Haskell (IR3) – Fonctions

Stéphane Vialette

4 octobre 2019

Question 1: Permutations

(a) Écrire la fonction `mirror :: Eq a => [a] -> [a] -> Bool` qui retourne `True` si et seulement si les deux listes sont en image miroir.

(b) Écrire la fonction `permute :: Eq a => [a] -> [a] -> Bool` qui retourne `True` si et seulement si les deux listes sont les permutations d'une même liste. Le temps d'exécution doit être quadratique.

(c) Écrire la fonction `permute' :: Ord a => [a] -> [a] -> Bool` qui retourne `True` si et seulement si les deux listes sont les permutations d'une même liste. (Cette dernière version doit être plus efficace : $O(n \log n)$.)

Question 2: Renversements (ou Permutations particulières)

Les *renversements* forment une famille de permutations qui correspondent à celles qui remplacent un segment d'indices par son image miroir. Plus précisément le renversement (i, j), $1 \leq i \leq j \leq n$ correspond à la permutation $(1, \ldots, i-1, j, j-1, \ldots, i+1, i, j+1, j+2, \ldots, n-1, n)$. En l'appliquant sur un mot u, le renversement $\sigma = (i, j)$ remplace $w = w_1 \ldots w_n$ par $\sigma(w) = w_1 \ldots w_{i-1} w_j w_{j-1} \ldots w_{i+1} w_i w_j w_{j+1} \ldots w_n$.

Un renversement (i, j) est dit *préfixe* si $i = 1$; il est dit *suffixe* si $j = n$.

(a) Écrire la fonction `reversal :: Int -> Int -> [a] -> [a]` qui retourne le renversement d'une liste. Le premier argument donne la position de départ du renversement dans la liste (les indices commencent à 0 !!!) et le second donne la position de fin du renversement.

(b) Il sera parfois plus commode d'utiliser la position de départ du renversement ainsi que sa longueur. Écrire la fonction `reversal' :: Int -> Int -> [a] -> [a]` qui retourne le renversement d'une liste. Le premier argument donne la position de départ du renversement dans la liste (les indices commencent toujours à 0 !!!) et le second donne la longueur du renversement.

(c) Écrire la fonction `prefixReversal :: Int -> [a] -> [a]` qui retourne le renversement préfix d'une liste. Le premier argument donne la longueur du renversement préfix.

(d) Écrire la fonction `suffixReversal :: Int -> [a] -> [a]` qui retourne le renversement suffix d'une liste. Le premier argument donne la longueur du renversement suffix.

(e) Écrire la fonction `isPrefixReversal :: [a] -> [a] -> Bool` qui retourne `True` si est seulement si la première liste passée en argument peut être obtenue par renversement préfix de la seconde liste passée en argument.

(f) Écrire la fonction `isSuffixReversal :: [a] -> [a] -> Bool` qui retourne `True` si est seulement si la première liste passée en argument peut être obtenue par renversement suffix de la seconde liste passée en argument.

(g) La fonction `isPrefixOf :: Eq a => [a] -> [a] -> Bool` définie dans `Prelude` retourne `True` si set seulement si le première liste est préfix de la seconde. Réécrire la fonction `isPrefixReversal :: [a] -> [a] -> Bool` en utilisant `isPrefixOf`.

(h) La fonction Écrire la fonction `isReversal :: Eq a => [a] -> [a] -> Bool` qui retourne `True` si set seulement si le première liste peut être obtenue par un renversment de la seconde.

(i) Écrire la fonction `allReversals :: Eq a => [a] -> [[a]]` qui retourne toutes les listes qui peuvent être obtenues par un renversement de la liste passée en argument. Notez que la fonction `nub :: Eq a => [a] -> [a]` définie dans `Data.List` supprime les doublons dans une liste. Réécrire maintenant la fonction `isReversal :: Eq a => [a] -> [a] -> Bool` en utilisant `allReversals`.

(j) Écrire la fonction `isReversalK :: Eq a => [a] -> [a] -> Bool` qui retourne `True` si set seulement si le première liste peut être obtenue par k renversements de la seconde.

(k) Écrire la fonction `reduce :: (Ord a) => [a] -> [Int]` qui prend en argument une liste et qui retourne la permutation correspondante. En cas d'égalité de lettres, la lettre la plus à gauche est supposée inférieure.

Par exemple

```
ghci> reduce ""
[]
ghci> reduce "bzab"
[2,4,1,3]
ghci> reduce [1..5]
[1,2,3,4,5]
```

Question 3: Merge sort

Le principe du tri fusion (merge sort) est très simple. Il consiste à fusionner deux sous-séquences triées en une séquence triée.

Il exploite directement le principe du divide-and-conquer qui repose en la division d'un problème en ses sous problèmes et en des recombinaisons bien choisies des sous-solutions optimales.

Le principe de cet algorithme tend à adopter une formulation récursive :
— On découpe les données à trier en deux parties plus ou moins égales.
— On trie les 2 sous-parties ainsi déterminées.
— On fusionne les deux sous-parties pour retrouver les données de départ.
Donc chaque instance de la récursion va faire appel à nouveau au programme, mais avec une séquence de taille inférieure à trier.

La terminaison de la récursion est garantie, car les découpages seront tels qu'on aboutira à des sous-parties d'un seul élément ; le tri devient alors trivial. Une fois les éléments triés indépendamment les uns des autres, on va fusionner (merge) les sous-séquences ensemble jusqu'à obtenir la séquence de départ, triée.

Écrivez la fonction `mergesort :: (a -> a -> Bool) -> [a] -> [a]` qui implémente le tri fusion. (vous identifierez les deux fonctions
— `split :: [a] -> ([a],[a])` et
— `merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]`
que vous implémenterez dans deux fonctions séparées).