Date : 25/06/2014
Durée : 2 heures
Documents authorisés
Ordinateurs interdits

Cours Programmation Fonctionnelle
Master 1 Informatique, Aix-Marseille Université

Examen

Typage, évaluation

Exercice 1. (6 pts). Considérez les trois définitions suivantes :

```
nombre_c [] = 0
nombre_c ('c':xs) = 1 + nombre_c xs
nombre_c ([x]:xs) = nombre_c xs
monMap f [] = []
monMap f (x:xs) = f x (monMap f xs)
trier [] p = []
trier [x] p = [x]
trier (x:xs) p = if p x y then x:(y:ys)
       else
           y:(trier (x:ys))
    where
        y:ys = trier xs
```

Pour chaque fonction définie, si l'analyse de son type produit une erreur :

1. expliquez pourquoi on a cette erreur ;
2. corrigez la définition par conséquent ;
3. donnez le type de l'expression définie ;

sinon, donnez simplement le type de l'expression définie.

Exercice 2. (4 pts). Considérez l'expression Haskell suivante :

```
(\x -> \f -> f x) 1 ((\y -> \z -> y + z) 1)
```

Question 2.1. Utilisez la stratégie par valeur (« *call-by-value* ») pour évaluer cette expression. (Fin question 2.1)

Question 2.2. Utilisez la stratégie par nom (« *call-by-name* ») pour évaluer cette expression. (Fin question 2.2)

Types récursifs

Exercice 3. (6 pts). Un *arbre binaire* est ou bien une feuille, ou bien un noeud ayant exactement deux enfants (l'un à gauche, l'autre à droite) ; chaque enfant est lui même un arbre binaire.

Question 3.1. Définissez en `Haskell` un type de données, nommé `ArbreBin`, pour coder les arbres binaires. (Fin question 3.1)

Question 3.2. Utilisez la réponse à la question précédente pour écrire une expression qui code l'arbre binaire sur la gauche de la figure 1. (Fin question 3.2)

Un *arbre à branchement fini* est un noeud ayant une liste ordonnée (de gauche à droite) d'enfants ; chaque enfant est lui même un arbre à branchement fini, il est possible que la liste d'enfants soit vide.

Question 3.3. Définissez en `Haskell` un type de données, nommée `ArbreBrf`, pour coder les arbres à branchement fini. (Fin question 3.3)

Question 3.4. Utilisez la réponse à la question précédente pour écrire une expression qui code l'arbre à branchement fini sur la droite de la figure 1. (Fin question 3.4)

On peut transformer un arbre binaire en un arbre à branchement fini, récursivement, de la façon suivante. Une feuille est transformée dans un noeud ayant une liste vide d'enfants ; un noeud avec deux enfants g et d (pour gauche et droite) est traité comme suit : on applique la procédure à g et d pour obtenir deux arbres à branchement fini l et r ; on retourne l'arbre qui est un noeud et dont la liste des enfants est obtenue de la liste des enfants de r en lui ajoutant en tête l'arbre l.
Par exemple, l'arbre à branchement fini en figure 1 sur la droite est obtenu de l'arbre binaire sur la gauche.

Question 3.5. Implémentez cette transformation en `Haskell` en écrivant la définition d'une fonction

```
arbreBintoArbreBrf ::
      ArbreBin -> ArbreBrf
```

(Fin question 3.5)

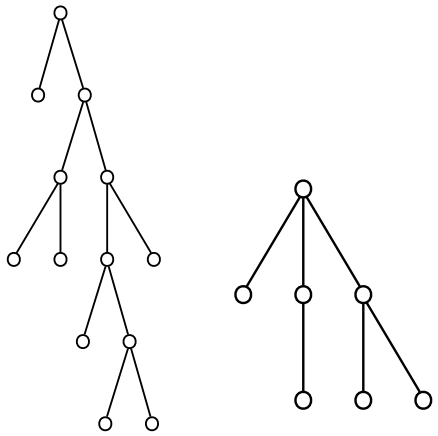

FIGURE 1 – Arbre binaire (gauche) et arbre à branchement fini (droite)

Sous-ensembles finis

Exercice 4. Le type `EnsFin` défini ci-dessous sert à manipuler des sous-ensembles finis (donc, pas d'éléments répétés) dont les élements sont tirés d'un type génerique `a` :

```
data EnsFin a = EnsFin [a] deriving Show
```

Question 4.1. (1 pts). Peut on optimiser la définition du type `EnsFin` ? Comment ? Justifiez votre réponse.

(Fin question 4.1)

Question 4.2. (1 pts). Expliquez, avec précision, ce qui se passe avec le mot clés `deriving` dans la définition du type `EnsFin`. (Fin question 4.2)

Considérez maintenant les définitions suivantes :

```
ensFinVersListe (EnsFin xs) = xs
vide = EnsFin []
appartient x (EnsFin []) = False
appartient x (EnsFin (y:ys)) =
    if x == y then True else appartient x (EnsFin ys)
ajouter_el x (EnsFin []) = EnsFin [x]
ajouter_el x (EnsFin (y:ys)) =
    if x == y then (EnsFin (y:ys)) else EnsFin (y:zs)
        where zs = ensFinVersListe (ajouter_el x (EnsFin ys))
test = ajouter_el 3 (EnsFin [1,3,4])
```

Question 4.3. (2 pts). Donnez un type à chacune des expressions définies ci-dessus.
Attention : n'oubliez pas les contraintes de classe quand elles sont nécessaires ; précisez aussi lesquelles parmi ces expressions sont polymorphes. (Fin question 4.3)

Question 4.4. (3 pts). Écrivez les définitions des fonctions

```
listeVersEnsFin :: Eq a => [a] -> EnsFin a
union :: Eq a => EnsFin a -> EnsFin a -> EnsFin a
intersection :: Eq a => EnsFin a -> EnsFin a -> EnsFin a
```

`listeVersEnsFin` transforme une liste vers un ensemble finis (attention : pas d'éléments répétés dans un ensemble), `union` et `intersection` sont les opérations ensemblistes usuelles. (Fin question 4.4)

Points totaux : 23