

Quiz (Week 4)

Sorting

Here is an implementation of selection sort in Haskell, as well as a small specification for it in terms of QuickCheck properties.

```
import Test.QuickCheck
import Text.Show.Functions

selSort :: [Int] -> [Int]
selSort xs = go xs []
  where
    -- At each iteration, we remove the maximum element
    -- from the list and put it at the front of our
    -- accumulator.
    -- Once we have removed all the maximums, we return
    -- our accumulator which is in sorted order.
    go [] acc      = acc
    go (x:xs) acc = let (max,xs') = removeMax x xs
                     in go xs' (max:acc)

    -- m is the current maximum. If x is bigger, m is placed
    -- into the list and x becomes the new current maximum
    -- in the recursive call.
    removeMax m [] = (m, [])
    removeMax m (x:xs) = let
      (max,rest) = removeMax m' xs
      (m',x') = if m <= x then (x, m) else (m, x)
    in (max, x':rest)

prop_sort xs = isSorted (selSort xs)
  where
    isSorted (x1 : x2 : xs) = (x1 <= x2) && isSorted (x2 : xs)
    isSorted _ = True

prop_perm xs x = count x xs == count x (selSort xs)
  where
    count x xs = length (filter (== x) xs)
```

Suppose we generalised `selSort` to a function that is parameterised by the ordering relation used, with `selSortBy` :

```
type OrderingFunction = Int -> Int -> Bool

selSortBy :: OrderingFunction -> [Int] -> [Int]
selSortBy o xs = go xs []
  where
    go [] acc      = acc
    go (x:xs) acc = let (max,xs') = removeMax x xs
                     in go xs' (max:acc)

    removeMax m [] = (m, [])
    removeMax m (x:xs) = let
      (max,rest) = removeMax m' xs
      (m',x')    = if m `o` x then (x, m) else (m, x)
    in (max, x':rest)

prop_sort' o xs = isSortedBy o (selSortBy o xs)
  where
    isSortedBy o (x : xs) = all (\y -> x `o` y) xs && isSortedBy o xs
    isSortedBy _ _ = True

prop_perm' o xs x = count x xs == count x (selSortBy o xs)
  where
    count x xs = length (filter (== x) xs)
```

If we try and run `quickCheck prop_sort'` however, we will find that QuickCheck will report numerous counterexamples! This is because `selSortBy` has a *precondition* on the provided function `o`. Not every function of type `Int -> Int -> Bool` meets the precondition.

Question 1

Check all the preconditions required for the ordering function `o` given to `selSortBy`, such that `prop_sort' o` and `prop_perm' o` both pass.

1. ☒ **Irreflexivity** - `not (x `o` x)` for all `x`
2. ☒ **Anti-symmetry** - `x `o` y && y `o` x` implies `x == y` for all `x` and `y`.
3. ☒ **Totality** - `x `o` y || y `o` x` for all `x` and `y`.
4. ☒ **Symmetry** - `x `o` y == y `o` x` for all `x` and `y`.
5. ☒ **Transitivity** - `x `o` y && y `o` z` implies `x `o` z` for all `x`, `y` and `z`.

Any property that is violated by `(<=)`, the operator we used originally in `selSort`, can be immediately ruled out. This rules out irreflexivity and symmetry. Because selection sort works by finding the maximum, we require totality (number 3), as otherwise a given list is not guaranteed to have a single maximum value. We can see that transitivity (number 5) is required as we store only one maximum at a time. Given an operator `o` we start with an initial maximum `a`, and we find a `b` in the list such that `a `o` b`, so our new maximum is `b`. Then we find a `c` in the list such that `b `o` c`. Without transitivity, we couldn't say for certain that `a `o` c` and thus wouldn't be able to use `c` as the new maximum. Anti-symmetry (number 2) is not required, and indeed we may want to use a relation that is not anti-symmetric (such as sorting numbers by their absolute value).

Question 2

QuickCheck provides a function `elements` with the following type signature:

```
elements :: [a] -> Gen a
```

It provides an easy way to make generators by randomly selecting from a list of provided values. Choose a generator that would be suitable for creating an `o : OrderingFunction` for `selSortBy o`, such that it passes `prop_sort' o` and `prop_perm' o`:

1. ☒ `elements [(<),(==),(>)]`
2. ☒ `elements [(<),(<=),(>),(>=)]`
3. ☒ `elements [(<=),(>=), (\x y -> y `mod` x == 0)]`
4. ☒ `elements [(<=),(>=), (\x y -> show x <= show y)]`
5. ☒ `elements [(<=),(/=)]`

Option 1 is incorrect as none of the given operators are total. Option 2 is incorrect as the strict inequality operators are not total. Option 3 is incorrect because the divisibility operator is not total. Option 4 is correct as all three operators are total and transitive. Option 5 is incorrect as `(/=)` is neither transitive nor total.

Sorting Stability

Suppose we further generalised `selSortBy` to accept lists of any type, not just `Int` :

```
type OrderingFunctionFor a = a -> a -> Bool

selSortBy' :: OrderingFunctionFor a -> [a] -> [a]
selSortBy' o xs = go xs []
  where
    go [] acc      = acc
    go (x:xs) acc = let (max,xs') = removeMax x xs
                     in go xs' (max:acc)

    removeMax m [] = (m, [])
    removeMax m (x:xs) = let
      (max,rest) = removeMax m' xs
      (m',x') = if m `o` x then (x, m) else (m, x)
    in (max, x':rest)

isSortedBy :: OrderingFunctionFor a -> [a] -> Bool
isSortedBy o (x : xs) = all (\y -> x `o` y) xs && isSortedBy o xs
isSortedBy _ _ = True

-- We restrict the type to Int here, to give QC something to generate.
prop_sort'' :: OrderingFunctionFor Int -> [Int] -> Bool
prop_sort'' o xs = isSortedBy o (selSortBy' o xs)

prop_perm'' :: OrderingFunctionFor Int -> [Int] -> Int -> Bool
prop_perm'' o xs x = count x xs == count x (selSortBy' o xs)
  where
    count x xs = length (filter (== x) xs)
```

This would allow us to, for example, sort playing cards by their value:

```
data Suit = Hearts | Diamonds | Clubs | Spades
data Card = Card { value :: Int, suit :: Suit }

valueLTEQ :: Card -> Card -> Bool
valueLTEQ x y = value x <= value y

test = selSortBy' valueLTEQ
      [Card 7 Spades,
       Card 5 Spades,
       Card 2 Hearts,
       Card 5 Hearts]
```





This however introduces the complication of sorting stability. It is often desirable for sorting algorithms to be *stable*, that is, they should not alter the relative ordering of

equivalent elements¹. For example, the above `test` example should, under a stable sort, return a list where the five of `Spades` always occurs *before* the five of `Hearts`, even though the comparison function judges them to be equivalent.

Question 3

Expressing sorting stability as a QuickCheck property is rather difficult, however it is acceptable sometimes to make *over-approximations*. In this case, it suffices to write a property that would *always* pass given a stable sorting algorithm and *probably* fail given an unstable one.

Which of the following properties is the best over-approximation of sorting stability?

1.  `selSortBy' o (selSortBy' o xs) == selSortBy' o xs`
2.  `selSortBy' o xs == selSortBy' o (reverse xs)`
3.  `isSortedBy o xs ==> selSortBy' o xs == xs`
4.  `reverse (selSortBy' o xs) == selSortBy' (flip o) xs`

Option 1 (idempotence) is an overapproximation of stability, but a very poor one, seeing as almost any correct sorting function would pass this test, stable or otherwise.

Option 2 expresses that the exact same list would occur regardless of the initial order of the input. This is the exact opposite of sorting stability.

Option 3 is an over-approximation of stability, as it states that `selSortBy' o` would not affect the order of equivalent elements given an already-sorted list. This would still pass unstable sort functions that first check if the list is sorted, however, so it's not an exact characterisation.

Option 4 once again expresses that the order of elements is entirely determined by the sort function and the ordering function, and is not at all influenced by the initial order of elements, which contradicts stability.

Thus, Option 3 is the best overapproximation of stability.

Question 4

What additional precondition on the provided relation `o`, if any, is required to ensure that `selSortBy'` is a stable sort?

1.  **Reflexivity** - `x `o` x` for all `x`
2.  **Anti-symmetry** - `x `o` y && y `o` x` implies `x == y` for all `x` and `y`.

3. **✗ Totality** - $x \leq y \vee y \leq x$ for all x and y .
4. **✗ Symmetry** - $x \leq y \implies y \leq x$ for all x and y .
5. **✗ Transitivity** - $x \leq y \wedge y \leq z$ implies $x \leq z$ for all x , y and z .
6. **✓** No additional preconditions are required.

While traditional in-place swap-based selection sort is famously unstable, the immutable linked-list version presented here is in fact already stable. This is because we start searching for the maximum from the beginning of the list, and prepend it to our accumulator in each iteration. Thus elements that occur towards the end of the list are prepended *earlier* to our accumulator, and thus elements that occur towards the end of the list appear towards the end in our result.

If the sort function used is not stable, ensuring that the ordering relation \leq is *anti-symmetric* is sufficient to ensure stability, but this essentially means that the ordering function must examine *all* of the data being compared, which, in the case of key-value stores or other use-cases where stability is important, is not usually the case.

Minimal Specifications

Question 5

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

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 :: OrderedList Int -> OrderedList Int -> Property
prop_1 (Ordered xs) (Ordered ys) = sorted (merge xs ys)

prop_2 :: [Int] -> [Int] -> Bool
```

```

prop_2 xs ys = length (merge xs ys) == length xs + length ys

prop_3 :: OrderedList Int -> OrderedList Int -> Property
prop_3 (Ordered xs) (Ordered ys) = merge xs ys == sort (xs ++ ys)

prop_4 :: (Int -> Int) -> [Int] -> [Int] -> Bool
prop_4 f xs ys = sort (map f (merge xs ys))
                == sort (merge (map f xs) (map f ys))

prop_5 :: (Int -> Bool) -> OrderedList Int -> OrderedList Int -> Property
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` , and `prop_3`
2. ✗ `prop_1` , `prop_2` , `prop_3` and `prop_4`
3. ✗ Just `prop_3`
4. ✓ `prop_2` , `prop_3` , and `prop_4`
5. ✗ `prop_1` , `prop_2` , `prop_3` , `prop_4` and `prop_5` – there are no redundant properties.

Property 5 is redundant, as we can use property 3 to construct an equational proof of property 5:

```

-- Assuming sorted xs && sorted ys
filter f (merge xs ys)
== -- prop_3 (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_3 [sym] (as sorted (filter f xs) && sorted (filter f ys))
   merge (filter f xs) (filter f ys)

```

Property 1 is also redundant due to property 3:

```
-- Assuming sorted xs && sorted ys
sorted (merge xs ys)
== -- prop_3 (as sorted xs && sorted ys)
   sorted (sort (xs ++ ys))
== -- sort is sorted
   True
```

Property 2 and 4 may at first seem similarly redundant due to property 3, 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 3 and 4 despite not passing property 2. And a function that replaces all elements with the minimum when the input lists are not sorted could still pass properties 2 and 3 despite not passing 4. Thus the correct answer is option 4.

Question 6

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 ys = rev (xs ++ ys) == rev ys ++ rev xs

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

prop_3 :: [Int] -> Int -> Bool
prop_3 xs x = count x xs == count x (rev xs)
  where
    count x xs = length (filter (== x) xs)
```

Which of the above properties is redundant?

1. 
2. 
3. 
4.  None of the above.

Property 3 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 2 is redundant.

Code Coverage

Question 7

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

1. ✓ Function coverage, statement coverage, branch coverage
2. ✗ Static coverage, dynamic coverage
3. ✗ Memory coverage, execution coverage, control-flow coverage
4. ✗ Condition coverage, program coverage, data 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 Two values a and b are *equivalent* iff $a \sim b$ and $b \sim a$.

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