Thesis for The Degree of Licentiate of Engineering

The Hole Story: Type-Driven Synthesis and Repair

Matthías Páll Gissurarson

CHALMERS

Division of Information Security
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

The Hole Story: Type-Driven Synthesis and Repair

Matthías Páll Gissurarson

Copyright ©2022 Matthías Páll Gissurarson
except where otherwise stated.
All rights reserved.

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

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

"GHC accepts source code as input and returns errors as output."
- Matti's SRC presentation, the main stage at ICFP, 2019

Abstract

Modern programs in languages like Haskell include a lot of information beyond what is strictly required for compilation, such as additional type information, unit tests and properties. This information is often used for post-compilation verification, by running the tests to verify that the code-as-written matches the specification provided by the types and properties. In this thesis, we explore ways of using this additional information to aid the developer during development. Firstly, we explore the integration of program synthesis into GHC compiler error messages using typed-hole suggestions to aid the completion of partial programs during development. Secondly, we present PropR, a tool based on type-driven synthesis aided by property-based testing and fault-localization in conjunction with genetic algorithms to automatically repair buggy programs, and evaluate its effectiveness. Finally, we present WRIT, a GHC plugin that allows developers to circumvent a too-restrictive type system in some cases in order to compile, run and test programs that are ill-typed but still executable, and explore its use in the context of information flow security for Haskell.

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

Acknowledgments

My sincerest thanks to my supervisor, David Sands, for getting me through some very tough times and getting this amazing WASP project in which I could continue working on typed-holes and synthesis despite earlier setbacks in my PhD. I want to thank the PhD crew for a lot of fun times, and especially my office mates Nachi and Agustín for bearing with me and my jokes and Benjamin, Alejandro (Gomez), and Agustín (again) for the all the video gaming that kept sane throughout the pandemic. Special thanks to my co-supervisors Alejandro Russo (officially) and Martin Monperrus (unofficially) as well for their input on my work so far, and my examiner John Hughes as well for setting the bar. Thanks to Matthew Sottile for advice and many Zoom calls during the pandemic, hopefully we'll get to meet in person one day! Thanks to my co-author Leonhard for forcing evaluation of our PropR paper, and many a fun hacking sessions. My friends and family back in Iceland, of course, for all their support in getting me to where I am today, and last but not least, I want to thank Anandi Rajan for all her love and support for the last two 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**

1.1 Motivation and Overview . . . 1
1.2 Background and Related Work . . . 2
1.2.1 Haskell . . . 3
1.2.2 Glasgow Haskell Compiler (GHC) . . . 3
1.2.3 Typed-Holes . . . 3
1.2.4 Program Synthesis . . . 7
1.2.5 Automatic Program Repair . . . 8
1.2.6 Property-Based Testing . . . 9
1.2.7 Weak Runtime-Irrelevant Typing . . . 9
1.3 Thesis structure . . . 10
Bibliography . . . 13

2 Suggesting Valid Hole Fits for Typed-Holes **15**

2.1 Introduction . . . 18
2.1.1 Contributions . . . 18
2.1.2 Background . . . 19
2.2 Case Studies . . . 20
2.2.1 Exercise from Programming in Haskell . . . 20
2.2.2 The Lens Library . . . 21
2.3 Implementation . . . 22
2.3.1 Inputs & Outputs . . . 22
2.3.2 Relevant Constraints . . . 23
2.3.3 Candidates . . . 23
2.3.4 Checking for Fit . . . 23
2.3.5 Refinement hole fits . . . 24
2.3.6 Sorting the Output . . . 24
2.3.7 Dealing with Side-effects . . . 25
2.4 An Additional Application . . . 26
2.5 Related Work & Ideas . . . 28

2.6 Conclusion 29
2.6.1 Future Work 29
2.6.2 Current Status 30
Bibliography 31

3 PropR: Property-Based Automatic Program Repair **33**
3.1 Introduction 35
3.2 Background and Related Work 37
3.2.1 Property-Based Testing 37
3.2.2 Haskell, GHC & Typed Holes 38
3.2.3 GenProg, Patch Representation, & Genetic Program Repair 40
3.2.4 Repair in Formal Verified Programs & Program Synthesis 40
3.3 Technical Details — PropR 42
3.3.1 Compiler-Driven Mutation 43
3.3.2 Fixes 46
3.3.3 Checking Fixes 47
3.3.4 Search 49
3.3.5 Looping and Finalizing Results 50
3.4 Empirical Study 50
3.4.1 Research Questions 50
3.4.2 Dataset 52
3.4.3 Methodology / Experiment Design 53
3.5 Results 54
3.6 Discussion 59
3.7 Threats to Validity 61
3.8 Conclusion 62
3.9 Online Resources 62
Bibliography 63

4 Weak Runtime-Irrelevant Typing for Security **71**
4.1 Programming with Type Constraints 73
4.2 Weakening Runtime-Irrelevant Typing 75
4.2.1 Ignoring Runtime-Irrelevant Constraints 75
4.2.2 Discharging Runtime-Irrelevant Equalities 76
4.2.3 Promoting Representationally-Equivalent Types . . . 76
4.2.4 Defaulting Runtime-Irrelevant Type Variables 77
4.2.5 Ensuring Runtime-Irrelevance 78
4.2.6 Turning Type-Errors into Warnings 78
4.3 Implementation 79

4.4 Conclusions and Future Work 80
Bibliography 83

Introduction

1.1 Motivation and Overview

When developers write programs, they have a specific goal in mind and an idea of how to achieve this goal. They often model this behavior in the source code itself by providing type alongside their functions and variables, and programs sometimes have a suite of tests to check that they run as intended. This means that the source code, provides the compiler with a lot of information beyond what is *strictly* required for the program to run as intended. The tests are checked only after the code has been written, and the type annotations are stricter than necessary or model restrictions that are checked at compile time but have no impact on runtime behavior. A lot of this information is currently only used in a yes-or-no manner: does the type of the code match the type annotations, do the tests succeed or fail? These are very useful questions to ask, but they do not guide the developer towards a correct implementation. Half of the time spent programming is spent on debugging [3], meaning that developers are working on almost complete programs. This means that there are usually some tests available (at least for the bug being fixed), and the types involved have stabilized. This can place a lot of constraints on the possible valid implementations of the program, which we could use to synthesize fixes to suggest to the developer to guide them towards a correct solution or even automatically repair an incorrect implementation. But how should we target our efforts? Program synthesis and constraint solving can be quite computationally heavy, and it is not tractable to synthesize whole programs, so a more focused approach is required. When much of the specification is inferred, it can also be hard to place blame on particular expressions: which one is the one that's wrong, the function or the argument? This brings us to the titular focus of this thesis: typed-holes. Typed-holes allow developers

to specify a *hole* in a program, which must be filled. These allow us to focus our synthesis efforts on the particular part of the program that the developer is most interested in, and avoid having to assign blame to any particular expression: the developer assigns blame by replacing the suspect expressions with typed-holes. This means we can improve the development experience where it counts the most: on the parts the developer's attention is directed at.

By using typed-hole directed synthesis to make more use out of the information already provided by the developers, we can make debugging easier or even completely automated while requiring little additional effort on the behalf of the developer.

Overview In this thesis, I (with the help of my paper co-authors) explore a few ways to go beyond yes-or-no responses by using typed-hole directed synthesis to guide the programmer towards a correct implementation via:

- typed-hole suggestions, error message additions that tell the programmer what they could use to replace a hole in the program with and the associated hole-fit plugins, compiler plugins that allow developers to customize the behavior of typed-hole suggestions,
- automated program repair, a technique to automatically fix programs based on to their type specifications, tests, and properties,
- and finally, automatic coercions that allow the users to weaken the type guarantees in cases where they are too restrictive, allowing them to make better use of available tests.

1.2 Background and Related Work

To get 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 with which our explorations have been conducted, we give a brief overview of program synthesis and the specific techniques we use to synthesize fixes, we 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, and finally, a short presentation of weak runtime-irrelevant typing and how it can be used to allow testing of ill-typed programs.

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 in many cases be done on an expression-by-expression basis, without having to consider side effects. 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 can be extrapolated into tests that cover a great deal more cases than a comparable amount of simple unit tests.

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 particularly relevant for 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 in this thesis [5]) and the subsequent hole-fit plugins (see section 1.2.3). These were initially implemented by me as a fork of the compiler and were eventually integrated into the official compiler release in version 8.6 and 8.10 respectively.

1.2.3 Typed-Holes

A typed-hole is a *hole* in the context of a program, with a type and the constraints on that type inferred by the compiler as if the hole was a free-variable. Inspired by a similar feature in Agda, a bare-bones implementation of typed-holes was initially added to GHC in version 7.8 [6]. An example of the typed-hole in `(_ "hello, world") :: [String]` can be seen in fig. 1.1.

```
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 1.1: An example of a typed-hole error message in GHCi 8.10.6.

Finding Valid Hole-Fits

Valid hole-fits were inspired by typed-hole suggestions in PureScript, but automatic proof-search was available prior in Agda as the *auto* command [6].

```
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 1.2: An example of valid hole-fits in GHCi, continued from the output in fig. 1.1. Presented without imports

As detailed in the first paper in this thesis and my master's thesis [5, 6], 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 fig. 1.2, 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 `map :: (a -> b) -> [a] -> [b]`,

the type of `[] :: forall a. [a]`, etc. We feed the type-checker with each of the equality types, as well as the hole's context and any relevant constraints[1], and ask the solver to solve the equality. If a solution is possible, then there is some 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.

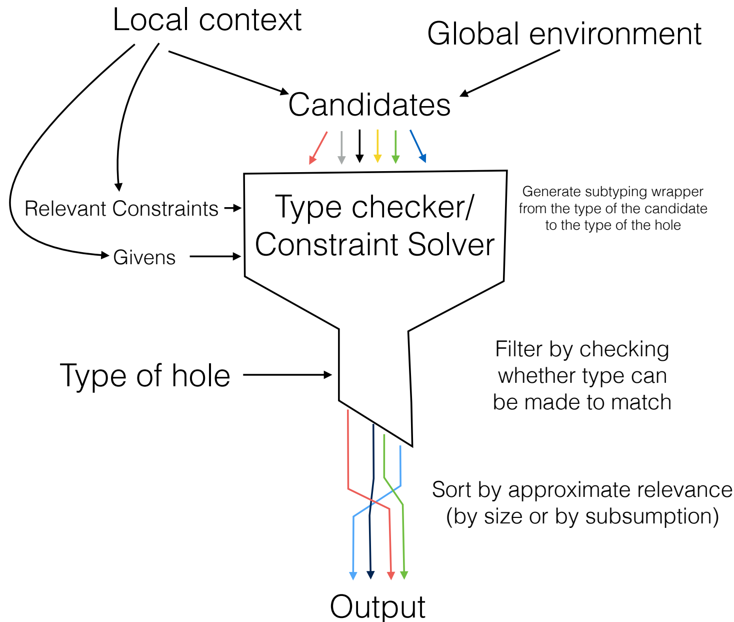

Figure 1.3: An overview of how valid hole-fit suggestions are found [6].

An overview of the process of finding valid hole-fits is shown in fig. 1.3.

Refinement Hole-Fits

Of special interest are the *refinement hole-fits*, which are an extension of valid hole-fits not found in PureScript [5]. 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

[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`}.

hole `_ :: [Int] -> Int`, where `_a :: Int -> Int -> Int` and `_b :: Int` are two new holes (here the refinement level is 2). An example of refinement hole-fits for the hole in fig. 1.1 can be seen in fig. 1.4.

```
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 1.4: An example of refinement hole-fits in GHCi, with the refinement level set to 2. Continued from the output in fig. 1.2. Presented without imports.

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 in this thesis and in my master's thesis [5, 6].

Hole-Fit Plugins

Hole-fit plugins are a recent addition of mine to GHC's plugin infrastructure that allow 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 [7].

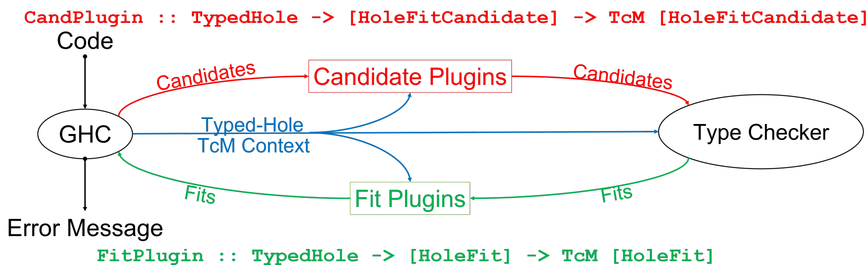

Figure 1.5: An overview of typed-hole plugins [7].

This enables us to e.g. 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 fig. 1.5[2].

1.2.4 Program Synthesis

Program synthesis is the generation of code based on a high-level specification of how that program should behave [19]. As there is an infinite amount of programs, efficiency is the key to practical program synthesis. One way to restrict the search space is to use input-output examples, such as in FlashFill [10]. Using only input-output examples can be restrictive, but works well when the target language is domain specific: this limits the search space by reducing the possible programs that can be written in a language. Another way to efficiently synthesize programs is to focus the synthesis on parts of the program, like with *sketching*, where users write a high-level sketch of a program but leave holes for the computer to synthesize low-level details [19].

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 [16]. How well type-directed synthesis can perform depends on the expressiveness of the type-system. For instance, expressive type-systems like the refinement types used in SynQuid allow developers to decorate the types with predicates from a decidable logic, meaning they can more precisely specify which programs are valid, which improve the program synthesis [16]. However, more expressiveness in the type-system comes at the

[2]Presented as part of the Haskell Implementors' Workshop and the ICFP student research competition in 2019 [7].

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 also has a long history in Haskell, such as the type-based Djinn synthesizer, which can synthesize Haskell expressions based on the type [1]. A more recent Haskell based synthesizer is Hoogle+, which uses type-guided abstract refinement to find programs composed from functions in popular Haskell libraries based on a type and input-output examples [11]. Typed-hole directed synthesis is a combination of using the contextual information as is done in sketching and using the type information to restrict the search space to only those programs that satisfy the type, such as the one used in Myth [14] and in the work of Perelman et al. for partial expression completion in C# [15], but differs in that it is integrated directly into the compiler itself. By integrating the synthesis into the compiler we can ensure the compatibility of the synthesized hole-fits, since the checking is done by the type-checker itself.

1.2.5 Automatic Program Repair

A practical application of program synthesis is automated program repair, where we fix bugs in programs according to its specification. There is already some examples of type-directed program repair such as Lifty, that uses the refinement type-based technique from SynQuid to repair security policy violations in a domain specific language [17]. To 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 [8] (the second paper of this thesis). We've already covered the type-directed synthesis using valid-hole fits in section 1.2.3, but for an overview of genetic-search-based program repair and property-based testing, see section 1.2.5 and section 1.2.6 below.

Genetic Program Repair

Genetic program repair is a successful generate-and-validate-based approach to automated program repair based on genetic search [12, 13]. 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 validity of fixes [12]. 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-perfoming fixes to low-perfoming fixes [12]. For some programs, this approach can find fixes that completely

eliminate the bug found by the tests [12]. Current state-of-the-art program repair tools like Astor have been based on the same approach, but mainly target Java [13]. A genetic approach allows us to focus on finding simple partial fixes and combining them, meaning we can do repair on a per-fault basis rather than having to consider the whole program.

1.2.6 Property-Based Testing

Property-based testing frameworks such as QuickCheck [4] 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 randomly generating data based on the type of the property and checking that the property holds. This allows one property to be the equivalent of hundreds or thousands of unit tests, and also generates a minimal counter-example when a property does not hold. These counter-examples 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 tests. We use properties and their counter-examples in the second paper of this thesis to do automated program repair [8].

1.2.7 Weak Runtime-Irrelevant Typing

Weak runtime-irrelevant typing is introduced in the third paper of this thesis [9]. When we use sophisticated type-systems a problem emerges: how can we test programs that fail to compile due to a type error? This can sometimes be the case when the program-as-written could be run and tested, but overzealous type-annotations prevent it from type-checking. This means that the types involved in the error are *runtime-irrelevant*, meaning that their runtime representation is the same, but there is additional type information that separates them such as a phantom type parameter. A common practice in Haskell is to use phantom type parameters to model non-functional properties in the type system, such as modeling information flow control using the MAC library [18]. In the third paper in this thesis [9], we present the WRIT plugin which allows users to specify certain additional rules in the type-system that can be used to improve the error message by turning cryptic type-system unification errors into domain specific ones or turn the type-errors into warnings using GHC's safe zero-cost coercions [2, 9]. This means we can compile and run the ill-typed programs in question, by disregarding their non-functional type annotations and allowing us to test their functional correctness using tests or properties, which is crucial for automatic program repair. This opens up an exciting avenue for further research

into automatic repair of security annotated programs [9].

1.3 Thesis structure

Paper 1: Suggesting Valid Hole Fits for Typed-Holes (Experience Report) [5]

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, and the more advanced subsumption sorting, where more "specific" types are treated as more "relevant" than more general types.

Statement of contributions Single authored

Appeared in: Haskell Symposium 2018

Paper 2: PropR: Property-Based Automatic Program Repair [8]

In the PropR paper, we introduce PropR, a tool that automatically repairs 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 counter-examples 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. I implemented the synthesis and repair as well as writing the technical section of the paper and parts of the intro, whereas Leonhard focused on the genetic repair algorithm and the experimental verification.

Preliminary version, accepted to the International Conference on Software Engineering 2022 (ICSE '22)

Paper 3: Short Paper: Weak Runtime-Irrelevant Typing for Security [9]

In this paper, we introduce the WRIT-plugin. The WRIT plugin allows users to add specific additional rules to GHC's type system that allow programs to compile that are ill-typed according to GHC but for which we know that the type-errors are runtime-irrelevant way, i.e. the differences in types would not result in a crash or incorrect functional behavior. This allows us to e.g. compile programs that have security errors (as defined by Russo's MAC library), but are otherwise functionally correct, meaning that we can run automatic QuickCheck tests even in the presence of type errors.

Statement of contributions Both authors contributed equally to the paper. I focused on the technical and implementation aspect, as well as the judgements that defined the behavior of the system.

Appeared in: Programming Languages and Security 2020

Bibliography

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

[2] J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 189–202, New York, NY, USA, 2014. Association for Computing Machinery.

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

[4] 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.

[5] 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.

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

[7] 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`.

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

[9] M. P. Gissurarson and A. Mista. Short Paper: Weak Runtime-Irrelevant Typing for Security. In *Proceedings of the 15th Workshop on Programming Languages and Analysis for Security*, pages 13–17, 2020. This is the third paper in this thesis, and is included as chapter 4.

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, Jan 2011.

[11] 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.

[12] 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.

[13] 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.

[14] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. *SIGPLAN Not.*, 50(6):619–630, Jun 2015.

[15] 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.

[16] 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.

[17] 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.

[18] A. Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, page 280–288, New York, NY, USA, 2015. Association for Computing Machinery.

[19] 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

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 [18]. 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.

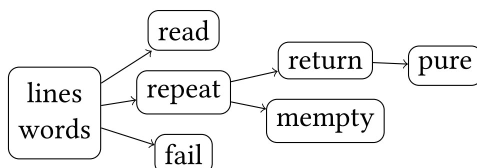

Figure 2.3: The subsumption graph for matches for `_ :: String -> [String]`. Here `lines` would come before `repeat`, `read`, and `fail`, `repeat` before `mempty` and `return`, etc.

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 simplification. 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 [1]. **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, 37], 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

ICSE 2022

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, 65], predictive-evaluation in search [71], and duplicate handling [10]. 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, 63], or by empirical evidence that fixes are often related to dependencies, not the code itself [4, 15]. 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 [12]. The fitness is defined as the number of failing tests in the suite [13], 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 wished for 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. [69] to address overfitting was to introduce tests generated with EvoSuite [16] 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 by addressing the overfitting problem by introducing properties [4] 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 various algorithms (random search, exhaustive search, and genetic algorithms). Our fixes follow a GenProg-like approach [12] 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* [5]. 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 [4]. 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 [4]. 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 [4].

There are some variations on property-based testing, e.g. SmallCheck, which performs an *exhaustive test* of the property [57]. 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.

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

```
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
```

```
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)
```

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

Figure 3.1: Comparison of Properties and Unit Tests for sin

of properties that the new tests are ad-hoc generated. We hope for this to address the problem of *overfitting* [52], as there are no *fixed* tests to fit on as long as seeds are changing. 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. Haskell's 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 [4] to efficiently test functions using properties, where the inputs are generated by QuickCheck based on a generator for data of a given type.

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* [5]. A "hole", denoted by an underscore character (_), allows a programmer to write an incomplete program, where the hole is a placeholder for some 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* [5]. 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 [18], 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, Patch Representation, & Genetic Program Repair

Search-based program repair revolved mostly around the work of Le Goues et al. [12] 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 [12] mostly for Java. Particularly Astor [39] enabled lots of research [60, 64, 67, 68] due to its modular approach, as well as real-world applications [58, 61]. 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 in Formal Verified Programs & Program Synthesis

Another field of research dominant in functional program is formal verification [7], in which mathematical attributes are defined to proof 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, 59], arguably some languages can be considered synthesizers around constraints (e.g. Prolog). For Haskell, these approaches are revolving around *Liquid Types* [48, 56], a framework that allows for convenient meta-programming. The existing approaches [16, 21, 25] 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* language is particularly relevant. Lifty is a domain-specific data-centric language where it can be statically and automatically verified that applications handle data according to declarative security policies, and suggests a provably correct repairs if there is a leak of sensitive data [17]. 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 [11]. 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 it synthesizes by type-enumeration after symbolic execution. To some degree, this is similar to our approach of an exhaustive search. There are some scalability concerns for symbolic execution [44] (which are subject to active research), which genetic algorithms successfully mitigate, motivating their usage in PropR.

Program repair benefits when compared to synthesis if we consider the developers to deliver a reasonable baseline program. Assuming this, we can exchange most of the search-related problems in favor of two separate challenges: fault localization and repair. Unlike most existing research in program repair, we do not utilize *donor-approaches* that transplant elements into faulty parts, but we perform located synthesis instead. Hence, to some degree, we consider this work a bridge between synthesis and program repair.

In terms of utilizing specifications, the primary benefit of QuickCheck is the (comparably) easy adoption for users. Formal verification comes with high entry-barriers for most programs and requires very 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. While a full formal verification-suite could produce better results, we try to ease the adoption of our approach by utilizing properties and tests instead.

3.3 Technical Details — PropR

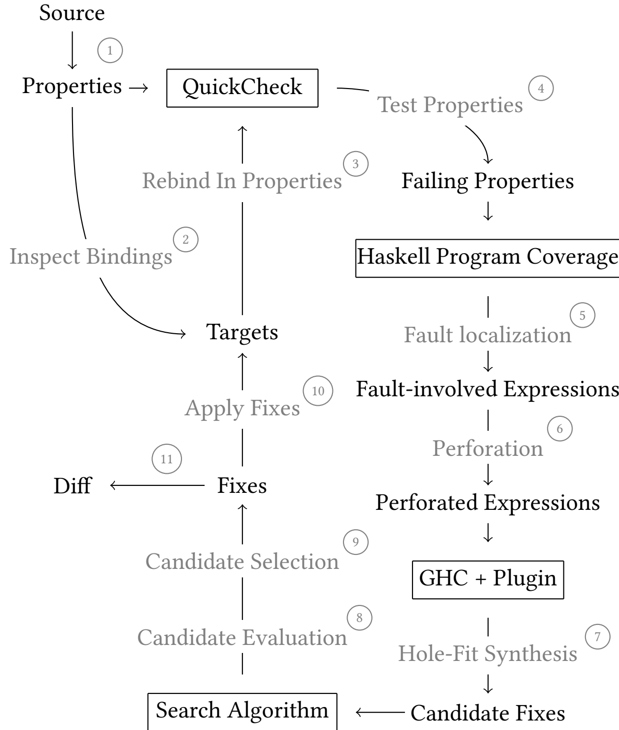

Figure 3.3: The PropR test-localize-synthesize-rebind loop

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 ⓝ 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.

```
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.

```
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 [18] 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 [13, 39]. GHC's *Haskell Program Coverage* (HPC) can instrument Haskell modules and get a count of how many times each expression is evaluated during execution [19]. Using QuickCheck, we find which properties are failing and generate a counterexample for each failing property ④. For properties without arguments (essentially unit tests), we do not

need any additional arguments, so we can run the property as-is: the counterexample is the property itself. By applying each property to its counterexample and instrumenting the resulting program with HPC, we can see exactly which expressions in the module are evaluated in a failing execution of property ⑤. The expressions evaluated in the counterexample of the property are precisely the expressions for which a replacement would have an effect: non-evaluated expressions cannot contribute to the failing of a property. We call these the *fault-involved expressions*. These will be *all* the expressions involved in failing tests/properties, and covers every expression invoked when running counter-examples.

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

Perforation For the targets, we generate a version of the AST with a new typed hole in it, in a process we call *perforation*. When we perforate a target, we generate a copy of its AST for each fault-involved expression in the target, where the expression has been replaced with a typed hole ⑥. The perforated ASTs are then compiled with GHC. Since they now have a typed hole, the compilation will invoke GHC's valid hole-fit synthesis [5] 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 result in 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`s or `Int`s) 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

```
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

seen in Figure 3.11, though we do additional work to extract the results in PropR. It might be the case that the resulting program does not compile: as our synthesis is based on the types, we might generate programs that do not 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 [16]. 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 indeed 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 completely deterministic except for the randomness inherent in QuickCheck. The primary reason for exhaustive search is to show the complexity of the problem, i.e., search is necessary over sheer brute force.

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, that is after the selection in chromosomes the program is re-bound and coverage evaluated. The authors observed that this is a bit over-engineered for small programs — within small problems the fault localization did not greatly change especially for programs with only a single failing property. Hence, we added a flag to skip the steps 5 to 7 in the loop, in order to speed up the actual search. This configuration was also enabled during experiments presented in 3.4.

The exhaustive and random search do not use any rebinds - once the loop reaches their step they exhaust the search budget.

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: ① it compiled ② it failed the unit test suite **and** the property-based test suite separately. An *error-producing test* is considered as a normal failure. We selected them by these criteria to draw per-data-point comparisons of properties to unit tests and their unison. We consider a separate investigation of repairing unit test failing programs versus properties failing programs and their overfitting future research.

```
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 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 docker container to 8 vCPUs @ 3.6ghz and 16gb RAM (The container's lifetime is exactly one data-point). Further information on the data collection can be found in the reproduction package.

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 [62] 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 [66] 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, unfortunately we had no specifications available to us for the dataset. Hence, 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.

[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

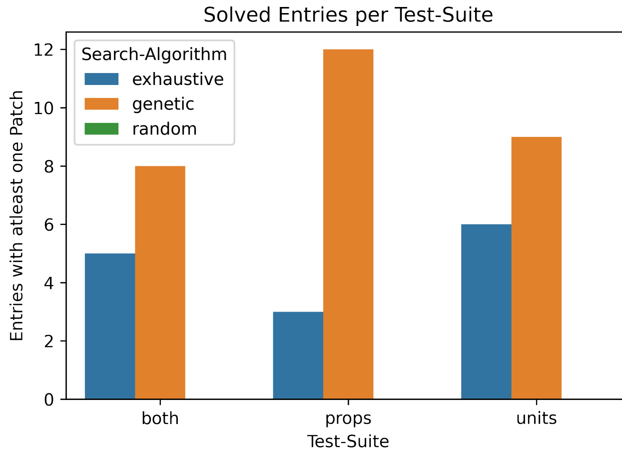

Figure 3.14: Solved Entries per Test-Suite and Algorithm

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

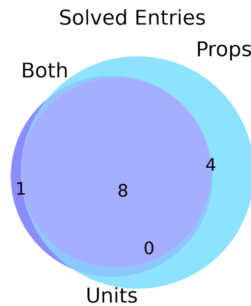

Figure 3.15: Venn-Diagram of Solved Entries per Suite

imply that properties are *better* — the patches can still be overfit and are to be evaluated in RQ3.

Summary RQ1

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

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

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. This behavior holds true for genetic and exhaustive search.

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*

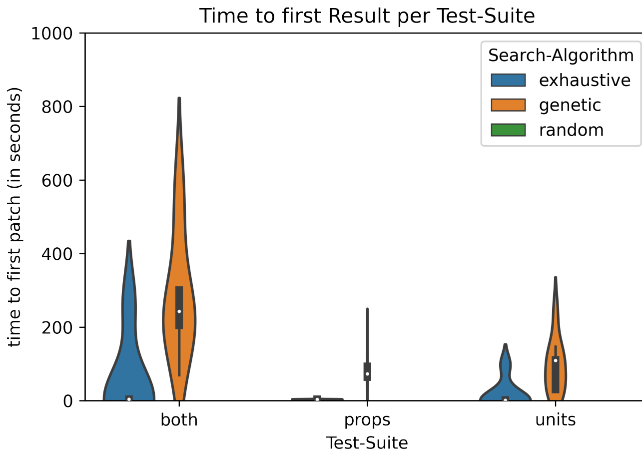

Figure 3.16: Distribution of Time to First Patch per Entry

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 [70] 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 test. This (actually) solves the unit test as expected and is therefore arguably

```
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

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 mitigate this is to provide a handful of unit tests or write a specific expression-generator used for the flaky property. We labeled empty patches to be overfit

as we do not consider them proper repairs.

Summary RQ3

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

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 in principle fails other properties such as multiplicative associativity, but (in rare cases) Quick-Check produced trivial 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 especially 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 effective 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 at the moment, 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. Hence, 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 widely used.

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 widely unsolvable: 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 [13, 17], can more easily 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.

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

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 [13] or ARJA [70] 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, 8] and either be used as mutations or as an initial population of chromosomes. Representing multiple types of changes is only a matter of representation within the chromosome — the remaining search, fitness and fault localization can be kept as is.

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

In terms of further evaluation, the next steps are user surveys and experiments on real world applications such as Pandoc[4] or Alex[5]. In particular, we envision a bot similar to Sorald [15] 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, despite may not being added to the code, contributed to fault localization or improvements of the test suite.

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 publish it on acceptance 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 would like to stress that student data has been successfully used in literature for program repair [12, 14, 31, 33].

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

A real-world study on program such as Pandoc [11] is part of future work. Pandoc, a popular Haskell document-converter, is in fact rich in properties that test e.g., for symmetry and reflectivity over different conversions.

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. What we observed was that properties did not increase the number of programs for which patches were found, but when it finds solutions they were less overfit and found faster. Overfitting that was 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-licence 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] P. Bjesse. What is formal verification? *SIGDA Newsl.*, 35(24):1–es, dec 2005.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[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 varghadelaney 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] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

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

[71] 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

Short Paper: Weak Runtime-Irrelevant Typing for Security

Matthías Páll Gissurarson and Agustín Mista

PLAS 2020

Abstract. Types indexed with extra type-level information are a powerful tool for statically enforcing domain-specific security properties. In many cases, this extra information is runtime-irrelevant, and so it can be completely erased at compile-time without degrading the performance of the compiled code. In practice, however, the added bureaucracy often disrupts the development process, as programmers must completely adhere to new complex constraints in order to even compile their code.
In this work we present WRIT, a plugin for the GHC Haskell compiler that relaxes the type checking process in the presence of runtime-irrelevant constraints. In particular, WRIT can automatically coerce between runtime equivalent types, allowing users to run programs even in the presence of some classes of type errors. This allows us to gradually secure our code while still being able to compile at each step, separating security concerns from functional correctness.
Moreover, we present a novel way to specify which types should be considered equivalent for the purpose of allowing the program to run, how ambiguity at the type level should be resolved and which constraints can be safely ignored and turned into warnings.

4.1 Programming with Type Constraints

Enforcing domain-specific properties is a complicated task that developers are forced to carefully address when designing complex systems. In the functional programming realm, strongly-typed languages like Haskell are an advantage since *one can use the type system to enforce domain-specific constraints*! However, this technique is not without flaws. To illustrate some of the issues with this technique, suppose we are writing a library for information-flow control over labeled pure values – loosely inspired by the `MAC` library by Russo [18]. For simplicity, we assume that the only labels are `L` for public and `H` for secret data. Then, we can use *phantom* types [3, 8] to label arbitrary data with security labels:

```
data Label = L | H

newtype Labeled (l :: Label) a = Labeled a
```

As an example, the value `Labeled 42 :: Labeled L Int` represents a public integer, whereas `Labeled "1234" :: Labeled H String` represents a secret string. It is important to note that in Haskell, `newtype`s are representationally equal to the type they wrap, meaning that the runtime representation of `Labeled 42` is the same as the one for `42`. Later, labeled values can be combined according to different security policies using type constraints [6, 7], as an example, we can enforce that no information flows from `H` to `L` by defining the empty type class:

```
class ((l :: Label) <= (l' :: Label))
```

and defining instances of `(<=)` only for the flows we allow:

```
instance (L <= L)
instance (L <= H)
instance (H <= H)
```

Since there is no instance for the forbidden flow `H <= L`, any code that triggers the constraint `H <= L` during compilation will produce a type error.

Note that the class (<=) has no methods, so it is represented by a computationally-irrelevant empty dictionary at runtime.

We can now use (<=) to implement combinators over labeled values that ensure that secrets do not leak into public data, e.g. the familiar `zip` combinator can be given the type:

```
zip :: (x <= z, y <= z) => Labeled x [a]
                        -> Labeled y [b]
                        -> Labeled z [(a, b)]
```

where `(x <= z, y <= z)` ensures that the label `z` of the output is greater or equal to both its inputs. Then, the definition:

```
bad :: Labeled L [(Usr, Pwd)]
bad = zip (Labeled [11111, 222222] :: Labeled L [Usr])
          (Labeled ["hun", "ter2"] :: Labeled H [Pwd])
```

will be rejected by GHC with a generic error indicating that we are missing a type class instance for the forbidden flow:

```
error: No instance for H <= L (...)
```

and indeed, we can see that there is a leak from the secret passwords in the list `["hun","ter2"]` to the public list `[(11111,"hun"), (222222,"ter2")]`. Ouch!

As shown so far, we can use Haskell's type system to accommodate domain-specific constraints about security labels using phantom types and type classes. Although this is a powerful strategy when it comes to writing domain-specific libraries [1, 9, 12, 16], it can be hard to use in practice:

- The code cannot be run unless it is provably secure, preventing users from testing the functional correctness of the program separately from its security properties.
- Users must tag all their data with an explicit `Label`, and cannot use features such as pattern matching without explicitly unwrapping and rewrapping the labels.
- Moreover, they need to tag both the secret and the public data, even though there might exist a sane default tag.
- The type errors are too general and hard to understand for users unfamiliar with Haskell's type system, and;
- Synthesizing type based suggestions [5] becomes harder, due to domain-specific constraints and ambiguous types.

4.2 Weakening Runtime-Irrelevant Typing

In GHC, type checking is based on constraint-based type inference. Albeit intricate in practice, the algorithm works by traversing the code to accumulate a set of type constraints (defined as part of the type system specification) and then invokes the constraint solver to solve those constraints [19]. In the latest GHC, constraints come in three main flavours [17]:

- *Givens* from type signatures, for which we have evidence,
- *Wanteds* from expressions, for which we want evidence,
- *Deriveds*, which are constraints that any solution must satisfy but we do not require evidence of (e.g. equalities arising from functional dependencies and superclasses).

The constraint solver solves the wanteds with respect to the givens and the typing rules of GHC (which include creating and unifying type variables), making sure that the solution satisfies the deriveds [17, 19]. This process is capable of type checking complex programs, but isn't perfect when it comes to domain-specific constraints like `(<=)`.

Luckily, the type checker can be extended with plugins to handle additional type checking rules, for example to simplify naturals or invoking an SMT solver [4, 10]. Type checker plugins are invoked by the compiler in order to *a*) simplify givens, where a plugin might find a contradiction, and, *b*) whenever there are unsolved constraints that the type checker could not solve.

For the purpose of weakening the type checking of runtime-irrelevant types, we developed WRIT,[1] a plugin that extends GHC's type system by adding the rules seen in Figure 4.1 for when type checking would not be able to proceed otherwise. Users of the plugin can selectively apply these rules to runtime-irrelevant constraints and equalities by writing instances of the **`Ignore`**, **`Discharge`**, **`Promote`**, and **`Default`** type families [14, 20] as described in the rest of this section.

4.2.1 Ignoring Runtime-Irrelevant Constraints

In Haskell, users can define empty typeclasses that have no methods (like `(<=)`), which represent runtime-irrelevant constraints. However, we would like to be able to turn these constraints into compile time warnings, so that functional correctness of the program can be verified separately from its se-

[1] The WRIT plugin is available at `https://github.com/tritlo/writ-plugin`.

curity. The IGNORE rule applies whenever there is an unsolved empty type-class constraint with an instance of the `Ignore` family:

```
type family Ignore (c :: Constraint) :: Message
```

By defining an instance of the `Ignore` family for `(<=)`:

```
type instance Ignore (H <= L) =
  Msg (Text "Found forbidden flow from H to L!")
```

Users can specify that the constraint `H <= L` can be ignored with the message shown above. With this instance in scope and WRIT enabled, the error for the `bad` function defined earlier will be turned into the following warning:

```
warning: Found forbidden flow from H to L!
```

4.2.2 Discharging Runtime-Irrelevant Equalities

With runtime-irrelevant types, we often want to ignore nominal equalties of the form `a` ∼ `b`, which are specially handled GHC primitives. As an example, we might want to turn `L` ∼ `H` into a warning when compiling insecure programs. The DISCHARGE rule applies to unsolved equalities of the form `a` ∼ `b`, for which there is an instance of the `Discharge` family for `a` and `b`:

```
type family Discharge (a :: k) (b :: k) :: Message
```

By defining an instance of `Discharge` for `L` and `H`:

```
type instance Discharge L H =
  Msg (Text "Using a public L as a secret H!")
```

Users can allow `L` ∼ `H` with the message shown above. This in conjunction with ignoring `H <= L` effectively negates any guarantees that our library provides.

4.2.3 Promoting Representationally-Equivalent Types

A special case of discharging is when `a` and `b` have the kind `(*)`, the kind of base types in Haskell. Discharging the equality `a` ∼ `b` effectively *promotes* `a` to `b`, meaning that `a` is treated as a `b`. This is only runtime-irrelevant when `a` and `b` have the same runtime representation, making `a` ∼ `b` runtime-irrelevant. This coincides with the `Coercible` constraint in GHC [2], so to handle this common case we define `Promote`:

```
type family Promote (a :: *) (b :: *) :: Message
```

And define an instance of `Discharge` for types of kind `(*)`:

```
type instance Discharge (a :: *) (b :: *) =
  OnlyIf (Coercible a b) (Promote a b)
```

Then, by defining an instance of `Promote` for labeled values:

```
type instance Promote a (Labeled l a) =
  Msg (Text "Promoting unlabeled " :<>: ShowType a
      :<>: Text " to " :<>: ShowType (Labeled l a))
```

Users can use any base type `a` (like `Int`) as a `Labeled l a`, where `l` is either `L` or `H`, e.g., it becomes possible to write: `[1,2] :: Labeled L [Int]`, where `[1,2]` is promoted and treated as a public `[Int]`.

4.2.4 Defaulting Runtime-Irrelevant Type Variables

When programming using runtime-irrelevant types, it frequently occurs that the type of a phantom type variable cannot be inferred. However, it is often the case that there is a "sane" value to choose when there are no restrictions, such as the label `L` for labeled data. The Default rule applies whenever there is an unsolved constraint with a free type variable of kind k for which there is an instance of the `Default` family:

```
type family Default k :: k
```

By defining an instance of the `Default` family for `Label`:

```
type instance Default Label = L
```

Users can specify that any free type variables of kind `Label` in an unsolved constraint should be set to `L`.

Now With Less Cruft! After defining the instances as shown above, WRIT can use them to weaken our library's domain-specific constraints. Users can then easily express and run (possibly insecure) programs operating on labeled values as if they had the underlying type without overhead:

```
labeledOr :: Labeled L [Bool] -> Labeled L Bool
labeledOr (x:xs) = if x then True else labeledOr xs
labeledOr _      = True
```

$$\frac{\Gamma, \mathrm{Default}\, k, a \sim \mathrm{Default}\, k \vdash c : \mathrm{Constraint},\ M}{\Gamma, a : k \in \mathrm{FV}(c), \mathrm{Default}\, k \vdash c : \mathrm{Constraint},\ M \cup \{m_{\mathrm{def}}\}}\ \textsc{Default}$$

$$\frac{\Gamma, \mathrm{Ignore}\, c \vdash \mathrm{Ignore}\, c \sim \mathrm{Msg}\, m,\ M}{\Gamma, \mathrm{Ignore}\, c \vdash c : \mathrm{Constraint},\ M \cup \{m\}}\ \textsc{Ignore}$$

$$\frac{\Gamma, \mathrm{Discharge}\, a\, b \vdash \mathrm{Discharge}\, a\, b \sim \mathrm{Msg}\, m,\ M}{\Gamma, \mathrm{Discharge}\, a\, b \vdash a \sim b,\ M \cup \{m\}}\ \textsc{Discharge}$$

$$\frac{\Gamma \vdash c : \mathrm{Constraint},\ M_c \qquad \Gamma \vdash m_a \sim m_b,\ M}{\Gamma \vdash \mathrm{OnlyIf}\, c\, m_a \sim m_b,\ M_c \cup M}\ \textsc{OnlyIf}$$

Figure 4.1: The typing rules that WRIT extends GHC's type system with. The judgement $\Gamma, \mathrm{F}\, a_1 \ldots a_n \vdash c, M$ here judges that with an instance $\mathrm{F}\, a_1 \ldots a_n$ in the context the constraint (or equality) c holds with the set of output messages M. Here, we write $c : \mathrm{Constraint}$ to denote a well formed constraint c, and m_{def} is a compiler generated message based on the source expression.

4.2.5 Ensuring Runtime-Irrelevance

Since it is not always safe to ignore or discharge, we allow users to recover some safety by using the `OnlyIf` constructor, as used above in the `Discharge` instance for `(*)` to assert `Coercible`. The OnlyIf rule is used to unravel $\mathrm{F}\, a_1 \ldots a_n \sim \mathrm{Msg}\, m_b$' when $\mathrm{F}\, a_1 \ldots a_n$ reduces to an $\mathrm{OnlyIf}\, c\, m_a$, and adds the additional constraints c and $m_a \sim m_b$ as obligations. This eventually results in an equality of the form $\mathrm{Msg}\, m_a \sim \mathrm{Msg}\, m_b$, causing GHC to unify m_b with m_a, inferring the message to be emitted. Note that $\mathrm{OnlyIf}\, a\, b$ only holds if both a and b hold, and b is only emitted if a holds.

4.2.6 Turning Type-Errors into Warnings

To model the fact that we often want to turn type-errors into warnings, all our rules produce a set of messages, M, which is a union of the messages produced by any obligations. The Discharge and Ignore rule add a user defined message to the set, whereas the Default rule adds a standardized message. The user defined messages are built using GHC's user type-error combinators, which allows them to use type families to compute the mes-

sage [15]. The resulting set of messages is reported as warnings at the end of type checking, or alternatively as type-errors, if the user passes the plugin the `keep-errors` flag.

4.3 Implementation

WRIT operates by examining the wanted and derived constraints passed to the plugin by GHC. Messages are handled as a set of logs with type variables for the messages and their origin. The logs are finalized before they are output, with the type variables representing messages are replaced with the messages themselves.

The plugin applies the Default rule by generating constraints of the form `a` $\sim$ `Default k` for any free type variable `a` of kind `k` in unsolved constraints, Then, e.g. `Default Label` will reduce to `L`, and the variable `a` is set to `L` in the context. In Haskell there are two types of type variables, rigid and flexible. Rigid type variables are variables mentioned in the givens, i.e. the constraints. Flexible type variables are type variables instantiated from a `forall`. For example, in `return :: Monad m => a -> m a`, `m` is a rigid type variable, while `a` is flexible type variable. When we default a type variable, we must distinguish between rigid and flexible type variables: for rigid type variables, the generated constraints take the form of a given, with assertion from WRIT that `a` is equivalent to `Default Label` as the evidence. For flexible type variables, we do not require evidence, so it suffices to emit a derived to unify `a` with `Default Label`.

For the Ignore rule, the plugin asserts that the constraint holds, which corresponds to the empty typeclass having an instance. It also emits a constraint that applying the `Ignore` family to the constraint results in a message wrapped in the `Msg` constructor, and adds it to the set of messages as a new type variable that will unify with the message itself.

Similarly for the Discharge rule, WRIT generates a proof by assertion that `a` $\sim$ `b` holds (e.g. `L` $\sim$ `H`), and adds the obligation that Discharge $a\,b$ reduces to a Msg m, with the fresh flexible type variable m added to the set of messages M. The evidence is an assertion in the form of a zero-cost coercion [2], which is safe for runtime-irrelevant types which have the same runtime representation.

WRIT applies the OnlyIf rule by generating an assertion that OnlyIf $c\, m_a \sim m_b$ and checking that both c and $m_a \sim m_b$ hold. As an optimization, we solve equalities of the form:

$$\text{OnlyIf}\, c_1\, (\text{OnlyIf}\, c_2\, (\ldots (\text{OnlyIf}\, c_n\, \text{Msg}\, m_a))) \sim \text{Msg}\, m_b$$

by checking all the constraints $c_1, \ldots, c_n$ and $\mathrm{Msg}\, m_a \sim \mathrm{Msg}\, m_b$, causing GHC to unify m_a with m_b.

4.4 Conclusions and Future Work

We presented WRIT, a type checker plugin for GHC to weaken the type checking process for runtime-irrelevant constraints and representationally-equivalent types. We believe our work will facilitate developers to adopt more secure programming practices in Haskell with less overhead, since it is now possible to start doing so in a more gradual manner. As this is a work in progress, there are a few avenues for future work:

Safety

The WRIT plugin gives users a lot of freedom and allows them to override the typing rules used in Haskell. We have yet to investigate which rules can be safely defined by the user, what can go wrong if they define an invalid rule, and whether we can prevent users from defining such rules.

Overlaps

Neither the compiler plugin nor the formalization deal with what happens when the user defined instances overlap, which can cause the typing rules of WRIT to overlap and it is unclear which one to choose. In the plugin itself, this is handled by preferring Discharge to Ignore and Ignore to Default. It is clear however that the choice should not affect the semantics of the compiled program (something yet to be proven), but which typing rule is preferred can affect the errors or warnings emitted in the process. One possibility is to design a heuristic that selects the most specific typing rule applicable, to emit more concrete (and useful) messages, as opposed to more generic ones.

Dynamic and Gradual Typing

We want to investigate how relaxing the type checking process could interact with Haskell's dynamic typing capabilities [11]. Whenever the type checker finds two expression producing a type mismatch error, it might be possible to promote them both to Haskell's dynamic representation, `Dynamic`. In this light, the invalid list expression `[42, "hello"]` could be promoted to a list of dynamic values by promoting both `42` and `"hello"` to a unified dynamic representation, i.e. `[42, "hello"] :: [Dynamic]`. Then,

dynamically typed values could be demoted to concrete types via runtime checks inserted automatically. This mechanism could shorten the gap between Haskell, a strongly typed language, and dynamically typed languages like Python or Erlang by simply toggling a compiler plugin, enabling us to do module-based *gradual* typing [18].

Bibliography

[1] J. Bracker and A. Gill. Sunroof: A monadic dsl for generating javascript. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324*, PADL 2014, page 65–80, Berlin, Heidelberg, 2014. Springer-Verlag.

[2] J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 189–202, New York, NY, USA, 2014. Association for Computing Machinery.

[3] J. Cheney and R. Hinze. Phantom types. Technical report, Cornell University, 2003.

[4] I. S. Diatchki. Improving haskell types with smt. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, page 1–10, New York, NY, USA, 2015. Association for Computing Machinery.

[5] 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.

[6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, Mar. 1996.

[7] S. P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, pages 1–16, 1997.

[8] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, DSL '99, page 109–122, New York, NY, USA, 2000. Association for Computing Machinery.

[9] G. Mainland and G. Morrisett. Nikola: Embedding compiled gpu functions in haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, page 67–78, New York, NY, USA, 2010. Association for Computing Machinery.

[10] D. Otwani and R. A. Eisenberg. The thoralf plugin: For your fancy type needs. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 106–118, New York, NY, USA, 2018. Association for Computing Machinery.

[11] J. Peterson. Dynamic typing in haskell. Technical report, Technical Report YALEU/DCS/RR-1022, Yale University, Department of Computer Science, 1993.

[12] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, page 25–36, New York, NY, USA, 2008. Association for Computing Machinery.

[13] A. Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, page 280–288, New York, NY, USA, 2015. Association for Computing Machinery.

[14] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 51–62, New York, NY, USA, 2008. Association for Computing Machinery.

[15] A. Serrano and J. Hage. Type error customization in ghc: Controlling expression-level type errors by type-level programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[16] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27:e5, 2016.

[17] G. Team. The ghc-8.10.1 library Constraint module, aug 2020.

[18] M. Toro, R. Garcia, and E. Tanter. Type-driven gradual security with references. *ACM Trans. Program. Lang. Syst.*, 40(4), Dec. 2018.

[19] D. Vytiniotis, S. Peyton jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4–5):333–412, Sept. 2011.

[20] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 53–66, New York, NY, USA, 2012. Association for Computing Machinery.