Corrigé Examen Semestriel TPGO (2CS) – ESI 20182019

Exercice 1 :

- La décomposition récursive de C(k,S) le nombre minimal de pièces ayant des valeurs dans $\{v_1, v_2, \dots v_k\}$ (en ordonnant les valeurs v_i en ordre croissant) et totalisant la somme S, peut être exprimée ainsi :

Cas particulier :
Pour totaliser une somme nulle, le nombre minimal de pièces à utiliser est toujours 0

C(k, 0) = 0 pour tout k

Cas général :
Si qu'on sait comment totaliser :
* une somme S avec un nombre minimal X de pièces dans $\{v_1, v_2, \dots v_{k-1}\}$ c-a-d X = C(k-1,S)
* une somme S-v_k avec un nombre minimal Y de pièces dans $\{v_1, v_2, \dots v_k\}$ c-a-d Y= C(k,S-v_k)
Donc on pourra totaliser S soit avec X pièces de $\{v_1, v_2, \dots v_{k-1}\}$ soit avec Y pièces de $\{v_1, v_2, \dots v_k\}$ plus une pièce de valeur v_k.
Le nombre minimal de pièces pour totaliser S sera donc le plus petit entre X et 1+Y
(c'est le principe d'optimalité qui vérifié par ce problème)

C(k, S) = min { C(k-1,S) , 1 + C(k,S-v_k) }
Si la valeur de la pièce k : v_k > S alors C(k, S) = C(k-1,S)
Si k=1 alors C(1, S) = 1 + C(1,S-v_1)
Si v_1 > S alors C(1, S) = 0

- Le problème de l'approche récursive pour le calcul de C(k , S) est que beaucoup de **sous-problèmes** vont être **calculés plusieurs fois**, ce qui va engendrer une exécution inefficace.

Exercice 2 :
Soit P le programme suivant :

```
i ← 0 ; R ← 0 ;
TQ ( (2*i+1) ≤ n )
        R ← R + 4*i + 1;
        i ← i+1
FTQ
```

- Donnez l'arbre de preuve complet (en mettant en évidence les règles d'inférence utilisées) généré pour la démonstration d'un énoncé de la forme E { P } S dans le système formel de HOARE.

Posons :
P = [P1 ; P2]
P1 = [i ← 0 ; R ← 0 ;]
P2 = [TQ (B) P3 FTQ]
B = (2*i+1) ≤ n
P3 = [R ← R + 4*i + 1; i ← i+1]

Pour que l'énoncé (racine) E{P}S soit vrai, il suffirait par la règle SEQ que :
E{P1}F et F{P2}S soient vrais aussi.

Pour que l'énoncé F{P2}S soit vrai, il suffirait par la règle IMP2 que :

F{P2}(F & non B) et (F & non B) $\Rightarrow$ S soient vrais aussi.

F est l'invariant de la boucle.

Pour que l'énoncé F{P2}(F & non B) soit vrai, il suffirait par la règle ITE que :

(F & B){P3}F soit vrais aussi (car P2 est une instruction TQ).

Pour que l'énoncé (F & B){P3}F soit vrai, il suffirait par la règle SEQ que :

(F & B){R $\leftarrow$ R + 4*i + 1}G et G{i $\leftarrow$ i+1}F soient vrais aussi.

L'énoncé G{i $\leftarrow$ i+1}F est vrai car en prenant G=[i+1/i]F, il devient un axiome de l'affectation (AFF).

Pour que l'énoncé (F & B){R $\leftarrow$ R + 4*i + 1}G soit vrai, il suffirait par la règle IMP1 que :

(F & B) $\Rightarrow$ H et H {R $\leftarrow$ R + 4*i + 1}G soient vrais aussi.

L'énoncé H {R $\leftarrow$ R + 4*i + 1} G est vrai car en prenant H=[R+4i+1/R]G, il devient un axiome de l'affectation (AFF).

Pour que l'énoncé E{P1}F soit vrai, il suffirait par la règle SEQ que :

E{i $\leftarrow$ 0}I et I{R $\leftarrow$ 0}F soient vrais aussi.

L'énoncé I{R $\leftarrow$ 0}F est vrai car en prenant I=[0/R]F, il devient un axiome de l'affectation (AFF).

Pour que l'énoncé E{i $\leftarrow$ 0}I soit vrai, il suffirait par la règle IMP1 que :

E $\Rightarrow$ J et J {i $\leftarrow$ 0} I soient vrais aussi.

L'énoncé J {i $\leftarrow$ 0} I est vrai car en prenant J=[0/i]I, il devient un axiome de l'affectation (AFF).

Pour compléter la preuve, il suffit de trouver un bon invariant F et vérifier les implications suivantes :

a) (F & non (2i+1 $\leq$ n)) $\Rightarrow$ S

b) (F & 2i+1 $\leq$ n) $\Rightarrow$ H avec H = [R+4i+1/R] [i+1/i] F

c) E $\Rightarrow$ J avec J = [0/i] [0/R] F

- Trouver un bon invariant de boucle pour P en considérant les prédicats d'entrée et de sortie suivants : E : (n $\geq$ 0 && impair(n)) et S : (R = n(n + 1)/2)

Avec la post-condition S donnée, l'implication a) devient :

a) (F & non(2i+1 $\leq$ n)) $\Rightarrow$ R = n(n + 1)/2

c-a-d (F & 2i+1 > n)) $\Rightarrow$ R = n(n + 1)/2

Il faudrait donc trouver une condition F pour que l'implication a) devienne vrai et il faudrait aussi que cette condition F soit invariante pour la boucle TQ.

Rappelons que l'invariant de boucle doit être vrai au début et à la fin de boucle. De plus il doit être vrai aussi au début et à la fin de chaque itération de la boucle.

Nous remarquons qu'à la fin de la boucle TQ, la condition (2i+1)>n devient vraie.

C-a-d, on aura i > (n-1)/2

D'un autre coté, durant les itérations de la boucle, la condition du TQ ($2i+1 \leq n$), c-a-d : $i \leq (n-1)/2$ reste vraie.
A la dernière itération du TQ, la variable 'i' est incrémentée pour rendre faux la condition du TQ.
On peut proposer alors que $i \leq (n-1)/2 + 1$ fasse partie de l'invariant. Elle est vraie durant les itérations du TQ et aussi à la fin du TQ (lorsque i devient supérieur à (n-1)/2)
$i \leq (n-1)/2 + 1$ se simplifie en : $i \leq (n+1)/2$
En déroulant la boucle TQ, on peut remarquer que la variable R contient toujours la somme des k+1 premiers entiers naturels $0 + 1 + 2 + \ldots + k$, avec k un nombre impaire.
Cela suggère que $R = k(k+1)/2 = (2(i-1)+1)\ (\ (2(i-1)+1) + 1\ ) / 2$ (en remplaçant k par 2(i-1)+1)
En simplifiant, on trouve $R = 2i^2-i$
L'invariant (F) proposé est donc : $R = 2i^2-i$ et $i \leq (n+1)/2$
Mais cela ne suffit pas pour rendre l'implication a) vraie
En effet : ($2i^2-i$ et $i \leq (n+1)/2$ et $2i+1 > n$)) $\Rightarrow$ $R = n(n + 1)/2$ que lorsque n est impair
Comme la variable 'n' ne change pas durant la boucle TQ, on peut alors rajouter cette condition dans F et cela restera un invariant.
L'invariant F proposé est donc **: impair(n) et $R = 2i^2-i$ et $i \leq (n+1)/2$**

L'implication a) est vraie car si n est impair et si $i \leq (n+1)/2$ et en même temps $i > (n-1)/2$ (car dans l'antécédent de l'implication on a $2i+1 > n$), 'i' doit forcément être égal à (n+1)/2. Dans ce cas alors, $R = 2i^2-i = 2((n+1)/2)^2-(n+1)/2 = n(n+1)/2$.

Pour que le F soit le bon invariant, il faudrait qu'il vérifie les deux autres implications :
b) (F & $2i+1 \leq n$) $\Rightarrow$ H avec H = [R+4i+1/R] [i+1/i] F
La condition H devient alors : impair(n) et $R+4i+1 = 2(i+1)^2-(i+1)$ et $i+1 \leq (n+1)/2$
L'implication est vraie car la condition $R+4i+1 = 2(i+1)^2-(i+1)$ se simplifie en $R = 2i^2-i$ (déjà présente dans l'antécédent de l'implication, dans F) et la condition $i+1 \leq (n+1)/2$ est impliquée par $2i+1 \leq n$ (déjà présente dans l'antécédent de l'implication).
L'implication b) est donc vraie.
c) E $\Rightarrow$ J avec J = [0/i] [0/R] F
La condition J devient : impair(n) et $0=0$ $(n+1)/2 \geq 0$
L'implication c) est donc vraie car impair(n) est déjà présente dans E et la condition $(n+1)/2 \geq 0$ est aussi impliquée par $n \geq 0$ qui est aussi présente dans E.

Exercice 3 :
- Donnez un programme Prolog (purement logique) permettant de répartir dans R, les éléments d'une liste L de nombres entiers en fonction d'une valeur pivot P : arranger(L , P , R)
Ce prédicat sera évalué à vrai, si la liste R contient les mêmes éléments que la liste L, réarrangés de sorte à ce que tous les éléments de L plus petits ou égaux à P se trouvent dans la partie gauche de R et tous les éléments de L plus grands que P se trouvent dans la partie droite de R.

Le résultat de l'arrangement d'une liste [X|L] par rapport à un pivot P donne une liste qui commence par X et se termine par une liste L2 c-a-d [X|L2], si $X \leq P$ et l'arrangement de L, donne comme résultat L2 :

```
arranger( [X|L] , P , [X|L2] ) :-  X ≤ P , arranger( L , P , L2 ).
```

Si par contre X était > P, alors cela donnera une liste résultat L2 qui se termine par X (par exemple en insérant X en fin de liste). Les autres éléments de la liste résultat L2 sont obtenus par l'arrangement de L :

```
arranger( [X|L] , P , L2 ) :-  X > P , arranger( L , P , L3 ) , ins_fin( X , L3 , L2 ).
```

L'insertion d'un élément X en fin d'une liste L peut être défini comme suit :

```
// l'insertion de X dans une liste vide [], donne une liste a un élément [X]
ins_fin( X , [] , [X] ).
// l'insertion de X dans une liste non vide [Y|L], donne une liste qui commence par le
// premier élément Y de la liste en entrée et le reste est obtenu en insérant X en fin de L
ins_fin( X , [Y|L] , [Y|L2] ) :-
                        ins_fin( X , L , L2 ).
```

- Quelle est l'utilité des clauses de Horn en programmation logique ?

L'utilisation des clauses de Horn en programmation logique à la place des clauses quelconques permet de rendre la procédure de dérivation plus rapide, car l'espace de recherche est alors plus restreint. Si les hypothèses sont décrite uniquement à l'aide d'assertion et d'implication et le but est sous forme d'une dénégation alors la recherche de la clause vide ne se fait que dans les branches dont l'une des prémisses est une dénégation. Le nombre d'alternatives à chaque étape est ainsi grandement diminué par rapport à la résolution générale.

Exercice 4 :

- En utilisant le théorème du point fixe, trouver la fonction calculée exactement par le programme récursif suivant :

P = λx.λy. SI (< x y) 1 (* y (P (- x y) y))

La fonction exactement calculée par le programme est le plus petit point fixe de l'équation
P = Tau(P)
avec Tau le programme récursif : (λf.λx.λy. SI (< x y) 1 (* y (f (- x y) y))) f

D'après le théorème du point fixe, la plus petite solution (au sens de la relations « moins définie que ») est : Sup { Tau^n(Oméga) }$_{n\geq 0}$ avec Oméga le plus petit élément de $\mathscr{I}$(E,F), c-a-d la fonction dont le domaine de définition est vide.

Calculons d'abord Tau(Oméga) :
Tau(Oméga) = λx.λy. SI (< x y) 1 (* y (Oméga (- x y) y))
Tau(Oméga) = 1 si x<y, sinon non définie

Calculons d'abord Tau^2(Oméga) :
Tau^2(Oméga) = λx.λy. SI (< x y) 1 (* y (Tau(Oméga) (- x y) y))
Tau^2(Oméga) = 1 si x<y et,
Tau^2(Oméga) = y*1=y si (x≥y & x-y<y), c-a-d si y≤x<2y

Calculons d'abord Tau^3(Oméga) :
Tau^3(Oméga) = λx.λy. SI (< x y) 1 (* y (Tau^2(Oméga) (- x y) y))
Tau^3(Oméga) = 1 si x<y et,
Tau^3(Oméga) = y*1=y si (x≥y & x-y<y), c-a-d si y≤x<2y
Tau^3(Oméga) = y*y=y^2 si (x-y≥y & x-y-y<y), c-a-d si 2y≤x<3y

On peut alors émettre l'hypothèse de récurrence, que
Tau^k(Oméga) = $y^{(x \text{ div } y)}$ si x < ky

Montrons que :
$Tau^{k+1}(Oméga) = y^{(x \text{ div } y)}$ si $x < (k+1)y$

D'après la définition de Tau, on a :
$Tau^{k+1}(Oméga) = \lambda x.\lambda y.$ SI (< x y) 1 (* y (Tau^{k}(Oméga) (- x y) y))
donc :
$Tau^{k+1}(Oméga) = 1$ si $x<y$ et,
$Tau^{k+1}(Oméga) = y*Tau^{k}(Oméga)\ (x-y,y)$ sinon
D'après l'hypothèse de récurrence, on a :
$Tau^{k}(Oméga)\ (x-y,y) = y^{(x-y \text{ div } y)}$ si $x-y < ky$, c-a-d si $x<(k+1)y$
or $y*y^{(x-y \text{ div } y)} = y^{(x \text{ div } y)}$
Donc :
$Tau^{k+1}(Oméga) = 1$ si $x<y$ et,
$Tau^{k+1}(Oméga) = y^{(x \text{ div } y)}$ si $y \leq x<(k+1)y$

On a donc bien :
$Tau^{k+1}(Oméga) = y^{(x \text{ div } y)}$ si $x < (k+1)y$

L'hypothèse de récurrence est donc vrai quelque soit k

Le plus petit point fixe est donc la fonction
$Tau^{+\infty}(Oméga) = y^{(x \text{ div } y)}$ pour tout x et y
C'est la fonction calculée exactement par le programme récursif donné.

- Peut-on donnez un seul programme récursif et purement fonctionnel permettant de calculer partiellement n'importe quelle fonction de $\mathscr{I}$(E,F) (l'espace de toutes les fonctions de E dans F) ?

Un programme P calcule partiellement une fonction f, si la fonction calculée exactement par P (son plus petit point fixe) est moins définie que f, c-a-d quelque soit l'entrée x, pour laquelle le programme P s'arrête, f(x) est définie et est égale au résultat retourné par P.
Puisque la fonction Oméga est le plus petit élément de $\mathscr{I}$(E,F) (au sens de la relation « moins définie que »), elle est donc moins définie que n'importe quelle fonction de $\mathscr{I}$(E,F). Un programme qui calcule Oméga, calcule donc partiellement toutes les fonctions de l'espace $\mathscr{I}$(E,F).
Pour calculer exactement Oméga, il suffit de donner un programme qui boucle quelque soit l'entrée donnée. Ce programme calcule partiellement toutes les fonctions de $\mathscr{I}$(E,F).
Par exemple : P = $\lambda x.$ (P x)