TP Haskell #3/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Pratique de la définition de fonctions.
2. Définitions de fonctions récursives.
3. Arbres et filtrage de motifs.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP3`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Accumulation sur les listes

Fichiers à rendre : `fold.hs`, `test_fold.ghci`.

- Écrire une fonction `somme_liste :: [Int] -> Int` qui calcule la somme des entiers d'une liste.

Ce motif consistant à *accumuler* les valeurs d'une liste est très récurrent ; on aimerait bien le factoriser. Commençons par généraliser l'opération.

- Écrire une fonction `acc_ints :: (Int -> Int -> Int) -> Int -> [Int] -> Int` qui prend en arguments une opération binaire, une valeur initiale, et une liste d'entiers. Elle devra accumuler les entiers de la liste, à partir de la valeur initiale, en appliquant l'opérateur binaire.
- En analysant votre code, déterminez si `acc_ints (-) 0 [x,y]` calcule `(0-x)-y` ou `x-(y-0)` (les deux sont possibles selon la façon dont vous avez écrit la fonction). Vérifiez avec un test.
- Écrire une fonction `max_liste :: [Int] -> Int` qui calcule le maximum d'une liste d'entiers positifs en utilisant une application partielle de `acc_ints`.

Finalement, le code de `acc_ints` ne semble plus être spécifique aux entiers, car l'addition est maintenant devenue une fonction générique.

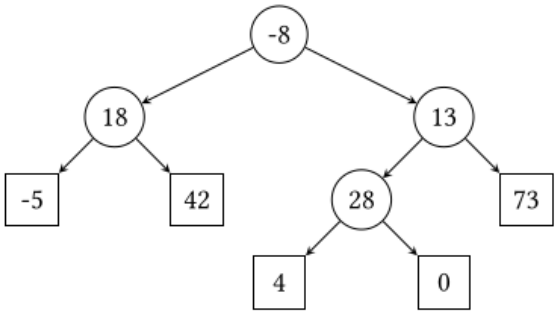
- Recopier la définition de `acc_ints` sous le nom `acc` (penser à renommer l'appel récursif), cette fois sans annoncer son type. Quel type GHCi infère-t-il pour la fonction ?
- En déduire la fonction `my_foldl :: (b -> a -> b) -> b -> [a] -> b` qui accumule une liste de valeurs de type `a` et produit un résultat accumulé de type `b`. L'ordre d'association est imposé : `my_foldl (-) 0 [x,y]` devra calculer `(0-x)-y`.
- Écrire de même la fonction `my_foldr :: (a -> b -> b) -> b -> [a] -> b` qui associe dans l'ordre inverse : `my_foldr (-) 0 [x,y]` calculera donc `x-(y-0)`.
- Utiliser une de ces deux fonctions pour écrire la fonction `join :: [String] -> String` qui fait le travail inverse de `mots` et reconcatène les éléments de la liste en les séparant par des espaces. *On s'autorisera un espace supplémentaire au début ou à la fin.*
- Dans le cas où la valeur initiale est neutre pour la fonction d'accumulation, donner une condition suffisante sur la fonction pour que `foldl` et `foldr` soient équivalentes.

À partir de maintenant, n'hésitez pas à utiliser les fonctions standards `foldl` et `foldr` quand elles vous semblent appropriées (ainsi que les fonctions `sum`, `minimum`, `elem`, ... qui sont construites avec).

Exercice 2 — Arbres binaires

Fichiers à rendre : `arbres.hs`, `test_arbres.ghci`.

Dans cet exercice, nous allons manipuler des *arbres binaires*, une structure de données classique en algorithmique. Un arbre binaire est une structure de ce style :



L'arbre est composé de *noeuds internes* (les ronds) et de *feuilles* (les carrés). Les noeuds internes ont chacun deux enfants qui sont eux-même des arbres (d'où le "binaire" — 2 enfants). Les feuilles n'ont pas d'enfant. Ici, on s'intéresse à un arbre dans lequel tous les noeuds portent une valeur (il y a de nombreuses variations).

On peut définir ce type ainsi en Haskell :

```haskell
data Bintree a =
  Leaf a |
  Node a (Bintree a) (Bintree a)
  deriving Show
```

- Expliquer la définition fournie, en commentaire du code. Que représente `a` ? À quel type de noeud de l'arbre correspond chaque constructeur ? Quels paramètres du constructeur `Node` correspondent aux deux enfants des noeuds internes ?
- Définir une constante `arbre_exemple :: Bintree Int` qui représente l'arbre du schéma ci-dessus.

La nature ”imbriquée” des arbres binaires fait que la structure se prête très bien à l'écriture de fonctions récursives.

- Écrire une fonction `nombre_feuilles :: Bintree a -> Int` qui compte le nombre de feuilles dans l'arbre.
- Écrire une fonction `hauteur :: Bintree a -> Int` calculant la *hauteur* de l'arbre, ie. la plus grande distance entre le sommet de l'arbre et une des feuilles. (Par exemple, la hauteur de `arbre_exemple` est 4.)

On se demande maintenant si on peut avoir un arbre d'entiers (un `Bintree Integer`) qui est ”trié” dans le sens où il vérifierait les deux propriétés suivantes :

1. Les valeurs associées aux noeuds sont uniques (ie. chaque valeur n'apparaît qu'une fois dans l'arbre).
2. Dans un noeud interne `Node v t1 t2`, toutes les valeurs présentes dans `t1` (le sous-arbre de gauche) sont inférieures à v, et toutes les valeurs présentes dans `t2` (le sous-arbre de droite) sont supérieures à v.

Un tel arbre s'appelle un *Arbre Binaire de Recherche (ABR)*.

- Construire un ABR `abr_exemple :: Bintree Integer` qui contient les mêmes valeurs que `arbre_exemple`.
- Écrire une fonction `rechercher_abr :: Bintree Integer -> Integer -> Bool` qui exploite la propriété d'ABR pour déterminer rapidement si un entier donné est présent dans l'arbre.
- Écrire une fonction `aplatir_abr :: Bintree a -> [a]` qui génère la liste triée des éléments de l'arbre.
- Comparer intuitivement la vitesse de `rechercher_abr` avec une recherche dans une liste et une recherche dichotomique dans un tableau. Y a-t-il différentes façons d'écrire `abr_exemple` qui impacteraient l'efficacité de la fonction `rechercher_abr` ?

Exercice 3 — Découpage en mots

Cet exercice est facultatif ; vous pourrez rendre les fichiers `mots.hs` et `test_mots.ghci`.

Le but de cet exercice est de lister les mots d'une chaîne de caractères comprenant des espaces, comme la méthode `split` de Python :

```
mots "  You shall   not pass!" -- ["You", "shall", "not", "pass!"]
```

Dans cet exercice, vous n'avez pas le droit aux fonctions standard sur les listes. ;D

- Écrire une fonction `espace :: Char -> Bool` qui détermine si le caractère passé en paramètre est un espace.
- Écrire une fonction `separe :: (Char -> Bool) -> String -> (String, String)` qui prend une chaîne et un prédicat sur les caractères, et sépare la chaîne en deux morceaux : le plus long préfixe de la chaîne dont tous les caractères vérifient le prédicat, et le reste.

  Par exemple `separe (== 'a') "aaabaaba"` renverra `("aaa", "baaba")`.

  Privilégiez l'utilisation d'un `let` ou un `where` plutôt que `fst` et `snd` pour découper la paire obtenue lors de l'appel récursif.

- Utiliser `separe` et `espace` pour écrire la fonction `grignote_espaces :: String -> String` qui grignote les espaces en tête d'une chaîne.

On a maintenant tout ce qu'il faut pour extraire rapidement un mot, puis tous les mots.

- Écrire une fonction `un_mot :: String -> (String, String)` qui prend une chaîne ne commençant pas par un espace, et renvoie un couple contenant son premier mot (ie. la série de non-espaces au début) ainsi que le reste (dont les espaces initiaux seront retirés avec `grignote_espace`). Utilisez le plus possible les fonctions précédentes.
- Écrire une fonction `mots :: String -> [String]` qui sépare la chaîne donnée en mots. Il vous est suggéré de définir et appeler une fonction auxiliaire qui travaille sur des chaînes ne commençant pas par des espaces.

Et si ça vous démange depuis le début :

- Bonus : écrire une version plus simple de `mots` en utilisant les fonctions standard `dropWhile` et `span` (possible en 3 lignes).

Exercice 4 — Système de fichiers

Cet exercice est facultatif ; vous pourrez rendre les fichiers `fichiers.hs` et `test_fichiers.ghci`.

On revisite maintenant les arbres pour modéliser un système de fichiers. Dans un système de fichiers, les noeuds internes sont les dossiers tandis que les feuilles sont les fichiers. Cette fois, les arbres ne sont plus nécessairement *binaires* : un dossier peut contenir plus que 2 éléments (et même aucun).

- En s'inspirant de la définition de `Bintree a`, définissez le type `Tree a` qui a les mêmes constructeurs mais dans lequel `Node` prend en paramètres une valeur de type `a` et une liste de sous-arbres.

En observant le dossier de travail du TP, on remarque la hiérarchie suivante :

```
CS222/
├── TP1/
│   ├── CS222_TP1.pdf
│   ├── applis.hs
│   ├── rec.hs
│   └── test_rec.ghci
├── TP2/
│   ├── CS222_TP2.pdf
│   ├── chaines.hs
│   └── tarot.hs
└── TP3/
```

- Construire un arbre `arbre_TP :: Tree String` représentant les contenus du dossier de TP. Sur chaque noeud vous indiquerez le nom du dossier (sans `/` final) ou du fichier.
- Écrire une fonction `descendre :: Tree String -> String -> Maybe (Tree String)` prenant en paramètre un arbre (supposé être un dossier), le nom d'une entrée, et renvoie le sous-arbre correspondant à cette entrée du dossier. Si l'arbre n'est pas un dossier ou l'entrée demandée n'existe pas, la fonction devra renvoyer `Nothing`.
- Utiliser `descendre` dans un appel bien choisi à une fonction `fold` pour obtenir une fonction `descendre_chemin :: Tree String -> [String] -> Maybe (Tree String)` qui descende le long d'une série de dossiers/fichiers. Par exemple, `descendre_chemin arbre_TP ["TP2", "tarot.hs"]` renverra le noeud du fichier `tarot.hs`.

- Écrire une fonction `liste_fichiers :: Tree String -> [String]` qui génère la liste des chemins de tous les fichiers de l'arbre, c'est-à-dire `["CS222/TP1/CS222_TP1.pdf", "CS222/TP1/applis.hs", ...]`.