CPSC 449 — Principles of Programming Languages
Theory Components of Assignment 2

Name: Haohu Shen
Student ID: 30063099
Tutorial: 03

Problem 3

(a) Prove that the implementation of merge sort in 2(c) always terminates.

Solution. In order to prove the implementation of merge sort in 2(c) always terminates, that is, the *mSort* function terminates for all inputs, we firstly prove all functions that are involved terminate for all inputs, that is, the *splitList*, *getEvenIndexItems*, *getOddIndexItems*, *mergeLists* functions terminate for all inputs. Before giving any prove, the definitions of all functions involved are listed below:

```
mergeLists :: [Integer] -> [Integer] -> [Integer]
mergeLists xs [] = xs -- (mergeLists.1)
mergeLists [] ys = ys -- (mergeLists.2)
mergeLists (x : xs) (y : ys) -- (mergeLists.3)
   | x <= y = x : mergeLists xs (y : ys)
   | otherwise = y : mergeLists (x : xs) ys

getEvenIndexItems :: [Integer] -> [Integer]
getEvenIndexItems [] = [] -- (getEvenIndexItems.1)
getEvenIndexItems (x:_:xs) = x:getEvenIndexItems xs -- (getEvenIndexItems.2)
getEvenIndexItems (x:xs) = x:getEvenIndexItems xs -- (getEvenIndexItems.3)

getOddIndexItems :: [Integer] -> [Integer]
getOddIndexItems [] = [] -- (getOddIndexItems.1)
getOddIndexItems (_:x:xs) = x:getOddIndexItems xs -- (getOddIndexItems.2)
getOddIndexItems (_:xs) = getOddIndexItems xs -- (getOddIndexItems.3)

splitList :: [Integer] -> ([Integer], [Integer])
splitList [] = ([], []) -- (splitList.1)
splitList xs = (getEvenIndexItems xs, getOddIndexItems xs) -- (splitList.2)

mSort :: [Integer] -> [Integer]
mSort [] = [] -- (mSort.1)
mSort [x] = [x] -- (mSort.2)
mSort xs = mergeLists (mSort ys) (mSort zs) -- (mSort.3)
  where
    (ys, zs) = splitList xs
```

i. Firstly we prove that *getEvenIndexItems* function terminates for all inputs, in order to do that, we define a function *rankGetEvenIndexItem* that maps the input list to the length of the list, which is a non-negative integer. Consider that the input list xs has length of n, then:

```
rankGetEvenIndexItem :: Integer -> Integer
```

```
rankGetEvenIndexItem 0 = 0 -- (rankE.1)
rankGetEvenIndexItem (1 + n) = 1 + rankGetEvenIndexItem n -- (rankE.2)
```

Thus, for the 2nd equation in the definition of *getEvenIndexItems* such that it involves 1 recursive call, we prove that
$\mathit{rankGetEvenIndexItem}(2 + \mathrm{n}) > \mathit{rankGetEvenIndexItem}\ \mathrm{n}$:

```
rankGetEvenIndexItem (2 + n)
= rankGetEvenIndexItem (1 + (1 + n))              by arith.
= 1 + rankGetEvenIndexItem (1 + n)                by (rankE.2)
= 1 + (1 + rankGetEvenIndexItem n)                by (rankE.2)
= 2 + rankGetEvenIndexItem n                      by arith.
> rankGetEvenIndexItem n                          by arith.
```

As required.
On the other hand, for the 3rd equation in the definition of *getEvenIndexItems* such that it involves 1 recursive call, we prove that
$\mathit{rankGetEvenIndexItem}\ (1 + \mathrm{n}) > \mathit{rankGetEvenIndexItem}\ \mathrm{n}$:

```
rankGetEvenIndexItem (1 + n)
= 1 + rankGetEvenIndexItem n                      by (rankE.2)
> rankGetEvenIndexItem n                          by arith.
```

As required.
Therefore, the function *rankGetEvenIndexItem* maps the function *getEvenIndexItems* to a natural number and there is a strict decrease of *rankGetEvenIndexItem* when the recursion occurs, that is, we can say the function *getEvenIndexItems* always terminates for all inputs.

ii. Secondly we prove that *getOddIndexItems* function terminates for all inputs, in order to do that, we define a function *rankGetOddIndexItem* that maps the input list to the length of the list, which is a non-negative integer. Consider that the input list xs has length of n, then:

```
rankGetOddIndexItem :: Integer -> Integer
rankGetOddIndexItem 0 = 0 -- (rankO.1)
rankGetOddIndexItem (1 + n) = 1 + rankGetOddIndexItem n -- (rankO.2)
```

Thus, for the 2nd equation in the definition of *getOddIndexItems* such that it involves 1 recursive call, we prove that
$\mathit{rankGetOddIndexItem}\ (2 + \mathrm{n}) > \mathit{rankGetOddIndexItem}\ \mathrm{n}$:

```
rankGetOddIndexItem (2 + n)
rankGetOddIndexItem (1 + (1 + n))                by arith.
= 1 + rankGetOddIndexItem (1 + n)                by (rankO.2)
= 1 + (1 + rankGetOddIndexItem n)                by (rankO.2)
= 2 + rankGetOddIndexItem n                      by arith.
> rankGetOddIndexItem n                          by arith.
```

As required.
On the other hand, for the 3rd equation in the definition of *getOddIndexItems* such that it involves 1 recursive call, we prove that
$\mathit{rankGetOddIndexItem}\ (1 + \mathrm{n}) > \mathit{rankGetOddIndexItem}\ \mathrm{n}$:

```
rankGetOddIndexItem (1 + n)
= 1 + rankGetOddIndexItem xs                     by (rankO.2)
```

```
> rankGetOddIndexItem xs                                         by arith.
```

As required.
Therefore, the function *rankGetOddIndexItem* maps the function *getOddIndexItems* to a natural number and there is a strict decrease of *rankGetOddIndexItem* when the recursion occurs, that is, we can say the function *getOddIndexItems* always terminates for all inputs.

iii. Thirdly we prove that *splitList* function terminates for all inputs, in order to do that, we define a function *rankSplitList* that maps the input list to the length of the list, which is a non-negative integer. Consider that the input list *xs* has length of n, then:

```haskell
rankSplitList :: Integer -> Integer
rankSplitList 0 = 0 -- (rankS.1)
rankSplitList n = rankGetEvenIndexItem n + rankGetOddIndexItem n --
    (rankS.2)
```

Thus, for the 2nd equation in the definition of *splitList* such that it involves 1 recursive call, the length of the input list is at least 1, thus

- **Case 1:** If the length of input is at least 2, then we prove that we prove that *rankSplitList* (2 + n) > *rankSplitList* n.

```
rankSplitList (2+n)
= rankGetEvenIndexItem(2+n)+rankGetOddIndexItem(2+n)         by (rankS.2)
= rankGetEvenIndexItem(1+(1+n))+rankGetOddIndexItem(2+n)     by arith.
= 1+rankGetEvenIndexItem(1+n)+rankGetOddIndexItem(2+n)       by (rankE.2)
= 1+rankGetEvenIndexItem(1+n)+rankGetOddIndexItem(1+(1+n))   by arith.
= 1+rankGetEvenIndexItem(1+n)+1+rankGetOddIndexItem(1+n)     by (rankO.2)
= 2+rankGetEvenIndexItem(1+n)+rankGetOddIndexItem(1+n)       by arith.
= 2+(1+rankGetEvenIndexItem n)+rankGetOddIndexItem(1+n)      by (rankE.2)
= 2+(1+rankGetEvenIndexItem n)+(1+rankGetOddIndexItem n)     by (rankO.2)
= 4+(rankGetEvenIndexItem n + rankGetOddIndexItem n)         by arith.
= 4+rankSplitList n                                          by (rankS.2)
> rankSplitList n                                            by arith.
```

- **Case 2:** Otherwise, the length of input is 1, then we prove that *rankSplitList* (1+n) > *rankSplitList* n.

```
rankSplitList (1+n)
= rankGetEvenIndexItem (1+n) + rankGetOddIndexItem (1+n)       by (rankS.2)
= (1 + rankGetEvenIndexItem n) + rankGetOddIndexItem (1+n)     by (rankE.2)
= (1 + rankGetEvenIndexItem n) + (1 + rankGetOddIndexItem n)   by (rankO.2)
= 2 + (rankGetEvenIndexItem n + rankGetOddIndexItem n)         by arith.
= 2 + rankSplitList n                                          by (rankS.2)
> rankSplitList n                                              by arith.
```

Therefore, the function *rankSplitList* maps the function *splitList* to a natural number and there is a strict decrease of *rankSplitList* when the recursion occurs, that is, we can say the function *rankSplitList* always terminates for all inputs.

iv. Next we prove that *mergeLists* function terminates for all inputs, in order to do that, we define a function *rankMergeLists* that maps the input of two lists of integers to a non-negative number, since the recursival call only exists in the 3rd definition of *mergeList*, we consider its input $(x : xs)$ and $(y : ys)$ as arguments:

- Suppose the length of xs is n and the length of ys is m.
- Then the *rankMergeLists* of the inputs is:

  ```
  rankMergeLists (1+n) (1+m)
  = 1 + rankMergeLists n (1+m)
  > rankMergeLists n (1+m)
  ```

  or

  ```
  rankMergeLists (1+n) (1+m)
  = 1 + rankMergeLists (1+n) m
  > rankMergeLists (1+n) m
  ```

  Such that n (1+m) or (1+n) m are the arguments of *rankMergeLists* in the recursive call on the RHS.

Since there is a strict decrease of value of *rankMergeLists* when recursion occurs, we can say the function *mergeLists* terminates for all inputs.

v. Finally we prove that the function *mSort* terminates for all inputs, in order to do that, we define a function *rank* that maps the input list to the length of the list, which is a non-negative integer. Consider that the input list xs has length of n, then:

```
rank :: Integer -> Integer
rank 0 = 0 -- (rankS.1)
rank 1 = 1 -- (rankS.2)
rank n = rank a + rank b -- (rankS.3)
```

Such that a is the length of *getEvenIndexItems* **xs**, b is the length of *getOddIndexItems* **xs**.
The 3rd equation involves 2 recursive calls. Since the length of xs is n and $n >= 2$ when the equation is called, we have:

- For the first recursive call, the argument of the recursive call is a. Since a is the length of *getEvenIndexItems* **xs** such that $n >= 2$, thus $a < n$. Moreover, *rank* maps the input list to the length of the list, thus *rank* n = n, *rank* a = a, thus *rank* n > *rank* a, which indicates that *rank* a is strictly smaller than the input rank.
- For the second recursive call, the argument of the recursive call is b. Since b is the length of *getOddIndexItems* **xs** such that $n >= 2$, thus $b < n$. Moreover, *rank* maps the input list to the length of the list, thus *rank* n = n, *rank* b = b, thus *rank* n > *rank* b, which indicates that *rank* b is strictly smaller than the input rank.

Since both ranks of argument decrease strictly as the recursion occurs, we can say that the function *mSort* is guaranteed to terminate for all inputs. □

(b) Consider the following function.

```
mystery :: [[Integer]] -> Integer
mystery [] = 0
mystery ((x:xs):ys) = x + mystery (xs:ys)
mystery ([]:ys) = mystery ys
```

i. Give a conjecture of what the *mystery* function returns.
Solution. The *mystery* function takes a list of lists of integers and returns the summation of the sum of all sub-lists.

ii. Prove that the *mystery* function terminates for all inputs.
Proof. In order to prove the function terminates for all inputs, we define a function *rank* that maps the input list to the sum of total length of sub-lists and the number of sub-lists inside the input list. That is:

```
rank :: [[Integer]] -> Integer
rank [] = 0 -- (rank.1)
rank ((x : xs) : ys) = 1 + rank (xs : ys) -- (rank.2)
rank ([] : ys) = 1 + rank ys -- (rank.3)
```

Thus, for the 2nd equation in the definition of *mystery* such that it involves 1 recursive call, we prove that $rank((x : xs) : ys) > rank(xs : ys)$:

```
rank ((x:xs):ys)
= 1 + rank (xs : ys)                                        by (rank.2)
> rank (xs : ys)                                            by arith.
```

On the other hand, for the 3rd equation in the definition of *mystery* such that it involves 1 recursive call, we prove that $rank([\,] : ys) > rank(ys)$:

```
rank ([]:ys)
= 1 + rank ys                                               by (rank.3)
> rank ys                                                   by arith.
```

Therefore, the function *rank* maps the function *mystery* to a natural number and there is a strict decrease of *rank* when the recursion occurs, that is, we can say the function *mystery* always terminates for all inputs.

Problem 4 Prove, by structural induction, for all finite lists xs, we have:

```
length xs = length (reverse xs)  -- (*)
```

You may assume the following equations for *reverse* and *length*:

```
reverse [] = [] -- (reverse.1)
reverse (x : xs) = (reverse xs) ++ [x] -- (reverse.2)

length [] = 0 -- (length.1)
length (x : xs) = 1 + (length xs) -- (length.2)
length (xs ++ ys) = (length xs) + (length ys) -- (length.3)
```

Adhere to the following steps in your proof.

(a) State the two proof goals (i.e., base case and induction step).
Proof Goals

- **Base Case**

```
length [] = length (reverse [])                             (base)
```

- **Induction Step**
  - **Assume:**

```
length xs = length (reverse xs)                                   (hyp)
```
– **Prove:**
```
length (x:xs) = length (reverse (x:xs))                           (ind)
```

(b) Prove the base case.

- **Want:**
```
length [] = length (reverse [])
```
- **Left-hand side:**
```
length []
= 0                                                    by (length.1)
```
- **Right-hand side:**
```
length (reverse [])
= length []                                            by (reverse.1)
= 0                                                    by (length.1)
= L.H.S.
```
Thus it shows that the two sides are the same, which completes the proof of the base case.

(c) Prove the induction step.

- **Assume:**
```
length xs = length (reverse xs)                                   (hyp)
```
- **Want:**
```
length (x:xs) = length (reverse (x:xs))
```
- **Left-hand side:**
```
length (x:xs)
= 1 + (length xs)                                      by (length.2)
= 1 + length xs
= 1 + length (reverse xs)                                    by (hyp)
```
- **Right-hand side:**
```
length (reverse (x:xs))
= length ((reverse xs) ++ [x])                         by (reverse.2)
= (length (reverse xs)) + (length [x])                 by (length.3)
= length (reverse xs) + length [x]
= length (reverse xs) + length (x : [])                by defn. of [x]
= length (reverse xs) + (1 + (length []))              by (length.2)
= length (reverse xs) + (1 + length [])
= length (reverse xs) + (1 + 0)                        by (length.1)
= length (reverse xs) + 1
= 1 + length (reverse xs)                                    by arith.
= L.H.S.
```
Since the final step makes the left- and right-hand sides equal, on the assumption that the induction hypothesis holds, This completes the induction step, and therefore the proof itself. □