Property Based Testing with Haskell and QuickCheck
CS 1632 - FINAL DELIVERABLE

https://github.com/blester125/CS_1632_deliverable6

Brian Lester

19 April 2016

1 Summary

During Deliverable 4 I did property based testing for the Arrays.sort() method in Java. This was pretty fun and interesting. One slight annoyance during this project was the fact we had to create the random input ourselves. This was easy but it seemed like something that could be done for us. Luckily Haskell has this in the form of QuickCheck. I have been learning some Haskell in my free time so when the final deliverable was announced to be free form it seemed like a perfect opportunity to learn more about Haskell and to test out the QuickCheck library.

In this deliverable I created some simple QuickChecks of various mathematical properties. After that I create some tests for various sorting methods I have implemented. Finally I created more complex tests that test a Data Structure (Binary Search Tree) that I implemented. All tests passed by the end of development.

This deliverable was a lot of fun, using some online resources I was able to expand my knowledge of Haskell in general (I had never implemented a BST before) and specifically my knowledge of testing with QuickCheck.

The only part about this that I would have concerns about is the fact that I was not able to apply these tests to an actual application. This is mostly do to my inexperience with Haskell. An idea I had was to implement Huffman Encoding in Haskell but I was having trouble applying these tests to the actual Huffman encoding (which I did get working). The only thing I could test was the BST used to back the Huffman tree. I would have liked to figure out how to do this but I unfortunately have to much other course work and am too inexperienced with Haskell. However I did have a lot of fun.

The quality of the Haskell Code I wrote is top notch. There are no failed tests and all the functionality works. The code is ready to be released as a usable Binary Search Tree that could be used to back a set or something similar. The Tree implements the BST invariant and the insertion and deletions maintain this invariant. The tests of the tree prove it's correctness which means that it can be released. The Sorts are also tested and fact that they have the properties of sorts (plus they match the library sort) show that they are correct and can be used to sort things in different conditions.

The code is available at https://github.com/blester125/CS_1632_deliverable6

2 Getting Started with QuickCheck

2.1 The First Test

The first step was obvious to test out QuickCheck. I started with a simple test that tested the properties of reversing lists. This properties is that if you append one list to another and then reverse the whole thing then it is the same as reversing each list and appending the first list to the second one. It would look like this:

$$
\begin{gathered}
x = [1,2,3] \\
y = [4,5,6] \\
\text{reverse } (x + y) = \text{reverse } ([1,2,3,4,5,6]) \\
= [6,5,4,3,2,1] \\
\text{and} \\
x = [1,2,3] \\
y = [4,5,6] \\
\text{reverse}(y) + \text{reverse}(x) \\
[6,5,4] + [3,2,1] \\
= [6,5,4,3,2,1]
\end{gathered}
$$

This example is used in several tutorials for QuickCheck so this seemed like a good starting point. I also include a few other tests to get acclimated to QuickCheck such as the commutative property of addition and the associativity of multiplication and addition.

This code can be found in "Example.hs". To run this code load it into GHCI with ":l Example.hs" and type main.

2.2 QuickCheck and Sorting

Sorting introduced me to property based testing (it makes sense, there are a lot of properties to test with sorting) so it makes sense that sorting would be the next logical step to exploring QuickCheck. For this part I tested the following properties of the sort methods:

1. Idempotent property of the sort
2. Increasing ordering of the final list
3. The minimum of the list is at the head of the list
4. The maximum item is at the end of the list
5. The sorted List is a permutation of the original list (this property covers the property that the number of elements in the input are equal to the number in the output, the property that there are no elements in the input that are not in the output and visa versa)
6. The smallest item in a sorted list composed of appending two lists is the smallest item in both lists.
7. Testing against a model in the form of the build in sort

I wrote several sort methods to test, quickSort (a Haskell Classic), mergeSort, and BubbleSort.

When writing these test the first hurdle I ran into was in the minimum tests. minimum is not defined for empty lists so I had to learn to write a test that will not generate a null test. This was also important in the appended list tests because the use of "head" in the tests

The next hurdle was testing three different sorting functions. I know about higher order functions and currying so I first tried to write one test that I could pass both a list and a sort function to use. I was unable to figure out how to get this to work with QuickCheck. I had to resort to writing different 3 different tests.

This code can be found in "Sorts.hs". To run this code load it into GHCI with " :l Sorts.hs" and type main.

3 More Advanced Checks - Binary Search trees

While the tests for the sort were quite long and I tested more properties than I did in deliverable 4 these tests were not too much more in depth than the tests from that deliverable. To expand on this idea I decided to do property based testing on a Data Structure I implemented. I chose to implement a Binary Search Tree because it has a nice invariant to test. This is the binary search tree property which states that all the entries in the left sub tree is smaller than the entries in the right subtree.

These tester were much harder to write than the sort tests. All the sort test were easy because all QuickCheck had to was generate random numbers. In this case we needed to generate correct Binary Search Trees otherwise things like the Insert test could fail, not because the insert failed but rather because the Binary Search Tree was not valid to begin with.

One problem with these tests is the delete tests. QuickCheck generates random numbers to delete from the tree. Not all of these are necessarily in the tree so the test is on average likely to be less thorough as I would like it to be.

This code can be found in "BST.hs". To run this code load it into GHCI with " :l BST.hs" and type main.