Software Specification and Testing Exam, October 26, 2016, with Answers

```
module ExamOct2016Answers where

import Data.List
import Test.QuickCheck
```

Problem 1

Which of the following are correct Hoare statements (assume that x ranges over integer numbers)?

(a)
$$\{\lambda x \mapsto x < 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x > 0\}.$$

(b)
$$\{\lambda x \mapsto x \leq 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x > 0\}.$$

(c)
$$\{\lambda x \mapsto x < 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x \geq 0\}.$$

(d)
$$\{\lambda x \mapsto x \leq 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x \geq 0\}.$$

Answer

(a) correct

(b) not correct

(c) correct (weakening of a)

(d) correct

Problem 2

Let R be the relation $\{(0,1),(1,2),(3,0)\}$ on $\{0,1,2,3\}$.

(a) What is $R \circ R$?

(b) What is the transitive closure of R?

Please state your answers as sets of pairs.

Answer

(a)

$$\{(0,2),(3,1)\}.$$

(b)

$$\{(0,1),(1,2),(3,0),(0,2),(3,1),(3,2)\}.$$

Problem 3

List all relations on $\{0,1,2\}$ that are equivalences, in two ways:

(a) as sets of pairs,

(b) as partitions.

(Recall that an equivalence is a relation that is reflexive, symmetric and transitive, and that every equivalence corresponds to a partition.)

Answer

(a)

$$\{(0,0),(1,1),(2,2)\},\{(0,0),(1,1),(2,2),(0,1),(1,0)\}$$
$$\{(0,0),(1,1),(2,2),(0,2),(2,0)\},\{(0,0),(1,1),(2,2),(1,2),(2,1)\},$$
$$\{(0,0),(1,1),(2,2),(0,1),0,2),(1,0),(1,2),(2,0),(2,1)\}.$$

(b)

$$\{\{0\},\{1\},\{2\}\},\{\{0,1\},\{2\}\},\{\{0\},\{1,2\}\},\{\{1\},\{0,2\}\},\{\{0,1,2\}\}.$$

Problem 4

A relation R is anti-symmetric on a domain A if for all $x, y \in A$ it holds that xRy and yRx together imply $x = y$.

Suppose relations are represented as lists of pairs:

```
type Rel a = [(a,a)]
```

Implement a function for checking whether a relation is anti-symmetric The domain is implicit: it consists of all items that are mentioned in the pairs in the relation. The type specification imposes the constraint that `a` is an equality type:

```
isAntiSymm :: Eq a => Rel a -> Bool
isAntiSymm r = error "not yet implemented"
```

How can you test whether your implementation is correct?

Answer

```
isAntiSymm :: Eq a => Rel a -> Bool
isAntiSymm rel = and [ x == y | (x,y) <- rel, (y,x) `elem` rel ]
```

Here is a test property:

```
prop_antisymm :: Eq a => Rel a -> Bool
prop_antisymm r = let
   r' = map (\ (x,y) -> (y,x)) r
  in
   isAntiSymm r == isAntiSymm r'
```

Testing this with QuickCheck gives success.

Another test is to remove all identical pairs from the relation. This should make no difference for the property:

```
prop_antisymm2 :: Eq a => Rel a -> Bool
prop_antisymm2 r = let
   r' = filter (uncurry (/=)) r
  in
   isAntiSymm r == isAntiSymm r'
```

Testing this with QuickCheck gives success.

For good measure you can also add a test that spoils the antisymmetry property of any relation:

```
prop_antisymm3 :: Eq a => a -> a -> Rel a -> Bool
prop_antisymm3 x y r =
  x == y || not (isAntiSymm ((x,y):(y,x):r))
```

Problem 5

A *Huffman tree* is a binary tree with characters at the leaf nodes, and weight information (given by an integer) at every node.

```
data HTree  = Leaf Char Int
            | Fork HTree HTree Int
            deriving (Show)
```

The weight of a tree is given by:

```
weight :: HTree -> Int
weight (Leaf _ w)    = w
weight (Fork _ _ w)  = w
```

Call the following property the *Huffman property*:

```
prop_huffman :: HTree -> Bool
prop_huffman (Leaf _ _) = True
prop_huffman (Fork t1 t2 w) = prop_huffman t1 && prop_huffman t2
                    && weight t1 + weight t2 == w
```

Character frequency tables are lists of pairs `(Char,Int)` where the second element gives the frequency of the first character in some text.

Huffman trees are built from character frequency tables, as follows:

```
createTree :: [(Char,Int)] -> HTree
createTree []      = error "empty input list"
createTree [(c,i)] = Leaf c i
createTree ((c,i):t) = merge (Leaf c i) (createTree t)
```

This uses the following merge function:

```
merge :: HTree -> HTree -> HTree
merge t1 t2 = Fork t1 t2 (weight t1 + weight t2)
```

Give an induction proof to show that it holds for any non-empty input list `xs :: [(Char,Int)]` that `createTree xs` has the Huffman property.

Answer

We proceed by induction on the length of the input list `xs`. We assume that this list is non-empty. So the base case is `xs == [(c,i)]`. In this case the tree `Leaf c i` is created, and this tree has the Huffman property by definition, for all leaf trees have the Huffman property.

Induction step. Assume `xs` is of the form `(c,i):ys`. The induction hypothesis is that the tree created from `ys` has the Huffman property. The tree from `(c,i):ys` is created by merging `(Leaf c i)` with the tree `createTree ys`. `(Leaf c i)` has the Huffman property by definition. `createTree ys` has the Huffman property by the induction hypothesis.

So the only thing we still have to check is whether the result of the merge of these two trees has the property that its weight is the sum of the weights of the two subtrees. But this is exactly what `merge` achieves.

Problem 6

Frequency tables are lists of pairs `(Char,Int)` where the second element gives the frequency of the first character in some text.

Frequency tables are built from strings with the following function:

```
freqList :: String -> [(Char,Int)]
freqList s = error "not yet implemented"
```

This should give, e.g.:

```
*ExamOct2016> freqList "Lab Exam"
[(' ',1),('E',1),('L',1),('a',2),('b',1),('m',1),('x',1)]
```

The composition of `freqList` and `createTree` gives us a map from strings to Huffman trees:

```
string2tree :: String -> HTree
string2tree = createTree . freqList
```

For more information about how Hufmann trees are used, see *Huffman coding*.

Your tasks are:

(a) to implement `freqList`,

(b) to test your implementation of `freqList` with QuickCheck for correctness, using some appropriate properties,

(c) to test with QuickCheck whether for any nonempty string `s` it is the case that `string2tree s` has the Huffman property.

Answer

```
freqList :: String -> [(Char,Int)]
freqList s = let
   units = zip (sort s) (repeat 1)
   f  :: [(Char,Int)] -> [(Char,Int)]
   f [] = []
   f [(x,m)] = [(x,m)]
   f ((x,m):(y,n):zs) | x == y    = f ((x,m+n):zs)
                      | otherwise = (x,m): f ((y,n):zs)
 in
   f units
```

An easy way to test this is by generating a string from a frequency table, and use this to construct a new frequency table. The two tables should be identical.

```
ft2string :: [(Char,Int)] -> String
ft2string = concatMap (\ (c,n) -> replicate n c)

prop_table :: String -> Bool
prop_table s = let
  lst = freqList s
 in lst == freqList (ft2string lst)
```

Next, test with Quickcheck:

```
test1 :: IO ()
test1 = quickCheck prop_table
```

This test succeeds.

For testing `string2tree`, we have to assume that the input string to create the frequency table has at least one element, for the input of `createTree` has to be non-empty. This gives us:

```
test2 :: IO ()
test2 = quickCheck (\s -> null s || prop_huffman (string2tree s))
```

This test succeeds.

Problem 7

Russian roulette is played with a revolver (a six-shooter) and a single bullet. The player inserts the bullet in the cylinder, spins the cylinder, puts the gun against his head, and pulls the trigger.

If you play this for one round, the chance that you die is $\frac{1}{6}$. If you survive a round you can play a next round by spinning the cylinder again and repeating the procedure.

If you play like this for at most 6 rounds, the chance that you die is given by:

$$1/6 + (5/6)(1/6) + (5/6)^2(1/6) + (5/6)^3(1/6) + (5/6)^4(1/6) + (5/6)^5(1/6).$$

If the example bothers you, you can also think about the probability of throwing at least one six with a fair die, in 6 rounds.

Explanation: the probability of a fatal outcome is given by: probability of fatal outcome in round 1, plus probability of surviving round 1 and fatal outcome in round 2, plus ..., plus probability of surviving rounds 1 through 5 and fatal outcome in round 6.

In general, if success (but in the example: fatal outcome) in one round happens with probability p, then failure in one round happens with probability $q = 1 - p$, and success in at most k rounds happens with probability:

$$p + qp + \cdots + q^{k-1}p.$$

You are asked to investigate the following three implementations of *probability of success in at most k rounds*, when a single round has probability p of success:

```
probSuccess1 :: Int -> Rational -> Rational
probSuccess1 k p = let q = 1 - p in
  sum [ p * q^m | m <- [1..k-1] ]
```

Here is a different implementation:

```
probSuccess2 :: Int -> Rational -> Rational
probSuccess2 k p = 1 - (1 - p)^k
```

And here is yet a different one:

```
probSuccess3 :: Int -> Rational -> Rational
probSuccess3 k p = let q = 1 - p in
  ((q^k - 1) / (q - 1)) * p
```

Which of these implementations is correct? Or are they all flawed? Or maybe they are all correct? State your answer and describe how you found out.

Answer

The first implementation is definitely flawed, for it gives:

```
*ExamOct2016Answers> probSuccess1 1 (1/6)
0 % 1
```

The error is that the parameter m should range over $[0..k-1]$, not $[1..k-1]$. The first summand in $p + qp + \cdots + q^{k-1}p$ equals $q^0 p$. Here is a corrected version of the first implementation:

```
probSuccess1' :: Int -> Rational -> Rational
probSuccess1' k p = let q = 1 - p in
  sum [ p * q^m | m <- [0..k-1] ]
```

The second implemenation is correct. To see why, observe that the probability of having at least one successful outcome in k attempts has to be the same as the complement of having no successful outcome in k attempts. That is:

$$p + qp + \cdots + q^{k-1}p = 1 - q^k.$$

How about the third implementation? From $\frac{q^k-1}{q-1}p$ we can derive, by substituting $1-p$ for q:

$$\frac{q^k - 1}{q - 1}p = \frac{(1-p)^k - 1}{1 - p - 1}p = \frac{1 - (1-p)^k}{p}p = 1 - (1-p)^k.$$

This shows that `probSuccess2` and `probSuccess3` are equivalent (but see below).

Finally, we relate the first implementation to the third. We derive

$$p + qp + \cdots + q^{k-1}p = \frac{q^k - 1}{q - 1}p.$$

To see this, observe that

$$q(p + qp + \cdots + q^{k-1}p) = qp + q^2p + \cdots + q^kp,$$

and therefore

$$q(p + qp + \cdots + q^{k-1}p) - (p + qp + \cdots + q^{k-1}p) = q^kp - p = (q^k - 1)p.$$

It follows that

$$(q - 1)(p + qp + \cdots + q^{k-1}p) = (q^k - 1)p,$$

and therefore

$$p + qp + \cdots + q^{k-1}p = \frac{q^k - 1}{q - 1}p.$$

Done.

Well, not quite.

If you prefer testing to reasoning, then here are some test properties:

```
prop_probsucc12 :: Int -> Rational -> Bool
prop_probsucc12 k p =
     k < 0 || probSuccess1' k p == probSuccess2 k p

prop_probsucc13 :: Int -> Rational -> Bool
prop_probsucc13 k p =
     k < 0 || probSuccess1' k p == probSuccess3 k p

prop_probsucc23 :: Int -> Rational -> Bool
prop_probsucc23 k p =
     k < 0 || probSuccess2 k p == probSuccess3 k p
```

Quite unexpectedly, this reveals a flaw in `probSuccess3`:

```
*Exam0ct2016Answers> quickCheck prop_probsucc23
*** Failed! Exception: 'Ratio has zero denominator' (after 1 test):
0
0 % 1
```

So there is a difference: `probSuccess3` throws an error if you call it with a probability argument of 0.

Looking back at the last derivation, we see that the final step is flawed: division by $q - 1$, i.e., $(1 - p) - 1$, i.e. $-p$, is only allowed if $p \neq 0$.
