Assignment 2
Testing with QuickCheck

Model-Based Testing
DIT848/GU and TDA260/Chalmers

Spring 2014

1   Introduction

The goal of this assignment is for you to learn how to test simple Haskell functions using QuickCheck, by devising suitable properties for them and then interpreting the results of the tests accordingly. In this assignment you will be given implementations of certain data structures in Haskell, and you will be required to write and test QuickCheck properties that they should satisfy.

2   Submitting your work

You have to submit your work through the Fire system `http://xdat09.ce.chalmers.se/mbt/`. Upload your source code and txt or pdf file describing your answers.
The deadline for this assignment is **Wednesday, 16 April 2014**.

3   Testing Stacks

Module `Stack` includes an implementation of stacks in Haskell. The module introduces a parameterised data type, `Stack`, such that for every type `a`, we have the type `Stack a` of stacks of as, *e.g.* a `Stack Int` is a stack holding integers.

This stack implementation is pure, *i.e.* there are no side-effects. As a consequence, the functions that manipulate stacks always return new stacks as their result, instead of modifying stacks in place. The interface for this module includes the following functions:

```
push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
isEmpty :: Stack a -> Bool
empty :: Stack a
```

The function `push` takes an element x and a stack s, and produces a new stack with x as its topmost element, followed by the elements of s. The constant `empty` represents the empty stack, while the function `isEmpty` simply checks whether its argument is the empty stack or not. The functions `pop` and `top` are used to take a stack apart: `pop` pops off an element from its argument, returning the resulting stack, and `top` returns the topmost element in its argument stack (without changing the stack). Their behaviour is undefined when applied to the empty stack.

There are some properties that the functions in this interface should satisfy. For example, we would like to check that, whenever we push an element into a stack, we get a nonempty stack as a result. This property could be encoded in QuickCheck as follows:

```
prop_ne x st = isEmpty (push x st) == False where _ = x :: Int
```

Exercise 1. Write **three** more properties for `Stack` and test them using QuickCheck. Try to express the way the functions in the interface should interact with each other, in order to preserve the informal semantics given above.

4 Testing binary search trees

Binary search trees are binary trees used to store data for which there is a valid notion of order, with a view to searching for specific data efficiently (as opposed to using a flat list structure). Module BST includes an implementation of binary search trees in Haskell. As before, we introduce type Tree such that Tree a is a tree of as, along with the following operations:

```haskell
insert :: Ord a => a -> Tree a -> Tree a
member :: Ord a => a -> Tree a -> Bool
isEmpty :: Tree a -> Bool
empty :: Tree a
```

As usual, insert takes an element of type a and a BST, and returns a new BST with the extra element in the correct position. Repeated elements are stored only once. Moreover, member checks that a given element is present in the tree. The function isEmpty checks whether the argument tree is empty, while the constant empty represents the empty tree.

The BST property. Given a tree t, we say that it has the *BST property* if either of these conditions hold:

- t is empty;

- if t is of the form Node lt x rt, then both lt and rt have the BST property, and for all $y \in$ lt we have $y < x$, and for all $z \in$ rt we have $x < z$, *i.e.* all elements of lt are lower than x, and all elements of rt are greater than x.

The function insert should preserve the BST property as an invariant. The function member utilises this invariant to make the search more efficient.

Exercise 2.

1. Write a QuickCheck property that checks the invariance of the BST property in this BST implementation.

2. Write **two** more properties that you think should hold for BST trees. Bear in mind that the properties should model the informal semantics as closely as possible. Be creative in your tests! **Hint:** Is it possible to sort a list by using BSTs as intermediate data structures?

3. Test all these properties using QuickCheck. Should any test fail, you are expected to find and report the error, and tell whether it is in the implementation itself or in your properties.

NB. In order to test properties about the Tree data type, QuickCheck requires a *generator* for trees, *i.e.* an instance of the Arbitrary class. For this exercise, this instance is provided in the module, which generates trees using insert (bear this in mind in your tests). More details can be found on section "Testing case study" in *Real World Haskell*, Chapter 11.

5 Calendar date generators

ISO 8601 Data elements and interchange formats is an international standard to store in and exchange date between computer systems. In this standard, date values are organized in three parts from the most to the least significant: year, month and day. YYYY-MM-DD Each value has a fixed number of digits that must be padded with leading zeros. [YYYY] indicates a four-digit year, 0000 through 9999. [MM] indicates a two-digit month of the year, 01 through 12. [DD] indicates a two-digit day of that month, 01 through 31.

1. Write a calendar date *generator* for type `data Date = Date Int Int Int` that is used to store valid dates.

2. Write a calendar date *generator* of type `Gen String` that generates string dates of ISO 8601 format.

NB. Complete the file Date.hs that contains the function prototypes of needed functions. You can also use the function sample to see a sample set of dates from your generator.