TP Haskell #5/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

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

Objectifs :

1. Mise en oeuvre d'algorithmes plus subtils sur les arbres.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP5_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 — Tri par tas

Fichiers à rendre: Tas.hs, test_tas.ghci.
Cet exercice fait partie du programme d'ordonnancement de tâches.

Vous connaissez plusieurs algorithmes de tri, et vous savez qu'ils n'ont pas tous la même complexité. Vous connaissez des algorithmes ayant la complexité optimale, O(n . log(n)), où *n* est le nombre de données à trier.

Il existe un algorithme de tri optimal qui s'appuie fortement sur une structure de données. Cet algorithme s'appelle le *tri par tas*; et la structure de données, c'est le *tas*. Dans cet exercice, nous allons recréer en Haskell une structure de données encodant les tas; nous nous en servirons ensuite pour implémenter l'algorithme de tri par tas.

Un tas est une structure arboresecente avec des valeurs dans les noeuds internes, et qui respecte une certaine propriété de tri partiel entre ses éléments. Cette propriété est que la valeur stockée dans un noeud doit toujours être plus petite ou égale aux valeurs stockées dans les noeuds fils.

Afin de garantir une complexité logarithmique pour les opérations élémentaires sur les tas (ajout, retrait du minimum), nous devrons veiller à ce que les tas ne soient pas trop déséquilibrés.

Un tas sera dit équilibré lorsque pour tout noeud interne, le nombre de valeurs du fils gauche et le nombre de valeurs du fils droit diffèrent d'au plus une unité.

Voici un exemple de tas. Vous noterez que ce tas est équilibré, et qu'il encode 8 valeurs entières.

Les tas seront représentés par le type `Tas a` défini ci-dessous:

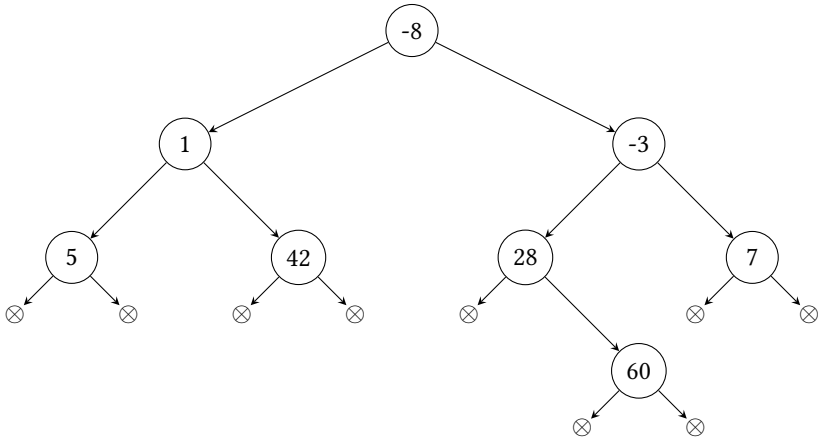

Figure 1: Exemple de tas.

```
data Tas a = Noeud Int a (Tas a) (Tas a) | Vide   deriving (Show)
```

Un élément de type Tas a est soit un tas vide, soit construit à partir d'un élément de type a, et de deux tas. Le constructeur Noeud embarque également un entier, qui encode le nombre total d'éléments du tas. Dans la suite, on appelle *taille de tas* le nombre d'éléments encodés dans le tas.

Dans la suite, il n'est pas (toujours) précisé de réaliser des tests, c'est à vous d'y penser au fur et à mesure. De même, dessiner des arbres sur papier est quelquefois (souvent, toujours ?) une bonne idée

1. Définir en Haskell un tas t1 correpondant au tas représenté sur la figure.
2. Définir une fonction `taille :: Tas a -> Int` qui retourne la taille du tas passé en paramètre.
3. Définir une fonction `tas_min :: Tas a -> a` qui retourne la valeur minimum d'un tas. Notez qu'en raison de l'encodage des tas, cette valeur est directement accessible pour un tas construit avec le constructeur Noeud. Cette fonction n'ayant pas de sens pour un tas vide, elle ne sera pas définie dans ce cas.

Avant de définir des fonctions de manipulation des tas, nous allons adopter une discipline qui nous permettra de manipuler les tas sans avoir à se soucier de l'entier qui encode la taille. C'est aussi une façon d'écrire du code plus sûr: en minimisant les manipulations sur cet entier, on minimise les risques d'erreur.

4. Ecrire une fonction `noeud :: a -> Tas a -> Tas a -> Tas a`, qui crée un tas à l'aide du constructeur Noeud, en calculant sa taille à partir des tailles des tas passés en paramètre.

Voici maintenant la discipline que nous vous proposons d'adopter:

- tous les tas seront créés exclusivement en faisant appel à la fonction `noeud` ou au constructeur `Vide`; le constructeur `Noeud` n'est utilisé pour construire des valeurs de `Tas a` que dans le code de `noeud`; ailleurs, il ne sera utilisé que pour effectuer le filtrage de motifs;
- lors du filtrage de motif sur les paramètres des fonctions définies sur les tas, le premier paramètre du constructeur `Noeud` sera toujours `_` (qui indique une variable anonyme).

Si vous suivez cette discipline, vous n'aurez pas à calculer la taille des noeuds construits; et vous ne risquez pas de faire d'erreur sur ces calculs de taille.

5. Ecrire une fonction `est_equilibre :: Tas a -> Bool`, qui décide si le tas passé en paramètre est équilibré. Notez bien que la propriété d'équilibre concerne tous les sous-tas du tas; une unique

vérification des fils de la racine ne suffit pas à déterminer si un tas est équilibré.

Dans la suite, vous prendrez soin de vérifier dans vos tests que les tas retournés par vos fonctions sont équilibrés.

6. Ecrire une fonction `ajouter`, qui prend un élément de type a et un tas équilibré, et qui ajoute cet élément au tas, de manière à produire un tas équilibré. Quel type doit avoir une telle fonction? Pensez bien à écrire des tests qui illustrent tous les chemins d'exécution possibles de votre fonction.

7. Ecrire une fonction `retirer_feuille :: Tas a -> (a, Tas a)`, qui prend en paramètre un tas non vide équilibré, et trouve une feuille dont le retrait produit un tas équilibré. La fonction doit retourner la feuille, et le tas obtenu après retrait de cette feuille.

Notez que cette fonction n'a de sens que pour un tas non vide. Vous l' implémenterez donc comme une fonction partielle: l'appel de cette fonction sur un tas vide doit provoquer une erreur à l'exécution.

8. Ecrire une fonction `equilibrer :: Ord a => Tas a -> Tas a`, qui prend en paramètre un tas non vide dont l'écart de taille entre les fils est au plus de deux, et qui retourne un tas équilibré, contenant les mêmes valeurs que le tas passé en paramètre. Commencez par réfléchir à comment rééquilibrer un tas dont l'écart de taille entre les fils est exactement deux. Vous pourrez vous servir de la fonction `retirer_feuille`.

9. Ecrire une fonction `retirer :: Ord a => Tas a -> Tas a`, qui prend en paramètre un tas non vide équilibré, et retourne le tas obtenu en retirant de ce tas son élément minimum. Le tas retourné doit être équilibré.

10. Ecrire une fonction `construit :: Ord a => [a] -> Tas a`, qui prend une liste d'éléments comparables entre eux (classe Ord), et produit un tas contenant les mêmes éléments.

11. Ecrire une fonction `deconstruit :: Ord a => Tas a -> [a]`, qui prend un tas, et retourne une liste triée de ses éléments.

12. Ecrire une fonction `tri :: Ord a => [a] -> [a]`, qui implémente le tri par tas pour des éléments d'un type appartenant à la classe Ord. Testez cette routine sur plusieurs exemples de taille au moins 10.

Quelques remarques finales: les bonnes complexité des opérations d'insertion et de suppression du minimum permettent d'autres usages au-delà du tri par tas. Par exemple, en entremêlant des insertions d'éléments avec un niveau de priorité (la valeur la plus basse étant la plus prioritaire), et des retraits, cette structure de données permet de maintenir ce qu'on appelle usuellement une *file de priorité*; nous utiliserons les tas pour cet usage dans le programme d'ordonnancement de tâches au dernier TP.

Exercice 2 — La structure "zipper"

Cet exercice est facultatif. Vous pourrez rendre les fichiers Zipper.hs et test_zipper.ghci.

Un `zipper` - appélation anglophone que l'on préférera à ses adaptations françaises, `fermeture éclair` ou encore `braguette` - est une structure de donnée fonctionnelle qui rend dans le monde fonctionnel des services comparables aux listes doublement chaînées dans les langages impératifs.

Un zipper est un container, dont les éléments sont rangés dans un certain ordre. Cela ressemble donc beaucoup à une liste. En plus des éléments enregistrés dans la structure, le zipper encode un emplacement dans la liste, et permet de manipuler les éléments voisins de cet emplacement.

Dans l'exemple ci-dessus, le zipper visite la liste [4, 2, 7, 1, 5]; il est positionné entre le 7 et le 1.

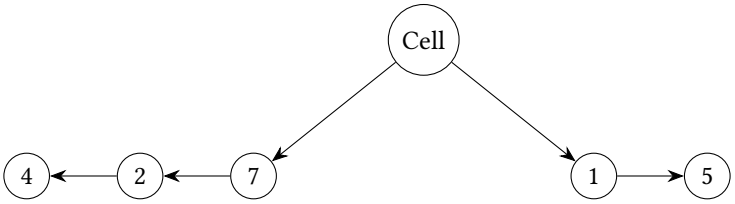

Figure 2: Exemple de zipper.

Plus techniquement, un zipper peut avoir pour interface les fonctions suivantes:

```
data Zipper a = ...

liste_en_zipper :: [a] -> Zipper a
zipper_en_liste :: Zipper a -> [a]
zipper_est_vide :: Zipper a -> Bool
est_au_debut :: Zipper a -> Bool
est_a_la_fin :: Zipper a -> Bool
recule :: Zipper a -> Zipper a
avance :: Zipper a -> Zipper a
rembobine :: Zipper a -> Zipper a
insere_avant :: a -> Zipper a -> Zipper a
insere_apres :: a -> Zipper a -> Zipper a
element_precedent :: Zipper a -> a
element_suivant :: Zipper a -> a
enleve_precedent :: Zipper a -> Zipper a
enleve_suivant :: Zipper a -> Zipper a
```

Certaines de ces fonctions ne peuvent pas être appelées sur certains types d'entrées. Par exemple, `avance` ne peut pas fonctionner sur un zipper qui répond vrai à `est_a_la_fin`. N'implémentez que ce qui a du sens; et laissez à l'appelant de votre fonction la responsabilité de vérifier que la donnée passée en paramètre a la bonne forme pour cet appel.

Voici un début d'implémentation:

```
module Zipper where

data Zipper a = Cell [a] [a]   deriving (Show)

liste_en_zipper :: [a] -> Zipper a
liste_en_zipper l = Cell [] l

est_au_debut :: Zipper a -> Bool
est_au_debut (Cell [] _) = True
est_au_debut (Cell (x:xs) _) = False

avance :: Zipper a -> Zipper a
avance (Cell l (x:xs)) = Cell (x:l) xs
```

- Complétez ce code en une implémentation complète d'un zipper.
- Illustrez le bon fonctionnement de votre code en enchaînant des appels aux différentes fonctions.
- D'après vous, quel genre de test peut-on effectuer pour garantir que votre implémentation des zippers est correcte?