Haskell (IR3) – Fonctions

Stéphane Vialette

10 janvier 2015

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 mirroir.

Solution:

```
mirror :: Eq a => [a] -> [a] -> Bool
mirror xs ys = xs == reverse 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. Le temps d'exécution doit être quadratique.

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 proposé ici n'est pas satisfaisante. En effet :

```
*Main: permute [1, 1, 2] [1, 2]
True
*Main:
```

On peut corriger avec la fonction suivante :

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

Noter que la fonction s'assure que les listes sont les permutation d'une même liste, mais les arguments (listes) ne sont pas forcément des permutations :

```
*Main: permute [1, 1, 2] [1, 2]
False
*Main: permute [1, 1, 2] [1, 2, 2]
True
*Main:
```

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

Solution:

```
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: 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 *préfix* si $i = 1$; il est dit suffix 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:

```
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'
```

(b) Il sera parfois plus commode d'utiliser la position de départ du renversement ainsi que sa longueur. `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` :

```
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` :

```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:

```haskell
isPrefixReversal :: [a] -> [a1] -> Bool
isPrefixReversal xs ys = True `elem` tests
  where
    tests = [prefixReversal i xs == ys | i <- [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 i xs | i <- [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 = True `elem` tests
  where
    tests = [suffixReversal i xs == ys | i <- [1..length xs]]
```

ou

```haskell
isSuffixReversal' :: Eq a => [a] -> [a] -> Bool
isSuffixReversal' xs ys = any (ys ==) xs'
```

```
  where
    xs' = [suffixReversal i xs | i <- [1..length xs]]
```

(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`.

Solution:

```
isPrefixReversal' :: Eq a => [a] -> [a] -> Bool
isPrefixReversal' xs ys = True `elem` tests
  where
    tests = [isPrefixOf (reverse p) ys | p <- drop 1 (inits xs)]
```

(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.

Solution:

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

(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`.

Solution:

```
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 = True `elem` [xs' == ys | xs' <- allReversals xs]
```

(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.

Solution:

```
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
```

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).

Solution:

```
-- 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
```

```haskell
-- 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)
```