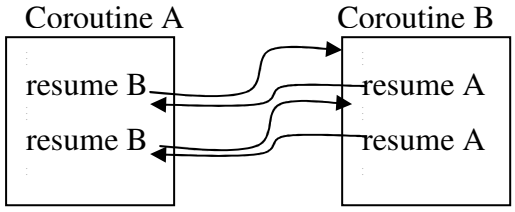
Université Mentouri de Constantine **Le 15/02/2009**
Faculté des Sciences de l'Ingénieur **Section : 4ième Année ing.**
Département d'Informatique **Durée : 1h30**

Sémantique Formelle et Paradigmes des langages de Programmation
Corrigé type de l'Examen N°01

Questions de cours (9 pts)

1. Oui, les **coroutines** sont un type spécial de sous programmes et possèdent plusieurs entrées appelées **points de reprise** (cas du langage Ada) (1 pt)



2. Notion de dépendance temporelle: à l'exécution d'un programme impératif, si plusieurs évaluations sont possibles, alors le résultat final dépendra, au cas où les sous expressions exercent un effet de bord sur l'état du programme, de l'ordre d'exécution (donc du temps).

Exemple :

on suppose que la variable n est initialement égale à 0 :
n + f(n) // on suppose que f réalise un effet de bord sur n, f(n) change la valeur de n en 1
= 0 + (n :=1) // on applique d'abord l'évaluation de la partie gauche
= 0 + 1
= 1
Si on évalue l'expression on commençant par la partie droite de l'équation, on aura :
n + f(n) // toujours avec n initialement égale à 0
= n + (n :=1) // on évalue f d'abord
= N + 1
=1+ 1
=2
(1pt)

3. L'évaluation paresseuse est une stratégie d'évaluation selon laquelle les arguments de fonctions ne sont évalués que s'ils sont nécessaires pour le calcul.

Exemple :

inf :: Int // l' infini est de type integer
inf = 1 + inf
si on évalue cette valeur
1 + (1+ (1 + …. : le calcul ne se termine pas
Considérons maintenant la fonction fst qui retourne le premier élément d'une paire.
L'évaluation par valeur où l'argument est d'abord évalué avant l'application de la fonction donne le résultat suivant :
fst (0, inf)
= fst (0 , 1+ inf)
= fst (0, 1+ (1+inf))
= fst (0, 1+(1+(1+(…..)))
Par contre, une évaluation paresseuse produit le bon résultat en une seule étape :
fst(0,inf)

= 0

(1 pt)

4. La curryfication d'une fonction est la transformation d'une fonction à n arguments en une fonction qui ne prend qu'un seul argument à la fois en exploitant la notion d'ordre supérieur.

Exemple

La fonction add

```
add ::  Int* Int -> Int
add(x, y) = x+ y
```

Version curryfiée de add :

```
add' :: Int -> (Int -> Int)
add'  x  y = x+ y
```

add'(1) = inc (la fonction qui incrémente un entier de 1)

(1 pt)

5. Notion de fonction d'ordre supérieur : une fonction possédant des paramètres fonctionnels ou/et retournant comme résultat une fonction.

Exemple en Haskell

```
filter p [ ]  = [ ]
filter p (x :xs)  | p x    =  x : filter p xs
                  | otherwise  filter p xs
```

(1 pt)

6. Avantage du schéma conceptuel de reprise : mécanisme extrêmement flexible. Désavantage : difficulté sémantique (lisibilité et maintenance du programme) (1 pt)
7. Un programme logique est plus abstrait car il implémente une relation (une correspondance cde type plusieurs-à-plusieurs) entre les entrés et les sorties, contrairement à un programme impératif ou fonctionnel qui implémentent un mapping (une correspondance plusieurs-à-un) des entrées vers une sortie (1 pt).
8. Portée d'une variable : **l'étendue des instructions où la variable est connue, c'est un attribut syntaxique,** Durée de vie : **le laps de temps pendant lequel il existe un emplacement mémoire pour la variable.** C'est un **attribut temporel**.
9. On peut simplifier algébriquement l'expression suivante x+ f(x,y) +x+y+f(x,y) en 2x+2f(x,y)+y si f n'a aucun effet de bord sur les variables x et y (1 pt):

Exercice N°01 : (4 pts)

la preuve de type:

1. (2 pts)

$$\dfrac{\dfrac{\dfrac{E \mapsto x : \alpha}{E \mapsto \lambda x \to x : \alpha \to \alpha}(\lambda - abstraction) \quad \dfrac{E(x) = \alpha}{E \mapsto x : \alpha}(Var)}{E \mapsto (\lambda x \to x)\ x : \alpha}(\lambda - application)}{E \mapsto lety = (\lambda x \to x)\ in\ y\ x : \alpha}(let)$$

2. (1.5 pts)

$$\dfrac{\dfrac{\dfrac{\dfrac{E \mapsto x : \alpha}{E \mapsto f : \alpha \to \beta}(\lambda - abstraction) \quad \dfrac{E(x) = \alpha}{E \mapsto x : \alpha}(Var)}{E \mapsto f\ x : \beta}(\lambda - application)}{E \mapsto f\ f\ x : \beta,\ E \mapsto f\ x : \alpha\ (\alpha \equiv \beta)}(\lambda - application)}{E \mapsto twice : (\alpha \to \alpha) \to \alpha \to \alpha}(\lambda - abstraction)$$

Exercice N°02 (3 pts) :

- Équivalence nominative : {a,b}, {c , d} {e , f }. (0.50 pt)
- Équivalence déclarative : {a ,b}, {c ,d} { e} ,{f}. (0.50 pt)
- Équivalence structurelle : {a,b,c,d,e,f}. (0.50 pt)

2)

a) Récursivité & pattern matching (1 pt)

[ ] !! n = Error

[x :xs] !! n = if n = = 0 then x else xs !! (n-1)

b) La fonction drop, et head (0.50 pt)

head (drop n xs) = xs!! n

Donc, !! = head o drop , où o est la composition de fonctions.

Exercice N°03 (4 pts)

foldr f v [ ] = v

foldr f v [x : xs] = f x (foldr f v xs)

1. foldr (+) 0 [1,2,3,4] =10, (1pt)

2. foldr (+) 0 = sum

sum :: Num a ⇒ [a] → a // signature de la fonction & contrainte de classe

sum [ ] = 0

sum [x: xs] = x + sum xs

3. foldr (*) 1 = product

product :: Num a ⇒ [a] → a // signature de la fonction & contrainte de classe

product [ ] = 1

product [x : xs] = x * product xs

4. type de foldr : **(a -> b -> b) -> b -> [a] -> b (1 pt):**