

Pedro Vasconcelos (<https://www.fpcomplete.com/user/pbv>) /
An introduction to QuickCheck testing (<https://www.fpcomplete.com/user/pbv/an-introduction-to-quickcheck-testing>)

An introduction to QuickCheck testing

24 Apr 2015 Pedro Vasconcelos
(<https://www.fpcomplete.com/user/pbv>)
View Markdown source
(<https://www.fpcomplete.com/tutorial-raw/2424/db9f17f9a3805ee46f5d176fd74b816cfae300b4>)
(<https://twitter.com/home?status=https://www.fpcomplete.com/user/pbv/an-introduction-to-quickcheck-testing>)

11 👍 (/auth/login)

Next content: A regular expression matcher (<https://www.fpcomplete.com/user/pbv/a-regular-expression-matcher>)

See all content by Pedro Vasconcelos (<https://www.fpcomplete.com/user/pbv>)

Sections

- Proving versus testing programs
- A case-study: string splitting
 - Manual testing
 - Thinking about properties
 - A first attempt
 - More thorough testing
 - Fixing split
- Conclusion

🔗 Proving *versus* testing programs

One of the advantages of the purely-functional paradigm over the imperative or object-oriented ones is that functional programs are easier to reason about. For example, consider the following distributivity law between `reverse` and `++`:

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

We can prove that this property holds for all finite lists using only high-school algebra and induction. In fact, such techniques have been taught in first-year university courses on functional programming since the 1980s.

But properties can also be used for *testing* programs when you don't want to invest the effort to do a formal proof. *QuickCheck* is an excellent Haskell library that allows expressing properties about programs and testing them with randomly-generated values.

A property for QuickCheck is simply a truth-valued function whose arguments are implicitly universally quantified. The library exports function called `quickCheck` that tests a property with 100 randomly-generated values (by default). QuickCheck uses types to generate test data, starting with short lists and small integers and generating progressively larger values. If the property fails for some value, a counter-example is shown; otherwise the test succeeds.

Run the code bellow to test our distributivity law for `reverse`.

Open in IDE (<https://www.fpcomplete.com/clone-active-code/v12JRIg1rI?env=ghc-7.8-stable-14.09>) 📄

```
import Test.QuickCheck

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

main = quickCheck prop_revapp
```

The tests failed! Can you spot what is wrong?

Show

The failure of the test case above shows another cool QuickCheck feature: when a counter-example is found, QuickCheck attempts to shorten it before presenting using a *shrinking* heuristic. This is great because randomly-generated data contains a lot of uninformative "noise"; a short counter-example is much more useful for debugging our programs.

In the test above QuickCheck actually found the *minimal* counter-example: the wrong equation still holds trivially if either of the lists is empty or if they contain identical values (e.g. if both lists are `[0]`) but fails for `[0]` and `[1]`.

☞ A case-study: string splitting

Consider the following list processing problem I gave my first-year functional programming students:

Define a recursive function `split :: Char -> String -> [String]` that breaks a string into substrings delimited by a given character. Some usage examples:

```
split '@' "pbv@dcc.fc.up.pt" = ["pbv","dcc.fc.up.pt"]
split '/' "/usr/include" = ["", "usr", "include"]
```

Suggestion: use the `takeWhile` and `dropWhile` functions from the standard Prelude.



The suggestion is to use some standard higher-order functions from the Prelude to break the string recursively. Here's an initial attempt at a solution:

```
split c [] = []
split c xs = xs' : if null xs'' then [] else split c (tail xs'')
  where xs' = takeWhile (/=c) xs
        xs'' = dropWhile (/=c) xs
```

However, the above definition has one subtle mistake. Let us employ testing to spot and correct the bug.

☞ Manual testing

Let us start by testing the definition with a few sample cases:

```
Open in IDE (https://www.fpcomplete.com/clone-active-code/taT9QIcEuT?env=ghc-7.8-stable-14.09)  

examples = [('@','pbv@dcc.fc.up.pt'), ('/','usr/include')]

test (c,xs) = unwords ["split", show c, show xs, "=", show ys]
  where ys = split c xs

main = mapM_ (putStrLn.test) examples
```

Our two test cases gave the right answers. At this point you might be able to guess some more interesting cases to check for "edge" conditions (you may edit the `examples` list above). But another issue is figuring out what the right answers should be for such cases. This is not too difficult for a simple function, but can be much harder for more complex problems --- even for experienced programmers.

Let us instead ask ourselves one question: *can we think of general properties that `split` should satisfy?*

☞ Thinking about properties

We can devise a property for `split` by consider the *inverse function* of `split` which I shall call `unsplit`. Here are some examples of what we want `unsplit` to do:

```
unsplit '@' ["pbv", "dcc.fc.up.pt"] = "pbv@dcc.fc.up.pt"
```

```
unsplit '/' ["", "usr", "include"] = "/usr/include"
```

Generalizing from such examples we can define `unsplit` as the composition of

`concat` (<https://www.fpcomplete.com/hoogle?q=Prelude.concat>) and `intersperse` (<https://www.fpcomplete.com/hoogle?q=Data.List.intersperse>) (from `Data.List`).

```
unsplit :: Char -> [String] -> String
unsplit c = concat . intersperse [c]
```



We can now express one interesting property: if we first split and then unsplit with the same delimiter, we should get back the original string (i.e. `unsplit` is the left inverse of `split`). This is expressed as the following QuickCheck property:

```
prop_split_inv c xs = unsplit c (split c xs) == xs
```

You should convince yourself that the property above should hold for *all* characters `c` and strings `xs` regardless of how many times `c` occurs in `xs` (including zero occurrences).

🔗 A first attempt

We can now attempt to test `prop_split_inv` using QuickCheck.



Open in IDE (<https://www.fpcomplete.com/clone-active-code/NOzj-Fsf4L?env=ghc-7.8-stable-14.09>)  

```
prop_split_inv c xs = unsplit c (split c xs) == xs

main = quickCheck prop_split_inv
```

Most likely you have observed that 100 random tests passed (or you may have hit a counter-example if you were lucky). In any case, we should be aware of Dijkstra's remark that *testing shows only the presence of bugs, not their absence!*

For random testing it is important to ensure that the test data is reasonably distributed, otherwise we simply get a false sense of security. QuickCheck allows us to collect information on data distribution easily: for example, let us collect the lengths of the `split` results.

Open in IDE (<https://www.fpcomplete.com/clone-active-code/p5KauARKcK?env=ghc-7.8-stable-14.09>)  



```
prop_split_inv c xs
  = let ys = split c xs in
    collect (length ys) $ unsplit c ys == xs

main = quickCheck prop_split_inv
```

The report shows that around 90% of the strings are split into lists of length 0 or 1. This is because when both the delimiter `c` and string `xs` are chosen independently it is unlikely that `c` occurs in `xs`, and if so, more than once. The property might still fail for other cases, which means we haven't tested our program thoroughly.

🔗 More thorough testing

To conduct a more thorough testing we should modify our property slightly: select delimiters that *do* occur in the string. We can do so with an explicit quantifier for choosing the delimiter.

Open in IDE (<https://www.fpcomplete.com/clone-active-code/2Spw1PWMIX?env=ghc-7.8-stable-14.09>)  

```
prop_split_inv xs
  = forAll (elements xs) $ \c ->
    unsplit c (split c xs) == xs

main = quickCheck prop_split_inv
```

This time we found a short counter-example very quickly where the string contains a single delimiter character.

```
split 'a' "a" = [""]
unsplit 'a' [""] = ""
```

This is a counter-example to our property because `"a" /= ""`. But what should we modify to fix the problem? In general, we would have three alternatives:

1. modify the definition of `unsplit`;
2. modify the definition of `split`;
3. revise the property `prop_split_inv` to exclude such cases.

In our case we can reject #1 and #3 because `unsplit` and `prop_split_inv` are short, self-evident definitions; hence we will fix the definition of `split`.

✂ Fixing split



It is easy to see that our property will hold in the counter-example case found if `split` gave the following result:

```
split 'a' "a" = ["", ""]
```

This implies that the equation defining `split` for the empty list is wrong; it should instead be:

```
split c [] = [""]
```

Better still: we can just delete this equation altogether because the general case will give this result anyway after a single recursion step. With this correction our solution passes all tests.

Open in IDE (<https://www.fpcomplete.com/clone-active-code/tRWVMeQ2Tf?env=ghc-7.8-stable-14.09>)  

```
split :: Char -> String -> [String]
split c xs = xs' : if null xs'' then [] else split c (tail xs'')
  where xs' = takeWhile (/=c) xs
        xs'' = dropWhile (/=c) xs

prop_split_inv xs
  = forAll (elements xs) $ \c ->
    unsplit c (split c xs) == xs

main = quickCheck prop_split_inv
```

Note that, considered in isolation, both definitions `split c [] = []` and `split c [] = [""]` seem acceptable (our property holds for the empty string in both cases); the first definition may appear simpler. It is only when we consider the recursion as a whole that we see that defining `split c [] = []` is *less orthogonal* because it breaks our property for strings terminated by the delimiter character.

✂ Conclusion

QuickCheck is a great way to test Haskell programs, not just because it allows conducting many tests cheaply, but mostly because it encourages thinking about general properties that our code should satisfy rather than just specific instances. Thinking early about properties leads to a cleaner, more orthogonal design (i.e., with fewer arbitrarily-specified corner cases).

There is a lot more about QuickCheck that I have not covered in this tutorial namely:

- writing custom generators and shrinking heuristic for user-defined data types;
- controlling the size of generated data;
- controlling the number of tests performed;
- testing monadic code (e.g. in the IO or ST monad).

For more information check the original QuickCheck paper (<http://dl.acm.org/citation.cfm?id=351266&CFID=317197023&CFTOKEN=61604740>) or the one on monadic

testing (<http://dl.acm.org/citation.cfm?doid=636517.636527>); you should also lookup the QuickCheck library documentation (<https://www.fpcomplete.com/hoogle?q=Test.QuickCheck>).

Ghostery blocked comments powered by Disqus.



(<https://twitter.com/home?status=https://www.fpcomplete.com/user/pbv/an-introduction-to-quickcheck-testing>)

[School \(/school\)](#) [Consulting \(/page/consulting\)](#) [Blog \(/blog\)](#) [Help \(/page/support\)](#)
[Feedback \(http://fpcomplete.desk.com/customer/widget/emails/new\)](http://fpcomplete.desk.com/customer/widget/emails/new)
[Log In \(https://www.fpcomplete.com/auth/login\)](https://www.fpcomplete.com/auth/login)
[Sign Up \(https://www.fpcomplete.com/auth/page/email/register\)](https://www.fpcomplete.com/auth/page/email/register)

Copyright © 2013-2015 FP Complete CORP. All rights reserved

[Licenses \(https://www.fpcomplete.com/page/licenses\)](https://www.fpcomplete.com/page/licenses)

- [Terms of Use \(https://www.fpcomplete.com/page/terms-of-use\)](https://www.fpcomplete.com/page/terms-of-use)
- [Privacy Policy \(https://www.fpcomplete.com/page/privacy-policy\)](https://www.fpcomplete.com/page/privacy-policy)