Date : 09/01/2015
Durée : 2 heures
Documents autorisés
Ordinateurs interdits

Cours Programmation Fonctionnelle
Master 1 Informatique, Aix-Marseille Université

Examen

Remarque : cet examen présente une choix d'exercices et questions ; pour obtenir la note maximum, ce n'est pas nécessaire de tous les résoudre de façon correcte.

Exercice 1 : typage. Considérez le script `Haskell` suivant :

```
class MyNum a where
    zero :: a
newtype MyInt = MyInt Int deriving Show
instance MyNum MyInt where
    zero = MyInt 0
newtype MyReal = MyReal Float deriving Show
instance MyNum MyReal where
    zero = MyReal 0
if1 = if True then show (zero::MyInt) else show (zero::MyReal)
if2 = if True then (zero::MyInt) else (zero::MyReal)
sum x y = MyReal (x + y)
mult (MyReal x) (MyReal y) = MyReal (x*y)
power x (MyInt 0) = MyReal 1
power x n = mult (power x (n-1)) x
```

Pour chaque expression parmi `if1, if2, sum, mult, power`, si lors de la compilation l'analyse du type produit une erreur :

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

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

Exercice 2 : évaluation, λ-calcul. Considérez l'expression du λ-calcul suivante :

$$(\lambda x.\lambda f.(f\,x))\,z\,((\lambda g.\lambda y.(g\,y))\,w)$$

Question 2.1. Quels sont les redexes qui apparaissent dans cette expression ? (Fin question 2.1)

Question 2.2. Évaluez cette expression en utilisant l'ordre applicatif (évaluation par l'intérieur).

(Fin question 2.2)

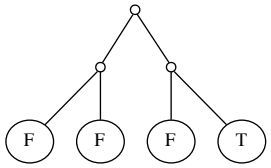
Question 2.3. Évaluez cette expression en utilisant l'ordre normal (évaluation par l'extérieur). (Fin question 2.3)

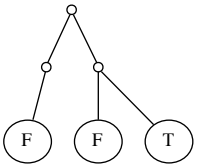
Exercice 3 : types récursifs et λ-calcul. Définissez en `Haskell` un type de données—qui sera appélé `LambdaTerm`—pour coder les termes (ou expressions) du λ-calcul. Écrivez ensuite une expression `Haskell` de type `LambdaTerm` qui soit le codage du terme

$$\lambda f.\lambda x.(f\,x)\,.$$

Exercice 4 : types récursifs. Un arbre de décision est un arbre dont les noeuds internes possèdent un ou deux enfants et dont les feuilles sont étiquetées par des valeurs d'un domaine donné. Ces arbres peuvent s'utiliser pour représenter des fonctions dont les arguments sont des tuplets de Booléens. Par exemple, la fonction AND, dont la table se trouve ci-dessous sur la gauche, peut se représenter par l'arbre ci-dessous au centre :

x_1	x_2	x_1 AND x_2
F	F	F
F	T	F
T	F	F
T	T	T





La méthode est la suivante : à partir de la racine, on parcourt l'arbre en lisant le tuplet de Booléens (False=descendre à gauche, True=descendre à droite) en entrée pour découvrir sur la feuille la valeur de la fonction quand appliquée à ce tuplet. On exploite la possibilité que les noeuds internes aient un seul enfant pour réduire la taille de la représentation d'une fonction lorsque la valeur à calculer ne dépend pas d'une variable. Par exemple, la fonction AND peut aussi se représenter par l'arbre de décision ci-dessus sur la droite, en utilisant le fait que quand la première variable vaut Faux, alors quelque soit la valeur de la deuxième variable, on obtient le résultat Faux.

Voici la définition en `Haskell` d'un type de données récursif pour les abres de décision binaires :

```
data ArbreDec a = Value a
                | FalseTrue (ArbreDec a) (ArbreDec a)
                | DontCare (ArbreDec a)
```

Question 4.1. Représentez la fonction AND par une expression `Haskell` ayant type `ArbreDec Bool`.

(Fin question 4.1)

Question 4.2. Dessinez l'arbre de décision qui représente la fonction Booléenne de trois variables qui retourne `Vrai` si et seulement si la première et la deuxième variables ont la même valeur, ou la deuxième et la troisième variables ont la même valeur. Représentez ensuite cet arbre par une expression `Haskell` de type `ArbreDec Bool`. (Fin question 4.2)

Question 4.3. Écrivez la définition d'une fonction

```
eval :: ArbreDec a ->  [Bool] -> a
```

qui prend en paramètre un arbre de décision (représentant une fonction) et une liste de Booléens (représentant un tuplet), et évalue cette fonction avec ce tuplet comme entrée. (Fin question 4.3)

Question 4.4. La fonction `eval` demande deux paramètres, un arbre de décision qui code une fonction demandant un tuplet de *longueur fixée* de paramètres Booléens, et une liste de *longueur variable* de Booléens. On pourrait donc avoir que la longueur de la liste ne correspond pas à la longueur du tuplet demandé par la fonction. Modifiez votre définition (et, si nécessaire, le type) de la fonction `eval` pour considérer cette éventualité comme une erreur. (Fin question 4.4)

Exercice 5 : suites d'entiers. On s'amuse, dans le script suivant, à définir trois listes infinies d'entiers :

```
suite1 = suite 1 2
    where suite n m = n:suite m (n+m)
suite2 = suite 2 1
    where suite n m = n^m:suite n (m+1)
suite3 = suite 1
    where suite n = n:map (n*) (suite (n+1))
```

Question 5.1. Décrivez en langue française (ou en utilisant une notation mathématique appropriée) quelles sont les suites définies par `suite1,suite2,suite3`. (Fin question 5.1)

Question 5.2. Avec les définitions ci-dessus, l'évaluation de l'expression

```
concat (map (\(x,xs) -> take x xs) (zip [4..6] [suite1,suite2,suite3]))
```

produit, en `Haskell`, une valeur. Pouvez vous deviner quelle est cette valeur ? Donnez une justification détaillée de votre réponse. (Fin question 5.2)