TP3. Types - Classes

Remarques générales :

Déroulement des TP

- Avant de démarrer les TP, un répertoire CS212 doit être créé sous le répertoire home (Linux)

- A chaque séance de TP, un répertoire correspondant (TP1, TP2 ou TP3) sera créé sous le répertoire CS212.

- La séance de TP se déroule par défaut sous le répertoire correspondant (tout changement de répertoire courant sera indiqué explicitement).

Evaluation des TP

Un CR de TP sera rendu 5 jours au plus tard après la dernière séance. Ce CR doit se présenter sous la forme **d'un seul fichier pdf** et comporter les éléments suivants :

- Une description de vos observations (pour les exercices d'observation),

- une copie des programmes sources et les captures d'écran montrant les résultats d'exécution pour les exercices demandant l'écriture de programmes Haskell.

- Ce fichier pdf sera téléchargé sur Chamilo.

Objectifs du TP3 :

1. Définitions de types
2. Définitions de classes

Exercice 1 (types et classes)

Cet exercice est un prolongement de l'exercice 3 du TP2. Vous pourrez vous inspirer de vos réponses aux questions de l'exercice 3 du TP2 pour répondre aux questions suivantes, dans un nouveau fichier source.

A. Versions plus génériques de `somme` et `cumule`

1) Ecrivez une version générique de `cumule`; son type sera donc :

```
(a -> a -> a) -> a -> [a] -> a
```

- Quels éléments de la définition de la fonction avez-vous eu besoin de retoucher ?

2) Ecrivez une nouvelle version de `somme2`, qui ne s'applique qu'aux *listes d'entiers*. Quel sera son type?

3) Ecrivez une nouvelle version de `somme` qui fonctionne pour toute liste d'éléments d'un type faisant partie de la classe `Num`.

B. Intégration des arguments de `cumule` à une classe de types

On va maintenant définir un module `Agregeable`, et donc pour cela créer un nouveau fichier.

4) Définissez une classe "`Agregeable a`" qui contient deux membres : un élément de type `a`, appelé `'neutre'`; et un élément de type `a -> a -> a`, appelé `'operation'`.

5) Ecrivez la déclaration d'instance de `Int` pour la classe `Agregeable`, en précisant ses éléments. Ces éléments seront utilisés ci-dessous pour définir un opérateur somme.

6) Ecrivez une nouvelle version de `cumule`, applicable aux éléments de la classe `Agregeable`. Son type sera donc: `cumule :: Agregeable a => [a] -> a`

7) Donnez un exemple d'appel de `cumule` sur une *liste d'entiers*, directement dans l'interprète. Indication: il faudra préciser le type des éléments de cette liste. Précisez la manière dont vous procédez pour ce faire.

8) Ecrivez une déclaration d'instance pour `Bool`, de sorte que la fonction `cumule` calcule le "`ou`" de tous les booléens de la liste.

9) Ecrivez une déclaration d'instance pour [a] dans la classe `Agregeable`.

C. `cumule` appliquée à la fonction `max`

Pour cette partie, vous pouvez compléter le code source de la partie B.

10) Créer un nouveau module `IE`, qui importe les définitions du module `Agregeable`.

11) Définir un type `Int_ex` ("Int extended", pour entiers étendus) qui représente à la fois les valeurs `Int` (vous pourrez nommer le constructeur associé "`I`"), et la valeur spéciale "`moins l'infini`", que vous pourrez nommer `Moins_inf`. Demandez à Haskell d'en faire une instance de la classe `Show`.

12) Définir une version de la fonction `max` pour ce type; vous pourrez la nommer :

```
max_ie :: Int_ex -> Int_ex -> Int_ex.
```

13) Ecrire la déclaration qui fait de `Int_ex` un membre de la classe `Agregeable`, de sorte que `cumule` calcule le maximum d'une liste. Quel sera l'élément neutre?

14) Que fait la fonction "`map I`"?

15) Ecrire une fonction "`max_liste`", de type "`[Int] -> Int_ex`", qui calcule le maximum d'une liste d'entiers, en faisant faire le travail à `cumule` (vous pourrez aussi utiliser votre réponse à la question précédente !). Que renvoie-t-elle sur une liste vide?

Exercice 2 : Quick sort

En utilisant la notation de définition d'ensemble par compréhension, écrivez une définition concise de la fonction `quick_sort`, qui prend une liste d'éléments de type `a`, et renvoie une liste triée comprenant les mêmes éléments. A quelle classe de type devra appartenir le type `a`?

Exercice 3 : Les tours de Hanoï

L'objectif de cet exercice est de résoudre le problème classique des tours de Hanoï. En résumé, il s'agit de trouver une suite de mouvements élémentaires d'objets entre 3 piles d'objets de taille décroissantes, de manière à faire migrer 1 pile complète de tous les objets vers un autre emplacement, sans jamais poser un objet plus grand sur un objet plus petit. Vous trouverez facilement des compléments d'information au sujet de ce problème sur Internet.

Nous allons commencer par nous approprier un module complet qui vous est fourni; puis nous détaillerons l'encodage de données proposés dans un module incomplet, avant de vous proposer un chemin vers une solution au problème.

1) Le module `Set` (fourni) définit le constructeur de type `Set a`, qui représente un ensemble d'éléments de type `a`, et quelques primitives pour travailler avec les ensembles :

- `set`, qui initialise un ensemble à partir d'une liste ;
- `set_difference`, qui calcule la différence de deux ensembles ;
- et `grab` qui prend un élément dans un ensemble non vide.

Démontrez l'usage de ces 3 primitives à l'aide de quelques exemples évalués dans l'interprète.

2) Prenez connaissance des types de données proposées dans le module `hanoi.ps`.

Que représente exactement un élément du type `Pos` ? Que représente exactement un élément du type `Movement`?

3) Ecrire une fonction `other`, de type `Pos -> Pos -> Pos`, qui retourne l'unique valeur qui n'a pas été fournie en paramètre. On suppose que l'appelant fournit toujours deux valeurs distinctes; il faudra vous en assurer dans la suite quand vous appellerez la fonction `other`.

4) La fonction qui répond au problème, `hanoi :: Int -> [Movement]` vous est intégralement fournie; mais elle fait appel à `hanoi_aux`, qu'il vous faudra écrire.

Le travail réalisé par l'appel (`hanoi_aux k p1 p2`) est la description de la suite de mouvements à effectuer pour déplacer les `k` plus petits objets de la position `p1` à la position `p2`. Sachant cela, et en analysant le type et le code de `hanoi`, expliquer précisément ce que fait l'appel (`hanoi n`).

5) Pour écrire le code de `hanoi_aux`, il vous suffit de traduire en Haskell l'idée suivante de résolution du problème des tours de Hanoï :

"pour déplacer une pile de `k` objets de la position `p1` à la position `p2`, il suffit de déplacer `k-1` objets de la position `p1` à la position `p3`, puis de déplacer l'objet de taille `k` de la position `p1` à la position `p2`, puis finalement de déplacer `k-1` objets de la position `p3` à la position `p2`, où `p3` désigne la position qui n'est ni `p1`, ni `p2`".

Pensez bien à encoder un cas initial. Finalisez le code, testez-le, et prouvez qu'il fonctionne correctement!