Property Testing With Hedgehog

Eating Your Bugs

Daniel Cartwright

Atlanta Functional Programming

August 06, 2019

Summary

What is Property Testing?

The Problems With Testing

- How expensive is breakage?
- Tooling (PL, build system, etc.)
- Types of testing, verification
- Unit Tests are Extremely common

```haskell
-- 'hspec' unit test example
myAssertion :: Assertion
myAssertion = myFunction exampleInput `shouldBe` expectedOutput
```

The Problems With Testing (Cont'd)

Definition

Unit Test A test that checks whether the output of a function, for a single example, is equal to the expected output.

But, there are problems!

- Coverage Problem: Coverage of all code paths
- Developer Cost: Costly w.r.t developer time
- Unenjoyable to write (bad for morale)

Property Testing As A Solution To The Coverage Problem

Property testing:

- Parametric in its inputs
- Based on declarative *properties*, rather than manually-constructed inputs
- The inputs are generated, rather than supplied by humans

How do you generate these inputs? Depends:

- Exhaustive property tests
- Randomised property tests
- Exhaustive, specialised property tests
- Randomised, specialised property tests

Example

```
prop_reverse :: [Char] -> Bool
prop_reverse str = reverse (reverse str) == str
```

Generators

- Randomised generators are a way to describe how you should generate some input to a property test
- Appropriate generators can be the difference between finding bugs, and not finding bugs
- Not all types are easy to generate (e.g.: Word8 vs String)
- Generators can often be constructed using combinators, allowing smarter construction of generators (e.g. filtering,sizing)

Example where a bad generator could fail:

```
prop_isSmall :: [a] -> Bool
prop_isSmall ls = length ls < 100
```

Shrinking

- Property test failures result in counterexamples can be large
- This can make reasoning about the failure harder
- What if we could shrink counterexamples to manageable sizes?

```
prop_allLower :: [Char] -> Bool
prop_allLower str = all isLower (map toLower str)

-- Counterexample
-- "As5Enu3au04"

-- Shrunken counterexample
-- "1"

--   toLower '1'
-- '1'
```

Note that the shrunk counterexample "1" is not necessarily part of the original counterexample.

Shrinking With Invariants

- Shrunk values must have been able to come from the generator that generated the unshrunk value

Consider:

$\forall n.(n > 6) \Rightarrow (n > 5 \wedge odd(n))$

If shrinking does not obey the invariants of the generator:

```
-- "Failed: the following number larger than six does not pass the
```

Approaches To Property Testing

Libraries

Table: Property Testing Libraries In Various Languages

Language	Libraries
C	theft
C++	CppQuickCheck
Clojure	test.check
Common Lisp	cl-quickcheck
Coq	QuickChick
Erlang	QuickCheck
F#	FsCheck Hedgehog
Golang	gopter quick

Libraries (Cont'd)

Table: Property Testing Libraries In Various Languages

Language	**Libraries**
Haskell	QuickCheck Hedgehog Validity SmallCheck
Java	QuickTheories
Javascript	jsverify
PHP	Eris
Python	Hypothesis
Ruby	Rantly
Rust	QuickCheck
Scala	ScalaCheck
Swift	SwiftCheck

QuickCheck

- Pioneer of randomised property testing
- uses ‘Arbitrary‘ typeclass for things that can be generated
- uses ‘Testable‘ typeclass for things that are testable

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

Example:

```
> quickCheck $ \(str :: String) -> reverse (reverse str) == str
```

QuickCheck (Cont'd)

```
reflexive :: Arbitrary a => (a -> a -> Bool) -> Property
reflexive rel = \x -> x `rel` x
```

But there's a problem:

- This combinator can only test reflexivity of a given binary relation if the relation is reflexive for *all* possible values that may be generated by the arbitrary generator.

This is already a problem with 'Rational' and 'Eq'. The 'Eq' instance for 'Ratio a' assumes that values are normalised:

```
> 2 :% 2 == 1 :% 1
False
```

Problems With QuickCheck

‘Abitrary‘ is a lawless typeclass, with no precise semantics!
How should we implement the ‘Arbitrary‘ instance for ‘Rational‘?:

- Should we ever let it generate 0 :% 0?
- Should we ever let it generate 1 :% 0?
- Should we ever let it generate 5 :% -1? How about -5 :% 1?
- Should we take the size parameter into account?

There are several considerations:

- Never generating un-normalised values could fail to test edge-cases
- Generating un-normalised values means we cannot test properties that require (==) to work properly

Solution: newtype wrappers?
But there’s a limitation there too...

Problems with QuickCheck (Cont'd)

- Expensive generators and shrinking
- QuickCheck docs: "There is no generic arbitrary implementation included because we don't know how to make a high-quality one."
- There is no default implementation of 'arbitrary'
- The default implementation of shrink is 'const []'. This means that by default, values are never shrunk!
- Developers must implement shrinking themselves, which costs developer time. Moreover, test code is already among the most sloppy, hastily written code in most any given codebase.
- Orphan instances for 'Arbitrary'

Hedgehog

- Very new approach to property testing (Stanley, J. 2017)

```haskell
import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range

prop_reverse :: Property
prop_reverse = property $ do
  xs <- forAll $ Gen.list (Range.linear 0 100) Gen.alpha
  reverse (reverse xs) === xs
```

Hedgehog

- Free shrinking: Paired with every generator internally
- Shrinks are represented as a lazy tree which just needs to be traversed for consecutive shrinks
- Much prettier/clearer output, with source

Hedgehog

```haskell
import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range
import qualified Data.List as List

genIntList :: Gen [Int]
getIntList =
  let listLength = Range.linear 0 10_000
  in Gen.list listLength Gen.enumBounded

prop_reverse :: Property
prop_reverse = property $ do
  xs <- forAll genIntList
  reverse (reverse xs) == xs

-- > check prop_reverse
--   prop_reverse passed 100 tests.
```

Hedgehog

Let's do what I do best and write some bad code...

```
-- Drops an element somewhere around the middle of the list.
fauxReverse :: [a] -> [a]
fauxReverse xs =
  let sx = List.reverse xs
      mp = length xs `div` 2
      (as, bs) = List.splitAt mp sx
  in as <> List.drop 1 bs

prop_fauxReverse :: Property
prop_fauxReverse =
  property $ do
    xs <- forAll genIntList
    fauxReverse xs === List.reverse xs
```

Hedgehog

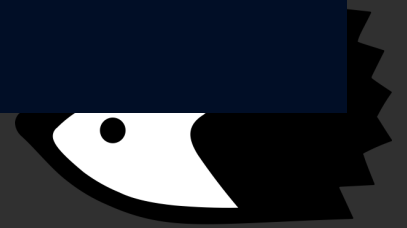
And now we test my bad code...

```
λ> check prop_fauxReverse
  ✗ <interactive> failed after 2 tests and 11 shrinks.

        ┏━━ Hedge.hs ━━
     30 ┃ prop_fauxReverse :: Property
     31 ┃ prop_fauxReverse =
     32 ┃   property $ do
     33 ┃     xs <- forAll genIntList
        ┃     │ [ -9223372036854775808 ]
     34 ┃     fauxReverse xs === List.reverse xs
        ┃     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        ┃     │ Failed (- lhs =/= + rhs)
        ┃     │ - []
        ┃     │ + [ -9223372036854775808 ]

    This failure can be reproduced by running:
    > recheck (Size 1) (Seed 6043815965182080260 5082204861531945465) <property>

False
λ> minBound :: Int
-9223372036854775808
```



Hedgehog

```haskell
-- A reverse function should preserve every element in the list!
prop_fauxReverseLength :: Property
prop_fauxReverseLength = property $ do
  xs <- forAll genIntList
  length (fauxReverse xs) === length xs
```

```
λ> check prop_fauxReverse_length
  ✗ <interactive> failed after 2 tests and 1 shrink.

    ┏━━ Hedge.hs ━━
 36 ┃ prop_fauxReverse_length :: Property
 37 ┃ prop_fauxReverse_length =
 38 ┃   property $ do
 39 ┃     xs <- forAll genIntList
    ┃     │ [ -9223372036854775808 ]
 40 ┃     length (fauxReverse xs) === length xs
    ┃     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    ┃     │ Failed (- lhs =/= + rhs)
    ┃     │ - 0
    ┃     │ + 1

  This failure can be reproduced by running:
  > recheck (Size 1) (Seed 4416742318514417762 31116095073768267) <property>

False
```

Real Examples Of Property Testing

Examples Of Property Testing: Show Me The Code!

https://github.com/chessai/hedgehog-talk

Acknowledgments

- Thanks to the Atlanta Functional Programming Group for letting me give this talk.
- Thanks to Jacob Stanley, Tim Humphries, and the many other Hedgehog contributors

References

Complete, F. *QuickCheck, Hedgehog, Validity.* https://www.fpcomplete.com/blog/quickcheck-hedgehog-validity.

Humphries, T. *Property testing with Hedgehog.* https://teh.id.au/posts/2017/04/23/property-testing-with-hedgehog/.

Questions?