QuickCheck
Automated Random Testing for Haskell

Jeff Vaughan

Revised May 4, 2005
Overview

Introducing Properties

Conditional Laws

Custom Generators

Foreshadowing the Monadic Extension
QuickCheck is a powerful unit testing tool.

- QuickCheck takes a set of parameterized assertions called \textit{properties} as input.
- Properties are checked against a large number of randomly generated cases.
- Programmers control the distribution of test cases with \textit{conditional laws} and \textit{generators}.
- To ensure good test coverage, QuickCheck can \textit{classify} cases (e.g. as trivial) or \textit{collect} data for a histogram.
Properties are Haskell functions that encode assertions.

For example, the following property asserts that addition and multiplication distribute:

```
prop_Distributive :: Int -> Int -> Int -> Bool
prop_Distributive a b c =
    a * (b + c) == (a * b) + (a * c)
```

We check this on the Hugs command line with:

```
Demo> quickCheck prop_Distributive
OK, passed 100 tests.
```
If a counter example to a property is found, QuickCheck reports it.

While integer operations distribute as expected, floating point math is notoriously strange. Let's use test the distributive property for Floats.

```
prop_Distributive ::
    Float -> Float -> Float -> Bool
prop_Distributive ' a b c =
    a * (b + c) == (a * b) + (a * c)
```
If a counter example to a property is found, QuickCheck reports it.

While integer operations distribute as expected, floating point math is notoriously strange. Let's use test the distributive property for **Floats**.

```haskell
prop_Distributive :: Float -> Float -> Float -> Bool
prop_Distributive a b c =
  a * (b + c) == (a * b) + (a * c)
```

Demo> quickCheck prop_Distributive
Falsifiable, after 7 tests:
3.0
−2.666667
3.75
Less trivial cases require more care.

For the next several slides we will consider a function which inserts an element into an ordered list.

\[
\text{insert } e \ (x:xs) = \\
\quad \text{if } e < x \text{ then } e:x:xs \text{ else } x:(\text{insert } e \ xs) \\
\text{insert } e \ [] = [e]
\]

- \text{insert} is polymorphic. However, we can only use QuickCheck on monomorphic types. Therefore we (arbitrarily) choose to test with \text{Ints}.
- Inputs to \text{insert} must be ordered. We will need a new operator to express this.
Less trivial cases require more care.

For the next several slides we will consider a function which inserts an element into an ordered list.

\[
\text{insert } e \ (x:xs) = \\
\quad \text{if } e < x \ \text{then } e:x:xs \ \text{else } x: (\text{insert } e \ xs)
\]

\[
\text{insert } e \ [] = [e]
\]

- \text{insert} is polymorphic. However, we can only use QuickCheck on monomorphic types. Therefore we (arbitrarily) choose to test with \textbf{Ints}.

- Inputs to \textbf{insert} must be ordered. We will need a new operator to express this. Why not just encode the constraint in a property?
Naively encoding preconditions leads to poor test coverage.

By using a the classify to determine how many tests are actually examining interesting cases. As expected, very few randomly generated lists are ordered and non-empty.

```
prop_InSNaive' :: Int -> [Int] -> Property
prop_InSNaive' e l =
    classify (ordered l && length l > 0)
    "non-trivial" $
    if ordered l then ordered (insert e l)
    else True
```

Demo> :l demo
Demo> quickCheck prop_InSNaive'
OK, passed 100 tests (18% non-trivial).
Naively encoding preconditions leads to poor test coverage.

By using a the classify to determine how many tests are actually examining interesting cases. As expected, very few randomly generated lists are ordered and non-empty.

```haskell
prop_insNaive' :: Int -> [Int] -> Property
prop_insNaive' e l =
  classify (ordered l && length l > 0) "non-trivial" $
  if ordered l then ordered (insert e l)
  else True
```

Demo> :l demo
Demo> quickCheck prop_insNaive'
OK, passed 100 tests (18% non-trivial).

Note that the return type of our property is now Property.
Conditional laws greatly improve test coverage.

The following property definition will attempt to run 100 test cases, all of which are ordered.

\[
\text{prop_ins} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property}
\]
\[
\text{prop_ins} \ e \ l = \text{ordered} \ l \implies \text{ordered} \ (\text{insert} \ e \ l)
\]

As before, this property returns a value of type Property. This allows the QuickCheck framework to ignore test cases which succeed due to a false precondition.
Conditional laws are not sufficient to capture many interesting behaviors. (1/2)

We can extend the `collect` function to generate a histogram of input lengths.

```haskell
prop_ins' :: Int -> [Int] -> Property
prop_ins' e l = ordered l =>
    collect (length l) $
    ordered (insert e l)
```
Conditional laws are not sufficient to capture many interesting behaviors. (2/2)

This shows we are over-testing the nil case and ignoring lists with more than five elements.

Demo> quickCheck prop_Inlns''
OK, passed 100 tests.
46% 0.
32% 1.
13% 2.
6% 3.
2% 4.
1% 5.
Conditional laws are not sufficient to capture many interesting behaviors. (2/2)

This shows we are over-testing the nil case and ignoring lists with more than five elements.

Demo> quickCheck prop_Ins''
OK, passed 100 tests.
46% 0.
32% 1.
13% 2.
6% 3.
2% 4.
1% 5.

What we really want is a way to quantify over the set of all ordered lists ...
Generators overcome the limitations of conditional laws.

The goal of using generators allow QuickCheck randomly sample the set of all values with some property (e.g. the set of all sorted lists).

Benefits of custom generators:

▶ The programmer can pick the sample space’s range.
▶ The programmer has fine grained control of randomized value distribution.
▶ Generating good cases is more efficient than generating a mix of cases and discarding bad ones.

Drawback of custom generators:

▶ Generators may be harder to implement than tests using conditional laws.
▶ Generators are instance of a type classes. Only one generator can be written for a type.
Generators are built monadically.

Generators are instances of the **Monad** class with the (simplified) concrete representation:

\[
\text{newtype \hspace{1em}} \text{Gen} \ a = \text{Gen} \ (\text{Rand} \rightarrow a)
\]

The types of bind and return suggest we can use them as combinators to build complex generators out of simpler ones:

\[
\text{return} \quad :: \quad a \rightarrow \text{Gen} \ a
\]

\[
(\gg\gg=) \quad :: \quad \text{Gen} \ a \rightarrow (a \rightarrow \text{Gen} \ b) \rightarrow \text{Gen} \ b
\]
Randomly generated values are consumed by `forall`.

The type class `Arbitrary` denotes types for which we can generate random values:

```haskell
class Arbitrary a where
    arbitrary :: Gen a
```

And these values are used in a property by applying `forall`:

```haskell
forall :: (Show a, Testable b) =>
        Gen a -> (a -> b) -> Property
```
Instantiating `Arbitrary` is straightforward for simple test distributions.

Given a function `choose :: (Int, Int) -> Gen Int`, we write:

```haskell
instance Arbitrary Int where
    arbitrary = choose (-42,42)
```

We can use the built-in `LiftM2` monad function to add pairs to the `Arbitrary` classes.

```haskell
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
    arbitrary = liftM2 (,) arbitrary arbitrary
```
The oneof and frequency combinators provide fine control over test distribution.

The oneof combinator randomly selects an element from a list. Elements are weighted equally.

```haskell
data Prof = Steve | Stephanie | Benjamin

instance Arbitrary Prof where
    arbitrary = oneof
        [ return Steve, return Stephanie, return Benjamin ]
```

We could also define Arbitrary [a] using oneof:

```haskell
instance Arbitrary a => Arbitrary [a] where
    arbitrary = oneof
        [ return [], liftM2 (:) arbitrary arbitrary ]
```
The frequency combinators provides finer control of test distributions.

Our previous instantiation of `Arbitrary [a]` created empty lists half the time. To fix this we use frequency:

```haskell
instance Arbitrary a => Arbitrary [a] where
  arbitrary = frequency
  [
    (1, return []),
    (4, liftM2 (: ) arbitrary arbitrary)
  ]
```

We can also instantiate a tree generator:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Arbitrary a =>
  Arbitrary Tree a where
  arbitrary = frequency
  [
    (1, LiftM Leaf arbitrary),
    (2, LiftM2 Branch arbitrary arbitrary)
  ]
```
The frequency combinators provides finer control of test distributions.

Our previous instantiation of `Arbitrary [a]` created empty lists half the time. To fix this we use frequency:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = frequency
  [(1, return []),
   (4, liftM2 (:) arbitrary arbitrary)]
```

We can also instantiate a tree generator:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Arbitrary a => Arbitrary Tree a a where
  arbitrary = frequency
  [(1, LiftM Leaf arbitrary),
   (2, LiftM2 Branch arbitrary arbitrary)]
```

What’s wrong with this?
The simplified Gen definition isn’t powerful enough to ensure finite test cases.

We can ensure generated data structures have finite size by adding an explicit size parameter to Gen. Our definition becomes

```haskell
newtype Gen a = Gen (Int -> Rand -> a)
```

and is used with a new combinator:

```haskell
sized :: (Int -> Gen a) -> Gen a
```
Using sized, we can write a bounded tree generator.

The following tree definition will produce a tree with no more elements than the parameter to arbTree. Note that this parameter is passed in by sized and is a global constant.

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Arbitrary a => Arbitrary Tree a where
  arbitrary = sized arbTree

arbTree 0 = liftM Leaf arbitrary
arbTree n =
  frequency
  [ (1, LiftM Leaf arbitrary),
    (2, LiftM2 Branch
      (arbtree (n `div` 2))
      (arbtree (n `div` 2)) ]
```
We can also generate random functions. (1/2)

The procedure:

1. Type of Gen (a→b) is \texttt{Int} → Rand → a → b
   1.1 This is equivalent to a → \texttt{Int} → Rand → b
   1.2 And a → Gen b
2. It’s not clear we can make a value of one type into a generator for another.
   2.1 However maybe we can use arbitrary \texttt{Ints} to transform generators:

   \texttt{variant :: Int} → Gen a → Gen a

2.2 We can certainly make specific types into \texttt{Ints}:

   \texttt{coarbitrary b = if b}
   \texttt{then variant 1}
   \texttt{else variant 0}
We can also generate random functions. (2/2)

3 In Haskell, the right way to generalize this is with a type class.

```haskell
class Coarbitrary a where
c   coarbitrary :: a => Gen b -> Gen b
```

4 We then define Arbitrary in terms of Coarbitrary (and a helper function to match the types).

```haskell
instance (Coarbitrary a, Arbitrary b) => Arbitrary (a -> b) where
    arbitrary = promote (\a -> coarbitrary a arbitrary)
```
It is possible to check monads using QuickCheck as defined above.

1. Build an “program” of monadic instructions to represent the test case.
2. Pass the program value into the monad and generate a concrete test case.
3. Run the test.
4. return the result.
We can test queues in the the ST monad.

Here \( q_{\text{Builder}} :: \text{queue} \) is pure value representing a queue to test, and \( \text{makeQueue} :: \text{queue} \rightarrow \text{ST impQ} \)

\[
\text{prop Ins} \ q_{\text{Builder}} \ x = \quad \text{RunST( do}
\begin{align*}
q &\leftarrow \text{makeQueue} \ q_{\text{Builder}} \\
() &\leftarrow \text{insert} \ x \ q \\
\text{return} \ (\text{front} \ q \ == \ x)
\end{align*}
\]
Unfortunately, we can’t add conditional laws directly.

The conditional operator, \( \Rightarrow \) is a Property combinator and cannot appear in an ST computation. Therefore we need to use a transformer to turn arbitrary monads into testable Property monads:

\[
\text{run} :: \text{monad m} \Rightarrow m \ a \to \text{PropertyM m a}
\]
Unfortunately, we can’t add conditional laws directly.

The conditional operator, $\implies$ is a Property combinator and cannot appear in an ST computation. Therefore we need to use a transformer to turn arbitrary monads into testable Property monads:

$$\text{run} :: \text{monad } m \implies m \ a \rightarrow \text{PropertyM } m \ a$$

Claessen and Hughes explain how to do this in their 2002 Haskell Workshop paper, “Testing Monadic Code with QuickCheck.”
QuickCheck is a flexible tool for software testing, which leverages cool Haskell features like type classes.
Acknowledgements

- These slides are based in part on slide decks by Benjamin Pierce and Jue Wang.
- QuickCheck was developed by Koen Claessen and John Hughes at Chalmers University of Technology.