MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce
Oregon State University
Corvallis, OR USA
{ledu,alipour,gopinath,alex}@eecs.oregonstate.edu

ABSTRACT

This paper presents MuCheck, a mutation testing tool for Haskell programs. MuCheck is a counterpart to the widely used QuickCheck random testing tool for functional programs, and can be used to evaluate the efficacy of QuickCheck property definitions. The tool implements mutation operators that are specifically designed for functional programs, and makes use of the type system of Haskell to achieve a more relevant set of mutants than otherwise possible. Mutation coverage is particularly valuable for functional programs due to highly compact code, referential transparency, and clean semantics; these make augmenting a test suite or specification based on surviving mutants a practical method for improved testing.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Applicative (functional) languages; D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Reliability, Languages

Keywords

Mutatation Testing, Functional Programming Languages, Mutation Operators, Haskell

1. INTRODUCTION

In mutation testing [1, 5, 12, 13], the source code of software under test is modified in small ways (mutated) multiple times, producing a set of programs that (usually) behave differently than the original program. A test suite is then applied to the mutants, and the suite is said to *kill* a mutant when some test (that passes for the original program) fails for the mutant. That counting killed mutants may provide an effective measure of test suite effectiveness in detecting real faults is widely known [2], and widely assumed in software testing research, especially when evaluating novel testing techniques [11] and even other coverage criteria [7].

Recent mutation testing research has focused mostly on imperative programming constructs. At the same time, some functional or semi-functional programming languages have attracted many developers. E.g., Twitter is largely written in Scala. Python, though less cleanly functional than Scala, is of course both very widely used and often used in a semi-functional manner. Moreover, some functional programming idioms, such as lambdas, have appeared in more traditionally imperative programming languages, such as Java and C++. Thus, there is a need for research in application of mutation testing to functional programming languages, or application of mutation testing in the presence of functional programming idioms.

As a starting point, we have developed the MuCheck[1] framework for mutation of Haskell program to allow researchers to run empirical or experimental studies on mutation testing of functional programming languages (in this case Haskell). We targeted Haskell because it has a relatively mature set of test frameworks that are popular in its community, allowing smooth integration of our mutation tool with the existing test frameworks. Haskell also serves as a kind of "laboratory" for functional programming research and development.

MuCheck works in conjunction with QuickCheck [4] and the HUnit test framework. QuickCheck is a declarative testing framework for Haskell. QuickCheck provides a domain specific language for programmers to declare the properties a function (or a program) should satisfy and to set parameters for random test generation. QuickCheck generates random tests to check the properties. MuCheck analyzes QuickCheck properties and uses mutation testing to determine the effectiveness of properties (and the tests QuickCheck generates based on those properties). It also supports the HUnit framework for unit testing Haskell programs.

Mutation testing for functional programming languages differs from mutation testing in imperative languages. Some features, such as higher order functions, introduce new potential sources of bugs which can translate into new mutation operators. Such features are either not common or they are irrelevant in imperative programs. Conversely, some mutation operators, such as the statement deletion operator, are irrelevant for strongly-typed functional programs. A new set of mutation operators is required for functional programs. MuCheck introduces a new set of mutation op-

[1]MuCheck is open source and can be accessed at `https://bitbucket.org/osu-testing/mucheck.git`.

```
type Rational = (Integer, Integer)                    1
equal:: Rational -> Rational -> Bool                  2
equal (_,0) (_,0) = True                              3
equal (_,0) _ = False                                 4
equal  _  (_,0) = False                               5
equal  (n1,d1) (n2,d2) =  n1*d2 == n2*d1              6
```

Figure 1: An example for pattern matching in Haskell. Function `equal` checks the equality of two rational numbers.

erators that target functional programs. It also provides a simple domain specific language (DSL) that can be used to define new mutation operators.

In this paper, we first introduce a set of mutation operators specific to functional programming languages (Section 2). We use examples in Haskell to explain each of the operators; however these operators can be adopted for other functional languages, including the ML family and Scala. Section 3, describes the architecture of MuCheck. Section 4 follows with an illustration of application of MuCheck on a small program. Section 5 discusses some of the key research questions for mutation testing of functional programs.

2. MUTATION OPERATORS

In this section we discuss the selection of mutation operators for functional programming languages. Proper selection of operators is key to successful mutation testing, given its underlying rationale of detecting the ability of a test suite to find "nearby" bugs.

As discussed earlier, functional programming languages introduce the possibility of new types of mutation operators. In this rest of this section, we propose mutation for basic constructs of functional programs — we consider these operators suitable for functional programs, but not sufficient. We use Haskell notation to illustrate operators. We also discuss the possible semantic effects of each mutation operator.

2.1 Re-ordering for Pattern Matching

Pattern matching is a common idiom in functional programs. It is a form of conditional statement that matches a variable with respect to its structure to different patterns. Each pattern is associated with rules, such that if the pattern matches, the rules are executed. The *ordering* of rules in patterns is often critical to the behavior of the program. That is, the program can behave differently given different ordering of the patterns.

Figure 1 shows an example of pattern matching in Haskell. This program defines rational numbers, `Rational`, as tuples of *(numerator,denominator)* (Line 1). Function `equal` defines a function to check equality between two rational numbers (Line 2). It first checks if the denominators are zero (Line 3). If both denominators are zero, both rational numbers are equal (note that a fraction with zero in the denominator evaluates to ∞). Then, in Lines 4 and 5, if the denominator of one of the numbers is zero, the numbers are not equal. Otherwise, Line 6 multiplies numerators and denominators to check equality. Note that symbol `_` is a wildcard that matches all patterns. Suppose we reorder the pattern matching by moving the pattern on line 6 to an earlier position, say Line 2. This reordering would change the semantics of `equal` and introduces a bug, because now `equal` returns true on $\frac{0}{0}$ equal to $\frac{1}{2}$.

```
take  0   _      =  []
take  _   []     =  []
take  n   (x:xs) =  x : take (n-1) xs
```

(a)

```
take' _   []     =  []
take' 0   _      =  []
take' n   (x:xs) =  x : take'(n-1) xs
```

(b)

Figure 2: An example of divergence induced by mutation of pattern matching

Re-ordering of pattern matching statements can also exhibit subtle behaviors of interest such as divergence. Consider the functions `take` and `take'` in Figure 2; both return the first n elements of a list. `take` and `take'` are similar except in the order of their patterns. Given a computation $\perp$ that does not terminate (e.g., a generator of an infinite list), `take 0` $\perp$ evaluates to `[]` while `take' 0` $\perp$ evaluates to $\perp$, i.e. it does not terminate.

In general, in pattern matching, if patterns of two or more rules are not mutually exclusive, any change in ordering of those rules potentially makes a semantic difference, and is likely to be an error in the program.

2.2 Mutation of Lists and List Expressions

Lists are the most commonly used data structures in functional programs. Many core functional idioms such as `map`, `filter`, and `reduce/fold` operate on lists. Thus, mutants based on lists are good candidates for mutations testing. We speculate that the following mutants represent a majority of list-related bugs.

- Replacing a list with the identity element (empty list).
- replacing `head:tail` with `tail` or `[head]`
- replacing `l1 ++ l2`, where `l1` and `l2` are lists and `++` denotes concatenation of two lists, with: `l2 ++ l1`, `l1`, or `l2`.

2.3 Type-aware Function Replacement

In functional programs, functions are first class citizens. That is, they can be passed to other functions as parameters and they can be returned by a function. In strongly typed functional languages, the type of a function is available at compile time. In traditional mutation of imperative programs, the mutation operators are fairly restricted, and well known, with "function" replacement usually limited to simple operators, as in the rules stated above.

Given that strongly typed functional languages offer a much richer type system, it is tempting to consider replacing any functions with all type-equivalent functions. However, this seems in practice to have two problems: (1) it may introduce a mutation explosion and (2) many of these mutants do not appear to be likely to correspond to likely actual faults. Therefore, it is more practical to allow users to add rules for any cases where function replacement is a useful mutation. There may be some cases that should be included as standard mutations: for instance, the effect of replacing a function of type `a -> a` with the identity function[2] corre-

[2]The identity function does not do any computation and returns the input as the output.

sponds to the "statement deletion" mutation that eliminates computation in imperative mutation.

3. MUCHECK ARCHITECTURE

In this section, we describe the major building blocks of MuCheck. These modules are: *Mutation DSL*, *Mutation Generation* and *Mutation Execution*.

3.1 Mutation DSL

MuCheck provides a DSL to implement definition of the mutation operators in Section 2. Users also can use this DSL to define new mutation operators or alter the existing ones.

In this DSL, a mutation operator is a function that replaces a given value with its mutant replacement. MuCheck provides three polymorphic operators `==>`, `==>*`, and `*==>*` as a *domain specific language* for defining mutation operators. `a ==> b` states that `a` should be replaced by `b`. `a ==>* [b1, b2]` states that `a` should be replaced by either `b1` or `b2`. `[a1, a2] *==>* [b1, b2]` indicates that either `a1` or `a2` should be replaced by either `b1` or `b2`.

3.2 Mutation Generation

The mutation generation module can be further divided into two smaller modules: *mutation configuration*, and *abstract syntax manipulation*.

3.2.1 Mutation Configuration

Users can configure different parameters of mutation testing: (1) choosing between first-order mutation and higher order mutation, (2) maximum number of mutants to generate, (3) mutation operators to apply, and (4) HUnit test cases and/or QuickCheck specifications.

3.2.2 Abstract Syntax Manipulation

MuCheck uses the *Scrap Your Boilerplate* library [14] to manipulate abstract syntax trees, which are produced by the *haskell-src-exts* library parser [3]. The mutation algorithm traverses the nodes in the syntax tree and non-deterministically applies each mutation operator. The algorithm outputs syntactically unique mutants.

3.3 Mutation Execution

MuCheck supports HUnit and QuickCheck properties. HUnit is a unit testing framework. QuickCheck takes an algebraic specification and checks it over a set of randomly generated test cases [4].

MuCheck uses the Haskell interpreter Hint package [10] to dynamically load mutants and run tests. After execution of mutants, MuCheck outputs a short test summary on the terminal and a detailed log file containing test results and giving a pointer to the location of all surviving mutants for examination by the user.

4. EXAMPLE: QUICK SORT

In this section, we show how to use MuCheck to generate mutants for a simple program. We use `qsort` (Figure 3) as the running example. `qsort` implements the quick sort algorithm. The following QuickCheck properties are defined to test the `qsort` function.

```
idempProp xs = qsort xs == qsort (qsort xs)
sortedProp xs = isSorted (qsort xs)
```

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

Figure 3: An Implementation of Quick Sort in Haskell

```
Total number of mutants: 13

quickCheckResult idempProp
Successes (not killed): 2 (15%)
Failures (killed): 11 (84%)
...
quickCheckResult sortedProp
Successes (not killed): 5 (38%)
Failures (killed): 8 (61%)
```

Figure 4: Snippet of results generated by QuickCheck

`idempProp` property states the idempotency of sorting operations. `sortedProp` property states that `qsort` must output a sorted list (we omit the implementation of `isSorted` for brevity).

Configuration: The following shows the configuration of mutation testing for this experiment.

```
stdArgs = StdArgs {muOps = allOps
                  , doMutatePatternMatches = True
                  , doMutateValues = True
                  , doNegateIfElse = True
                  , doNegateGuards = True
                  , maxNumMutants = 30
                  , genMode = FirstOrderOnly }
```

`muOps` is set to use a set of pre-defined mutation operators, but these can be replaced by user-defined operators. `doMutatePatternMatches` specifies whether MuCheck will permute pattern-matching cases. `doMutateValues` enables mutating integer values to four constants: `(+1)`, `(-1)`, `0`, and `1`. `doNegateIfElse` negates the Boolean formula of `if-then-else` statements, while `doNegateGuards` provides the same functionality for guards. `maxNumMutants` limits the maximum number of mutants to be generated. `genMode` is either `FirstOrderOnly` or `FirstAndHigherOrder`. `FirstOrderOnly` limits the application of mutation operators to one operator per mutant. `FirstAndHigherOrder` will apply operators once, and then re-apply operators on generated mutants when possible.

Result of mutation: Given above configuration, MuCheck produces the results in in Figure 4, showing that 13 mutants have been generated and tested. `idempProp` kills 11 out of 13 mutants while `sortedProp` only detects 8 out of 13 mutants.

5. DISCUSSION

Functional languages such as Haskell and OCaml have been traditionally limited to specific (if important) sectors such as the financial and security industries. Recently, functional programming languages have gained broader popularity. Use of Scala in the backbone of Twitter and Erlang in WhatsApp signifies the growing importance of functional approaches. Properties such as code compactness and referential transparency have convinced designers of imperative languages such as C++ and Java to add functional idioms into their languages.

The increasing use of functional programming languages and idioms requires adapting and devising testing techniques for them. In this section, we summarize questions that need to be addressed in mutation of functional languages, and discuss how MuCheck can be utilized to answer them.

5.1 Competent Programmer Hypothesis

According to Demillo et al. [6], programmers write software that is close to being correct. The competent programmer hypothesis, along with coupling (the idea that if a test kills a complex change, it tends to kill the changes that compose it), forms the foundation of mutation analysis. The competent programmer hypothesis is also the basis for various mutation operators proposed — operators are often designed to model the common mistakes a competent programmer makes, so that the mutations introduced are similar to real faults.

While this seems intuitively true, there exists very little evidence to support this assertion. Secondly, our recent study [9] found that the kind of mistakes made in programs are very much dependent on the language of the program is produced. Different languages tend to have different distributions of syntactical fault patterns. Critically, Haskell was found to be very different from the other languages studied (C, Java, and Python) in the distribution of bug patterns.

This suggests that mutation operators from other languages may not be a good fit for languages such as Haskell. We have, in this paper adopted a strategy of rethinking the mutation operators needed for Haskell, so that our mutations are better able to emulate fault patterns in Haskell.

5.2 Mutation Operators

In this paper, we have defined and implemented a number of mutation operators. There may be other useful mutation operators for functional programs. The mutation DSL of MuCheck allows researchers to easily define such operators and evaluate their effectiveness.

5.3 Reducing the Cost of Mutation Testing

As in traditional mutation testing, mutation of functional programs can result in a large number of mutants. Selective mutation [15, 16] and mutation sampling [8, 17, 18] have shown how to reduce the number of mutants while preserving the power of mutation testing. Selective mutation and sampling techniques have not been evaluated for functional programs. The presence of type-aware mutation operators can potentially introduce a large number of mutants. MuCheck can be used to investigate these issues.

6. CONCLUSION

This paper presented MuCheck, a mutation testing tool for Haskell programs. We presented some mutation operators for functional programming languages in general, and have implemented them for the Haskell language. We explained the architecture and basic use of MuCheck and discussed potential applications of MuCheck in further studies in (functional) mutation testing.

7. REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, 1979.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411, 2005.

[3] N. Broberg. The haskell-src-exts package. http://hackage.haskell.org/package/haskell-src-exts.

[4] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, pages 53–64, 2011.

[5] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[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, 1978.

[7] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *ACM International Symposium on Software Testing and Analysis*. ACM, 2013.

[8] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce. An empirical comparison of mutant selection approaches. under submission.

[9] R. Gopinath, C. Jensen, and A. Groce. Mutant census: An empirical examination of the competent programmer hypothesis. In *Technical Report, School of EECS, Oregon State University*.

[10] D. Gorin. The hint package - runtime haskell interpreter. http://hackage.haskell.org/package/hint.

[11] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.

[12] R. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, SE-3(4):279–290, 1977.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[14] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, 2003.

[15] A. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Computer Software and Applications Conference, 1991. COMPSAC '91., Proceedings of the Fifteenth Annual International*, pages 604–605, 1991.

[16] A. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pages 100–107, 1993.

[17] W. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995.

[18] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2013.