TP Haskell #4/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

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

Objectifs :

1. Pratique de définition et utilisation des classes de types.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP4_VotreNom`. 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 — Agrégation générique avec la classe `Foldable`

Fichiers à rendre : `Foldable.hs`, `test_foldable.ghci`.

Dans le TP précédent, nous avons utilisé les fonctions `foldl` et `foldr` pour calculer des agrégations sur des listes:

```haskell
my_foldl :: (b -> a -> b) -> b -> [a] -> b
my_foldr :: (a -> b -> b) -> b -> [a] -> b
```

L'intuition est que beaucoup de structures qui ne sont pas des listes peuvent être *parcourues* comme des listes, et ainsi se prêter à une agrégation. Par exemple, dans l'exercice sur les arbres binaires de recherche, on a parcouru un ABR d'une façon bien choisie pour obtenir une liste triée de ses éléments. On peut écrire une fonction d'agrégation sur l'arbre qui se comporte comme `foldr` sur la liste triée.

- Redéfinir le type `Bintree a` des arbres binaires portant des valeurs de type `a`, vu au TP3 (sans relire le sujet, si vous y arrivez !).
- Écrire la fonction `parcours_infixe :: Bintree a -> [a]` qui calcule la liste des valeurs de l'arbre dans l'ordre infixe (c'est-à-dire que pour un noeud interne on met d'abord les valeurs du sous-arbre de gauche, ensuite la valeur portée par le noeud lui-même, ensuite les valeurs du sous-arbre de droite). Cette fonction devra générer la même séquence que la fonction `aplatir_abr` du TP3. Écrire les tests associés.
- Écrire maintenant une fonction `foldr_arbre :: (a -> b -> b) -> b -> Bintree a -> b`, qui accumule de la même façon que `foldr` les éléments de l'arbre pris dans l'ordre infixe. Pour

cette fonction, vous n'utiliserez pas `parcours_infixe` (écrivez une nouvelle fonction récursive).

- Dans vos tests dans `test_foldable.ghci`, vous prendrez soin de comparer `foldr_arbre` avec `foldr` appliqué au résultat de `pacours_infixe` (qui doivent être identiques).

Il est courant, à la fois dans la bibliothèque standard Haskell et dans les applications, d'écrire des fonctions qui parcourent et accumulent des valeurs en utilisant `foldr` ou un équivalent. Haskell nous fournit la classe de types `Foldable` pour capturer de façon générique ce concept ; ainsi, n'importe quelle fonction qui appelle `foldr` marche sur les arbres sans avoir besoin de mentionner `foldr_arbre`.

- Inspecter le type de `foldr` dans GHCi avec la commande `:type`. Remarquez l'argument de type `t a`. Quelles sont les deux "valeurs" de `t` (constructeurs de types) qu'on a étudiées jusqu'ici ?
- Écrire une fonction `to_list :: Foldable t => t a -> [a]` qui transforme une structure de données en liste en utilisant `foldr` appliqué à un opérateur binaire bien choisi.
- Définir une nouvelle instance de `Foldable` pour le constructeur de types `Bintree` en utilisant `foldr_arbre` comme définition de la fonction `foldr`. Cela suffit pour obtenir accès à *l'intégralité* des fonctions de la classe `Foldable` (dont vous pouvez obtenir la liste avec la commande `:info Foldable`), ainsi que la fonction `to_list` définie précédemment.

On rappelle ici la syntaxe pour les instances de type:

```
instance <classedeType> <untype> where
    <fundef1> = ...
```

- Sans définir de nouvelle fonction, calculer la somme des éléments d'un arbre d'entiers à l'aide de la fonction standard `sum`. Expliquer en commentaire le type de `sum`.

Interlude — Un programme d'ordonnancement de tâches

À titre d'application de Haskell, on va maintenant construire un programme de traitement de tâches dont le rôle sera de lire et exécuter des tâches de calcul. On imagine que ce programme sera lancé sur un serveur ; il aura une liste de tâches en attente, qu'il exécutera selon un ordre de priorité. Dans le cas où le programme a encore des tâches en attente lorsqu'on l'arrête, il les sauvegardera dans un fichier pour reprendre leur exécution plus tard.

Ce programme sera construit progressivement : on va explorer divers aspects comme le stockage dans des fichiers ou la gestion des priorités dans des exercices individuels sur chaque TP, et on réunira le tout pour construire l'application finale au dernier TP.

Exercice 2 — Sérialisation JSON

Fichiers à rendre : `Json.hs`, `test_json.ghci`.
Cet exercice fait partie du programme d'ordonnancement de tâches.

JSON est un format de données extrêmement répandu (né comme une description d'objets en Javascript mais généralisé depuis), utilisé pour représenter sous forme de texte des données structurées. Une valeur JSON peut être :

- Une constante : entier (`42`), flottant (`7.3`), chaîne (`"JSON!"`), booléen (`true`), ou `null`.
- Un tableau de valeurs, noté entre crochets : `[42, true, "JSON!", [false, 0]]`.
- Un "objet" (en réalité un dictionnaire), noté entre accolades, avec des chaînes de caractères comme clés : `{"entier": 42, "texte": "Haskell!", "tableau": [true]}`.

JSON est fréquemment utilisé pour *sérialiser* des données, c'est-à-dire les représenter sous forme d'une simple séquence (ici de caractères) sans pointeurs. La sérialisation est très utile pour stocker des données (par exemple une sauvegarde dans un jeu) ou pour les transmettre (par exemple répondre à une requête web sur le réseau). Le format est de plus indépendant des langages de programmation et des architectures. L'opération consistant à reconstruire la donnée originale s'appelle *désérialiser*.

Dans le programme d'ordonnancement de tâches, la sérialisation nous sera utile pour sauvegarder dans un fichier la liste des tâches en attente à la fin de l'exécution.

On se donne pour représenter des données JSON dans un format simplifié avec des constantes de type `Integer`, `Bool` et `String` :

```
data JSON =
  JSON_Int Integer |
  JSON_Bool Bool |
  JSON_String String |
  JSON_Array [JSON] |
  JSON_Object [(String, JSON)]
```

- Définir les constantes `exemple_tableau :: JSON` et `exemple_objet :: JSON` encodant le tableau et l'objet donnés en exemples ci-dessus. Les valeurs construites avec `JSON_Array` contiennent d'autres valeurs de type `JSON`. Quelle structure de données vue précédemment avait aussi cette caractéristique ?

À ce stade, on n'a pas de quoi afficher les valeurs de type `JSON` dans le terminal. Pour cela, il nous faudrait une instance de la classe de types `Show`, dont la fonction principale est `show :: a -> String`.

- Définir une fonction `show_json :: JSON -> String` qui donne la notation d'une valeur JSON comme dans les exemples introductifs. On utilisera `show` pour obtenir la représentation textuelle des entiers et des chaînes de caractères, et la fonction intercalate peut aider.
- Instancier `JSON` dans la classe en `Show` en fournissant une unique fonction `show` égale à `show_json`. Constater qu'on peut maintenant évaluer `exemple_tableau` ou `exemple_objet` directement dans GHCi et obtenir en réponse leur notation JSON. Ajouter des tests de ce comportement dans `test_json.ghci`.

On se donne maintenant le type suivant correspondant à une tâche de calcul : soit afficher un nombre, soit afficher la somme de deux nombres.

```
data Task =
  PrintVal Integer |
  PrintSum Integer Integer
```

- Écrire une fonction `serialize_task :: Task -> JSON` qui encode une tâche en JSON :
  - `PrintVal <x>` sera encodé par `{"op": "PrintVal", "x": <x>}` ;
  - `PrintSum <x> <y>` sera encodé par `{"op": "PrintSum", "x": <x>, "y": <y>}`.

  Ajouter quelques tests dans `test_json.ghci` si ce n'est pas encore fait.
- Écrire à l'inverse une fonction `deserialize_task :: JSON -> Maybe Task` qui essaie de décoder une valeur JSON correspondant à une tâche. Vous renverrez `Nothing` si la valeur JSON ne correspond à aucun des deux encodages valides. (Remarquez que grâce au filtrage de motifs il suffit presque d'inverser les arguments et valeurs de retour de `serialize_task`.)

Comme la séralisation est une tâche très courante, on aimerait bien ne pas avoir une fonction `serialize_X` pour tous les types intéressants (on aura notamment besoin de sauvegarder une *liste*

de tâches dans le programme). On définit donc une classe de types pour avoir les mêmes fonctions `serialize` et `deserialize` pour tous les types:

```
class Serializable a where
  serialize :: a -> JSON
  deserialize :: JSON -> Maybe a
```

- Construire une instance `Serializable Task` en utilisant les fonctions définies précédemment.
- Construire une instance `Serializable a => Serializable [a]` que l'on utilisera pour sérialiser des listes de valeurs sous la forme d'un tableau JSON. La contrainte `Serializable a` vous permet d'utiliser `serialize` et `deserialize` sur les éléments de la liste. Dans la fonction de désérialistion, vous renverrez un résultat (`Just`) uniquement si tous les éléments ont pu être désérialisés ; si l'un d'eux échoue, renvoyez `Nothing`.

Exercice 3 — Tri fusion

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

L'objectif de cet exercice est de coder en Haskell un tri fusion sur des listes d'entiers, sous la forme d'une fonction `tri_fusion :: [Int] -> [Int]`.

- Écrire la fonction `halve :: [Int] -> ([Int], [Int])` qui coupe une liste en deux listes de tailles égales (à une unité près). Ne pas utiliser la longueur de la liste. Il n'est pas nécessaire de garder l'ordre des éléments de l'entrée !
- Écrire la fonction `combine :: [Int] -> [Int] -> [Int]` qui prend deux listes triées par ordre croissant en entrée, et combine ces deux listes en une unique liste triée par ordre croissant.
- En utilisant les fonctions `halve` et `combine`, écrire la fonction `tri_fusion` Il faudra traiter séparément les cas des listes à 0 et 1 élément.