Software testing: week 4 report

Tuba Kaya Chomette, Sander Leer, Martijn Stegeman

September 29, 2013

```
module Week4Sol

where

import SetOrd
import Techniques
import Data.List
import Week3Sol
import TAMO
```

Random integer set generator for testing

The generator is limited to sets of at most 10 elements, each of which can be an integer between 0 and 10. Length of the resulting set is influenced by the integers that are generated to go into the set, as any duplicates will limit the length.

```
randomSet :: IO (Set Int)
randomSet = getRandomInt 10 >>= randomSetOfMaxLength

randomSetOfMaxLength :: Int -> IO (Set Int)
randomSetOfMaxLength 0 = return (Set [])
randomSetOfMaxLength x = do
    element <- getRandomInt 10
    set <- randomSetOfMaxLength (x-1)
    return (insertSet element set)
```

Set operations

Let's define three operations on `Set`.

```
setUnion, setIntersect, setDiff :: (Eq a, Ord a) => Set a -> Set a -> Set a
```

They can be defined in terms of the `union`, `intersect` and `(\\)` functions as provided in `Data.list`.

```
setUnion (Set xs) (Set ys) = Set (union xs ys)
setDiff (Set xs) (Set ys) = Set (xs \\ ys)

--setIntersect (Set xs) (Set ys) = Set (intersect xs ys)
```

```
setIntersect _ (Set []) = Set []
setIntersect (Set []) _ = Set []
setIntersect (Set (x:xs)) set2 | inSet x set2 = insertSet x (setIntersect (Set xs) set2 )
                               | otherwise = setIntersect (Set xs) set2

removeSetFromSet :: Ord a => Set a -> Set a -> Set a
removeSetFromSet _ (Set []) = Set []
removeSetFromSet (Set []) y = y
removeSetFromSet (Set (x:xs)) set2
    | isEmpty (Set (x:xs)) == False = removeSetFromSet (Set xs) (deleteSet x set2)
    | otherwise = set2
```

Random testing for the set operations

The first property we can say something about is the length of the resulting set after applying one of the operations to sets S_1 and S_2:

$$\begin{array}{rcl} |S_1 \cup S_2| & = & |S_1| + |S_2| \\ |S_1 \cap S_2| & \leq & |S_1| + |S_2| \\ |S_1 - S_2| & \leq & |S_1| \end{array}$$

So we define an auxiliary function `size` to calculate any set's size, and three functions that verify the length of result sets:

```
size :: Set a -> Int
size (Set a) = length a

checkUnionSize, checkIntersectionSize, checkDifferenceSize ::
  (Ord a, Eq a) => Set a -> Set a -> Bool

checkUnionSize s1 s2 = (size (setUnion s1 s2)) <= (size s1) + (size s2)
checkIntersectionSize s1 s2 = (size (setIntersect s1 s2)) <= (min (size s1) (size s2))
checkDifferenceSize s1 s2 = (size (setDiff s1 s2)) <= (size s1)
```

Now we can define a test function that verifies the length properties for one random case:

```
testSetOps :: IO Bool
testSetOps = do
    s1 <- randomSet
    s2 <- randomSet
    putStrLn (show s1 ++ " and " ++ show s2)
    putStrLn ("  union       : " ++ show (setUnion s1 s2))
    putStrLn ("  intersection: " ++ show (setIntersect s1 s2))
    putStrLn ("  difference  : " ++ show (setDiff s1 s2))
    putStrLn (show ((checkUnionSize s1 s2) &&
```

```
             (checkIntersectionSize s1 s2) &&
             (checkDifferenceSize s1 s2)))
    return ( (checkUnionSize s1 s2) &&
             (checkIntersectionSize s1 s2) &&
             (checkDifferenceSize s1 s2) )

autoTest :: Int -> IO Bool -> IO Bool
autoTest 0 _ = return True
autoTest x f = do
    t <- f
    ts <- autoTest (x-1) f
    return (t && ts)

autoTestSetOps = autoTest 1000 testSetOps
```

We can also test for the invariant property that the sets have no duplicate members. We know that the Haskell list union and intersection functions, as well as our own difference function are created to uphold this invariant. However, lists in general do allow duplicates in Haskell, so testing is very useful.

```
hasDup :: Eq a => Set a -> Bool
hasDup (Set a) = a /= nub a

testSetOpsInvariant :: IO Bool
testSetOpsInvariant = do
    s1 <- randomSet
    s2 <- randomSet
    putStrLn (show s1 ++ " and " ++ show s2)
    putStrLn ("  s1 has dup          : " ++ show (hasDup s1))
    putStrLn ("  s2 has dup          : " ++ show (hasDup s2))
    putStrLn ("  union has dup       : " ++ show (hasDup (setUnion s1 s2)))
    putStrLn ("  intersection has dup: " ++ show (hasDup (setIntersect s1 s2)))
    putStrLn ("  difference has dup  : " ++ show (hasDup (setDiff s1 s2)))
    return (not (or [(hasDup s1), (hasDup s2),
        (hasDup (setUnion s1 s2)),
        (hasDup (setIntersect s1 s2)), (hasDup (setDiff s1 s2))]))

testSetOpsInv = autoTest 1000 testSetOpsInvariant
```

We can also test some expected output from combinations of these set operations:

```
-- intersection of a "Set a = x" and "Set b = 0 + x" should be "Set a"
randomTestIntersection1 :: (Num a, Ord a, Eq a) => Set a -> Bool
randomTestIntersection1 x = (setIntersect x (insertSet 0 x)) == x
```

```
-- intersection of a "Set a = x" and "Set b = (first element of x)" should be "Set b"
randomTestIntersection2 :: (Ord a, Eq a) => Set a -> Bool
randomTestIntersection2 x
    | x == (list2set []) = True
    | otherwise = let y = (list2set [x !!! 0]) in ((setIntersect x y) == y )

-- intersection of a "Set a = x" and "Set b = []" should be "Set b" or "[]"
randomTestIntersection3 :: (Ord a, Eq a) => Set a -> Bool
randomTestIntersection3 x = let y = (list2set []) in (y == (setIntersect x y))

-- intersection of a "Set a = []" and "Set b = x" should be "Set a" or "[]"
randomTestIntersection4 :: (Ord a, Eq a) => Set a -> Bool
randomTestIntersection4 x = let y = (list2set []) in (y == (setIntersect y x))

testSetIntersection :: IO Bool
testSetIntersection = do
    t <- (randomSet)
        putStrLn (show t)
        return (randomTestIntersection1 t
        && randomTestIntersection2 t
        && randomTestIntersection3 t
        && randomTestIntersection4 t)

autoTestIntersection = autoTest 1000 testSetIntersection

-- setDiff of a "Set a = x" and "Set b = 0 + x" should be ""
randomTestSetDifference1 :: (Num a, Ord a, Eq a) => Set a -> Bool
randomTestSetDifference1 x = ((setDiff x (insertSet 0 x)) == list2set [] )

-- setDiff of a "Set a = x" and "Set b =x - (first element of x)" should be (first element of x)
randomTestSetDifference2 :: (Num a, Ord a, Eq a) => Set a -> Bool
randomTestSetDifference2 x
    | x == (list2set []) = True
    | otherwise = let y = (x !!! 0) in ((setDiff x (deleteSet y x)) == (list2set [y]))

-- setDiff of a "Set a = x" and "Set b = []" should be "Set a"
randomTestSetDifference3 :: (Ord a, Eq a) => Set a -> Bool
randomTestSetDifference3 x = x == (setDiff x (list2set []))

-- setDiff of a "Set a = []" and "Set b = x" should be "Set a"
randomTestSetDifference4 :: (Ord a, Eq a) => Set a -> Bool
randomTestSetDifference4 x = let y = (list2set []) in ((setDiff y x) == y)
```

```
testSetDifference :: IO Bool
testSetDifference = do
    t <- (randomSet)
    putStrLn (show t)
    return (randomTestSetDifference1 t
         && randomTestSetDifference2 t
         && randomTestSetDifference3 t
         && randomTestSetDifference4 t)


autoTestSetDifference = autoTest 1000 testSetDifference
```

Transitive closure

Let us introduce the notation for relations as lists of tuples in Haskell, as specified in the assignment:

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

infixr 5 @@
(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s = nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

And a function for checking closures inspired by the Relations chapter of HR:

```
transR :: Ord a => Rel a -> Bool
transR [] = True
transR s = and [ trans pair s | pair <- s ] where
    trans (x,y) r = and [ elem (x,v) r | (u,v) <- r, u == y ]
```

We now implement a function that produces the transitive closure of a relation. It repeatedly applies `(@@)` to add parts needed to achieve transitive closure. In the base case, it `sort`s and `nub`s for easy visual inspection.

```
trClos :: Ord a => Rel a -> Rel a
trClos xs | transR xs = sort $ nub xs
          | otherwise = trClos ((xs @@ xs) ++ xs)
```

Random testing for the transitive closure calculator

Some properties we can check:

- The transitive closure of relation S is *always* a superset of S.
- The result of `trClos` should *always* be a transitive closure.

- *If* `trClos` is applied to a set R in transitive closure, the result should be equal to R.

So, we introduce these properties as functions:

```
prop1 :: Rel Int -> Bool
prop1 r = subRel r (trClos r)

prop2 :: Rel Int -> Bool
prop2 r = transR (trClos r)
```

Note: we use `transR` in this check, which is also used in the definition of `trClos`. Does this make sense?

```
prop3 :: Rel Int -> Bool
prop3 r = transR r ==> ((sort $ nub r) == trClos r)

subRel :: (Ord a) => Rel a -> Rel a -> Bool
subRel [] _       = True
subRel (x:xs) rel = (elem x rel) && subRel xs rel
```

Then, we set up some random relation generators:

```
getRandomRel :: IO (Rel Int)
getRandomRel = do
    len <- getRandomInt 10
    r1 <- randomInts len
    r2 <- randomInts len
    return (zip r1 r2)

getRandomRels :: Int -> IO [(Rel Int)]
getRandomRels 0 = return []
getRandomRels x = do
    r <- getRandomRel
    rs <- getRandomRels (x-1)
    return (r:rs)
```

And finally, the function that can run a number of tests on a given property, together with the main function that runs the tests for all three properties.

```
testRels :: Int -> (Rel Int -> Bool) -> [Rel Int] -> IO ()
testRels n _ [] = putStrLn ("  " ++ show n ++ " tests passed")
testRels n p (f:fs) =
    if p f
    then do testRels n p fs
    else error ("failed test on:" ++ show f)
```

```
runTest = do
    rs <- getRandomRels 1000
    putStr "testing prop1..."
    testRels 1000 prop1 rs
    putStr "testing prop2..."
    testRels 1000 prop2 rs
    putStr "testing prop3..."
    testRels 1000 prop3 rs
```

Time spent

- Time spent on getting the `Makefile` to work well for literate programming: 40 minutes.
- Time spent on the random integer set generator: 20 minutes.
- Time spent on the set operations: 20 minutes.
- Time spent on random testing for the set operations: 1,5 hours.
- Time spent on the transitive closure generator: 1 hour.
- Time spent on the transitive closure tester, including fixing Week 3's random list generator: 3 hours.