Property Based Testing Tutorial - Haskell

Willem van den Ende, Marc Evers, Rob Westgeest

29 May 2015

Workshop materials and experiments for property based testing tutorial at the Joy of Coding 2015.

This tutorial is self-paced, progressing from painting by numbers to more freeform exercises, with some asides on test suites and debugging. Do it at your own pace, in the order you feel like. Marc, Rob and Willem are on hand if you have any questions. We hope you experience the joy of properties!

Contents

1 What is Property Based Testing?

In practice we usually do *example based testing.* We write an example of how a function or system under test behaves. Examples can communicate very well. It might be difficult to catch all corner cases. And with example based test, we try to find a minimal set of examples, to keep the maintenance burden low.

Example based testing provides just a few data points to capture the system under test. How do you know that you've actually caught those pesky corner cases in your examples?

Property based testing takes a different approach:

- you define properties - invariants of the system-under-test
- the property based testing library generate lots of random input data
- the system under test is run with this input data and the testing library verifies that the invariants hold under all inputs

1.1 Benefits

In our experience, property based testing can be beneficial in several cases:

- Characterization testing - understand existing libraries, quickly test a wide range of cases (e.g. numeric types or libraries to represent money).
- Testing validation logic
- Testing mappings / adapters
- Finding pesky corner cases
- When example based testing gets repetitive and/or you want to check many cases, and there is no obvious way to express the variations succinctly. Or when your examples don't express your intent very well, no matter how hard you try.

Furthermore, Property Based Testing drives your design, something it has in common with Test Driven Development (although it drives your design in a different way). Property based testing forces you to think about invariant properties of your code and capture these explicitly. Defining custom generators helps you get a better understanding of the preconditions of your code.

Property based testing is a relatively new practice. We are still learning about its applicability and usefulness.

1.2 Key concepts

Properties: invariants of the code under test; invariants should always be true under all valid inputs

Arbitraries & generators: A generator is a function that can generate arbitrary data of a specific type. An arbitrary combines a generator with a shrinking function.

Shrinking: Shrinking is used when a counter example is found, to reduce the counter example to the smallest one that still fails the property

1.3 I want this in my favorite language!

Property based testing is available for most programming languages, like:

- Haskell: Quickcheck
- Scala: ScalaCheck
- Java: https://github.com/pholser/junit-quickcheck
- Javascript: many libraries available, like *jsverify* and *claire*
- C# / .Net: FsCheck

We provide this tutorial in Haskell, because that is where it all started. There is also a javascript version of this tutorial.

2 Getting Started

Open the sample project. There is an empty haskell module file in:

```
src/GettingStarted.lhs
```

The main file is in src/joy.hs in case you want to add your own modules, the cabal file is called hsmoney.cabal.

Once it compiles you can build and run with cabal

```
cabal -j build && ./dist/build/joy/joy
```

```
    -- We will explain why we need this extension later:
  {-# LANGUAGE ScopedTypeVariables #-}
module GettingStarted where
```

We need the Test.QuickCheck module to generate random data and to calculate properties.

```
import Test.QuickCheck
```

Let's start with a simple example using some math functions, to introduce the basic concepts. An invariant of the square root and square functions is:

the square root of a number's square is equal to that number, in pseudocode:

```
sqrt( x * x ) = x
```

In an example based test approach, we would write a small number of examples showing that this holds. We now want to 'proof' that the mentioned invariant holds for all numbers.

Define the 'system under test'. We have a 'squared' function, and we don't care that much yet about what type its input 'x' is, so we leave it off for now.

$$\mathit{squared}\ x = x * x$$

We want to show that for all numbers, the following holds:

```
sqrt (squared n) == n
```

So we define a property stating just that. We add a type signature for a fitting number type, so that QuickCheck can choose what kind of values to generate. At this stage it would compile just fine without a type signature. Compilation will fail once you use it with quickCheck, because there are several Floating instances to choose from. We choose Double. y

$$\mathit{prop\_SquareRootOfNSquaredEqualsN} :: \mathit{Double} \rightarrow \mathit{Bool}$$
$$\mathit{prop\_SquareRootOfNSquaredEqualsN}\ n = (\mathit{sqrt}\ (\mathit{squared}\ n)) \equiv n$$

Starting it with 'prop_' is more than just a naming convention, we will use this in later chapters to generate a test suite. You can load this into ghci and play around with it for some n. In this chapter we will show the quickCheck invocation from a

```
mainN
```

at each step, so you can more easily follow along in

```
cabal repl
```

.

And now comes the magic:

$$\mathit{main0} = \mathit{quickCheck}\ \mathit{prop\_SquareRootOfNSquaredEqualsN}$$

add it to

```
runTests
```

in

```
GettingStarted.hs
```

and use

```
cabal -j build && ./dist/build/joy/joy
```

.

or load it in cabal repl.

This will fail after some number of steps. For me it failed after three:

```
*** Failed! Falsifiable (after 5 tests and 1080 shrinks):
5.0e-324
```

What happened? QuickCheck generates input test data and checks if the property holds (returns true) for all the generated input values. If it finds a case for which the property does not hold, it fails, and returns the counter example.

How does it know to generate test data? QuickCheck will use the property's type to generate data. Default generators are available for many built-in types such as numbers, strings, and even functions. In this case it will generate Doubles, because that is what we specified in the type of the property.

QuickCheck tells us that the failing input is a very small number with mantissa 5 and exponent -324. What happens when we multiply and perform sqrt on such a small number?

Using QuickCheck for just a little bit, we realize that when we deal with floats, rounding is a problem. QuickCheck generates a really small number as value for ‘n‘, very close to zero and then runs out of precision when doing (sqrt (squared n)).

This is annoying. Let's work around it by making our own ‘sqrti‘ function that rounds to the nearest integer, and write a new property for which QuickCheck will generate only integers. Otherwise we could round inside the property, but that would be very messy.

$$\begin{aligned}&\mathit{sqrti} :: \mathit{Integral}\ a \Rightarrow a \rightarrow a\\ &\mathit{sqrti} = \mathit{floor} \circ \mathit{sqrt} \circ \mathit{fromIntegral}\\ &\mathit{prop\_SqrtiOfSquaredNEqualsN} :: \mathit{Int} \rightarrow \mathit{Bool}\\ &\mathit{prop\_SqrtiOfSquaredNEqualsN}\ n = (\mathit{sqrti}\ (\mathit{squared}\ n)) \equiv n\end{aligned}$$

Now it fails again:

```
*** Failed! Falsifiable (after 3 tests and 1 shrink):
-1
```

You notice that the output mentions tests and shrinks. The number of tests and shrinks you see will be different. When it finds a counterexample, QuickCheck shrinks the input so that you get the ‘smallest‘ input that fails. In this case the smallest number it can find. For lists it is the shortest list, etc.

The test fails because the invariant does not hold for negative numbers. We should restrict the generated input to natural numbers only (numbers $\geqslant 0$). We can do this by changing the type of our generator to NonNegative a, in our case NonNegative Int. NonNegative is a Modifier:

https://hackage.haskell.org/package/QuickCheck-2.8.1/docs/Test-QuickCheck-Modifiers.html.

```
prop_SqrPositive :: NonNegative Int → Bool
prop_SqrPositive (NonNegative n) = (sqrti $ squared n) ≡ n
main3 = quickCheck prop_SqrPositive
```

This way we can communicate our intention for the property's input in its type.

We can also use generators explicitly inside the property to achieve a similar effect. You can find many available generators by browsing the documentation for the Gen module.

We could for instance say we are only interested in numbers between 1 and 100. For that we use the 'choose' function together with forAll. Since we made the type of sqrti a bit general, we have to specify what Integral we want, so we choose (n :: Int).

```
smallPositiveInteger = choose (1, 100)
prop_SqrSmallInt = forAll smallPositiveInteger $ λn →
  (sqrti (squared n)) ≡ (n :: Int)
main4 = quickCheck prop_SqrSmallInt
```

There is one subtlety we did not show: if you use forAll, the type of your property changes. It no longer results in a Bool, but in a Property. We can declare it like this:

```
prop_SqrSmallInt :: Property
```

```
prop_SqrPositiveForGreaterEqualsZero n =
  n ⩾ 0 ==> (sqrti (squared n)) ≡ (n :: Int)
```

Here we show how to turn this into a small test suite by hand.

```
runTests = do
  quickCheck prop_SquareRootOfNSquaredEqualsN
  quickCheck prop_SqrSmallInt
  quickCheck prop_SqrPositive
  quickCheck prop_SqrtiOfSquaredNEqualsN
  quickCheck prop_SqrPositiveForGreaterEqualsZero
```

This acts as a nice summary for what we have just covered. We have written a deceptively simple function that, when it comes to testing, has a few edge cases we had to think about. Writing QuickCheck properties forced us to think about the type of inputs we can deal with and the constrainst to apply to that input. We have also played with a few ways to express constraints for a property.

Writing a test suite like this is boring. In the next seciont we will generate a test suite using Template Haskell.

3 A suite of properties

If you've done example based testing, you are probably familiar with the concept of a test suite. While QuickCheck can be used with hspec and tasty, in the following chapters we will use a bit of template haskell to collect our properties for us, so we don't need to introduce a test framework as well.

The module Test.QuickCheck.All contains a few '*checkAll' functions to automatigically run all the properties in a module. This way we can quickly create a suite of properties.

We'll demonstrate it with quickCheckAll, you can look up the others in the module documentation. To let quickCheckAll generate code, we need the TemplateHaskell language extension.

```
{-# LANGUAGE TemplateHaskell #-}
module PropertySuite where
```

We've seen Test.QuickCheck, and Test.QuickCheck.Property before. We need the Template Haskell (TH) module in addition to the language pragma.

```
import Test.QuickCheck
import Language.Haskell.TH
import Test.QuickCheck.All (quickCheckAll)
import Test.QuickCheck.Property ((===))
```

We'll introduce two small properties to show we can run them. Their names start with prop_ so quickCheckAll can find them.

```
prop_reverse xs = (reverse (reverse xs)) === xs
prop_associative :: Int → Int → Int → Property
prop_associative x y z = ((x + y) + z) === (x + (y + z))
```

The following code has to go at the bottom of your module, otherwise TemplateHaskell can not find all the functions.

```
return [ ]  -- needed in GHC 7.8 series, not after
runTests = $quickCheckAll
```

There are a few variants of *CheckAll that you may find useful, See the Test.QuickCheck.All documentation. So now you know what the TemplateHaskell and the bit of magic at the bottom of the following chapters are for.

4 Modelling Money

In this exercise, we are going to build a class representing Money, step by step. Representing money in software might sound easy, but it is actually non trivial and is often done wrong. Doing it wrong can result in small or big rounding errors or just losing a few millions on a bad day...

4.1 Getting started

Open the Money.hs file, and add the quickcheck boilerplate, and import fmap and sequential application from Control.Appliccative:

```
  -- for Arbitrary Amount and Currency, and to add Amounts
 {-# LANGUAGE GeneralizedNewtypeDeriving #-}
  -- to generate a test suite as explained in TODO REF
 {-# LANGUAGE TemplateHaskell #-}
module Money where
import Test.QuickCheck
import Test.QuickCheck.Property ((===))
import Language.Haskell.TH
  -- to generate Arbitrary instances
import Control.Applicative ((<$>), (<*>))
```

Create a GADT to represent money. It is constructed with an amount and a currency. This is a good moment to think of the types of amount and currency. (TODO from quickCheck manual: use newtypes ??)

We learnt in the introduction that QuickCheck could easily force us to change our mind about the types that we use. Primitive obsession (using Strings, Ints and other primitive data types instead of types we define ourselves) is painful enough. Lets keep our options open and define an Amount as an Int, meaning a number in cents. For the purpose of this exercise we don't want to get into rounding errors again, we've been there already today. The Currency will be represented as a String.

```
newtype Amount = Amount {amountValue :: Int}
  deriving (Arbitrary, Num, Eq, Show)
newtype Currency = Currency {currencyValue :: String}
  deriving (Eq, Show)
```

For Amount we use GeneralizedNewTypeDeriving, so we can derive an Arbitrary instance to generate arbitrary amounts. The Eq and Show instance are needed so our properties can compare amounts, and quickcheck can show us the failures.

Currency has a string on the inside. We could of course do this with a proper datatype, but for this tutorial the strings make it a bit more interesting, because we don't want arbitrary strings, we want strings representing a currency.

```
data Money = Money {currency :: Currency
  , amount :: Amount
  } deriving (Eq, Show)
```

Choosing explicit data types also will help us generate very specific arbitrary data with QuickCheck.

4.2 Currency Properties

When we add two amounts of the same currency, we get a new amount. Our property needs two money objects, so we define a property function with two arguments:

```
prop_moneyOfSameCurrencyAddsUp m1@(Money _ (Amount a1))
  m2@(Money _ (Amount a2)) = (a1 + a2) === (total mtotal)
  where total (Right money) = amountValue $ amount $ money
      -- TODO Magic number
    total (Left msg) = -4242424242424242
    mtotal = addMoney m1 m2
```

This is quite possibly some of the ugliest haskell code I've ever written. And we are ignoring the currency!

Exercise: After you've defined the generator for currencies and Arbitrary Money below, get rid of the magic number, and find a way to express the constraint that two Moneys should be of the same currency.

Now we have to generate arbitrary money. We'll do this in small steps, a bit smaller than we would do in production code, so you can see another example of a custom generators as well as Arbitrary instances for more complex data types.

QuickCheck can generate random amounts for us without our intervention. But we don't want random strings for currencies! We can generate the currency using the 'elements' generator:

```
currencies :: Gen Currency
currencies = elements (map Currency ["EUR", "USD", "GBP"]);
```

How do we generate Money instances? We define an instance of the Arbitrary typeclass and create an implementation for its' arbitrary generator. Arbitrary is what provides shrinking behaviour etc, but the only function we really have to implement is arbitrary.

We use the Applicative instance for Arbitrary to construct Money, because there is no dependency between currency and arbitrary. We could have used the Monad instance here (as you will see in the original quickCheck paper), but that would have been overkill. The Appliccative instance makes it almost look like we are just constructing a record the regular way. This makes it quite easy to create Arbitrary instances for all our domain objects.

```
instance Arbitrary Money where
  arbitrary = Money < $ > currencies < * > arbitrary
```

We could also define an Arbitrary for currency like this,

```
instance Arbitrary Currency where
  arbitrary = elements (map Currency ["EUR", "USD", "GBP"])
```

or simply reusing the 'currencies' generator we defined above. When do you think deriving an Arbitrary is more useful than a generator function? When do you think defining a generator function is more useful than an Arbitrary instance? Does it make sense to create new data types specifically so that we can make Arbitraries?

Lots of choice here. Play around with them and see if and how they influence the design of your properties and production code.

Now we can generate Money objects!

4.3 Implement behaviour for Money

Now add the behaviour that Money can only be added to money of the same currency. Express that different currencies are not supported (e.g. by using an Either). Make sure your property handles this correctly.

```
newtype ErrorMessage = ErrorMessage { errorMsg :: String }
addMoney :: Money → Money → Either ErrorMessage Money
  -- your implementation here
```

Add a new property that checks that adding is not possible whenever the currencies are different.

4.4 Exercise: Money allocation

We want to be able to allocate money: divide it in equal parts, without money getting lost in rounding. An example: dividing EUR 100 by 3 should yield:

```
[EUR 33, EUR 33, EUR 34]
```

We are assuming whole numbers for now.

Define a property that states that no money gets lost when dividing an amount.

Note that we probably should make the integer assumption of the amounts in the example explicit in the types.

4.5 Exercise: Converting from string

We want to be able to create money objects from strings; we support EUR and USD. Some legal values:

- 100 (default is EUR)
- EUR 10
- USD 300
- USD 40- (negative value)
- EUR 10.000 (. as thousand separator for euros)

And some illegal values:

- USD 4.999 (USD has , as thousand separator and . as cents separator)
- 1,000 (illegal because EUR is default and EUR uses . as thousand separator)
- HFL 50 (illegal currency)
- EUR 10bla

In our domain model, we work with valid objects. This part focuses on processing and validation outside, 'untrusted' data that is transformed into domain objects (e.g. data coming in from a web form). We want this transformation to be robust, so that it can handle any data and only results in valid domain objects when it is able to (and we don't need to do defensive programming in our domain).

There are actually two concerns we need to address:

- Given the input string is valid, will our code create a valid Money object with the correct contents? (correctness)
- Is the conversion code robust for every possible input string? The code should always either return a valid Money object or a rejection of the input.
- It should not throw unexpected exceptions or create invalid Money objects (robustnesss).

Let's capture these in properties.

Define conversion function that returns a JSON object that is either money: ¡valid money object¿ or error: 'some error message'

Start e.g. with something like:

```
-- TODO sample code
```

We can start with either correctness or robustness first. It would be interesting to try both approaches and see whether and how it drives you to different design decisions.

4.5.1 Correctness

Given correct input, the conversion should produce valid Money objects. How would you define this as property?

Write an arbitrary that creates valid input strings. You can again use smap to create valid strings from primitive values.

Hints and tips: - If you let your arbitrary generate a tuple (array) of an input string together with the corresponding amount and currency values, it will be easier. for your property function to check correctness:

```
['EUR 100', 'EUR', 100]
```

- Take baby steps; start e.g. with regular input with a currency; add support for the default currency; add support for thousand separators. - Make sure your test fails for the right reason at every step.

4.5.2 Robustness

Given any input, the conversion should always produce either a valid Money object or a validation error:

```
{ money: ... } or { error: 'some message' }
```

Define a property that captures this.

Write an arbitrary that creates all kinds of input strings. Is the 'asciistring' arbitrary suited for this? Why (not)?

Corner cases that almost look like valid input are particularly interesting, like "EUR 12jsde", "HFL 30", and "300 EUR". Adapt your arbitrary so that it will generate cases like these.

Refine your conversion function and make sure the correctness property is also satisfied.

Hints and tips: - jsverify provides the 'oneof' function that combines two arbitraries into one that generates values taking values from both arbitraries.

4.6 Reflection

How did defining the properties and the arbitraries influence you in thinking about the problem?

Where would you like to put the property logic? To what extent is it test code, to what extent should/could it be part of production code?

What happens if you would start with robustness first and then correctness? Do the exercise again, but start with robustness and see if this leads you to different design decisions.

We end with the usual boilerplate to generate tests with TemplateHaskell.

```
return []  -- needed in GHC 7.8 series, not after
runTests = $quickCheckAll
```

5 Debugging

Out of the box QuickCheck does not give that much feedback about what fails. It will show the input values for the property that failed, but what happens inside that property is invisible.

We've seen in the introduction that we can use the '===' operator to at least see the expected and actual output inside a property. But what if we don't want to compare with equality, or if we want to see some other values in our property or the code under test.

We will import Haskell's Debug.Trace module, so we can print values to the console, even though we are otherwise operating without IO.

```
module Debugging (runTests) where
import Test.QuickCheck
import Debug.Trace
```

5.1 Using standard haskell debugging

If you are experienced in debugging haskell programs, you can skip to the next section, otherwise it is a good idea to follow the example below, so that you don't get stuck doing the more elaborate exercise.

On a failure, QuickCheck only shows the resulting value, not the input. If you want to see all the inputs and results, use traceShow inside the property. This can be very useful when developing new properties, or understanding a regression.

We use the same squared function as in the introduction:

```
squared :: (Num x) => x -> x
squared x = x * x
```

This is the property we had. We'll use this below to illustrate some other forms of debugging as well.

```
prop_TraceSqrNEqualsN :: Double -> Property
prop_TraceSqrNEqualsN n = (sqrt $ squared n) === n
```

We add the debugging code in a where block, so we can easily reuse or remove it:

```
prop_TraceSqrNEqualsNDebug :: Double -> Property
prop_TraceSqrNEqualsNDebug n = debugShow $ result === n
  where
    result = sqrt $ squared n
    debugShow = traceShow $ "input: " ++ show n
      ++ " result: " ++ show result
```

This will produce a lot of output. Even if the property fails after just a few tests, this will also print for all the shrinks. Here is a tail of the test run for the above property:

```
"input: 1.0e-323 result: 0.0"d 1077.1 shrinks)...
"input: 0.0 result: 0.0"ts and 1078 shrinks)...
"input: 5.0e-324 result: 0.0"d 1078.1 shrinks)...
"input: 0.0 result: 0.0"ts and 1079 shrinks)...
```

```
Falsifiable (after 2 tests and 1079 shrinks):
5.0e-324
0.0 /= 5.0e-324
*** Failed! Falsifiable (after 2 tests and 1074 shrinks):
5.0e-324
```

5.2 Playing with the number of test cases

By default it is configured to generate 100 test cases. You can specify a different number using quickCheckWith, here with a the property we just defined.

```
tenTests = quickCheckWith stdArgs {maxSuccess = 10}
  prop_TraceSqrNEqualsNDebug
```

5.3 Expecting failure

We can mark tests as expectFailure, so a property suite as a whole will pass. This comes in handy for writing quickCheck tutorials - some of our properties should fail.

```
shouldFail = expectFailure prop_TraceSqrNEqualsN
```

The result is a success, and the output shows that the property fails as expected, so there is no mystery here:

```
*Debugging> res <- runTests
+++ OK, failed as expected. Falsifiable (after 3 tests and 1083 shrinks):
5.0e-324
0.0 /= 5.0e-324
```

5.4 TODO explain use of sample to see what data gets generated.

5.5 Running only the failing property

We collect only the one property that should fail, to demonstrate that we can run it just like any other property:

```
runTests = quickCheck shouldFail
```

6 A vending machine with a coin box

In this exercise, we are focusing on a (drinks) vending machine, and specifically the cash register part of such a machine.

Start with modelling a coin box, in which customers can insert different coins and after making a selection, the appropriate amount is checked out. In the end, the coin box should also correcly handle change: any money left over is returned to the customer, using appropriate coins.

6.1 Getting started

Let's start simple. We want to create a Coinbox with 'insert' and 'checkout' functions. We can model the coin box as having two collections of coins: the 'inbox' (coins inserted but not yet processed) and the 'vault' (coins collected by the coin box after doing checkouts. The coin box keeps the coins until it is emptied by the owner. The coins from the vault can also be used to return the correct change, depending on what types of coins are needed.

We can create representations of all kinds of different coins, but it is sufficient for now to just model two or three (like 10 cents and 50 cents).

Start with a few simple invariants, like:

- the total value of the coins in the inbox should always be >= 0
- the total value of the coins in the vault should always be >= 0

Create a new module with the necessary imports. Create the properties one by one, first letting them fail. Write the simplest version of the Coinbox that satisfies a property.

6.2 Mutating state: doing checkouts

Now proceed with implementing checkouts. For now, assume that you do a checkout of all the coins present in the inbox.

First, a property concerning checkouts: first, a checkout empties the inbox. Write a property for this, see it fail for the right reason.

Second, a checkout should not modify the total value of the coins in the inbox and vault.

Write the property, see it fail for the right reason. Implement the checkout function for Coinbox.

Observe closely what happens. Do you get failures? Are they what you expect? If not, what happens?

6.2.1 Some tips

- It is useful to derive Show for Coinbox and Coins.
- The === operator will show the left- and right-handside on failure.

See the Debugging section for more tips.

6.3 Inserting coins

Let's add an insert function for the customer to insert one coin.

What property/properties can you define regarding inserting coins?

Code these properties and implement the insert. Make sure all other properties keep on being satisfied!

6.4 Possible next steps

If you want to continue with this exercise, your next steps could be:

- Conditional checkout: pass a price to the checkout function and only check out if the inbox contains a sufficient amount of money.
- Refine the checkout so that a specific amount is checked out and the remaining amount is returned as change. How will you make the coin box return the right coins?