Quiz (Week 4)

In a break with tradition, this quiz will involve content from both Week 3 and Week 4 lectures, as there was some stuff we missed last time.

Minimal Specifications

Question 1

Here is the fair merge function we wrote in last week's quiz, and a number of QuickCheck properties that specify its correctness.

```
import Test.QuickCheck
import Test.QuickCheck.Modifiers
import Data.List

merge :: (Ord a) => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x < y     = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
merge xs [] = xs
merge [] ys = ys

sorted :: Ord a => [a] -> Bool
sorted (x1 : x2 : xs) = (x1 <= x2) && sorted (x2 : xs)
sorted _ = True

prop_1 :: [Int] -> [Int] -> Bool
prop_1 xs ys = length (merge xs ys) == length (xs ++ ys)

prop_2 :: (Int -> Int) -> [Int] -> [Int] -> Bool
prop_2 f xs ys = sort (map f (merge xs ys))
              == sort (merge (map f xs) (map f ys))

prop_3 :: OrderedList Int -> OrderedList Int -> Bool
prop_3 (Ordered xs) (Ordered ys) = sorted (merge xs ys)

prop_4 :: OrderedList Int -> OrderedList Int -> Bool
prop_4 (Ordered xs) (Ordered ys) = merge xs ys == sort (xs ++ ys)

prop_5 :: (Int -> Bool) -> OrderedList Int -> OrderedList Int -> Bool
prop_5 f (Ordered xs) (Ordered ys) = filter f (merge xs ys)
                                  == merge (filter f xs) (filter f ys)
```

However, running tests for all of these properties takes too long for the impatient programmer. This is because many of these properties are logically *redundant*. If property A implies property B, then testing for property B is not likely to fail if testing for property A passes.

What subset of these properties imply all of the others? That is, which of the above properties are sufficient to establish correctness of `merge`?

Hint: Think about possible `merge` implementations that would fail the above properties. If you can't write a `merge` implementation that fails a given property but passes all of the others, it's a good indication that the property is redundant.

1. ✗ `prop_1`, `prop_2`, `prop_3`, `prop_4` and `prop_5` – there are no redundant properties.
2. ✗ `prop_1`, `prop_2`, `prop_3` and `prop_4`
3. ✓ `prop_1`, `prop_2`, and `prop_4`
4. ✗ `prop_2`, `prop_3`, and `prop_4`
5. ✗ Just `prop_4`

Property 5 is redundant, as we can use property 4 to construct an equational proof of property 5:

```
-- Assuming sorted xs && sorted ys
  filter f (merge xs ys)
== -- prop_4 (as sorted xs && sorted ys)
  filter f (sort (xs ++ ys))
== -- sort/filter interchangeable
  sort (filter f (xs ++ ys))
== -- filter distributes over (++)
  sort (filter f xs ++ filter f ys)
== -- prop_4 [sym] (as sorted (filter f xs) && sorted (filter f ys))
  merge (filter f xs) (filter f ys)
```

Property 3 is also redundant due to property 4:

```
-- Assuming sorted xs && sorted ys
  sorted (merge xs ys)
== -- prop_4 (as sorted xs && sorted ys)
  sorted (sort (xs ++ ys))
== -- sort is sorted
  True
```

Property 1 and 3 may at first seem similarly redundant due to property 4, however these properties do not assume that the input lists are sorted. A function that returns the empty list if the input lists are not sorted could still pass properties 2 and 4 despite

not passing property 1. And a function that that replaces all elements with the minimum when the input lists are not sorted could still pass properties 1 and 4 despite not passing 2. Thus the correct answer is option 2.

Question 2

Here's a standard (inefficient) reverse function for lists, and a collection of QuickCheck properties to specify it.

```
rev :: [Int] -> [Int]
rev (x:xs) = rev xs ++ [x]
rev []     = []

prop_1 :: [Int] -> Int -> Bool
prop_1 xs x = count x xs == count x (rev xs)
  where
    count x xs = length (filter (== x) xs)

prop_2 :: [Int] -> [Int] -> Bool
prop_2 xs ys = rev (xs ++ ys) == rev ys ++ rev xs

prop_3 :: [Int] ->  Bool
prop_3 xs = length xs == length (rev xs)
```

Which of the above properties is redundant?

1. ✗ `prop_1`
2. ✗ `prop_2`
3. ✓ `prop_3`
4. ✗ None of the above.

Property 1 essentially says that the output of reverse is a permutation of its input. Given that, we already know that the lengths will be the same. Thus property 3 is redundant.

Data Invariants

We are representing an *undirected graph* using an *adjacency matrix*. For example, a graph of four nodes arranged in a rectangle:

```
0 ----- 1
|       |
```

```
|       |
2 ----- 3
```

Would be represented as the matrix:

```
type Graph = [[Bool]]

m :: Graph
m = [[False, True,  True,  False],
     [True,  False, False, True ],
     [True,  False, False, True ],
     [False, True,  True,  False]]
```

That is, the value `(m !! a) !! b` is `True` iff the node numbered `a` is connected to the node numbered `b`.

Question 3

Select all data invariants that should apply to our `Graph` type to ensure it always represents a valid adjacency matrix for a graph, assuming `g :: Graph`:

1. ✓ `transpose g == g`
2. ✗ `reverse g == g`
3. ✗ `any and g`
4. ✓ `all (\x -> length x == length g) g`

To represent an undirected graph, the transpose of the matrix must be equal to the original matrix (it is diagonally symmetrical), hence property 1. The second property also holds for our example, but not in general (for example if node 1 is not connected to node 3). The third property is meaningless and not true if you have a node disconnected from all others. In order to be an adjacency matrix, the number of rows and columns must be equal, hence property 4 is needed.

Question 4

We have the following graph operations:

```
newGraph :: Int -- number of nodes
         -> Graph
newGraph n = replicate n (replicate n False)

connected :: Graph -> (Int, Int) -> Bool
connected g (x, y) | x < length g && y < length g = (g !! x) !! y
```

```
                  | otherwise                    = False

connect :: (Int, Int) -> Graph -> Graph
connect (x, y) = modify y (modify x (\_ -> True)) . modify x (modify y (\_ -> True))
  where
    modify :: Int -> (a -> a) -> [a] -> [a]
    modify 0 f (x:xs) = f x : xs
    modify n f (x:xs) = x : modify (n - 1) f xs
    modify n f []     = []
```

We have defined a wellformedness predicate for our data invariants, called `wellformed`. Select the properties we should assert about our operations[1].

1. ✗ `wellformed (connect (x, y) g) ==> wellformed g`
2. ✗ `wellformed (connect (x, y) g)`
3. ✓ `wellformed (newGraph n)`
4. ✓ `wellformed g ==> wellformed (connect (x, y) g)`

The second property does not necessarily apply, if the input graph is not well formed. The first property may apply in this case, but it is not necessary nor sufficient to show our data invariants are preserved.

The other properties states that outputs are wellformed when inputs are wellformed, which is correct.

Data Refinement

For our abstract model, it's easier to think of our graph as an edge list, along with the overall number of nodes in the graph.

```
data Model = M Int [(Int, Int)]

newGraphA :: Int -> Model
newGraphA n = M n []

connectedA :: Model -> (Int, Int) -> Bool
connectedA (M n es) (x,y) = (x,y) `elem` es

connectA :: (Int, Int) -> Model -> Model
connectA (x, y) (M n es)
  | x < n && y < n = M n ((x,y):(y,x):es)
  | otherwise      = M n es
```

Question 5

What is an appropriate refinement relation to connect our abstract model and our matrix implementation?

1. ✗

```
ref :: Graph -> Model -> Bool
ref g (M n es) = length g == n
              && any (\(x,y) -> ((x,y) `elem` es) == ((g !! x) !! y))
                     [(x,y) | x <- [0..n], y <- [0..n]]
```

2. ✗

```
ref :: Graph -> Model -> Bool
ref g (M n es) = all (\(x,y) -> ((x,y) `elem` es) == ((g !! x) !! y))
                     [(x,y) | x <- [0..n-1], y <- [0..n-1]]
```

3. ✓

```
ref :: Graph -> Model -> Bool
ref g (M n es) = length g == n
              && all (\(x,y) -> ((x,y) `elem` es) == ((g !! x) !! y))
                     [(x,y) | x <- [0..n-1], y <- [0..n-1]]
```

4. ✗

```
ref :: Graph -> Model -> Bool
ref g (M n es) = length g == n
              && all (\(x,y) -> (g !! x) !! y) es
```

The first answer uses `any` rather than `all`, and its bounds are off by one. The second answer does not ensure that the number of nodes in the two graphs are the same. The fourth answer only looks at edges that are in the abstract graph. It would allow edges to be present in the matrix but not in the abstract graph. The third answer is correct.

Question 6

As discussed in lectures, refinement relations aren't that useful for QuickCheck, because it's unlikely to generate two related inputs by chance. Which of the following approaches would remedy this?

1. ✗

   Use this abstraction function:

```
toAbstract :: Graph -> Model
toAbstract g = let n = length g
```

```
                in M n $ filter (\(x,y) -> (g !! x ) !! y)
                                [(x,y) | x <- [0..n-1], y <- [0..n-1]]
```

2. ✗
   Use this abstraction function:

```
toAbstract :: Graph -> Model
toAbstract g = let n = length g
                in M n $ filter (\(x,y) -> (g !! x ) !! y)
                                [(x,y) | x <- [0..n], y <- [0..n]]
```

3. ✓
   Use a refinement function:

```
toConcrete :: Model -> Graph
toConcrete (M n es)
   = map (\x -> map (\y -> (x,y) `elem` es) [0..n-1]) [0..n-1]
```

4. ✗ None of the above.

Using an abstraction function would mean that the resultant graph `connectA (x,y) (toAbstract g)` should be *equal* to `toAbstract (connect (x,y) g)`. This is not the case for either of the provided abstraction functions.

The refinement function is correct.

Code Coverage

Question 7

What are some examples of test *code coverage* measures?

1. ✗ Condition coverage, program coverage, data coverage
2. ✗ Static coverage, dynamic coverage
3. ✗ Memory coverage, execution coverage, control-flow coverage
4. ✓ Function coverage, statement coverage, branch coverage

Statement coverage is sometimes called "expression" coverage in a language like Haskell, which has no statements. Branch coverage is sometimes called "decision" coverage, and is closely related to "condition" coverage which checks the results of evaluating conditions in (e.g.) `if` expressions to determine if conditions have evaluated to both `True` and `False`.

Question 8

Why is full *path coverage* generally infeasible?

1. ✗ Loops may lead to infinite paths
2. ✗ Full path coverage analysis is undecidable in general
3. ✗ The number of paths grows at least exponentially with the size of the computation.
4. ✗ Full path coverage requires simulating the program for every possible input.
5. ✓ All of the above.

This is not to say that fully checking *some properties* for *some programs* is impossible. In general, the field of Model Checking is devoted to solving this kind of intractable problem for restricted models, properties, or both.

Footnotes:

[1] : This is not the same as selecting the properties which are true. Which properties are necessary to maintain our data invariants?

Submission is already closed for this quiz. You can click here to check your submission (if any).