Haskell (IR3) – Listes

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

8 novembre 2022

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.

Solution:

```haskell
mirror :: Eq a => [a] -> [a] -> Bool
mirror xs ys = xs == reverse ys
```

Deux propositions moins efficaces, mais pour se familiariser avec les fonctions d'ordre supérieur :

```haskell
mirror :: Eq a => [a] -> [a] -> Bool
mirror xs ys = and (zipWith (==) xs (reverse ys))
```

```haskell
mirror :: Eq a => [a] -> [a] -> Bool
mirror xs ys = all (uncurry (==)) (zip xs (reverse ys))
```

En fait, on préferera les écritures suivantes en haskell :

```haskell
mirror :: Eq a => [a] -> [a] -> Bool
mirror xs = and . zipWith (==) xs . reverse
```

```haskell
mirror :: Eq a => [a] -> [a] -> Bool
mirror xs = all (uncurry (==)) . zip xs . reverse
```

À ne surtout pas faire (pourquoi ?) :

```haskell
mirror :: Eq a => [a] -> [a] -> Bool
mirror = aux
  where
    aux [] [] = True
    aux [] _  = False
    aux _  [] = False
    aux xs ys = head xs == last ys && aux (tail xs) (init ys)
```

(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. Notez que `Ord a` n'apparaît pas dans le type de la fonction.

Solution:
Une première idée est la suivante :

```
permute :: Eq a => [a] -> [a] -> Bool
permute []       _  = True
permute (x : xs) ys = x `elem` ys && permute xs ys
```

Cependant, la fonction `permute` telle qu'elle est proposée ici n'est pas satisfaisante. En effet :

```
λ: permute [1, 1, 2] [1, 2]
True
```

On peut (sans succès) essayer de corriger avec la fonction suivante :

```
permute :: Eq a => [a] -> [a] -> Bool
permute xs ys = length xs == length ys && aux xs ys
  where
    aux []       _  = True
    aux (x : xs) ys = x `elem` ys && aux xs ys
```

```
λ: permute [1, 1, 2] [1, 2]
False
λ: permute [1, 1, 2] [1, 2, 2]
True
```

Une solution (qui fonctionne) est la suivante :

```
permute :: Eq a => [a] -> [a] -> Bool
permute []       [] = True
permute []       _  = False
permute (x : xs) ys = x `elem` ys && permute xs (delete x ys)
```

Elle est cependant très mauvaise, il faut reconstruire une liste à chaque fois (où ?).

Une solution en utilisant `Data.Map` (notez le `Ord a` supplémentaire. Pourquoi est-il nécessaire ?)

```
import qualified Data.Map as M

permute :: (Eq a, Ord a) => [a] -> [a] -> Bool
permute xs ys = aux xs == aux ys
  where
    aux = M.assocs . foldl f M.empty
      where
        f m z = M.insertWith (+) z 1 m
```

(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 : $O(n \log n)$.)

Solution:
C'est finalement beaucoup plus facile en $O(n \log n)$.

```
import Data.List|:

permute' :: Ord a => [a] -> [a] -> Bool
permute' xs ys = sort xs == sort ys
```

Noter ici le `Ord a` qui est nécessaire pour comparer les éléments lors du tri.

Question 2: Boite à outils

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

```
λ: 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)]
```

Solution:
Une version récursive immédiate

```
pairs :: [a] -> [(a, a)]
pairs []             = []
pairs [_]            = []
pairs (x : x' : xs) = (x, x') : pairs (x' : xs)
```

Sur cette voie, on préfèrera néanmoins

```
pairs :: [a] -> [(a, a)]
pairs []                = []
pairs [_]               = []
pairs (x : t@(x' : xs)) = (x, x') : pairs t
```

Haskell est paresseux…

```
pairs :: [a] -> [(a, a)]
pairs xs = zip xs $ tail xs
```

Pourquoi cette deuxième version fonctionne-t-elle correctement pour la liste vide et les listes contenant un unique élément.

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

```
λ: mapM_ print [evenIdxElt [1..n] | n <- [0..5]]
[]
[1]
[1]
[1,3]
```

```
[1,3]
[1,3,5]
```

Solution:

```
evenIdxElt :: [a] -> [a]
evenIdxElt []             = []
evenIdxElt [x]            = [x]
evenIdxElt (x : x' : xs) = x : evenIdxElt xs
```

Plus lisible :

```
evenIdxElt :: [a] -> [a]
evenIdxElt []           = []
evenIdxElt [x]          = [x]
evenIdxElt (x : _ : xs) = x : evenIdxElt xs
```

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

```
λ: 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)]
```

Solution:
Ce n'est pas très difficile avec une compréhension de liste :

```
subLength xss = [(xs, length xs) | xs <- xss]
```

Il est bien sûr possible d'utiliser un `map` (module `Data.List` :—) :

```
subLength :: [[a]] -> [([a], Int)]
subLength = map (\ xs -> (xs, length xs))
```

(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.

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

Solution:
Encore une fois, avec ou une compréhension de liste (remarquez le pattern matching sur l'action) :

```
appOnPairs :: (a -> c) -> (b -> d) -> [(a, b)] -> [(c, d)]
appOnPairs f g xys = [(f x, g y) | <- (x, y) <- xys]
```

ou en utilisant un `map` :

```
appOnPairs :: (a -> c) -> (b -> d) -> [(a, b)] -> [(c, d)]
appOnPairs f g = map (\ (x, x') -> (f x, g x'))
```

Pour les plus curieux d'entre vous (très très curieux !) :

```haskell
import Control.Arrow

appOnPairs' :: Arrow a => a b c -> a b' c' -> a (b, b') (c, c')
appOnPairs' f g = f *** g
```

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

```
λ: sort $ factors ""
[""]
λ: sort $ factors "a"
["","a"]
λ: sort $ factors "abc"
["","a","ab","abc","b","bc","c"]
λ: 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.

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

Solution:
En utilisant uniquement des fonctions de `Data.List` :

```haskell
factors :: (Eq a) => [a] -> [[a]]
factors = nub . aux
  where
    aux [] = [[]]
    aux xs = tail (inits xs) ++ aux (tail xs)
```

ou encore une version récursive :

```haskell
factors :: (Eq a) => [a] -> [[a]]
factors = nub . aux
  where
    aux []       = [[]]
    aux (x : xs) = map (x :) (inits xs) ++ aux xs
```

L'implémentation de `subseqs` est une variation :

```haskell
subseqs :: (Eq a) => [a] -> [[a]]
subseqs = nub . aux
  where
```

```
    aux [] = [[]]
    aux (x : xs) = xss ++ map (x :) xss
      where
        xss = aux xs
```

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.

Solution:

Une première solution :

```
reversal :: Int -> Int -> [a] -> [a]
reversal i j xs = p ++ r ++ s
  where
    p = take i xs
    r = reverse . take (j-i+1) $ drop i xs
    s = drop (j+1) xs
```

Nous pouvons utiliser la fonction `splitAt :: Int -> [a] -> ([a], [a])` de `Data.List` :—. (Où trouver des informations sur cette fonction ?)

```
reversal :: Int -> Int -> [a] -> [a]
reversal i j xs = ps ++ reverse ys ++ ss
  where
    (ps, ss') = splitAt i xs
    (ys, ss)  = splitAt (j-i+1) ss'
```

Pour les plus courageux, comment écrire vous-même `splitAt` ? A l'aide du pattern matching, ce n'est finalement pas très compliqué.

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt 0 xs       = ([],      xs)
splitAt _ []       = ([],      [])
splitAt n (x : xs) = (x : xs', xs'')
  where
    (xs', xs'') = splitAt (n - 1) xs
```

Ou alors plus simplement (et dans l'esprit fonctionnelle) :

```
splitAt n xs = (take n xs, drop n xs)
```

(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.

Solution:
Bien sûr, nous allons réutiliser la fonction `reversal` :

```haskell
reversal' :: Int -> Int -> [a] -> [a]
reversal' i l = reversal i (i+l-1)
```

(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.

Solution:
En réutilisant la fonction `reversal` :

```haskell
prefixReversal :: Int -> [a] -> [a]
prefixReversal l = reversal 0 (l-1)
```

(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.

Solution:
Toujours en utilisant la fonction `reversal` (c'est un aspect essentiel de la programmation fonctionnelle que de réutiliser encore et encore les fonctions simples) :

```haskell
suffixReversal :: Int -> [a] -> [a]
suffixReversal l xs = reversal (n-l) (n-1) xs
  where
    n = length xs
```

Nous pourrions réutiliser `prefixReversal`, la fonction fait par contre ici deux appels à `reverse` :

```haskell
suffixReversal' :: Int -> [a] -> [a]
suffixReversal' l xs = reverse $ prefixReversal l (reverse xs)
```

(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.

Solution:
Une solution, un peu naïve, qui utilise une compréhension de liste.

```haskell
isPrefixReversal :: [a] -> [b] -> Bool
isPrefixReversal xs ys = or tests
  where
    tests = [prefixReversal l xs == ys | l <- [1..length xs]]
```

Une autre solution (plus élégante) est :

```haskell
isPrefixReversal' :: Eq a => [a] -> [a] -> Bool
isPrefixReversal' xs ys = any (ys ==) xs'
  where
    xs' = [prefixReversal l xs | l <- [1..length xs]]
```

(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.

Solution:

```haskell
isSuffixReversal :: [a] -> [a] -> Bool
isSuffixReversal xs ys = or tests
  where
    tests = [suffixReversal l xs == ys | l <- [1..length xs]]
```

ou

```haskell
isSuffixReversal' :: Eq a => [a] -> [a] -> Bool
isSuffixReversal' xs ys = any (ys ==) xs'
  where
    xs' = [suffixReversal l xs | l <- [1..length xs]]
```

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

Solution:

```haskell
isReversal :: Eq a => [a] -> [a] -> Bool
isReversal xs ys = or tests
  where
    n     = length xs
    tests = [reversal i j xs == ys | i <- [0,1..n-1],
                                     j <- [i..n-1]]
```

(h) 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`.

Solution:

```haskell
allReversals :: Eq a => [a] -> [[a]]
allReversals xs = nub [reversal i j xs | i <- [0,1..n-1],
                                         j <- [i..n-1]]
  where
    n = length xs

isReversal' :: Eq a => [a] -> [a] -> Bool
isReversal' xs ys = or [xs' == ys | xs' <- allReversals xs]
```

(i) 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.

Solution:

```haskell
allReversalsK :: (Num a, Eq a1, Eq a) => a -> [a1] -> [[a1]]
allReversalsK k xs = allReversalsK' k [xs]
  where
    allReversalsK' 0 xss = xss
    allReversalsK' k xss = allReversalsK' (k-1) yss
      where
        yss = nub [ys | xs <- xss, ys <- allReversals xs]

isReversalK :: (Num a, Eq a1, Eq a) => a -> [a1] -> [a1] -> Bool
isReversalK k xs ys = ys `elem` allReversalsK k xs
```

(j) 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

```
λ: reduce ""
[]
λ: reduce "bzab"
[2,4,1,3]
λ: reduce [1..5]
[1,2,3,4,5]
```

Solution:

```haskell
import qualified Data.Function as Function

reduce :: (Ord a) => [a] -> [Int]
reduce = map fst . sortByIdx . zip [1..] . sortByElt . zip [1..]
  where
    sortByElt = sortBy (compare `Function.on` snd)
    sortByIdx = sortBy (compare `Function.on` (fst . snd))
```

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).

Solution:

```haskell
-- The mergesort function applies a predicate to a list of items
-- that can be compared using that predicate. For a simple list
-- (one element or empty), we just return a duplicate of the
-- current list. For longer lists, we split the list into two
-- halves, recurse on each half, then merge the two halves
-- according to the predicate
mergesort :: (a -> a -> Bool) -> [a] -> [a]
mergesort pred []  = []
mergesort pred [x] = [x]
mergesort pred xs  = merge pred xs1' xs2'
  where
    (xs1, xs2) = split xs
    xs1' = mergesort pred xs1
    xs2' = mergesort pred xs2

-- Merge is the heart of the algorithm and operates by
-- interleaving the elements of two ordered lists in such a way
-- that the combined list is ordered.
merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
merge pred xs []         = xs
merge pred [] ys         = ys
merge pred (x : xs) (y : ys)
  | pred x y  = x : merge pred xs       (y : ys)
  | otherwise = y : merge pred (x : xs) ys

-- To break the list into two halves without having to first
-- measure its length (an extra traversal) we count in twos
-- over it, and use another pointer into the list to advance in
-- steps of one to get the two halves, keeping the original order
-- to ensure a stable sort.
split :: [a] -> ([a],[a])
split xs = go xs xs where
  go (x : xs) (_:_:zs) = (x : us, vs) where (us, vs) = go xs zs
  go  xs      _        = ([], xs)
```