Mutation Testing of Functional Programming Languages

Duc Le
Oregon State University
ledu@eecs.oregonstate.edu

Mohammad Amin Alipour
Oregon State University
alipour@eecs.oregonstate.edu

Rahul Gopinath
Oregon State University
gopinath@eecs.oregonstate.edu

Alex Groce
Oregon State University
alex@eecs.oregonstate.edu

***Abstract*—**

Mutation testing has been widely studied in imperative programming languages. The rising popularity of functional languages and the adoption of functional idioms in traditional languages (e.g. lambda expressions) requires a new set of studies for evaluating the effectiveness of mutation testing in a functional context. In this paper, we report our ongoing effort in applying mutation testing in functional programming languages. We describe new mutation operators for functional constructs and explain why functional languages might facilitate understanding of mutation testing results. We also introduce MuCheck, our mutation testing tool for Haskell programs.

[1] ***Keywords*—Mutation Analysis, Haskell**

I. Introduction

In mutation testing [1]–[5], the source code of software under test (the SUT) 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 at detecting real faults is widely known [6], and widely used in software testing research, especially to evaluate novel testing techniques [7] and even other coverage techniques [8].

The true potential of mutation testing, however, is not exploited in its use as a mere score for a test suite. One of the core advantages of mutation testing over other coverage measures is that it provides much deeper information about the inadequacies of a test suite. In particular, statement, branch, path, dataflow, and even predicate-complete test coverage [9] omissions can only explain how a test suite fails to exercise all the *behaviors* of the SUT. However, when behavior is fully exercised but the test oracle is faulty (considering some bad behaviors good), this cannot be exposed by code or state space coverage. Failing to kill a mutant, however, can often be ascribed to oracle inadequacy, one of the most important potential defects in a test suite. This is perhaps the most fundamental difference between mutation testing and other coverage approaches. Even when oracle problems are not considered, however, a failure to kill a mutant provides information qualitatively different than in most other forms of test suite evaluation. While simple coverage metrics such as statement coverage have a direct mapping to understandable omissions in a test suite, many more powerful coverages cannot easily map to omissions. That a path is never explored in testing may be highly uninteresting, in that the path has no behavioral consequences. Non-equivalent mutants, however, always map to a statement that the current test suite would miss a particular hypothetical fault in the SUT. Mutation testing, like branch or statement testing, therefore, should be a source not only of suite quality information, but of information directly useful in *improving a test suite or test oracles*. Mutation testing subsumes the direct information found in e.g., statement coverage — "You didn't run this line, so even if it reads `assert(false);` you won't find that bug" — and enriches it with deeper behavioral information and oracle failures.

In practice, however, to our knowledge surviving mutants are almost never used to improve test suites or test oracles. Mutation testing is rarely used for any purpose in real-world development, though this may be changing [10] slowly, as computing power increases. Even software researchers well acquainted with mutation testing, however, essentially use it only as a way to compare test suites. The few attempts to use mutations to improve oracle or suite quality are restricted to simple assertions in unit tests or choosing observation variables [11], [12]. We ascribe the failure to make use of mutation testing's true power to two primary issues:

- First, and most importantly, moving from a surviving mutant to an understanding of a test suite's inadequacies is difficult, when the reason is more complex than that the mutated code is not covered. Even expert developers and seasoned software engineering researchers do not find this an easy task.
- Second, the number of equivalent mutants or mutants that do not change behavior relevant to the testing in question (e.g., only modifying logging behavior) is sometimes high.

In this paper, we propose that these problems can be greatly mitigated by applying mutation testing to functional programming languages. Functional programs have important traits that should make mutation testing more effective for use by humans intending to improve test suites. The key for improving testing using mutation testing is the *understanding of why mutants survive a test suite.* Functional languages typically have certain features, including highly compact code, referential transparency, simplicity of data flow, and well-defined language semantics, that should make understanding the reasons a covered mutant survives testing much easier.

Figure 1 shows how we imagine mutation testing fitting

[1] Technical Report Oregon State University

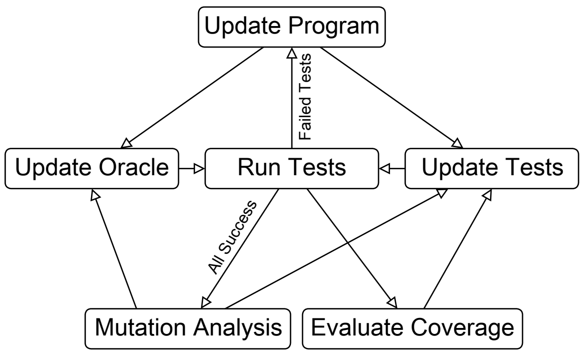

Fig. 1. A workflow for effective testing

into an effective testing process. During and after implementation of the SUT itself, a set of tests and an oracle are developed. In imperative languages these two aspects of test development are often tightly interlaced, with the oracle being a set of assertions in unit tests. In functional programming, it has become common to apply automated tools such as QuickCheck [13], which generates random inputs to the SUT, and write explicit equational specifications for each tested function. In all cases, once a test suite is defined (either explicitly or via a generative method such as random testing), tests are executed. Failing tests lead to fault correction in the SUT (or, on occasion, in the oracle or test suite), but when all tests succeed this may indicate not that the SUT is correct but that the testing or oracle is insufficient. Code coverage provides information about SUT behavior not explored by the tests, leading to test improvements. Understanding of surviving mutants can lead to improvements in either the test generation process or the test oracle, as the reason a mutant is not killed indicates. Functional programming already makes part of this workflow cycle easier for developers to carry out, since tools like QuickCheck make test and oracle implementation and debugging easier, and many modern functional languages (Haskell, F#, OCaml) provide code coverage tools. We propose that mutation testing fits seamlessly into this workflow, and can integrate well with the automation already widely used by functional programmers.

The primary contributions of this paper are threefold.

- We propose that mutation testing can become a considerably more powerful tool for improving testing by applying it in the context of functional programming languages, which we argue are in many ways ideally suited for the testing workflow enabled by mutation testing (Section II).
- We provide the first (to our knowledge) discussion of the application of mutation testing to functional languages, which requires different operators and assumptions than in imperative languages (Section III).
- We demonstrate our claims by applying our mutation tool, MuCheck, to a case study (Section IV and V).

II. Advantages of Functional Programming for Understanding Mutation Survival

In this section, we elaborate our argument that functional programming languages should make it easier to understand why a mutation survives testing than traditional imperative languages. To clarify the difficulties of mutant survival understanding in imperative code, we examined a random sample of 45 mutants covered but not detected by 5,000 swarm tests [7] of the YAFFS2 flash file system [14]. Mutations were generated by an approach (and software) shown to provide a good proxy for fault detection by Andrews et.al. [6]. Our original intention was to identify the cause of each survival, but this proved to be even more onerous than we had expected. While our view that understanding the survival of these mutants was too difficult is an opinion, not a validated empirical result, our experience with file system development and random testing [15], and familiarity with (testing) the YAFFS2 source code specifically gives this opinion some weight.

Compactness. Functional programs are generally thought to be much more compact than traditional imperative programs with the same semantic content. For example, the following Java 8 code (taken from [16]) utilizes the map idiom. Implementing it imperatively can require up tolines of code.

```
myCollection.parallelStream().map(e-> e.length)
```

Mutation testing relies on the assumption that programs are often "almost correct" in a syntactic sense and thus small syntactic changes should produce realistic bugs that predict a suite's ability to detect real faults. In functional languages, the space of nearby syntactically valid programs is typically smaller, due to a greater expressive power for each syntactic unit. This should lead to a smaller, but more meaningful, set of mutants to examine. Most surviving mutants, if not equivalent, should indicate serious defects in testing. Moreover, the greater semantic "weight" of syntactic modification in smaller programs ought to result in fewer equivalent mutants.

Referential Transparency. Referential transparency means that an expression can be replaced with its value without changing its behavior — informally, it means that for the same inputs, a function will always return the same value. Pure code (with no side effects) in any language is referentially transparent, but code in a pure functional language is always referentially transparent. Referential transparency has several advantages, but in the context of testing perhaps its more important feature is that a referentially transparent function can be effectively tested without knowing the context it resides in, so long as a specification of the relationship between inputs and outputs can be established. This naturally compositional verifiability encourages the use of tests and specifications at the individual function level, which often means that when a mutant survives a test, it is easy to establish that tests for the mutated function itself should have detected the problem. In imperative programs, code much more often depends on a complex context (initialization of global data structures, complex pointer-based data structures etc.) or is called partly for side effects whose validity is difficult to check in isolation, so complete specification and effective testing for individual functions is much less frequent.

Simplicity of Data Flow. Attempting to understand why mutants survive testing in an imperative program often involves an effort to chase the flow of a modified data value. Of the 45 surviving YAFFS2 mutants, 18 consist of a modification or deletion of an assignment to a variable. In some cases, these values are stack-local, and in other cases they are global values. In either case, understanding the mutant's survival requires determining where the value assigned (or not assigned in the case of statement deletion) is next used. This is difficult in the case of global variables used throughout the code, and extremely painful in the case of values assigned through pointer dereferences, which may in general require understanding the aliasing structure of the program. For example, one mutant changes assignment to a field in a structure passed as a pointer to the mutated function. The function is called in 5 places in the code, in some cases passing another argument taken as a pointer. Simply following the data flow in cases like this is very hard, even though YAFFS2 is not a particularly alias-intensive program by C standards. In functional code, in contrast, data flows via one mechanism, function call evaluation. Finding where a mutated value affects program semantics is a simple matter of following the callees of the code containing the mutated value. Equally importantly, in a purely functional language, there is no equivalent of aliasing. Even in impure functional languages (e.g. ML), aliasing is far less frequently used than in languages such as C, C++, and Java. Arguably, this is one area where object orientation increases the difficulty of understanding, as many OO styles encourage not only heavy use of state mutation and references that cannot easily be understood in a static reading of code, but also make use of virtual functions, which can further complicate matters. Of course, code in functional languages often calls a function taken as a parameter, but in the absence of mutation the effect of such calls on data flow is much easier to understand.

Clean Semantics. Another perplexing kind of survival is the case where the mutant's survival seems, on the face of it, simply impossible. One mutant of the YAFFS2 code removes the return statement from a function called in every single test execution, which reading the code shows "should" result in an invalid initialization of the emulated flash device. Removing the return statement from a non-void function in C, however, results in an undefined program semantics. As it happens, the compiler we are using produces code such that this clearly "bad" mutant is in fact equivalent to the original program. The semantic equivalence is presumably not stable across architectures, but examining such "accidentally equivalent" mutants that would be detected easily by the suite as soon as they become non-equivalent can require considerable effort. Before realizing that the compiler was responsible, we examined the code to see if the (we assumed) garbage return value was not being actually used during initialization, etc. This effort would have been considerably larger in non-initialization code, where determining that garbage values would lead to a definite crash would have been much more difficult because of the difficulty of following the flow of the "wrong" data. In almost all functional languages, programs with undefined semantics are caught at compile time. Even functional languages such as Scheme that lack strong static typing generally guarantee that improper operations will cause a well-defined error behavior, rather than arbitrary results. While well-defined semantics are not unique to functional languages, two very popular imperative languages (C and C++) make remaining within the well-defined behavior of the language a challenge even for normal code, much less mutated code. One mitigation in C and C++ is to reject mutants where the compiler emits serious warnings. This has two problems, however: first, it is insufficient as many undefined behaviors are not detected by compilers; second, some mutants rejected by the compiler might reveal defects in the test suite.

```
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
```

Fig. 2. An example for pattern matching in Haskell. Function `equal` checks the quality of two rational numbers.

III. Mutation Operators for Functional Programs

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. Andrews et al. propose four types of operators for C program mutation testing [6]:

- Replacing integer constant N with one of {0, 1, -1, N+1, N-1},
- replacing an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another of the same class,
- negating the conditional in `if` or `while` statements, or,
- deleting a statement.

In 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.

A. Reordering 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, 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 on different orders of pattern matching. The following example illustrates this.

Example. Figure 2 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 the equality between two rational numbers (Line 2).It first checks if the denominators are zero (Line 3). If both denominators and 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

```
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)

Fig. 3. An example of divergence induced by mutation of pattern matching

the semantics of `equal` and introduces a bug, because now `equal` returns true on $\frac{0}{0}$ and $\frac{1}{2}$.

Re-ordering of pattern matching statements can also exhibit subtle behaviors of interest like divergence. Consider the functions `take` and `take'` in Figure 3 which 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 semantically different behavior, and is likely to be an error in the program.

B. Mutation of lists and list expressions

Lists are the most common data structures in functional programming languages. Most functional idioms like *map* and *filter* 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 list identity element, i.e. empty list.
- Removing a part in list expressions, e.g.
  - replacing `head:tail` with `tail` or `[head]`
  - replacing `list1 ++ list2` with `list2 ++ list1`, `list1`, or `list2`, where `list1` and `list2` are lists.

C. Type-aware Function Replacement

In functional programs, functions are first class citizens. That is, they have types, they can be passed to other higher-order functions, 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 real likely errors. 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] is similar to "statement deletion" mutation that eliminates a computation.

IV. The MuCheck Tool: A Simple DSL for Mutation Testing of Haskell Programs

The workflow shown in Figure 1 is already well-supported in some functional programming languages, with the exception of the use of mutation testing. The QuickCheck tool [13], originally developed for Haskell, but since implemented in many languages, uses automated random testing (generating what QuickCheck calls arbitrary values of a given type) to test a program. QuickCheck lets programmers write specifications of program behavior as functions that take test inputs as values, and then generates random values to try to falsify the specification, reporting a counterexample when the property does not hold. QuickCheck is popular because it is simple, powerful, and highly configurable. In essence, QuickCheck provides an expressive Domain Specific Language (DSL) for writing correctness properties and customizing random input generators.

MuCheck's basic design is somewhat inspired by QuickCheck: rather than aiming at a universally capable tool, MuCheck aims to be easily extended and modified by experienced Haskell programmers, and assumes the use of QuickCheck for test case generation and evaluation.

A. Customizing MuCheck

MuCheck supports a set of customizable standard arguments.

```
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, which has four possibilities: `(+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. There are two possible values of `genMode`, 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 those operators on generated mutants when possible.

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

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

Fig. 4. Implementation of QuickSort in Haskell

B. *Mutation Operator Implementation*

A mutation operator is a function that replaces the original value with its mutated replacement. For instance, applying the operator `Ident "pred" ==> Ident "succ"` will replace instances of `pred` by `succ` one at a time. The use of the constructor `Ident` ensures that only identifiers are affected by the operator. The statement

```
Symbol "+" ==>* [Symbol "-", Symbol "*"]
```

creates two operators, one replacing + with − and one replacing + with ⋆. The statement

```
[Symbol ">", Symbol "<"] *==>*
       [Symbol "<=", Symbol ">="]
```

creates four operators, replacing each of >, < with each of <=, >=. The three functions ==>, ==>⋆, and ⋆==>⋆ constitute a simple DSL available for MuCheck's users.

Mutation operators simplify tree traversal, yet they can only deal with cases where the values to mutate are constants. Changes such as adding 1 to an integer, permuting pattern-matching cases, etc. cannot be represented using this form of mutation operator. MuCheck deals with this problem by inspecting the source code once to retrieve all constant values such as `Int 7`. MuCheck then generates operators based on the retrieved constants. For `Int 7`, the operators are `Int 7 ==>* [Int 0, Int 1, Int 6, Int 8]`.

V. A Simple Case Study: Quick Sort

Figure 4 is an implementation of the quick sort algorithm in Haskell. For testing it with QuickCheck, the programmer should write specifications in the form of Boolean functions that, given a test input, determine if the tested code passed the test. For instance, one simple property of any sort is that it should be idempotent: sorting a sorted list leaves the list unchanged:

```
idempProp xs = xs == qsort (qsort xs)
```

QuickCheck instantly informs us that this specification does not hold, and gives a counterexample, automatically reduced in size:

```
Prelude Test.QuickCheck> quickCheck idempProp
*** Failed! Falsifiable (after 6 tests and 6 shrinks):
[1,0]
```

Looking closely, we realize we have incorrectly defined the idempotency property, forgetting to apply `qsort` to `xs` one on side of the equality:

```
idempProp xs = qsort xs == qsort (qsort xs)
```

Now QuickCheck reports no failed tests:

```
Prelude Test.QuickCheck> quickCheck idempProp
+++ OK, passed 100 tests.
```

Even without the aid of mutation testing, we can see that idempotency is not a very complete specification for a sorting function. We therefore also verify the ordering of elements in the list:

```
isSorted [] = True
isSorted (x:[]) = True
isSorted (x:y:xs) | x <= y = isSorted (y:xs)
                  | otherwise = False
sortedProp xs = isSorted (qsort xs)
```

Again, QuickCheck shows that all test pass. It is tempting at this point to believe that we have a complete test and oracle for sorting. However, mutation testing can reveal that many mutants survive these properties. If we apply our MuCheck tool, it reports on the weakness of this specification:

```
Total number of mutants: 11
Errors: 0 (0%)
Successes (not killed): 5 (45%)
Failures (killed): 6 (54%)
```

Almost half of qsort mutants survive our "good" specification. Examining the log, we see the mutants not killed by each property. That some mutants survive the idempotence property is not surprising, but that checking sortedness is insufficient may be a surprise. Because our example code is so short and simple, there is no need to apply Haskell's coverage tool — QuickCheck is certainly covering our code completely. We therefore examine one of the mutants that survives the sortedness property:

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

The problem is that the specification does not check that a list, once sorted, has the same length as the original list. We can fix our oracle by adding one more property:

```
lenProp xs = (length xs) == (length (qsort xs))
```

Adding `lenProp` to our list of properties to check, we find that only one mutant survives testing. Examining it shows a weakness of the current implementation of MuCheck:

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

This mutant is based on re-ordering patterns, but in this case the patterns are non-overlapping so the mutant is equivalent. Fortunately, this is easy to figure out. At this point, though our specification is weak (since the precise property to establish is that the new list is a sorted permutation of the input list), it is effective for catching faults "near" the original source code of `qsort`. If we applied our specification to an algorithm where producing sorted equal-length results that changed values was a short syntactic distance from our code, mutation testing would reveal the need to further refine the specification.

Mutation testing can detect problems that are not simple oracle insufficiency or coverage inadequacy. Consider the case where we declare `qsort` to be a polymorphic sort that applies to any ordered type, and forget to force our QuickCheck to generate data of an interesting type, such as `Int`. Without this information, QuickCheck generates lists of the unit type, which is ordered but in a degenerate sense (it only has one concrete instance). In this event, even the buggy idempotence property is successful, and code coverage does not reveal the problems with test input generation. MuCheck fortunately reports that mutation survival is extremely high for even obviously bad mutants, which may lead to an examination of the test generation process (since the oracle is clearly strong enough to detect these mutants).

Finally, mutation testing can help us simplify and understand our specification, even once it kills all mutants. The power of different specifications may not be obvious. For instance, in our quick sort example, the most powerful specification is the length property, which actually kills all non-equivalent mutants! The sortedness specification is the least powerful, killing 54% of mutants. Even idempotence is more powerful, killing 81% of mutants. In a sense, erroneous behavior in the presence of duplicates is a more "likely missed" error in a sorting algorithm than simple failure to produce sorted results.

VI. Conclusions and Future Work

This paper proposes that, while using mutation testing to understand test suite failures and therefore improve testing (either by improving the test suite itself or the oracle used to check correctness) is too difficult for widespread adoption in imperative programming languages, it may be a valuable testing practice in functional programming languages. We argue that certain features more commonly found in functional than imperative languages make understanding the survival of mutants a much easier task. Mutation testing in functional languages requires a somewhat different set of operators, and we have proposed a useful initial set of such operators. As with imperative languages, considerable empirical investigation may eventually be required to establish an ideal set of operators. A few examples, ranging from trivial (quick sort) to minor (AVL trees) to realistic (XMonad) demonstrate the potential of mutation testing as a tool for humans hoping to improve test suites and oracles. MuCheck, a prototype tool for mutation testing in Haskell that integrates smoothly with QuickCheck and enables users to easily modify the set of mutation operators applied to their programs, is available at https://bitbucket.org/osu-testing/mucheck.git.

Future work can be divided into four increasingly ambitious projects. First, as noted our implementation of mutation testing for Haskell is far from complete and ideal. Further work on operator selection, rejecting equivalent mutants, etc. is required. Second, mutation testing would almost certainly be useful for languages other than Haskell. Other popular functional languages such as those in the ML family (SML/NJ and OCaml) and the LISP family (Scheme, Clojure) are obvious choices. Moreover, some of the features that make functional programs attractive targets for mutation testing have become more and more widely adopted in languages that are not simply functional languages. Python, Ruby, and other modern scripting languages provide `map`, `fold`, `filter`, and λ constructs and other functional features, and are often very concise. The latest iterations of Java and C++ incorporate some form of lambda expression. Investigating whether mutation understanding is more effective when applied to imperative programs written using a functional style is therefore a third extension. Given that even when using functional features aliasing and state mutation are likely to remain widely used, it is far from clear that the benefits expected in functional languages will be easy to arrive at in an imperative world. This suggest a fourth extension: if mutation testing proves to be a valuable addition to the standard testing workflow in functional languages, it may be worth building static and dynamic analysis tools designed specifically to help with understanding mutation survival in imperative languages. We therefore propose the development of mutation understanding tools that help provide a *causal explanation* for why a given mutant has survived, as the testing analogues to fault localization and explanation tools for faults in code.

References

[1] J. A. T. Acree, "On mutation," Ph.D. dissertation, Georgia Institute of Technology, 1980.

[2] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis," Georgia Institute of Technology, Tech. Rep., 1979.

[3] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[4] R. Hamlet, "Testing programs with the aid of a compiler," *Software Engineering, IEEE Transactions on*, vol. SE-3, no. 4, pp. 279–290, 1977.

[5] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, 2011.

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

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

[8] 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.

[9] T. Ball, "A theory of predicate-complete test coverage and generation," in *Formal Methods for Components and Objects*. Springer, 2005, pp. 1–22.

[10] H. Coles, "Pit mutation testing," http://pittest.org/.

[11] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278–292, 2012.

[12] M. Staats, G. Gay, and M. P. E. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *International Conference on Software Engineering*, 2012, pp. 870–880.

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

[14] "Yaffs: A flash file system for embedded use," http://www.yaffs.net/.

[15] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *International Conference on Software Engineering*, 2007, pp. 621–631.

[16] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *ESEC/FSE 2013*, 2013, pp. 543–553.