Random Testing in Haskell

Shae M Erisson

2022-04-26

Outline

What it is

What it isn't

What it could be

What it is

What is Random Testing?

Random testing is like unit testing, for all the things!

Unit testing checks the happy path, this input should give that output.

What about everything else? Are there any corner cases I have forgotten?

At one of my previous jobs, someone put unicode into our input forms and the entire application crashed!

How do I make it go?

Random testing is declaring a property that should be true, and handing that off to software that generates inputs to check whether the property holds.

For example, the associative property says any three numbers when multiplied in any order, should give the same result:

```
(a * b) * c == a * (b * c)
```

The reflexive property is even simpler, where for any value, we expect that value to be equal to itself:

```
x == x
```

Both of these are true for all values in a computer, riiight?

QuickCheck

QuickCheck gives you all the things in the type.

Let's see the associative and reflexive properties in QuickCheck.

```
prop_associative :: Float -> Float -> Float -> Bool
prop_associative x y z = ((x * y) * z) == (x * (y * z))

prop_reflexive :: Float -> Bool
prop_reflexive f = (f / 0) == (f / 0)
```

QuickCheck Results

```
*** Failed! Falsified (after 4 tests and 16 shrinks):
-0.1
0.1
-9.0e-2
*** Failed! Falsified (after 1 test):
0.0
```

QuickCheck gives you some value in the type, but you don't get to choose. If you wanted some specific range of values, you want Hedgehog.

Hedgehog

Hedgehog gives you exactly what you requested.

```
prop_associative :: Property
prop_associative = property $ do
  a <- forAll $ Gen.float (Range.linearFrac (-100) (100))
  b <- forAll $ Gen.float (Range.linearFrac (-10000) (10000
  c <- forAll $ Gen.float (Range.linearFrac 5 1000000)
  ((a * b) * c) === (a * (b * c))

prop_reflexive :: Property
prop_reflexive = property $ do
  a <- forAll $ Gen.float (Range.linearFrac (-100) (100))
  (a / 0) === (a / 0)
```

Hedgehog Results

```
Example
 prop_reflexive passed 100 tests.
 prop_associative_float failed at Main.hs:26:3
  after 2 tests and 75 shrinks.
[..]
```

Oh hey, did you notice that part about about 75 shrinks? Both QuickCheck and Hedgehog will shrink an input to the smallest failing input. It's easier to understand a failing input that's one line long intead of thirty lines long.

Hedgehog surprise

Did you notice that `prop_reflexive` succeeded?

If you change `Gen.float (Range.linearFrac (-100) (100))` to `Gen.float (Range.linearFrac 0 (100))` it fails as show in the previous QuickCheck example.

This leads us to the next point ...

Generators are hard to do well

At first, I thought coming up with properties was the hard part, but it turns out... Generators are hard.

```
genIPv4 :: Gen IPv4
genIPv4 = do
  let rng = IPv4.toList $ IPv4.fromBounds (IPv4.fromOctets
  let len = length rng
  ix <- Gen.int (Range.linear 0 (len - 1))
  pure $ rng !! ix
```

What it isn't

Generated inputs are often small

When asked to generate any list, QuickCheck usually generates many small lists, and that doesn't exercise your property so well.

That led to smallcheck and later leancheck, designed to start with the smallest values of a type and enumerate to larger types.

Did you really exercise your code?

One downside of property based testing is that it's entirely random, you don't know how much of the code under test was exercised.

You can get around that somewhat by using the Haskell program coverage tool hpc to see how much of your program was executed during a property test run. Here's the result of running tests on a simple bittorrent encoder. `./hpc.gif`

Interesting questions

When using continuous integration, what about a fixed seed for property tests? At my last job, we had a fixed seed given to the property test framework when it executed in Jenkins. Does this remove the benefits of random testing?

What it could be

coverage driven property testing

Much of the content here is borrowed from Dan Luu's post on how to improve testing.

What about hooking hpc code coverage into QuickCheck's random testing?

At a miminum we'd know when to stop testing because everything is covered.

Even better, we could interactively extend the generated value to explore code paths, and stop when coverage does not increase.

Ask me about my project to do exactly this thing!

bonus content

Rudy Matela's PhD thesis has a big pile of cool tools.

- LeanCheck is a good take on enumerative testing in the style of smallcheck.
- FitSpec can find duplicate or overlapping properties by mutating properties.
- Speculate can find equations and inequalities
- Extrapolate can generalize counterexamples.

Nick Smallbone maintains several cool tools.

- quickspec discovers equational laws for you.
- quickcheck-with-counterexamples lets you get the failing input as a Haskell value