An introduction to property-based testing with QuickCheck

Jesper Cockx

version of February 9, 2022

Contents

1 Introduction

When you were first learning to program, at some point you were probably told about the importance of writing *unit tests*: small test cases that each test a small piece of functionality of your code. And while it is true that writing unit tests is important, it is also at the same time *boring* and *difficult*. It is boring because you need to write many separate unit tests for each piece of functionality, which all look more or less the same. And it is difficult because it is very easy to miss a certain combination of inputs for which the program crashes. Would it not be nice if we could just write down how the program should behave and have the test cases be generated automatically? That is precisely the approach of **property-based testing**.

In short, property-based testing is an approach to testing where you as the programmer write down properties that you expect to hold of your program. When running the tests, the test runner will generate a lot of different random input values, and then check that the property holds for all these (combinations of) input values. Compared to writing individual unit tests, property-based testing has several advantages:

- You spend **less time writing test code**: a single property can often replace many hand-written test cases.
- You get **better coverage**: by randomly generating inputs, QuickCheck will test lots of combinations you'd never test by hand.
- You spend **less time on diagnosis of errors**: if a property fails to hold, QuickCheck will automatically produce a minimized counterexample.

QuickCheck is a tool for property-based testing of Haskell code. Since its introduction for Haskell in 1999, QuickCheck has become very popular as a testing framework and has been ported to many other programming languages such as C, C++, Java, JavaScript, Python, Scala, and many others[1] However, QuickCheck really benefits from the fact that Haskell is a pure language, so that is where the approach continues to be the most powerful.

[1] See https://en.wikipedia.org/wiki/QuickCheck for a more complete list.

This introduction will show you the basic usage of QuickCheck for testing properties of Haskell code, as well as how to use alternative random generators. All the functions that are used come from the module `Test.QuickCheck` from the QuickCheck package. To use QuickCheck in a Haskell project that uses Stack, add the following line to the list of dependencies in `package.yaml`:

```
- QuickCheck >= 2.14
```

2 *Basic usage of QuickCheck*

To write a QuickCheck test case, all you have to do is define a Haskell function that defines a **property** of your program that you expect to hold. In the simplest case, a property is just a value of type `Bool`. For example, suppose we have written a simple Haskell function to calculate the distance between two integers:

```haskell
distance :: Int -> Int -> Int
distance x y = abs (y-x)
```

We can then express the property that the distance between `3` and `5` equals `2`:

```haskell
prop_dist35 :: Bool
prop_dist35 = distance 3 5 == 2
```

By convention, names of QuickCheck properties always start with `prop_`. We can express more general properties by defining a function that returns a `Bool`:

```haskell
-- The distance between any number and itself is always 0
prop_dist_self :: Int -> Bool
prop_dist_self x = distance x x == 0

-- The distance between x and y is equal to the distance between y and x
prop_dist_symmetric :: Int -> Int -> Bool
prop_dist_symmetric x y = distance x y == distance y x
```

When testing a property that takes one or more inputs, QuickCheck will randomly generate several inputs (100 by default) and check that the function returns `True` for all inputs.

The main function used to call QuickCheck is `quickCheck`, which is defined in the module `Test.QuickCheck`. To import it, you can either add `import Test.QuickCheck` at the top of your file or import it manually if you are working from GHCi. Assuming you have installed the QuickCheck package, you can then load the file and run tests by calling `quickCheck`:

```
> ghci
GHCi, version 8.10.2: https://www.haskell.org/ghc/  :? for help
Loaded package environment from ~/.ghc/x86_64-linux-8.10.2/environments/default
> :l QuickCheckExamples.hs
> quickCheck prop_dist35
+++ OK, passed 1 test.
```

QuickCheck tells us that everything is as it should be: it ran the test and got the result `True`. Since there are no inputs to the test, it is run only once. Let us try out some more properties!

```
> quickCheck prop_dist_self
+++ OK, passed 100 tests.
> quickCheck prop_dist_symmetric
+++ OK, passed 100 tests.
```

Huge success! For each of the tests, QuickCheck has generated 100 random inputs and verified that for each one the property returns `True`.

To get more information about the test inputs that are generated by QuickCheck, you can replace the function `quickCheck` with `verboseCheck`. This will print out each individual test case as it is generated. Try it out for yourself!

Shrinking counterexamples

What happens if there's a mistake in our code? Say we forgot to write `abs` in the definition of `distance`?

```
> quickCheck prop_dist_symmetric
*** Failed! Falsified (after 2 tests):
0
1
```

QuickCheck has found a counterexample: if the first input `x` is 0 and the second input `y` is 1, then `y-x` is not equal to `x-y`.

When QuickCheck finds a counterexample, it will not always return the first one it encounters. Instead, QuickCheck will look for the smallest counterexample it can find. As an example, let us try to run QuickCheck on the (false) property stating that every list is sorted.

```
sorted :: Ord a => [a] -> Bool
sorted (x:y:ys) = x <= y && sorted (y:ys)
sorted _        = True

-- A (false) property stating that every list is sorted
prop_sorted :: [Int] -> Bool
prop_sorted xs = sorted xs
```

```
> verboseCheck prop_sorted
Passed:
[]

Passed:
[]

Passed:
[0]

Failed:
[2,1,3]

Passed:
[]

Passed:
[1,3]

Passed:
[2,3]

Failed:
[2,1]
```

```
...

*** Failed! Falsified (after 4 tests and 3 shrinks):
[1,0]
```

The first list that is generated that is not sorted is `[2,1,3]`. Note that this will be a different list every time we run QuickCheck since it is randomly generated. However, QuickCheck does not stop there and instead tries smaller and smaller lists until it converges to a minimal counterexample: `[1,0]`. This process is called **shrinking**.

It is worth noting that despite the inherent randomness of QuickCheck, shrinking will often converge to one of a small set of minimal counterexamples. For example, if we run `quickCheck` many times on `prop_sorted`, we always end up with either `[1,0]` or `[0,-1]` as a counterexample.

The precise strategy that QuickCheck uses for shrinking counterexamples depends on the type of the counterexample:

- For numeric types such as `Int`, QuickCheck will try a random number that is smaller in absolute value (i.e. closer to 0).
- For booleans of type `Bool`, QuickCheck will try to replace `True` with `False`.
- For tuple types `(a,b)`, QuickCheck will try to shrink one of the components.
- For list types, QuickCheck will try to either delete a random element from the list, or try to shrink one of the values in the list.

Testing many properties at once

Instead of running individual tests from GHCi, you can also combine all your tests in a `main` function:

```
main = do
  quickCheck prop_dist35
  quickCheck prop_dist_self
  quickCheck prop_dist_symmetric
```

This code makes use of Haskell `do` keyword that we will study in the chapter on monads. Once you have defined this `main` function, you can invoke it by calling `runghc` from the command line:

```
> runghc QuickcheckExamples.hs
+++ OK, passed 1 test.
```

When you are writing code in the WebLab instance for this course, you do not need to write a main function for QuickCheck tests: WebLab will automatically collect all functions in the `Test` tab whose name starts with `prop_` and run `quickCheck` on each one.

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Note that a file may only contain a single `main` function. In a realistic project, we would instead create a separate file that just defines all QuickCheck properties and puts them together in a `main` function.

3 *Discovering properties to test*

The biggest challenge in making effective use of QuickCheck lies in coming up with good properties to test. So let us take a look at some examples of good properties to test.

Roundtrip properties

When one function is an inverse to another function, we can create a property test for that. For example, we can test that reversing a list is its own inverse:

```haskell
prop_reverse_reverse :: [Int] -> Bool
prop_reverse_reverse xs = reverse (reverse xs) == xs
```

As another example, we can test that inserting an element into a list and then deleting it again results in the same list:

```haskell
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys) | x <= y    = x:y:ys
                | otherwise = y:insert x ys

delete :: Int -> [Int] -> [Int]
delete x [] = []
delete x (y:ys) | x == y    = ys
                | otherwise = y:delete x ys

prop_insert_delete :: [Int] -> Int -> Bool
prop_insert_delete xs x = delete x (insert x xs) == xs
```

In general, it might take more than two functions to get back to the point where we started from. Any property of the form `f (g (... (h x))) == x` is called a **roundtrip property**.

Equivalent implementations

When we have two functions that should be functionally equivalent but have a different implementation, we can test that this is indeed the case. For example, we can test that a function `qsort :: Ord a => [a] -> [a]`, we can define a property that tests it has the same behaviour as the builtin Haskell function `sort`:

```
prop_qsort_sort :: [Int] -> Bool
prop_qsort_sort xs = qsort xs == sort xs
```

If there is an alternative implementation available, defining a property of this kind is usually a very good idea since it can catch a broad range of errors in the implementation.

Another variant of this technique can be applied when you replace the implementation of a function with a new version (perhaps because it is more efficient or more general). In that case, you can keep the old code under a different name and add a property to test that both implementation produce the same result. This way you can make sure that the behaviour of the program has not changed by accident!

Warning. When testing a polymorphic function such as `qsort`, it seems attractive to also define a polymorphic test case:

```
prop_qsort_sort' ::
  Ord a => [a] -> Bool
prop_qsort_sort' xs =
  qsort xs == sort xs
```

However, this does not work as you might expect: by default, `quickCheck` will instantiate all type parameters to the empty tuple type `()`. So the property that `quickCheck` will actually test is `prop_qsort_sort'' :: [()] -> Bool`. Since a list of empty tuples `[(),(),...,()]` is always sorted, this test will always return `True` even if the function `qsort` does nothing! So in general it is a good idea to **always give a concrete type (without type parameters) to property tests**.

Algebraic laws

When we have a function that implements a certain idea from algebra, we can use the **algebraic laws** as inspiration for property tests. For example, suppose we have a function `vAdd :: (Int,Int) -> (Int,Int) -> (Int,Int)` that defines addition on two-dimensional vectors, we can test that it is commutative, associative, and has a neutral element:

```
prop_vAdd_commutative :: (Int,Int) -> (Int,Int) -> Bool
prop_vAdd_commutative v w = vAdd v w == vAdd w v

prop_vAdd_associative :: (Int,Int) -> (Int,Int) -> (Int,Int) -> Bool
prop_vAdd_associative u v w = vAdd (vAdd u v) w == vAdd u (vAdd v w)

prop_vAdd_neutral_left :: (Int,Int) -> Bool
prop_vAdd_neutral_left u = vAdd (0,0) u == u

prop_vAdd_neutral_right :: (Int,Int) -> Bool
prop_vAdd_neutral_right u = vAdd u (0,0) == u
```

As another example, some functions such as sorting functions are *idempotent*: applying them twice produces the same result as applying them just once.

```
prop_qsort_idempotent :: [Int] -> Bool
prop_qsort_idempotent xs = qsort (qsort xs) == qsort xs
```

4 *Testing with different distributions*

Often we want to test a certain property but it does not hold for all possible inputs. For example, let us try to test that each element in the result of `replicate n x` is equal to `x` (the function `replicate :: Int -> a -> [a]` is defined in the standard prelude).

```
prop_replicate :: Int -> Int -> Int -> Bool
prop_replicate n x i = replicate n x !! i == x
```

However, when we try to test this property we get an error:

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

Oh no! QuickCheck generated a list of length 0, which means any index `i` is out of bounds. To resolve this problem, we could change the property so it always evaluates to `True` in case the index is out of bounds:

```
prop_replicate :: Int -> Int -> Int -> Bool
prop_replicate n x i = i < 0 || i >= n || replicate n x !! i == x
```

Now testing the property is successful:

```
> quickCheck prop_replicate
+++ OK, passed 100 tests.
```

However, this actually gives us a false sense of security: in the vast majority of the test cases that are generated, the index is out of bounds and thus nothing about the behaviour of `replicate` is tested.

You can verify this by running `verboseCheck` on this property and counting the cases where `i` is in between `0` and `n`. In effect much fewer than 100 inputs are tested, so there is a high chance that mistakes go undetected.

We could try to counteract this by running more test cases. However, how many test cases is enough? Do we need 1000? 10000? What about if we want to test a property where it is even rarer that we generate a valid input, for example when a property only holds for sorted lists?

Instead of running more test cases, we can make sure that the test cases we generate are always valid. We will discuss two methods provided by the QuickCheck library that allow us to do this: **conditional properties** and **quantified properties**.

Conditional properties

The first way we can restrict the inputs that QuickCheck uses is by using a conditional property of the form

```
<condition> ==> <property>
```

For example, we can rewrite `prop_replicate` as follows:

```
prop_replicate :: Int -> Int -> Int -> Property
prop_replicate n x i =
  (i >= 0 && i < n) ==> replicate n (x :: Int) !! i == x
```

Notice that the return type is no longer `Bool` but `Property`, a new type that is introduced by QuickCheck. Luckily we do not need to know much about this type in order to use QuickCheck; all we need to know is that we can use `quickCheck` to test functions that return a `Property` just like functions that return a `Bool`!

```
> quickCheck prop_replicate
+++ OK, passed 100 tests; 695 discarded.
```

What just happened? QuickCheck tests the property `replicate n (x :: Int) !! i == x` as before, but now it discards all test cases that do not satisfy the condition `i >= 0 && i < n`. In the output, we can see that it generated a total of 795 test cases, of which 695 were discarded and the remaining 100 passed successfully.

Conditional properties work well when there is a reasonable chance that a randomly generated input will satisfy the condition. However, it breaks down when valid inputs are very rare. For example, let us try to test that inserting an element into an ordered list again results in an ordered list:

```
prop_insert_sorted :: Int -> [Int] -> Property
prop_insert_sorted x xs = sorted xs ==> sorted (insert x xs)
```

If we ask QuickCheck to test this property, it gives up:

```
> quickCheck prop_insert_sorted
*** Gave up! Passed only 80 tests; 1000 discarded tests.
```

The lesson here is that we should not use conditional properties when the condition is very unlikely to be satisfied.

Quantified properties

Instead of first generating random values and then filtering out the ones that are not valid, we can instead use a different random generator that only generates valid inputs in the first place. For this purpose, QuickCheck allows us to define *quantified properties* by using custom random generators of type `Gen a`. For example, we can use the generator `orderedList` to generate a random ordered list. We can use random generators to define properties by using the function `forAll`:

```
prop_insert_sorted :: Int -> Property
prop_insert_sorted x = forAll orderedList (\xs -> sorted (insert x xs))
```

The first argument of `forAll` is the random generator. The second argument is a *function* that maps the result of the generator to the property we want to test (in other words, `forAll` is a *higher-order function*). Here we have given the function as a lambda expression `\xs -> sorted (insert x xs)`. Note that the list `xs` is no longer an input to the overall property `prop_insert_sorted`, as it is already an argument to the lambda expression. Now if we run `quickCheck` again, we see that the tests now pass:

```
> quickCheck prop_insert_sorted
+++ OK, passed 100 tests.
```

You can also use `verboseCheck` to verify that all the generated lists are indeed sorted.[2]

[2] Do it!

QuickCheck provides a long list of random generators that we can use to test our properties. Here are a few useful examples:

- The generator `choose` will choose a random elements between two bounds. For example, `choose (1,6)` will generate a random number between 1 and 6 (both bounds included).
- The generator `elements` will choose a random element from a given list of values. For example, `elements ['a','e','i','o','u']` will generate a random vowel.

- The generator `pure` will always generate the same value. For example `pure 4` will always generate the number `4` (which is guaranteed to be random[3]).

- The generator `frequency` can be used to combine several generators into one. Each generator is given a weight that determines how likely it is that it is used. For example, `frequency [(99,pure True),(1,pure False)]` will generate `True` 99% of the time and `False` 1% of the time.

- The generator `vector` generates lists of a given length. For example, `vector 42` generates random lists of length 42.

- The generator `shuffle` generates lists with the same elements as a given list but in a random order. For example, `shuffle [1..10]` generates random lists containing the numbers `1` to `10`.

[3] https://xkcd.com/221/

You can find many other generators by looking at the module `Test.QuickCheck`. In addition, it is possible to define your own random generators, but this goes beyond the scope of this introduction.

If you want to play around with these random generators, you can use the function `sample` from GHCi to generate a couple of random values with the given generator:

```
> import Test.QuickCheck
> sample (shuffle [1..10])
[9,1,8,7,3,2,10,4,6,5]
[3,10,5,7,2,1,9,4,6,8]
[6,1,4,2,3,7,8,10,5,9]
[6,8,9,10,3,5,4,7,2,1]
[8,9,3,7,10,5,2,4,1,6]
[7,5,2,9,10,4,6,8,3,1]
[9,5,4,6,1,7,3,10,2,8]
[5,3,1,10,9,7,8,4,6,2]
[3,4,10,2,7,6,8,1,9,5]
[3,9,2,8,5,4,1,10,6,7]
[6,5,4,9,2,3,8,7,10,1]
```

5 *Testing properties of functions*

In addition of being able to generate values of basic types such as integers, booleans, and lists, QuickCheck can also generate random *functions* that you can use to write tests. However, for technical reasons related to shrinking, QuickCheck cannot directly generate an element of type, say, `String -> Bool`. Instead, QuickCheck introduces a new (generic) type `Fun a b` with two parameters `a` and `b`, as well as a function `applyFun :: Fun a b -> a -> b`.

For example, we can test that for any function `p :: Int -> Bool`, we have that `p x == True` for all elements of the list `[ x | x <- xs , p x ]`:

```
prop_filter :: Fun Int Bool -> [Int] -> Property
prop_filter p xs =
      -- Filter elements not satisfying p.
  let ys = [ x | x <- xs , applyFun p x ]
      -- If any elements are left...
  in  ys /= [] ==>
        -- ...generate a random index i...
        forAll (choose (0,length ys-1))
          -- ...and test if p (ys!!i) holds.
          (\i -> applyFun p (ys!!i))
```

```
> quickCheck prop_filter
+++ OK, passed 100 tests; 41 discarded.
```

As another (silly) example, let us try to test the property that each function of type `String -> Int` produces the same result on at least two out of three values of `"banana"`, `"monkey"`, and `"elephant"`:

```
prop_bananas :: Fun String Int -> Bool
prop_bananas f =
  applyFun f "banana" == applyFun f "monkey" ||
  applyFun f "banana" == applyFun f "elephant" ||
  applyFun f "monkey" == applyFun f "elephant"
```

```
> quickCheck prop_bananas
*** Failed! Falsified (after 2 tests and 163 shrinks):
{"banana"->0, "elephant"->1, _->2}
```

The function `{"banana"->0, "elephant"->1, _->2}` generated by QuickCheck maps `"banana"` to `0`, `"elephant"` to `1`, and all other strings to `2`. As you can verify, this is indeed a counterexample to the property we defined. It might come as a surprise that QuickCheck is able to come up with this counterexample by itself!

6 Bonus: "Behind the scenes"

The content of this section is not required to use QuickCheck, but it might help you to get a deeper understanding of the types and type classes that make QuickCheck tick.

Let us start by trying to analyse the type of the function `quickCheck`:

```haskell
quickCheck :: Testable prop => prop -> IO ()
```

It takes an argument of polymorphic type `prop` – which must satisfy the `Testable` typeclass – and produces an output of type `IO ()`. Values of type `IO ()` are interactive programs that we can run in GHCi or compile to an executable program; we will look into this type in more detail later in the course. Meanwhile, `Testable` is a typeclass just like other classes we have seen before: `Eq`, `Ord`, `Show`, `Num`, ...

The `Testable` class determines what kind of properties can be tested by `quickCheck`, i.e. what types are valid inputs to `quickCheck`. It has the following relevant instances:

- `Bool` is an instance of `Testable`, which is what allows us to run `quickCheck` on properties of type `Bool`.
- Likewise, the type `Property` is an instance of `Testable`, so we can also run `quickCheck` on properties such as `condition ==> prop` and `forAll gen (\x -> prop)`.
- Finally, a function type `a -> b` is an instance of `Testable` provided `b` is an instance of type `Testable` *and* `a` is an instance of another class `Arbitrary`. This is what allows us to run `quickCheck` on properties that take one or more inputs that are randomly generated.

Next we have the `Arbitrary` typeclass. Types that are an instance of this typeclass have a 'default' random generator `arbitrary :: Arbitrary a => Gen a`. For example, `Bool` is an instance of `Arbitrary` with `arbitrary = elements [True,False]`. In addition, there is also a function `shrink :: Arbitrary a => a -> [a]` that computes a list of all 'shrunk' versions of a value.

When testing a property, QuickCheck will first generate random inputs using the `arbitrary` generator associated to the input type. It will then test the property by running it on the generated inputs. Finally, if it finds a counterexample it will try to `shrink` it step by step as long as the test keeps failing.

7 *Further reading*

- The QuickCheck package on Hackage[4]
- Official QuickCheck manual[5] (somewhat outdated)
- Introduction to QuickCheck v2[6]
- A QuickCheck Tutorial: Generators[7]

[4] https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck.html

[5] http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

[6] https://wiki.haskell.org/Introduction_to_QuickCheck2

[7] https://www.stackbuilders.com/news/a-quickcheck-tutorial-generators

- John Hughes: *Specification Based Testing with QuickCheck*[8] (this contains lots of real-world examples where QuickCheck is used in other languages)

[8] https://www.cs.utexas.edu/~ragerdl/fmcad11/slides/tutorial-a.pdf

- John Hughes: *How to Specify It! - A Guide to Writing Properties of Pure Functions.*[9] (this goes more in depth on how to discover properties that make good property-based tests)

[9] https://research.chalmers.se/publication/517894/file/517894_Fulltext.pdf