Contents

What is QuickCheck?

QuickCheck is a tool for testing Haskell programs automatically. The programmer provides a *specification* of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

Why Should I Use QuickCheck?

A Simple Example

A simple example of a property definition is

```
prop_RevRev xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

To check the property, we load this definition in to hugs and then invoke

```
Main> quickCheck prop_RevRev
OK, passed 100 tests.
```

When a property fails, QuickCheck displays a counter-example. For example, if we define

```
prop_RevId xs = reverse xs == xs
  where types = xs::[Int]
```

then checking it results in

```
Main> quickCheck prop_RevId
Falsifiable, after 1 tests:
[-3,15]
```

Using QuickCheck

To use QuickCheck, you must download the module `QuickCheck.hs`, and preferably the script `quickCheck` also. Import module `QuickCheck` into every module containing specifications or test data generators. You can then test properties by loading the module they are defined in into hugs, and calling

```
quickCheck <property-name>
```

or by running the script

```
> quickCheck <options> <file names>
```

which checks every property defined in the modules given. You can use the same ommand line options as for hugs.

You need not use hugs to check properties: any Haskell 98 implementation ought to suffice. However, the `quickCheck` script assumes that hugs is installed on your system. You will probably need to edit the script to insert the location of runhugs.

How can I tell which property is being tested?

Some versions of hugs display the expression to be evaluated before evaluating it; thus you can see which property is being checked at any time, and which property failed. If your version of hugs does not do so, give `quickCheck` the flag `+names`,

```
> quickCheck +names <options> <file names>
```

which will print each property name before checking it.

What do I do if a test loops or encounters an error?

In this case we know that the property does not hold, but `quickCheck` does not display the counter-example. There is another testing function provided for this situation. Repeat the test using

```
verboseCheck <property-name>
```

which displays each test case before running the test: the last test case displayed is thus the one in which the loop or error arises.

Properties

Properties are expressed as Haskell function definitions, with names beginning with `prop_`. Properties are universally quantified over their parameters, so

```
prop_RevRev xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

means that the equality holds for *all* lists `xs`. Technical note.

Properties must have *monomorphic* types. `Polymorphic' properties, such as the one above, must be restricted to a particular type to be used for testing. It is convenient to do so by stating the types of one or more arguments in a

```
  where types = (x1 :: t1, x2 :: t2, ...)
```

clause. Note that `types` is not a keyword; this is just a local declaration which provides a convenient place to restrict the types of `x1`, `x2` etc.

The result type of a property should be `Bool`, unless the property is defined using other combinators below.

Conditional Properties

Properties may take the form

```
        <condition> ==> <property>
```

For example,

```
ordered xs = and (zipWith (<=) xs (drop 1 xs))
insert x xs = takeWhile (<x) xs++[x]++dropWhile (<x) xs

prop_Insert x xs = ordered xs ==> ordered (insert x xs)
  where types = x::Int
```

Such a property holds if the property after ==> holds whenever the condition does.

Testing discards test cases which do not satisfy the condition. Test case generation continues until 100 cases which do satisfy the condition have been found, or until an overall limit on the number of test cases is reached (to avoid looping if the condition never holds). In this case a message such as

```
Arguments exhausted after 97 tests.
```

indicates that 97 test cases satisfying the condition were found, and that the property held in those 97 cases.

Quantified Properties

Properties may take the form

```
        forAll <generator> $ \<pattern> -> <property>
```

For example,

```
prop_Insert2 x = forAll orderedList $ \xs -> ordered (insert x xs)
  where types = x::Int
```

The first argument of `forAll` is a *test data generator*; by supplying a custom generator, instead of using the default generator for that type, it is possible to control the distribution of test data. In the example, by supplying a custom generator for ordered lists, rather than filtering out test cases which are not ordered, we guarantee that 100 test cases can be generated without reaching the overall limit on test cases. Combinators for defining generators are described below.

Observing Test Case Distribution

It is important to be aware of the distribution of test cases: if test data is not well distributed then conclusions drawn from the test results may be invalid. In particular, the ==> operator can skew the distribution of test data badly, since only test data which satisfies the given condition is used.

QuickCheck provides several ways to observe the distribution of test data. Code for making observations is incorporated into the statement of properties, each time the property is actually tested the observation is made, and the collected observations are then summarised when testing is complete.

Counting Trivial Cases

A property may take the form

```
        <condition> `trivial` <property>
```

For example,

```
prop_Insert x xs = ordered xs ==> null xs `trivial` ordered (insert x xs)
  where types = x::Int
```

Test cases for which the condition is `True` are classified as trivial, and the proportion of trivial test cases in the total is reported. In this example, testing produces

```
Main> quickCheck prop_Insert
OK, passed 100 tests (58% trivial).
```

Classifying Test Cases

A property may take the form

```
        classify <condition> <string>$ <property>
```

For example,

```
prop_Insert x xs =
        ordered xs ==>
                classify (ordered (x:xs)) "at-head"$
                classify (ordered (xs++[x])) "at-tail"$
                ordered (insert x xs)
  where types = x::Int
```

Test cases satisfying the condition are assigned the classification given, and the distribution of classifications is reported after testing. In this case the result is

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
58% at-head, at-tail.
22% at-tail.
4% at-head.
```

Note that a test case may fall into more than one classification.

Collecting Data Values

A property may take the form

```
        collect <expression>$ <property>
```

For example,

```
prop_Insert x xs =
        ordered xs ==> collect (length xs)$
                       ordered (insert x xs)
  where types = x::Int
```

The argument of `collect` is evaluated in each test case, and the distribution of values is reported. The type of this argument must be in class `Show`. In the example above, the output is

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
58% 0.
26% 1.
13% 2.
3% 3.
```

Combining Observations

The observations described here may be combined in any way. All the observations of each test case are combined, and the distribution of these combinations is reported. For example, testing the property

```
prop_Insert x xs =
        ordered xs ==>
                collect (length xs)$
                classify (ordered (x:xs)) "at-head"$
                classify (ordered (xs++[x])) "at-tail"$
                ordered (insert x xs)
  where types = x::Int
```

produces

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
58% 0, at-head, at-tail.
22% 1, at-tail.
13% 2.
4% 1, at-head.
3% 3.
```

from which we see that insertion at the beginning or end of a list has not been tested for lists longer than one element.

Test Data Generators: The Type `Gen`

Test data is produced by *test data generators*. QuickCheck defines default generators for most types, but you can use your own with `forAll`, and will need to define your own generators for any new types you introduce.

Generators have types of the form `Gen a`; this is a generator for values of type `a`. The type `Gen` is a monad, so Haskell's **do**-syntax and standard monadic functions can be used to define generators.

Generators are built up on top of the function

```
choose :: Random a => (a, a) -> Gen a
```

which makes a random choice of a value from an interval, with a uniform distribution. For example, to make a random choice between the elements of a list, use

```
do i<-choose (0,length xs-1)
   return (xs!!i)
```

Choosing Between Alternatives

A generator may take the form

```
        oneof <list of generators>
```

which chooses among the generators in the list with equal probability. For example,

```
oneof [return True, return False]
```

generates a random boolean which is true with probability one half.

We can control the distribution of results using the function

```
        frequency :: [(Int, Gen a)] -> Gen a
```

instead. Frequency chooses a generator from the list randomly, but weights the probability of choosing each alternative by the factor given. For example,

```
frequency [(2,return True), (1,return False)]
```

generates `True` two thirds of the time.

The Size of Test Data

Test data generators have an implicit *size* parameter; `quickCheck` begins by generating small test cases, and gradually increases the size as testing progresses. Different test data generators interpret the size parameter in different ways: some ignore it, while the list generator, for example, interprets it as an upper bound on the length of generated lists. You are free to use it as you wish to control your own test data generators.

You can obtain the value of the size parameter using

```
        sized :: (Int -> Gen a) -> Gen a
```

`sized g` calls `g`, passing it the current size as a parameter. For example, to generate natural numbers in the range 0 to size, use

```
sized $ \n -> choose (0, n)
```

The purpose of size control is to ensure that test cases are large enough to reveal errors, while remaining small enough to test fast. Sometimes the default size control does not achieve this. For example, towards the end of a test run arbitrary lists may have up to 50 elements, so arbitrary lists of lists may have up to 2500, which is too large for efficient testing. In such cases it can be useful to modify the size parameter explicitly. You can do using

```
        resize :: Int -> Gen a -> Gen a
```

`resize n g` invokes generator `g` with size parameter `n`. The size parameter should never be negative. For example, to generate a random matrix it might be appropriate to take the square root of the original size:

```
matrix = sized $ \n -> resize (round (sqrt n)) arbitrary
```

Generating Recursive Data Types

Generators for recursive data types are easy to express using `oneof` or `frequency` to choose between constructors, and Haskell's standard monadic combinators to form a generator for each case. For example, if the type of trees is defined by

```
data Tree = Leaf Int | Branch Tree Tree
```

then a generator for trees might be defined by

```
tree = oneof [liftM Leaf arbitrary,
              liftM2 Branch tree tree]
```

However, there is always a risk that a recursive generator like this may fail to terminate, or produce very large results. To avoid this, recursive generators should always use the size control mechanism. For example,

```
tree = sized tree'
tree' 0 = liftM Leaf arbitrary
tree' n | n>0 =
        oneof [liftM Leaf arbitrary,
               liftM2 Branch subtree subtree]
  where subtree = tree' (n `div` 2)
```

Note that

- We guarantee termination by forcing the result to be a leaf when the size is zero.
- We halve the size at each recursion, so that the size gives an upper bound on the number of nodes in the tree. We are free to interpret the size as we will.
- The fact that we share the subtree generator between the two branches of a `Branch` does not, of course, mean that we generate the same tree in each case.

Useful Generator Combinators

If `g` is a generator for type `t`, then

- `two g` generates a pair of `t`s,
- `three g` generates a triple of `t`s,
- `four g` generates a quadruple of `t`s,
- `vector n g` generates a list of `n` `t`s.

If `xs` is a list, then `elements xs` generates an arbitrary element of `xs`.

Class `Arbitrary`

QuickCheck uses Haskell's overloading mechanism to define a default test data generator for each type. This is done using the class

```
class Arbitrary a where
  arbitrary   :: Gen a
  coarbitrary :: a -> Gen b -> Gen b
```

QuickCheck defines instances for the types `()`, `Bool`, `Int`, `Integer`, `Float`, `Double`, pairs, triples, quadruples, lists, and functions.

The class method `arbitrary` is the default generator for type `a`. You can provide a default generator for any other type by declaring an instance of class `Arbitrary` that implements the `arbitrary` method.

Class method `coarbitrary` is used to generate random function values: the implementation of `arbitrary` for a type `a->b` uses `coarbitrary` for type `a`. If you only want to generate random values of a type, you need only define method `arbitrary` for that type, while if you want to generate random functions over the type also, then you should define both class methods.

The `coarbitrary` method interprets a value of type `a` as a *generator transformer*. It should be defined so that different values are interpreted as independent generator transformers. These can be programmed using the function

```
        variant :: Int -> Gen a -> Gen a
```

For different natural numbers `i` and `j`, `variant i g` and `variant j g` are independent generator transformers. The argument of `variant` must be non-negative, and, for efficiency, should be small. Instances of `coarbitrary` can be defined by composing together generator transformers constructed with `variant`.

For example, if the type `Tree` is defined by

```
data Tree = Leaf Int | Branch Tree Tree
```

then a suitable instance of `Arbitrary` could be defined by

```
instance Arbitrary Tree where
  arbitrary = sized tree'
    where tree' 0 = liftM Leaf arbitrary
          tree' n | n>0 =
                oneof [liftM Leaf arbitrary,
                  liftM2 Branch subtree subtree]
            where subtree = tree' (n `div` 2)
  coarbitrary (Leaf n) =
        variant 0 . coarbitrary n
  coarbitrary (Branch t1 t2) =
        variant 1 . coarbitrary t1 . coarbitrary t2
```

Properties of Functions

QuickCheck can generate random function values, and thereby check properties of functions. For example, we can check associativity of function composition as follows:

```
prop_ComposeAssoc f g h x =
  ((f . g) . h) x == (f . (g . h)) x
  where types = [f, g, h] :: [Int->Int]
```

However, before we can *test* such a property, we must see to it that function values can be printed (in case a counter-example is found). That is, function types must be instances of class `Show`. To arrange this, you must import module `ShowFunctions` into every module containing higher-order properties of this kind. If a counter-example is found, function values will be displayed as `"<function>"`. Problems Can Arise

---

Tip: Using newtype

QuickCheck makes it easy to associate a test data generator with each type, but sometimes you will want a different distribution. For example, suppose you are testing a program which manipulates syntax trees, with a constructor for variables:

```
data Expr = Var String | ...
```

Although the variable names are represented as strings, it's unlikely that teh default test data generator for strings will produce a good distribution for variable names. For example, if you're generating random expressions, you may want name clashes to occur sometimes in your test data, but two randomly generated strings (such as "p}v(\231\156A.") are very unlikely to clash.

Of course, you can write a custom test data generator for variable names, maybe choosing randomly from a small set, and try to remember to use it wherever a string plays the role of a name. But this is error prone. Much better is to define a new *type* of names, isomorphic to String, and make your custom generator the default for it. For example,

```
newtype Name = Name String

instance Arbitrary Name where
  arbitrary = oneof ["a", "b", "c", "d", "e"]
```

If you are careful to use the type Name wherever you *mean* a name, then your properties will be both easier to write and more often correct!