Software Testing Assignment 5

Cindy Berghuizen, Omar Pakker, Chiel Peter, Maria Gouseti

October 6, 2013

Exercise 1

Assertion

A valid output of the function *mergerSrt* would be the sorted version of the input list. In order to assert that, the postcondition should comprise of two checks:

1. The output has exactly the same elements as the input.
2. The output list is sorted.

Assertive version

```
--Assertion 1 : The input set has the same elements as the output set
--Assertion 2 : The elements in the output set are sorted

sElements :: Eq a => [a] -> [a] -> Bool
sElements [] [] = True
sElements _ [] = False
sElements (x:xs) ys =
  elem x ys && sublist xs (ys \\ [x])

mergeSrtA :: Ord a => [a] -> [a]
mergeSrtA = assert1 (\ _ -> sorted) $ assert1 (sElements) mergeSrt
```

Time spent: 15 minutes

Exercise 2

The *mergeSrtSplit* function uses the function *split* to split the list in half until it reaches either an empty list or a list with one element and then starts merging the parts of the list with the *merge* function.

```
mergeSrtSplit :: Ord a => [a] -> [a]
mergeSrtSplit [] = []
mergeSrtSplit [x] = [x]
mergeSrtSplit x = merge (mergeSrtSplit y) (mergeSrtSplit z) where (y,z) = split x
```

Assertion

The assertions are the same with the assertions of the *mergerSrt* function.

Assertive version

```
mergeSrtSplitA :: Ord a => [a] -> [a]
mergeSrtSplitA = assert1 (\ _ -> sorted) $ assert1 (sElements) mergeSrtSplit
```

Time spent: 15 minutes

Exercise 3

Formal statement of constraint

Row Constraint:
$\forall i,j,k \in \{1,...,9\}\ j \neq k \rightarrow grid(i,j) \neq grid(i,k)$
Column Constraint:
$\forall i,j,k \in \{1,...,9\}\ j \neq k \rightarrow grid(j,i) \neq grid(k,i)$
Grid Constraint:
$\forall x,y \in \{1,4,7\}\ \forall i,j,k,l \in \{1,...,9\}\ ((x \leq i,k \leq x+2) \wedge (y \leq j,l \leq y+2) \wedge grid(i,j) = grid(k,l)) \rightarrow (i = k \wedge j = l)$
New Grid Constraint:
$\forall x,y \in \{2,6\}\ \forall i,j,k,l \in \{1,...,9\}\ ((x \leq i,k \leq x+2) \wedge (y \leq j,l \leq y+2) \wedge grid(i,j) = grid(k,l)) \rightarrow (i = k \wedge j = l)$

Sudoku solution

Added the second set of subgrids.

```
blocks2 :: [[Int]]
blocks2 = [[2..4],[6..8]]
```

Adapted the *show* function to show the new subgrids as the example in the assignment.

```
showRow :: [Value] -> IO()
showRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] =
 do  putChar '|'         ; putChar ' '
     putStr (showDgt a1) ; putChar ' '; putChar ' '
     putStr (showDgt a2) ; putChar ' '
     putStr (showDgt a3) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a4) ; putChar ' '; putChar ' '
     putStr (showDgt a5) ; putChar ' '; putChar ' '
     putStr (showDgt a6) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a7) ; putChar ' '
     putStr (showDgt a8) ; putChar ' '; putChar ' '
     putStr (showDgt a9) ; putChar ' '
     putChar '|'         ; putChar '\n'

showSRow :: [Value] -> IO()
showSRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] =
 do  putChar '|'         ; putChar ' '
     putStr (showDgt a1) ; putChar ' '; putChar '|'
     putStr (showDgt a2) ; putChar ' '
     putStr (showDgt a3) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a4) ; putChar '|'; putChar ' '
     putStr (showDgt a5) ; putChar ' '; putChar '|'
     putStr (showDgt a6) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a7) ; putChar ' '
     putStr (showDgt a8) ; putChar '|'; putChar ' '
     putStr (showDgt a9) ; putChar ' '
     putChar '|'         ; putChar '\n'

showGrid :: Grid -> IO()
showGrid [as,bs,cs,ds,es,fs,gs,hs,is] =
 do putStrLn ("+--------+---------+--------+")
    showRow as
    putStrLn ("|   +----|--+   +--|----+   |")
    showSRow bs; showSRow cs
    putStrLn ("+--------+---------+--------+")
    showSRow ds;
    putStrLn ("|   +----|--+   +--|----+   |")
    showRow es;
    putStrLn ("|   +----|--+   +--|----+   |")
```

```
    showSRow fs
    putStrLn ("+--------+---------+--------+")
    showSRow gs; showSRow hs;
    putStrLn ("|   +----|--+   +--|----+   |")
    showRow is
    putStrLn ("+--------+---------+--------+")
```

The function *bl*2 is the same as *bl* but refers to the case of the new subgrids.

```
bl2 :: Int -> [Int]
bl2 x = concat $ filter (elem x) blocks2
```

The conflicting values from the subgrids include the new blocks as well.

```
subGrid :: Sudoku -> (Row,Column) -> [Value]
subGrid s (r,c) =
  subGrid1 s (r,c) `union` subGrid2 s (r,c)

subGrid1 :: Sudoku -> (Row,Column) -> [Value]
subGrid1 s (r,c) =
  [ s (r',c') | r' <- bl r, c' <- bl c ]

subGrid2 :: Sudoku -> (Row,Column) -> [Value]
subGrid2 s (r,c) =
  [ s (r',c') | r' <- bl2 r, c' <- bl2 c ]


subgridInjective :: Sudoku -> (Row,Column) -> Bool
subgridInjective s (r,c) = injective vs && injective vs2 where
           vs = filter (/= 0) (subGrid1 s (r,c))
           vs2 = filter (/= 0) (subGrid2 s (r,c))
```

A position is in the same block with another position either if they belong to the same standard subgrid or to the new ones defined by NRC.

```
sameblock :: (Row,Column) -> (Row,Column) -> Bool
sameblock (r,c) (x,y) = (bl r == bl x && bl c == bl y) || (bl2 r == bl2 x && bl2 c == bl2 y)
```

Time spent: 2,5 hours

Exercise 4

Instead of importing *Week*5 in the *RandomSudoku*, the module *Week*5*NRC* is imported as a result *RandomSudoku* will use the new rules and constraints from the NRC-Sudoku problem.
Time spent: 10 minutes

Exercise 5

Test report

In order to test the NRC-Sudoku solver, three checks are used. The first check tests if the solution is a consistent NRC Sudoku (*testConsistent*), the second check tests if the solution is minimal (*testMinimal*), meaning it has a unique solution, and the last check is responsible for ensuring that the initialized values from the input sudoku are not changed in the output sudoku (*testValues*). So the *genRandomProblem* function returns a list with the input sudoku and all its solutions. The function *testAll* apply the three checks we described at the list returned from *genRandomProblem* and returns *True* if the sudoku problem is solved successfully.

```
--Check          : Check if the solution is consistent
--Check2         : Check if the values are the same as the original problem
--Check3         : Check if the Random solutions are minimal

testAll :: IO Bool
testAll =       do
                                        x <- genRandomProblem
                                        return( testMinimal x && testConsistent x && testValues x)
```

```haskell
-- Check if the solution is minimal (There is one solution and the original problem in the list)
testMinimal x = (length x == 2)

-- Check if the solution is consistent
testConsistent (x:y:xs) = consistent (fst y)
testConsistent _                = False

--Check if the (first) solution has the same values as the original problem
testValues :: [Node] -> Bool
testValues (z:zs:zss)           = all (\(x,y) -> x == y || x == 0) (combine x y)
                                                        where
                                                        x = sud2grid(fst(z))
                                                        y = sud2grid(fst(zs))
testValues _                            = False

--genRandomProblem gives the problem as the first element of the List and the solution afterwards
genRandomProblem :: IO [Node]
genRandomProblem =      do
                                                [r] <- rsolveNs [emptyN]
                                                s  <- genProblem r
                                                return( initNode (sud2grid((fst s))) ++ (solveNs [s]) )

--Combine two grids into a list of tuples
combine :: [[a]] -> [[b]] -> [(a, b)]
combine [] _ = [];
combine _ [] = [];
combine (x:xs) (y:ys) = (zip x y) ++ combine xs ys
```

Time spent: 2,5 hours