PropR: Property-Based Automatic Program Repair

Matthías Páll Gissurarson[*]
Chalmers University of Technology
Gothenburg, Sweden
pallm@chalmers.se

Leonhard Applis[*]
TU Delft
Delft, Netherlands
L.H.Applis@Tudelft.nl

Annibale Panichella
TU Delft
Delft, Netherlands
A.Panichella@Tudelft.nl

Arie van Deursen
TU Delft
Delft, Netherlands
Arie.vanDeursen@Tudelft.nl

David Sands
Chalmers University of Technology
Gothenburg, Sweden
dave@chalmers.se

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.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; *Automatic programming*; Functional languages; Source code generation.

KEYWORDS

automatic program repair, search based software engineering, synthesis, property-based testing, typed holes

ACM Reference Format:
Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. 2022. PropR: Property-Based Automatic Program Repair. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510620

1 INTRODUCTION

Have you ever failed to be perfect? Don't worry, so have automatic program repair (APR) approaches. APR faces many challenges, some inherited from search-based software engineering (SBSE), like overfitting [52, 67], predictive-evaluation in search [73], and duplicate handling [9]. Other challenges are unique to the domain itself, such as deriving ingredients for a fix [41] and producing valid programs [28]. Consequently, APR has open research in all of its core aspects: search-space, search-process, and fitness-evaluation. The research community is shifting its focus towards other solutions, either leaving behind boundaries of search space using generative neural networks [36, 42, 65], or by empirical evidence that fixes are often related to dependencies, not the code itself [4, 14]. Fixes are usually validated by running against the test suite of the program, assuming that a solution that passes all tests is a valid patch. However, Le Goues et al. [54] showed that Program Repair can *overfit*, i.e., that a fix passes the test suite despite removing functionality or just bypassing single tests.

Usually, generated patches are evaluated against a unit test suite of the buggy program [34]. The fitness is defined as the number of failing tests in the suite [40], making a fitness of zero a potential fix. The problem is the quality of the tests — often not all important cases are covered, and the search finds something that passes all tests but doesn't provide all wanted functionality [52]. This is considered an *overfit* repair attempt. A particularly good example for this is the Kali approach [54], that removes random statements of a program. In a later study, Martinez et al. [38] showed that out of 20 of the repair attempts that passed the tests, only one was a real fix. One approach by Yz et al. [71] to address overfitting was to introduce tests generated with EvoSuite [15] to have a stronger test suite, reporting only an improvement in speed, not in found solutions. Unfortunately, EvoSuite introduces a new problem: If the program was faulty (which programs that we are trying to repair are), an automatically generated test suite may assert the faulty behavior and make test-based repairs unable to ever produce a correct program, despite passing the (generated) test suite. Thus, current automated test-case generation is not the be-all and end-all for overfitting in APR.

This work aims to improve APR with addressing the overfitting problem by introducing properties [8] in addition to unit tests. A software property is an attribute of a function (e.g., symmetry, idempotency, etc.) that is evaluated against randomly created instances of input data. Well-written properties often cover hundreds of (unit) tests, making them attractive candidates for fitness evaluation.

[*] The first two authors contributed equally to this paper

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510620

We argue that properties can be an improvement to the overfitting challenge in APR. While property-based testing frameworks exist for a range of languages, the practice is particularly natural for functional programming, and widely used in the Haskell community. Therefore, we implement a tool called PropR, which utilizes properties for Haskell-Program-Repair and evaluate the repair rates and overfitting rates for different algorithms (random search, exhaustive search, and genetic algorithms). Our fixes follow a GenProg-like approach [34] of representing patches as a set of changes to the program, with the major difference that our patch ingredients (mutations) are sourced by the Haskell compiler using a mechanism called *typed holes* [19]. A typed hole can be seen as a placeholder, for which the compiler suggests elements that produce a compiling program. As these suggestions cover all elements in scope (not only those used in the existing code), we overcome to some degree the redundancy assumption [41], i.e., the concept that patches are sourced from existing code or patterns, which is common to GenProg-like approaches.

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

Our contributions can be summarized as follows:

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

The remainder of the paper is organized as follows: Section 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 we present the primary aspects of the repair tool and their reasoning. Section 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 5 and discussed in Section 6. After the threats to validity in Section 7 we summarize the work in Section 8. The shared artifacts are described in Section 9.

2 BACKGROUND AND RELATED WORK

2.1 Property-Based Testing

Property-based testing is a form of automated testing derived from random testing [22]. While random testing executes functions and APIs on random input to detect error states and reach high code coverage, property-based testing uses a developer defined attributes called *properties* of functions that must hold for any input of that function [8]. Random tests are performed for the given property; If an input is found for which the property returns false or fails with an error, the property is reported as *failing* along with the input as a counter example [8]. Some frameworks will additionally *shrink* the counter example using a previously supplied shrinking function to offer better insight into the root cause of the failure [8].

```
prop_1 :: Double -> Test       unit_1 :: Test
prop_1 x =                     unit_1 =
  sin x ~== sin (x + 2*π)        sin π ~== sin (3*π)

prop_2 :: Double -> Test       unit_2 :: Test
prop_2 x =                     unit_2 = sin 0 == 0
  sin (-1*x) ~== -1*(sin x)
                               unit_3 :: Test
prop_3 :: Test                 unit_3 = sin (π/2) == 1
prop_3 = sin (π/2) == 1
                               unit_4 :: Test
prop_4 :: Test                 unit_4 =
prop_4 = sin 0 == 0              sin (-1*π/2) == -1*(sin π/2)

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

Figure 1: Comparison of Properties and Unit Tests for `sin`

There are some variations on property-based testing, e.g. SmallCheck, which performs an *exhaustive test* of the property [58]. QuickCheck approximates this behavior with a configurable number of random inputs (by default 100 random samples). Figure 1 provides an example comparison of properties and unit tests of a sine function. The properties require an argument `Double -> Test` and must hold for any given Double. On any single QuickCheck run, 202 tests are performed, forming a much stronger test suite for a comparable amount of code.

A remaining question is whether one cannot just reproduce these 202 tests by unit tests. For a single seed, this is doable — but it is a special strength of properties that the new tests are randomly generated on demand. We hope this addresses the problem of *overfitting* [52], as there are no *fixed* tests to fit on as long as the seed changes. Furthermore, we stress that maintaining 2 properties is easier than maintaining 200 (repetitive) unit tests.

2.2 Haskell, GHC & Typed Holes

Haskell. Haskell is a statically typed, non-strict, purely functional programming language. Its design ensures that the presence of side effects is always visible in the type of a function, and it is typical programming practice to cleanly separate code requiring side effects from the main application logic. This facilitates a modular approach to testing in which program parts can be tested in isolation without needing to consider global state or side effects. Haskell's rich type system and type classes allow tools such as QuickCheck [8] to efficiently test functions using properties, where the inputs are generated by QuickCheck based on a generator for a given datatype.

Valid Hole-Fits. Our tool is based on using the Glasgow Haskell Compiler (*GHC*), which is widely used in both industry and academia. GHC has many features beyond the Haskell standard, including a feature known as *typed holes* [19]. A "hole", denoted by an underscore character (_), allows a programmer to write an incomplete program, where the hole is a placeholder for currently missing code.

Using a hole in an expression generates a type error containing contextual information about the placeholder, including, most importantly, its inferred type. In addition to contextual information, GHC suggests some *valid hole-fits* [19]. Valid hole fits are a list of identifiers in scope which could be used to fill the holes without any type errors. As a simple example, consider the interaction with the GHC REPL shown in Figure 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 2: Example code with a hole and its valid hole-fits

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

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

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

Hole-Fit Plugins. By default, GHC considers every identifier in scope as a potential hole-fit candidate, and returns those that have a type corresponding to the hole as hole-fits. However, users might want to add or remove candidates or run additional search using a different method or external tools. For this purpose, GHC added hole-fit plugins [17], which allows users to customize the behavior of the hole-fit search. When using GHC as a library, this also allows users to extract an internal representation of the hole-fits directly from a plugin, without having to parse the error message.

2.3 GenProg, Genetic Program Repair & Patch Representation

Search-based program repair centered mostly around the work of Le Goues et al. [34] in GenProg, which provided genetic search for C-program repair. One of the primary contributions was the representation of a patch as a change (addition, removal, or replacement) of existing statements. Genetic search is based around the mutation, creation and combination of *chromosomes* — the basic building bricks of genetic search. A chromosome of APR is a list of such changes rather than a full program (AST), making the approach lightweight. Utilizing changes is based on the *Redundancy Assumption* [32], i.e., assuming that the required statements for the fix already exists. The code might just use the wrong variable or miss a null-check to function properly. This assumption has been verified by Martinez et al. [41], showing that the redundancy assumption widely holds for inspected repositories. We adopted the patch-representation in our tool, but were able to weaken the redundancy assumption (see Section 3).

Since GenProg, much has been done in genetic program repair [11] mostly for Java. Particularly Astor [39] enabled lots of research [61, 66, 69, 70] due to its modular approach, as well as real-world applications [59, 62]. This modularity, mostly the separation of fault localization, patch-generation and search is a valuable lesson learned by the community that we adopted in our tool. Compared to this body of research, our scientific contributions lie within the patch-generation and the search-space (see Section 3.1).

2.4 Repair of Formally Verified Programs & Program Synthesis

Another field of research dominant in functional programming is formal verification, in which mathematical methods are used to prove the correctness of programs. Due to its strengths it has been widely applied to various tasks, such as hardware-verification [26], cryptographic protocols [43] or lately smart contracts [6]. But formal verification has also been applied to the domain of program repair and synthesis [30, 60], and some languages can arguably be considered synthesizers around constraints (e.g. Prolog). Using specification-based synthesis in combination with a SAT solver can be effective, however the accuracy is closely tied to the completeness of the post-condition constraints [20]. For Haskell, these approaches revolve around *liquid types*, which enrich Haskell's type system with logical predicates that are passed on to an SMT solver during type checking [48, 56, 57, 64]. The existing approaches [21, 25, 50] focus primarily on the search-aspects of program synthesis due to the (infinite) search space and often perform a guided search similar to proof-systems. The approach used in the *Lifty* [51] language is especially relevant: Lifty is a domain-specific data-centric language in which applications can be statically and automatically verified to handle data specified as per declarative security policies, and suggest provably correct repairs when a leak of sensitive data is detected. Their approach differs in that they target a domain-specific language and focus on type-driven repair of security policies and not general properties. Another interesting approach is the TYGAR based Hoogle+ API discovery tool, where users can specify programming tasks using either a type, a set of input-output tests, or both, and get a list of programs composed from functions in popular Haskell libraries and examples of behavior [24]. It is however focused on API discovery and not program repair, although incorporating Hoogle+ into PropR is an interesting avenue for future work. The approach by Lee et al. [35] is in many ways similar; They also operate on student data and find very valuable insights from repair and identical challenges. The approach they developed (FixML) exploits typed holes to align buggy student programs with a given instructor-program based on symbolic execution. FixML is different as it requires a gold standard, and synthesizes by type-enumeration after symbolic execution. To some degree, this is similar to our implementation of an exhaustive search. Semantics-based repair using symbolic-execution like that of *Angelix* [44] can be very effective in fixing real-world bugs, and uses symbolic expressions similarly to our typed-holes. However, there are some scalability concerns for symbolic execution, and while they can be mitigated using a carefully chosen number of suspicious expression and their derived angelic forests [44], they can also be mitigated using genetic algorithms and the more lightweight property-based analysis, motivating their usage in PropR. Compared to program synthesis, program repair is better able to take advantage of a "reasonable" baseline program from the developers.

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

3 TECHNICAL DETAILS — PROPR

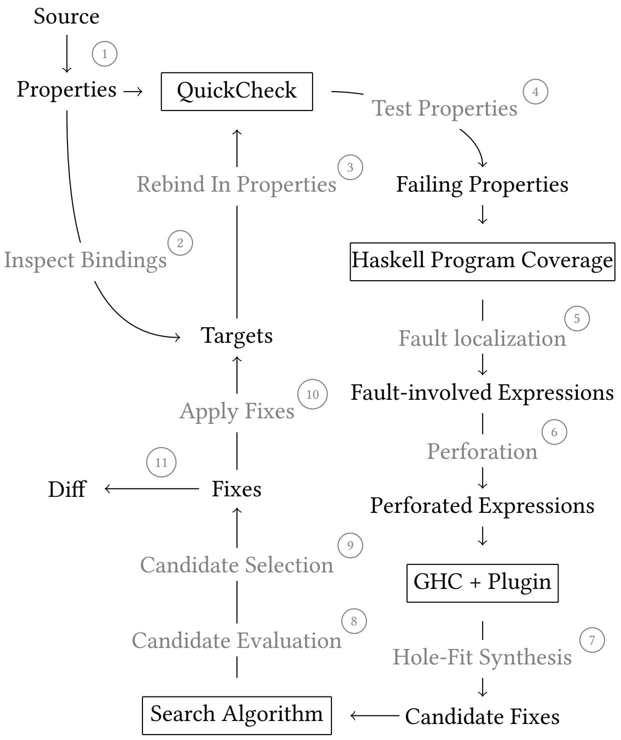

Figure 3: The PropR test-localize-synthesize-rebind loop

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 plug-ins 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 3.4 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. The circled numbers ⓝ in this section refer to the labels in Figure 3.

```
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 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.

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 4.

3.1 Compiler-Driven Mutation

To repair a program, we use GHC to parse and type-check the source into GHC's internal representation of the type-annotated Haskell AST. By using GHC as a library, we can interact with GHC's rich internal representation of programs without resorting to external dependencies or modeling. We determine the tests to fix by traversing the AST for top-level bindings with either a type (`TestTree`) or name (`prop`) that indicates it is a test ①. We use GHC's ability to derive data definitions for algebraic data types [17] and the Lens library [27] to generate efficient traversals of the Haskell AST. To determine the function bindings to mutate, we traverse the ASTs of the properties and find variables that refer to top-level bindings in the current module ②. We call these bindings the *targets*.

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

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

```
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 5: The parametrized properties for len

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

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

Figure 6: The parametrized properties applied to a different implementation of `len`, the standard library `length`

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 4 in Figure 5.

The new properties in Figure 6, `abc_prop` and `double_prop` will now behave the same as the original `prop_abc` and `prop_dup`, but with every instance of `len` replaced with `length`:

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

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

Fault localization. PropR uses an expression-level fault localization spectrum [1], to which we apply a binary fault localization method (touched or not touched by failing properties). A notable difference to other APR tools like Astor is that we can perform fault localization for the *mutated* targets. This enables PropR to adjust the search space once a partial repair has been found, i.e. one that passes a new subset of the properties. Since fault localization is expensive, by default we only perform it on the initial program similarly to Astor [39, 40]. GHC's *Haskell Program Coverage* (HPC) can instrument Haskell modules and get a count of how many times each expression is evaluated during execution [18]. Using QuickCheck, we find which properties are failing and generate a counterexample for each failing property ④. For properties without arguments (essentially unit tests), we do not need any additional arguments, so we can run the property as-is: the 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 [19] ⑦. We present a few examples of the perforated versions of `len` in Figure 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 7: A few perforated versions of `len`. N.B. the empty branch is not perforated, as it is not involved in the fault

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

We derive hole-fit candidates directly from GHC's valid hole-fits, as seen in Figure 8, giving rise to the fixes in Figure 9. These take the form of an identifier (e.g., `sum`), or an identifier with additional holes (e.g., `foldl1 _`) for refinement fits.

Since we synthesize only well-typed programs, we cannot use refinement hole-fits directly: the resulting program would produce a typed hole error. To use refinement hole-fits, we recursively synthesize fits for the holes in the refinement hole-fits up to a depth

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

Figure 9: Candidate fixes derived from the valid hole-fits in Figure 8. The location refers to `product` in `len`

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

To be able to find fixes that include constants (e.g., `String` or `Int`) or fixes that would otherwise require a high and deep refinement level, we search the program under repair for *expression candidates* [37]. These are injected into our custom hole-fit plugin and checked whether they fit a given hole using machinery similar to GHC's valid hole-fit synthesis, but matching the type of an expression instead of an identifier in scope. In our example, these would include `0`, `(1 :: Int)`, `(x ++ x)`, and more. For each expression candidate, we then check that all the variables referred to in the expressions are in scope, and that the expression has an appropriate type. We also look at *application candidates* of the form `(_ x)`, where `x` is some expression already in the program, and _ is filled in by GHC's valid hole-fit synthesis. This allows us to find common data transformation fixes, such as `filter (not . null)`.

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

3.3 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 9 give rise to the definitions shown in Figure 10. We then construct a checking program that applies the parametrized properties and tests to these new target definitions and compile the result. A simplified example of this can be seen in Figure 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 11 that `len3` and `len7` pass all the properties, meaning that replacing `product` with `length` or `sum` qualifies as a repair for the program.

```
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 10: New targets defined by applying the fixes in Figure 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 len7), quickCheck (prop'_dup len7)]]
-- Evaluates to:
[[False, False],[False, False],[True, True],[False, False]
,[False, False],[False, False],[True, True]]
```

Figure 11: Checking our new targets from Figure 10

3.4 Search

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

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

For the **genetic search**, PropR implements best practices and algorithms common to other tools such as Astor [39] or EvoSuite [15]. A mutation consists of either dropping a replacement of a fix, or adding a new replacement to it. The initial population is created as picking n random mutations. The crossover randomly picks cut points within the parent chromosomes, and produces offspring by swapping the parents' genes around the cut points. We support environment-selection [23] with an elitism-rate [3] for truncation. *Elitism* means that we pick the top $x\%$ percent of the fittest candidates for the next generation, filling the remaining $(100 - x)\%$ with (other) random individuals from the population. We choose random pairs from the last population as parents and perform environment selection on the parents and their offspring. Our manual sampling of repairs-in-progress on the data points showed that genetic search requires high *churn* in order to be effective: changing a single expression of the program usually failed more properties than it fixed. Hence, the resulting configurations for the experiment have a low elitism- and high mutation- and crossover-rate.

Within **random search**, we pick (up to a configurable size) evaluated holes at random and pick valid hole-fits at random with which to fill them. We then check the resulting fix and cache it. The primary reason for using random search is to show that the genetic search is an improvement over *guessing*. Nevertheless, Qi et al. [53] showed that random search sometimes can be superior to genetic search, further motivating its application. Besides, random search is a standard baseline in search-based software engineering to assess whether more "intelligent" search algorithms are needed for the problem under analysis.

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

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

Within the test-localize-synthesize-rebind loop (Figure 3) we perform one generation of genetic search per loop, and after the selection of chromosomes the program is re-bound and coverage re-evaluated. The authors observed that this is a bit over-engineered for small programs — the fault localization did not greatly change when the programs had only a single failing property. As an optimization, we added a flag to skip the steps ⑤ to ⑦ in the loop to speed up the actual search. This configuration was enabled during experiments presented in Section 4. The exhaustive and random search do not perform any rebinding.

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.

4 EMPIRICAL STUDY

4.1 Research Questions

Given the concepts presented in Section 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.

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

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

Research Question 1

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

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

Research Question 2

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

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

Research Question 3

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

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

Table 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 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 18) or about simplification (seen in Figure 13), which are core-parts of the assignment. The calculator makes a particularly good example for properties, as attributes such as commutativity, associativity etc. are easy to assert but harder to implement. Hence, we argue that the calculator-exercise makes a case for typical programs that implement properties (i.e., they are not *artificially* added for APR).

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

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

Figure 13: A property asserting the idempotency of simplify

The anonymized data is provided in the reproduction package.

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

RQ3 can (to the best of our knowledge) only be answered by human evaluation. Existing research on automatic patch-validation by Qi [68] requires an automatic test-generation framework (which is not available for Haskell) as well as a gold-standard fix to work as an oracle. They used existing git-fixes as oracles, but we expect some data points to be correct despite not matching the sample-solution. Similarly, work by Nilizadeh et al. [46] utilizes formal verification to automatically verify generated patches, but unfortunately, no specifications were available for the dataset. Instead, we perform the analysis manually, similar to [54] and [38]. As there are too many results to manually inspect, we sampled 70 fixes[1] and let two authors label them as *overfit* or *not overfit*. The authors do so based on their domain-knowledge and in accordance with a given gold-standard. On disagreement, the authors provide a short written statement before discussing and agreeing on the fix-status. The conclusion of the discussion is also documented with a short statement. The manual labels as well as the statements are shared within the replication package.

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 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 14 and Figure 15.

For every entry, we performed a Fisher exact test based on the repair per seed of every test suite. The contingency tables are based on whether the specific seed found patches for the test suite. It

[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 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

showed that 4 of the 13 repaired entries were significantly better in producing repairs with properties (E1, E3, E4, and E14 from Table 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.

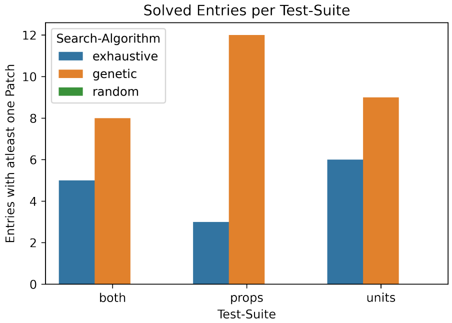

Figure 14: Solved Entries per Test-Suite and Algorithm

Figure 14 shows genetic search outperforming exhaustive search in any test suite configuration, and most effectively for properties.

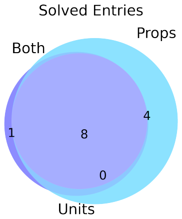

Figure 15: Venn-Diagram of Solved Entries per Suite

Figure 15 shows the overlap of *solved* entries by test suite. It shows that four entries were uniquely solvable by using only properties and one entry was uniquely solvable by the combined test suite. All entries solved by unit tests have also been solved by the properties. This does not necessarily imply that properties are *better* — the patches can still be overfit and are to be evaluated in RQ3.

Summary RQ1

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

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

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

An overview of the time-to-first-result can be seen in Figure 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.

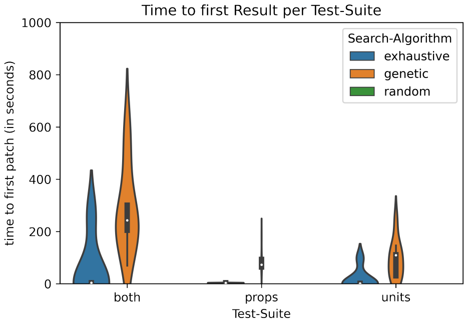

Figure 16: Distribution of Time to First Patch per Entry

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%)

[2] Wilcoxon-RankSum with *less*

and better than GenProg [38](1-4%). Arja [72] reports an effective repair rate of 8% which we slightly outperform.

A typical example found by manual inspection was adding space-stripping to the *addition*-case of showExpr, as seen in Figure 17.

```
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 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 18: The unit test corresponding to the fix in Figure 17

There is a single unit test (see Figure 18) to assert a printed addition without spaces. Within the patch only the "+" case gets *repaired* — this is due to the precedence of the expression which is correctly picked up. Hitherto, the change in the addition actually removes all white-space and correctly passes the test. This (actually) solves the unit test as expected and is therefore arguably not truly overfitting. Nevertheless, a developer would perform the string-stripping on all cases, not only on the addition. Here we see a shortcoming of the test suite — this would have not been possible if we had a property prop_showExpr_printNoSpaces or if we simply had unit tests for all cases. In other data points, where the showExpr had a unified top-level expression (not an immediate pattern match), the repair was successful by adding top-level string-stripping. We would also like to stress the quality of the patch generated despite overfitting: It draws 4 elements (filter, toLower, isSpace, (.)) which were not in the code beforehand and applied them at the correct position.

Another issue observed were empty patches — these appeared when the QuickCheck properties exhibited inconsistent behavior. We suspect a property that tests for the idempotency of simplify seen in Figure 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.

6 DISCUSSION

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

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

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

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

Drastically Increased Search-Space. Due to the novel approach to finding repair candidates, the search space drastically increased as compared to using existing expressions or statements only. This can be seen with the absence of random-search findings. Other studies showed at least some results with random search, sometimes reporting random search as most successful [53]. As we find (many) patches with exhaustive search, the problems are generally solvable with small changes. This implies that the only reason for random search to yield no results is the increased search space.

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

Transference to Java. As Java is the most prominent language for APR, it begs the question of which results can be transferred from Haskell into more mainstream approaches. Properties are supported by JUnit-Plugins[3] and can easily be added to any common test suite and build-tool. The positive effects of properties as presented in Section 5 only require Java programs with sufficient properties. However, the current Java-ecosystems are not utilizing properties; even less sophisticated JUnit-Features, such as parametrized tests, are not widely adopted. This is in stark contrast to functional programming communities, where tools like QuickCheck are popular.

The hole-fitting repair approach cannot be easily reproduced for Java; The JavaC, unlike GHC, is not intended to be used as a library. Nevertheless, Java is strictly typed and the basic hole-fitting-approach can be integrated using meta-programming libraries like Spoon [47]. Many challenges remain: As Java's methods are not pure functions, they cannot be *just transplanted*. Side effects can wreak havoc and just on a technical level polymorphism, that is often only resolvable dynamically, bares huge follow-up-challenges.

But not all is lost for the JVM: Repair approaches that focus on the bytecode [12, 16], can easier adapt hole-fitting. In particular, one could imagine a tool that produces holes for bytecode and introduces the hole-fits utilizing more strict JVM Compilers such as Closure or Scala. We consider this extension a hard but valuable track for further research.

Future Work. The primary research challenge we see is to combine existing approaches with the newly introduced PropR hole-fitting. A hybrid approach that could produce high churn with techniques from Astor [40] or ARJA [72] in combination with the fine-grained changes produced by PropR could solve a broader range of issues. Specific to Haskell is the need to introduce left-hand side definitions, i.e. new pattern matches or functions. These could be provided by generative neural networks [2, 7] and either be used as mutations or as an initial population of chromosomes. Representing multiple types of changes is only a matter of representation within the chromosome — the remaining search, fitness and fault localization can be kept as is.

For fault localization, we currently use *all* the expressions involved in the 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 [14] that provides patch-suggestions on failing pull-requests. We would like to ask maintainers and the public community to give feedback on the quality of repairs, and whether the suggested patches contributed to fault localization or improvements of the test suite even if not added to the code.

7 THREATS TO VALIDITY

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

External Threats. The dataset is based on student data, which could be considered *artificial*. We stress that student data has been used in literature for program repair previously [11, 13, 31, 33]. A real-world study on program such as Pandoc [10] is part of future work. Pandoc, a popular Haskell document-converter, is rich in properties that test e.g., for symmetry over conversions.

8 CONCLUSION

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

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

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

9 ONLINE RESOURCES

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

ACKNOWLEDGMENTS

We thank Matthew Sottile for his feedback on the implementation of PropR, as well as Martin Monperrus for advice on the evaluation. We also thank the reviewers for their insightful feedback.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation. The members of TU Delft were partially funded by ICAI AI for Fintech Research, an ING - TU Delft collaboration.

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

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035 SI: TAIC PART 2007 and MUTATION 2007.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL]

[3] Chang Wook Ahn and R.S. Ramakrishna. 2003. Elitism-based compact genetic algorithms. *IEEE Transactions on Evolutionary Computation* 7, 4 (2003), 367–385. https://doi.org/10.1109/TEVC.2003.814633

[4] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 254–265.

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

[6] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper *(PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[8] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[9] Zhen Yu Ding. 2020. Patch Quality and Diversity of Invariant-Guided Search-Based Program Repair. *arXiv preprint arXiv:2003.11667* (2020).

[10] Massimiliano Dominici. 2014. An overview of Pandoc. *TUGboat* 35, 1 (2014), 44–50.

[11] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. 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)*. https://arxiv.org/abs/1905.11973

[12] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test* (Austin, Texas) *(AST '16)*. Association for Computing Machinery, New York, NY, USA, 85–91. https://doi.org/10.1145/2896921.2896931

[13] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. Technical Report. Universite Lille 1. https://hal.archives-ouvertes.fr/hal-01272126/document

[14] Khashayar Etemadi, Nicolas Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. 2021. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *arXiv preprint arXiv:2103.12033* (2021).

[15] Gordon Fraser and Andrea Arcuri. 2011. 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* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[16] Ali Ghanbari and Lingming Zhang. 2019. PraPR: Practical Program Repair via Bytecode Mutation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1118–1121. https://doi.org/10.1109/ASE.2019.00116

[17] GHC Contributors. 2021. GHC 8.10.4 users guide. https://downloads.haskell.org/~ghc/8.10.4/docs/html/users_guide/index.html

[18] Andy Gill and Colin Runciman. 2007. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1291201.1291203

[19] Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 179–185. https://doi.org/10.1145/3242744.3242760

[20] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–188.

[21] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (dec 2019), 28 pages. https://doi.org/10.1145/3371080

[22] Richard Hamlet. 1994. Random testing. *Encyclopedia of software Engineering* 2 (1994), 971–978.

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

[24] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 205 (nov 2020), 27 pages. https://doi.org/10.1145/3428273

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

[26] Christoph Kern and Mark R. Greenstreet. 1999. Formal Verification in Hardware Design: A Survey. *ACM Trans. Des. Autom. Electron. Syst.* 4, 2 (apr 1999), 123–193. https://doi.org/10.1145/307988.307989

[27] Edward Kmett. 2021. The lens library. https://hackage.haskell.org/package/lens

[28] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair?. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 194–204.

[29] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings, 06301)*, Rainer Koschke, Ettore Merlo, and Andrew Walenstein (Eds.). Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2007/962

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

[31] Claire Le Goues, Yuriy Brun, Stephanie Forrest, and Westley Weimer. 2017. Clarifications on the Construction and Use of the ManyBugs Benchmark. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1089–1090.

[32] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software Quality Journal* 21, 3 (2013), 421–443. https://doi.org/10.1007/s11219-013-9208-0

[33] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[34] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[35] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018. Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 158 (oct 2018), 30 pages. https://doi.org/10.1145/3276528

[36] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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*. 101–114.

[37] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 48–61. https://doi.org/10.1145/1065010.1065018

[38] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4 arXiv:1811.02429

[39] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705

[40] Matias Martinez and Martin Monperrus. 2019. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software* 151 (2019), 65–80. https://doi.org/10.1016/j.jss.2019.01.069 arXiv:1802.03365

[41] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. 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* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 492–495. https://doi.org/10.1145/2591062.2591114

[42] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. *arXiv preprint arXiv:2103.11626* (2021).

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

[44] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[45] Geoffrey Neumann, Mark Harman, and Simon Poulding. 2015. Transformed Vargha-Delaney Effect Size. In *Search-Based Software Engineering*, Márcio Barros and Yvan Labiche (Eds.). Springer International Publishing, Cham, 318–324.

[46] Amirfarhad Nilizadeh, Gary T. Leavens, Xuan-Bach D. Le, Corina S. Păsăreanu, and David R. Cok. 2021. Exploring True Test Overfitting in Dynamic Automated Program Repair using Formal Methods. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 229–240. https://doi.org/10.1109/ICST49551.2021.00033

[47] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[48] Ricardo Peña. 2017. An introduction to liquid haskell. *arXiv preprint arXiv:1701.03320* (2017).

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

[50] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[51] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *Proc. ACM Program. Lang.* 4, ICFP, Article 105 (aug 2020), 30 pages. https://doi.org/10.1145/3408987

[52] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *2013 IEEE International Conference on Software Maintenance*. 180–189. https://doi.org/10.1109/ICSM.2013.29

[53] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. 254–265.

[54] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[55] Michel Raymond and Francois Rousset. 1995. An Exact Test for Population Differentiation. *Evolution* 49, 6 (1995), 1280–1283. http://www.jstor.org/stable/2410454

[56] Patrick Redmond, Gan Shen, and Lindsey Kuper. 2021. Toward Hole-Driven Development with Liquid Haskell. *arXiv preprint arXiv:2110.04461* (2021).

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

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

[59] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.

[60] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/1706299.1706337

[61] Chadi Trad, Rawad Abou Assi, Wes Masri, and Fadi Zaraket. 2018. CFAAR: Control Flow Alteration to Assist Repair. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 208–215.

[62] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. 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)*. IEEE, 95–104.

[63] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[64] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. *SIGPLAN Not.* 52, 10 (sep 2017), 63–74. https://doi.org/10.1145/3156695.3122963

[65] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163* (2017).

[66] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172* (2017).

[67] Qi Xin. 2017. Towards Addressing the Patch Overfitting Problem. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 489–490. https://doi.org/10.1109/ICSE-C.2017.42

[68] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2017), 226–236. https://doi.org/10.1145/3092703.3092718

[69] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670.

[70] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825.

[71] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2017. Test case generation for program repair: A study of feasibility and effectiveness. *arXiv preprint arXiv:1703.00198* (2017).

[72] Yuan Yuan and Wolfgang Banzhaf. 2017. ARJA: Automated repair of Java programs via multi-objective genetic programming. *arXiv* 46, 10 (2017), 1040–1067. arXiv:1712.07804

[73] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. 2018. An investigation of compression techniques to speed up mutation testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 274–284.