Property Based Testing in MIT-Scheme

Andres Buritica Monroy and Rick Ono

Code: https://github.com/gispisquared/6515-pbt

1 Introduction

Property based testing was introduced in the QuickCheck Haskell framework in 1999 by Koen Claessen and John Hughes at Chalmers Institute of Technology. While it is widely understood that there is tremendous value in testing for software quality, there is a high cost associated with writing meaningful tests.

Software behavior can be boiled down a collection of properties that specify the program. For example we might say that `(my-sort lst)` that returns a value `sorted` should exhibit the following properties:

Given `lst` is a `list` of `integers`:

- `lst` and `sorted` are the same length.
- `lst` and `sorted` must have the same elements.
- For every item $s_i =$`(list-ref sorted i)` and $s_{i+1} =$`(list-ref sorted (+ i 1))`, $s_{i+1} \geq s_i$.

The goal of property based testing is to use just these specifications to test a program's behavior over a large input space. Once a user specifies the preconditions and properties for their code, a property based testing framework generates many inputs that fit the preconditions until it finds one that violates the given property. Once it finds one, it shrinks the input value to produce a minimal counterexample, often more useful for debugging.

Property based testing offers several benefits that complement more traditional methods. Classical testing methods such as fuzzing and hand written unit tests do offer an engineer some assurance that their program is correct, but excel either in input scope coverage or feature compliance. It may be hard to use a fuzzing approach to find useful bugs, and writing unit tests to cover a large domain of inputs is time consuming and challenging. Property based testing allows users to write arbitrarily simple or complex invariants, eliminating the need for many tedious boilerplate unit tests. For example, we could relax our property on `my-sort` to test only that it maintains a list of the same length, or if we know for some reason that `my-sort` struggles with prime numbers, write a test that restricts the input to a list of prime integers.

In our framework, we supply a new `test` procedure for MIT Scheme. We allow users to specify preconditions for their program by supplying a generator function composed from our supplied generator constructors. `test` is called with a procedure to test, a Scheme function that tests for a property, and the generator, and returns the smallest input it can find from generator for which the procedure applied to the input fails the property.

2 API and Specification

The user facing API for our framework consists of the `test` procedure, as well as the generator constructors for the user to specify inputs.

2.1 Generator Specification

We supply primitive generators that produce a value, as well as combinators that take a generator as input to generate complex inputs. Each of our primitive generator constructors return a generator, which is called with no inputs to return a value.

- `(g:constant value)`: produces the value `value`.
- `(g:integer min max)`: produces integers between `min` and `max`.
- `(g:float min max)`: produces a float between `min` and `max`.
- `(g:string charset len)`: produces a string with length `len` with characters chosen from `charset`.
- `(g:boolean probability)`: produces `#t` with probability `probability`, and `#f` with probability `(- 1 probability)`.
- `(g:random-choice choices)`: picks a value in `choices` randomly.
- `(g:random-subset choices size)`: picks and returns a list of `size` distinct random values from `choices`.

We also supply constructors for combinators that allow users to compose generators arbitrarily.

- `(g:cons gen1 gen2)`: produces a `cons`, where the first item is generated from `gen1` and the second from `gen2`.
- `(g:list gen len)`: produces a length `len` list of items generated by `gen`.
- `(g:amb gen1 gen2 probability)`: generates a value using `gen1` with probability `probability`, and a value from `gen2` otherwise.
- `(g:one-of gen1 gen2 ...)`: generates a value using one of the provided generators, all with equal probability.

Lastly, we supply a constructor `(g:restrict predicate generator)` which produces a new generator that only generates values that satisfy the predicate.

We can specify our input to our sort function using an integer generator and the list combinator:

```
(define (integer-list-gen)
  ((g:list (g:integer 0 50) 10)))

(integer-list-gen) ; => (43 1 44 32 28 49 38 42 45 12)
```

This defines a new generator `integer-list-gen` that generates a length 10 list of integers between 0 and 50. Suppose we also want to test varying length lists. We can generate the length value itself to produce varying list lengths:

```
(define (integer-list-gen)
  ((g:list (g:integer 0 50) ((g:integer 0 15)))))

(integer-list-gen) ; => (23 42 2 8 42 10)
```

Now, our generator produces a list with length between 0 and 15, with values between 0 and 50. Restricting our values to be prime numbers is as easy as adding a restrict predicate:

```
(define (integer-list-gen)
  ((g:list (g:restrict prime? (g:integer 0 50)) ((g:integer 0
     15)))))

(integer-list-gen) ; => (11 43 23 13)
```

2.2 Testing API

We define the top level testing procedure as such:

```
(define (test f property generator #!optional times timeout)
    ...)
```

The `test` procedure expects the following types for input:

- `f`: a procedure that handles inputs generated by generator.
- `property`: a procedure that takes two values, and returns `#t` or `#f`. This function will be called with an `input` generated from `generator`, and `(f input)`.
- `generator`: a function taking no inputs that generating a value to be fed into `f`. Expects the generator to be constructed from our supplied constructors, so that the shrinker can reproduce values.
- `times` (optional): how many times a value should be generated to attempt to find a counterexample.
- `timeout` (optional): how long we should wait before killing a process (for example, we do not want to wait forever for a procedure to finish, should there be an infinite loop).

Our `my-sort` function could be tested as:

```
;; the my-sort function that we'd like to test
(define (my-sort l) ...)

;; tests that our ouput matches the built-in sort
(define (sorted-version? orig l)
    (eq-vals? l (sort orig <)))

;; tests that l1 and l2 have the same values
(define (eq-vals? l1 l2) ...)

;; the generator we wrote previously
(define (integer-list-gen)
```

```
  ((g:list (g:integer 0 50) 10)))

(test my-sort sorted-version? integer-list-gen 100)
    ; => #t
    ; or, perhaps we have a bug
    ; => (0 2 5 5)
```

2.3 Model Based Testing

We can also use `test` to test that the behavior of two models, such as two implementations of an abstract data type, behave in the same way. Models must be written in a message-passing style, for example for an implementation of a set of integers:

```
(model 'add 10)    ; => 'done
(model 'delete 10) ; => 'done
(model 'has? 10)   ; => #f
```

We define generators for each of the commands, as well as a generator to generate a sequence of commands. We supply a constructor `commands-gen` that takes a list of command generators and generates a list of commands with some maximum length.

```
(define (gen-add)
  (list 'add ((g:integer 0 10))))
(define (gen-remove)
  (list 'remove ((g:integer 0 10))))
(define (gen-has)
  (list 'has ((g:integer 0 10))))
(define (gen-commands)
  ((commands-gen (list gen-has gen-remove gen-add) 100)))
```

We can feed this into `test` to test that two implementations behave the same. This could be particularly useful if we make refactoring or performance changes to a model, and want to test that our implementation does not stray from the original. We provide a utility `run-both-models` that takes two models and runs the set of commands on each model.

```
(test
  (run-both-models rep-list-set-buggy rep-alist-set)
  (lambda (in out)
    (equal? (car out) (cdr out)))
  gen-commands)) ; => ((add 0) (add 0) (remove 0) (has 0))
```

3 Implementation

3.1 Generator

Our initial design for a generator simply created a random input each time it was called. However, if we are shrinking a generated value it is helpful to have state saved so that a generated value can be reproduced and perturbed slightly. This state should be accessible whether the generator was called at a top level, or allocated within another generator, so that composed generators can still be reproduced.

Because generators need access to state regardless of where they are called, we use global variables `generator-state` and `reproduce-state` to read and write state. Every time a generator is called, it saves information allowing the behavior to be recreated to `generator-state`. To reproduce values, we copy the state from `generator-state` to `reproduce-state`, at which point the generator knows to reproduce a value from state. We also have a global boolean value `shrinking` which indicates to the generator whether it should shrink the input.

Initializing state We set the state in our functions `sample-from`, `reproduce`, and `shrink`:

```
(define (sample-from generator)
  (set! generator-state '())
  (set! reproduce-state '())
  (set! shrinking #f)
  (generator))

(define (reproduce generator original-state)
  (set! generator-state '())
  (set! reproduce-state original-state)
  (set! shrinking #f)
  (generator))

(define (shrink generator original-state)
  (set! generator-state '())
  (set! reproduce-state original-state)
  (set! shrinking #t)
  (generator))
```

In `sample-from`, we clear the `generator-state` and `reproduce-state`, and set `shrinking` to be false. This prompts the generator to create a fresh value when it is called. In `reproduce`, we copy the `original-state` into `reproduce-state`, so that the generator sees that it is in "reproduce mode" when it generates its next value. Lastly, `shrink` sets the `shrinking` value to true, so that the generator attempts to shrink the value it sees in its `reproduce-state`.

Saving State When called, the generators save state to the `generator-state` to keep track of which random values they used. We do the work for this in

our `make-atomic-generator` function, which we use to create our primitive generator constructors.

```
(define (((make-atomic-generator rand-gen transform)
                                                . params))
  (let* ((name-and-value (apply rand-gen params))
         (name (car name-and-value))
         (value (cdr name-and-value)))
    (set! generator-state
        (append generator-state (list name-and-value)))
    (transform value params)))
```

In our `make-atomic-generator`, we take our random generator and a function to transform the input. The generator applies the `params` to `rand-gen`, and sets `generator-state` to be the current `generator-state`, with the generated value appended at the end. We append the value as opposed to rewriting it so that successive generated values can all be reproduced. Then at the end the transformation is applied.

The primitive `g:integer`, along with `g:float` and `g:boolean` are written with `make-atomic-generator`:

```
(define g:integer
  (make-atomic-generator
    (lambda (mn mx) (g:random (- mx mn)))
    (lambda (value params) (+ value (car params)))))
```

Our other generators use these three generators within them, which write to state and allow us to reconstruct values. For example our `g:random-choice` uses `g:integer`, which writes to state which choice it took:

```
(define ((g:random-choice choices))
  (list-ref choices ((g:integer 0 (length choices)))))
```

And our `g:string` uses random choice:

```
(define ((g:string charset len))
  (string-append*
    ((g:list (g:random-choice charset) len))))
```

Reading State We do the work for reading state in calls to `random` and `random-real`, which have the following behavior:

```
(define (g:random)
  (if (null? reproduce-state)
    (random)
    (let ((old-val (car reproduce-state)))
      (set! reproduce-state (cdr reproduce-state))
      old-val)))
```

If the `reproduce-state` is null, we want to create a new random value. If we do have contents in the reproduce state, we reproduce that old value, take it out

of the reproduce state, and return the old value instead of producing a random one.

We implement `g:random` and `g:random-real` using an auxiliary function `make-random` as shown below, with details about shrinking omitted.

`make-random` keeps track of a name along with the parameters in the state. This way, even if there is a branch in the user-provided generator that is not indicated in our state, we will be able to tell that we are calling the wrong random generator. For example, if we see that we are calling `(random 0 10)` instead of `(random 5 10)`, or `random` instead of `random-real`, we will know to exit reproduce mode.

```
(define ((make-random rand shrink name) . params)
  ;; Check if we are in "reproduce mode"
  (define reproduce
    (and
      (pair? reproduce-state)
      (equal? (caar reproduce-state) (cons name params))))
  ...
  (cons (cons name params)
    (if (not reproduce)
      ;; If we are not, then we create a new value
      (begin
        (if (pair? reproduce-state)
          (set! reproduce-state (cdr reproduce-state)))
        (apply rand params))
      ;; If we are, reproduce the old one
      (let ((old (cdar reproduce-state)))
        (set! reproduce-state (cdr reproduce-state))
        ...
          old)))))
```

`g:random` is then simply defined as

```
(define g:random (make-random random random 'random))
```

Given this, we are able to generate and then reproduce values:

```
(define symbol-gen (g:symbol '(a b c) 5))
(sample-from symbol-gen) ; => 'aacbb
(reproduce symbol-gen generator-state)) ; => 'aacbb

(sample-from (g:random-subset (iota 10) 3)) ; => (7 5 0)
(reproduce
  (g:random-subset (iota 10) 3) generator-state) ; => (7 5 0)
```

Predicates and Assertions Initially, a predicate was simply a way of restricting the output of a generator so that it satisfies a certain condition: specifically, `(g:restrict pred gen)` creates a new generator restricted to values that satisfy the predicate.

However, when the predicates interact with previously generated values, it is possible that `gen` cannot generate a value satisfying `pred`. For example, consider the following generator for Pythagorean triples:

```
(define (gen-pythag)
  (define a ((g:integer 1 100)))
  (define b ((g:integer 1 100)))
  (define c ((g:restrict
               (lambda (c)
                 (= (+ (square a) (square b))
                    (square c)))
               (g:integer 1 100))))
  (list a b c))
```

If `restrict` only retries the generator until success, this would likely fall into an infinite loop (since not all values of `a` and `b` allow a value of `c` satisfying the conditions). In this case, it is possible to write a generator that works with this behaviour of `restrict`:

```
(define gen-pythag-restrict
  (g:restrict
    (lambda (l) (= (+ (square (first l)) (square (second l)))
                   (square (third l))))
    (g:list (g:integer 1 100) 3)))
```

However, in larger generators, this would have worse performance and shrinking behaviour since it would need to regenerate all of the variables that a predicate depends on in order to test the predicate again. (For example, if instead of generating Pythagorean triples we wanted to generate a 10-element list such that the sum of the squares of the first 9 elements equals the square of the 10th, it would take a long time to generate this list by chance since the program would simply try random 10-element lists.)

Instead, we define a backtracking-based restriction system based on the `amb` system presented in class. The entry point is `g:assert`, which tests a condition and starts backtracking to the last saved state if the condition is not met:

```
(define (g:assert condition)
  (if (not condition)
    (read-global-state)))
```

This function `g:assert` also provides a nice way to write conditions that depend on multiple generated values:

```
(define (gen-pythag-assert)
  (define a ((g:integer 1 100)))
  (define b ((g:integer 1 100)))
  (define c ((g:integer 1 100)))
  (g:assert (= (+ (square a) (square b)) (square c)))
  (list a b c))
```

Now `g:restrict` is easy to define in terms of `g:assert`:

```
(define ((g:restrict predicate generator))
  (define val (generator))
  (g:assert (predicate val))
  val)
```

To backtrack, we keep a stack of continuations that also includes information about the global state at the time the continuation was saved, and how many times the continuation has been called:

```
(define continuations '())
(define (save-global-state)
  (call/cc
    (lambda (k)
      (set! continuations
        (cons
          (list k 0 generator-state reproduce-state shrinking
   )
          continuations)))))
```

Now by calling `(save-global-state)`, we save the current state to the stack. In particular, we do this at the start of our `make-random` function:

```
(define ((make-random rand shrink name) . params)
  (save-global-state) ; backtrack to here
  ...)
```

If a condition we are asserting fails, we can then backtrack to the saved state at the front of the stack. To break out of infinite loops, once a continuation has been called 100 times we remove it from the stack. The function `(read-global-state)` responsible for this is defined as follows:

```
(define (read-global-state)
  (if (null? continuations) (error "No more backtracking
   possible - assert could not be satisfied"))
  (let ((state-to-read (car continuations)))
    ; increment how many times the continuation has been
    called
    (set-car! (cdar continuations)
              (+ 1 (cadar continuations)))
    ; backtrack further if necessary:
    (cond ((> (cadar continuations) 100)
           (set! continuations (cdr continuations))
           (read-global-state)))
    ; restore the global state present at that time:
    (set! generator-state (third state-to-read))
    (set! reproduce-state (fourth state-to-read))
    (set! shrinking (fifth state-to-read))
    ; call the continuation:
    ((first state-to-read) #f)))
```

3.2 Shrinker

Our main logic for shrinking is in our `make-random` function. The important parts for shrinking are shown in the snippet below.

```
(define ((make-random rand shrink name) . params)
  (define reproduce ...)
  ...
  (cons (cons name params)
    (if (not reproduce)
      ; ... not reproduce case omitted ...
      (let ((old (cdar reproduce-state)))
        (set! reproduce-state (cdr reproduce-state))
        (if shrinking
          (cond
            ((and
               (> old 0)
               (< (random-real) 0.5))
             (set! shrinking #f)
             (shrink old))
            (else
              (set! reproduce #f)
              old))
          old)))))
```

If we are in both in "reproduce mode" and `shrinking` is true, then we shrink our old value using the provided `shrink` procedure. We introduce some randomness, as it is sometimes unclear that a value cannot be shrunk further without breaking a predicate, for example if we have a generator

```
(g:restrict prime? (g:integer 200 250))
```

and `old` is `211`, we do not want to repeatedly try to shrink until we find a number between 200 and 210 that is prime, because such a number does not exist. The `shrink` for `random` is simply `random` which generates a new random number between 0 and the old value.

To shrink combinators, we keep additional structure in our `generator-state` for combinators. If we have a generator that generates a `cons` of a `list` and `float`, the `generator-state` after sampling will store the length of the list in addition to the elements.

```
(define ((g:list gen len))
  ...
  (set! generator-state (append original-state
                                (list (cons (cons 'list len)
                                            (cdr all-gen)))))
 ...)

 ....

(define (random-length-list-and-float)
```

```
  ((g:cons
  (g:list (g:integer 0 10) ((g:integer 0 5)))
  (g:float 0 5))))
(sample-from random-length-list-and-float)
  ; => ((0 8 1) . 2.0696241484665436)
generator-state
  ; => (((random 5) . 3) (cons (((list . 3) (((random 10) .
    0)) (((random 10) . 8)) (((random 10) . 1)))) ((
    random-real) . .41392482969330874)))
```

If the generated 4-element list is shrunk to a 1-element list, we need to remove most of the `reproduce-state` corresponding to the list:

```
(shrink random-length-list-and-float reproduce-state)
  ; => ((3) . 4.295354403700995)
generator-state
  ; => (((random 5) . 1) (cons (((list . 1) (((random 10) .
    3)))) ((random-real) . .859070880740199)))
```

3.3 Test

Our top-level `test` function is a simple loop that attempts to find an input that fails the given property.

```
(define (test f property generator #!optional times timeout)
  (if (default-object? times)
    (set! times 100))
  (if (default-object? timeout)
    (set! timeout 100))
  (let lp ((n times))
    (if (eq? n 0) #t
      (let ((result (test-property
                     property
                     f
                     (sample-from generator)
                     timeout)))
        (if (eq? result #t)
          (lp (- n 1))
          (test-shrinks f
                        property
                        generator
                        generator-state
                        timeout))))))
```

Our main loop runs a maximum of `times` times, and in each iteration tests the property using `test-property` using a fresh input from the generator. If the property passes, we try again, but if it does not, we attempt to shrink it inside `test-shrinks`.

Timeouts are handled within `test-property`, and a default timeout of 100ms is given. If the function does not return in this time, we count this as a failed

example and continue to shrink the input to find a minimal timing out example. This allows the user to find potential infinite loops, though if they find that the counterexample returned does satisfy their property, they may want to try supplying a longer timeout. In verbose mode, we print the cases that failed from a timeout, so users have more insight into why their code failed.

```
(define (test-property property f input timeout)
  (define done #f)
  (register-timer-event
    timeout
    (lambda () (if (not done) (error "timed out"))))
  (define result
    (guard
      (condition
        (else
          (if (equal? (error-object-message condition)
                      "timed out")
              'time-out
              'internal-error)))
      (property input (f input))))
  (set! done #t)
  result)
```

Once a counterexample is found we enter `test-shrinks`, which attempts to shrink the input that failed the property. Here we count `'time-out` and `'internal-error` as a failed test, and continue to shrink until we find a minimal failing example.

```
(define (test-shrinks f property generator original-state
    timeout)
  (let lp ((n (max 100 (num-leaves original-state))))
    (if (eq? n 0)
      (reproduce generator original-state)
      (let* ((input (shrink generator original-state))
             (result (test-property property
                                    f input timeout)))
        (cond
          ((or shrinking (eq? result #t))
           (lp (- n 1)))
          (else
            (cond
              (verbose
                (cond ((eq? result 'time-out)
                       (display "failed (timeout): "))
                      ((eq? result 'internal-error)
                       (display "failed (internal error): "))
                      (else (display "failed: ")))
                (pp input)))
              (test-shrinks f
                            property
                            generator
```

```
                                generator-state
                                timeout)))))))
```

4 Applications

4.1 Problem set 0

We can apply property based testing to simple applications such as problem set 0's cryptography. In problem set 0, we defined a system to use the Diffie-Hellman protocol to encode and decode messages. We can easily write tests to find minimal strings that are not handled by our system.

```
(define dh-system (public-dh-system 100))

(define Alyssa (eg-receiver dh-system))

(define (send-alyssa message)
  (eg-send-message message Alyssa))

(define (string-generator)
  (let ((str-len ((g:integer 0 100))))
    ((g:string (list "a" "b" "c" "d" "e") str-len))))

(test send-alyssa equal? string-generator 100 10000)
  ; =>  "caaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
```

Sure enough, the returned string has 42 characters, which we found to be the failing point for this protocol.

The `prime?` predicate in problem set 0 used Fermat's Little Theorem to test for primality, which we know does not work in every case. We can define a `better-prime?` predicate and feed it into our tester to allow us to find these so called Carmichael Numbers.

```
(define (prime-property in out)
  (eq? out (better-prime? in)))

(test prime? prime-property (g:integer 0 1000))
  ; => 561
(test prime? prime-property (g:integer 1000 10000) 100 1000)
  ; => 2465
```

4.2 Symbolic arithmetic

We can also use property based testing to find behavior like floating point errors.

First we define a generator `gen-symbolic` to generate symbolic arithmetic expressions. This generates a symbolic expression containing arithmetic operators arbitrarily applied to floats and symbols `a`, `b`, and `c`. Then we define a generator `gen-symbolic-and-values` which produces a cons of a symbolic expression, and then assignments of values to `a`, `b`, and `c`.

```
(define (gen-symbolic)
  ((g:one-of
```

```
    (g:random-choice '(a b c))
    (g:float 0 1)
    (g:cons
      (g:random-choice '(+ - *))
      (g:list gen-symbolic 2)))))

(define gen-symbolic-and-values
  (g:cons
    gen-symbolic
    (lambda ()
      (list
        (zip '(a b c) ((g:list (g:float 0 1) 3)))))))
```

One thing to note here is that this generator is recursive: if we write it carelessly, it can create an infinite recursion. Specifically, if the `gen-symbolic` case occurs first instead of third, `g:one-of` will shrink towards it and create longer and longer expressions. Furthermore, if `gen-symbolic` calls itself more than once in expectation, it does not terminate with probability 1. Since we want the test to not trigger our timeout due to the expression generated being too large, we have written `gen-symbolic` so that it calls itself $\frac{2}{3}$ times in expectation.

Then we define functions that allow us to substitute the values and evaluate.

```
(define (substitute expr alist)
  (if (pair? expr)
    (cons (substitute (car expr) alist)
          (substitute (cdr expr) alist))
    (let ((expr-val (assq expr alist)))
      (if (pair? expr-val)
        (cadr expr-val)
        expr))))

(define (evaluate symbolic-and-values)
  (eval (apply substitute symbolic-and-values)
        system-global-environment))
```

Now we can generate symbolic expressions and values associated, and evaluate the expressions using those values. One property that should hold is that simplifying the symbolic expression with `algebra-2` from SDF, then evaluating should yield the same result as evaluating the symbolic expression. Let's test that.

```
(define (simplification-works symbolic-and-values)
  (define first (evaluate symbolic-and-values))
  (define second
     (evaluate
       (cons
         (algebra-2 (car symbolic-and-values))
         (cdr symbolic-and-values))))
  (= first second))
```

When we test with `simplification-works`, we should get a counterexample in `symbolic-and-values`.

```
(define symbolic-and-values
  (test simplification-works
        (lambda (in out) out)
        gen-symbolic-and-values
        1000
        1000))
(evaluate symbolic-and-values)    ; => 1.2306956622401857
(evaluate
    (cons
      (algebra-2 (car symbolic-and-values))
      (cdr symbolic-and-values))) ; => 1.230695662240186
```

We can find out what expressions, and what values substituted for them, produced the floating-point discrepancy:

```
(car symbolic-and-values)
; => (+ (+ (- .33441419848879145 a) .8962814637513944) a)
(algebra-2 (car symbolic-and-values))
; => (+ .8962814637513944 a (- .33441419848879145 a))
(cadr symbolic-and-values)
; => ((a .6823164351138803) (b 0) (c 0))
```

5 Conclusion

5.1 Future work

Syntactic closure While it should be possible to test any program in our framework, there are potential issues when it comes to naming. If a program supplied to our `test` uses names that conflict with our internal functions, we may get naming conflicts that result in unintended behavior. One solution to this would be to run the functions on the generated inputs inside a syntactic closure, assigning unused fresh names to each symbol.

Better shrinking The shrinker we have written adequately shrinks most inputs, though there are many heuristics that could be added to converge on failing inputs with higher probability. Given a large failing example, we do try to shrink the values and the size of collections, but certain mutations might offer a larger chance of a successful shrink.

Testing other behavior Currently, our properties can test that the output of a function on an input has some property, but there are functions that have other effects that this framework currently cannot test. Though it is possible to write wrappers around tested functions in order to simulate this functionality, there are many features and syntactic sugars that we could add to make the user experience more streamlined.

5.2 Conclusion

We hope that this property based testing framework provides a solution to testing programs. The framework is robust in its ability to produce complex inputs and properties, and is capable of producing counterexamples to non-trivial programs.