# Mutation testing effectiveness for checking QuickCheck tests derived from Agda type signatures

**Mateusz Pietrzak** [1]

**Supervisor(s): Jesper Cockx** [1]**, Nathaniel Burke** [1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Mateusz Pietrzak
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Nathaniel Burke, Annibale Panichella

## Abstract

agda2hs is a compiler that allows compilation of dependently typed Agda language into readable Haskell modules. This has a benefit that a formal proof of correctness can be carried out on Agda side, and then used in more realistic context in Haskell. However, sometimes it might be beneficial to compile yet unproven functions to Haskell, to be able to use traditional testing methods to quickly discard incorrect implementations. In this paper, I explore the usefulness of using mutation testing in validating such property tests. I present modifications made to the existing Haskell MuCheck library to accommodate this workflow: a QuickCheck test adapter, an annotation mechanism that links tests to the functions they cover, and a way to mark already proven functions so they are not mutated. I evaluate the tool on several test cases, both written by hand and generated from Agda, including sorting algorithms and lambda calculus with De Bruijn indices. I find that the share of mutants equivalent to the original code varies greatly between functions, which makes the ratio of killed mutants hard to interpret on its own. The current implementation does not prove very practical at this stage, and I propose some improvements that could make this a useful tool for this particular workflow.


# 1 Introduction

Mutation testing is a technique of verifying the strength of a test suite. It modifies the tested code in an automated manner, tweaking its behavior, and checking if the tests fail on the "mutant" functions [6]. Provided the mutant generations were carried out well, surviving mutants can be an indication of a poorly designed test suite, hinting at potential problems that can slip through. In this paper, I look into integrating this technique to enable validation of property tests generated in Haskell using agda2hs. This tool allows users to write parts of a Haskell code base in Agda complemented with proofs of correctness, and later use clean, functionally equivalent Haskell code in the rest of the code. The main research questions of this paper are

- How is mutation testing already being used in Haskell and other languages for more general use?
- How to extract Haskell's Abstract Syntax Tree of functions we wish to test with mutations?
- What functions are fit for being mutated, and how can the programmer impact and tweak which parts of the code undergo mutation?
- Can the resulting mutations help identify too lenient properties that allow incorrect code to slip through?

To that end, I forked an already existing proof-of-concept implementation of mutation testing in Haskell called MuCheck, and modified it to fit better into the investigated pipeline.

I conclude, that while the technique is interesting, more work would have to be done on both the theoretical and implementation side to make it truly useful, though some success in identifying poor test suites is already possible.

## 1.1 Structure

First, in the section 2, I give more background to the problem, going a bit deeper into what agda2hs is and how it can be useful, why would we want to generate property tests, and more in depth on what mutation testing is. section 3 talks about related work, and how it forms a base for the rest of the research. Next, the section 4 goes deeper into my work, exploring the implementation of the tooling. section 5 presents the test cases, the results, and their analysis. In section 6, I briefly discuss the topic of responsible research in the context of this project, and finally section 7 ends the paper with conclusions and future recommendations.

# 2 Background

This section presents groundwork for understanding the rest of the paper.

## 2.1 Dependently typed languages and agda2hs

Ensuring the correctness of code can be crucial, especially in domains where such correctness is critical, such as healthcare sector. Strongly typed languages already give some degree of confidence in the code, reducing undefined behaviors and runtime exceptions, but do very little to actually verify the correctness of the code. Testing, on the other hand, does not prove the correctness, and is better suited to showing that there are bugs present, or to catch some regressions made during later development.

To address those issues, so-called dependently typed languages, such as Agda [2], or Rocq [1] (previously Coq) are being developed. The type systems of those languages are much more expressive, allowing more of the programmer's intent to be encoded in just the type signature. Additionally, they allow for proofs to be embedded into the code itself. This lets the programmer have full confidence in the program outcome, as such proof carries a mathematical certainty about the correctness of a particular program.

This however, does not come without drawbacks. One of them is that by the niche nature of this class of languages, there is not a lot of useful code or an ecosystem around them. Additionally, writing the proofs can be quite cumbersome and difficult, especially since trying to prove correctness of a program that is in fact incorrect can waste a lot of programmer's time and effort.

To address the first of these problems, agda2hs project has been created [5]. It is a compiler which allows for translation of Agda modules into readable and usable Haskell code. This allows users to still write safe and verified code, while being able to benefit from the rich ecosystem of the Haskell programming language. It also lets users rewrite parts of preexisting Haskell projects, in critical points which need the correctness in Agda, and use them as a drop-in replacement without any issues with interfacing.

### 2.2 QuickCheck

Still, the problem of trying to prove everything in Agda remains. If a function written by a user in Agda is incorrect, they might spend a significant amount of time on a proof of correctness for a function, where no such proof is possible. That's why our group is investigating broadly the process of creating QuickCheck property tests and using them as additional verification on the Haskell side. QuickCheck is a popular Haskell library that through randomized search tries to disprove a property set by the programmer [4]. It does not provide a proof of correctness, but might help to find some broken edge cases early, and can serve as an additional assurance in the case of additional properties, which might be difficult to prove formally.

### 2.3 Mutation testing

While the failing tests give us confidence that the code has problems and must therefore be fixed, the same cannot be said the other way around: passing tests are not proofs of correctness. I therefore investigate mutation testing to increase the programmer's confidence in the strength of a test suite, either automatically generated from various type annotations in Agda, or written there manually. The high level intuition is as follows: the framework takes the tested modules, automatically tweaks parts of the function's execution, and reruns the test suite. If the tests now fail, it should indicate that the test suite is likely rigorous, as small tweaks to how the function works is caught by the test suite. If the tests continue to pass however, that may be an indication of lenient tests that allow a larger domain of functions to pass than required.

## 3 Related work

For more mainstream languages with greater adoption than Haskell, there exist actively maintained mutation libraries, such as Pitest for Java or Stryker for JavaScript, C# and Scala. For Haskell, there currently exists only one working implementation, MuCheck [9]. While working to an extent, the framework is very limited. Throughout the project, I adapted it to fit better into the aforementioned `agda2hs` pipeline, which is described in more detail in section 4.

### 3.1 Challenges of mutation testing

Implementing mutation testing in a practical setting presents challenges, though research has emerged to address these limitations to some extent. First, how does the program carrying out mutations know which parts of the code are relevant to be mutated? Actively developed libraries handle this by relying on per-test coverage analysis. The framework analyzing the test suite runs the tests one after another, recording the visited paths, marking them as spots fit for mutation. MuCheck has a coarser implementation of this idea in place, where the coverage is recorded on a per-module basis.

Another issue, that stems from the very idea of the method, is the large time overhead. In large codebases, simply running the test suite once can take a significant amount of time. Mutation testing increases that time many times over, as tests need to be ran for each generated mutant. There are numerous techniques to alleviate that issue to some extent, for example by sampling only some mutants, or by running only parts of the function to catch changed behavior sooner [8]. I however do not explore this issue deeper here, as this was outside of the scope of my implementation, and all test cases were small enough to compute the mutants in reasonable time.

Finally, checking equivalence between two mutated modules, or between a mutated and original module is in general undecidable. This means, that one can never be sure that the change made by the framework actually changed the function, or that two independent mutations did not create equivalent mutants. There exist some algorithms that aim to mitigate this problem, but these techniques were out of the scope for me for this project [10].

## 4 Implementation

This section details the implementation part of the project, showing how I adapted MuCheck to better fit the aforementioned `agda2hs` pipeline. All of the work is publicly available on GitHub[1].

### 4.1 MuCheck

```
-- Example input function
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x : xs) = qsort l ++ [x] ++ qsort r
    where l = filter (< x) xs
          r = filter (>= x) xs

-- Example mutant 1
qsort :: [Int] -> [Int]
qsort [] = []

-- Example mutant 2
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x : xs) = qsort l ++ [x] ++ qsort r
  where l = filter (< x) xs
        r = filter (<= x) xs

-- Example mutant 3
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x : xs) = qsort l ++ [x] ++ qsort r
  where l = filter (<= x) xs
        r = filter (>= x) xs
```

Figure 1: Example of an original function together with some of the automatically generated mutants

MuCheck is a tool written for mutation testing in Haskell [9]. It allows users to specify a single Haskell module, including both the implementation to be tested, and the tests themselves, then perform the mutation pipeline. The library first classifies the functions between tests and module code, and creates mutated versions of the module in a temporary folder. Example of an original function together with some example mutants can be found in Figure 1. Then, each mutant module

[1] https://github.com/MateuszPietrzak/mucheck

is compiled to bytecode, and interpreted to check for killed/alive tests. Out of the box, the library only provides a simple test adapter which takes a boolean value, and returns a value of "successful" or "failed". The adapter then interprets these results in an aggregated summary.

The original Bitbucket repository has not been updated since 2015, but a GitHub fork has some updates from 2024. Still, the project is not actively maintained, and with 7 stars on GitHub, does not seem to be widely used. I was unable to run the tool aided with the coverage checking information, and could not get the provided example to execute. Still, the codebase provided a significant value, as the part of the code generating the mutated modules worked well, not requiring adaptation from my side.

### 4.2 Supporting libraries

MuCheck's internal implementation and my later modifications and features depend on two main libraries: `haskell-src-exts` [3] and `hint` [7]. `haskell-src-exts` is a parser for the Haskell syntax. It allows to represent the abstract syntax tree as a type-safe structure in code, allowing easy interpretation and modification of the code. A naive approach, utilizing for instance regular expressions, would allow much less control, while being more error prone. `hint` is a bytecode interpreter based on the GHC compiler for Haskell. It allows dynamic loading and evaluation of Haskell code as string at program runtime. This is useful for the automatic mutation pipeline, for running the generated mutants. Otherwise new system threads running the modules would be necessary, increasing the complexity of the program.

### 4.3 Implemented features

In order to be able to carry out useful tests in the context of using it in tandem with the `agda2hs` and the QuickCheck properties, I implemented a few additional features into the base MuCheck codebase: a test adapter for QuickCheck, annotation mechanism for specifying tested functions, and a way for some functions to be marked as proven to avoid being mutated.

#### QuickCheck test adapter

As mentioned above, MuCheck has a very simple test adapter that allows users to write a boolean result unit tests. The test functions are then expected to return MuCheck's `AssertStatus`, which is a boolean wrapper over a passed or failed result. This is very limiting already for regular tests, and becomes infeasible for use with QuickCheck. I have thus implemented a dedicated test adapter for this exact purpose. The tests are written as regular properties. This has a benefit of user's module with implementation and tests not depending at all on the MuCheck's interface or adapters, and the module only needs to import QuickCheck, making it convenient to use. Then, after the mutation is carried out, the interpreter loads the module and executes the QuickCheck tests inside of the `hint` interpreter.

#### Test annotations function linking

Originally, the library only requires tests to be marked as a "Test" with an annotation. Haskell allows annotations using an `ANN` keyword in comments, and allows a module or a symbol to be annotated with an arbitrary Haskell expression. Refer to the example in the Figure 2.

```
sortRev :: AssertStatus
sortRev = assertCheck $
    qsort [4,3,2,1] == [1,2,3,4]
{-# ANN sortRev "Test" #-}
```

Figure 2: Annotated test from MuCheck repository's example

```
prop_LenPreserved :: [Int] -> Bool
prop_LenPreserved xs =
    length xs == (length . qsort) xs
{-# ANN prop_LenPreserved "Test qsort" #-}
```

Figure 3: Property test together with specification of the function name being tested

This already allows the framework to know which functions to run and, as an added value, categorizes functions into unannotated for mutation and annotated, that do not get modified. Further, the intended use flow recommends to have a test program in a separate module that executes all tests in a program. This allows for collection of coverage information, which as noted in section 2 is a common way of determining places in the code worth mutating. However, the way it is implemented in MuCheck as of today is very limiting: because such coverage is not collected on a per-test basis, but rather for the entire test suite of the module, in case one module houses multiple functions, each with their own tests, numerous mutations end up happening in functions that are completely irrelevant for some tests that refer to a different part of the code.

Due to the time constraint of the project, I was not able to implement this per-test coverage testing, and opted for a simpler approximation of it. Each test now needs to have the name of the tested function in the annotation. Then, when generating the mutants, the program groups all tests by the tested function and then traverses the syntax tree recursively for each tested function, finding a set of all potentially reachable functions. Though without running the code with coverage one cannot be sure that all functions within this set are truly reachable for a given input, this improves our chance that the mutations are generated in relevant places. Together with the QuickCheck adapter, an example property test is presented in Figure 3.

#### Marking proven functions

The last feature implemented into the MuCheck is allowing for functions to be marked as "Proven", as can be seen in Figure 4. While writing an Agda module, one might wish to write and prove required properties about some simpler functions, and use them in a more complex context. These more involved functions might be what the user in turn wants to compile to Haskell, and experiment with using QuickCheck. When using mutation testing, we do not wish to modify these previously proven functions, as property tests are not made to check their correctness.

```
dropWhileNeq :: Char -> String -> String
dropWhileNeq c = dropWhile (/=c)
{-# ANN dropWhileNeq "Proven" #-}

split :: Char -> String -> [String]
split c xs = xs' : if null xs'' then [] else
    split c (tail xs'')
    where xs' = takeWhile (/=c) xs
          xs''= dropWhileNeq c xs
```

Figure 4: Example of a support function annotated as "Proven"

## 5 Analysis and Results

For the analysis of the tool, I used a combination of manually created Haskell modules, as well as examples generated from Agda code using `agda2hs` provided to me by my peers.

### 5.1 Mutation operators

For all of the mutations, I leverage the implemented mutation operators already provided by the MuCheck library, as this part of the code works well for the sake of this study without modifications. The operators are as follows:

**Function replacement**

The occurrences of common functions are replaced by different ones with identical signature. This can be extended to custom functions within the framework, but here left unchanged. Some examples of changes are ordering operators (like >= or ==) being changed between each other, or simple arithmetic operators.

**Pattern matches**

Especially prevalent in functional programming, pattern matching can be used to avoid messy conditional logic. These pattern matches do not have to represent a disjoint set of cases necessarily, so their order may change the logic of the function, and thus their reordering might change the function outcome. Also, in Haskell it is not actually necessary to exhaustively cover all branches, so the framework may choose to remove some branches outright.

**Integer values**

Changes hard-coded integer values by incrementing, decrementing, or setting them to 0 or 1.

**Switching If/Else**

Replacing the outcomes of the two branches of an if/else expression.

### 5.2 Methodology

To prepare for testing I have come up with a few test case modules, with slightly different properties. Some of them were hand-made in Haskell, while others utilized the `agda2hs` pipeline. For each of the modules, I divided the module property tests into two parts: first one, which already tests some foundational properties, but is not a rigorous suite, and the second one, which included more tests, that cover more of the correctness requirements.

**Simple quick sort**

The first test case which was taken from the MuCheck's example directory is the canonical Haskell implementation of quick sort algorithm. For the tests, the weak property only verifies that the length of the list is preserved, and the strong property additionally verifies that the list becomes sorted after the function is run.

**Insertion sort**

Next, I used a test case provided by one of my peers, alongside the properties. The entire module was generated using `agda2hs`. The implementation derived from the Agda code and QuickCheck property tests created from lemmas stated on the Agda side. Once again, I picked the property about preserving the length for the weak test portion, and the remaining lemmas for the strong part.

**Lambda Calculus with De Bruijn indexing**

To be able to compare how a code generated from Agda might differ from one that was implemented manually in Haskell, I created two implementations of this module, one derived using `agda2hs`, and one implemented straight in Haskell. De Bruijn indexing is a tool for representing terms of lambda calculus in a way that does not require naming of variables. Instead, each variable is provided as an index to how far in lambda chain it is bound. For example, the term that normally might be expressed as

$$\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.zx) \tag{1}$$

would become

$$\lambda(\lambda 1(\lambda 1))(\lambda 2\ 1). \tag{2}$$

Because each variable is context dependent, and multiple different digits can refer to the same binding, renaming and substitution have quite subtle implementation details, while still remaining a small and self-contained test scenario. Another convenient property of this scheme that makes using it in tests very simple is that equivalent statements are structurally the same, making it very easy to check for equality. This would not be the case with regular lambda calculus, where changing variable names preserves the structure, but it's harder to test for it.

The simple test only verifies identity properties: if you rename variables in a statement using an identity function, you get the same statement back, and if you substitute variables to variables with the same indices, the statement is again unchanged. While a trivial property, it does already test the implementation to some extent. Then, further properties test more properties, for example that two renamings in a row are equivalent to one combined renaming.

### 5.3 Results

The metric I test for is the percentage of killed mutants. Ideally, this value should approach 100%, which would mean that all changes made by the framework were meaningful to the program execution and were caught by the test suite. The results are presented in Table 1. Alongside the percentages, the amount of mutants created is also shown for reference.

| Test case | Weak tests killed % | Strong tests killed % |
|---|---|---|
| quicksort | 92% (12/13) | 100% (13/13) |
| insertion sort | 33% (4/12) | 75% (8/12) |
| De Bruijn lambda calc. renaming (Haskell) | 57% (15/26) | 65% (17/26) |
| De Bruijn lambda calc. substitution (Haskell) | 28% (14/50) | 64% (32/50) |
| De Bruijn lambda calc. renaming (Agda) | 45% (5/11) | 45% (5/11) |
| De Bruijn lambda calc. substitution (Agda) | 27% (6/22) | 45% (10/22) |

Table 1: Killed percentage per test case

### 5.4 Results analysis

In all but one of the presented cases, improving the strength of the test suite increases the amount of killed mutants. Since mutations that are equivalent to the original code cannot ever be killed, this means that for those cases there were successful mutations that changed the function of the code and yet stayed alive during those runs. From my analysis, all mutants that were still alive in strong test suites did not change the functioning of the functions at all, hence could not be killed regardless of the strength of the suite.

Two things are discouraging in the context of using this mutation tool for the main goal of this paper.

**No target for strong suite**

It can be observed from the table that the amount of killed mutants for the strong test cases vary drastically all the way from $45\%$ up to $100\%$. This means, that without the use of a human oracle to verify which of the mutants are equivalent, the killed mutants rate is not very useful in a vacuum. This means that for this current implementation, one can never be sure without further investigation what the results actually mean, or if the test suite is weak or strong in any given scenario.

**Agda code style varies from Haskell**

Implementing a functionality in Agda, one has to make sure that the termination checker can prove that the function is total. One example of the resulting difference in the comparison implementations of the De Bruijn indexing was with how the natural numbers are represented. A comparison of this having an impact on the implementation can be found in Figure 5 and Figure 6. To ease up the implementation on the Agda side, instead of `Int` type, I used Peano numerals with custom implementation. While in Agda there exists a ready implementation of Natural numbers with this encoding, `agda2hs` does not allow pattern matching on `zero` and `suc` constructors, as Haskell's `Natural` type is defined as just regular bounded integer, thus not having such constructors on Haskell side. While they still could have likely been used without using pattern matching on constructors aided by an erased proof, I opted to go this route for ease of implementation.

This resulted in an unfortunate outcome where the current implementation was unable to produce any mutant that would not be equivalent to the original function or fail at runtime. One mutation that the framework was able to carry out in the Haskell implementation was to drop the increment or decrement, by changing one to zero. This is at the very least more difficult for the framework to carry out in the second case. It could for example drop the `TmSuc`, effectively removing the increment operation, but it would need to have enough context to know that it would preserve the type, or a custom rewrite rule could be written from `TmSuc` to `id`. For the constructor unwrapping pattern match however, it would be harder to specify a case that would allow the match to be dropped.

```
liftRen :: (Int -> Int) -> Int -> Int
liftRen _ 0 = 0
liftRen r n = r (n - 1) + 1

ren :: Tm -> (Int -> Int) -> Tm
ren (Var n)   r = Var (r n)
ren (App t u) r = App (ren t r) (ren u r)
ren (Lam t)   r = Lam (ren t (liftRen r))
```

Figure 5: Snippet of code written directly in Haskell

```
data TmNat = TmZero
           | TmSuc TmNat
             deriving (Eq, Show)

liftRen :: (TmNat -> TmNat) -> TmNat -> TmNat
liftRen _ TmZero = TmZero
liftRen r (TmSuc n) = TmSuc (r n)

ren :: Tm -> (TmNat -> TmNat) -> Tm
ren (Var n) r = Var (r n)
ren (App t u) r = App (ren t r) (ren u r)
ren (Lam t) r = Lam (ren t (liftRen r))
```

Figure 6: Snippet of code originally written in Agda, compiled using `agda2hs`

## 6 Responsible Research

This project did not use any public datasets or databases, and there were no outside people involved in collecting data, filling questionnaires and such. This does not however mean that some important ethical consideration cannot be made.

### 6.1 Reproducibility

The first step in ensuring reproducibility is making sure the code is accessible for anyone to audit and modify. The original code for the MuCheck repository, initially hosted on Bitbucket, and later on GitHub is freely available to the public under a permissive GPLv2 License. Because of this, the fork created for the purposes of this project is also freely

distributed under the same license, and available publicly on GitHub.

Apart from this, it needs to be possible to compile the code locally, and run the tool to be able to repeat and verify the experiments. The project is structured as a Cabal project, and can be compiled using the standard Haskell's GHC compiler using still supported version 9.8.2. This means that anybody with internet access and a working Haskell environment is able to experiment with the tool and audit the source code.

### 6.2 User reliance on mutation results

While as described, the tool in current state is not useful for end users, the hope is that at some point it will be. An important point to keep in mind then, is not to be overly reliant on this to verify the strength of the test suite over a formal proof on Agda side. Even if a QuickCheck suite is created, and verified as strong by a mutation testing framework, one still cannot be fully sure about the correctness of the code. This needs to be conveyed clearly to the user.

### 6.3 Use of AI tools

While some AI tools were used to get initial grasp of the MuCheck codebase, and explore it deeper, all the code described in the Implementation section of this paper was written fully by myself. Some of the test cases were created with help of AI tools. The content of this paper was written without the use of AI tools.

## 7 Conclusions

The object of the research was to explore the usefulness of utilizing mutation testing in checking the robustness of QuickCheck properties in Haskell, especially generated from Agda code using `agda2hs` tool.

The research sub-questions split my research in two parts. The first three regarded a more technical, implementation side of the research, while the last one related to qualitative answer to the main topic of the research.

For the first part, I used MuCheck, a proof-of-concept implementation of mutation testing in Haskell as a starting point. I was able to implement all the features I planned to in the duration of the ten week project, and put in place a pipeline that allows the user to specify tests, proven functions, and leveraged the MuCheck's existing functionality to perform the actual mutations.

However, the current implementation would not be very useful in its current form. First problem, is that the amount of mutants, that are equivalent to the original code, and hence can never be killed varies greatly between different functions. This means that any given single killed mutants ratio is not very informative to the user, without manually inspecting surviving mutants for ones that actually changed the functionality. This approach however would be both impractical, and potentially error prone.

### 7.1 Future Work

While I myself was unable to provide a useful framework for verification of tests of the code coming from `agda2hs`, there are some potential improvements that could make the tool more feasible.

First, for some cases it is possible to catch and eliminate some mutants that are equivalent to the original code. The simplest example would be to prevent mutations switching order of pattern matches or dropping some cases. Because of the way Agda code is often written, using interactive case splitting, these cases can prove not to be very useful. So while these mutations have merit in context of Haskell itself, here removing them might bring the baseline killed mutants for strong test suites closer to 100%.

Another improvement to be implemented could be more context-aware mutations. For example, if the framework could infer that a function maps a single argument to the same type, automatic detection and mutation into an identity function could create some interesting cases.

## References


[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science An EATCS Series. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[2] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674, pages 73–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[3] Niklas Broberg and Dan Burton. haskell-src-exts: Manipulating haskell source: abstract syntax, lexer, parser, and pretty-printer. https://hackage.haskell.org/package/haskell-src-exts. Accessed: 2026-06-10.

[4] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.

[5] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pages 108–122, Ljubljana Slovenia, September 2022. ACM.

[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[7] Samuel Gélineau. hint: A haskell interpreter built on top of the ghc api. https://hackage.haskell.org/package/hint. Accessed: 2026-06-10.

[8] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011.

[9] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: an extensible tool for mutation testing of haskell programs. In *Proceedings of the*

*2014 International Symposium on Software Testing and Analysis*, pages 429–432, San Jose CA USA, July 2014. ACM.

[10] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, January 2014.