A simple incremental development of a property-based testing tool (Functional Pearl)

Rudy Braquehais
University of York
rmb532@york.ac.uk

Michael Walker
University of York
msw504@york.ac.uk

José Manuel Calderón Trilla
Galois Inc.
jmct@jmct.cc

Colin Runciman
University of York
colin.runciman@york.ac.uk

Abstract

Property-based testing tools, like QuickCheck, are widely used for testing Haskell programs. Since QuickCheck's introduction in 2000, several other similar tools and techniques have been developed. There have been papers, book chapters, and countless blog posts on how to use those tools. In this paper, we describe how to write one. The purpose is to be educational: we don't present new techniques for generation of values or property-based testing. Instead, we present a way to derive such techniques. We start with a very simple implementation (25 LOC), then iteratively refine it into a full featured property-based testing tool (<100 LOC).

CCS Concepts •**Software and its engineering →Software testing and debugging;**

Keywords property-based testing, systematic testing, Haskell.

ACM Reference format:
Rudy Braquehais, Michael Walker, José Manuel Calderón Trilla, and Colin Runciman. 2017. A simple incremental development of a property-based testing tool (Functional Pearl). In *Proceedings of Haskell Symposium 2017, Oxford, UK, September 2017 (Haskell'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Testing is by far the most common approach to ensure software quality, but writing good tests can be difficult. In Haskell, we commonly use *property-based* testing to make this easier. Rather than manually listing input–output pairs, property-based testing instead allows the programmer to write *properties* which should hold for *all* input values. A property-testing tool then takes care of generating inputs, freeing the programmer from having to decide which inputs are necessary to test.

There are several tutorial articles, such as [4] and [13], explaining how to *use* these tools. Our aim in this paper is to give a tutorial development showing how to *write* one. We start with a very simple implementation in (§2), then iteratively refine it in (§3–7).

The target audience of this paper is:

- late undergraduate students and early graduate students;
- lecturers intending to explore property-based testing;
- researchers starting to work with property-based testing.

Haskell'17, Oxford, UK

DOI: 10.1145/nnnnnnn.nnnnnnn

Example 1.1. Consider the following (faulty) `sort` function:

```
sort :: Ord a => [a] -> [a]
sort []     =  []
sort (x:xs) =  filter (< x) xs
            ++ [x]
            ++ filter (> x) xs
```

Tests In a traditional approach to software testing, the programmer explicitly lists tests by providing expected input–ouput pairs for functions. For example, the following are tests for `sort`:

```
sortTests :: [Bool]
sortTests  =  [ sort []      == []
              , sort [1,2,3] == [1,2,3]
              , sort [3,2,1] == [1,2,3]
              ]
```

This approach is usually known as *unit testing* [19].

Properties Tests can be parameterized over values. We call such parameterized tests *properties*. The following are two properties of a `sort` function:

```
prop_sortOrdered :: Ord a => [a] -> Bool
prop_sortOrdered xs = ordered (sort xs)

prop_sortCount :: Ord a => a -> [a] -> Bool
prop_sortCount x xs = count x (sort xs) == count x xs
```

The first property states that for all lists, sorting yields an ordered list. The second states that the counts of elements do not change after sorting. Together these two properties form a complete specification of `sort`. As *single samples* of test results, we have:

```
prop_sortOrdered [1,2,3] == True
prop_sortCount 1 [1,2,3] == True
```

Property-based testing Property-based testing tools provide a `check` function that takes a property, tests it by automatically generating test values, then reports the results. The following illustrates typical usage:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK
> check (prop_sortCount :: Int -> [Int] -> Bool)
*** Failure:  0 [0,0]
```

The `sort` function follows the first property but fails the second due to a fault: it discards repeated elements. □

The above example uses of check exemplify what makes property-based testing compelling. When the system is able to find a counterexample it not only reports that the property failed, it also reports the inputs that caused the property to fail. It is not the user's responsibility to find the crucial test values; they are found automatically. This is the heart of property-based testing. Because the specified properties should hold for *all* inputs we leave it to the testing tool to generate candidate input values. The strategy that a testing tool uses to generate input values is important and we revisit the design of such a value-generator throughout this paper.

Roadmap In this paper we describe the implementation of a property-based testing tool as a series of refinements of a basic first attempt. Section 2 presents the initial version. Sections 3–7 detail succssive refinements. Section 8 briefly compares the result with LeanCheck. Section 9 provides an overview of related work. Section 10 offers some closing thoughts.

2 Mark I: Generate and Test

Listable types As we implied in the introduction, a key ingredient of every property-based testing library is a generator of test values. In this paper, we choose to enumerate, or more precisely *list*, test values. Because we need to be able to generate values of many types, we take advantage of Haskell's typeclass machinery. We say that `Listable` types are those for which there is a declared `list` of (ideally all) values of that type.

```
class Listable a where
  list :: [a]
```

Booleans For types with finitely many values, we can simply list them directly. Here is a `Listable` instance for `Bool`:

```
instance Listable Bool where
  list = [False,True]
```

Integers The set of integers extends infinitely in both directions, so we cannot simply enumerate them in order. When generating test values we often find it useful to combine two separate lists. For this reason we define a function, `\/`, that lazily interleaves two lists:

```
(\/) :: [a] -> [a] -> [a]
[]     \/ ys = ys
(x:xs) \/ ys = x:(ys \/ xs)
```

Using `\/`, we combine the infinite list of positive integers with the infinite list of negative integers, giving us the following `Listable` instance for `Int`:

```
instance Listable Int where
  list = [0,-1..] \/ [1..]
```

Evaluating `list :: [Int]` yields:

```
[0, 1, -1, 2, -2, 3, -3, 4, -4, 5, ...]
```

This pattern of combining multiple infinite lists to create an enumeration is used extensively in this paper.

Pairs Before defining `Listable` pairs, we define a function (><) with the following type signature:

```
(><) :: [a] -> [b] -> [(a,b)]
```

It takes the product of two lists as a list of pairs. We might be tempted to simply define it as:

```
xs >< ys = [(x,y) | x <- xs, y <- ys]  -- WRONG!
```

but that will not do! If `ys` is infinite, we are stuck with enumerating the `head` of `xs` paired with infinitely many values of `ys`. Therefore, we interleave lists starting with each `x`:

```
[]     >< ys = []
xs     >< [] = []
(x:xs) >< ys = [(x,y) | y <- ys] \/ (xs >< ys)
```

Using `><`, we define `Listable` pairs:

```
instance (Listable a, Listable b)
      => Listable (a,b) where
  list = list >< list
```

This declaration is not recursive. Three different instances of `list` are involved!

```
  list :: [(a,b)] = (list :: [a]) >< (list :: [b])
```

Lists Using lists of pairs, `Listable` lists can be defined as follows:

```
instance Listable a => Listable [a] where
  list = [] : [x:xs | (x,xs) <- list]
```

We start with the empty list. Then, we list all lists of the form `x:xs`. Since we can list pairs, that is done by simply listing pairs of type `(a,[a])` then applying the list constructor `(:)`. This declaration *is* recursive.

Searching for counter-examples Once we can list values of a type, we can check a property using a finite subset of them. We define a `counterExamples` function that lists any counter-examples of a property found by applying it to a limited number of test values:

```
counterExamples :: Listable a
                => Int -> (a -> Bool) -> [a]
counterExamples n p =
  [x | x <- take n list, not (p x)]
```

Showing test results Using the `counterExamples` function, we define the `checkFor` function. For a given maximum number of test values, it reports whether a property is true:

```
checkFor :: (Show a, Listable a)
         => Int -> (a -> Bool) -> IO ()
checkFor n p =
  case counterExamples n p of
    []    -> putStrLn $ "+++␣OK!"
    (x:_) -> putStrLn $ "***␣failed␣for:␣" ++ show x
```

For convenience, we also provide a `check` function that fixes the maximum number of test values:

```
check :: (Show a, Listable a) => (a -> Bool) -> IO ()
check = checkFor 200
```

Example 1.1 (revisited). We can now test the properties from the introduction:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK
> check (uncurry prop_sortCount :: (Int,[Int]) -> Bool)
*** failed for: (0,[0,0])
```

The second property has to be uncurried. In §4 we will describe what is needed for it to appear naturally as a curried function. □

3 Mark II: Algebraic Datatypes

Though lists and tuples are often used in functional programs, so are algebraic data-types (ADTs). For example, take "Hutton's Razor" [9]:

```
data Expr = Val Int
          | Add Expr Expr
  deriving (Show, Eq)
```

For our tool to be applicable to a wider range of programs, we must be able to work with ADTs such as `Expr`. In the rest of this section we present additions to Mark I that gain us the ability to do so.

Shape shifting The key insight is that we already have the necessary tools for generating values of datatypes such as `Expr` but they are just the wrong *shape*.

Take `Val`s for instance, we already know how to list `Int`s, we just need to wrap them in `Val` constructors. To this end we define `cons1`, for constructors with 1 argument:

```
cons1 :: Listable a => (a -> b) -> [b]
cons1 c = [c x | x <- list]
```

This idea generalises nicely to constructors with more than one argument, which are really just tuples with a different outermost constructor:

```
cons2 :: (Listable a, Listable b)
      => (a -> b -> c) -> [c]
cons2 c = [c x y | (x,y) <- list]
```

The combinators `cons3`, `cons4`, ..., `cons<N>` can be created similarly.

Nullary constructors are also useful, and so we provide `cons0` that simply wraps the value in a list:

```
cons0 :: a -> [a]
cons0 c = [c]
```

With these functions defined, it is a simple matter to define an instance of `Listable` for `Expr`:

```
instance Listable Expr where
  list = cons1 Val
      \/ cons2 Add
```

Some readers may have realised that `cons1` and its siblings are not strictly necessary. Without the `cons1` and `cons2` functions our `Listable` instance for `Expr` could be defined as follows:

```
instance Listable Expr where
  list = [Val  c   | c     <- list]
      \/ [Expr x y | (x,y) <- list]
```

However, as our ADTs grow in size (in number of constructors or number of fields) this method becomes tedious and error-prone, which is what motivated the `consN` abstractions.

In fact, we can redefine the instances for `Bool` and `[a]` using our new, more general, tools:

```
instance Listable Bool where
  list = cons0 False
      \/ cons0 True

instance Listable a => Listable [a] where
  list = cons0 []
      \/ cons2 (:)
```

***Testing properties of* `Expr`s** We can now test properties of `Expr`s. The constructor `Add` is not commutative, as the automatically derived (`==`) for `Expr`s is structural:

```
> check $ \(e1,e2) -> Add e1 e2 == Add e2 e1
*** failed for: (Add (Val 0) (Val 0),Val 0)
```

But, with `eval` defined as

```
eval :: Expr
eval (Val i) = i
eval (Add e1 e2) = eval e1 + eval e2
```

the constructor `Add` is commutative under `eval`:

```
> check $ \(e1,e2) -> eval (Add e1 e2)
>                  == eval (Add e2 e1)
+++ OK!
```

4 Mark III: Multi-argument Properties

Mark I and II have a problem: `check` and related functions can only test properties with one argument. Multiple arguments have to be encoded in tuples. It might seem that writing rather than

```
check $ \(x,y,z) -> x + (y + z) == (x + y) + z
```

```
check $ \x y z -> x + (y + z) == (x + y) + z
```

would be

`Testable` types Ideally, we would like to test properties with types:

```
  a           -> Bool
  a -> b      -> Bool
  a -> b -> c -> Bool
  ...
```

Where all of the argument types are `Listable`.

In other words, we would like to overload our `check` function so that its property argument can have any arity. For that, we define the typeclass of `Testable` properties:

```
class Testable a where
  results :: a -> [Result]
```

Its only function, `results`, takes a `Testable` property, and returns a list of `Result`s. Where the type `Result` is the following type synonym:

```
type Result = ([String],Bool)
```

The first element of the pair is a list of `String`s, with each element of the list representing an argument to the property. The second element of the pair is the boolean result of testing the property for these arguments.

Testable booleans We can now define our first `Testable` instance: `Bool`. A boolean value is a property with no arguments where the only result is its value.

```
instance Testable Bool where
  results p = [([],p)]
```

On its own being able to test properties of type `Bool` is not very useful. However, all of the properties we are concerned with will return a boolean value. The instance above is the base case in a type-level recursion.

Testable functions The recursive, and final, case is our instance for functions:

```
instance (Show a, Listable a, Testable b)
      => Testable (a -> b) where
  results p =
    foldr (\/) [] [resultsFor x | x <- list]
    where
    resultsFor x =
      [(show x:as,r) | (as,r) <- results (p x)]
```

For testable properties of type `a -> b` the argument type, `a`, must have `Show` and `Listable` instances so that we can represent the arguments as strings and generate test argument values, respectively. The result type, `b`, must be `Testable`: note the partial application `p x` giving a specialised version of property `p` with `x` fixed as the test value for the first argument. As `(->)` associates to the right, `a -> (c -> Bool)` is the same as `a -> c -> Bool`, and we can instantiate `b` at a function type as long as the final result type is `Bool`.

Finding counter-examples of* `Testable` *values We can now generalise the `counterExamples` function to take any `Testable` argument, not just a unary predicate:

```
counterExamples :: Testable a
                => Int -> a -> [[String]]
counterExamples m p =
  [as | (as,False) <- take m (results p)]
```

The functions `checkFor` and `check` change correspondingly:

```
checkFor :: Testable a => Int -> a -> IO ()
checkFor n p =
  case counterExamples n p of
    []     -> putStrLn $ "+++ OK!"
    (ce:_) -> putStrLn $ "*** failed for: "
                      ++ unwords ce

check :: Testable a => a -> IO ()
check = checkFor 200
```

In `checkFor`, we apply `unwords` to the counter-example, as it is a list of strings.

Now `check` can be used with properties of any arity:

```
> check $ \x y z -> (x + y) + z == x + (y + z :: Int)
+++ OK
```

Example 1.1 (revisited). The `prop_sortCount` property from §1 can now be tested in its natural curried form:

```
> check (prop_sortCount :: Int -> [Int] -> Bool)
*** failed for: 0 [0,0]
```

□

5 Mark IV: Fair Enumeration

Mark I, Mark II and Mark III all share two related drawbacks: we do not always get the *simplest* counter-example (see Example 5.1) and we have to configure a unreasonable number of tests to find some simple counter-examples (see Example 5.2)

Example 5.1. Consider the following two functions for rotating `Expr`s with the intended property that one reverses the effect of the other:

```
rotateL :: Expr -> Expr
rotateL (Add e1 (Add e2 e3))  =  Add (Add e1 e2) e3

rotateR :: Expr -> Expr
rotateR (Add (Add e1 e2) e3)  =  Add e1 (Add e3 e2)

prop_rotRotId :: Expr -> Expr -> Expr -> Bool
prop_rotRotId e1 e2 e3 = rotateR (rotateL e) == e
  where e = Add e1 (Add e2 e3)
```

Passing `prop_rotRotId` to `check`, we get the following:

```
> check prop_rotRotId
*** failed for: (Val 0) (Add (Val 0) (Val 0)) (Val 0)
```

The function `rotateR` is faulty. The expressions `e2` and `e3` are swapped in the function result. However, this is not the simplest counter-example to illustrate the fault. Ideally, we would like to get something like:

```
*** failed for: (Val 0) (Val 0) (Val 1)
```

as it is smaller, both in string length and in the number of constructors. This arguably makes debugging easier. □

Example 5.2. Consider this faulty implementation of a `merge` function

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys  =  ys
merge xs []  =  take 2 xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys
```

that does not adhere to the following expected property:

```
prop_elemMerge :: Int -> [Int] -> [Int] -> Bool
prop_elemMerge x xs ys =
  (elem x xs || elem x ys) == elem x (merge xs ys)
```

Yet using Mark III (§4) and the default number of tests (200), we get no counter-example:

```
> check prop_elemMerge
+++ OK!
```

If we increase the number of tests to 10000, we find a counter-example:

```
> checkFor 10000 prop_elemMerge
*** failed for: 1 [0,0,1] []
```

The counter-example is not complex, yet it only appears as the 8190^{th} test in our enumeration. □

Cause of these issues What causes the issues in Examples 5.1 and 5.2? The problem can be illustrated with pairs of `Int`s:

```
> list :: [(Int, Int)]
[(0,0),(1,0),(0,1),(-1,0),(0,-1),(1,1),(2,0),...]
```

The first few values above are as we might expect. However, if we look a little further:

```
> drop 100 $ list :: [(Int, Int)]
[(0,-25),(1,13),(0,26),(2,-3),(0,-26),(1,-13)...]
```

the enumeration varies the second element of our tuples more rapidly than the first. For instance:

```
> import Data.List (findIndex)
> let list' = list :: [(Int, Int)]
> findIndex (== (0,9)) list'
Just 34
> findIndex (== (9,0)) list'
Just 131071
```

Intuitively, `(0,9)` and `(9,0)` are equally simple. We would prefer them to appear close to each other in our enumeration.

Even more surprising, but for the same reason, in the current enumeration, `(2,2)` appears later than `(0,9)` being the 55^{th} and 34^{th} enumerated values, respectively. As `(2,2)` is intuitively simpler than `(0,9)`, we would prefer it to appear first.

5.1 Tiered enumeration

To solve the problem of unfair enumeration, we reify the intuition of "simplicity". We create *tiers* of values and redefine the `Listable` class in terms of these tiers.

```
class Listable a where
  tiers :: [[a]]
```

A `Listable` instance's `tiers` value is a possibly infinite list of *finite* sublists of values characterised by some notion of *size*. Each sublist represents a tier: the first tier contains values of size 0, the second tier contains values of size 1, and so on. Now, to list all values of a type, we concatenate tiers:

```
list :: Listable a => [a]
list = concat tiers
```

Goal Before going into the details of how to define tiers for arbitrary types, we can gain a greater intuition for `tiers` by studying simple examples:

Booleans Both False and True have size 0:

```
tiers :: [[Bool]] = [[False,True]]
```

Words We define the following for tiers of `Word`:

```
tiers :: [[Word]] = [[0],[1],[2],[3],[4],[5],...]
```

There is only one natural of each size: `0` has size `0`, `1` has size `1`, `2` has size `2`, `3` has size `3`, and so on.

Pairs The size of pairs is given by the sum of sizes of its elements. For pairs of words, we then have:

```
tiers :: [[(Word,Word)]] =
  [ [(0,0)]
  , [(0,1),(1,0)]
  , [(0,2),(1,1),(2,0)]
  , [(0,3),(1,2),(2,1),(3,0)]
  , ...
  ]
```

Revisiting the examples of `Int` pairs:

```
> findIndex (== (2,2)) list'
Just 24
> findIndex (== (0,9)) list'
Just 153
> findIndex (== (9,0)) list'
Just 170
```

`(0,9)` and `(9,0)` now appear closer in the enumeration, and `(2,2)` appears before both.

5.2 Manipulating tiers

`(\/)` *for tier-lists* The sum of two tier-lists is defined by:

```
(\/) :: [[a]] -> [[a]] -> [[a]]
xss \/ []  = xss
[]  \/ yss = yss
(xs:xss) \/ (ys:yss) = (xs ++ ys) : xss \/ yss
```

For tier-lists with the same number of tiers, `\/` is equivalent to `zipWith (++)` .

The old version of `\/` is still useful, so we rename it `interleave`:

```
interleave :: [a] -> [a] -> [a]
[]     `interleave` ys = ys
(x:xs) `interleave` ys = x:(ys `interleave` xs)
```

`(><)` *for tier-lists* The product of two tier-lists is defined by:

```
(><) :: [[a]] -> [[b]] -> [[(a,b)]]
_        >< []  = []
[]       >< _   = []
(xs:xss) >< yss = map (xs **) yss
                \/ delay (xss >< yss)
  where xs ** ys = [(x,y) | x <- xs, y <- ys]
```

Note the use of `delay`. As we peel-off one tier in the pattern match to extract `xss`, we need to delay the enumeration of the second argument of `\/`.

The function `delay` is defined by:

```
delay :: [[a]] -> [[a]]
delay = ([]:)
```

It increases the size assigned to elements in a tier enumeration by prepending an empty list. So:

```
delay [[x,y],[a,b]] = [[],[x,y],[a,b]]
```

Constructing tiers Now, we redefine the `cons<N>` family of functions to return tier-lists.

Arity 0 The function `cons0` simply wraps the value in a tier-list with a single value of size 0:

```
cons0 :: a -> [[a]]
cons0 x = [[x]]
```

Arity 1 The function `cons1` maps a given constructor into a tiered enumeration, delaying it once.

```
cons1 :: Listable a => (a -> b) -> [[b]]
cons1 f = delay (mapT f tiers)
```

The function `mapT`, a variant of `map` for tier-lists, is defined as follows.

```
mapT :: (a -> b) -> [[a]] -> [[b]]
mapT = map . map
```

Further arities For `cons2` and others, it is just a matter of uncurrying, mapping and delaying:

```
cons2 :: (Listable a, Listable b)
      => (a -> b -> c) -> [[c]]
cons2 f = delay (mapT (uncurry f) tiers)

cons3 :: (Listable a, Listable b, Listable c)
      => (a -> b -> c -> d) -> [[d]]
cons3 f = delay (mapT (uncurry3 f) tiers)
  where
  uncurry3 f (x,y,z) = f x y z
```

Note `cons<2>`, …, `cons<N>` need matching `Listable` tuple instances, defined in the next section.

5.3 `Listable` instances using `tiers`

Listable algebraic data types (revisited) Apart from the renaming `list` to `tiers`, all instances defined in §3 that use the sum-of-`cons<N>` pattern are unchanged. For example:

```
instance (Listable a) => Listable [a] where
  tiers = cons0 []
        \/ cons2 (:)
```

Listable tuples (revisited) We define `tiers` of pairs using a product of tiers of element values:

```
instance (Listable a, Listable b)
      => Listable (a,b) where
  tiers = tiers >< tiers
```

`tiers` of triples are defined by:

```
instance (Listable a, Listable b, Listable c)
      => Listable (a,b,c) where
  tiers = mapT (\(x,(y,z)) -> (x,y,z)) tiers
```

Listable integers (revisited) We could define `Listable Int` as:

```
instance Listable Int where
  tiers = [[0]] ++ [ [n,-n] | n <- [1..] ]
```

In this definition, the size of an integer is its absolute value. However, from a practial point of view, as we use `Int`s inside other structures, this would make the enumeration "blow-up" faster (Table 1). Having one `Int` per tier works better in practice, even though the notion of size becomes less intuitive here: `0` has size `0`, `1` has size `1`, `-1` has size `2`, `2` has size `3`, and so on, alternating between positives and negatives.

Table 1. Numbers of values in each tier for two alternative `Listable Int` instances. When using the absolute value as size (1), the enumeration of compound types containing `Int`s "blows-up" faster than with one-integer-per-tier (2).

(Enum.) – Type	Numbers of values for tier of size								
	0	1	2	3	4	5	6	7	8
(1) – `Int`	1	2	2	2	2	2	2	2	2
(2) – `Int`	1	1	1	1	1	1	1	1	1
(1) – `(Int,Int)`	1	4	8	12	16	20	24	28	32
(2) – `(Int,Int)`	1	2	3	4	5	6	7	8	9
(1) – `[Int]`	1	1	3	7	17	41	99	239	577
(2) – `[Int]`	1	1	2	4	8	16	32	64	128
(1) – `Expr`	0	1	2	3	6	12	26	59	138
(2) – `Expr`	0	1	1	2	3	6	11	23	47

```
instance Listable Int where
  tiers = map (:[]) $ [0,-1..] `interleave` [1..]

> tiers :: [[Int]]
[[0],[1],[-1],[2],[-2],[3],...]
```

Convenience In our new `Listable` typeclass definition, the value `tiers` can exist alongside `list` with default definitions of each in terms of the other:

```
class Listable a where
  tiers :: [[a]]
  list  :: [a]
  tiers = map (:[]) list
  list  = concat tiers
```

So the user can define any `Listable` instance in the manner most convenient for their use-case. For types where a notion of tiers is not useful, defining only `list` provides the same enumeration as the earlier versions of our tool. We can redefine a `Listable` instance for `Int` as follows:

```
instance Listable Int where
  list = [0,-1..] `interleave` [1..]
```

5.4 `Testable` typeclass: tiers of tests

Recall our result type that represents a test result by a list of arguments and a boolean test result for those arguments:

```
type Result = ([String],Bool)
```

We redefine our `Testable` typeclass with one function, `resultiers`. Given a `Testable` property, it returns tiers of results.

```
class Testable a where
  resultiers :: a -> [[Result]]
```

The simpler `results` list can be obtained by concatenating `resultiers`:

```
results :: Testable a => a -> [Result]
results = concat . resultiers
```

We can now redefine our first `Testable` instance — the type-level base case `Bool`. Again, a boolean value is a property with no arguments, where the only test result is its value.

```
instance Testable Bool where
  resultiers p = [[([],p)]]
```

The type-level recursive case, as before, is a functional type with a `Testable` result:

```
instance (Show a, Listable a, Testable b)
      => Testable (a -> b) where
  resultiers p = concatMapT resultiersFor tiers
    where
    resultiersFor x =
      (\(as,x) -> (show x:as,r))
      `mapT`
      resultiers (p x)
```

The type-level recursion is the same as in §4. The difference is that we operate on tier-lists instead of value-lists. The `concatMapT` function is the tiered equivalent of `concatMap`:

```
concatT :: [[ [[a]] ]] -> [[a]]
concatT = foldr (\+:/) [] . map (foldr (\/) [])
  where xss \+:/ yss = xss \/ delay yss

concatMapT :: (a -> [[b]]) -> [[a]] -> [[b]]
concatMapT f = concatT . mapT f
```

5.5 Finding counter-examples and reporting test results

The functions `counterExamples` and `check` from §4 do not need to be redefined. It is enough that we have redefined `results`.

Let us now revisit the examples from the start of this section.

Example 5.1 (revisited). When running `check prop_rotRotId` we now get one of the simplest counter-examples:

```
*** failed for: (Val 0) (Val 0) (Val 1)
```

Example 5.2 (revisited). Running `check prop_elemMerge` (with the default number of tests of 200), we get:

```
*** failed for: 0 [1,1,0] []
```

Before, we had to configure 10000 tests to get a counter-example.

6 Mark V: Conditional Properties and Data Invariants

Often we do not expect a property to hold in *all* cases, but only those which meet some precondition. For example, for non-negative values of `x`:

```
\x -> x == abs x
```

We can express such constraints either by embedding a precondition into a property itself or by applying an invariant condition in a generator.

Conditional properties We can define the logical implication operator as a normal Haskell function and use it to reformulate the `abs` property:

```
(==>) :: Bool -> Bool -> Bool
False ==> _ = True
True  ==> p = p
infixr 0 ==>

\x -> x >= 0 ==> x == abs x
```

This approach has the advantage of not needing any changes in the property testing tool, but has the significant downside that there is no reduction in the number of cases checked. As our `Listable` instance for `Int` alternates positive and negative values, only half of those values make it past the precondition. The property-testing tool does not care *how* the property passes or fails, only what the result is. A property passing because the precondition failed is considered just as interesting a test as a property passing because the actual condition of interest held.

Data invariants Often we do not want to check a property for every possible value, but just pushing the precondition into the property leaves much to be desired. We can address this problem by instead restricting the generated values: the same number of test cases will be tried, but now they will *all* meet the precondition.

We can redefine the standard function `filter` to work over tiers:

```
filterT :: (a -> Bool) -> [[a]] -> [[a]]
filterT = map . filter
```

and a convenient flipped version

```
suchThat :: [[a]] -> (a -> Bool) -> [[a]]
suchThat = flip filterT
```

that can be used when defining `Listable` instances of types that follow a data invariant, for example:

```
newtype NonNeg n = NonNeg n

instance (Listable n, Num n, Ord n)
      => Listable (NonNeg n) where
  tiers = cons1 NonNeg `suchThat` nonNegOk
    where
    nonNegOk (NonNeg n) = n >= 0
```

So,

```
> tiers :: [[NonNeg Int]]
[ [], [NonNeg 0], [NonNeg 1], [], [NonNeg 2], ... ]
```

The function `suchThat` generalizes nicely over types with several constructors, allowing different invariants for each (or none):

```
  tiers = cons<N> <Cons1> `suchThat` <someCondition>
       \/ cons<N> <Cons2>
       \/ cons<N> <Cons3> `suchThat` <someCondition>
       \/ cons<N> <Cons4>
```

We can use the `NonNeg` type in the definition of the `abs` property to ensure that only non-negative values are checked:

```
\(NonNeg x) -> x == abs x
```

7 Mark VI: Functions as Test Values

Functional programs often use higher-order functions like `map` and `filter`. As they take functional arguments, properties about them require functional arguments. In this section we explore one approach of testing properties with functions as test values.

Example 7.1. The following is an (incorrect) property from [2] stating that `map` and `filter` commute:

```
prop_mapFilter ::
  Eq a => (a -> a) -> (a -> Bool) -> [a] -> Bool
prop_mapFilter f p xs =
  map f (filter p xs) == filter p (map f xs)        □
```

To test it, we need to define a `Listable` instance for functions (`a -> b`). However, a complete enumeration of functions over recursive types is known to be impossible. Even for primitive recursive functions, an enumeration is not feasible in practice [10].

Mutating functions We can enumerate a useful though limited class of functions by starting with constant-valued functions, then adding exceptions. Each single-case mutation of a function is defined by an exception pair. The `mutate` function mutates a function given a list of exception pairs:

```
mutate :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mutate f ms = foldr mut f ms
  where
  mut (x',fx') f x = if x == x' then fx' else f x
```

We used `mutate` in our previous work on the FitSpec tool [1], but there mutation is applied to given functions under test, not to constant-valued ones.

Enumerating exceptions The `exceptionPairs` function takes tiers of argument and result values, and gives tiers of lists of ordered pairs of argument and result values.

```
exceptionPairs :: [[a]] -> [[b]] -> [[ [(a,b)] ]]
```

For example:

```
> exceptionPairs (tiers :: [[Word]]) (tiers :: [[Word]])
[ [[]]
, [[(0,0)]]
, [[(0,1)],[(1,0)]]
, [[(0,2)],[(1,1)],[(0,0),(1,0)],[(2,0)]]
, ... ]
```

Here is how we define `exceptionPairs`:

```
exceptionPairs xss yss  =
  concatMapT (`excep` yss) (properSubsetsOf xss)
  where
  excep :: [a] -> [[b]] -> [[ [(a,b)] ]]
  excep xs sbs  =  zip xs
           `mapT` products (const sbs `map` xs)
```

Note the application of the function `properSubsetsOf`. It returns tiers of proper sublists of values from a given tier-list. In this way, we avoid most but not all repetition.

Enumerating functions Now, using `mutate` and `exceptionPairs`, we are ready to enumerate tiers of functions. The combining operator (`-->>`) takes tiers of argument values and tiers of result values and gives tiers of functions.

```
(-->>) :: Eq a => [[a]] -> [[b]] -> [[a -> b]]
xss -->> yss =
  concatMapT
    (\(r,yss) -> mapT (const r `mutate`)
                      (exceptionPairs xss yss))
    (choices yss)
```

The function `choices :: [[a]] -> [[(a,[[a]])]]` returns `tiers` of choices for result values. Each choice is a pair of an element taken from the argument tiers and a copy of the argument tiers without that element. Its definition is omitted here.

So, our `Listable (a -> b)` instance is just:

```
instance (Eq a, Listable a, Listable b)
      => Listable (a -> b) where
  tiers = tiers -->> tiers
```

As with the `Testable (a -> b)` instance in §4, the above definition suffices for function types of any arity: `Listable (a->b->c)`, `Listable (a->b->c->d)` and so on. Arguments should be instances of both `Eq` and `Listable`. Results should be instances of `Listable`.

Example 7.2. These are the enumerated functions of type `Bool -> Bool`:

```
tiers :: [Bool -> Bool]  =
  [ [ const False
    , const True ]
  , [ const False `mutate` [(False,True)]
    , const False `mutate` [(True,True)]
    , const True  `mutate` [(False,False)]
    , const True  `mutate` [(True,False)] ] ]
```

This enumeration includes two repeated functions:

```
const True  `mutate` [(False,False)]
const True  `mutate` [(True,False)]
```

which are equivalent to, respectively:

```
const False `mutate` [(True,True)]
const False `mutate` [(False,True)]
```

That is more apparent if we show each function extensionally:

```
tiers :: [Bool -> Bool]  =
  [ [ \x -> case x of  False -> False; True -> False
    , \x -> case x of  False -> True;  True -> True ]
  , [ \x -> case x of  False -> True;  True -> False
    , \x -> case x of  False -> False; True -> True
    , \x -> case x of  False -> False; True -> True
    , \x -> case x of  False -> True;  True -> False ]
  ]
```

and indeed, we can define a `Show` instance for functional types that shows functions in a similar extensional form by enumerating arguments and recording results. We omit details here. □

Example 7.1 (revisited). We can now use LeanCheck to get a counter-example to `prop_mapFilter` for boolean element values:

```
> check (prop_mapFilter
  :: (Bool->Bool) -> (Bool->Bool) -> [Bool] -> Bool)
*** failed for:
\x -> case x of  False -> False;  True -> False
\x -> case x of  False -> True;   True -> False
[True]                                          □
```

Example 7.3. Note the application of `mutate` at different functional levels when enumerating functions of type `Word->Word->Word`:

```
  [ [ const (const 0) ]
  , [ const (const 1) ]
  , [ const (const 0) `mutate` [(0,const 1)]
    , const (const 1) `mutate` [(1,const 0)]
    , const (const 0 `mutate` [(0,1)])
    , const (const 1 `mutate` [(0,0)])
    , const (const 2) ]
  , ... ]                                                    □
```

Example 7.4. Functions taking or returning algebraic datatypes can also be enumerated. Consider the following incorrect property over the `Expr` type that states that all binary operators on expressions are commutative:

```
prop_alwaysCommute :: (Expr -> Expr -> Expr)
                   -> Expr -> Expr -> Expr
prop_alwaysCommute f e1 e2  =  e1 `f` e2 == e2 `f` e1
```

Using check, we can see that `prop_alwaysCommute` is incorrect:

```
> check prop_alwaysCommute
*** Failure:
\x y -> case (x,y) of
        (Val 0,Val 0) -> Val 1
        (Val 0,Val 1) -> Val 1
        (Val 1,Val 0) -> Val 0
        (Val 0,Val (-1)) -> Val 1
        ...
(Val 0)
(Val 1)                                                      □
```

Bacause of the `Eq` restriction in the functional enumeration from §7, the described technique is not able to enumerate *higher-order* functions.

The class of enumerated functions is limited to only mutations of a constant function. Take for example a simple function like `even :: Int -> Bool`. LeanCheck is not able to enumerate it, only approximations, like:

```
const False `mutate` [(0,True), (2,True), (4,True)]
```

So, we conjecture this technique won't be able to find counter-examples to some properties of higher-order functions.

Table 2 shows the numbers of functions in successive tiers for several types. The number grows by a factor of less than three.

— ◇ —

Although there is scope for further development Mark VI is the last version we present here.

8 LeanCheck

The library described in this paper is a tutorial reconstruction of a property-based testing library called LeanCheck. There are a few differences, notably:

- support for existential properties (§8.1);
- support for automatic `Listable` instance derivation (§8.2);
- an improved functional enumeration (§8.3).

Towards the end of this section, we list a few limitations of LeanCheck (§8.4).

Table 2. Numbers of functions in successive tiers for several types

Tier	Number of mutants of type					
	`::Int ->Bool`	`::Bool ->Bool ->Bool`	`::Int ->Int`	`::Int ->Int ->Int`	`::[Int] ->[Int]`	`::Expr ->Expr`
0	2	2	1	1	1	0
1	2	8	1	1	1	1
2	2	32	3	5	4	1
3	4	24	5	13	10	2
4	4	–	10	35	29	3
5	6	–	16	81	75	8
6	8	–	30	201	206	17
7	10	–	48	460	539	41
8	12	–	80	1063	1428	91
9	16	–	129	2374	3721	213

8.1 Existential properties

It can be useful to ensure that a property is true on *at least* one set of inputs. An example of this is `isPrefixOf` from `Data.List`. For this use-case we can define the function `exists`, which checks whether there *exists* an assignment of values that satisfies a property up to a number of test values:

```
exists :: Testable a => Int -> a -> Bool
exists n = or . take n . map snd . results
```

For example, we can declare and check the following property:

```
> check $ \xs ys -> xs `isPrefixOf` ys
                 == exists 100 (\xs' -> xs++xs' == ys)
+++ OK
```

This is a simplistic solution, as it depends on a delicate balance between the number of tests performed by `check` and `exists`. Automatically adjusting the number of tests performed by `exists` based on the tests of `check` is future work.

8.2 Automatic derivation of `Listable` instances

Except when values have to follow a data invariant (§6), `Listable` instances follow a very simple pattern. Their production can be automated using Template Haskell [17]. Using these techniques, we could derive an instance of the `Listable` typeclass for our `Expr` type from earlier, with the following top-level declaration:

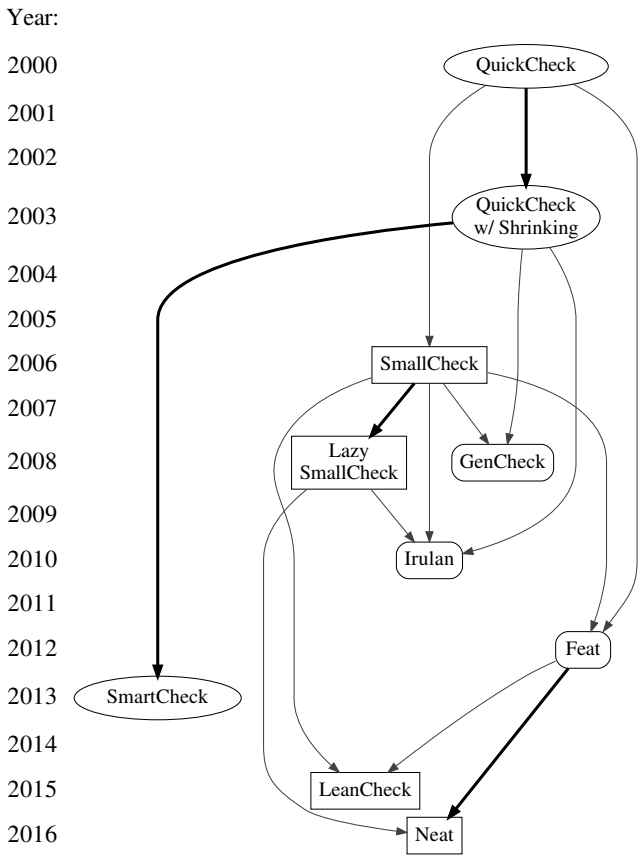
```
deriveListable ''Expr
```

Because the definition of `deriveListable` is straightforward, albeit lengthy, we omit it here. It is provided as part of the LeanCheck package (§10).

Although we chose Template Haskell, we believe the automatic derivation could alternatively be implemented using GHC's Generics [11] or Uniplate [12].

8.3 An improved function enumeration

The actual implementation of function enumeration provided on the LeanCheck package is able to avoid the repeated mutants mentioned on Example 7.2 at the cost of more complex programming. A different process is used when when dealing with finite argument types.



Legend: A thin grey arrow from A to B loosely indicates B uses ideas of and/or is inspired by A. Thick black arrows indicate direct improvements. Round, square and rounded nodes indicate random, enumerative and mixed test data geration.

Figure 1. Flow of ideas between property-testing tools for Haskell

8.4 Limitations of LeanCheck

LeanCheck does not support randomised testing or test-case pruning using laziness. Those issues are resolved by two more elaborate tools, Feat and Neat. We discuss these tools and how they address these issues in §9.

8.5 Availability

LeanCheck is freely available with a BSD3-style license from either:

- https://hackage.haskell.org/package/leancheck
- https://github.com/rudymatela/leancheck

Note however, LeanCheck is not as small as the version described in this paper: LeanCheck has been "realworldified" with several features for practical use. The version on this paper has a little less than 100 lines of code. LeanCheck still has a small core, with 200 LOC. But, accounting all modules, LeanCheck has 1500 LOC in total!

9 Related Work

QuickCheck QuickCheck [3] established the central ideas of classes of test-value types and a class of testable propeties. Unlike our `Listable` class, QuickCheck's `Arbitrary` class is used to generate random (or arbitrary!) values. As QuickCheck is nondeterministic, multiple executions are not guaranteed to find the same failing cases.

Writing `Arbitrary` instances can be a bit harder than `Listable` instances. The following is an arbitrary instance for `Expr` (§3):

```
instance Arbitrary Expr where
  arbitrary = sized arb
    where
    arb 0 = liftM Val arbitrary
    arb n = oneof [ arb 0
                  , liftM2 Add (arb (n `div` 2))
                               (arb (n `div` 2)) ]
  shrink (Val n)     = [Val n' | n' <- shrink n]
  shrink (Add e1 e2) = [e1,e2]
                    ++ [ Add e1' e2'
                       | (e1',e2') <- shrink (e1,e2) ]
```

The optional `shrink` function above is used by QuickCheck to find local-minimal counterexamples. Given a value, `shrink` produces a finite list of values which are, in some sense, like the original but a little smaller.

SmartCheck SmartCheck [14] is an extension to QuickCheck that incorporates techniques to automatically shrink and generalise counterexamples. For example:

```
> smartCheck scStdArgs $ \xs -> nub xs == (xs::[Int])
*** Extrapolated value, for all values xs:  (2:2:xs)
```

SmallCheck SmallCheck [16] was the first enumerative property-based testing tool for Haskell. Values are enumerated by depth instead of size and for this reason, the number of values tends to grow quickly as SmallCheck explores further (cf. Table 3). SmallCheck defines the `Serial` typeclass, which has the same combinators as the `Listable` typeclass in this paper:

```
instance Serial Expr where
  series = cons1 Val
         \/ cons2 Add
```

Lazy SmallCheck Lazy SmallCheck [15, 16] works similarly to SmallCheck being enumerative and depth bounded. However, Lazy SmallCheck exploits laziness, by using partially defined test values. If a property returns a boolean result for a partially defined value, Lazy SmallCheck does not enumerate versions of this value that are more defined.

Example 5.2 (revisited). For the faulty `merge` function from §5, Lazy SmallCheck reports the following counter example:

```
0  (-2:-1:0:_)  []
```

The tail of the list does not even need to be defined for the property to fail. □

Feat Feat [6] is another property-based testing tool that is able to do exhaustive enumeration of values, random testing and even a mixed strategy. To do that, Feat uses an efficient indexing function `index :: Int -> a`, that maps an integer to a value in the enumeration. More specifically, Feat partitions the set of values by their *size* to obtain a function `select :: Int -> Int -> a`, that maps the *size* and index of values of that *size* into a value. The size of a

Table 3. Numbers of integer lists in succesive tiers (for Feat and LeanCheck) or depths (for SmallCheck)

SmallCheck's depth	1, 2, 7, 36, 253, 2278, 25059, 325768, ...
Feat's size	0, 1, 0, 0, 2, 2, 4, 12, 24, 52, 120, 264, ...
LeanCheck's tiers	1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ...

value is an *arbitrary* measure depending on the type and defined in the `Enumerable` implementation but, like LeanCheck, usually counts the number of constructors plus the size of sub-values.

Feat works for types that are instances of the `Enumerable` type class: these have an `enumerate` function. Instances are defined in a similar way to instances of the `Listable` typeclass, though with somewhat higher level combinators.

```
instance Enumerable Expr where
  enumerate = consts [ pure Val
                     , unary (funcurry Add) ]
```

As the size increases, the number of generated values grows much more slowly than it does with increasing depth in SmallCheck. This gives a much more fine grained control on the number of generated test cases. On wider types, this offers a significant advantage. Table 3 shows the progression in the number of enumerated values using both methods for integer lists.

As the Feat authors note *"for completeness, Feat should support enumerating functional values. ... This is largely a question of finding a suitable definition of size for functions, or an efficient bijection from an algebraic type into the function type."* Perhaps something similar to the enumeration of functions defined in §7 would also be suitable in Feat.

Neat Neat [7] mixes the idea of size-bounded enumeration of Feat with a clever algorithm for search space pruning.

GenCheck GenCheck, like LeanCheck and Feat, enumerates values by size. It separates generators from what it calls "test strategies". No paper has been published on it. Its source code and tutorials are avaliable on Hackage and GitHub [8, 18].

Beyond property-based testing Properties can be more than just test cases, they can also serve as concise documentation. QuickSpec [5] is a tool which can discover properties of collections of functions through generating and testing terms. Discovered properties can be included in a testsuite and used as documentation, and the inclusion of an unexpected property (or lack of an expected one) can often lead to additional insights into the original code.

Summary Figure 1 summarizes how property-based testing tools for Haskell are related regarding how ideas were propagated between tools. Table 4 shows whether different features are present in each tool.

10 Conclusion

In this paper we detailed how to write a property-based testing tool by implementing the three necessary basic components:

- a typeclass for test values with a generator (§2);
- a combinator library for defining generators (§3);
- a typeclass for testable properties (§4).

In addition, we covered aspects of defining fair generators (§5), conditional properties (§6) and enumeration of functions (§7). As the main example, we used an enumerative library, but all three basic components exist in libraries that generate test values randomly.

In §8, we listed the differences to the reconstruction on this paper with the actual tool it is based on: LeanCheck.

Towards the end of the paper (§9), we summarized a few of property-based testing libraries for Haskell.

Acknowledgements

We thank Trevor Elliot, Jamey Sharp, and Ben Davis for their comments on earlier drafts.

Rudy Braquehais is supported by CAPES, Ministry of Education of Brazil (Grant BEX 9980-13-0).

References

[1] Rudy Braquehais and Colin Runciman. 2016. FitSpec: refining property sets for functional testing. In *Haskell'16*. ACM, 1–12.
[2] Koen Claessen. 2012. Shrinking and Showing Functions. In *Haskell'12*. ACM, 73–80.
[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00*. ACM, 268–279.
[4] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. 2003. Testing and Tracing Lazy Functional Programs Using QuickCheck and Hat. In *AFP'03*. LNCS, Vol. 2638. Springer, 59–99.
[5] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010*. LNCS, Vol. 6143. Springer, 6–21.
[6] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Haskell'12*. ACM, 61–72.
[7] Jonas Duregård. 2016. *Automating Black-Box Property Based Testing*. Ph.D. Dissertation. Chalmers University of Technology.
[8] Gordon J. Uszkay and Jacques Carette. 2012. GenCheck Tutorial. https://github.com/JacquesCarette/GenCheck/blob/master/tutorial/GenCheck_Tutorial.pdf. (2012).
[9] Graham Hutton. 1998. Fold and Unfold for Program Semantics. In *ICFP'98*. ACM, 280–288.
[10] Stefan Kahrs. 2006. The Primitive Recursive Functions are Recursively Enumerable. (2006). http://www.cs.kent.ac.uk/people/staff/smk/primrec.pdf
[11] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Haskell'10*. ACM, 37–48.
[12] Neil Mitchell and Colin Runciman. 2007. Uniform Boilerplate and List Processing. In *Haskell'07*. ACM, 49–60.
[13] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. 2008. *Real World Haskell*. O'Reilly.
[14] Lee Pike. 2014. SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization. In *Haskell'14*. ACM, 59–70.
[15] Jason S. Reich, Matthew Naylor, and Colin Runciman. 2013. Advances in Lazy SmallCheck. In *IFL'13*. LNCS, Vol. 8241. Springer, 53–70.
[16] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell'08*. ACM, 37–48.
[17] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Haskell'02*. ACM, 1–16.
[18] Gordon J. Uszkay and Jacques Carette. 2012. GenCheck. https://github.com/JacquesCarette/GenCheck. (2012).
[19] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (Dec. 1997), 366–427.

Table 4. Summary of differences between property-based testing tools for Haskell.

	QuickCheck	SmartCheck	SmallCheck	Lazy SmallCheck	**LeanCheck**	Feat	Neat	GenCheck	Irulan	Reach
Test data generation										
random	●	●	○	○	○	●	○	●	●	○
enumerative	○	○	●	●	●	●	●	●	●	○
depth-bounded	○	○	●	●	○	○	○	○	●	○
size-bounded	○	○	○	○	●	●	●	●	○	○
mixed random & enumerative	○	○	○	○	○	●	○	●	●	○
demand-driven / directed	○	○	○	●	○	○	●	○	●	●
automatic generator instance derivation	●	●	●	●	●	●	??	??	●	–
Existential properties	○	○	●	●	●	○^{p}	○^{p}	○^{p}	○	○
Higher order properties	●	●	●	●	●	○^{p}	○^{p}	○^{p}	○	○
Generalized counterexamples	○	●	○	◐	○	○	○	○	○	○
Ease of use/writing generators	○	○	●	●	●	◐	◐	◐	–	–

Legend: ● Yes/Good. ○ No/Poor. ◐ Partial/Median. ^{p} Potential support