Testing in Haskell: using HUnit

Notes, thanks to 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:

\[
test1 = \text{filter even} \ [1..10] == [2,4,6,8,10]
\]

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

\[
test3 = \text{filter} \ (<5) \ [100,99..1] == [4,3,2,1]
\]
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
  (Or from [http://hackage.haskell.org](http://hackage.haskell.org))

- Built-in to recent versions of Hugs and GHC

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

```haskell
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:

```haskell
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
```

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:

```haskell
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:

\[
\text{merge} :: [\text{Int}] \to [\text{Int}] \to [\text{Int}]
\]

\[
\text{merge} \ 	ext{xs} \ \text{ys} = []
\]

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:

\[
\text{merge} \quad :: \quad [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\text{merge} \; [] \quad ys \; = \; ys
\]

\[
\text{merge} \; (x:xs) \; ys \; = \; ys
\]
Refining the Definition (4):

Repeat …

\[
\begin{align*}
\text{merge} & \quad :: \, [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}] \\
\text{merge } [\,] & \quad ys = ys \\
\text{merge } (x:xs) [\,] & \quad = x:xs \\
\text{merge } (x:xs) (y:ys) & \quad = x:xs
\end{align*}
\]
Refining the Definition (5):

Use guards to split into cases:

\[
\text{merge} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\text{merge} \; [] \; ys = ys
\]

\[
\text{merge} \; (x:xs) \; [] = x:xs
\]

\[
\text{merge} \; (x:xs) \; (y:ys)
\]

\[
\quad | \; x < y = x : \text{merge} \; xs \; (y:ys)
\]

\[
\quad | \; \text{otherwise} = y : \text{merge} \; (x:xs) \; ys
\]
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:

\[
\begin{align*}
\text{merge} & : [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}] \\
\text{merge} \ [\ ] \ y & = y \\
\text{merge} \ (x:xs) \ [\ ] & = x:xs \\
\text{merge} \ (x:xs) \ (y:ys) & \\
& \quad | x < y \ = \ x : \text{merge} \ xs \ (y:ys) \\
& \quad | y < x \ = \ y : \text{merge} \ (x:xs) \ ys \\
& \quad | x == y \ = \ x : \text{merge} \ xs \ ys
\end{align*}
\]
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:

\[
\begin{align*}
\text{merge} & \quad :: \ [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}] \\
\text{merge} (x:xs) (y:ys) & \\
| \quad x < y & = x : \text{merge} \ xs \ (y:ys) \\
| \quad y < x & = y : \text{merge} \ (x:xs) \ ys \\
| \quad x == y & = x : \text{merge} \ xs \ ys \\
\text{merge} \ xs \ y & = xs ++ ys
\end{align*}
\]

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