Testing in Haskell:

an introduction to HUnit and QuickCheck

Mark P Jones

Portland State University

Testing, Testing, Testing, ...

Testing:

- Testing can confirm expectations about how things work
- Conversely, testing can set expectations about how things should work
- It can be dangerous to generalize from tests

  "Testing can be used to show the presence of bugs, but never to show their absence" [Edsger Dijkstra, 1969]
- But testing does help us to find & avoid:
  - Bugs in the things we build
  - Bugs in the claims we make about those things

Example: filter

```
filter :: (a -> Bool) -> [a] -> [a]

filter even [1..10] = [2,4,6,8,10]

filter (<5) [1..100] = [1,2,3,4]

filter (<5) [100,99..1] = [4,3,2,1]
```

Making Tests Executable:

```
test1 = filter even [1..10] == [2,4,6,8,10]

test2 = filter (<5) [1..100] == [1,2,3,4]

test3 = filter (<5) [100,99..1] == [4,3,2,1]
```

Making Tests Executable:

```
test1 = filter even [1..10] == [2,4,6,8,10]

test2 = filter (<5) [1..100] == [1,2,3,4]

test3 = filter (<5) [100,99..1] == [4,3,2,1]

tests = test1 && test2 && test3
```

Pros:

- Tests are simple functional programs
- Tests are self-checking

Cons:

- Have to run tests manually
- Testing stops as soon as one test fails
- No indication of which test failed
- No summary statistics (e.g., # tests run)
- Harder to handle complex behavior (e.g., testing code that performs I/O actions, raises an exception, …)

Unit Testing in Haskell

Enter HUnit:

- A library for unit testing
- Written in Haskell
- Available from http://hunit.sourceforge.net
- (Or from http://hackage.haskell.org)

- Built-in to recent versions of Hugs and GHC

- Just “import Test.HUnit” and you’re ready!

Defining Tests:

```
import Test.HUnit

test1 = TestCase (assertEqual
                    "filter even [1..10]"
                    (filter even [1..10])
                    [2,4,6,8,10])
test2 = …
test3 = …
tests = TestList [test1, test2, test3]
```

Running Tests:

```
Main> runTestTT tests
Cases: 3  Tried: 3  Errors: 0  Failures: 0

Main>
```

Detecting Faults:

```
import Test.HUnit

test1 = TestCase (assertEqual
                    "filter even [1..10]"
                    (filter even [1..10])
                    [2,4,6,9,10])
test2 = …
test3 = …
tests = TestList [test1, test2, test3]
```

Using HUnit:

```
Main> runTestTT tests
### Failure in: 0
filter even [1..10]
expected: [2,4,6,8,10]
 but got: [2,4,6,9,10]
Cases: 3  Tried: 3  Errors: 0  Failures: 1

Main>
```

Labeling Tests:

```
…

tests = TestLabel "filter tests"
          $ TestList [test1, test2, test3]
```

Using HUnit:

```
Main> runTestTT tests
### Failure in: filter tests:0
filter even [1..10]
expected: [2,4,6,8,10]
 but got: [2,4,6,9,10]
Cases: 3  Tried: 3  Errors: 0  Failures: 1

Main>
```

The Test and Assertion Types:

```
data Test       = TestCase Assertion
                    | TestList [Test]
                    | TestLabel String Test

runTestTT     :: Test -> IO Counts

assertFailure :: String -> Assertion
assertBool    :: String -> Bool -> Assertion
assertEqual   :: (Eq a, Show a) =>
                          String -> a -> a ->
   Assertion
```

Problems:

- Finding and running tests is a manual process (easily skipped/overlooked)
- It can be hard to trim tests from distributed code
- We still can’t solve the halting problem ☺

Example: merge

Let’s develop a merge function for combining two sorted lists into a single sorted list:

```
merge :: [Int] -> [Int] -> [Int]
merge = undefined
```

What about test cases?

Merge Tests:

- Simple examples:

  merge [1,5,9] [2,3,6,10] == [1,2,3,5,6,9,10]

- One or both arguments empty:

  merge [] [1,2,3] == [1,2,3]
  merge [1,2,3] [] == [1,2,3]

- Duplicate elements:

  merge [2] [1,2,3] == [1,2,3]
  merge [1,2,3] [2] == [1,2,3]

Capturing the Tests:

```
mergeTests
   = TestLabel "merge tests”
   $ TestList [simpleTests, emptyTests, dupTests]

simpleTests
   = TestLabel "simple tests”
   $ TestCase (assertEqual "merge [1,5,9] [2,3,6,10]"
                           (merge [1,5,9] [2,3,6,10])
                           [1,2,3,5,6,9,10])

emptyTests
   = ...
```

Capturing the Tests:

```
Main> runTestTT mergeTests
Cases: 6  Tried: 0  Errors: 0  Failures: 0
Program error: Prelude.undefined

Main>
```

Refining the Definition (1):

Let's provide a little more definition for merge:

```
merge          :: [Int] -> [Int] -> [Int]
merge xs ys = []
```

What happens to the test cases now?

Back to the Tests:

```
Main> runTestTT mergeTests
### Failure in: merge tests:0:simple tests
merge [1,5,9] [2,3,6,10]
expected: []
 but got: [1,2,3,5,6,9,10]
…
Cases: 6  Tried: 6  Errors: 0  Failures: 5

Main>
```

Refining the Definition (2):

Let’s provide a little more definition for merge:

```
merge          :: [Int] -> [Int] -> [Int]
merge xs ys = xs
```

What happens to the test cases now?

Back to the Tests:

```
Main> runTestTT mergeTests
### Failure in: merge tests:0:simple tests
merge [1,5,9] [2,3,6,10]
expected: [1,5,9]
 but got: [1,2,3,5,6,9,10]
### Failure in: merge tests:2:duplicate elements:0
merge [2] [1,2,3]
expected: [2]
 but got: [1,2,3]
Cases: 6  Tried: 6  Errors: 0  Failures: 2

Main>
```

Refining the Definition (3):

Use type information to break the definition down into multiple cases:

```
merge              :: [Int] -> [Int] -> [Int]
merge []       ys  = ys
merge (x:xs) ys = ys
```

Refining the Definition (4):

Repeat …

```
merge               :: [Int] -> [Int] -> [Int]
merge []      ys  = ys
merge (x:xs) []  = x:xs
merge (x:xs) (y:ys)
                  = x:xs
```

Refining the Definition (5):

Use guards to split into cases:

```
merge               :: [Int] -> [Int] -> [Int]
merge []       ys   = ys
merge (x:xs) []     = x:xs
merge (x:xs) (y:ys)
            | x<y = x : merge xs (y:ys)
            | otherwise = y : merge (x:xs) ys
```

Back to the Tests:

```
Main> runTestTT mergeTests
### Failure in: merge tests:2:duplicate elements:0
merge [2] [1,2,3]
expected: [1,2,2,3]
 but got: [1,2,3]
### Failure in: merge tests:2:duplicate elements:1
merge [1,2,3] [2]
expected: [1,2,2,3]
 but got: [1,2,3]
Cases: 6  Tried: 6  Errors: 0  Failures: 2

Main>
```

Refining the Definition (6):

Use another guards to add another case:

```
merge                 :: [Int] -> [Int] -> [Int]
merge []       ys     = ys
merge (x:xs) []       = x:xs
merge (x:xs) (y:ys)
          | x<y  = x : merge xs (y:ys)
          | y<x  = y : merge (x:xs) ys
          | x==y        = x : merge xs ys
```

Back to the Tests:

```
Main> runTestTT mergeTests
Cases: 6  Tried: 6  Errors: 0  Failures: 0

Main>
```

Modifying the Definition:

Suppose we decide to modify the definition:

```
merge               :: [Int] -> [Int] -> [Int]
merge (x:xs) (y:ys)
         | x<y  = x : merge xs (y:ys)
         | y<x  = y : merge (x:xs) ys
         | x==y      = x : merge xs ys
merge xs     ys   = xs ++ ys
```

Is this still a valid definition?

Back to the Tests:

```
Main> runTestTT mergeTests
Cases: 6  Tried: 6  Errors: 0  Failures: 0

Main>
```

Lessons Learned:

- Writing tests (even before we've written the code we want to test) can expose key details / design decisions
- A library like HUnit can help to automate the process (at least partially)
- Development alternates between coding and testing
- Bugs are expensive, running tests is cheap
- Good tests can last a long time; continuing use as code evolves

Testing Laws with QuickCheck

Lawful Programming:

How can we give useful information about a function without necessarily having to give all the details of its definition?

- Informal description:

  "map applies its first argument to every element in its second argument …"

- Type signature:

  ```
  map :: (a -> b) -> [a] -> [b]
  ```

- Laws:
  - Normally in the form of equalities between expressions …

Algebra of Lists:

- (++) is associative with unit []

  xs ++ (ys ++ zs) = (xs ++ ys) ++ zs

  [] ++ xs = xs = xs ++ []

- map preserves identities, distributes over composition and concatenation:

  map id = id

  map (f . g) = map f . map g

  map f (xs ++ ys) = map f xs ++ map f ys

… continued:

- filter distributes over concatenation

  filter p (xs ++ ys) = filter p xs ++ filter p ys

- filter and map:

  filter p . map f = map f . filter (p . f)

- composing filters:

  filter p . filter q = filter r

    where r x = q x && p x

Uses for Laws:

Laws can be used:

- To capture/document deep intuitions about program behavior
- To support reasoning about program behavior
- To optimize or transform programs (either by hand, or in a compiler)
- As properties to be tested
- As properties to be proved

Wanted! Reward!

- However: In the short-term, programmers don't see any reward for writing laws …

- … so they won't write them.

- If programmers can derive some benefit from writing laws, then perhaps they will do it …

Laws for Merge:

What laws might we formulate for merge?

- If xs and ys are sorted, then merge xs ys is sorted
- merge (sort xs) (sort ys) should be sorted
- merge xs ys == merge ys xs
- merge xs xs == xs
- …

From Laws to Functions:

```
mergeProp1        :: [Int] -> [Int] -> Bool
mergeProp1 xs ys = sorted xs ==>
                     sorted ys ==>
                       sorted (merge xs ys)

(==>)          :: Bool -> Bool -> Bool
x ==> y        = not x || y

sorted      :: [Int] -> Bool
sorted xs = and [ x <= y | (x,y) <- zip xs (tail xs) ]
```

Testing mergeProp1:

```
Main> mergeProp1 [1,4,7] [2,4,6]
True
Main> mergeProp1 [1,4,7] [2,4,1]
True
Main> sorted [1,4,7]
True
Main> sorted [2,4,1]
False
Main>
```

Question: to test merge, I wrote more code …

If I don't trust my programming skills, why am I writing even more (untrustworthy) code?

Formulate More Tests!

```
import List(sort)

sortSorts     :: [Int] -> Bool
sortSorts xs = sorted (sort xs)

sortedEmpty :: Bool
sortedEmpty = sorted []

sortIdempotent     :: [Int] -> Bool
sortIdempotent xs = sort (sort xs) == sort xs
```

More Laws to Functions:

```
mergePreservesOrder  :: [Int] -> [Int] -> Bool
mergePreservesOrder xs ys
   = sorted (merge (sort xs) (sort ys))

mergeCommutes :: [Int] -> [Int] -> Bool
mergeCommutes xs ys
  = merge us vs == merge vs us
    where us = sort xs
          vs = sort ys
```

etc...

Testing mergeProp1:

```
Main> mergeCommutes [1,4,7] [2,4,6]
True
Main> mergeCommutes [1,4,7] [2,4,1]
True
Main> mergePreservesOrder [1,4,7] [2,4,6]
True
Main> mergePreservesOrder [1,4,7] [2,4,1]
True
Main>
```

Automated Testing:

- Of course, we can run as many individual test cases as we like:
  - Pick a test case
  - Execute the program
  - Compare actual result with expected result
- Wouldn't it be nice if the environment could help us to go directly from properties to tests?
- Wouldn't it be nice if the environment could run the tests for us automatically too?

QuickCheck:

- This is a job for QuickCheck!
- "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" by Koen Claessen and John Hughes, Chalmers University, Sweden. (Published at ICFP 2000)
- In GHC/Hugs: import Test.QuickCheck

Lawful Programming:

```
reverse    :: [a] -> [a]
reverse xs = …


{- reverse satisfies the following:
     reverse (xs ++ ys)
        ==
     reverse ys ++ reverse xs
-}
```

Lawful Programming:

```
reverse    :: [a] -> [a]
reverse xs = …

prop_RevApp xs ys
   = reverse (xs++ys)
        ==
     reverse ys ++ reverse xs
```

Laws are type checked as part of the main program source text

If the laws and the code are inconsistent, then an error will be detected!

Running QuickCheck:

Prelude> :load reverse.hs

Main> reverse [1,2,3]
[3,2,1]

Main> quickCheck prop_RevApp
OK, passed 100 tests
Main>

Not All Laws are True:

```
Main> quickCheck (\b -> b == not b)
Falsifiable, after 0 tests:
True

Main>
```

- Sometimes this points to a bug in the program.
- Sometimes this points to a bug in the law.

The Testable Class:

```
quickCheck :: Testable a => a -> IO a

instance Testable Bool where …

instance (Arbitrary a,
          Show a,
          Testable b)=> Testable (a -> b)
  where …
```

Indicates an ability to generate arbitrary values of type a.

The Testable Class:

quickCheck :: Testable a => a -> IO a

instance Testable Bool **where** …

instance (Arbitrary a,
Show a,
Testable b)=> Testable (a -> b)
where …

Indicates an ability to display arguments for counter examples

Generating Arbitrary Values:

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

arbitrary is a generator of random values

```
instance Arbitrary ()
instance Arbitrary Bool
instance Arbitrary Int
instance Arbitrary Integer
instance Arbitrary Float
instance Arbitrary Double
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b)
instance Arbitrary a => Arbitrary [a]
```

Quantified or Parameterized?

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

Main> quickCheck (prop_revApp [1,2,3])
OK, passed 100 tests.

Main>
```

- If you don’t give a specific value for an argument, quickCheck will generate arbitrary (i.e. random) values for you.

QuickCheck-ing merge:

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

Main> quickCheck mergePreservesOrder
OK, passed 100 tests.

Main>
```

So far, so good ...

Continued ...

```
mergeProp1         :: [Int] -> [Int] -> Bool
mergeProp1 xs ys = sorted xs ==>
                     sorted ys ==>
                       sorted (merge xs ys)
```

What happens?

```
Main> quickCheck mergeProp1
Falsifiable, after 7 tests:
[-1,-5,5,4,3,-5]
[5,-6,2,6,-6,0]

Main>
```

Huh?

What went wrong?

```
Main> sorted [-1,-5,5,4,3,-5]
False
Main> sorted [5,-6,2,6,-6,0]
False
Main> sorted (merge [-1,-5,5,4,3,-5] [5,-6,2,6,-6,0])
False
Main> False ==> False ==> False
False
Main> False ==> (False ==> False)
True
Main>
```

A Fix! (in fact, infix)

```
infixr ==>
(==>)         :: Bool -> Bool -> Bool
x ==> y       = not x || y
```

What happens?

```
Main> quickCheck mergeProp1
OK, passed 100 tests.

Main>
```

Hooray!!!

Are we Happy Now?

```
mergeProp1          :: [Int] -> [Int] -> Bool
mergeProp1 xs ys = sorted xs ==>
                      sorted ys ==>
                         sorted (merge xs ys)
```

100 tests passed!

But how many of them were trivial (i.e., one or both arguments unsorted)?

Understanding Test Results:

- Use the collect combinator:

```
mergeProp1sorted xs ys
 = collect (sorted xs, sorted ys) (mergeProp1 xs ys)
```

- Testing:

```
Main> quickCheck mergeProp1sorted
OK, passed 100 tests.
45% (False,False).
25% (True,True).
20% (True,False).
10% (False,True).

Main>
```

Understanding Test Results:

- Or use the classify combinator:

```
mergeProp1long xs ys
  = classify (length xs > 10) "long"
  $ classify (length xs <= 5) "short"
  $ mergeProp1 xs ys
```

- Testing:

```
Main> quickCheck mergeProp1long
OK, passed 100 tests.
49% short.
29% long.

Main>
```

Understanding ==>:

- The real (==>) operator is not a standard "implies" function of type Bool -> Bool -> Bool
- When we test a property p ==> q, QuickCheck will try to find 100 test cases for which p is true, and will test q in each of those 100 cases
- If it tries 1000 candidates without finding enough solutions, then it will give up:

```
Main> quickCheck (\b -> (b == not b) ==> b)
Arguments exhausted after 0 tests.
Main>
```

- QuickCheck can be configured to use different numbers of tests/attempts

Writing Custom Generators:

Instead of generating random values and selecting only some, we can try to generate the ones we want directly:

```
sortedList :: Gen [Int]
sortedList = do ns <- arbitrary
                return (sort ns)
```

More Examples:

Now we can use QuickCheck's forAll combinator to define:

```
prop_mergePreservesOrder = forAll sortedList $ \xs ->
                             forAll sortedList $ \ys ->
                              sorted (merge xs ys)

prop_mergeCommutes       = forAll sortedList $ \xs ->
                             forAll sortedList $ \ys ->
                              merge xs ys == merge
    ys xs

prop_mergeIdempotent     = forAll sortedList $ \xs ->
                             merge xs xs == xs
```

Lessons Learned:

- QuickCheck is a useful and lightweight tool that encourages and rewards the lawful programmer!
- There is a script that automatically runs quickCheck on all of the properties in a file that have names of the form prop_XXX
- Interpreting test results may require some care …
- "Good" (random) test data can be hard to find …