Chapter 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`}.

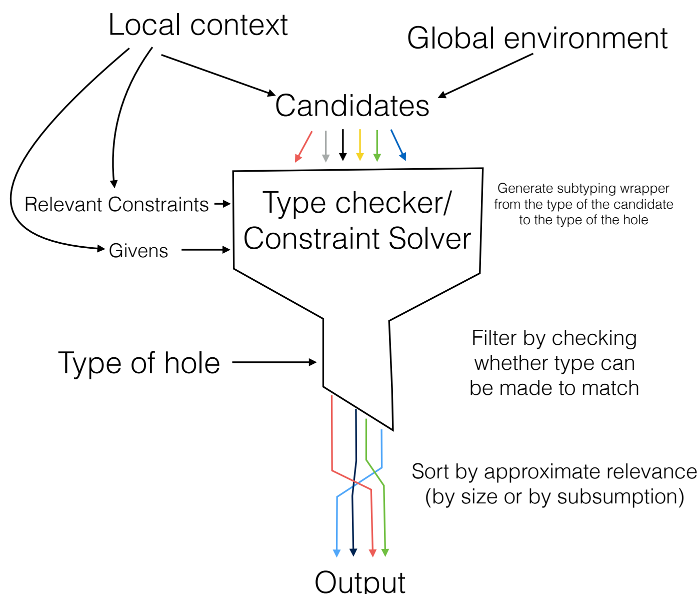

Figure 1.8: An overview of how valid hole–fit suggestions are found [12].

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].

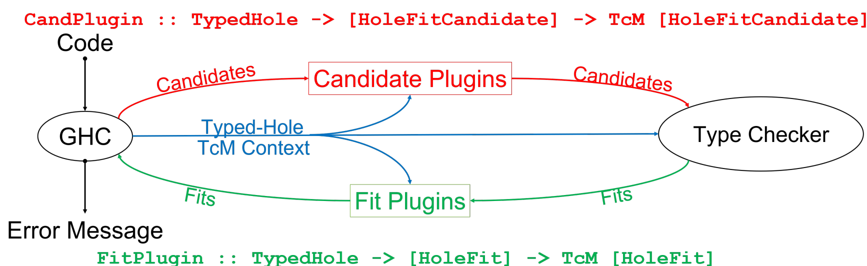

Figure 1.10: An overview of typed–hole plugins [13].

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

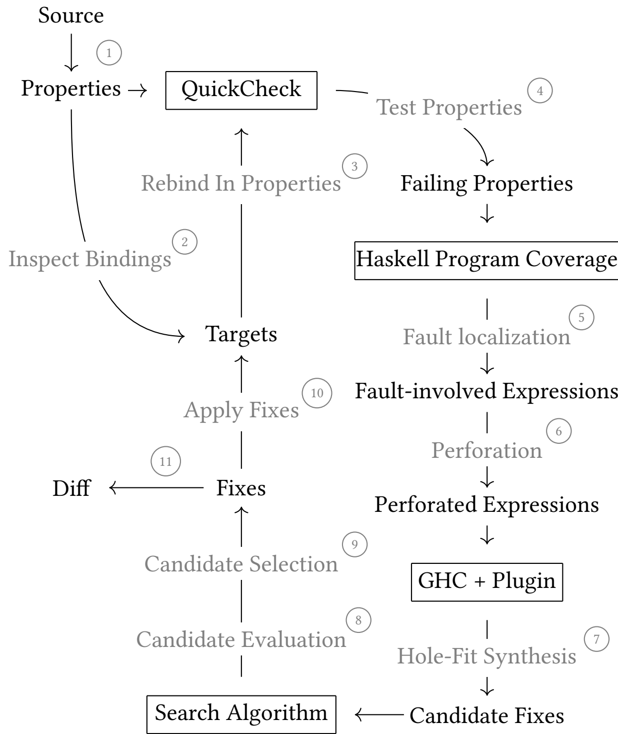

Figure 1.11: The PropR test–localize–synthesize–rebind loop

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.

```
3 gcd' :: Int -> Int -> Int
4 gcd' 0 b = gcd' 0 b
5 gcd' a b | b == 0 = a
6 gcd' a b =
7   if (a > b)
8     then gcd' (a - b) b
9     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.

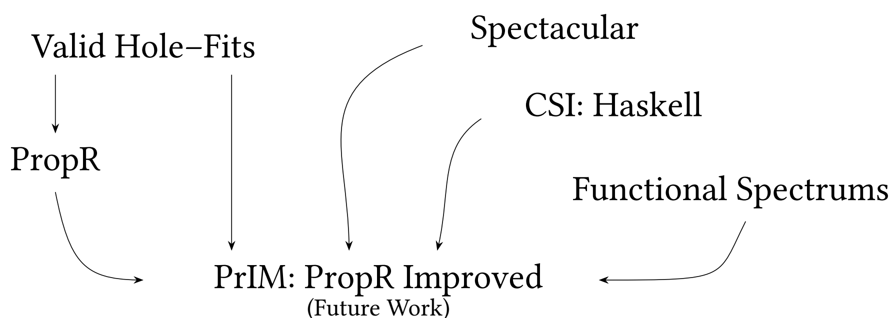

Figure 1.15: Thesis Structure Overview

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 hand-written 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.