MUTAGEN: Working title

ANONYMOUS AUTHOR(S)

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

Additional Key Words and Phrases: random testing, mutations, heuristics

ACM Reference Format:
Anonymous Author(s). 2021. MUTAGEN: Working title. In *Proceedings of OOPSLA '21: ACM SIGPLAN conference on Systems, Programming, Languages, and Applications (OOPSLA '21).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

The contributions of this work are:

- Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio.
- Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.
- Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio.

OOPSLA '21, October 17–22 2021, Chicago, IL

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

2 BACKGROUND

In this section we briefly introduce the concept of Property-based Testing along with *QuickCheck*, the most popular tool of this sort and often considered as the baseline when comparing PBT algoritms. Moreover, we describe the ideas and limitations behind *FuzzChick*, which enhances the PBT approach with execution information and serves as the foundation for MUTAGEN. *FuzzChick* is implemented as an extension of *QuickChick*, Coq's own reimplementation of *QuickCheck*.

2.1 Property-Based Testing and *QuickCheck*

Property-based testing is a powerful technique for finding bugs without having to write test cases by hand. Originally introduced by **?** in tandem with the first version of *QuickCheck*, this technique focuses on aiming the developer's efforts into testing systems via executable specifications using randomly generated inputs. Moreover, tools like *QuickChick* and Isabelle's *QuickCheck* demonstrate that PBT can also be used in the formal verification realm. There, one can quickly spot bugs in system specifications before directing the efforts into pointlessly trying to prove bogus propositions.

In the simplest form, there are four main elements the user needs to provide in order to perform property-based testing on their systems:

- one or more *executable properties*, often implemented simply as a boolean predicates,
- *random data generators*, used to repeatedly instantiate the testing properties,
- *printers*, used to show the user the random inputs that falsify some testing property (the counterexample) whenever a bug is found, and
- *shrinkers*, to minimize counterexamples making them easier to understand by humans.

In this work we focus solely on the first two elements introduced above, namely the testing properties and the random data generators used to feed them. Printers and shrinkers, for the most part, can be obtained automatically using generic programming capabilities present in the compiler, and although being crucial for the testing process as a whole, their role becomes irrelevant when it comes to *finding* bugs.

Perhaps the simplest PBT technique is to repeatedly generate random inputs and instantiate the testing properties until they either get falsified by a counterexample, or we ran a sufficiently large amount of tests — suggesting that the properties holds. *QuickCheck* implements a testing loop that closely follows this simple idea, which is outlined in Algorithm 1, where P is the testing property, N is the maximum number of tests to perform, and gen is the random generator to be used to instantiate P.

Algorithm 1: *QuickCheck* Testing Loop

```
Function Loop($P, N, gen$):
    i ← 0
    while $i < N$ do
        x ← Sample(gen)
        if not $P(x)$ then return Bug(x)
        i ← i+1
    return Ok
```

To illustrate this technique, let us focus on the same motivating example used by Lampropoulos et al., who propose a simple property defined over binary trees. Such data structure can be easily defined in Haskell using a custom data type with two data constructors for leaves and branches respectively:

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
```

The type parameter `a` indicates that trees can be instantiated using any other type as payload, so the value `Leaf Bool` has type `Tree Bool`, whereas the value `Branch (Leaf 1) 2 (Leaf 3)` has type `Tree Int`. Then, we can define tree reflections using a simple recursive function that pattern matches against the two possible constructors, inverting the order of the subtrees whenever it encounters a branch:

```
mirror :: Tree a -> Tree a
mirror (Leaf x)       = Leaf x
mirror (Branch l x r) = Branch (mirror r) x (mirror l)
```

Later, a reasonable requirement to assert for is that `mirror` must be *involutive*, i.e., reflecting a tree twice always yields the original tree. We can simply capture this property using a boolean predicate written as a normal function:

```
prop_mirror :: Tree Int -> Bool
prop_mirror t = mirror (mirror t) == t
```

For simplicity, here we instantiate the tree payload with integers, altough this predicate should clearly hold for any other type with a properly defined notion of equality as well.

With our simple specification in place, the last missing piece is a random generator of trees. In *QuickCheck*, this is usually done via the type class mechanism, instantiating the `Arbitrary` type class, providing a random generator as the implementation of the overloaded `arbitrary` operation:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized gen
    where
      gen 0 = do { x <- arbitrary; return (Leaf x) }
      gen n = oneof [ do {x <- arbitrary; return (Leaf x) }
                    , do {l <- gen (n-1); x <- arbitrary; r <- gen (n-1); return (Branch l x r)} ]
```

Let us break this definition into parts. The first line states that we will provide an `Arbitrary` instance for trees with payload of type `a`, provided that values of type `a` can also be randomly generated. This allows us to use `arbitrary` to generate `a`'s inside the definition of our tree generator.

Moreover, *QuickCheck* internally keeps track of the *maximum generation size*, a parameter that can be tuned by the user in order to limit the size of the randomly generated values. Our definition exposes this internal parameter via *QuickCheck*'s `sized` combinator, allowing us to parameterize the maximum size of the randomly generated trees. If the generation size is zero (`gen 0`), our generator is limited to produce just leaves with randomly generated payloads. In turn, when the generation size is strictly positive (`gen n`), the generator is able to perform a random uniform choice between generating either a single leaf or a branch. When generating branches, the generator calls itself recursively in order to produce random subtrees (`gen (n-1)`). Notice the importance of reducing the generation size on each recursive call. This way we ensure that randomly generated trees using a generation size `n` are always finite and have at most `n` levels.

Finally, we are ready to let *QuickCheck* test `prop_mirror` against a large number of inputs (100 by default) produced by our brand new random tree generator:

```
quickCheck prop_mirror
+++ Ok ####
```

Should we mistakenly implement `mirror`, e.g., by dropping the right subtree altogether:

```
mirror (Branch l x r) = Branch (mirror l) x (mirror l)
```

then *QuickCheck* will quickly falsify `prop_mirror`, reporting a minimized counterexample that we can use to find the root of the issue:

```
quickCheck prop_mirror
*** Failed ####
```

At this point, it is clear that the *quality* of our random generators is paramount to the performance of the overall PBT process. Random generators that rarely produce interesting values will fail to trigger bugs in our code, potentially leaving entire parts of the codebase virtually untested.

Recalling our tree generator, the reader (far from mistaken) might already have imagined better ways for implementing it. For most practical purposes, this generator is in fact quite bad. However, it follows a simple type-directed fashion, and it is a good example of what to expect from a random generator synthesized automatically using a process that knows very little about the values to be generated apart from their (syntactic) data type structure.

As introduced earlier, there exist multiple tools that can automatically derive better random generators solely from the static information present in the codebase. Sadly, these tools lack the domain knowledge required to generate random data with complex invariants — especially those present in programming languages like well-scopedness and well-typedeness.

In particular, automatically derived generators are remarkably uneffective when used to test properties with sparse preconditions. Let us continue with the example by Lampropoulos et al. to illustrate this problem in more detail. For this, consider that we want to use our `Tree` data type to encode binary-search trees (BST). Then, given a predicate `isBST` that asserts if a tree satisfies the BST invariants, we might want to use it as pre- and post-condition to assert that BST operations like `insert` preserve them:

```
prop_bst_insert :: Tree a -> a -> Bool
prop_bst_insert t a =
  isBST t ==> isBST (insert a t)
```

Attempting to test this property using *QuickCheck* does not work well:

```
quickCheck prop_bst_insert
*** Gave up! ####
```

Here, *QuickCheck* discards random inputs as soon as it finds they do not pass the precondition (`isBST t`). Sadly, most of the inputs generated by our naïve generator suffer from this problem, and the interesting part of the property (`isBST (insert a t)`) is tested very sporadically as a result.

At this point it is reasonable to think that, to obtain the best results when using PBT over complex systems, one is forced to put a large amount of time on developing manually-written generators. In practice, that is most often the case, no automatic effort can beat a well-thought manually-written generator that produces interesting complex values and finds bugs in very few tests. Not all is lost, however. It is still possible to obtain acceptable results automatically by incorporating dynamic information from the system under test into the testing loop. The next subsection introduces the clever technique used by *FuzzChick* to find bugs in complex systems while using simple automatically derived random generators.

2.2 Coverage-Guided Property-Based Testing with *FuzzChick*

To alleviate the problem of testing properties with sparse preconditions while using simple automatically derived random generators, *FuzzChick* introduces *coverage-guided property-based testing* (CGPT) by enhancing the testing process with two key characteristics: (1) *target code instrumentation*, to capture execution information from each test case; and (2) *high-level, well-typed mutations*, to produce syntactically valid test cases by altering existing ones at the datatype level.

Using code instrumentation in tandem with mutations is a well-known technique in the fuzzing community. Generic fuzzing tools like AFL, libFuzzer or HonggFuzz, as well as language-specific ones like Crowbar use execution traces to recognize interesting test cases, e.g, those that exercise previously undiscovered parts of the target code. Later, such tools use generic mutators to combine and produce new test cases from previously executed interesting ones. *FuzzChick*, however, does this in a clever way. Instead of mutating any previously executed test case that discovers a new part of the code, *FuzzChick* integrates these fuzzing techniques into the PBT testing loop itself.

Since it is possible to distinguish semantically valid test cases from invalid ones, i.e., those passing the sparse preconditions of our testing properties as opossed to those that are discarded early, *FuzzChick* exploits this information in order to focus the testing efforts into mutating valid test cases with a higher priority than those that were discarded.

In addition, high-level mutators are better suited for producing syntactically valid mutants, avoiding the time wasted by using generic low-level mutators that act at the "serialized" level and know very little about the structure of the generated data, thus producing syntactically broken mutants most of the time. This grammar-aware mutation technique has shown to be quite useful when fuzzing systems accepting structurally complex inputs. Tools like Criterion, XSmith and LangFuzz use existing grammars to tailor the generic mutators to the specific input structure used by the system under test. In *FuzzChick*, external grammars are not required. The datatypes used by the inputs of the testing properties already describe the structure of the random data we want to mutate in a concrete manner, and specialized mutators acting at the data constructor level can be automatically derived directly from their definition.

The next subsections describe *FuzzChick*'s testing loop and well-typed mutations in detail.

2.2.1 Testing loop. Outlined in Algorithm 2, the process starts by creating two queues, *QSucc* and *QDisc* for valid and discarded previously executed test cases, respectively. Enqueued values are stored along with a given mutation energy, that controls how many times a given test case can be mutated before being finally discared.

Once inside of the main loop, *FuzzChick* picks the next test case using a simple criterion: if there are valid values enqueued for mutation, it picks the first one, mutates it and returns it, decreasing its energy by one. If *QSucc* is empty, then the same is attempted using *QDisc*. If none of the mutation queues contain any candidates, *FuzzChick* generates a new value from scratch. This selection process is illustrated in detail in Algorithm 3.

Having selected the next test case, the main loop proceeds to execute it, capturing both the result (passed, discarded, or failed) and its execution trace over the system under test. If the test case fails, it is immediately reported as a bug. If not, *FuzzChick* evaluates whether it was interesting (i.e., it exercises a new path) based on its trace information and the one from previously executed test cases (represented by *TLog*). If the test case does in fact discover a new path, it is enqueued at the end of its corresponding queue, depending on whether it passed or was discarded. This process alternates between generation and mutation until a bug is found or we reach the test limit.

The energy assigned to each test case follows that of AFL's power schedule: more energy to test cases that lead to shorter executions, or that discover more parts of the code. Moreover, to favour mutating interesting valid test cases, they get more energy than those that were discarded.

2.2.2 Well-typed mutations. Mutators in *FuzzChick* are no more than specialized random generators, parameterized by the original input to be mutated. They use a simple set of mutation operations that are randomly applied at the datatype level. In simple terms, these operations encompass:

- *shrinking the value*, replacing its top-level data constructor with one that contains a subset of its fields, reusing existing subexpressions;
- *growing the value*, replacing its top-level data constructor with one that contains a superset of its fields, reusing existing subexpressions and generating random ones when needed;

Algorithm 2: *FuzzChick* Testing Loop

```
Function Loop(P, N, gen, mut):
    i ← 0
    TLog, QSucc, QDisc ← ∅;
    while i < N do
        x ← Pick(QSucc, QDisc, gen, mut)
        (result, trace) ← WithTrace(P(x))
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            e ← Energy(TLog, x, trace)
            if not Discarded(result) then
                Enqueue(QSucc, (x, e))
            else
                Enqueue(QDisc, (x, e))
        i ← i+1
    return Ok
```

Algorithm 3: *FuzzChick* Seed Selection

```
Function Pick(QSucc, QDisc, gen, mut):
    if not Empty(QSucc) then
        (x,e) ← Deque(QSucc)
        if e > 0 then
            PushFront(QSucc, (x, e-1))
        return Sample(mut(x))
    else if not Empty(QDisc) then
        (x,e) ← Deque(QDisc)
        if e > 0 then
            PushFront(QDisc, (x, e-1))
        return Sample(mut(x))
    else return Sample(gen)
```

```
mutate_tree :: (a -> Gen a) -> Tree a -> Gen (Tree a)
mutate_tree mutate_a (Leaf x) =
  oneof [ do { x' <- mutate_a x; return (Leaf x')}                     -- Mutate recursively
        , do { l <- arbitrary; r <- arbitrary; return (Node l x r) } ] -- Grow constructor
mutate_tree (Node l x r) =
  oneof [ return l                                                     -- Return subexpression
        , return r                                                     -- Return subexpression
        , return (Leaf x)                                              -- Shrink constructor
        , do { l' <- mutate_tree l; return (Node l' x r) }             -- Mutate recursively
        , do { x' <- mutate_a    x; return (Node l x' r) }             -- Mutate recursively
        , do { r' <- mutate_tree r; return (Node l x r') } ]           -- Mutate recursively
```

Fig. 1. *FuzzChick* mutator for the `Tree` data type.

- *returning a subexpression of the same type*;
- *mutating recursively*, applying a mutation operation over an immediate subexpression.

Fig. 1 illustrates the idea behind a random *FuzzChick* mutator for our previously used `Tree` data type example. Since trees are parametric, for clarity this definition is also parameterized by a mutator for the payload (`mutate_a`), although in practice this can be abstracted away by the means of the type class system (where `mutate_tree` and `mutate_a` can be transparently coerced into an overloaded `mutate` function.)

In this mutator, branches can be shrinked into leaves by dropping the subtrees, whereas leaves can grow into branches, by reusing the payload and generating two random subtrees. Moreover, branches can be replaced with one of their subtrees. Finally, mutations can be recursively applied over both the payload and the subtrees. At the top level, all these operations are put together using the `oneof` combinator that randomly picks one of them with uniform probability.

2.2.3 Limitations of FuzzChick. Lampropoulos et al. demonstrated empirically that *FuzzChick* lies comfortably in the middle ground between using naïve automatically derived random generators and complex manually-written ones. Their results suggest that CGPT is an appealing technique for finding bugs while still offering a mostly automated workflow.

However, the authors acknowledge that certain parts of their implementation have room for improvement, especially when it comes to the mutators design. Moreover, when we recreated the evaluation of the IFC stack machine case study (described in detail in Section 5), we found that after 30 runs (as opposed to the 5 runs used original by Lampropoulos et al.), *FuzzChick was only able to find 5 out of the 20 injected bugs with a failure rate of 1*, the hardest one being found only around 13% of the time after an hour of testing. These results are presented in detail in Section 6.

At the light of these observations, we identified several aspects of *FuzzChick* that can be improved upon — and that constitute the main goal of this work. In no particular order:

- *Mutators distribution:* if we inspect the mutator defined in Fig. 1, there are two compromises that the authors of *FuzzChick* adopted for the sake of simplicity. On one hand, deep recursive mutations are very unlikely, since their probability decreases multiplicatively with each recursive call. For instance, mutating a subexpression that lies on the third level of a `Tree` happens with a probability smaller than $(1/6)^3 =\sim 0.0046$, and this only worsens as the type of the mutated value becomes more complex. Hence, *FuzzChick* mutators can only be effectively used to transform to shallow data structures, potentially excluding interesting applications that might require producing deeper valid values, e.g., programming languages, network protocols interactions, etc. Ideally, mutations should be able to happen on every subexpression of the input seed in a reasonable basis.
  On the other hand, using random generators to produce neeeded subexpressions when growing data constructors can be dangerous, as we are introducing the very same "uncontrolled" randomness that we wanted to mitigate in the first place! If the random generator produces an invalid subexpression (something quite likely), this might just invalidate the whole mutated test case. We believe that growing data constructors needs to be done carefully. For instance, by using just a minimal piece of data to make the overall mutated test case type correct. If that mutated test case to be interesting, that subexpression can always be mutated later.
- *Enqueuing mutation candidates: FuzzChick* uses two single queues for keeping valid and discarded mutation candidates. Whenever a new test case is found interesting, it is placed *at the end of its corresponding queue*. If this test case happens to have discovered a whole new portion of the target code, it will not be further mutated until the rest of the queue ahead of it gets processed. This can limit the effectiveness of the testing loop if the queues tend to grow more often than they tend to shrink, as interesting mutation candidates can get buried at the end of a long queue that only exercises the same portion of the target code. In the extreme case, they might not processed at all within the testing budget. Ideally, one would like a mechanism that prioritizes mutating test cases that discovers new portions of the code right away, and that is capable of jumping back and forth from mutation candidates whenever this happens. We show in Section 4 how this can be achieved by analyzing the execution information in order to prioritize test cases with novel execution traces.
- *Power schedule:* It is not clear how the power schedule used to assign energy to each mutable test case in *FuzzChick* works in the context of high-level well-typed mutations. If it assigns too much energy to certain not-so-interesting seeds, some bugs might not be discovered in a timely basis. Conversely, assigning too little energy to interesting test cases might cause that some bugs cannot be discovered at all unless the right mutation happens within the small available energy window — randomly generating the same test case later on does not help, as it becomes uninteresting based on historic trace information.

> To keep the comparison fair, the authors replicated the same power schedule configuration used in AFL. However, AFL uses a different mutation approach that works at the bit level. This raises the question about what is the best power schedule configuration when using a high-level mutation approach — something quite challenging to characterize in general given the expressivity of the data types used to drive the mutators.

The next section introduces Mutagen, our CGPT tool written in Haskell that aims to tackle the main limitations of *FuzzChick* using an exhaustive mutation approach that requires very little randomness and no power schedule.

3 MUTAGEN: TESTING MUTANTS EXHAUSTIVELY

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

```
type Mutation a = a -> [Mutant a]
```

```
data Mutant a = PURE a | RAND (Gen a)
```

```
mutate (Leaf x)     = [ PURE (Node def x def) ]   -- Swap constructor
mutate (Node l x r) = [ PURE l, PURE r            -- Return subexpression
                      , PURE (Leaf x)             -- Swap constructor
                      , PURE (Node l x l)         -- Rearrange subexpressions
                      , PURE (Node r x r)         -- Rearrange subexpressions
                      , PURE (Node r x l) ]       -- Rearrange subexpressions
```

```
positions (Leaf x)       = node [ (0, positions x) ]
positions (Branch l x r) = node [ (0, positions l), (1, positions x), (2, positions r) ]
```

```
inside []        mut x              = mut x
inside (0 : pos) mut (Leaf x)       = [ Leaf x'       | x' <- inside pos mut x ]
inside (0 : pos) mut (Branch l x r) = [ Branch l' x r | l' <- inside pos mut l ]
inside (1 : pos) mut (Branch l x r) = [ Branch l x' r | x' <- inside pos mut x ]
inside (2 : pos) mut (Branch l x r) = [ Branch l x r' | r' <- inside pos mut r ]
```

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

4 MUTAGEN HEURISTICS

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo.

Algorithm 4: Mutagen Testing Loop

```
Function Loop(P, N, R, gen, mut):
    i ← 0
    TLog, QSucc, QDisc ← ∅;
    while i < N do
        x ← Pick(QSucc, QDisc, gen, mut)
        (result, trace) ← WithTrace(P(x))
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            if not Discarded(result) then
                muts ← Mutants(x, mut, R)
                Enqueue(QSucc, muts)
            else if Passed(Parent(x)) then
                muts ← Mutants(x, mut, R)
                Enqueue(QDisc, muts)
        i ← i+1
    return Ok
```

Algorithm 5: Mutagen Seed Selection

```
Function Pick(QSucc, QDisc, gen):
    if not Empty(QSucc) then
        muts ← Deque(QSucc)
        if Empty(muts) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, Rest(muts))
            return First(muts)
    if not Empty(QSucc) then
        muts ← Deque(QSucc)
        if Empty(muts) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, Rest(muts))
            return First(muts)
    else return Sample(gen)
```

Algorithm 6: Mutants Initialization

```
Function Mutants(x, mut, R):
    muts ← ∅
    for pos in Flatten(Positions(x)) do
        for mutant in Inside(pos, mut, x) do
            switch mutant do
                case PURE $\hat{x}$ do
                    Enqueue($\hat{x}$, muts)
                case RAND gen do
                    repeat R times
                        $\hat{x}$ ← Sample(gen)
                        Enqueue($\hat{x}$, muts)
    return muts
```

4.1 Priority FIFO Scheduling

Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

4.2 Detecting Trace Space Saturation And Tuning Random Mutations

Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis.

```
Algorithm 7: Priority FIFO Heuristic
Function Loop(P, N, R, gen, mut):
    . . .
    (result, trace) ← WithTrace(P(x))
    . . .
    if Interesting(TLog, trace) then
        if not Discarded(result) then
            muts ← Mutants(x, mut, R)
            prio ← BranchDepth(TLog, trace)
            PushFront(QSucc, prio, muts)
        . . .

Function Pick(QSucc, QDisc, gen):
    if not Empty(QSucc) then
        (muts, prio) ← DequeMin(QSucc)
        if Empty(muts) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, prio, Rest(muts))
            return First(muts)
    . . .
```

```
Algorithm 8: Trace Saturation Heuristic
Function Loop(P, N, gen, mut):
    boring ← 0
    reset ← 1000
    R ← 1
    . . .
    while i < N do
        if boring > reset then
            TLog ← ∅
            reset ← reset * 2
            R ← R * 2
        . . .
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            . . .
        else  boring ← boring + 1
        . . .
```

Fig. 2. The two main heuristics implemented in Mutagen. Statements in red indicate important changes to the base algorithm. Ellipses denote parts of the code that are not relevant for the point being made.

Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

4.3 Mutation Inheritance

Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

5 CASE STUDIES

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

5.1 IFC Stack Machine

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

5.2 WebAssembly Engine

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

5.2.1 Testing the WebAssembly Validator. Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

```
prop_validator m =
  isValidHaskell m ==> isValidSpec m
```

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

```
isValidHaskell m =
  case validate m of
    Left  validationError -> return False
    Right validModule     -> return True
```

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

```
isValidSpec m = do
  writeFile "testcase.wasm" (dumpModule m)
  res <- shell "./wasm" ["-d", "-i", "testcase.wasm"]
  return (res == "")
```

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

5.2.2 Testing the WebAssembly Interpreter. Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

```
mk_module ty name fun =
  emptyModule { types     = [ ty ]
              , functions = [ fun ]
              , exports   = [ Export name (ExportFunc 0) ]
              , mems      = [ Memory (Limit 1 Nothing) ]
              }
```

```
prop_interpreter ty (args, fun) =
  discardAfter 20
  (do let m = mk_module ty "f" fun;
      resHs   <- invokeHaskell m "f" args;
      resSpec <- invokeSpec    m "f" args;
      return (equiv resHs resSpec))
```

```
prop_interpreter_i32 (i, f, fun) =
  prop_interpreter
    (FuncType { params = [I32, F32], result = [I32]})
    ([VI32 i, VF32 f], fun)
```

5.2.3 Real Bugs and Discrepancies found by Mutagen in haskell-wasm.

Bug #1: Invalid Memory Alignment Validation. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

Bug #2: Lax invocation checks. Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Bug #3: Allowed Out-of-bounds Memory Access. Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

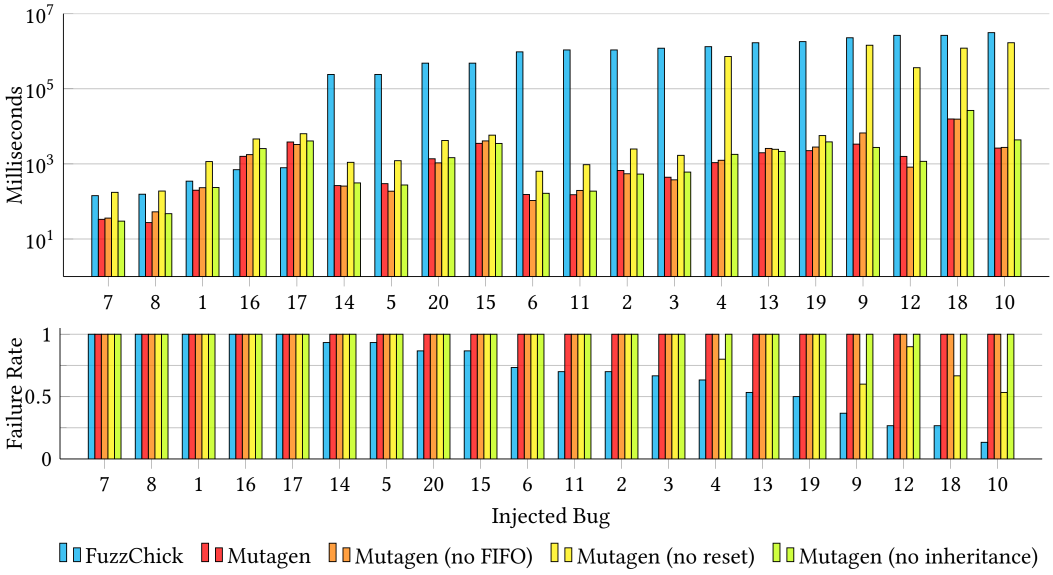
Discrepancy #1: Different Reinterpretation Semantics of NaN Values. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

Discrepancy #2: Allowed Blocks with Multiple Return Types. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

6 EVALUATION

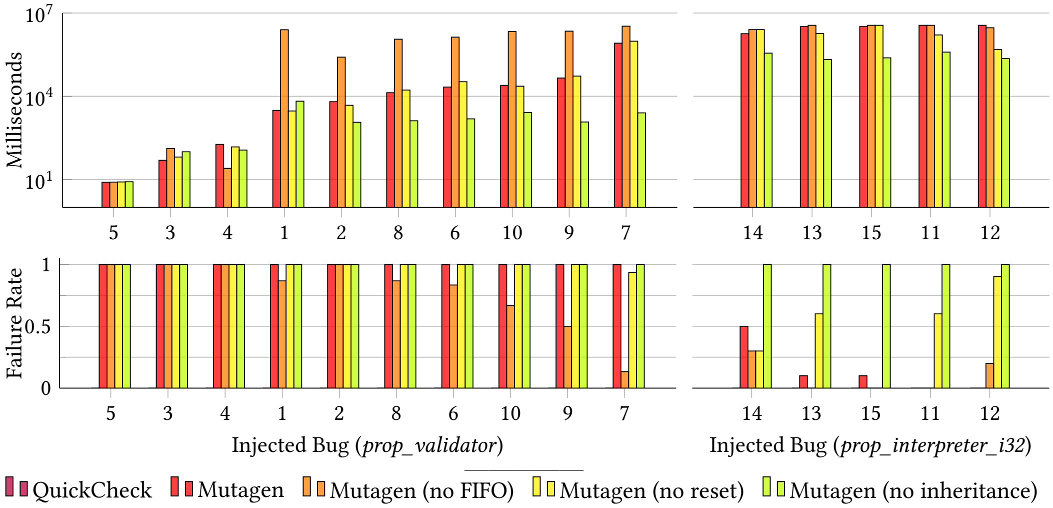
6.1 IFC Stack Machine

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.



6.2 Webassembly Engine

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.



7 IMPLEMENTATION

7.1 Deriving Mutation Machinery

Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

7.2 Tracing Haskell Programs

```
sorted []       = True
sorted [x]      = True
sorted (x:y:xs) = if x <= y then sorted (y:xs) else False
```

```
sorted []       = _trace_ 1 (True)
sorted [x]      = _trace_ 2 (True)
sorted (x:y:xs) = _trace_ 3 (if x <= y then _trace_ 4 (sorted (y:xs)) else _trace_ 5 (False))
```

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies.

8 RELATED WORK

8.1 Grammar-based Fuzzing

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

8.2 Random Data Generation

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

8.3 Exhaustive Bounded Testing

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

9 CONCLUSIONS

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

REFERENCES

Gergely Dévai, Dániel Leskó, and Máté Tejfel. 2013. The EDSL's Struggle for Their Sources. In *Central European Functional Programming School*. Springer, 300–335.

Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

A DETAILED EMPIRICAL RESULTS

Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.