Quiz (Week 3)

Properties for Functions

Question 1

The ROT13 Cipher is a simple substitution cipher which rotates the alphabet by thirteen places, that is, `A` becomes `N` and `Z` becomes `M`. Here is a simple (rather inefficient) Haskell implementation:

```haskell
rot13 :: String -> String
rot13 = map $ \x ->
          case lookup x table of
            Just y  -> y
            Nothing -> x
  where
    table = table' 'A' 'Z' ++ table' 'a' 'z'
    table' a z = zip [a..z] (drop 13 (cycle [a..z]))
```

Select all the properties that this function satisfies (assuming ASCII strings).

1. ✔ `length x == length (rot13 x)`
2. ✔ `rot13 (map toUpper x) == map toUpper (rot13 x)`
3. ✗ `rot13 (map f x) == map f (rot13 x)`
4. ✔ `all (not . isAlpha) x ==> rot13 x == x`
5. ✔ `rot13 (a ++ b) == rot13 a ++ rot13 b`
6. ✗ `not (null x) ==> ord (head x) + 13 == ord (head (rot13 x))`
7. ✔ `rot13 (rot13 x) == x`

Famously, the ROT13 cipher is an *involution*, that is, it is its own inverse. This makes property 7 true.

Property 6 is false, as the difference in ASCII codes can be very different from 13, for example, the space character is left unaltered by ROT13 and so the difference may be zero.

Property 5 is true, as concatenating two ciphered strings is the same as ciphering their concatenation.

Property 4 is true, as rot13 does not affect any characters except alphabetical ones.

Property 3 does not hold, for example when `f` is the `succ` function.

Property 2 holds as case is preserved across ciphering (capital letters are changed to other capital letters, and lowercase letters are changed to other lowercase letters).

Property 1 is also true, as the ciphertext is always the same length as the plaintext in a substitution cipher.

Question 2

Here is a function that fairly merges ordered lists, e.g. `merge [2,4,6,7] [1,2,3,4] == [1,2,2,3,4,4,6,7]`.

```
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
```

Select all the properties that this function satisfies.

1. ✔ `merge (sort a) (sort b) == sort (merge a b)`
2. ✗ `merge a b == sort (a ++ b)`
3. ✔ `length (merge a b) == length a + length b`
4. ✗ `merge (filter f a) (filter f b) == filter f (merge a b)`
5. ✗ `merge (map f a) (map f b) == map f (merge a b)`
6. ✔ `sort (merge a b) == sort (a ++ b)`

Property 1 is true, as given two sorted lists (`sort a` and `sort b`), `merge` should produce a sorted list containing all of the elements of the original lists. Even if the input lists are not sorted, all elements will still be contained in the output, so `sort`-ing `merge a b` will produce the same result.

Property 2 is false, for example when `a` is `[3,1]` and `b` is `[2,3]`.

Property 3 is true, as the `merge` always contains all elements from the input list.

Property 4 is false, for example when `a` is `[3,1]` and `b` is `[2,3]`, and `f` is `(/= 3)`. The two filtered lists are `[1]` and `[2]`, resulting in a merge of `[1,2]` whereas merging first would result in the list `[2,1]`.

Property 5 is false, for example if `f` is the absolute value function `abs`, then any lists involving both negative and positive numbers will result in different orderings.

Property 6 is true, as a merge contains all the elements of both lists, as does a concatenation, and `sort` canonicalises their order.

Question 3

The following code converts Haskell `Int` values to and from strings containing their binary representation (as a sequences of `'1'` and `'0'` characters).

```
toBinary :: Int -> String
toBinary 0 = ""
toBinary n = let (d,r) = n `divMod` 2
              in toBinary d
                  ++ if r == 0 then "0"
                                else "1"

fromBinary :: String -> Int
fromBinary = fst . foldr eachChar (0,1)
  where
    eachChar '1' (sum, m) = (sum + m, m*2)
    eachChar _   (sum, m) = (sum    , m*2)
```

Select all properties that these functions satisfy.

1. ✔ `i >= 0 ==> fromBinary (toBinary i) == i`
2. ✗ ``all (`elem` "01") s ==> toBinary (fromBinary s) == s``
3. ✗ ``all (`elem` "01") s ==> read s >= fromBinary s``
4. ✔ `i > 0 ==> length (toBinary i) >= length (show i)`
5. ✔ ``all (`elem` "01") s ==> fromBinary s == fromBinary ('0':s)``

Property 1 is true as converting to a binary string and then back should result in the same number.

Property 2 is false as while `toBinary` is injective (there is a unique string for every number), `fromBinary` is not, even if the strings are restricted to binary digits. For example, adding any number of leading zeroes onto the binary string will result in the same number from `fromBinary`, so a counterexample to this property can easily be found with `s = "01"`.

Property 3 is false as `read` will throw an exception when given the empty list for `s`.

Property 4 is true, as for *positive* (i.e. nonzero) integers the `toBinary` representation will always be longer than the decimal string representation.

Property 5 is true, as adding leading zeroes to a binary number does not change its value.

Question 4

The following function removes adjacent duplicates from a list.

```
dedup :: (Eq a) => [a] -> [a]
dedup (x:y:xs) | x == y = dedup (y:xs)
               | otherwise = x : dedup (y:xs)
dedup xs = xs
```

Assume the presence of the following `sorted` predicate:

```
sorted :: (Ord a) => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted xs = True
```

Select all properties that `dedup` satisfies.

1. ✔ `sorted xs ==> sorted (dedup xs)`
2. ✔ `sorted xs ==> dedup xs == nub xs`
3. ✔ `sorted xs ==> dedup (dedup xs) == dedup xs`
4. ✗ `sorted xs && sorted ys ==> dedup xs ++ dedup ys == dedup (xs ++ ys)`
5. ✗ `sorted xs ==> length (dedup xs) < length xs`
6. ✔ ``(x `elem` xs) == (x `elem` dedup xs)``

All of these properties are true except 4, as can be seen when both `xs` and `ys` are just the singleton list `[1]`.

Property 1 is true as removing adjacent duplicates from a sorted list does not ruin the sorted ordering.

Property 2 is true as for sorted lists, removing adjacent duplicates and removing all duplicates are identical.

Property 3 is true as removing adjacent duplicates shouldn't find any more adjacent duplicates the second time around.

Property 5 is false as a list that already has no duplicates will not get any shorter.

Property 6 is true as `dedup` will not remove the last of any given value in the list.

Functions for Properties

Question 5

Here are a set of properties that the function `foo` must satisfy:

```
foo :: [a] -> (a -> b) -> [b]
foo = undefined -- see below

prop_1 :: [Int] -> Bool
prop_1 xs = foo xs id == xs

prop_2 :: [Int] -> (Int -> Int) -> (Int -> Int) -> Bool
prop_2 xs f g = foo (foo xs f) g == foo xs (g . f)
```

Choose an implementation for `foo` that satisfies the above properties, and type-checks:

1. ✗

   ```
   foo xs f = []
   ```

2. ✗

   ```
   foo xs f = xs
   ```

3. ✗

   ```
   foo [] f = []
   foo (x:xs) f = x : foo xs f
   ```

4. ✓

   ```
   foo [] f = []
   foo (x:xs) f = f x : foo xs f
   ```

5. ✗

   ```
   foo [] f = []
   foo (x:xs) f = foo xs f
   ```

These are the standard laws (*functor* laws) that `map` has to obey. And, indeed, the correct answer is a `map` implementation.

Note that it is actually impossible to write a terminating function that typechecks and obeys those properties *without* correctly implementing `map`. Try to write an incorrect, terminating `map` that is well-typed and obeys those laws! You will find it is impossible. Later on in the course we will discuss why this is so and how we can exploit it to write better programs.

Question 6

```
bar :: [Int] -> [Int]
bar = undefined

prop_1 :: [Int] -> Bool
prop_1 xs = bar (bar xs) == xs

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

prop_3 :: [Int] -> (Int -> Int) -> Bool
prop_3 xs f = bar (map f xs) == map f (bar xs)
```

Choose all implementations for `bar` that satisfy the above properties, and type-check:

1. ✗

```
bar []     = []
bar (x:xs) =      bar (filter (<=x) xs)
           ++ x : bar (filter (> x) xs)
```

2. ✔

```
bar xs = go xs []
  where go []     acc = acc
        go (x:xs) acc = go xs (x:acc)
```

3. ✗

```
bar []     = []
bar (x:xs) = xs ++ [x]
```

4. ✔

```haskell
bar = id
```

5. ✗

```haskell
bar xs = nub xs
```

6. ✗

```haskell
bar xs = replicate (length xs) (maximum xs)
```

The first property says the function has to be an *involution*, that is, applying it twice should have the same effect as not applying it at all. Property 2 says that the number of elements must remain the same. And property 3 says that `bar` must commute with `map`: This effectively means that we cannot act upon the contents of the list, as this would allow the function given to `map` to be crafted to break this property.

Thus, we must permute the elements of the given list without altering them or changing their quantity, and we must choose the output permutation without inspecting the input values. Thus, the fast reverse implementation in 2, and the identity function in 4 are all correct implementations.

The rotation function in 3 breaks idempotence property 1. The remove duplicates function in 5 breaks the length property in 2. And the replace-with-maximum function in 6 breaks the map-commutation property in 3, for example if `f` is the absolute value function `abs` and the list given is `[2,-4]`. The quicksort function breaks property 3, as the map function could change the relative ordering of the elements.

Question 7

```haskell
baz :: [Integer] -> Integer
baz = undefined

prop_1 :: [Integer] -> [Integer] -> Bool
prop_1 xs ys = baz xs + baz ys == baz (xs ++ ys)

prop_2 :: [Integer] -> Bool
prop_2 xs = baz xs == baz (reverse xs)

prop_3 :: Integer -> [Integer] -> Bool
prop_3 x xs = baz (x:xs) - x == baz xs
```

Choose a law-abiding definition for `baz`, that type checks:

1. ✔

```
baz = foldr (+) 0
```

2. ✗

```
baz []     = 0
baz (x:xs) = 1 + baz xs
```

3. ✗

```
baz []     = 1
baz (x:xs) = x + baz xs
```

4. ✗

```
baz xs = 0
```

This has to be a `sum` function (option 1). The game is almost given away by the first property alone. However `prop_1` by itself allows for a function that merely returns zero (option 4) or a function that returns the length of the list (option 2), however `prop_3` rules these out for us. Option 3 includes an off-by-one error, as the additive identity is 0 not 1, and thus would break `prop_1` for even empty lists.

The `prop_2` property is not really needed here, and is included as a bit of a red herring.

Question 8

Here is a definition of a function `fun`, and properties for another function `nuf`:

```
fun :: [Integer] -> [Integer]
fun []       = []
fun [x]      = []
fun (x:y:xs) = (y-x):fun (y:xs)

nuf :: [Integer] -> Integer -> [Integer]
nuf = undefined

prop_1 :: [Integer] -> Integer -> Bool
prop_1 xs x = nuf (fun (x:xs)) x == (x:xs)
```

```
prop_2 :: [Integer] -> Integer -> Bool
prop_2 xs x = fun (nuf xs x) == xs
```

Choose a definition for `nuf` that type checks and satisfies the given properties:

1. ✗

   ```
   nuf [] i = []
   nuf (x:xs) i = (i + x) : nuf xs (i + x)
   ```

2. ✗

   ```
   nuf [] i = [i]
   nuf (x:xs) i = (i + x) : nuf xs (i + x)
   ```

3. ✓

   ```
   nuf xs i = scanl (\v x -> v + x) i xs
   ```

4. ✗

   ```
   nuf [] i = []
   nuf (x:xs) i = (i + x) : nuf xs i
   ```

5. ✗

   ```
   nuf xs i = i : scanl (\v x -> v + x) i xs
   ```

The `fun` function is essentially a discrete differentiation function, finding the gradient at each data point. Then, `nuf` is described by our properties as its inverse operation (given a constant `c` as integration requires).

If we run `fun [1,4,6,3,6]` we will get the derivative `[3,2,-3,3]`. Trying each possible implementation:

```
*> nuf1 [3,2,-3,3] 1
[4,6,3,6]
*> nuf2 [3,2,-3,3] 1
[4,6,3,6,6]
*> nuf3 [3,2,-3,3] 1
[1,4,6,3,6]
*> nuf4 [3,2,-3,3] 1
[4,3,-2,4]
*> nuf5 [3,2,-3,3] 1
[1,1,4,6,3,6]
```

As can be seen, only `nuf3` gives a correct answer that gets us back to our starting point.

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