Haskell (IR3) – Listes

Fabian Reiter (fabian.reiter@univ-eiffel.fr)
Stéphane Vialette (stephane.vialette@univ-eiffel.fr)

11 octobre 2020

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: Boite à outils

(a) Écrire la fonction `pairs :: [a] -> [(a, a]` qui groupe les éléments d'une liste par paires chevauchantes.

```
ghci> mapM_ print [pairs [1..n] | n <- [0..5]]
[]
[]
[(1,2)]
[(1,2),(2,3)]
[(1,2),(2,3),(3,4)]
[(1,2),(2,3),(3,4),(4,5)]
```

(b) Écrire la fonction `evenElts :: [a] -> [a]` qui conserve les éléments d'indice pair d'une liste.

```
ghci> mapM_ print [evenElts [1..n] | n <- [0..5]]
[]
[1]
[1]
[1,3]
[1,3]
[1,3,5]
```

(c) Écrire la fonction `subLength :: [[a]] -> [([a], Int)]` qui prend une liste de listes et retourne la liste des paires liste/longueur.

```
ghci> subLength [[1..n] | n <- [1..5]]
[([1],1),([1,2],2),([1,2,3],3),([1,2,3,4],4),([1,2,3,4,5],5)]
```

(d) Écrire la fonction `appOnPairs :: (a ->c) -> (b -> d) -> [(a, b)] -> [(c, d)]` qui prend une liste deux fonctions et une liste de paires, et qui retourne la liste des paires obtenues en applicant les deux fonctions sur le premier et second élément de la paire, respectivement. liste/longueur.

```
ghci> appOnPairs (+1) (*2) [(i, i+10) | i <- [1..5]]
[(2,22),(3,24),(4,26),(5,28),(6,30)]
```

(e) Écrire la fonction `factors :: (Eq a) => [a] -> [[a]]` qui retourne tous les facteurs (suite d'élément consécutifs) distincts d'une liste.

```
ghci> sort $ factors ""
[""]
ghci> sort $ factors "a"
["","a"]
ghci> sort $ factors "abc"
["","a","ab","abc","b","bc","c"]
ghci> sort $ factors "abab"
["","a","ab","aba","abab","b","ba","bab"]
```

Écrire ensuite la fonction `subseqs :: (Eq a) => [a] -> [[a]]` qui retournent toutes les sous-séquences (suite d'élément non nécessairement consécutifs) distinctes d'une liste.

```
ghci> sort $ subseqs ""
[""]
ghci> sort $ subseqs "a"
["","a"]
ghci> sort $ subseqs "abc"
["","a","ab","abc","ac","b","bc","c"]
ghci> p xs ys = not (null xs) && not (null ys) && head xs == head ys
ghci> mapM_ print . groupBy p . sort $ subseqs "abab"
[""]
["a","aa","aab","ab","aba","abab","abb"]
["b","ba","bab","bb"]
```

Question 3: 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 $\sigma(w) = w_1 \ldots w_{i-1} w_j w_{j-1} \ldots w_{i+1} w_i w_{j+1} \ldots w_n$.

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 et 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 4: 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).