints imposed by the call-site of the hole. As an example, consider the holes `_a` and `_b` in the following:

```
f :: Show a => a -> String
f x = show (_b (show _a))
```

Here, the type of `_a` and the return type `_b` need to fulfill a show constraint. These constraints constitute the set of unsolved simple constraints {**Show** t_a, **Show** t_b}, where t_a is the type of `_a`, and **String** -> t_b is the type of `_b`. Since valid hole fits are only considered for one hole at a time, the unsolved simple constraints are filtered to only contain constraints relevant to the current hole. For hole `_a`, this would be {**Show** t_a}, and for hole `_b` this would be {**Show** t_b}. This is done by discarding those constraints whose types do not share any free type variables with the type of the hole. I call this filtered set of constraints the *relevant constraints*.

2.3.3 Candidates

Candidate hole fits are identifiers gathered from the environment. We consider only the elements in the global reader and the local bindings at the location of the hole (discarding any shadowed bindings). The global reader contains identifiers that are imported or defined at the top-level of the module. Using the local bindings allows us to include candidates bound by pattern matching (such as function arguments) or in **let** or **where** clauses. As an example, in:

```
f (x:xs) = let a = () in _
  where k = head xs
```

the global reader elements considered as candidates are the functions in the **Prelude** and `f`, while the local binding candidates are `f`, `x`, `xs`, `a` and `k`. When shadowed bindings are removed, the `f` from the global reader is discarded. For global elements, a lookup is performed in the type checker to find their associated identifiers, discarding any elements not associated with an identifier or data constructor (like type constructors or type variables). Candidates from `GHC.Err` (like `undefined`) are discarded, since they can be made to match any type at all, and are unlikely to be the function that the user is looking for.

2.3.4 Checking for Fit

Each of these candidates is checked in turn by invoking the `tcCheckHoleFit` function. This function starts by capturing the set of constraints and wrapper

emitted by the `tcSubType_NC` function when invoked on the type of the candidate and the type of the hole. The `tcSubType_NC` function takes in two types and returns the core wrapper needed to go from one type to the other, emitting the constraints which must be satisfied for the types to match. The relevant constraints are added to this set of constraints, to ensure that any constraints imposed by the call-site of the hole are satisfied as well. This extended set is wrapped in the implications that the hole was nested in, so that any givens contained in the implications (such as that `a` satisfies the show constraint in the example above) are passed along. These are passed to the simplifier, which checks the constraints. If the set is soluble, the candidate is a valid hole fit, and the wrapper is returned. The wrapper is used later to show *how* the type of the fit matches the type of the hole by showing the type application, like `product @[] @Int` in figure 2.2.

2.3.5 Refinement hole fits

For refinement hole fits, N fresh flexible type variables are created, $a_1, \ldots, a_N$, where N is the refinement level set by the `-frefinement-level-hole-fits` flag. We then look for fits not for the type of the hole, t_h, but for the type $a_1 \to \cdots \to a_N \to t_h$. These additional type variables allow us to emulate additional holes in the expression. To limit the number of refinement hole fits, additional steps are taken after we have checked whether the type fits, to check whether all the fresh type variables ended up being unified with a concrete type. This ensures that fits involving fresh variables such as `id (_ :: a1 -> a2 -> a) (_ :: a1) (_ :: a2)` are discarded unless explicitly requested by the user by passing the `-fabstract-refinement-hole-fits` flag. If a match is found, the fresh type variables are zonked and the type they were unified with read off them, allowing us to show the types of the additional holes (like `Int -> Int -> Int` for the hole in the `foldl1 (_ :: Int -> Int -> Int)` fit).

2.3.6 Sorting the Output

As with relevant bindings, only 6 valid hole fits are displayed by default. To increase the utility of the valid hole fits, we sort the fits by relevance, which is approximated in two ways.

Sorting by Size: The default approximation sorts by the size of unique types in the type application needed to go from the type of the fit to the type of the hole, as defined by the core expression wrapper returned when the fit was found. The size is computed by applying the `sizeTypes` function, which counts the number of variables and constructors:

Table 2.1: Sizes of matches for `_ :: String -> [String]`

Fit	Type	Application	Size
`lines`	`String -> [String]`		0
`repeat`	`a -> [a]`	`String`	2
`mempty`	`Monoid a => a`	`String -> [String]`	6

Only unique types are considered, since fits that require many different types are in some sense "farther away" than fits that require only a few unique types. This method is faster and returns a reasonable ordering in most cases.

Sorting by Subsumption: The other approximation is enabled by the `-fsort-by-subsumption-hole-fits` flag. When sorting by subsumption, a subsumption graph is constructed by checking all the fits that have been found for whether they can be used in place of any other found fit. A directed graph is made, in which the nodes are fits and the edges are the result of the subsumption check, where fit a has an edge to fit b fit if b could be used anywhere that a could be used. An example of such a graph can be seen in figure 2.3. The fits are sorted by a topological sort on this graph, so that if b could be used anywhere a could be used, then b appears after a in the output. This ordering ensures that more specific fits (such as those with the same type as the hole) appear earlier than more abstract, general fits.

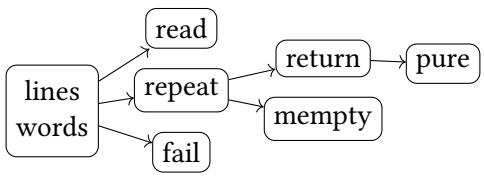


Figure 2.3: The subsumption graph for matches for `_ :: String -> [String]`. Here `lines` would come before `repeat`, `read`, and `fail`, `repeat` before `mempty` and `return`, etc.

2.3.7 Dealing with Side-effects

When GHC simplifies constraints, it does so by side-effect on the type variables involved and the evidence contained within implications. To ensure that checks for fits do not affect later checks, we must encapsulate these side-effects.

Using Quantification: My first (naive) approach to avoid side-effects was to wrap the type with any givens from the implications and quantifying any free type variables, which meant that any effects on the variables only affected fresh variables introduced by the type checker during simpli-

fication. However, this approach rejected some valid hole fits and accepted some invalid hole fits since the type `forall a. a` is not equivalent to `a` in most cases.

Using a Wrapper: The current approach to avoid side-effects uses a wrapper that restores flexible meta type variables back to being flexible after the operation has been run, reverting any side-effects on those variables.

2.4 An Additional Application

The reason I started looking into valid hole fits for typed-holes was to be able to interact with libraries of functions annotated with non-functional properties.

A Library of Sorting Algorithms annotated with computational complexity and memory complexity is one example. We can define a type to represent simple asymptotic polynomials for a simplistic encoding of big O notation:

```
{-# LANGUAGE TypeInType, TypeOperators, TypeFamilies,
             UndecidableInstances, ConstraintKinds #-}
module ONotation where

import GHC.TypeLits as L
import Data.Type.Bool
import Data.Type.Equality

-- Simplistic asymptotic polynomials
data AsymP = NLogN Nat Nat

-- Synonyms for common terms
type N     = NLogN 1 0
type LogN  = NLogN 0 1
type One   = NLogN 0 0

-- Just to be able to write it nicely
type O (a :: AsymP) = a

type family (^.) (n :: AsymP) (m :: Nat) :: AsymP where
  (NLogN a b) ^. n = NLogN (a L.* n) (b L.* n)

type family (*.) (n :: AsymP) (m :: AsymP) :: AsymP where
  (NLogN a b) *. (NLogN c d) = NLogN (a+c) (b+d)

type family OCmp (n :: AsymP) (m :: AsymP) :: Ordering where
  OCmp (NLogN a b) (NLogN c d) =
```

```
    If (CmpNat a c == EQ) (CmpNat b d) (CmpNat a c)

type family OGEq (n :: AsymP) (m :: AsymP) :: Bool where
  OGEq n m = Not (OCmp n m == 'LT)

type (>=.) n m = OGEq n m ~ True
```

We can now annotate a library of sorting functions to use O notation to convey complexity information:

```
{-# LANGUAGE TypeInType, TypeOperators, TypeFamilies,
             TypeApplications #-}
module Sorting ( mergeSort, quickSort, insertionSort
               , Sorted, runSort, module ONotation) where

import ONotation
import Data.List (insert, sort, partition, foldl')

-- Sorted encodes average computational and auxiliary
-- memory complexity. The complexities presented
-- here are the in-place complexities, and do not match
-- the naive but concise implementations included here.
newtype Sorted (cpu :: AsymP) (mem :: AsymP) a
               = Sorted {runSort :: [a]}

insertionSort :: (n >=. O(N^.2), m >=. O(One), Ord a)
              => [a] -> Sorted n m a
insertionSort = Sorted . foldl' (flip insert) []

mergeSort :: (n >=. O(N*.LogN), m >=. O(N), Ord a)
          => [a] -> Sorted n m a
mergeSort = Sorted . sort

quickSort :: (n >=. O(N*.LogN) , m >=. O(LogN), Ord a)
          => [a] -> Sorted n m a
quickSort (x:xs) = Sorted $ (recr lt) ++ (x:(recr gt))
  where (lt, gt) = partition (< x) xs
        recr = runSort . quickSort @(O(N*.LogN)) @(O(LogN))
quickSort [] = Sorted []
```

Using valid hole fits, we can then search the sorting library by specifying the desired complexity in the type of a hole to find functions with those properties (or better):

```
Valid hole fits include
  mergeSort :: forall (n :: AsymP) (m :: AsymP) a.
               (n >=. O (N *. LogN), m >=. O N, Ord a)
               => [a] -> Sorted n m a
  quickSort :: forall (n :: AsymP) (m :: AsymP) a.
               (n >=. O (N *. LogN), m >=. O LogN, Ord a)
               => [a] -> Sorted n m a
```

Figure 2.4: Valid hole fits found in GHCi version 8.6 for the hole in `_ [3,1,2] :: Sorted (O(N*.LogN)) (O(N)) Integer`

2.5 Related Work & Ideas

Hoogle is *the* type directed search engine for Haskell, and allows users to easily search all of Hackage for functions by type or name [12]. Hoogle, however, does not integrate with the type checker of GHC, and can have difficulties with handling complex types and type families. Hoogle uses data extracted from the Haddock generated documentation of packages [12], meaning that unexported functions in the current, local module and local bindings like function arguments and bindings defined in `let` or `where` clauses are not discoverable. For searching the Haskell ecosystem however, Hoogle remains unparalleled.

Program Synthesis: Finding valid hole fits can be considered a special case of type-directed program synthesis. **Djinn** is a program synthesis tool that generates Haskell code from a type, and can generate total functions rather than just single identifiers fom user provided types and functions [2]. **Synquid** is a command line tool and algorithm that can synthesize programs from polymorphic refinement types in an ML-like language [14]. Other program synthesis tools include **InSynth** and **Prospector** [6, 11], however none of these are integrated with a compiler or type checker of a language, but are rather stand-alone tools or IDE plugins.

PureScript: The valid hole fits as presented in this report are modeled on the type directed search that Hegemann implemented in PureScript as part of his Bachelor's thesis work [8]. In PureScript, the type directed search looks for matches when a typed-hole is encountered [8]. The valid hole fits as I have implemented them in GHC go further than those in PureScript in that the output is sorted, and additional arguments are available via refinement hole fits.

Agda: The typed-holes of GHC were originally inspired by Agda [7]. Agda is dependently typed, and thus can offer very specific matches. The emacs mode of Agda offers the `Auto` command to automatically fill a hole

with a term of the correct type, and the `Refine` command can split a hole into cases containing additional holes [1]. The dependent typing has the drawback that type inference is in general undecidable, and users must explicitly provide more types than required in Haskell [13].

Idris, like Agda, is dependently typed, and offers a proofsearch command that can construct terms of a given type [3]. Idris also has a type directed search command, but in Idris the command also gives (and denotes) matches with a more specific type, in addition to matches of the same or more general type [3]. This allows users to find functions that match `Eq a => [a] -> a` when searching for `[a] -> a`, even though it requires an additional constraint [3]. Idris does not integrate these commands with typed-holes.

2.6 Conclusion

As can be seen from the examples in this report, valid hole fits can be useful in many different scenarios. They can improve the user experience for Haskell programmers working with prelude functions like `foldl` or advanced features like `lens` or `TypeInType`. The implementation makes use of the already present type checking mechanisms of GHC, and integrates well with typed-holes in a non-intrusive manner. I believe it to be good addition to the typed-holes of GHC; it should help facilitate Type-Driven Development in Haskell.

I learned a great deal from this project. Extending GHC was certainly non-trivial, however, the modularity of GHC allowed me to reuse a lot of code and to focus on the *what* rather than the *how*. A few pitfalls were encountered (like type checking by side-effect), and while the documentation of GHC internals is not so great (being mostly spread around in comments and assuming a lot of knowledge from the reader), the community was very helpful to a newcomer.

2.6.1 Future Work

When working with typed-holes, a few issues come to light: **Too General Fits**: The types inferred by GHC are sometimes too polymorphic for the valid hole fits to be useful. One such example is if we consider the function `f x = (_+x)/5`. Here, GHC will happily infer the most general type, namely that `f :: Fractional a => a -> a`. A sensible hole fit for the hole in `f` is `pi :: Floating a => a`, but that would constrain `f` to the more specific type of `Floating a => a -> a`. If `f` is not explicitly typed, then `pi` should be a valid hole fit. However, `f` having a more specific type might invalidate

other code that uses `f`, if those uses are explicitly typed with a `Fractional` constraint and not a `Floating` constraint. We would like to suggest such hole fits, for example by including a list of more specific hole fits, such as offered by Idris [3].

Built-in Syntax: Functions that are built-in syntax are not considered as candidate hole fits, since they are not in the global reader. However, functions like `(,)`, `[_]`, and `(:) :: a -> [a] -> [a]` are very common, and suggesting them would improve the user experience. Since these functions are syntax, they are not "in scope" in the global reader and no list of these functions is defined in GHC, making the addition of built-in syntax candidates non-trivial. One solution would be to hard-code these as candidates.

Functions with Fewer Arguments: There is no way to find functions that take in fewer arguments than required, and users must resort to binding the arguments (with e.g. `(\x -> _)`) in order to find these suggestions. Considering lambda abstractions as candidates could improve this case.

Specifying Behavior: It can be hard to choose which fit to use when multiple fits with the right type but different behaviors are suggested. Being able to hint to GHC how the function should behave would allow us to discard wrong hole fits. One approach would be integrating the valid hole fits with something like the refinement types of Liquid Haskell:

```
{-@ isPositive :: x:Int -> {v:Bool | v <=> x > 0} @-}
```

in which users can specify invariants for behavior [15].

2.6.2 Current Status

My contributions to GHC have been accepted. A basic version of the valid hole fits is in GHC version 8.4, an improved version with sorting, refinement hole fits and local binding suggestions in GHC version 8.6, and on GHC HEAD, a version is available with a flag to display documentation for hole fits in the output (to explain opaque function names). All code is available in the `TcHoleErrors` module in GHC.

Bibliography

[1] Agda Contributors. Agda Documentation 2.5.3, 2017.

[2] L. Augustsson. The Djinn package, 2014.

[3] E. Brady. *Type-Driven Development with Idris*. Manning Publications Company, 2017.

[4] GHC Contributors. GHC 8.2.1 users guide, 2017.

[5] GitHub. The Open Source Survey, 2017.

[6] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI '13, Proc. the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–38. ACM, 2013.

[7] Haskell Wiki Contributors. Typed holes in GHC, 2014.

[8] C. Hegemann. Implementing type directed search for PureScript. BSc. Thesis, University of Applied Sciences, Cologne, 2016.

[9] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2016.

[10] E. Kmett. The lens library, 2018.

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05, Proc. the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61. ACM, 2005.

[12] N. Mitchell. Hoogle overview. *The Monad. Reader*, 12:27–35, 2008.

[13] U. Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[14] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI '16, Proc. the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 522–538. ACM, 2016.

[15] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *ICFP '14, Proc. the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 269–282. ACM, 2014.

3

PropR: Property-Based Automatic Program Repair

Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands

*International Conference on Software Engineering 2022 (**ICSE '22**)*

Abstract. 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.

3.1 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 [52, 67], predictive-evaluation in search [73], and duplicate handling [9]. Other challenges are unique to the domain itself, such as deriving ingredients for a fix [41] and producing valid programs [28]. 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 [36, 42, 65], or by empirical evidence that fixes are often related to dependencies, not the code itself [4, 14]. 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. [54] 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 [34]. The fitness is defined as the number of failing tests in the suite [40], 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 [52]. This is considered an *overfit* repair attempt. A particularly good example for this is the Kali approach [54], that removes random statements of a program. In a later study, Martinez et al. [38] 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. [71] to address overfitting was to introduce tests generated with EvoSuite [15] 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 [8] 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 [34] 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* [19]. 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 [41], 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 paper is organized as follows: Section 3.2 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 3.3 we present the primary aspects of the repair tool and their reasoning. Section 3.4 presents the data used in the empirical study, and declares research questions and methodology. The results of the research questions are covered in Section 3.5 and discussed in Section 3.6. After the threats to validity in Section 3.7 we summarize the work in Section 3.8. The shared artifacts are described in Section 3.9.

3.2 Background and Related Work

3.2.1 Property-Based Testing

Property-based testing is a form of automated testing derived from random testing [22]. 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 [8]. 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 [8]. 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 [8].

There are some variations on property-based testing, e.g. SmallCheck, which performs an *exhaustive test* of the property [58]. QuickCheck approximates this behavior with a configurable number of random inputs (by default 100 random samples). Figure 3.1 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.

```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 3.1: Comparison of Properties and Unit Tests for sin

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* [52], 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.

3.2.2 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 [8] 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* [19]. 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* [19]. 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 3.2.

```
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 3.2: 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 [17], 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.

3.2.3 GenProg, Genetic Program Repair and Patch Representation

Search-based program repair centered mostly around the work of Le Goues et al. [34] 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* [32], 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. [41], 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 3.3).

Since GenProg, much has been done in genetic program repair [11] mostly for Java. Particularly Astor [39] enabled lots of research [61, 66, 69, 70] due to its modular approach, as well as real-world applications [59, 62]. 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 3.3.1).

3.2.4 Repair of Formally Verified Programs and 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 [26], cryptographic protocols [43] or lately smart contracts [6]. But formal verification has also been applied to the domain of program repair and synthesis [30, 60], 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 [20]. 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 [48, 56, 57, 64]. The existing approaches [21, 25, 50] 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* [51] 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 [24]. 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. [35] 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* [44] 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 [44], 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.

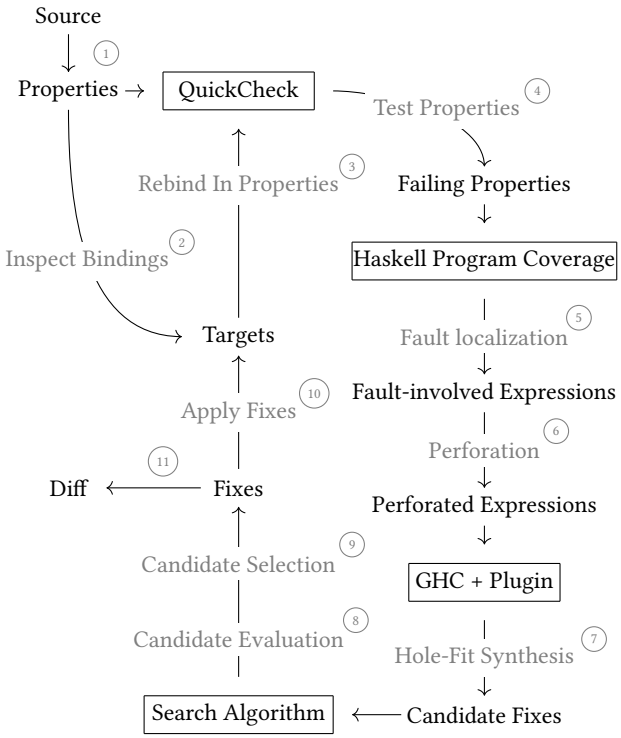


Figure 3.3: The PropR test-localize-synthesize-rebind loop

3.3 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 3.3.4 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 3.3. The circled numbers (n) in this section refer to the labels in Figure 3.3.

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 3.4.

```haskell
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 3.4: 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.

```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 3.5: The parametrized properties for len

3.3.1 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 bindings with either a type (`TestTree`) or name (`prop`) that indicates it is a test ①. We use GHC's ability to derive data definitions for algebraic data types [17] and the Lens library [27] 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`}.

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

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

Figure 3.6: The parametrized properties applied to a different implementation of `len`, the standard library `length`

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 3.4 in Figure 3.5.

The new properties in Figure 3.6, `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`:

```
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 [1], 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 [39, 40]. GHC's *Haskell Program Coverage* (HPC) can instrument Haskell modules and get a count of how many times each expression is evaluated during execution [18]. 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 coun-

terexample 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 counterexamples.

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 [19] ⑦. We present a few examples of the perforated versions of `len` in Figure 3.7.

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

3.3.2 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 3.8: 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 3.9: Candidate fixes derived from the valid hole-fits in Figure 3.8. The location refers to `product` in `len`

We derive hole-fit candidates directly from GHC's valid hole-fits, as seen in Figure 3.8, giving rise to the fixes in Figure 3.9. 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* [37]. 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.

3.3.3 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. 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 3.9 give rise to the definitions shown in Figure 3.10. 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 3.11, though we do additional work to extract the results in

```
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 3.10: New targets defined by applying the fixes in Figure 3.9 to the original `len`

```
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 len4), quickCheck (prop'_dup len4)]
 ,[quickCheck (prop'_abc len5), quickCheck (prop'_dup len5)]
 ,[quickCheck (prop'_abc len6), quickCheck (prop'_dup len6)]
 ,[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 3.11: Checking our new targets from Figure 3.10

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 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 3.11 that `len3` and `len7` pass all the properties, meaning that replacing `product` with `length` or `sum` qualifies as a repair for the program.

3.3.4 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 [39] or EvoSuite [15]. 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 [23] with an elitism-rate [3] 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. [53] 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 3.3) 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 3.4. The exhaustive and random search do not perform any rebinding.

3.3.5 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.

3.4 Empirical Study

3.4.1 Research Questions

Given the concepts presented in Section 3.3, 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:

```
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 3.12: The final result of our repair for `len`

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?

3.4.2 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 3.18) or about simplification (seen in Figure 3.13), 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: (A) it compiled (B) 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-datapoint 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.

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

Figure 3.13: A property asserting the idempotency of simplify

Table 3.1: 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

The anonymized data is provided in the reproduction package.

3.4.3 Methodology / Experiment Design

To evaluate RQ1 and RQ2 we perform a grid experiment on the dataset with the parameters presented in Table 3.1. 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 datapoint). Further information on the data collection can be found in the reproduction package.

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 [29, 45].

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 [55] 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 [49] 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 [63] 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 [68] 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. [46] 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 [54] and [38]. 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.

3.5 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 3.2 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 3.14 and Figure 3.15.

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

[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 3.2: 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

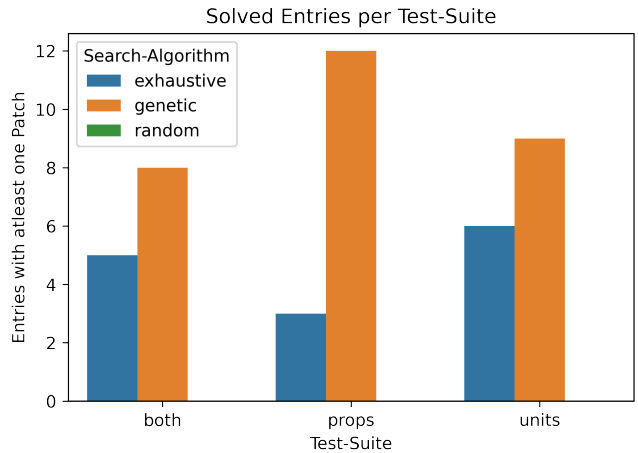


Figure 3.14: Solved Entries per Test-Suite and Algorithm

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 3.2).

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

Figure 3.15 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.

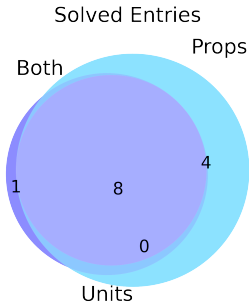


Figure 3.15: Venn-Diagram of Solved Entries per Suite

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.

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 3.16. 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.

[2]Wilcoxon-RankSum with *less*

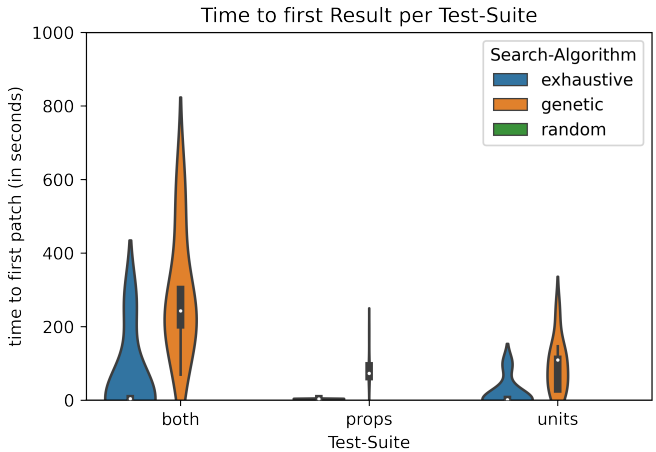


Figure 3.16: Distribution of Time to First Patch per Entry

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 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 [38] (13%) and better than GenProg [38](1-4%). Arja [72] 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 3.17. There is a single unit test (see Figure 3.18) 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

```
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 3.17: 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 3.18: The unit test corresponding to the fix in Figure 3.17

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 3.13, 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 uncovered due to the hyper-frequent evaluation. The only way to mit-

igate 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.

3.6 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 [53]. 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 3.5 only require Java programs with sufficient properties. However, the current Java-ecosystems are not utilizing properties; even less sophisticated JUnit-Features, such as parametrized 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 [47]. 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 [12, 16], 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 [40] or

[3] `https://github.com/pholser/junit-quickcheck`

ARJA [72] 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, 7] 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 counterexamples. 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 [14] 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.

3.7 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 [5]. 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 [11, 13, 31, 33]. A real-world study on program such as Pandoc [10] is part of future work. Pandoc, a popular Haskell document-converter, is rich in properties that test e.g., for symmetry over conversions.

[4] `https://pandoc.org/`
[5] `https://www.haskell.org/alex/`

3.8 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.

3.9 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`

Bibliography

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009. SI: TAIC PART 2007 and MUTATION 2007.

[2] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation, 2021.

[3] C. W. Ahn and R. Ramakrishna. Elitism-based compact genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 7(4):367–385, 2003.

[4] M. Alfadel, D. E. Costa, E. Shihab, and M. 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.

[5] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*, pages 33–47. Springer, 2011.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. PLAS '16, page 91–96, New York, NY, USA, 2016. Association for Computing Machinery.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang,

I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.

[8] K. Claessen and J. 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.

[9] Z. Y. Ding. Patch quality and diversity of invariant-guided search-based program repair. *arXiv preprint arXiv:2003.11667*, 2020.

[10] M. Dominici. An overview of pandoc. *TUGboat*, 35(1):44–50, 2014.

[11] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, 2019.

[12] T. Durieux and M. 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.

[13] T. Durieux and M. Monperrus. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Technical report, Universite Lille 1, 2016.

[14] K. Etemadi, N. Harrand, S. Larsen, H. Adzemovic, H. L. Phu, A. Verma, F. Madeiral, D. Wikstrom, and M. Monperrus. Sorald: Automatic patch suggestions for sonarqube static analysis violations. *arXiv preprint arXiv:2103.12033*, 2021.

[15] G. Fraser and A. 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.

[16] A. Ghanbari and L. Zhang. Prapr: Practical program repair via bytecode mutation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1118–1121, 2019.

[17] GHC Contributors. GHC 8.10.4 users guide, 2021.

[18] A. Gill and C. 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.

[19] M. P. 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.

[20] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using sat. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[21] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[22] R. Hamlet. Random testing. *Encyclopedia of software Engineering*, 2:971–978, 1994.

[23] J. H. Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[24] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova. Digging for fold: Synthesis-aided api discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[25] S. Katayama. Magichaskeller: System demonstration. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*, page 63, 2011.

[26] C. Kern and M. R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, apr 1999.

[27] E. Kmett. The lens library, 2021.

[28] X. Kong, L. Zhang, W. E. Wong, and B. 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.

[29] R. Koschke. Survey of research on software clones. In R. Koschke, E. Merlo, and A. 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.

[30] C. Kreitz. Program synthesis. In *Automated Deduction—A Basis for Applications*, pages 105–134. Springer, 1998.

[31] C. Le Goues, Y. Brun, S. Forrest, and W. Weimer. Clarifications on the construction and use of the manybugs benchmark. *IEEE Transactions on Software Engineering*, 43(11):1089–1090, 2017.

[32] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[33] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[34] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[35] J. Lee, D. Song, S. So, and H. Oh. Automatic diagnosis and correction of logical errors for functional programming assignments. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[36] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. 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.

[37] D. Mandelin, L. Xu, R. Bodík, and D. 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.

[38] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[39] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of ISSTA*, 2016.

[40] M. Martinez and M. Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, 151:65–80, 2019.

[41] M. Martinez, W. Weimer, and M. 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.

[42] E. Mashhadi and H. Hemmati. Applying codebert for automated program repair of java simple bugs. *arXiv preprint arXiv:2103.11626*, 2021.

[43] C. A. Meadows. Formal verification of cryptographic protocols: A survey. In *International Conference on the Theory and Application of Cryptology*, pages 133–150. Springer, 1994.

[44] S. Mechtaev, J. Yi, and A. 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.

[45] G. Neumann, M. Harman, and S. Poulding. Transformed vargha-delaney effect size. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, pages 318–324, Cham, 2015. Springer International Publishing.

[46] A. Nilizadeh, G. T. Leavens, X.-B. D. Le, C. S. Păsăreanu, and D. 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.

[47] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[48] R. Peña. An introduction to liquid haskell. *arXiv preprint arXiv:1701.03320*, 2017.

[49] T. Pohlert. The pairwise multiple comparison of mean ranks package (pmcmr). *R package*, 27(2019):9, 2014.

[50] N. Polikarpova, I. Kuraj, and A. 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.

[51] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama. Liquid information flow control. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.

[52] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189, 2013.

[53] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[54] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[55] M. Raymond and F. Rousset. An exact test for population differentiation. *Evolution*, 49(6):1280–1283, 1995.

[56] P. Redmond, G. Shen, and L. Kuper. Toward hole-driven development with liquid haskell. *arXiv preprint arXiv:2110.04461*, 2021.

[57] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[58] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *Acm sigplan notices*, 44(2):37–48, 2008.

[59] R. K. Saha, Y. Lyu, H. Yoshida, and M. 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.

[60] S. Srivastava, S. Gulwani, and J. 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.

[61] C. Trad, R. Abou Assi, W. Masri, and F. Zaraket. Cfaar: Control flow alteration to assist repair. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 208–215. IEEE, 2018.

[62] S. Urli, Z. Yu, L. Seinturier, and M. 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.

[63] A. Vargha and H. 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.

[64] N. Vazou, L. Lampropoulos, and J. Polakow. A tale of two provers: Verifying monoidal string matching in liquid haskell and coq. *SIGPLAN Not.*, 52(10):63–74, sep 2017.

[65] K. Wang, R. Singh, and Z. Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[66] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172*, 2017.

[67] Q. Xin. Towards addressing the patch overfitting problem. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 489–490, 2017.

[68] Q. Xin and S. 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.

[69] Q. Xin and S. 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.

[70] H. Ye, M. Martinez, T. Durieux, and M. Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, 171:110825, 2021.

[71] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *arXiv preprint arXiv:1703.00198*, 2017.

[72] Y. Yuan and W. Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *arXiv*, 46(10):1040–1067, 2017.

[73] Q. Zhu, A. Panichella, and A. 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.

4

Spectacular: Finding Laws from 25 Trillion Programs

Matthías Páll Gissurarson, Diego Roque, and James Koppel

*International Conference on Software Testing 2023 (**ICST '23**)*

Abstract. We present Spectacular, a new tool for automatically discovering candidate laws for use in property-based testing. By using the recently-developed technique of ECTAs (Equality-Constrained Tree Automata), Spectacular improves upon previous approaches such as QuickSpec: it can explore vastly larger program spaces and start generating candidate laws within 20 seconds from a benchmark where QuickSpec runs for 45 minutes and then crashes (due to memory limits, even on a 256 GB machine). Thanks to the ability of ECTAs to efficiently search constrained program spaces, Spectacular is fast enough to find candidate laws in more generally typed settings than the monomorphized one, even for signatures with dozens of functions.

4.1 Introduction

Testing is the art of checking that a program works in some scenarios in order to gain evidence that it works in all. This is especially relevant in the age of LLMs, where establishing ground-truth to verify auto-generated programs is important. In its basic form, a developer must manually create a set of sample inputs and the expected behavior on each. For 20 years, the Haskell community has boasted their ability to automate this by writing down just a few properties, and letting a *property-based testing* tool [8] generate the inputs automatically. But this only replaces hard labor with hard thought: it is still difficult to think of the right properties.

Yet every program implies its set of properties. By generating and testing a vast number of properties that might hold for a given program, a developer need merely select from the smörgåsbord that does hold. This is the idea of QuickSpec, capable of generating interesting properties on numerous data structures starting with just a list of functions to consider — so long as that list is small. Beyond the single digits, the exponential growth overwhelms its search abilities.

We introduce Spectacular, a new tool for automatically discovering program properties which uses advances in program synthesis to search spaces of candidate programs which are orders-of-magnitude larger than can be searched by QuickSpec. For example, for the 5 functions and 3 constants in Figure 4.1, Spectacular finds 60 laws compared to QuickSpec's 28; even running in a restricted mode, it still finds more laws than QuickSpec in less than half the time. It does this thanks to its use of the recently-introduced ECTA (Equality-Constrained Tree Automata) data structure [24], capable of compactly representing a space of trillions of possible programs, and efficiently enumerating all the ones which are well-typed (or satisfy any other property encodable with equality constraints). The benefits over QuickSpec get larger for larger modules. Thanks to ECTAs and our custom enumeration, Spectacular can start generating laws within minutes from the space of all terms up to size 6 on a signature with 92 functions and constants, a

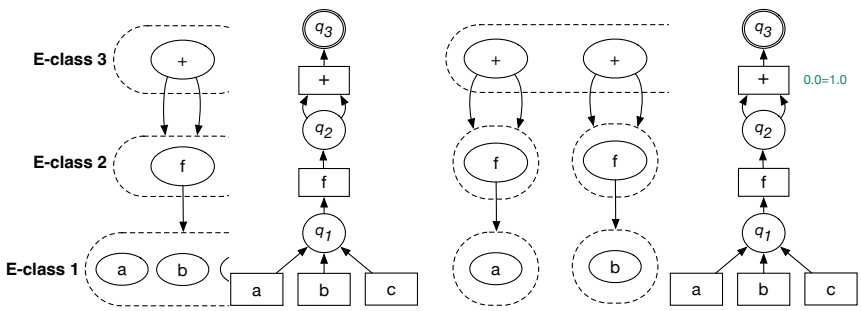
```
main = tacularSpec [
  con "reverse" (reverse :: [a] -> [a]),
  con "++" ((++) :: [a] -> [a] -> [a]),
  con "[]" ([] :: [a]),
  con "map" (map :: (a -> b) -> [a] -> [b]),
  con "length" (length :: [a] -> Int),
  con "concat" (concat :: [[a]] -> [a]),
  con "0" (0 :: Int),
  con "1" (1 :: Int) ]
```

Figure 4.1: Example signature for Spectacular, presentation slightly simplified.

```
reverse (reverse xs) == xs
reverse (xs ++ ys) == reverse ys ++ reverse xs
map f (concat lists) == concat (map (map f)) lists)
length (xs ++ ys) == length xs + length ys
```

Figure 4.2: Example laws generated by Spectacular from the signature in figure 4.1.

space of over 25 trillion terms, with memory consumption never exceeding 1GB. QuickSpec, on the same benchmark gets stuck enumerating terms of size 4 and crashes from memory exhaustion after 45 minutes on a machine with 256GB of RAM. And it does all this in only 1400 LOC, compared to QuickSpec's 8300.



(a) E-graph for $\mathcal{T}$ **(b)** FTA for $\mathcal{T}$ **(c)** E-graph for $\mathcal{U}$ **(d)** ECTA for $\mathcal{U}$

Figure 4.3: Representations of $\mathcal{T} = \{\mathsf{f}(t_1)+\mathsf{f}(t_2)\}$ and $\mathcal{U} = \{\mathsf{f}(t)+\mathsf{f}(t)\}$, where $t, t_1, t_2 \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

In summary, this paper makes the following contributions:

- An automated program property discovery approach based on modern program synthesis techniques, specifically ECTAs.

- The Spectacular tool, capable of discovering program properties orders-of-magnitude faster than previous approaches on large examples. The source is available at `https://zenodo.org/record/7565003` [18], along with scripts used to generate the benchmarks in the evaluation. Note that Spectacular itself is in the spectacular sub-folder.

4.2 Background

4.2.1 Equality-Constrained Tree Automata (ECTAs)

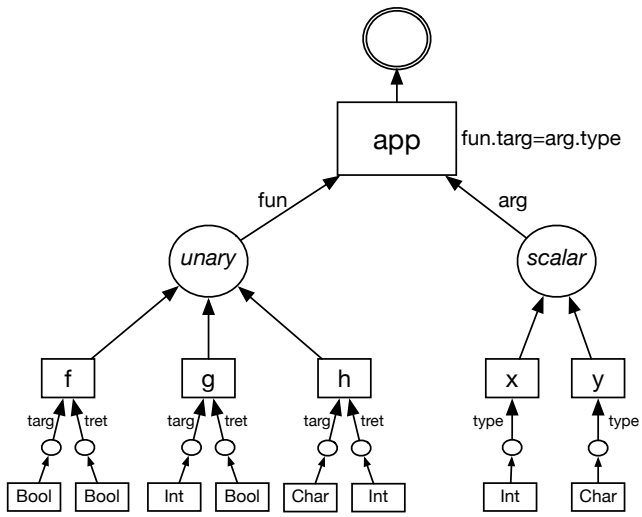


Figure 4.4: ECTA representing all well-typed size-two terms in the environment $\Gamma_1 = \{x : \texttt{Int}, y : \texttt{Char}, f : \texttt{Bool} \to \texttt{Bool}, g : \texttt{Int} \to \texttt{Bool}, h : \texttt{Char} \to \texttt{Int}\}$.

Equality-constrained tree automata (ECTAs) [24] are a new data structure for representing and enumerating a large space of terms with constraints between subterms. They are most easily explained as an alternative to e-graphs [13, 32, 44] suitable when different subterms cannot be chosen independently. We begin with a brief discussion of e-graphs; see Willsey et al [44] for additional background. We will then give an abridged version of the explanation of ECTAs from [24].

E-graphs = Independence.

Consider selecting a term from some large space of possibilities, where each successive node from the top down is a distinct choice. E-graphs are a compact representation of such spaces where each choice can be made

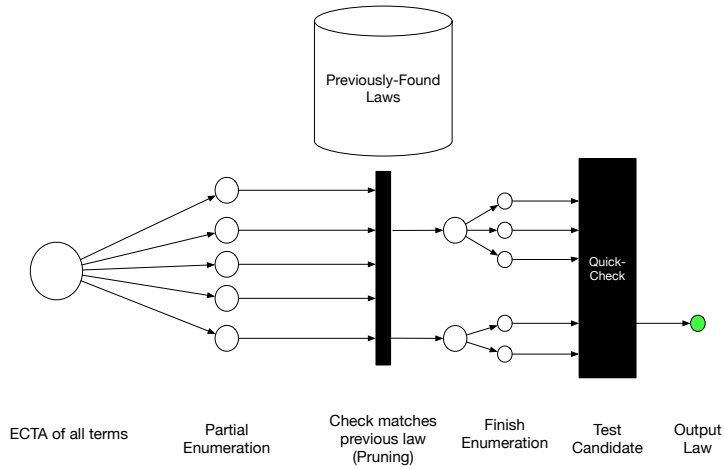


Figure 4.5: An overview of Spectacular

independently. For instance, consider the space $\mathcal{T} = \{\mathsf{f}(t_1) + \mathsf{f}(t_2)\}$ where $t_1, t_2 \in C = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. An e-graph can be constructed by first constructing a node "E-class 1" representing the choice $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, then a node "E-class 2" representing $\{\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{b}), \mathsf{f}(\mathsf{c})\}$, then a node "E-class 3" which sums two independent choices drawn from E-class 2. Figure 4.3a depicts this e-graph. Though the size of this space is clearly quadratic in $|C|$, the size of its e-graph is linear. Thanks to this ability, e-graphs have seen application from program synthesis [31, 44] to superoptimization [45] to semantic code search [35] to theorem proving [13]. E-graphs are now known [23, 33] to be equivalent representationally to *finite tree automata* (FTAs). Figure 4.3b shows an FTA that represents the same term space as the e-graph in figure 4.3a.

Tree Automata.

A finite tree automaton (FTA) consists of *states* (circles) and *transitions* (rectangles), with each transition connecting zero or more states to a single state. Intuitively, FTA transitions correspond to e-graph nodes, and FTA states correspond to e-classes. Importantly, both data structures, along with the similar version space algebras [34], thrive on spaces where terms share some top-level structure, while their divergent sub-terms can be chosen *independently* of each other.

Challenge: Dependent Joins. Consider now the term space $\mathcal{U} = \{\mathsf{f}(t) + \mathsf{f}(t)\}$, where $t \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, that is, a sub-space of $\mathcal{T}$ where both arguments to f *must be the same term.* Such "entangled" term spaces arise naturally in many domains. For example, in term rewriting or logic programming, we want to represent the subset of $\mathcal{T}$ that matches the non-linear pattern $X + X$. More relevantly, we want to represent the space of well-typed Haskell terms,

where, in each application, the type of an argument must equal the parameter type of the function.

Existing data structures are incapable of fully exploiting shared structure in such entangled spaces. Figure 4.3c shows an e-graph representing $\mathcal{U}$: here, the + cannot be reused because its children are *independent*, whereas our example requires a dependency between the two children of +.

Solution: ECTA. ECTAs address this problem by representing dependent spaces by tree automata whose transitions can be annotated with *equality constraints*. For example, figure 4.3d shows an ECTA that represents the term space $\mathcal{U}$. It is identical to the FTA in figure 4.3b save for the constraint $0.0 = 1.0$ on its + transition. This constraint restricts the set of terms accepted by the automaton to those where the sub-term at path 0.0 (the first child of the first child of +) equals the sub-term at path 1.0 (the first child of the second child of +). The constraint enables this ECTA to represent a dependent join while still fully exploiting shared structure, unlike the e-graph in figure 4.3c.

Most importantly, ECTAs come equipped with **efficient algorithms for enumerating all terms that satisfy the constraints** based on constraint processing via automata intersection, made available in the optimized ECTA library.

Type-driven Synthesis. A striking example of the power of ECTAs is HECTARE [24], the main existing application of ECTAs. HECTARE is designed as a replacement for HOOGLE+ [20], which solves the problem of *polymorphic type-driven program synthesis*: given a Haskell type such as `(a -> a) -> a -> Int -> a`, intended to take a function f and apply it n times to some input x, it synthesizes a small Haskell program of this type, included the intended solution `\g x n -> foldr ($) x (replicate n g)`. HOOGLE+ clocks in at over 4,000 lines, using an SMT encoding specifically tuned for this problem. HECTARE is a measly 400 lines: it constructs an ECTA using the ECTA library, and simply runs the standard enumeration procedure. And yet HECTARE is 8× faster.

The SPECTACULAR tool in this paper uses a similar encoding to HECTARE to build an ECTA representing the space of well-typed Haskell programs of a given signature, but tuned to the set of types occurring in the functions of interest, and uses finer-grained enumeration to reduce the number of candidate terms inspected by over 1000× on some benchmarks.

Figure 4.4 shows an example of a simplified ECTA for types, where type variables are restricted to just one of a few base types. We refer to the ECTA paper [24] for discussion of this encoding and its generalization to arbitrary polymorphism.

4.2.2 QuickCheck and the QuickSpec problem

QuickCheck

is a property-based testing framework that uses observational equivalence based on generating arbitrary data and testing to establish the validity of properties [8].

QuickSpec

is a state-of-the-art theory exploration system for Haskell that uses QuickCheck to automatically generate laws based on a set of functions called a *signature* [40]. Originally a naive equation generation system, QuickSpec V2 and onwards uses sophisticated techniques based on enumerating terms and *schemas* in order to quickly explore a system of equations [40]. We use QuickSpec as the gold standard to compare against in this paper, and many of the techniques we use in Spectacular are based on the techniques used in QuickSpec, albeit augmented with ecta. However, QuickSpec does struggle with large signatures, as we detail further in section 4.4.

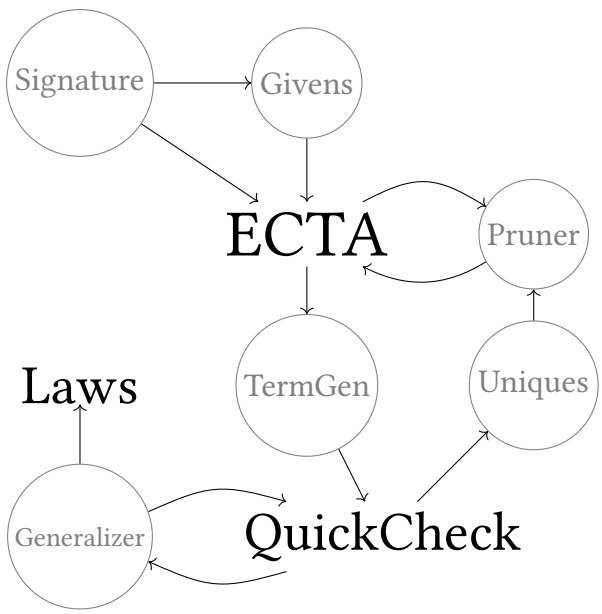


Figure 4.6: An overview of the Spectacular loop

```
Laws according to Haskell (==):
--------------------------------
1. 0 == length []
2. [] == reverse []
3. length xs0_<[A]> == length (reverse xs0_<[A]>)
4. xs0_<[A]> == reverse (reverse xs0_<[A]>)
5. [] == ((++) []) []
6. xs0_<[A]> == ((++) []) xs0_<[A]>
7. xs0_<[A]> == ((++) xs0_<[A]>) []
8. [] == (map f0_<A -> A>) []
9. f1_<A -> A> x0_<A>
 ==  f1_<A -> A> (f1_<A -> A> (f1_<A -> A> x0_<A>))
10. length xs0_<[A]>
 == length ((map f0_<A -> A>) xs0_<[A]>)
11. (map f0_<A -> A>) (reverse xs0_<[A]>)
 == reverse ((map f0_<A -> A>) xs0_<[A]>)
12. length (((++) xs0_<[A]>) xs1_<[A]>)
 == length (((++) (reverse xs0_<[A]>)) xs1_<[A]>)
13. reverse (((++) xs0_<[A]>) xs1_<[A]>)
 == ((++) (reverse xs1_<[A]>)) (reverse xs0_<[A]>)
14. ((++) xs0_<[A]>) (((++) xs1_<[A]>) xs2_<[A]>)
 == ((++) (((++) xs0_<[A]>) xs1_<[A]>)) xs2_<[A]>
15. ((++) (reverse xs0_<[A]>)) xs1_<[A]>
 == reverse (((++) (reverse xs1_<[A]>)) xs0_<[A]>)
Fully monomorphic phase finished..199 terms examined.
43 unique terms discovered.
Starting phase with more types....
Monomorphic phases finished..335 terms examined.
50 unique terms discovered.
Starting mono-polymorphic phase....
16. [] == concat []
17. ((++) (concat xs0_<[[A]]>)) (concat xs1_<[[A]]>)
 == concat (((++) xs0_<[[A]]>) xs1_<[[A]]>)
18. xs0_<[[A]]> == (map ((++) [])) xs0_<[[A]]>
19. xs0_<[[A]]>
 == (map reverse) ((map reverse) xs0_<[[A]]>)
Mono-polymorphic phase done! 1662 terms examined.
96 unique terms discovered.
Starting fully-polymorphic phase....
20. (map f0_<A -> B>) (concat xs0_<[[A]]>)
 == concat ((map (map f0_<A -> B>)) xs0_<[[A]]>)
Done! 3558 terms examined.
100 unique terms discovered
```

Figure 4.7: Example Spectacular output for terms up to size 5 in 4 phases for the signature in figure 4.1. in 3.3sec/39MB

4.3 The Spectacular tool

The design of Spectacular is inspired by QuickSpec's [40] approach of property discovery by finding and maintaining a set of unique terms which are used for comparison with newly enumerated terms. We emulate QuickSpec's signature interface for ease of comparison. We use an ECTA to efficiently represent and enumerate the space of well-typed terms of increasing size that are compared to each other to find properties. We make non-trivial changes to the enumeration of the ECTA to efficiently *prune redundant terms*, reducing the amount of terms to be examined. An overview of Spectacular can be found in figure 4.5, with an overview of the loop itself in figure 4.6. We denote Haskell data types with red text, a la `Type`. An example output of Spectacular is provided in figure 4.7. Here, `x_Int` denotes an arbitray `Int`, `xs_[A]` an arbitrary list of `A`s, and so on.

4.3.1 Signatures

Users interact with Spectacular by defining a signature that describe the system of equations that they want to explore. We give an example signature consisting of a collection of list functions in figure 4.1.[1] This shall serve as our running example. Note that Spectacular is capable of handling of much larger spaces, so it can handle entire modules instead of just manually-specified signatures.

4.3.2 Supplying Givens

To generate and check equations, Spectacular needs to be able compare values of a given type for equality and to generate arbitrary values of that type. When compiling Haskell with GHC, this is done by instance resolution during compile time to find the corresponding arbitrary data generators and equality functions, and insert them into the generated code. However, since Spectacular is generating and evaluating equations at *runtime*, the associated instances must be resolvable at runtime as well. This means that Spectacular requires a runtime mechanism for instance resolution for arbitrary data generators and equalities. This can be done using the `Dynamic` datatype, allowing us to look up and store the associated instances in a data structure at compile-time, and defer the selection of which instances to use to runtime. This means that it can test terms without having to resort to code generation, making the tool simpler and faster. Spectacular does this by looking at the

[1] The pseudocode in this figure is almost-identical to the real code, which uses a mechanism to create runtime representations of types and type variables.

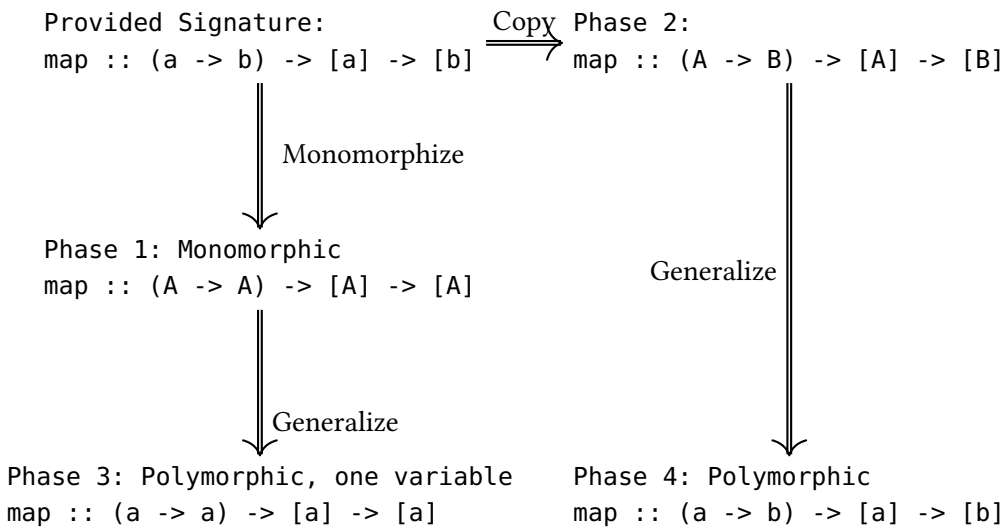


Figure 4.8: The four phases of SPECTACULAR. Each consecutive phase uses laws and uniques found in the previous phases, but is slightly more general than the last when it comes to the types it explores.

signature and generating its *universe of types* [40], meaning all function arguments and return types. for figure 4.1, these would be `[A]`, `[[A]]`, `[B]`, `(A -> B)`, and `Int`. This universe is used to generate the *givens* for the signature, which are the equality functions and random variable generators for types in the universe which can then be looked up at runtime. Generating the givens at compile-time from the universe of types means that the users do not have to manually provide these instances, though it comes with the caveat that types that are not in the signature are not considered synthesis targets.

4.3.3 Enumerating Terms

An efficient way to explore the space of equations is to enumerate *terms* instead of equations themselves [40]. However, it is important to enumerate only *well-typed* terms. For example, for the terms `reverse :: [a] -> [a]` and `(++) :: [a] -> [a] -> [a]` with added givens `xs :: [a]` and `ys :: [a]`, there are 5460 possible programs of size ≤ 6 of which only 128 are well-typed and 84 are base values!

ECTAs

To efficiently represent the space of well-typed Haskell terms of a given type, Spectacular constructs an ECTA for a fixed size n. The ecta library provides an efficient interface for enumeration. It conceptually provides the following API:

```
partiallyEnumerate :: ECTA -> PartiallyEnumeratedTerm
```

where a partially-enumerated term can be thought of as a template like `map _ (reverse _)`, where the two underscores stand for ECTAs representing some smaller space of terms[2]. This allows Spectacular to decide whether to expand or discard a branch (see section 4.3.4).

Phasing

Ordering is important when generating laws. As a simple example, if the tool discovers `reverse (reverse x) == x` early, large swaths of the search space need not be enumerated at all, and undesired redundant rules such as `tail (reverse (reverse x)) == tail x` will not be generated. To increase efficiency, and to provide the user results immediately, Spectacular splits its search into several *phases*, each using a larger search space than the previous. Each phase is further stratified by size, guaranteeing that smaller terms are discovered before larger. An overview is provided in figure 4.8.

1. **Monomorphic**: The first phase monomorphizes all types representing type variables, so that `b`, `c`, and `d` all become the concrete type `A`, where `A` is an arbitrary constant type. For example, the function `map :: (a -> b) -> [a] -> [b]` from figure 4.1 is monomorphized into `map :: (A -> A) -> [A] -> [A]`. This yields a very small search space, enumerable in seconds. Searching this space first allows Spectacular to discover rewrites that allows aggressive pruning in later phases. This phase can discover rules such as `reverse (reverse xs) == xs`.

2. **Uninterpreted**: The second phase replaces all type variables with distinct constant types (such as `A` and `B`). This means that `map :: (a -> b) -> [a] -> [b]` is specialized into `map :: (A -> B) -> [A] -> [B]`. This phase does not generally result in many *laws*, but does discover a few *unique*, previously unseen terms from the new types, such as `snd (x_A,x_B) :: B`, used in later phases.

[2]Partially-enumerated terms also store equality constraints between the unenumurated parts; in this case, that the argument type of `map`'s first argument must match the element type of the list in the second argument.

3. **Single-variable polymorphism**: This phase replaces all type variables with the single type variable `a`, but treats this variable as standing for an arbitrary type. For example, `map :: (a -> b) -> [a] -> [b]` becomes `map :: (a -> a) -> [a] -> [a]`. Unlike in the previous phases, this time it contains a proper type variable, `a`. This is the approach taken in QUICKSPEC [40], and allows SPECTACULAR to find most of the laws, unless they require more than one type variable. An example law first discovered at this phase is `reverse (concat reverse xs) == concat (map reverse xs)`.

4. **Arbitrary polymorphism** (optional): This phase generalizes all the type variable representing types in the signature into type variables. This means that `map` will have the polymorphic type `map :: (a -> b) -> [a] -> [b]`. This phase is not run by default because of the vast size of the search space. When running this phase on large term sizes, progress grinds to a halt. An example of a law first discovered at this phase is `concat (reverse (map reverse lists)) == reverse (concat lists)`. Finding this is harder in a monomorphic setting: when `map :: (a -> b) -> [a] -> [b]` becomes monomorphized to `map :: (a -> a) -> [a] -> [a]`, the argument function `f` must have the same input and output type, `f :: a -> a`. But the function `concat :: [[a]] -> [a]` returns a different type than its input.

4.3.4 Pruning

The key to efficient exploration of the equation space is the pruning that SPECTACULAR applies *directly* in the ECTA during enumeration. As described in section 4.3.3, ECTA enumeration is based on a loop that does repeated expansion of partially-enumerated terms, which are terms for which there are still some choices to be made. The enumeration runs in a monadic environment that captures the branching that happens during enumeration when choices are made. This means that It is helpful to avoid exploring branches known to only contain terms containing a sub-term that can be rewritten. As an example, if the tool has previously discovered that `x == reverse (reverse x)`, it can discard any partially-enumerated terms containing `reverse (reverse _)`, since any such term is equivalent to the smaller term containing just `x`. Choosing the right pruning strategy is important, we want to be as aggressive as possible while still being sound, in order to find all the relevant equations without having to enumerate and check an intractable amount of terms. In SPECTACULAR, the pruning strategy

is based on matching the non-unique terms that Spectacular has discovered up until that point (i.e. terms that are equivalent to any of the unique terms) with the partial terms being enumerated. These non-unique terms are either direct rewrites of an expression (such as `reverse [] => []`), or include a variable, e.g. `xs ++ [] => xs`.

Matching partial terms

A partial term matches another term when the top-level symbol are the same and all of their sub-expressions are the same. However, it might be the case that some sub-expressions of the partial term are not enumerated yet. In that case, Spectacular *suspends* the pending sub-matches, and runs them whenever the pending partial term gets enumerated, allowing the pruning of the branch as soon as Spectacular knows enough about any of the partial terms to make the decision that this branch will not be productive. Enumeration of a term always starts from the top-down, so most of the time the pending matches are suspended on any of the sub-expressions of the top-level term. When a match is found, Spectacular immediately stops enumeration of that branch, and continues with the next one.

4.3.5 Interleaving Testing and Enumeration

For freshly generated terms, Spectacular starts by trying to rewrite it to a smaller term using the previously-discovered equations. Since it explores the space of terms in order of size, the existence of a rewrite to a smaller term means that an equivalent term has already been inspected, and thus any laws using the larger term would be redundant. The larger term can thus be discarded. A lot of the performance comes from being able to discard a term before it is tested or even before it is generated. By interleaving testing and enumeration [40], Spectacular can learn rewrites that allow it to prune more aggressively, making it a lot more efficient than generating all the terms at once and then start testing. In Spectacular, this is done by generating terms in batches, and generating and testing all terms of a given *type* and size before proceeding to the next one. This allows us to learn all the rewrites for a certain size before proceeding to a bigger size, and so Spectacular can prune aggressively when we know a sub-term has a rewrite. Generating per type is also beneficial, since Spectacular might generate the same expression at a different type multiple times (e.g. `[] == [] ++ [] :: [a]`, `[] ++ [] :: [Int]`, etc.). By learning the rewrite for one type and generalizing it, Spectacular can prune those expressions at other types.

4.3.6 Testing of Terms

Spectacular tracks a set of *unique terms* of each type that are not morally equivalent [12] to any other term Spectacular has encountered so far. When a new candidate term has been generated, Spectacular tests it against all the other unique terms we've discovered of that type. Let `xs :: [a]` be the unique term and `xs ++ []` be the candidate term. By using the generated equality instance Spectacular can construct the term representing the equality `xs == xs ++ []`. Once we have a term, we have to turn it into a `Property` that we can test using QuickCheck. Using the equality and random variable generators from the given we generated from the signature, we have all the components of the term represented as `Dynamic` instances. This means that we do not need to do any compilation or code-generation step, we can immediately use the random variable generators to generate variable assignments, and apply the dynamic representations of the functions and equality to generate the `Property`.

Implementation

To do this, Spectacular generates a `Dynamic` containing a `Property`. It then generates a variable assignment for every variable in the expression, making sure that the same variable gets the same value, no matter how many copies there are in the expression. A specific GADT is needed to represent the `Dynamic` generator, since we must ensure that the generated values are `Typeable`, so that we can wrap them in `Dynamic` once they've been assigned. The function then generates the `Dynamic` representation of each of the subexpressions, by looking up the value in the variable assignment map or looking up the `Dynamic` representation of the function from the signature. To support the monomorphization of the type variables, we need to use a function that circumvents the checks done by `Dynamic` at runtime, so that e.g. `A` can be used in place of `B`. `A` and `B` and other type variable representatives are defined as `data A = A Any`, and so can be safely coerced between. Since we only synthesize *well-typed* expressions, we know these coercions are safe. By generating a `Dynamic` representation of the property this way, we can efficiently test equations for validity.

4.3.7 Generalization of Laws

To reduce the search space even further, Spectacular only enumerates terms with the one variable for each type, and reuses that variable whenever a variable of that type is needed. This means that when we generate the associativity check, it will be discovered as `(xs ++ xs) ++ xs == xs ++ (xs ++`

`xs)` This is based on the observation that [40] if an equation holds for any arbitrary `xs` and `ys`, it must in particular also hold whenever `xs == ys`. This means it suffices to explore terms with however many copies of the *same* variable, (`xs` in our example) and then to generalize the law once found. To generalize a law, Spectacular generates all possible variations of the law by renaming each variable and adding more variables as needed. This would generalize the trivial law `(xs ++ xs) ++ xs == xs ++ (xs ++ xs)` to

- `(xs ++ xs) ++ ys == xs ++ (xs ++ ys)`,
- `(xs ++ ys) ++ xs == xs ++ (ys ++ xs)`, and also
- the actual law: `(xs ++ ys) ++ zs == xs ++ (ys ++ zs)`.

We make sure to generate these laws so that the variables are always in the same order to avoid duplicates and remove the ones that are equivalent up to renaming, We then test the most general law first (the one with the most variables) and so on until we find a law that hold (if none is found, the original law is the most general). This way we use variables as a limited form of *schemas* [40]. When a law has been discovered, its most general form is reported, and Spectacular continues until all the phases are finished for all types and type constructors.

4.4 Evaluation

Spec	Tool	Time (s)	Memory	Laws
Lists	Spectacular (P2)	3.54	21.4 MB	32
	QuickSpec	8.55	100.2 MB	28
	Spectacular (P3)	55.44	88.7 MB	73
	Spectacular (P4)	105.25	155.34 MB	93
Octonions	Spectacular (P2)	0.63	19.9 MB	8
	Spectacular (P3)	0.76	20.9 MB	8
	Spectacular (P4)	0.92	21.1 MB	8
	QuickSpec	0.97	20.3 MB	15
Regex	Spectacular (P2)	10.64	16.9 MB	42
	Spectacular (P3)	14.83	17.2 MB	42
	Spectacular (P4)	19.5	17.3 MB	42
	QuickSpec	458.9	88.9 MB	64
ListMonad	Spectacular (P2)	0.90	15.3 MB	8
	Spectacular (P3)	1.85	22.7 MB	8
	QuickSpec	2.44	30.5 MB	11
	Spectacular (P4)	53.56	213.7 MB	42
HugeLists (3)	Spectacular (P2)	0.30	11.7 MB	22
	Spectacular (P3)	0.68	15.8 MB	27
	Spectacular (P4)	2.12	29.2 MB	33
	QuickSpec	251.4	160 MB	49
HugeLists (4)	Spectacular (P2)	0.73	13.8 MB	37
	Spectacular (P3)	4.43	35.9 MB	68
	Spectacular (P4)	91.13	125.3 MB	90
	QuickSpec	>3600	12.3 GB	-
HugeLists (5)	Spectacular (P2)	2.92	23.4 MB	66
	Spectacular (P3)	83.7	106.9 MB	144
	Spectacular (P4)	>3600	301.3 MB	-
	QuickSpec	>3600	12.7 GB	-
HugeLists (6)	Spectacular (P2)	20.70	40.2 MB	99
	Spectacular (P3)	3164.75	434.4 MB	211
	Spectacular (P4)	>3600	434.4 MB	-
	QuickSpec	-	-	-
HugeLists (7)	Spectacular (P2)	201.51	54.6	186
	Spectacular (P3)	>3600	383.2	-
	Spectacular (P4)	>3600	381.4	-
	QuickSpec	-	-	-

Table 4.1: Performance of Spectacular against QuickSpec. Here (Pn) refers to until what phase we run Spectacular. Note that we did not attempt running QuickSpec on HugeLists (6) and (7), due to the timeout already being hit in (5). A timeout is denoted with (>3600), and the maximum resident memory of the process up to that point is given, e.g. 12.3 GB for QuickSpec on HugeLists (4). The benchmarks used are taken from QuickSpec, generated using the benchmark script in `https://zenodo.org/record/7565011`

We evaluate the performance of Spectacular against QuickSpec. We match the parameters when possible in both tools. We take the examples from the QuickSpec repository, including two shown in the paper, and adapt them to Spectacular. This involves removing any QuickSpec specific options from the signatures, and adding implementations of random data generators of the user-provided types defined by the example so that we can run QuickCheck.

These tests were run on a cloud-based machine with 32GB of RAM and 6 Intel Xeon E312xx @2GHz 64bit vCPUs.

The signatures we consider are as follows:

- Lists: The signature shown in figure 4.1, repeated here. All involving basic list functions. This has 6 components.

```
main = tacularSpec [
  con "reverse" (reverse :: [a] -> [a]),
  con "++" ((++) :: [a] -> [a] -> [a]),
  con "[]" ([] :: [a]),
  con "map" (map :: (a -> b) -> [a] -> [b]),
  con "length" (length :: [a] -> Int),
  con "concat" (concat :: [[a]] -> [a]),
  con "0" (0 :: Int),
  con "1" (1 :: Int) ]
```

- Octonions: This example defines an octonion data type and an Arbitrary instance for it. The signature is then defined as below, and has 3 components:

```
main = tacularSpec [
 con "*" (* :: Oct -> Oct), -- product
 con "inv" (recip :: Oct -> Oct) -- inverse
 con "1" (1 :: Oct)] -- identity
```

- Regex: This example defines a Regex algebra, including an equality based on NFAs, and the signature contains the standard Kleene operations. This has 7 components.

```
main = tacularSpec [
con "char" (Char :: Sym -> Regex Sym),
con "any" (AnyChar :: Regex Sym),
con "e" (Epsilon :: Regex Sym),
con "0" (Zero :: Regex Sym),
con ";"
 (Concat :: Regex Sym -> Regex Sym -> Regex Sym),
con "|"
```

```
 (Choice :: Regex Sym -> Regex Sym -> Regex Sym),
con "*" (star :: Regex Sym -> Regex Sym)]
```

- ListMonad: This signature contains the basic monad functions instantiated for List, as well as concatenation. This has 4 components.

```
main = tacularSpec [
 con "return" (return :: A -> [A]),
 con ">>=" ((>>=) :: [A] -> (A -> [B]) -> [B]),
 con "++"  ((++) :: [A] -> [A] -> [A]),
 con ">=>"
  ((>=>) :: (A -> [B]) -> (B -> [C]) -> A -> [C]) ]
```

- HugeLists: The benchmark from the QuickSpec paper, mentioned in the abstract, consisting of 33 list functions from `Prelude`, ranging from standard functions such as `length`, to more exotic functions like `(>=>)`, as well as some internal functions from QuickSpec like `usort` that uses a different implementation of sort. See figure 4.11 for the spec itself.

On both tools we look for terms of size up to 7, unless a particular size is specified in parenthesis next to the signature. We stopped the execution in the experiment at 3600 seconds, though on a different machine QuickSpec *did* finish for HugeList (4) in 38 minutes, whereas HugeLists (4) P4 took 9 seconds on the same machine.

4.4.1 Improvements upon QuickSpec

Performance

As seen in table 4.1 and figure 4.10, Spectacular is generally faster than QuickSpec, and consistently using less memory. This difference is more stark when scaling the size of terms we look for and the size of the signature, as we can see in the HugeLists benchmark.

Due to different patterns of exploration, Spectacular and QuickSpec sometimes disagree on the number of laws. One example is in HugeLists, Spectacular finds that `sort` from the prelude is the same as `usort` from the QuickSpec internals used in the benchmarks. Spectacular thus discards any laws that mention `usort` in favor of `sort`, whereas QuickSpec does not, and reports additional laws involving `usort`. However `usort == sort` is not a *true* equivalence, since `usort` discards duplicates! In this case, the underlying arbitrary data generator does not generate duplicate elements so

Spectacular reports this as a law. This highlights the fact that care must be taken to interpret the "laws" only in the context of their generators.

The memory improvement as seen in figure 4.9, in particular on the examples with larger signatures, makes it feasible to run on bigger sizes in a memory constrained environment, like cloud instances where memory rather than time is more expensive. It also shows that the limiting factor for Spectacular is time and not memory requirements.

Running Spectacular until phase 2 gives adequate performance, with the trade-off being fewer laws. Each subsequent phase is more expensive but often returns new kinds of laws. We also see that for the cases where polymorphism isn't present (Octonions and Regex), the performance penalty with respect to phase 2 is reasonable. A non-trivial law like `length (concat xs) == sum ((map length) xs)` (HugeLists (4)) can be discovered in < 30 seconds by Spectacular but takes an hour or more in QuickSpec. Being able to control which parts of the type space to explore makes users of Spectacular able to adapt the search to their specific requirements.

Scalability

As stated before, the biggest difference in performance happens with HugeLists. Under similar parameters to QuickSpec, Spectacular completes its search in less time and with less memory. This gives evidence that Spectacular scales better than QuickSpec. In particular, the memory consumption does not blow up like in QuickSpec. In a different test, done with a x2gd.4xlarge AWS instance[3], Spectacular generated properties for HugeLists (6) within half a minute for P2. On the other hand, QuickSpec ran out of memory while generating terms of size 4 after 45 minutes. For P2, Spectacular even manages to finish for the default term size 7, whereas QuickSpec does not return any properties for terms bigger than 4.

Differences in Discovered Laws

QuickSpec does have an advantage when it comes to the *heuristics* it uses during exploration, allowing them to quickly find laws like commutativity, associativity and distributivity, using hard-coded heuristics during the search phase and generalizing templates such as `(_ + _)` [40]. This is particularly important for specifications such as the Octonions, where these form a majority of the laws, allowing QuickSpec to find laws beyond the term size using these heuristics as a guide. These are laws with multiple variations

[3] x2gd.4xlarge instances have 16 2.5GHz vCPUs and 256GB of RAM

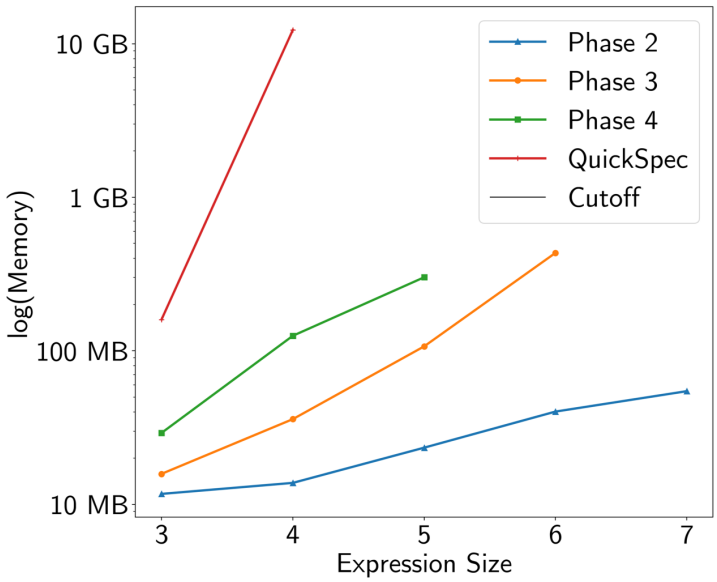


Figure 4.9: HugeList memory use

like `(x*x)*y = (x*(x*y))` and `(x*y)*x = x*(y*x)`, whereas SPECTACULAR finds only `x*(x*y) = (x*x)*y`. SPECTACULAR does perform better for cases like Lists, where SPECTACULAR finds laws such as `reverse (concat xss) == concat (map reverse (reverse xss))`, `concat (concat xsss) == concat (map concat xsss)`, and `map length xss == map length (map (map f) xss)` which QUICKSPEC does not, though the difference in could be explained by the tactics used for search space enumeration and merging of laws.

4.5 Related Work

The story of QUICKSPEC and its offshoots is often told narrowly: it extends property-based testing. But automatically discovering useful laws has far greater reach. In this section, we discuss both immediately-related work in property-based testing and program synthesis, and similar techniques used in math, physics, and other parts of computer science.

Property-Based Testing (PBT) Property-Based Testing [16] is the checking of software correctness by finding *properties* that should hold of a correct implementation and gathering evidence they hold, often by random testing [8]. It has vast research literature and multiple industrial libraries [3, 28].

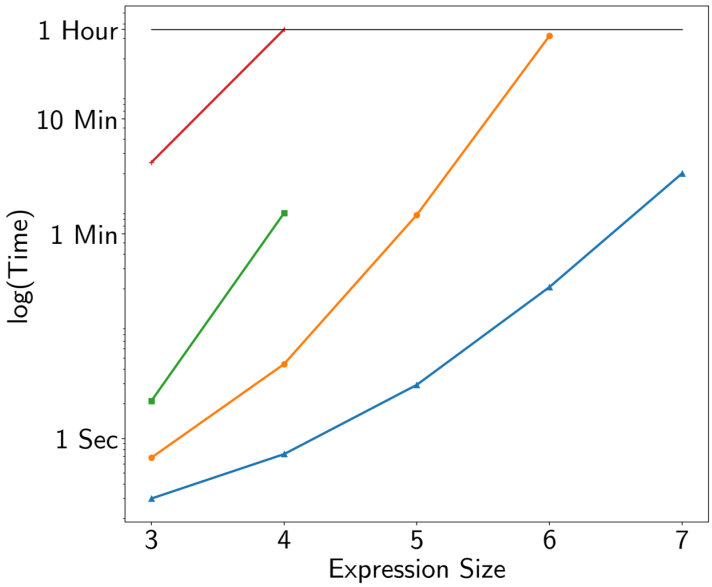


Figure 4.10: HugeList compute time

In Haskell, it was popularized by QuickCheck [8], and adopted as a golden standard for testing libraries.

Synthesis of PBT Properties The pioneer in the problem of automatically generating properties for property-based testing is QuickSpec [10, 40]. It has been applied to lemma discovery in automated theorem proving [9, 22], and extended with mutation-testing [6] and the ability to discover inequalities and conditional laws [7, 41]. Variants have also been implemented using comparison of symbolic rather than concrete terms [2, 4].

Data-Driven Invariant Generation In pure functional programming, data from random testing can only be collected about the final output of a term. In imperative programming, such data can also be collected about the intermediate states of a function, and used to suggest invariants that hold at that point. This is the idea of Daikon [14].

Daikon has spawned a massive amount of follow-up research as well as three for-profit companies (most notably Agitar [5]). Of special relevance, Daikon-like techniques have been used to discover properties used in program verification, namely loop invariants [38] and simulation relations for equivalence checking [39]. Do note that Daikon-like techniques are primarily restricted to properties that hold of a single function, while property-

```
main = tacularSpec [
 con "length" (length :: [A] -> Int),
 con "sort" (sort :: [Int] -> [Int]),
 con "scanr" (scanr :: (A -> B -> B) -> B -> [A] -> [B]),
 con "succ" (succ :: Int -> Int),
 con ">>=" ((>>=) :: [A] -> (A -> [B]) -> [B]),
 con "snd" (snd :: (A, B) -> B),
 con "reverse" (reverse :: [A] -> [A]),
 con "0" (0 :: Int),
 con "," ((,) :: A -> B -> (A, B)),
 con ">=>" ((>=>) :: (A -> [B]) -> (B -> [C]) -> A -> [C]),
 con ":" ((:) :: A -> [A] -> [A]),
 con "break" (break :: (A -> Bool) -> [A] -> ([A], [A])),
 con "filter" (filter :: (A -> Bool) -> [A] -> [A]),
 con "scanl" (scanl :: (B -> A -> B) -> B -> [A] -> [B]),
 con "zipWith" (zipWith :: (A -> B -> C) -> [A] -> [B] -> [C]),
 con "concat" (concat :: [[A]] -> [A]),
 con "zip" (zip :: [A] -> [B] -> [(A, B)]),
 con "usort" (usort :: [Int] -> [Int]),
 con "sum" (sum :: [Int] -> Int),
 con "++" ((++) :: [A] -> [A] -> [A]),
 con "map" (map :: (A -> B) -> [A] -> [B]),
 con "foldl" (foldl :: (B -> A -> B) -> B -> [A] -> B),
 con "takeWhile" (takeWhile :: (A -> Bool) -> [A] -> [A]),
 con "foldr" (foldr :: (A -> B -> B) -> B -> [A] -> B),
 con "drop" (drop :: Int -> [A] -> [A]),
 con "dropWhile" (dropWhile :: (A -> Bool) -> [A] -> [A]),
 con "span" (span :: (A -> Bool) -> [A] -> ([A], [A])),
 con "unzip" (unzip :: [(A, B)] -> ([A], [B])),
 con "+" ((+) :: Int -> Int -> Int),
 con "[]" ([] :: [A]),
 con "partition" (partition :: (A -> Bool) -> [A] -> ([A], [A])),
 con "fst" (fst :: (A, B) -> A),
 con "take" (take :: Int -> [A] -> [A]) ]
```

Figure 4.11: The HugeLists benchmark from QuickSpec we use to compare Spectacular and QuickSpec when there are many terms in scope. Note that from this spec, Spectacular also adds additional generators for the types involved, and constants such as empty lists of various types.

based testing is primarily concerned with hyperproperties/hypersafety, comparing multiple programs.

Symbolic Regression Symbolic regression [37] is the problem of finding the best mathematical formula to fit a dataset. It has been used to generate equations defining mathematical constants [36], physical laws [25], and conjectures over generalized integers [15]. Notable recent work exploits symmetry and learned features for inductive bias, discovering a great number of famous physics formulas. [42, 43].

Of less relevance to this work is the field sometimes called Automated Theorem Discovery, developing systems which propose mathematical theorems by means other than data [11, 26, 27, 29, 30].

Enumerative Program Synthesis Both Spectacular and QuickSpec are *enumerative program synthesizers*, employing both application-specific and standard techniques from this field. Gulwani et al [19] gives a review of this area.

4.6 Conclusion

Spectacular is an efficient tool for discovering laws for property-based testing, and has the potential to scale to settings where law discovery has previously been intractable, such as settings with generalized types. In doing so, we hope to continue to grow the usability of property-based testing. Beyond testing, the recent development of ECTA-based synthesis techniques hints at great leaps in the general usability of synthesis outside limited domains, and their simple implementation promises easy integration into more tools.

Future Work

Spectacular is a recent development, and there are still many avenues to explore using Spectacular.

Creating generators on-the-fly

One of the challenges for Spectacular is that it is unaware of recursive generators, e.g. `Arbitrary [a] => Arbitrary [[a]]` can be derived from `Arbitrary a => Arbitrary [a]`, and so on. In the current implementation, these must be generated and made available in the ECTA for terms that require a list of arbitrary depth, e.g. `concat (concat xs) == concat (map concat xs)`, which requires a generator `xs :: [[[a]]]`. Currently,

this is done by generating instances specifically for lists during initialization, and these instances added to the signature. However, integrating the available type-classes (and specifically the generators of arbitrary data) into the ECTA itself would be more efficient, as the current implementation of adding a list type for every type in the signature makes the search space a lot bigger.

Synthesizing non-equations and implications

Theory-exploration usually focuses on tautologies such as equations, but properties often only hold for a subset of the domain such as positive integers. ECTAs are good for encoding such dependencies between premises and conclusions and should excel at synthesizing such implications and other non-equations.

Efficient node-based pruning

Branch-pruning reduces the number of inspected terms by orders of magnitude, but most of the time is still spent on expanding unification variables to enumerate out the next term instead of testing of terms. But there are tree-automata algorithms that can eliminate all undesired terms before even beginning enumeration [1], which we hope to extend to ECTAs.

More directed enumeration

The ECTA-based technique allows a lot finer control over how the program space is enumerated, and the simplicity of the ECTA allows our implementation to do a lot more exploration on which branches to select in order to generate more valuable laws (i.e. more general ones first, etc.). Exploring how to control the enumeration from the outside to direct it towards parts likely to contain laws is an exciting avenue of research, and offering better enumeration heuristics could greatly speed up the current exploration. Of special interest is would be the ability to heuristically direct the enumeration towards such laws as associativity and commutativity that often hold for many data-structures.

Rapid exploration of modules

Spectacular is quite efficient at generating terms in a fully monomorphized setting, even for large signatures. This could allow users to rapidly explore the properties of a module without having to manually specify which functions are interesting in a signature. One issue however is how to generate

the dynamic instances and generators at runtime, though some combination of template haskell and using GHC as a library might be feasible.

Rapid on-demand generation of properties

Properties can be used to prevent overfitting in program repair, as well as help with fault localization [17]. Properties are scarce in the wild which limits the use of property-based repairs. Being able to rapidly generate properties for a module when it is stable can be of great use in program repair to fix small bugs that might creep in during development, and can serve as a checkpoint for a current state of a module and function as regression tests, ensuring that repairs are patching the issue and not meddling with the correctness of the whole system.

Efficiency of overall pipeline

While Spectacular uses an efficient data-structure for coming up with potential candidates to test, the overall efficiency of the pipeline could be improved, for example by improving the testing part by storing values to quickly reject false properties in a manner similar to QuickSpec [40]. For testing whole modules, determining the "interesting" parts of the modules will be important.

Applicability beyond Haskell

The speed of Spectacular is highly dependent on the type system of Haskell, which allows us to massively restrict the search space for valid terms and test only well-typed programs. In languages such as Python, the amount of "well-typed" terms becomes harder to model. However, with recent additions such as type hints, an ECTA based approach to synthesizing Python programs might be possible. In general, languages that allow random data-generation and have some constraints on the "shape" of valid terms will admit techniques similar to Spectacular, though care must be taken to accurately model and/or sandbox side-effects, but this has been done for both C and Erlang [21].

Acknowledgements

We want to thank Moa Johansson and Nicholas Smallbone from the QuickSpec team for answering our questions and helping us understand its limitations, and John Hughes for his excellent feedback. This work was partially

supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation.

Bibliography

[1] M. D. Adams and M. Might. Restricting grammars with tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–25, 2017.

[2] M. Alpuente, M. A. Feliú, and A. Villanueva. Automatic Inference of Specifications Using Matching Logic. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, pages 127–136, 2013.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10, 2006.

[4] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for First Order Curry Programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 25–34, 2012.

[5] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, 2006.

[6] R. Braquehais and C. Runciman. FitSpec: Refining Property Sets for Functional Testing. In *Proceedings of the 9th International Symposium on Haskell*, pages 1–12, 2016.

[7] R. Braquehais and C. Runciman. Speculate: Discovering Conditional Equations and Inequalities about Black-Box Functions by Reasoning from Test Results. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 40–51, 2017.

[8] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[9] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating Inductive Proofs Using Theory Exploration. In *International Conference on Automated Deduction*, pages 392–406. Springer, 2013.

[10] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *International Conference on Tests and Proofs*, pages 6–21. Springer, 2010.

[11] S. Colton, A. Bundy, and T. Walsh. On the Notion of Interestingness in Automated Mathematical Discovery. *International Journal of Human-Computer Studies*, 53(3):351–375, 2000.

[12] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. *ACM SIGPLAN Notices*, 41(1):206–217, 2006.

[13] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[15] H. Ferguson, D. Bailey, and S. Arno. Analysis of PSLQ, An Integer Relation Finding Algorithm. *Mathematics of Computation*, 68(225):351–369, 1999.

[16] G. Fink and M. Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[17] M. P. Gissurarson, L. Applis, A. Panichella, A. van Deursen, and D. Sands. PropR: Property-Based Automatic Program Repair. In *The 44th IEEE/ACM In-ternational Conference on Software Engineering (ICSE)*. IEEE/ACM, 2022.

[18] M. P. Gissurarson, J. Koppel, E. de Vries, Z. Guo, and D. A. R. Montoya. Tritlo/spectacular: ICST 2023, Jan. 2023.

[19] S. Gulwani, O. Polozov, R. Singh, et al. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[20] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020.

[21] J. Hughes. Experiences with quickcheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*, pages 169–186. Springer, 2016.

[22] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen. Hipster: Integrating Theory Exploration in a Proof Assistant. In *International Conference on Intelligent Computer Mathematics*, pages 108–122. Springer, 2014.

[23] J. Koppel. Version Space Algebras are Acyclic Tree Automata, 2021.

[24] J. Koppel, Z. Guo, et al. Searching entangled program spaces. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2022.

[25] P. Langley. Data-Driven Discovery of Physical Laws. *Cognitive Science*, 5(1):31–54, 1981.

[26] D. B. Lenat. Automated Theory Formation in Mathematics. In *IJCAI*, volume 77, pages 833–842. Citeseer, 1977.

[27] D. B. Lenat and J. S. Brown. Why AM and EURISKO Appear to Work. *Artificial intelligence*, 23(3):269–294, 1984.

[28] D. R. MacIver, Z. Hatfield-Dodds, et al. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[29] R. McCasland, A. Bundy, and S. Autexier. Automated Discovery of Inductive Theorems. *Special Issue of Studies in Logic, Grammar and Rhetoric on Computer Reconstruction of the Body of Mathematics: From Insight to Proof: Festschrift in Honor of A. Trybulec*, 10(23):135–149, 2007.

[30] R. L. McCasland and A. Bundy. MATHsAiD: A Mathematical Theorem Discovery Tool. In *2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 17–22. IEEE, 2006.

[31] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[32] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.

[33] J. Pollock and A. Haan. E-Graphs Are Minimal Deterministic Finite Tree Automata (DFTAs) · Discussion #104 · egraphs-good/egg, 2021.

[34] O. Polozov and S. Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[35] V. Premtoon, J. Koppel, and A. Solar-Lezama. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1066–1082, 2020.

[36] G. Raayoni, S. Gottlieb, Y. Manor, G. Pisha, Y. Harris, U. Mendlovic, D. Haviv, Y. Hadad, and I. Kaminer. Generating Conjectures on Fundamental Constants with the Ramanujan Machine. *Nature*, 590(7844):67–73, 2021.

[37] M. Schmidt and H. Lipson. Distilling Free-Form Natural Laws from Experimental Data. *science*, 324(5923):81–85, 2009.

[38] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *European Symposium on Programming*, pages 574–592. Springer, 2013.

[39] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-Driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 391–406, 2013.

[40] N. Smallbone, M. Johansson, K. Claessen, and M. Algehed. Quick Specifications for the Busy Programmer. *Journal of Functional Programming*, 27:e18, 2017.

[41] C. Smith, G. Ferns, and A. Albarghouthi. Discovering Relational Specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 616–626, 2017.

[42] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark. AI Feynman 2.0: Pareto-Optimal Symbolic Regression Exploiting Graph Modularity. *Advances in Neural Information Processing Systems*, 33:4860–4871, 2020.

[43] S.-M. Udrescu and M. Tegmark. AI Feynman: A Physics-Inspired Method for Symbolic Regression. *Science Advances*, 6(16):eaay2631, 2020.

[44] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[45] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar. Equality Saturation for Tensor Graph Superoptimization. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 255–268, 2021.

5

CSI: Haskell – Tracing Lazy Evaluations in a Functional Language

Matthías Páll Gissurarson and Leonhard Applis

*Symposium on Implementation and Application of Functional Languages 2023 (**IFL '23**)*

Abstract. 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 happen 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.

5.1 Introduction

Problem and Motivation In crime scene investigation (CSI), establishing the proper sequence of the events constituting the crime is a key technique in solving cases. While less dramatic, programs can still die: after making it through compilation, 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 hard 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 [19].

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.

```haskell
module Main where
divs :: Int -> [Int]
divs n = go 2
  where go i | i == n = []
        go i = if d i
               then i:(go (i+1))
               else go (i+1)
        d i = n `mod` i == 0

smallestDiv n = head (divs n)

main :: IO ()
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.

```
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 non-invasive 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 [32]. The data consist of a selection of the `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 preprocessing 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. The 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 for 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), whereas in Haskell they are used for all computation. When the value of that expression is required, the thunk is evaluated and resolved into 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

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

result saved. This is then *shared* if the value is needed again at a later time, without requiring recomputation.

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") [12, 14]. 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 for 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. [7] investigate differences originating from different perspectives of bug reports. One finding 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. [6] 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 improvements of error messages. Among their primary findings relevant to this work are: students and programmers alike actually read error messages and stack traces [5], students are discouraged when error messages do not point toward the faults [28, 33, 38] and that cognitive load should be considered in the design and presentation of errors [24, 25, 31] (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. [29] 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. [2] and Marlow [19] tackled the first issues appearing 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 [2]. 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 fully sufficient to locate the fault. Manual annotations with `HasCallStack` are non-optimal for programmers (they should be removed in releases) 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. [36] compared three systems available in 2000: HAT [9], HOOD [13], and Freja [21]. Another system mentioned is Buddha [20]. 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* [35]. These types of checks are integrated via a GHC plugin [27], 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* [15]: 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 [10]. 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 [10], which extends the debugger HOOD [13] 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 [10]. 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 to be a debugger, but instead provides large-scale 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.

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 [18]. For

$$\textcircled{1}\quad \Gamma_0 \vdash \texttt{head}^2 \Downarrow \lambda \texttt{xs.\ head' xs}, \Gamma_1$$

$$\textcircled{2}\quad \frac{(\ldots)^6 \qquad (\ldots)^7}{\Gamma_3 \vdash \texttt{divs' n} \Downarrow \texttt{xs'}, \Gamma_{n-1}}$$

$$\textcircled{3}\quad \frac{\dfrac{\Gamma_1 \vdash \texttt{divs}^4 \Downarrow \lambda \texttt{n.\ divs' n}, \Gamma_2 \qquad \Gamma_2 \vdash \texttt{n}^5 \Downarrow \texttt{n'}, \Gamma_3 \ \texttt{where}\ [\![n = n']\!] \qquad \textcircled{2}}{\Gamma_1 \vdash (\texttt{divs n})^3 \Downarrow \texttt{xs'}, \Gamma_n \ \texttt{where}\ [\![xs = xs']\!]}}{\Gamma_1 \vdash \texttt{head' xs} \Downarrow \texttt{v}, \Gamma_n}$$

$$\frac{\textcircled{1} \qquad \textcircled{3}}{\Gamma_0 \vdash (\texttt{head (divs n)})^1 \Downarrow \texttt{v}, \Gamma_n}$$

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, while circled numbers refer to the formulas themselves (for presentation only).

Haskell, this evaluation has been linearized using implementations of machines such as Sestoft's lazy abstract machine [30], 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 "⇓" 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. [14]. 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 at runtime, 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 [14]. 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 as 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

$$[E_1, E_2, E_2, E_3, E_4, E_4, E_5, E_5, \ldots, E_3, E_1],$$

with the associated evaluation depths

$$[0, 1, 2, 1, 2, 3, 2, 3, 2, \ldots, 2, 1].$$

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 non-invasive 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 [3]. This has been explored with a short prototype by the authors and is feasible; however, 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 er-

```
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.

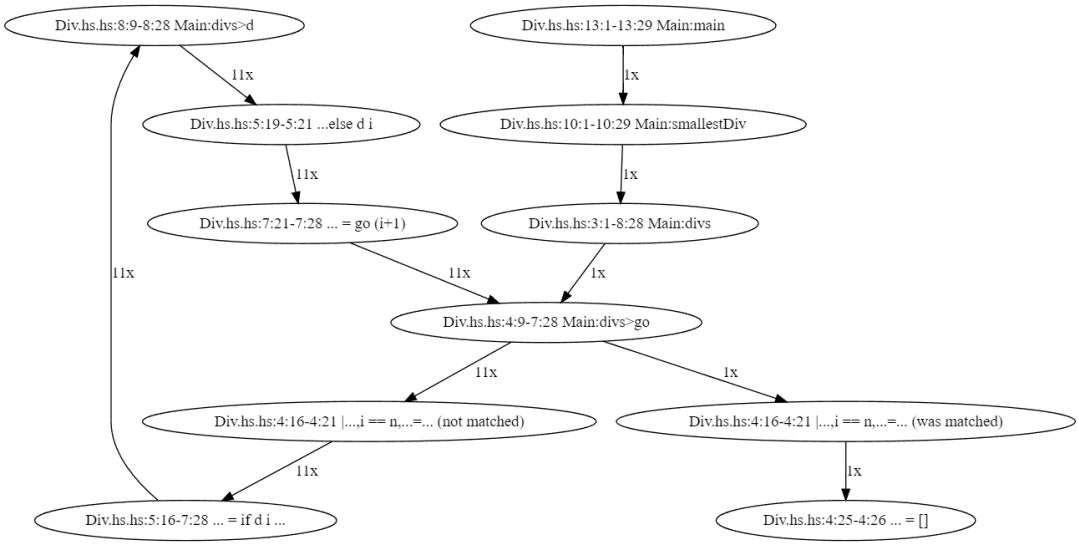


Figure 5.5: A graphical representation of the trace in Figure 5.4 generated by CSI: Haskell and an external script.

ror 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 rest as "previous expressions", whose evaluation was complete before the evaluation of the expression on top of the call stack started, and thus 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 [16, 22], 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 a suggestion 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 [32]. In this data set, Silva introduced artificial bugs of various categories to the data points of the `nofib` benchmark [23] 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

Error Type \ `nofib-buggy`	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 outlined in Section 5.3. After obtaining crash logs, two authors looked at each log separately, deriv-

ing 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 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. However, 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.

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 [8] summarizing the distribution of the calculated time deltas for all data points. It presents a smooth

[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

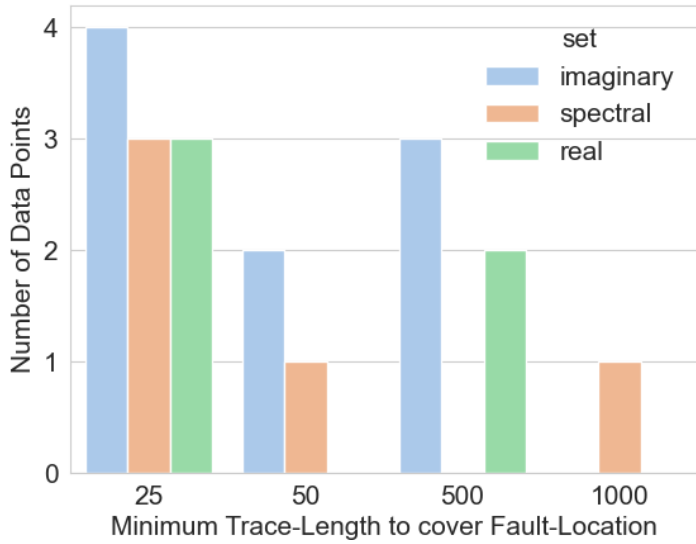


Figure 5.6: Minimum trace length to cover the error

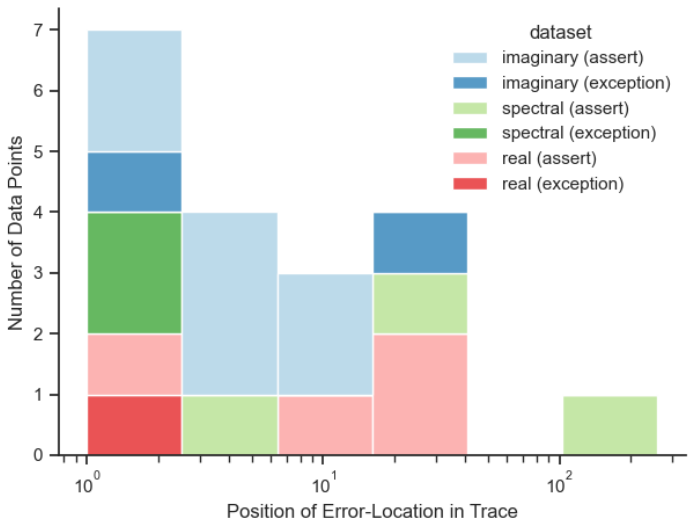


Figure 5.7: Histogram of position in the trace by data set and error type. Note that the x-axis is logarithmic.

growth of wall-clock time for increasing trace length, with the majority of data points needing between ~100% and ~300% longer. We can also observe

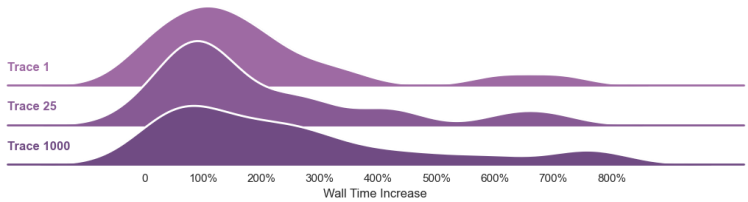


Figure 5.8: Kernel density estimate plot of increased compute time with varying trace lengths

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 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 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.

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

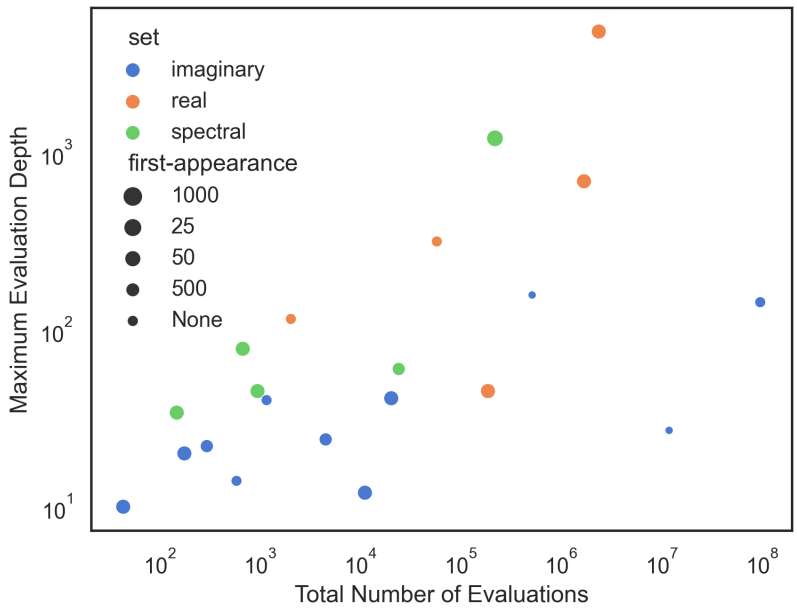


Figure 5.9: Distribution of maximum evaluation depth and total number of evaluations

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 for Tic-Tac-Toe but instead performing a mini-min. 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.

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.

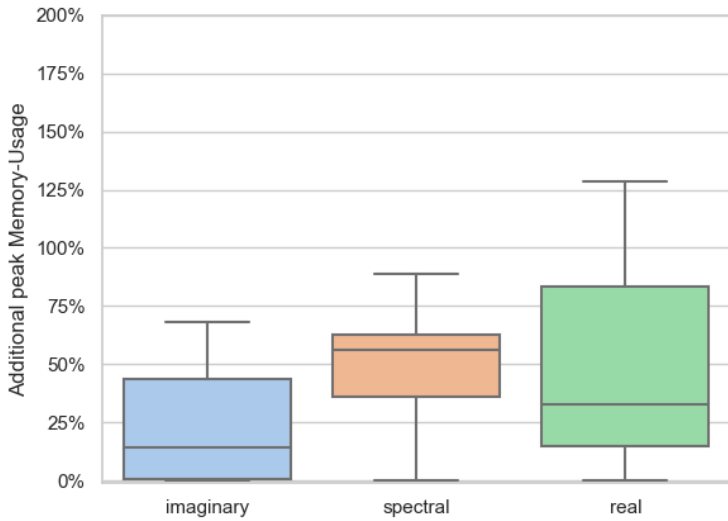


Figure 5.10: Additional memory usage per data set for a trace length of 1000

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

[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].

of only 50 locations per module is sufficient in 62% of the cases. When 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 that, unless necessary for follow-up analysis, capturing the last evaluations is the favorable approach, with a little fine-tuning in trace length depending on the number of evaluations.

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`

```
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'`.

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. [39]): 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.

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 [34].

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.

The current evaluation traces are also limited by *sharing* [18]. 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 a list. There is an error in this function that causes the ks to be invalid out-of-bound indices

```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

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

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

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, as discussed in 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 scare developers away from reading the error messages[28]. and time spent at the wrong places is a waste and reduces the trust towards traces and error messages[6, 33]. 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 [6, 26], 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 [11] 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 A definite 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 [4]. 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 the trace data, there are opportunities to accumulate data from multiple sources (test success, program coverage, etc.) and perform program slicing [37]. 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 thus 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[3] 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. However, 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. 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 [1, 17]. 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. However, this would involve more advanced termination checking than feasible for this paper, but would 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 paper presented 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 length 1000, with most locations appearing 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 three 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.

Acknowledgements

We thank David Sands for his input on evaluation trees and theory, and Matthew Sottile for his efforts on visualization and advice on the design of CSI: Haskell. We thank the attendants of the IFL workshop for their input, particularly the concept of evaluation asserts, as well as the attendants of the internal Chalmers talks. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation.

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. J. 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.

[2] T. O. Allwood, S. Peyton Jones, and S. 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.

[3] P. Anderson, L. Kot, N. Gilmore, and D. 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.

[4] L. Applis and A. 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.

[5] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. 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.

[6] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera, J. L. Pearce, and J. 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.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. 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.

[8] Y.-C. Chen. A tutorial on kernel density estimation and recent advances. *Biostatistics & Epidemiology*, 1(1):161–187, 2017.

[9] O. Chitil, C. Runciman, and M. Wallace. Transforming haskell for tracing. In *Symposium on Implementation and Application of Functional Languages*, pages 165–181, Madrid, Spain, 2002. Springer, Springer.

[10] M. Faddegon and O. Chitil. Algorithmic debugging of real-world haskell programs: deriving dependencies from the cost centre stack. *ACM SIGPLAN Notices*, 50(6):33–42, 2015.

[11] K. Ferdowsi. Towards human-centered types & type debugging. Plateau Workshop.

[12] GHC Contributors. GHC 8.10.4 users guide, 2021.

[13] A. Gill. Debugging haskell by observing intermediate data structures. *Electron. Notes Theor. Comput. Sci.*, 41(1):1, 2000.

[14] A. Gill and C. 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.

[15] W. T. Hallahan, A. Xue, M. T. Bland, R. Jhala, and R. 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.

[16] 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.

[17] T. Janssen, R. Abreu, and A. J. 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.

[18] J. 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.

[19] S. Marlow. Hiw 2012: Why can't i get a stack trace?, 2012.

[20] L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18:401–408, 1996.

[21] H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated software engineering*, 4:121–150, 1997.

[22] K. Noda, T. Kobayashi, T. Toda, and N. 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.

[23] W. 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.

[24] J. Prather, R. Pettit, K. McMurry, A. Peters, J. Homer, and M. Cohen. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 41–50, Espoo, Finland, 2018. ACM.

[25] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. 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.

[26] M. M. Rahman, S. Yeasmin, and C. 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.

[27] Ranjit Jhala. LiquidHaskell is a GHC Plugin, 2020.

[28] P. C. Rigby and S. 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.

[29] A. Schröter, N. Bettenburg, and R. 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.

[30] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[31] D. Shaffer, W. Doube, and J. 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.).

[32] J. Silva. The buggy benchmarks collection, 2007. Josep Silva self-published on his website / university.

[33] V. J. Traver. On compiler error messages: What they say and what they mean. *Adv. in Hum.-Comp. Int.*, 2010, jan 2010.

[34] M. Vassena, J. Breitner, and A. 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.

[35] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.

[36] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for haskell: a new hat. In R. 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).

[37] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, 1(4):352–357, 1984.

[38] J. Wrenn and S. 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.

[39] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. ESEC/FSE '09, page 43–52, New York, NY, USA, 2009. Association for Computing Machinery.

6

Functional Spectrums – Exploring Spectrum–Based Fault Localization in Functional Programming

Leonhard Applis, Matthías Páll Gissurarson, and Annibale Panichella

Manuscript

Abstract. 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 spectrums with types and AST structure. We introduce a *rule-based system* tailored to capture more complex attributes of spectrums. Spectrums are generated by adding an extra pass to the Tasty test framework, which allows easy adoption and reproducability. Through an empirical study involving 11 real-world programs, we investigate the generated spectrums along with the effectiveness of the rule-based system and their correlation to faults. For most bugs, conventional spectrum-based formulas perform promisingly well in a functional context and are only outperformed by classifiers that incorporate these formulas.

6.1 Introduction

Functional programming has earned a reputation for being at the forefront of type and programming language research, but we believe it can also be a champion of tooling and software engineering practices.

It remains open *what* tooling functional programmers really want, but tooling they need. Haskell is known for innovating in more niche features like software transactional memory and techniques like property-based testing. It has innovative tools like the Hoogle search engine, but we believe we can innovate further and introduce tools that would be harder to develop for other paradigms.

Tools are quite important to software developlent. According to modern developer surveys, approximately 50% of development is spent debugging, half of which is spent fixing bugs [3]. An important part of the debugging process is *fault localization*, i.e. determining *which* part of the program is at fault, which can be assisted by specialized tools. This challenge spans most software paradigms, including functional programming [10, 17].

One way to assist the fault localization process is to introduce automated tools [11, 24], for example, spectrum-based fault localization (SBFL). A *program spectrum* is a matrix where rows represent test results and columns represent code locations. Each entry indicates whether that location was involved in the test or not, with an additional column that indicates whether the test passed or failed. A program spectrum is created by running individual tests and collecting program coverage [27]; thus capturing different aspects of the program by branching over the test suite. Program spectrums have been successfully applied in imperative languages, 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. However, they have yet to become established in functional communities.

6.1.1 Example

Consider the function and properties in figure 6.1.

```
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 6.1: A buggy program and associated properties

Table 6.1: A spectrum for the code in figure 6.1

name	type	result	2:18	3:31	3:35-36	3:22-37	3:43	3:22-43	2:1-3:43
type identifier			Int	Int→Int→Int f	[Int] xs	Int	Int x	Int	
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

Here, we intended to implement a `fold`, 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 6.1, produces the spectrum in table 6.1. A standard spectrum consists of only the tests, whether they pass or fail, and the locations involved in each test. In this paper however, we produce *augmented* spectrums. These augmented spectrums 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.

$$\textbf{Tarantula: } \frac{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}}{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}+\frac{n_{e_p}}{n_{e_p}+n_{t_p}}} \quad \textbf{Ochiai: } \frac{n_{e_f}}{\sqrt{(n_{e_f}+n_{t_f})(n_{e_f}+n_{e_p})}}$$

Figure 6.2: Standard SBFL formulas. n_{e_f} and n_{e_p} are the number of times the expression e is involved in a failing or passing test respectivley, while n_{t_f} and n_{t_p} is how many total failing and passing tests there were.

Using the formulas detailed in figure 6.2 on the spectrum, we can score the locations as detailed in figure 6.3. Here, the most suspect location is

Table 6.2: A spectrum for the code in figure 6.1, 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			Int	Int	Int→Int→Int	Int	[Int]	Int	Int	
identifier			z	x	f	z	xs			
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

indeed the underlined `0` in `2:18`: it is involved in more failing tests than other locations, apart from the definition of the `foldInt` that spans lines 2 and 3.

Location	Ochiai	Tarantula
2:18	0.577	0.5
2:1-3:43	0.577	0.5
3:31	0	0

Figure 6.3: Top 3 suspiciousness scores from 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 and correctly point to `0` as the most suspicious expression in the spectrum. Still, its not often as clear which location is at fault. If we got the base case correct but had written `f x (foldInt f z xs)` in line 3, we would have the traditional `foldr` instead of a flipped `foldr` as presented here. Running the properties again, this accidental `foldr` pro-

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

Figure 6.4: The program from figure 6.1, slightly modified.

duces the spectrum in table 6.2. 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 6.5, formulas fall short in this case.

Location	Ochiai	Tarantula
2:18	0.577	0.5
3:24	0.577	0.5
3:37	0.577	0.5
3:39-40	0.577	0.5
3:26-41	0.577	0.5
3:22-41	0.577	0.5
2:1-3:41	0.577	0.5
3:35	0.0	0.0

Figure 6.5: Classic SBFL formula scores (bug location in bold).

While this would be a challenge to traditional SBFL formulas, our rule-based approach allows us to distinguish these cases, by inspecting the AST structure, types, and identifiers. The rule-based approach is detailed further in section 6.3.2, 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 6.3.2), 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 6.1. With that, we can rank `3:22-41` as the most suspicious. This motivates us to do a detailed analysis to shed light on which attributes are important.

6.1.2 Contributions

In this paper, we apply spectrums and existing suspiciousness scoring algorithms to Haskell and enrich it with unique, novel features: We use Haskell Program Coverage (HPC) instrumentation to determine whether an expression was touched during a test, but also to extract *how often* the location was evaluated. We note the test-framework (QuickCheck, Hunit, etc.) for later processing, and capture the *type*, constraints, and identifiers of locations that correspond to *expressions* within the spectrum, forming a richer spectrum than existing literature. We aim to cover many Haskell-specific attributes of test-cases and programs by these changes.

We provide the tool for spectrum generation as an `ingredient`[1] for the

[1] currently anonymized for peer-review

popular *Tasty* test framework.

To explore the spectras and generate new findings, we implement a rules-based approach to merge novel features and existing approaches. The targets for rules are (1) test attributes (test types, executions, frequency), (2) program attributes (AST structure), (3) existing SBFL formulas and (4) type-based complexity measures (constraints, arity, order).

The overall goal is to see whether the additional rules beyond existing literature improve fault localization for Haskell programs. To rank the locations for their suspiciousness, we concatenate the rule results into a vector and apply simple machine learning (ML) algorithms such as linear regression, decision trees, and (shallow) neural networks. We chose simple predictors to maintain explainability and ease of comparison. 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. They directly correlate with different features and form an insight themselves.

Rather than striving for improved outcomes by selectively interpreting metrics or meta-tuning classifiers, we offer insights and trends encompassing both successful and unsuccessful techniques. We provide an easy-to-adapt tool for practitioners and researchers to extract rich spectra. Popular open-source projects are used to verify the feasibility of spectrum extraction. Real-world bugs are analyzed in detail by formulating rules that capture spectrum attributes. Known SBFL formulas are applied and investigated for suitability and some simple ML algorithms are tested with rule-based vectors.

The total overview of the pipeline is seen in Figure 6.6. The novel elements and contributions are marked as bright-blue.

Research Questions We first investigate the spectrums and look what attributes distinguish them from their non-faulty counterparts.

RQ1.A: Spectrums of functional Programs

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

Before adaptations, it is worth looking at how existing research performs for typed functional programs. We thus apply common spectrum-based formulas from literature, summarized in table 6.7 in the appendix.

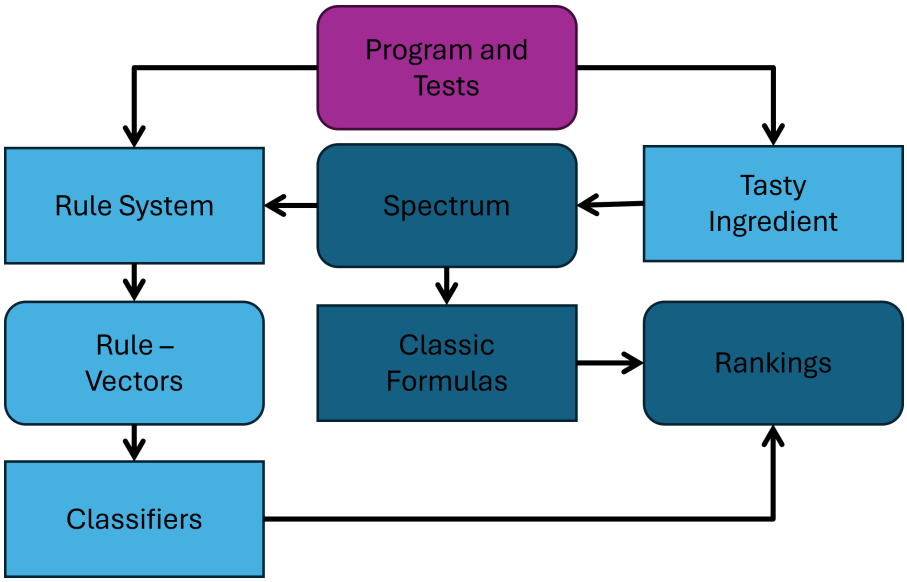


Figure 6.6: Overview of the fault localization Pipeline

RQ1.B: SBFL Formulas for functional Programs

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

We capture certain attributes of spectrums and their expressions through rules. After implementation, it remains to see if they are applicable. RQ1.C investigates the characteristics of the rules when applied to the spectrum:

RQ1.C: Applicability of Spectrum-Based Rules

What are the most prominent rules triggered by faulty expressions?

As a final subject of investigating the underlying programs, we analyze correlations between the rules and formulas.

RQ1.D: 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 simple classifiers and regressors to the data. Due to the exploratory nature, 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?

A common 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 among 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.

6.2 Background

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 [27], and is considered one of the most prominent due to its efficiency and effectiveness [29]. 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 statements 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 [11, 12, 16, 32] to assign suspiciousness scores to different parts of the program. While details differ, 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 statements and report the most suspicious locations.

An important piece of work from which we draw is Naish et al. [22] 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.

Other Fault Localization efforts for functional programming Fault localization in a functional setting has been explored in Liquid Haskell [30] 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.

6.3 Implementation & Experiment Setup

6.3.1 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 spectrums, 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 includes a *GHC source plugin*, which integrates with the compiler and extracts type and identifier information from modules during compilation to generate `.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 triggers a "bump" in the array, allowing HPC to track the number of times each expression was evaluated in the module. This array allows access, manipulation and re-initialization at runtime.

Spectrums 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. To compress the data, we include only locations that are involved in the tests by excluding those that have zero evaluations across the test suite.

6.3.2 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 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 analysis of the available types and constraints of a location.
- *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 6.4 provides an overview of the type-based rules.

We want to further motivate some of the rules presented in table 6.7. 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. 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. Taking this into account, *no one test is better than the others* - but there might be patterns that we only find when inspecting them separately.

Table 6.3: Rules based on AST-based behavior

rASTLeaf	Counts the distance of this node to the nearest leaf.
rFailUniqueBranch	Times this location is touched by failing test that touches none of its sibling expressions.
rFailFreqDiffParent	Ratio of evaluations compared to parent-evaluations.
rDistToFailure	Distance to a location touched in a failing test, by counting links to a common parent.

AST rules (seen in table 6.3) are based on existing research on active and algorithmic debugging [4, 8, 18]. They aim to capture differences in executions relative to parents and neighbors and reflect control-structures and program flow.

With the group of type rules in table 6.4, 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. `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. `rTypeArity` and `rTypePrimitives` allow us to identify correlations of faults with parts of a type and form basis of analysis, e.g. if faults occur in basic elements or complex compositions.

Unlike SBFL formulas, our novel rules are not intended as ranking algorithms, but rather as intermediate results for analysis, model features, and tie breaking (e.g. in table 6.2).

Table 6.4: Rules based on the expressions type

rTypeArity & rTypeConstraints	Number of arguments and 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.

6.3.3 Data

We draw data from two Haskell fault datasets, `HasBugs` [1] and `HaFla` [14]. 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. We determine faulty expressions as all expressions that are completely within faulty lines, extracted from the git-difference.

A subset of the data was chosen to produce the spectrums that met the required versions of Cabal, tasty (>v1.0), and GHC (>= 8.6). Other limitations excluded projects like Purescript (many of the tests run against compiled Javascript) or Cabal (all bug-asserting tests are package-level tests outside the tasty test suite). This results in a total of 11 programs[2] from 3 projects - **Pandoc**, **Duckling** and an **HLS**-plugin. An overview of the data points used is presented in table 6.5.

Pandoc is a document converter and, outside of language-specific tooling (GHC, Cabal, HLS, etc.), the biggest 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 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.

Comparison with Defects4J - Comparing the spectra between paradigms is challenging, but to approximate, we consult some data from Defects4J [13]. We draw our data from a public repository shared by René Just[3] that provides statistics from applying GZoltar [28] to a subset of 395 bugs from Defects4J.

The Defects4J bugs inspected have a mean *Source lines of code*(`SLOC`)[23][4]

[2] 9 from HasBugs, two from HaFla

[3] https://bitbucket.org/rjust/fault-localization-data/src/master/

[4] SLOC are lines of code, after removing whitespace and comments.

of 57.7k and a median of 62.5k. The mean number of tests in Defects4J is 1439, with a median of 202, with an average of 2 failing tests. Under the assumption that most of the SLOCs represent line-level statements, the resulting spectrums will have a comparable number of elements. The we approximate faulty SLOC for Defects4J as an average of 2.56, based on the lines removed by the patch. In conclusion, the programs and bugs used in this work are comparable in size to Defects4J.

6.3.4 Experimental Setup

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 mentioned in table 6.5, marking tests that fail before and after the changes as *noisy*. As the next step, the cabal file is altered to include spectrum generation and coverage, following the description in section 6.3.1 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 [9]. Within `TopX`, the recommended elements are sorted by their *suspiciousness*, and the *correct* classifications (truly faulty expressions) within the first `X` are counted. For this work, we considered the `Top10`, `Top50` and `Top100`, following previous literature.

Another common metric is `EXAM` [26], 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 spectrums, 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 `TopX` metrics [33].

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 [25].

Table 6.5: Overview of the used data points

Data Point	Issue	Faulty LOC	Faulty Expressions	Total Expressions	Failing Tests	Noisy Test-Failures	Total Tests
pandoc-3be256efb	Wrong application of 'Big Note' highlighting when converting to LaTeX. Reordering necessary.	1	6	88k	6	0	3254
pandoc-4	Failure converting combined code and bold text to LaTeX.	3	12	91k	1	1	3056
pandoc-5	Misinterpretation of code blocks when converting to ROFF MS. Requires escaping.	1	8	61k	2	6	2400
pandoc-6	Misconverting code blocks starting with (1) into enumerations.	5	39	59k	10	13	2365
pandoc-7	Empty multi-cells not picked up 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. Regex adjustment.	1	5	288k	1	0	364
duckling-4cfe88ea	Missing combined durations cases (e.g "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 combinators) in Portuguese language.	19	26	277k	1	1	360

At last, we considered a genetic algorithm using Pymoo [2] 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: *all*, *classic* (existing sbfl formulas), *original* (only novel rules added by this work) and *cherries* (a handpicked set of rules and formulas). To account for different value ranges, we re-run all experiments with `min-max-scaling`, mapping all features to values between 0 and 1. In the remainder of the paper, this is represented by the terms *scaled* (min-max scaling applied) and *unscaled*.

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. In the remainder of the paper models are named after their training data, e.g. *Pandoc-3 model*. For persisting trends of a single project, *pandoc models* refers to all models based on pandoc programs.

GA-based regression GAs utilize a custom fitness function to optimize the ranking of the first reported faulty locations, effectively optimizing on `TopX`. For GAs, we set the population to 200 individuals and use Latin Hypercube Sampling [19] to generate the initial population. The population is then evolved trough subsequent generations, by using *binary tournament selection* [20], for selecting the solutions (regression weights) for reproduction based on their fitness. *Simulated Binary Crossover* [7] SBX is used to recombine the selected solutions, and *polynomial mutation* [6] (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 [6]. 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 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 `TopX`, while for classifiers, true and false positives are evaluated. A global seed was used to account for inherent randomness.

6.4 Results

Attributes of Spectrums The created spectrums 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 tests, but compilation time of projects is longer as the `-fhpc` flag is required.

Table 6.8 groups the expressions into those touched by failing tests and those that are not. This allows for shrinking the spectrum, assuming that statements without failing tests are *innocent*. 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. We suspect that the tests do not run against the *original source*, but 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. For `pandoc-4` there are faulty locations on a *reader* that need changes in the data format. The relevant `golden test` runs with a compiled binary of `pandoc` (unlike the other `pandoc` data points) that is invoked by tasty, which is not collected in 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. Stemming from real projects, the tests are realistic and express community efforts. Although tests cover bugs *semantically*, it does not cover the faulty code and require new spectrum techniques. Due to common usage of Haskell for domain-specific languages, parsers, and code generation, we expect these types of faults to be more common in functional paradigms than in other languages.

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. This results in 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.

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 few represent an identifier. Less than half of the faulty expressions correspond to identifiers, and 4 data points do not have any faulty expressions that correspond to identifiers. The identifiers encountered match the project vocabulary (e.g., `parseMultiCell` in `pandoc-7`) with no trend of shorter identifiers being more faulty. This diverges from existing research's focus on *off-by-one* errors [21] or issues in predicates [15], which also focus on elements with identifiers.

RQ1.A: Attributes of Spectrums

3 Data points (`pandoc-4`,`duckling-4cfe88ea` & `duckling-ea8a4f6d`) do not have faulty expressions covered by a failing test, due to code-generation (`duckling`) and the test-suite utilizing binaries (`pandoc`). Two-thirds of expressions are AST-leaves, whereas about half the faults are AST-leaves. Almost all faulty expressions have a type, but identifiers are rare.

Existing SBFL-Formulas Table 6.6 shows the `Top50` results when applying existing formulas and sorting the statements by their resulting score. We focus on `Top50`, as `Top10` struggled with expression-level granularity and `Top100` showed the same trends at a bigger scale. For readability, we reduced table 6.6 to the best performing formulas.

Ochiai is the formula that performs best with our data, followed by DStar. 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. We expect that Ochiai performs best as it applies the square root in its denominator, which scales better for large number of expressions and tests. 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.

The best average scores are achieved by the strong performance of some formulas on `pandoc-6` and `HLS-afac9b18`. Our 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 ex-

pressions 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.

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 considered formulas.

The overall applicability 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 `duckling`, organizing tests into a corpus in combination with code generation makes formulas unsuitable.

RQ1.B: Existing SBFL Formulas

`Ochiai` and `DStar` produce the best `Top50` results with an average of 4.7 and 4.3 errors correctly reported in the first 50 expressions. All formulas struggle with `duckling` and `pandoc-4`, due to the faulty expressions not being touched by failing tests: A challenge to all spectrum-based methods, and not specific to the functional context.

Applicability of Rules and Correlations To investigate the correlation, we applied the Pearson correlation coefficient after combining the spectrums across projects, shown in figure 6.11.

Some correlations verify our assumptions that we considered trivial, e.g. that type lengths correlate with the number of subtypes. For most type-based rules this correlation is not statistically significant, but more complex types tend to be *longer*, have higher arity and order, and result more function application. A second block we see are SBFT formulas from literature with Ochiai, Tarantula, DStar, and OptimalP. This is mathematically plausible, as they are proportional to n_{ef}, the number of failing tests for this expression, in their formulas (see table 6.7). 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 seems 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 proportional to the number of tests (analogue to `rTPass` and `rTPassFreq`).

Table 6.6: 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
mean	17	2.8	4.7	4.3	4
median	12	0	1	0	0

This finding motivates us to investigate formulas focusing on frequency, as they seem more distinct from test failures than expected.

In general, the lack of correlation proves some *trivial* assumptions to be false, motivating further research and adjustments of existing formulas. We expect imperative programs to have similar patterns, but they can only be found as clearly in functional programs. Inferred type information at the expression level is uncommon in other paradigms, and investigating correlations between types, constraints, function arity and faults is out of reach for most imperative languages.

RQ1.D: 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.

Attributes of SBFL Models

Logistic & Linear Regression In both logistic and linear regression for both scaled and unscaled features, the resulting weights result in significant

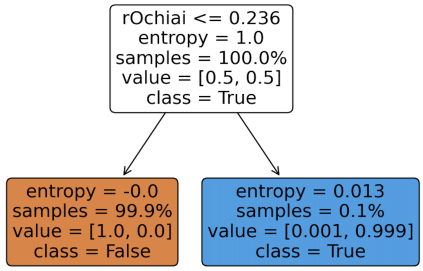


Figure 6.7: Decision Tree for Pandoc-5 (scaled features, `classic`-rules)

variance, indicating overfitting. For example, many type rules differ in logistic regression polarity despite being correlated (see RQ1.D).

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 6.7. 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. Trees that were striking to the authors are those that use one of the well-performing formulas (e.g., `DStar`), as root of the tree, and then use a niche rule such as `rHamming`, `rRogot1` or `rNumGoldFails`, which apply to very few locations.

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`, 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.

RQ2.A: Development of SBFL Models

Most models struggled with forms of overfitting. Linear and logistic regression, as well as decision trees, struggled with the sparse data. Genetic algorithms face issues converging for programs with untouched faults.

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 features. An overview of the transfer performance of the classifiers is shown in figure 6.8. 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 6.5.1).

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 the most challenging 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, 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 exist-

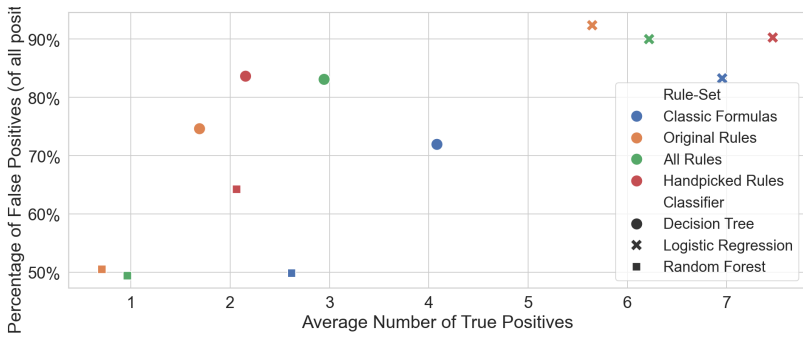


Figure 6.8: Transfer Performance of Classifiers

ing formulas, with the exception of genetic algorithms seen in figure 6.9 (and `Top10` in appendix figure 6.12). We see that especially for the `Top10` genetic search produces much better averages than the formulas.

We 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 6.10. The orange bars indicate the achieved `Top50`-score, while the blue frame indicates the maximum possible faults.

Figure 6.10a show 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 tops 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 less prone to overfitting. Figure 6.10b 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 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.

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.

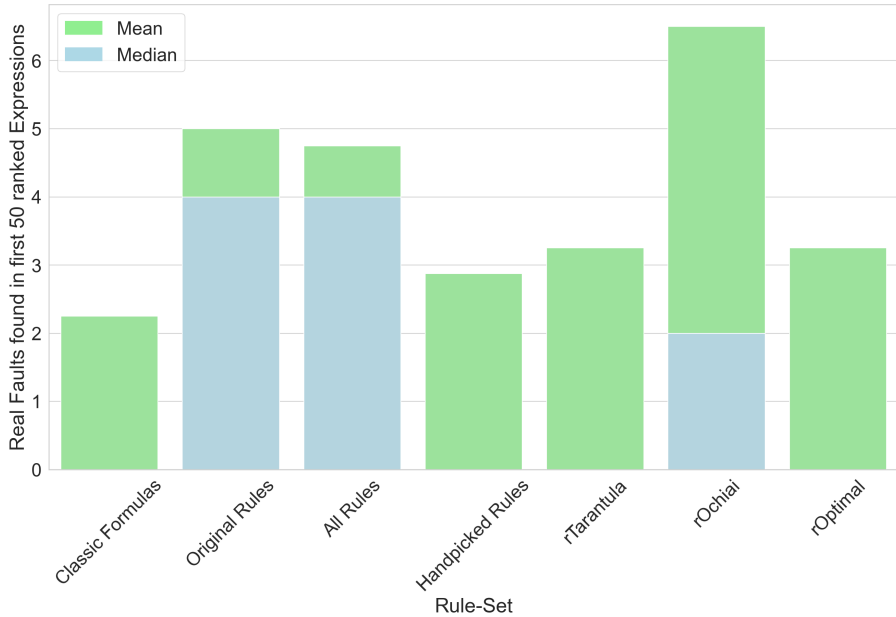


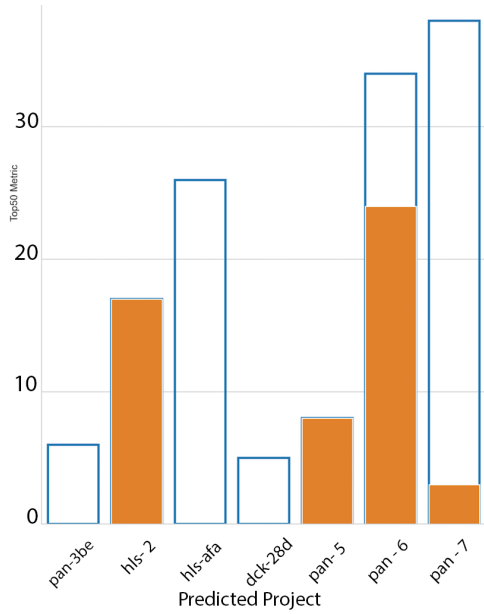
Figure 6.9: Averaged `Top50`-score for genetic search

RQ2.B: Applicability of Models

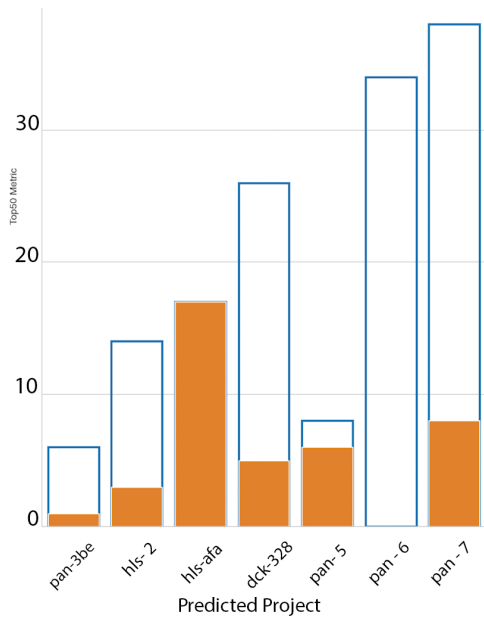
Classifiers performed better for scaled rules, whereas regressors had difficulties with unscaled features. Logistic regression produces high recall, but suffers from false positives. Random forests produced a good ratio of true to false positives, but did not have a high recall. Of the regressors, only genetic search beat the original formulas in average and median, especially in `Top10`.

6.5 Discussion

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



(a) Top50 from `HLS-2` with `original` rules



(b) Top50 from `duckling-28ddc3bf` with `original` rules

Figure 6.10: Example performance of promising search models

coverage through tests. Another way to get a better result is the reduction of a spectrum, possibly 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. We suggest that ensemble-style classifiers are used to utilize the best of both worlds. For most bugs, the existing formulas seem sufficient (or, one of them is), while unique features might play a role heavily dependent on the programs structure. Classifier fitted over multiple projects, also including *bug-free* ones, are a promising next step.

Project & Test Structure One recurring consideration throughout all results was the strong dependency on the project structure and 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 programs being plugins. This separation already leads to drastically smaller spectrums, 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 [5].

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.

6.5.1 Future Work

IDE integration 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. Apart from technical challenges in balancing performance and information, experiments can identify user needs when engaging with such tooling.

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. It might be extended to other concepts, e.g. *innocent types* or *innocent modules* from user-declaration. Innocent locations can be excluded from the fault localization process.

6.6 Related Work and Conclusion

Li et al. Comparable work on spectrum-based fault localization for Haskell originates from Li et al. [14]. 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 spectrums 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 implements the Ochiai and Tarantula algorithms for Haskell code [31]. They develop a custom compiler that compiles the program into S K I-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.

6.6.1 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 spectrums with expression–level granularity, including additional information on types and iden-

tifiers. 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 the spectrums 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 approximation 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 6.1) 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.

Acknowledgements

We thank David Sands and Matthew Sottile for many conversations about spectrums, and Alejandro Russo for his insight on FP-specific adaptions. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation.

Appendix

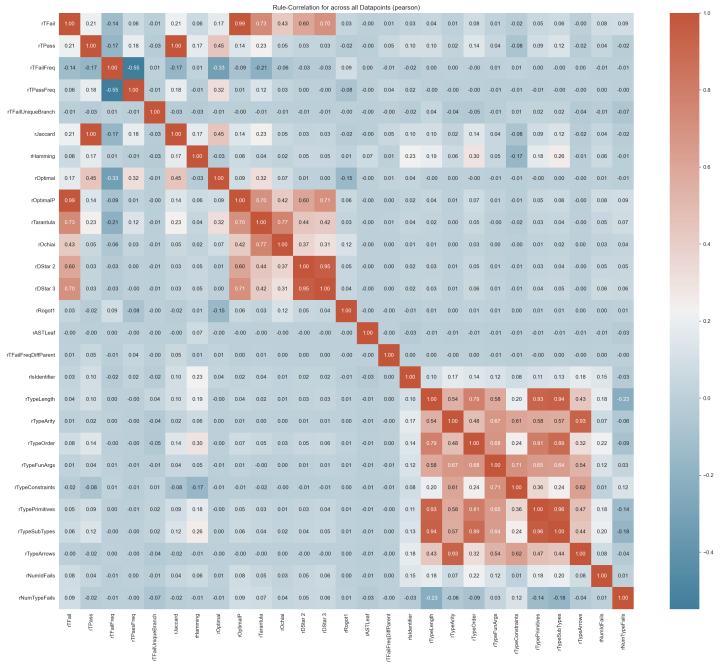


Figure 6.11: Pearson Correlation Matrix

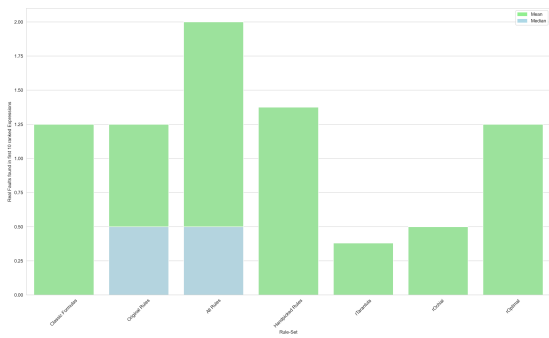


Figure 6.12: Averaged `Top10`-score for genetic search

Table 6.7: 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 k	$\frac{(n_{e_f})^k}{n_{t_f}+n_{e_p}}$
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 rules	See table 6.3
Type-based formula rules	See table 6.4

Table 6.8: Test-coverage within gathered spectrums

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

Bibliography

[1] L. Applis and A. Panichella. Hasbugs-handpicked haskell bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 223–227. IEEE, 2023.

[2] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[3] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229, 2013.

[4] R. Caballero, A. Riesco, and J. Silva. A survey of algorithmic debugging. *ACM Computing Surveys (CSUR)*, 50(4):1–35, 2017.

[5] T. Dao, N. Meng, and T. 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.

[6] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[7] K. Deb and R. Agrawal. Simulated binary crossover for continuous search space. *Complex systems*, 9(2):115–148, 1995.

[8] M. Faddegon and O. Chitil. Algorithmic debugging of real-world haskell programs: deriving dependencies from the cost centre stack. *ACM SIGPLAN Notices*, 50(6):33–42, 2015.

[9] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM Journal on discrete mathematics*, 17(1):134–160, 2003.

[10] R. L. Huang, E. Pertseva, M. Coblenz, and S. Lerner. How do haskell programmers debug? Plateau Workshop.

[11] J. A. Jones and M. J. 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.

[12] J. A. Jones, M. J. Harrold, and J. 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.

[13] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[14] F. Li, M. Wang, and D. 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.

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[16] D. Lo, L. Jiang, A. Budi, et al. Comprehensive evaluation of association measures for fault localization. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[17] J. Lubin and S. E. Chasins. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[18] S. Marlow, J. Iborra, B. Pope, and A. Gill. A lightweight interactive debugger for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 13–24, 2007.

[19] M. D. McKay, R. J. Beckman, and W. 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.

[20] B. L. Miller, D. E. Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[21] B. Mosolygó, N. Vándor, G. Antal, and P. Hegedűs. On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 495–499. IEEE, 2021.

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

[23] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

[24] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[25] 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.

[26] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[27] T. Reps, T. Ball, M. Das, and J. 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.

[28] A. Riboira and R. Abreu. The gzoltar project: A graphical debugger interface. In *International Academic and Industrial Conference on Practice and Research Techniques*, pages 215–218. Springer, 2010.

[29] Q. I. Sarhan and A. Beszedes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022.

[30] A. Tondwalkar, R. Recto, W. Weimer, and R. Jhala. Finding bugs in liquid haskell,-. 2016.

[31] V. Vasconcelos and M. A. Bigonha. Haskellfl: A tool for detecting logical errors in haskell. *International Journal of Computer and Systems Engineering*, 15(8):479–493, 2021.

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

[33] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.