Programmation fonctionnelle

Devoir surveillé intermédiaire

13 février 2017

Durée : 1h15
Documents (notes de cours, TD, TP et mini-haddock) autorisés

Vous composerez votre DS sur deux copies séparées afin de paralléliser la correction.

1 Typage et structures de données

À composer sur la première copie

1.1 Typage

Soient les définitions suivantes :

```
v1   = '1' : "23"
v2   = '1' : [2, 3]
v3   = "123" ++ ['4', '5']
v4   = 2 + 4.5
v5   = map (+)
v6 f = map f "abc"
```

Q 1. Pour chaque v_i, indiquez s'il est bien ou mal typé : s'il est bien typé, donnez son type (le plus général que vous voyez) ; s'il est mal typé, expliquez pourquoi.

Soient les signatures suivantes :

```
f1 :: [a] -> [a]
f2 :: [a] -> [b] -> [(a, b)]
f3 :: (a -> Bool) -> [a] -> Bool
f4 :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Q 2. Donnez une définition pour chaque f_i et expliquez en français ce qu'elle calcule. Le type inféré par GHC sur votre code devra correspondre à la signature attendue.

1.2 Structure de données pour XML

On se propose de définir une structure de données permettant de représenter un document XML. Pour simplifier nous ignorerons les attributs.

La figure 1 donne un exemple de document XML et la structure de données que nous voulons construire pour le représenter.

```
<div>
    <h1>titre</h1>
    <p>contenu <em>important</em> ou pas</p>
</div>
```

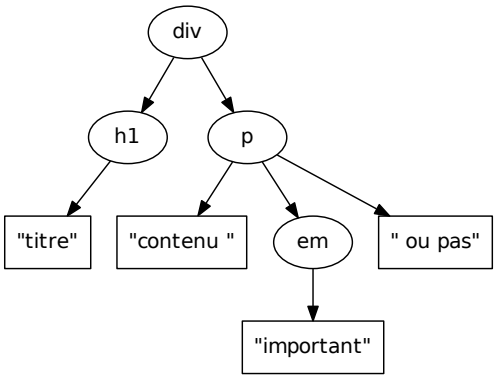

FIG. 1: Document XML et sa représentation

Cette structure devra donc avoir :

— des *balises*, qui ont un nom (`div`, `h1`, etc.) et un nombre quelconque de fils (0 ou plus),
— des éléments de *texte*, sans descendants.

Q 3. Déclarez une structure de données pour représenter de tels documents.

Q 4. Définissez une fonction `taille` (en particulier, donnez son type) permettant de calculer le nombre de balises et d'éléments de *texte* dans un document.

Cette fonction devra ainsi retourner 8 sur l'exemple de la figure 1.

Q 5. Définissez une fonction plus générique de sorte que :

— `taille` soit une application partielle de cette fonction,

— les fonctions qui calculent :

— le nombre d'éléments de texte d'un document (4 dans l'exemple),
— le nombre de `div` d'un document (1 dans l'exemple)

en soient aussi des applications partielles.

Vous donnerez les définitions de ces trois fonctions comme applications partielles de votre fonction générique.

2 Tris génériques

À composer sur la seconde copie

L'objectif de cet exercice est de définir des fonctions de tri génériques.

Bien entendu, vous n'utiliserez pas les fonctions de tri de la bibliothèque standard.

On considère la fonction `quickSort` suivante :

```
quickSort :: Ord a => [a] -> [a]
quickSort     [] = []
quickSort (x:xs) = lesser ++ [x] ++ greater
    where lesser  = quickSort (filter (<= x) xs)
          greater = quickSort (filter (>  x) xs)
```

Q 6. Expliquez le type de la fonction `quickSort`.

Si on désire trier une liste de chaînes de caractères par longueur croissante, l'expression suivante ne fonctionne pas :

```
ghci> quickSort ["toit","escalier","porte"]
["escalier","porte","toit"]
```

car la fonction Haskell trie par défaut les chaînes selon l'ordre alphabétique.

On choisit donc d'écrire une nouvelle fonction de tri `quickSortBy` qui prend en entrée une fonction de comparaison de type `a -> a -> Ordering`, ainsi que la liste de type `[a]` des éléments à trier. On rappelle que le type `Ordering` est un type algébrique contenant les constructeurs `LT, EQ, GT` (less than, equal, greater than).

Par exemple la fonction suivante :

```
compareStringByLength :: String -> String -> Ordering
compareStringByLength s1 s2 | l1 < l2   = LT
                            | l1 > l2   = GT
                            | otherwise = EQ
                where l1 = length s1
                      l2 = length s2
```

permet de trier par longueur croissante :

```
ghci> quickSortBy compareStringByLength ["toit","escalier","porte"]
["toit","porte","escalier"]
```

Q 7. Donnez le type de la fonction `quickSortBy`.

Q 8. Définissez `quickSortBy` en adaptant la fonction `quickSort` ci-dessus (en particulier, vous conserverez `filter`).

Dans certains cas, la fonction de comparaison est très coûteuse et on change donc d'approche : on décide d'associer un *poids* à chaque élément puis de les trier suivant ce poids. La fonction `quickSortOn` prendra ainsi en argument une fonction de poids de type `Ord b => a -> b`.

```
-- Trie la liste par taille croissante
ghci> quickSortOn length ["toit","escalier","porte"]
["toit","porte","escalier"]
-- Trie une liste de couples suivant la première valeur
ghci> quickSortOn fst [(14,"Jean"),(3,"Pierre"),(6,"Patrick")]
[(3,"Pierre"),(6,"Patrick"),(14,"Jean")]
-- Trie suivant la somme des deux composantes
ghci> quickSortOn (\(x,y) -> x + y) [(1.2,3.5),(-0.1,0.5),(1.0,4.5),(3.1,6.5)]
[(-0.1,0.5),(1.2,3.5),(1.0,4.5),(3.1,6.5)]
```

Pour que cette approche soit avantageuse, il faudra que la fonction `quickSortOn` calcule au plus une fois le poids de chaque élément du tableau à trier.

Q 9. Donnez le type de la fonction `quickSortOn`.

Q 10. Donnez une définition possible de la fonction `quickSortOn`. Vous pourrez par exemple transformer la liste de type `[a]` en une liste de type `[(a,b)]` et réutiliser les fonctions précédentes.