Reliable Binary Search in Haskell

Daniel Turner

15th April 2012

1 Introduction

In the book *Programming Pearls*, Jon Bentley presents the binary search algorithm, and uses it as a running example to extol the virtues of:

Testing. The use of software to automatically run test cases is presented,

Assertions. Assertions are used throughout his implementation of binary search in order to help maintain program invarients.

Formal Methods. He applies formal methods to his code in depth to prove it 'correct'

Asymptotic Analysis. The use of analytical methods allow him to place an upper bound on his program's runtime and memory requirements.

All of the above should give rise to robust software which can meet a specification in as many cases as possible, and still have good performance. Similarly, I shall use the binary search as a recurring example in this article. This article will demonstrate how to use most of the above techniques in Haskell.

2 The First Implementation

This implementation will be a simpler version of binary search which is easier to understand, avoids the Maybe monad but still has many of the properties useful for the example. In particular, it will only tell you if the key is in the sorted input; not where in the input it is. Therefore, it will have the type signature:

```
binarySearch :: (Ord a) => [a] -> a -> Bool
```

From here, we can start the implementation with our base case. If the input list is empty, it can't contain any keys, hence the result is always false.

```
binarySearch [] _ = False
```

Now we can begin to reason about the behaviour of a longer list, xs. We need to check the midpoint of the list for the value we're looking for, x. If that value is larger or smaller than the value we're looking for, then we need to search in the upper or lower half of xs for the value. If the midpoint is equal to x, then we simply return true.

```
module BinarySearch (binarySearch) where

binarySearch :: (Ord a) => [a] -> a -> Bool
binarySearch [] _ = False
binarySearch [x] a = x == a
binarySearch xs a
  | a == (xs !! midpoint) = True
  | a < (xs !! midpoint) = binarySearch lower a
  | a > (xs !! midpoint) = binarySearch upper a
  where
    midpoint = floor $ (fromIntegral $ length xs) / 2
    upper = drop (midpoint + 1) xs
    lower = take midpoint xs
```

We can use this as out base for QuickCheck. QuickCheck is a library which allows us to specify properties about our software that must hold in all cases. QuickCheck then generates random data and tests to see if the properties hold for that data.

We can reason that, for any input list, every element of that list is in that list, and therefore, searching for any element of that list in the sorted version of that list should always be true.

It is also true that given an input list and an element x, if that element x is not in the input list, then searching for that element in the list will always return false. We write two QuickCheck properties to account for these two properties.

```
import BinarySearch (binarySearch)
import Data.List
import Test.QuickCheck

prop_contains :: [Int] -> Property
prop_contains xs = True ==>
          let ys = sort xs in and [ binarySearch ys y | y <- ys ]

prop_not_contains :: [Int] -> Int -> Property
prop_not_contains xs x = x `notElem` xs ==> not $ binarySearch (sort xs) x

main = do {
   quickCheck prop_contains;
   quickCheck prop_not_contains; }
```

We use true as the precondition for prop_contains, because this particular property should hold for every list that QuickCheck can use for this test. This

is then run using the command line runhaskell tool:

```
$> runhaskell binarySearchTests.hs
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
$>
```

QuickCheck is told in this case to use a concrete type, [Int] for the input list as this simplifies the code slightly for us. There is a polymorphic variant of QuickCheck, however, I leave that aside, as the idea is vastly the same.

3 Analysis of our first implementation

Our first implementation shows many desirable properties. It's short, easy to read, easy to test and apparently easy to reason about. However, there are subtle bugs lurking in our implementation that QuickCheck is very unlikely to find, as well as serious performance issues.

3.1 The Subtle Bug

While at first glace, our implementation appears to be entirely correct, and the testing increases our confidence in this, it is actually a slightly buggy implementation. This bug is shared by Bentley's implementation and stayed hidden in Java's standard library for nearly 20 years. It was only recently brought to my attention by a blog post[**?**] by Joshua Bloch, who wrote the buggy Java implementation. The bug is an Integer overflow, and it occurs on the following line:

```
midpoint = floor $ (fromIntegral $ length xs) / 2
```

If we assume a 64-bit architecture, length xs returns an Int which can overflow. In particular if $|xs| \geq 2$ then length xs $\neq |xs|$. Specifically, if $|xs| = 2^64$ then length xs $= 0 \neq |xs|$. Further more, this overflow means that it computes the midpoint to be at the beginning of the list in this case!

One fix for this is to switch away from Int all together to Integer, which does not overflow. In the Data.List package, there exists several functions which can manipulate lists using Integers.

```
genericLength :: (Num a) => [b] -> a
genericTake :: (Integral i) => i -> [a] -> [a]
genericDrop :: (Integral i) => i -> [a] -> [a]
genericIndex :: (Integral i) => [b] -> i -> b
```