Testing

CIS 194 Week 09
25 March 2015

Suggested reading:

- **RWH Chapter 11: Testing and Quality Assurance**, but note that the QuickCheck library's interface has evolved a bit since this was written

```
{-# LANGUAGE RankNTypes #-}
import Test.QuickCheck
import Test.HUnit
import Data.List
import Control.Monad
```

Unit tests

Let's say I want to merge two sorted lists.

```
-- | Assuming the input lists are sorted, combine the lists into a
-- sorted output.
merge1 :: Ord a => [a] -> [a] -> [a]
merge1 (x:xs) (y:ys)
  | x < y     = x : merge1 xs ys
  | otherwise = y : merge1 xs ys
merge1 _      _      = []
```

Does this function work as expected? I could run a few tries in GHCi, but that's a little unsatisfactory: I have to do the work to think up a test, but I get to use it only once. Instead, it's much better to write the test in my file, and that way I can re-run it every time I update my merge function.

The technique described above is often referred to as *unit testing* and is used exensively in the real world. But is unit testing even that great? Sure you can re-run all your tests whenever you want, but that doesn't get around the issue that you actually have to *write* all of the tests in the first place. Coming up with specific test cases is often tedious, repetitive, and *arbitrary*.

Can we do better?

Property-based testing

Writing test cases is boring. And, it's easy to miss out on unexpected behavior. Much better (and, more along the lines of *wholemeal programming*) is to define *properties* we wish our function to have. Then, we can get the computer to generate the test cases for us.

QuickCheck is the standard Haskell library for property-based testing. The idea is that you define a so-called *property*, which is then tested using pseudo-random data.

For example:

```
prop_numElements_merge :: [Integer] -> [Integer] -> Bool
prop_numElements_merge xs ys
```

```
  = length xs + length ys == length (merge1 xs ys)
```

This property is saying that the sum of the lengths of the input lists should be the same as the length of the output list. (It is customary to begin property names with `prop_`.) Let's try it!

```
*Main> quickCheck prop_numElements_merge
*** Failed! Falsifiable (after 5 tests and 4 shrinks):
[]
[0]
```

(Your results may differ slightly. Remember: it's using randomness.)

The first thing we notice is that our function is clearly wrong, with lots of stars and even an exclamation point! We then see that QuickCheck got through 5 tests before discovering the failing test case, so our function isn't terrible. QuickCheck tells us what the failing arguments are: `[]` and `[0]`. Indeed GHCi tells us that `merge1 [] [0]` is `[]`, which is wrong.

What's so nice here is that QuickCheck found us such a nice, small test case to show that our function is wrong. The way it can do so is that it uses a technique called *shrinking*. After QuickCheck finds a test case that causes failure, it tries successively smaller and smaller arguments (according to a customizable definition of "smaller") that keep failing. QuickCheck then reports only the smallest failing test case. This is wonderful, because otherwise the test cases QuickCheck produces would be unwieldy and hard to reason about.

A final note about this property is that the type signature tells us that the property takes lists of integers, not any type `a`. This is so that GHC doesn't choose a silly type to test on, such as `()`. We must always be careful about this when writing properties of polymorphic functions. Numbers are almost always a good choice.

Implications

In order to make our properties more universal, let's generalize our test over implementations of our merging operation. You likely will not need to do this in your own code, but it is useful for us to be able to reuse properties for multiple implementations of the merge function.

```
type MergeFun = [Integer] -> [Integer] -> [Integer]

prop_numElements :: MergeFun -> [Integer] -> [Integer] -> Bool
prop_numElements merge xs ys
  = length xs + length ys == length (merge xs ys)
```

And, we take another stab at our function:

```
merge2 :: MergeFun
merge2 all_xs@(x:xs) all_ys@(y:ys)
  | x < y     = x : merge4 xs all_ys
  | otherwise = y : merge4 all_xs ys
merge2 xs           ys            = xs ++ ys
```

```
*Main> quickCheck (prop_numElements merge4)
+++ OK, passed 100 tests.
```

Huzzah!

Is that it? Are we done? Not quite. Let's try another property:

```
prop_sorted1 :: MergeFun -> [Integer] -> [Integer] -> Bool
prop_sorted1 merge xs ys
  = merge xs ys == sort (xs ++ ys)
```

```
*Main> quickCheck (prop_sorted1 merge4)
*** Failed! Falsifiable (after 4 tests and 3 shrinks):
[]
[1,0]
```

Drat. QuickCheck quite reasonably tried the list `[1,0]` as an input to our function. Of course, this isn't going to work because it's not already sorted. We need to specify an implication property:

```
prop_sorted2 :: MergeFun -> [Integer] -> [Integer] -> Property
prop_sorted2 merge xs ys
  = isSorted xs && isSorted ys ==> merge xs ys == sort (xs ++ ys)

isSorted :: Ord a => [a] -> Bool
isSorted (a:b:rest) = a <= b && isSorted (b:rest)
isSorted _          = True    -- must be fewer than 2 elements
```

In `prop_sorted`, we see the use of the operator `(==>)`. Its type is `Testable prop => Bool -> prop -> Property`. It takes a `Bool` and a `Testable` thing and produces a `Property`. Note how `prop_sorted` returns a `Property`, not a `Bool`. We'll sort these types out fully later, but I wanted to draw your attention to the appearance of `Property` there.

Let's see how this works:

```
*Main> quickCheck (prop_sorted2 merge2)
*** Gave up! Passed only 21 tests.
```

(And that took maybe 20 seconds.) There aren't any failures, but there aren't a lot of successes either. The problem is that QuickCheck will run the test only when both randomly-generated lists are in sorted order. The odds that a randomly-generated list of length `n` is sorted is `1/n!`, which is generally quite small odds. And we need *two* sorted lists. This isn't going to work out well.

QuickCheck's types

How does QuickCheck generate the arbitrary test cases, anyway? It uses the `Arbitrary` class:

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

We'll leave `shrink` to the online documentation and focus on `arbitrary`. The `arbitrary` method gives us a `Gen a` – a generator for the type `a`. Of course, the `arbitrary` method for lists doesn't care about ordering (indeed, it can't, due to parametricity), but we do. Luckily, this is a common problem, and QuickCheck offers a solution in the form of `OrderedList`, a wrapper around lists that have the right `Arbitrary` instance for our needs:

```
newtype OrderedList a = Ordered { getOrdered :: [a] }
instance (Ord a, Arbitrary a) => Arbitrary (OrderedList a) where ...
```

(`newtype` is almost just like `data`. Poke around online for more info.)

Now, let's rewrite our property:

```
prop_sorted3 :: MergeFun
             -> OrderedList Integer -> OrderedList Integer -> Bool
prop_sorted3 merge (Ordered xs) (Ordered ys)
  = merge xs ys == sort (xs ++ ys)
```

```
*Main> quickCheck (prop_sorted3 merge2)
+++ OK, passed 100 tests.
```

Huzzah! Just by changing the types a bit, we can affect instance selection to get what we want.

Yet, this all seems like black magic. How does QuickCheck do it? Let's look more in depth at the types.

```
quickCheck :: Testable prop => prop -> IO ()

class Testable prop where ...
instance Testable Bool where ...
instance Testable Property where ...
instance (Arbitrary a, Show a, Testable prop) => Testable (a -> prop) where ...
```

We can `quickCheck` anything that's `Testable`. Boolean values are `Testable`, as are the somewhat mysterious `Property`s. But it's the last instance listed here of `Testable` that piques our curiosity. It says that a *function* is `Testable` as long as its argument has an `arbitrary` method, the argument can be printed (in case of failure), and the result is `Testable`.

Is `[Integer] -> [Integer] -> Bool` `Testable`? Sure it is. Recall that `[Integer] -> [Integer] -> Bool` is equivalent to `[Integer] -> ([Integer] -> Bool)`. Because `[Integer]` has both an `Arbitrary` instance and a `Show` instance, we can use the last instance above as long as `[Integer] -> Bool` is `Testable`. And that's `Testable` because we (still) have an `Arbitrary` and a `Show` instance for `[Integer]`, and `Bool` is `Testable`. So, that's how `quickCheck` works – it uses the `Arbitrary` instances for the argument types. And, that's how changing the argument types to `OrderedList Integer` got us the result we wanted.

Generating arbitrary data

When you want to use QuickCheck over your own datatypes, it is necessary to write an `Arbitrary` instance for them. Here, we'll learn how to do so.

Let's say we have a custom list type

```haskell
data MyList a = Nil | Cons a (MyList a)

instance Show a => Show (MyList a) where
  show = show . toList

toList :: MyList a -> [a]
toList Nil            = []
toList (a `Cons` as) = a : toList as

fromList :: [a] -> MyList a
fromList []     = Nil
fromList (a:as) = a `Cons` fromList as
```

If we want an `Arbitrary` instance, we must define the `arbitrary` method, of type `Gen (MyList a)`. Luckily for us, `Gen` is a monad (did you see that coming?), so some of its details are already familiar. We also realize that if we want arbitrary lists of `a`, we'll need to make arbitrary `a`s. So, our instance looks like

```haskell
instance Arbitrary a => Arbitrary (MyList a) where
  arbitrary = genMyList1
```

At this point, it's helpful to check out the combinators available in the "Generator combinators" section of the **QuickCheck documentation**.

At this point, it's helpful to think about how you, as a human, would generate an arbitrary list. One way to do it is to choose an arbitrary length (say, between 0 and 10), and then choose each element arbitrarily. Here is an implementation:

```haskell
genMyList1 :: Arbitrary a => Gen (MyList a)
genMyList1 = do
  len <- choose (0, 10)
  vals <- replicateM len arbitrary
  return $ fromList vals
```

Let's try it out:

```
*Main> sample genMyList1
[(),(),(),(),(),()]
[]
[(),(),(),(),(),(),(),(),()]
[(),(),(),(),(),()]
[(),(),(),(),(),(),(),(),()]
[()]
[(),(),(),(),(),(),(),()]
```

```
[(),(),(),(),(),(),(),(),()]
[(),(),()]
[(),(),(),(),(),()]
[(),(),(),(),(),(),(),(),(),()]
```

The arbitrary lengths are working, but the element generation sure is boring. Let's use a type annotation to spruce things up (and override GHC's default choice of `()`)!

```
*Main> sample (genMyList1 :: Gen (MyList Integer))
[0,0,0,0,0,0,0,0,0,0]
[]
[-2,3,1,0,4,-1]
[-5,0,2,1,-1,-3]
[-5,-6,-7,-2,-8,7,-3,4,-6]
[4,-3,-3,2,-9,9]
[]
[10,-1]
[9,-7,-16,3,15]
[0,14,-1,0]
[3,18,-13,-17,-20,-8]
```

That's better.

This generation still isn't great, though, because perhaps a function written over `MyList`s fails only for lists longer than 10. We'd like unbounded lengths. Here's one way to do it:

```haskell
genMyList2 :: Arbitrary a => Gen (MyList a)
genMyList2 = do
  make_nil <- arbitrary
  if make_nil    -- type inference tells GHC that make_nil should be a Bool
     then return Nil
     else do
       x <- arbitrary
       xs <- genMyList2
       return (x `Cons` xs)
```

```
*Main> sample (genMyList2 :: Gen (MyList Integer))
[0,0,0,0,0,0,0]
[]
[3,-3]
[]
[]
[-1,-1]
[-10]
[]
[]
[11]
[-20,-14]
```

The lengths are unbounded (you'll just have to trust me there), but we're getting a *lot* of empty lists. This is because at every link in the list, there's a 50% chance of producing `Nil`. That means that a list of length `n` will appear only one out of `2^n` times. So, lengths are unbounded, but very unlikely.

The way to make progress here is to use the `sized` combinator. QuickCheck is set up to try "simple" arbitrary things before "complex" arbitrary things. The way it does this is using a size parameter, internal to the `Gen` monad. The more generating QuickCheck does, the higher this parameter gets. We want to use the size parameter to do our generation.

Let's look at the type of `sized`:

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

An example is the best way of explaining how this works:

```
genMyList3 :: Arbitrary a => Gen (MyList a)
genMyList3 = sized $ \size -> do
  len <- choose (0, size)
  vals <- replicateM len arbitrary
  return $ fromList vals
```

```
*Main> sample (genMyList3 :: Gen (MyList Integer))
[]
[-2]
[-1,3,4]
[-4,-2,1,-1]
[]
[]
[12,3,11,0,3,-12,10,5,11,12]
[-4,-8,-9,2,14,5,8,11,-1,7,11,-8,2,-6]
[6,10,-5,15,6]
[-3,-18,-4]
[9,19,13,-19]
```

That worked nicely – the lists tend to get longer the later they appear. The idea is that `sized` takes a *continuation*: the thing to do with the size parameter. We just use a lambda function as the one argument to `sized`, where the lambda binds the `size` parameter, and then we can use it internally. If that's too painful (say we just want to produce the size parameter, without using a continuation), you could always do something like this:

```
getSize :: Gen Int
getSize = sized return
```

I'll leave it to you to figure out how that works. Follow the types!

As one last example, we can also choose arbitrary generators from a list based on frequency. Although the length method of `genMyList3` works well for lists, the following technique is much better for trees:

```
genMyList4 :: Arbitrary a => Gen (MyList a)
genMyList4 = sized $ \size -> do
  frequency [ (1, return Nil)
            , (size, do x <- arbitrary
                        xs <- resize (size - 1) genMyList4
                        return (x `Cons` xs) )]
```

```
*Main> sample (genMyList4 :: Gen (MyList Integer))
[]
[2,0]
[4,0,-1]
[-6,-2,3,-1,-1]
[6,6]
[2,2,6,6]
[-6,-9,5,-5]
[8,7,5,7,7,-1,-2,-1,-5,-3]
[15,-12,14,13,-5,-10,-9,-8,-2]
[12,-11,-8,6,-6,-4,11,11]
[-7,1,-3,4,-3,-9,4,6,-2,10,-9,-7,5,7,1]
```

Let's look at the type of `frequency`:

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

It takes a list of `(Int, Gen a)` pairs and produces a `Gen a`. The numbers in the list give the likelihood of choosing that element. Above, we fixed the frequency of `Nil` at 1, but let the likelihood of `Cons` vary according to the desired size. Then, in the recursive call to `genMyList4`, we used `resize` to lower the `size` parameter. Otherwise, it's too likely to get a runaway list that goes on toward infinity.

```
Generated 2015-03-24 21:45:39.534479
```

Powered by **shake**, **hakyll**, **pandoc**, **diagrams**, and **lhs2TeX**.