# Mutation Testing: Determining The Robustness of QuickCheck Tests Generated with agda2hs

## 1. Background

**Mutation testing** is a way of checking a strength of a test suite by modifying the code tested in an automated manner, and checking that the tests report a failure.

**Agda** is a dependently typed language that allows to write formal proofs of correctness about properties of functions.

**agda2hs** [1] is a compiler that allows to convert Agda modules into clean and readable Haskell modules that can be used in that ecosystem.

**Property testing** is a testing strategy where some required property about a function is postulated, and then a framework tries to generate tests to disprove it in an automated manner. The original and most prominent library for this in Haskell is **QuickCheck**.

**Core idea**: proving properties in Agda can be hard. Try to bring Agda module into Haskell and create traditional tests first, to catch early some of the incorrect implementations, only prove afterwards.

## 2. Research Questions

How can mutation testing be applied to determine the robustness of QuickCheck tests derived from Agda type signatures?

- 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?

## 3. Implementation

**MuCheck** [2] is a proof-of-concept Haskell implementation of mutation testing, based on **haskell-src-exts** and **hint** libraries. I used it as a base to implement the following additional features:

- **QuickCheck** properties adapted into the framework
- Per-test tested function annotation for improving where the framework performs tests
- Marking functions that had all their necessary properties proven on Agda side as “Proven”.

Example annotated test

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

Example of a function marked as “Proven”

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

### Function mutation example

Example input function

```
qsort :: [Int] → [Int]
qsort [] = []
qsort (x : xs) = qsort l ++ [x] ++ qsort r
    where l = filter (< x) xs
          r = filter (≥ x) xs
```

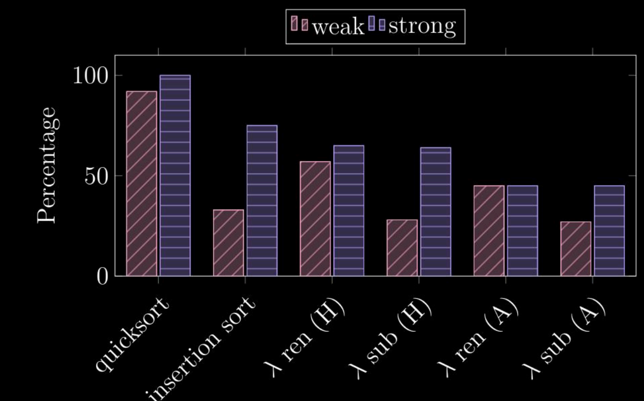
Example output mutants

```
qsort :: [Int] → [Int]
qsort [] = []
qsort (x : xs) = qsort l ++ [x] ++ qsort r
  where l = filter (< x) xs
        r = filter (== x) xs

qsort :: [Int] → [Int]
qsort [] = []
qsort (x : xs) = qsort l ++ [x] ++ qsort r
  where l = filter (< x) xs
        r = filter (> x) xs

qsort :: [Int] → [Int]
qsort (x : xs) = qsort l ++ [x] ++ qsort r
  where l = filter (< x) xs
        r = filter (≥ x) xs
```

## 4. Results and Conclusion

- There is a strong base for further development of the method in Haskell for this purpose
- Stronger test suites result in more killed mutants in almost every test case
- A lot of the mutations don’t change the functions outputs, thus skewing the results in unpredictable ways, meaning any given score is not very useful
- Haskell code created from idiomatic Agda results in fewer useful mutations with the current “generic” implementation
- A better implementation catching some of equivalent mutations and working better on code from Agda is needed if this technique is to be useful

### Raw test results comparison




While for the most cases killed mutants improvement is visible for stronger tests, the scores between modules investigated vary substantially. “ren”- renaming, “sub”- substitution. “H”- code written in Haskell, “A”- code generated with agda2hs.

## 5. References


[1] 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.

[2] 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.



Mateusz Pietrzak (m.j.pietrzak@student.tudelft.nl)

TU Delft - Research Project (CSE3000)

Responsible Professor: Jesper Cockx
Supervisor: Nathaniel Burke