QuickCheck and Magic of Testing

By Alexey Kuleshevich, January 24, 2017

Share this

Haskell is an amazing language. With its extremely powerful type system and a pure functional paradigm it prevents programmers from introducing many kinds of bugs, that are notorious in other languages. Despite those powers, code is still written by humans, and bugs ar inevitable, so writing quality test suites is just as important as writing an application itself.

Over the course of history buggy software has cost industry billions of dollars in damage and even lost human lives, so I cannot stress enough, how essential testing is for any project.

One of the ways to test software is through writing unit tests, but since it is not feasible to test all possible inputs exhaustively for most functions, we usually check some corner cases and occasionally test with other arbitrary values. Systematic generation of random input, that is biased towards corner cases, could be very helpful in that scenario, and that's where QuickCheck comes into play. This state of th art property testing library was originally invented in Haskell, and, because it turned out to be so powerful, it was later ported to other languages. However, the real power of random testing is unleashed when combined with purity of Haskell.

Properties

Let's start by looking at this exemplar properties of a `reverse` function:

```
reverse (reverse xs) == xs

reverse (xs ++ ys) == reverse ys ++ reverse xs
```

We know, that they will hold for all finite lists with total values. Naturally, there are ways to prove them manually and there are even tools for Haskell, such as LiquidHaskell, that can help you automate proving some properties. Formal proof of correctness of a program is not always possible: some properties are either too hard or impossible to prove. Regardless of ability to prove a property of a function, we at least need to check that it works correctly on some finite set of inputs.

```
import Test.QuickCheck

prop_RevRev :: Eq a => [a] -> Bool
prop_RevRev xs = reverse (reverse xs) == xs

prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

We [illegible] load those properties into GHCi and run `quickCheck` on them. Here is a quick way on how to do it from a terminal, and a detaile [illegible] how to get started with `stack`.

```
$ stack --resolver lts-7.16 ghci --package QuickCheck
Configuring GHCi with the following packages:
GHCi, version 8.0.1: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /tmp/ghci3260/ghci-script
Prelude> :load examples.hs
[1 of 1] Compiling Main             ( examples.hs, interpreted )
Ok, modules loaded: Main.
*Main> quickCheck prop_RevRev
+++ OK, passed 100 tests.
*Main> quickCheck prop_RevApp
+++ OK, passed 100 tests.
*Main>
```

What just happened? QuickCheck called `prop_RevRev` and `prop_RevApp` 100 times each, with random lists as arguments and declare those tests as passing, because all calls resulted in `True`. Far beyond what a common unit test could have done.

Worth noting, that in reality, not only `prop_RevRev`, but both of those properties are polymorphic and `quickCheck` will be happy to wo with such functions, even if type signatures were inferred, and it will run just fine in GHCi. On the other hand, while writing a test suite, we have to restrict the type signature for every property to concrete type, such as `[Int]` or `Char`, otherwise type checker will get confused For example, this program will not compile:

```
import Test.QuickCheck

main :: IO ()
main = quickCheck (const True)
```

For the sake of example let's write couple more self-explanatory properties:

```
prop_PrefixSuffix :: [Int] -> Int -> Bool
prop_PrefixSuffix xs n = isPrefixOf prefix xs &&
                         isSuffixOf (reverse prefix) (reverse xs)
  where prefix = take n xs

prop_Sqrt :: Double -> Bool
prop_Sqrt x
  | x < 0            = isNaN sqrtX
  | x == 0 || x == 1 = sqrtX == x
  | x < 1            = sqrtX > x
  | x > 1            = sqrtX > 0 && sqrtX < x
  where
    sqrtX = sqrt x
```

Now, this is great, but how did we just pass various functions with different number of arguments of different types to `quickCheck`, and how did it know what to do with them? Let's look at it's type signature:

```
λ> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

Testable

So, it seems, that QuickCheck can test anything that is `Testable`:

```
λ> :i Testable
class Testable prop where
  property :: prop -> Property
  exhaustive :: prop -> Bool
instance [safe] Testable Property
instance [safe] Testable prop => Testable (Gen prop)
instance [safe] Testable Discard
instance [safe] Testable Bool
instance [safe] (Arbitrary a, Show a, Testable prop) => Testable (a -> prop)
```

The last instance is for a function `(a -> prop)`, that returns a `prop`, which, in turn, must also be an instance of `Testable`. This magic trick of a recursive constraint for an instance definition allows `quickCheck` to test a function with any number of arguments, as long as each one of them is an instance of `Arbitrary` and `Show`. So here is a check list of requirements for writing a testable property:

- Zero or more arguments, which have an instance of `Arbitrary`, that is used for generating random input. More on that later.
- Arguments must also be an instance of `Show`, so if a test fails, offending value can be displayed back to a programmer.
- Return value is either:
  - `True`/`False` - to indicate pass/fail of a test case.
  - `Discard` - to skip the test case (eg. precondition fails).
  - `Result` - to customize pass/fail/discard test result behavior, collect extra information about the test outcome, provide callbacks and other advanced features.
  - `Property` for a much finer control of test logic. Such properties can be used as combinators to construct more complex test cases.
  - `Prop` used to implement `Property`
- Start with `prop_` or `prop`, followed by the usual `camelCase`, but that is just a convention, not a requirement.
- Has no side effects. Also not a requirement, but strongly suggested, since referential transparency is lost with `IO` and test results c be inconsistent between runs. At the same time there are capabilities for testing Monadic code, which we will not go into here.

Preconditions

Here is another very simple property of lists `xs !! n == head (drop n xs)`, so let's define it as is:

```
prop_Index_v1 :: [Integer] -> Int -> Bool
prop_Index_v1 xs n = xs !! n == head (drop n xs)
```

Naturally, you can see a problem with that function, it cannot accept just any random `Int` to be used for indexing, and `quickCheck` quickly finds that problem for us and prints out violating input along with an error:

```
λ> quickCheck prop_Index_v1
*** Failed! Exception: 'Prelude.!!: index too large' (after 1 test):
[]
0
```

Interestingly, if you try to run this example on any computer, there is a very good chance that it will give exactly the same output, so, it seems that input to properties is not completely random. In fact, thanks to the function `sized`, the first input to our property will always an empty list and an integer `0`, which tend to be really good corner cases to test for. In our case, though, `!!` and `head` are undefined for empty lists and negative numbers. We could add some guards, but there are facilities provided for such common cases:

```
prop_Index_v2 :: (NonEmptyList Integer) -> NonNegative Int -> Bool
prop_Index_v2 (NonEmpty xs) (NonNegative n) = xs !! n == head (drop n xs)
```

This version is still not quite right, since we do have another precondition `n < length xs`. However, it would be a bit complicated to describe this relation through the type system, so we will specify this precondition at a runtime using implication operator (⇒). Note, tha return type has changed too:

```
prop_Index_v3 :: (NonEmptyList Integer) -> NonNegative Int -> Property
prop_Index_v3 (NonEmpty xs) (NonNegative n) =
  n < length xs ==> xs !! n == head (drop n xs)
```

Test cases with values, that do not satisfy the precondition, will simply get discarded, but not to worry, it will still generate the 100 tests. I fact it will generate up to a 1000 before giving up. An alternate way to achieve similar effect would be to generate a valid index within a property itself:

```
prop_Index_v4 :: (NonEmptyList Integer) -> Property
prop_Index_v4 (NonEmpty xs) =
  forAll (choose (0, length xs-1)) $ \n -> xs !! n == head (drop n xs)
```

```
λ> quickCheck prop_Index_v3 >> quickCheck prop_Index_v4
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Just in case, let's quickly dissect this for all (∀) business. It takes a random value generator, which `choose` happens to produce, a proper that operates on it's values and returns a property, i.e. applies values from a specific generator to the supplied property.

```
λ> :t forAll
forAll :: (Show a, Testable prop) => Gen a -> (a -> prop) -> Property
λ> sample' $ choose (0, 3)
[0,2,2,3,3,3,0,1,0,1,3]
```

There is a very subtle difference between the last two versions, namely `_v3` will discard tests that do not satisfy a precondition, while `_v` will always generate a value for `n` that is safe for passing to index function. This is not important for this example, which is good, but that not always the case. Whenever precondition is too strict, QuickCheck might give up early while looking for valid values for a test, but mor importantly, it can give a false sence of validity, since most of the values that it will find could be trivial ones.

Pitfalls

For this section we will use prime numbers in our examples, but rather than reinventing the wheel and writing functions for prime numbe ourselves we will use primes package. Just for fun, let's write a property for `primeFactors`, which is based on Fundamental Theorem of Arithmetic:

```
prop_PrimeFactors :: (Positive Int) -> Bool
prop_PrimeFactors (Positive n) = isPrime n || all isPrime (primeFactors n)
```

That was incredibly easy and is almost a direct translation of a theorem itself. Let's consider a fact that every prime number larger than 2 odd, thus we can easily derive a property that sum of any two prime numbers greater than 2 is even. Here is a naive way to test that property:

```
prop_PrimeSum_v1 :: Int -> Int -> Property
prop_PrimeSum_v1 p q =
  p > 2 && q > 2 && isPrime p && isPrime q ==> even (p + q)
```

As you can imagine it is not too often that a random number will be prime, this certainly will affect the quality of this test:

```
λ> quickCheck prop_PrimeSum_v1
*** Gave up! Passed only 26 tests.
```

It only found 26 satisfiable tests out of a 1000 generated, that's bad. There is even more to it, in order to convince ourselves, that we are testing functions with data that resembles what we expect in real life, we should always try to inspect the values being generated for a property. An easy way to do that is to classify them by some shared traits:

```
prop_PrimeSum_v1' :: Int -> Int -> Property
prop_PrimeSum_v1' p q =
  p > 2 && q > 2 && isPrime p && isPrime q ==>
  classify (p < 20 && q < 20) "trivial" $ even (p + q)
```

```
λ> quickCheck prop_PrimeSum_v1'
*** Gave up! Passed only 29 tests (96% trivial).
λ> quickCheckWith stdArgs { maxSuccess = 500 } prop_PrimeSum_v1'
*** Gave up! Passed only 94 tests (44% trivial).
```

Almost all values this property was tested on are in fact trivial ones. Increasing number of tests was not much of a help, because, by default, values generated for integers are pretty small. We could try to fix that with appropriate types, but this time we will also generate histogram of unique pairs of discovered prime numbers:

```
prop_PrimeSum_v2 :: (Positive (Large Int)) -> (Positive (Large Int)) -> Property
prop_PrimeSum_v2 (Positive (Large p)) (Positive (Large q)) =
  p > 2 && q > 2 && isPrime p && isPrime q ==>
  collect (if p < q then (p, q) else (q, p)) $ even (p + q)
```

```
λ> quickCheck prop_PrimeSum_v2
*** Gave up! Passed only 24 tests:
16% (3,3)
 8% (11,41)
 4% (9413,24019)
 4% (93479,129917)
 ...
```

This is better, there are less trivial values, but still, number of tests is far from satisfactory. It is also extremely inefficient to look for prime values that way, and, for any really large value passed to the property, it will take forever to check its primality. Much better approach would be to choose from a list of prime values, which we have readily available for us:

```
prop_PrimeSum_v3 :: Property
prop_PrimeSum_v3 =
  forAll (choose (1, 1000)) $ \ i ->
    forAll (choose (1, 1000)) $ \ j ->
      let (p, q) = (primes !! i, primes !! j) in
      collect (if p < q then (p, q) else (q, p)) $ even (p + q)
```

```
λ> quickCheck prop_PrimeSum_v3
+++ OK, passed 100 tests:
 1% (983,6473)
 1% (953,5059)
 1% (911,5471)
 ...
```

Arbitrary

There could be a scenario where we needed prime values for many tests, then it would be a burden to generate them this way for each property. In such cases solution is always to write an instance for `Arbitrary`:

```
newtype Prime a = Prime a deriving Show

instance (Integral a, Arbitrary a) => Arbitrary (Prime a) where
  arbitrary = do
    x <- frequency [ (10, choose (0, 1000))
                   , (5, choose (1001, 10000))
                   , (1, choose (10001, 50000))
                   ]
    return $ Prime (primes !! x)
```

Calculating large prime numbers is pretty expensive, so we could simply use something like `choose (0, 1000)`, similarly to how it was done in `prop_PrimeSum_v3`, but there is no reason why we should exclude generating large prime numbers completely, instead, we can reduce their chance by describing a custom distribution with `frequency` function.

Now writing `prop_PrimeSum` is a piece of cake:

```
prop_PrimeSum_v4 :: Prime Int -> Prime Int -> Property
prop_PrimeSum_v4 (Prime p) (Prime q) =
  p > 2 && q > 2 ==> classify (p < 1000 || q < 1000) "has small prime" $ even (p + q)
```

```
λ> quickCheck prop_PrimeSum_v4
+++ OK, passed 100 tests (21% has small prime).
```

CoArbitrary

There are quite a few instances of `Arbitrary`, many common data types from `base` are, but the most peculiar one is a function:

```
λ> :i Arbitrary
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
...
instance [safe] (CoArbitrary a, Arbitrary b) => Arbitrary (a -> b)
...
```

That's right, QuickCheck can even generate functions for us! One of restrictions is that an argument to the function is an instance of `CoArbitrary`, which also has instance for a function, consequently functions of any arity can be generated. Another caveat is that we need an instance of `Show` for functions, which is not a standard practice in Haskell, and wrapping a function in a `newtype` would be more appropriate. For clarity we will opt out from this suggestion and instead demonstrate this cool feature in action. One huge benefit is that allows us to easily write properties for higher order functions:

```
instance Show (Int -> Char) where
  show _ = "Function: (Int -> Char)"

instance Show (Char -> Maybe Double) where
  show _ = "Function: (Char -> Maybe Double)"

prop_MapMap :: (Int -> Char) -> (Char -> Maybe Double) -> [Int] -> Bool
prop_MapMap f g xs = map g (map f xs) == map (g . f) xs
```

HSpec

One of the first concerns, that programmers usually raise when coming from other languages to Haskell, is that there are situations when unit tests are invaluable, but QuickCheck does not provide an easy way to do that. Bear in mind, QuickCheck's random testing is not a limitation, but rather is a priceless feature of testing paradigm in Haskell. Regular style unit tests and other QA functionality (code coverage, continuous integration, etc.) can be done just as easily as they are done in any other modern language using specialized libraries. In fact, those libraries play beautifully together and complement each other in many ways.

Here is an example of how we can use hspec to create a test suite containing all properties we have discussed so far, plus few extra unit tests for completeness of the picture.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.16 runghc --package QuickCheck --package hspec --package primes
module Main where
import Test.Hspec
import Test.QuickCheck

...

main :: IO ()
main = hspec $ do
  describe "Reverse Properties" $
    do it "prop_RevRev" $ property prop_RevRev
       it "prop_RevApp" $ property prop_RevApp
       it "prop_PrefixSuffix" $ property prop_PrefixSuffix
  describe "Number Properties" $
    do it "prop_Sqrt" $ property prop_Sqrt
  describe "Index Properties" $
    do it "prop_Index_v3" $ property prop_Index_v3
       it "prop_Index_v4" $ property prop_Index_v4
       it "unit_negativeIndex" $ shouldThrow (return $! ([1,2,3] !! (-1))) anyException
       it "unit_emptyIndex" $ shouldThrow (return $! ([] !! 0)) anyException
       it "unit_properIndex" $ shouldBe (([1,2,3] !! 1)) 2
  describe "Prime Numbers" $
    do it "prop_PrimeFactors" $ property prop_PrimeFactors
       it "prop_PrimeSum_v3" $ property prop_PrimeSum_v3
       it "prop_PrimeSum_v4" $ property prop_PrimeSum_v4
  describe "High Order" $
    do it "prop_MapMap" $ property prop_MapMap
```

Conclusion

Random testing can be mistakenly regarded as an inferior way of software testing, but many studies have certainly shown that it is not th case. To quote D. Hamlet:

> By taking 20% more points in a random test, any advantage a partition test might have had is wiped out.

It is very easy to start using QuickCheck to test properties of pure functions. There is also a very similar toolbox included in the library fo testing monadic functions, thus allowing for a straightforward way of testing properties of functions that do mutations, depend on state, run concurrently and even perform I/O. Most importantly, this library provides yet another technique for making Haskell programs even safer.

Writing tests doesn't have to be a chore, it can be fun. We certainly find it fun at FPComplete and will be happy to provide training, consulting or development work.

Further reading

- An introduction to QuickCheck testing
- QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs
- Testing monadic code with QuickCheck
- QuickFuzz: an automatic random fuzzer for common file formats

Subscribe to our blog via email

Email subscriptions come from our Atom feed and are handled by Blogtrottr. You will only receive notifications of blog posts, and can unsubscribe any time.

Do you like this blog post and need help with DevOps, Rust or functional programming? Contact us.

Share this

Contact Us

Corporate Office
10130 Perimeter
Parkway
Suite 200
Charlotte, NC 28216

+1 858-617-0430

sales@fpcomplete.com

Services

Custom Software
Development
DevSecOps
Blockchain
Rust
Haskell
Training
All Services

Products

Kube360®
Zehut
Amber
Konsole360
Idiom
Kafka Library

Resources

Blog Posts
Video Library
Case Studies
White Papers

Our Company

Our Journey
Our Mission
Our Leadership
Our Engineers
Our Clients
Jobs

FPComplete

Privacy Policy