Corrigé - Examen Semestriel – Théorie de la Programmation (TPGO) – 2CS – ESI 2016/2017

***1**- Questions de cours*

a) Quel type de parcours (en profondeur ou en largeur) est effectué par 'Branch and Bound' ?

⇒ En largeur, l'algorithme utilise une file d'attente ordonnée (file de priorité) pour explorer les différents chemins engendrés.

b) Pourquoi a t-on besoin, généralement, d'une fonction d'estimation d'une configuration dans la procédure MinMax avec ou sans élagage alpha-bêta ?

⇒ Car généralement la profondeur des arbres de jeux considéré est trop importante pour atteindre les configurations feuilles dès le début. On utilise alors des fonction d'estimations permettant de retourner des coûts estimés sur des configurations non terminales. Ainsi on peut limiter la profondeur d'exploration et avoir un temps de réponse raisonnable même si l'espace de recherche est très grand.

c) Si on représente un programme récursif (fonctionnel) par une équation à point fixe, expliquez comment on pourrait trouver la fonction exactement calculée par le programme ?

⇒ Pour cela on utilise le théorème du pont fixe qui donne un moyen de construire de manière progressive la fonction exactement calculée par le programme.
Cette fonction est la plus petite solution de l'équation f = T(f) et qui est donnée par :
Sup {w, T(w), T(T(w)), …. T(T(T….(w))))) …. }

d) Quel est le rôle procédural des opérations d'unification utilisées dans le processus de preuve, lors de l'exécution d'un programme logique ?

⇒ L'exécution d'un programme logique se fait à l'aide d'un démonstrateur de théorème par réfutation, utilisant la règle de la résolution en vue de montrer la contradiction (génération de la clause vide).
Les unifications utilisées pour l'obtention des différentes résolvantes lors du processus de démonstration, constituent un ensemble d'affectations exécutées (par effets de bords) par l'interpréteur et représentent de ce fait une vision procédurale du programme logique en cours d'exécution.

e) Qu'est-ce qui caractérise la programmation purement fonctionnelle ?

⇒ L'absence d'affectation, l'absence d'instruction de contrôle du séquencement (séquentielle, boucle, branchement …) et une interprétation basée uniquement sur l'application de fonctions (l'exécution d'un programme purement fonctionnel se résume en l'évaluation d'une seule expression fonctionnelle)

***2**- Soit F(x,n,r) une procédure récursive et soit P son corps :*

```
F( x, n : entier ; var r:entier )          // x et n des entrées et r une sortie
  SI ( n = 0 )    r ← 1
  SINON
        F( x , n div 2 , r ) ;
        r ← r * r ;
        SI ( (n mod 2) = 1 )
              r ← x * r
        FSI
  FSI
```

a) Donnez la complexité de cet algorithme

Posons T(n) le temps nécessaire pour un appel à F(… n …).
On peut alors écrire l'équation de récurrence :
T(0) = a et T(n) = T(n/2) + b (pour n>0)

Par substitution, on peut facilement calculer la forme générale :
$T(n) = T(n/2) + b = T(n/4) + 2b = T(n/8) + 3b = \ldots T(0) + \log_2(n)\, b = a + b \log_2(n)$

Donc la complexité de cet algorithme est O(log n).

b) Montrez à l'aide du système formel de Hoare, que : (n ≥ 0) {P} (r = x^n)

Posons E : (n ≥ 0) et S : (r = x^n)
Alors l'énoncé initial (a) : E { P } S sera vrai par CND2 ssi les 2 énoncés suivants sont vrais aussi :
(b) E & n = 0 { r ← 1 } S
(c) E & n ≠ 0 { F(x , n div 2 , r) ;r ← r * r ; SI ((n mod 2) = 1) r ← x * r FSI } S

L'énoncé (b) est vrai par IMP1 car d'une part l'implication :E & n = 0 ⇒ (1 = x^n) est vraie et d'autre part l'énoncé : (1 = x^n) { r ← 1 } S est aussi vrai (axiome de l'affectation AFF).

L'énoncé (c) est vrai par SEQ si ces deux prémisses suivantes sont vraies aussi :
(d) E & n ≠ 0 { F(x , n div 2 , r) ;r ← r * r } F
(e) F {SI ((n mod 2) = 1) r ← x * r FSI } S

L'énoncé (e) est vrai par CND1 si on trouve un prédicat F vérifiant les deux prémisses ci-dessous :
(f) F & (n mod 2) = 1 { r ← x * r } S
(g) F & (n mod 2) ≠ 1 ⇒ S

L'énoncé (f) peut être vrai (par IMP1 et AFF) en choisissant prédicat F vérifiant :
(h) F & (n mod 2) = 1 ⇒ (x*r = x^n), l'énoncé (x*r = x^n) { r ← x * r } S étant un axiome.

Il faut donc choisir le prédicat F afin que les deux implications (g) et (h) soient vraies.
Le prédicat suivant convient :
F : ((n mod 2) = 1 & (x*r = x^n)) ou ((n mod 2) ≠ 1 & (r = x^n)

L'énoncé (d) peut être démontrer à l'aide la règle SEQ. Pour cela il faudrait alors que les deux énoncés suivants soient vrais :
(i) E & n ≠ 0 { F(x , n div 2 , r) } G
(j) G { r ← r * r } F

L'énoncé (j) est vrai par AFF en prenant pour G la condition :
((n mod 2) = 1 & (x*r^2 = x^n)) ou ((n mod 2) ≠ 1 & (r^2 = x^n)

L'énoncé (i) est vrai par IMP1 ssi :
(k) :E & n ≠ 0 ⇒ (n div 2 ≥ 0) et (l) : (n div 2 ≥ 0) { F(x , n div 2 , r) } G sont vrais.

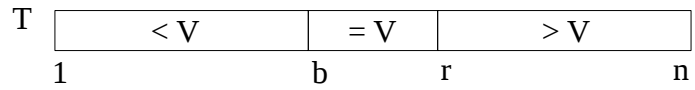
L'implication (k) est toujours vraie.

L'énoncé (l) est vrai par IMP2 ssi :
(m) : (n div 2 ≥ 0) { F(x , n div 2 , r) } (r = $x^{n \text{ div } 2}$) et
(n) :(r = $x^{n \text{ div } 2}$) ⇒ ((n mod 2) = 1 & (x*r^2 = x^n)) ou ((n mod 2) ≠ 1 & (r^2 = x^n)

L'énoncé (m) est l'hypothèse selon laquelle les appels internes sont considérés comme correctes pour effectuer la preuve en cours.

L'implication (n) est vraie dans le cas où n est pair (c-a-d : n mod 2 ≠ 1)
Dans ce cas n div 2 = n/2 et donc on aura : r^2 = $x^{n \text{ div } 2}$ * $x^{n \text{ div } 2}$ = x^n
L'implication (n) est vraie aussi dans le cas où n est impair (c-a-d : n mod 2 = 1)
Dans ce cas n div 2 = (n-1)/2 et donc on aura : x*r^2 =x * $x^{n \text{ div } 2}$ * $x^{n \text{ div } 2}$ = x^n

***3**- Soit Q l'algorithme de tri partiel de type « Drapeau Hollandais » :*
A partir d'une valeur donnée V (par exemple la 1ere valeur d'un tableau), l'algorithme réarrange, en une seule passe, le contenu du tableau de sorte à ce que toutes les valeurs inférieures à V soient placée en début du tableau et toutes les valeurs supérieures à V en fin de tableau. Les occurrences de la valeur V sont donc placées au milieu du tableau.




```
Q :     b ← 1 ; w ← 2 ; r ← n+1 ; V ← T[1] ;
        TQ ( r ≠ w )
            SI ( T[w] = V )   w ←w+1
            SINON  SI ( T[w] < V )
                        x ← T[w] ; T[w] ← T[b] ; T[b] ← x ;
                        b ← b+1 ; w ← w+1
                   SINON
                        x ← T[w] ; T[w] ← T[r-1] ; T[r-1] ← x ;
                        r ← r-1
                   FSI
            FSI
        FTQ
```

- Proposer un candidat pour un bon invariant de la boucle TQ, sachant que les prédicats d'entrée et de sortie sont comme suit :
E : (n ≥ 1 et V = T[1])
*S : (b ∈ [1,n] **et** r ∈ [b+1, n+1] **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<r ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V))*

Le principe de l'algorithme est de classer les éléments du tableau par des permutations en gérant quatre zones au niveau de la boucle TQ :
La zone 1 définie dans [1, b[contenant les valeurs déjà traitées et inférieures à V
La zone 2 définie dans [b, w[contenant les valeurs déjà traitées et égales à V
La zone 3 définie dans [b, r[contenant les valeurs non encore traitées
La zone 4 définie dans [r, n] contenant les valeurs déjà traitées et supérieures à V

A chaque itération, le premier élément de la zone 3 est transférer (par permutations) soit vers la fin de la zone 1 (s'il est < V) soit vers la fin de la zone 2 (c-a-d il reste sur place, s'il est = V) soit vers le début de la zone 4 (s'il > V). Dans tous les cas l'étendue de la zone 3 diminue d'un élément. L'algorithme s'arrête lorsque cette dernière (la zone 3) devient vide (tous les éléments auront été traitées).

L'invariant de la boucle devrait donc exprimer les relations entre les variables b,w et r permettant de maintenir ces propriétés.
En s'inspirant de la postcondition S on peut proposer comme invariant (F), la condition suivante :
F : [b ∈ [1,n] **et** w ∈ [b+1, n+1] **et** r ∈ [b+1, n+1] **et** w≤r **et**
(∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)]

Pour vérifier que c'est un bon invariant il faut vérifier certaines implications dans l'arbre de preuves :

La première implication, concerne la sortie de boucle, c-a-d lorsque la condition d'arrêt de la boucle devient vérifiée, il faudrait alors avoir (1) F & w=r ⇒ S
Cette implication est vraie car lorsque w=r, toutes les composantes de S sont déjà dans l'antécédent de l'implication (dans F).

La deuxième implication concerne l'entrée de boucle. Elle provient de la preuve de la partie :
E { b ← 1 ; w ← 2 ; r ← n+1 ; V ← T[1] } F
En appliquant SEQ, AFF et IMP1, l'énoncé précédent sera vrai si l'implication suivante est vraie :
(2) : E ⇒ [1 ∈ [1,n] **et** 2 ∈ [2, n+1] **et** n+1 ∈ [2, n+1] **et** 2 ≤ n+1 **et**
(∀j 1≤ j<1 ⇒ T[j]<T[1]) **et** (∀j 1≤j<2 ⇒ T[j]=T[1]) **et** (∀j n+1≤j≤n ⇒ T[j]>T[1])]
Cette implication (2) est vraie car tous les composants du conséquent sont soit impliqués par un composant de E (n ≥ 1) soit sont vrais (cas des 3 dernières implications du conséquent).

La dernière implication provient de la preuve du corps de la boucle :
F et r ≠ w { P } F avec P le corps de la boucle TQ
En évaluant la plus faible précondition (G) en début d'itération, on trouve :
G : (T[w]=V & [w+1/w]F) ou (T[w]≠V & (T[w]<V & ([b+1/b][w+1/w]F)$_{en\_permutant\_T[w]\_T[b]}$) ou
(T[w]≥V & ([r-1/r]F)$_{en\_permutant\_T[w]\_T[r-1]}$))
L'implication (3) est donc :
[b ∈ [1,n] **et** w ∈ [b+1, n+1] **et** r ∈ [b+1, n+1] **et** w≤r **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)]
⇒
(T[w]=V & [b ∈ [1,n] **et** w+1 ∈ [b+1, n+1] **et** r ∈ [b+1, n+1] **et** w+1≤r **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w+1 ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)])
ou
(T[w]≠V &
(T[w]<V & [b+1 ∈ [1,n] **et** w+1 ∈ [b+2, n+1] **et** r ∈ [b+2, n+1] **et** w+1≤r **et** (∀j 1≤ j<b+1 ⇒ T[j]<V) **et** (∀j b+1≤j<w+1 ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)]$_{en\_permutant\_T[w]\_T[b]}$)
ou
(T[w]≥V & [b ∈ [1,n] **et** w ∈ [b+1, n+1] **et** r-1 ∈ [b+1, n+1] **et** w≤r-1 **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w ⇒ T[j]=V) **et** (∀j r-1≤j≤n ⇒ T[j]>V)]$_{en\_permutant\_T[w]\_T[r-1]}$)
)

En appliquant les permutations spécifiées, on aura pour l'implication (3) :

[b ∈ [1,n] **et** w ∈ [b+1, n+1] **et** r ∈ [b+1, n+1] **et** w≤r **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)]
⇒
(T[w]=V & [b ∈ [1,n] **et** w+1 ∈ [b+1, n+1] **et** r ∈ [b+1, n+1] **et** w+1≤r **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w+1 ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)])
ou
(T[w]≠V &
(T[w]<V & [b+1 ∈ [1,n] **et** w+1 ∈ [b+2, n+1] **et** r ∈ [b+2, n+1] **et** w+1≤r **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)])
ou
(T[w]≥V & [b ∈ [1,n] **et** w ∈ [b+1, n+1] **et** r-1 ∈ [b+1, n+1] **et** w≤r-1 **et** (∀j 1≤ j<b ⇒ T[j]<V) **et** (∀j b≤j<w ⇒ T[j]=V) **et** (∀j r≤j≤n ⇒ T[j]>V)])
)

4*- On peut représenter les entiers en λ-calcul par des expressions de la forme :*

0 : λf.λx.x
1 : λf.λx.(f x)
2 : λf.λx.(f(f x))
… …
n : λf.λx.(f^n x)

- Que représente alors l'expression suivante : λn.(n (λx.False) True)

En appliquant cette expression sur 0, on obtient True :
λn.(n (λx.False) True) (λf.λx.x) → ((λf.λx.x) (λx.False) True) → ((λx.x) True) → True

En appliquant cette expression sur un nombre différent de 0, on obtient False :
λn.(n (λx.False) True) (λf.λx.(f x)) → ((λf.λx.(f x)) (λx.False) True) → ((λx.((λx.False) x)) True) → ((λx.False) True) → False

Montrons par récurrence : ∀ N > 0, *λn.(n (λx.False) True) N → False*
Pour N = 1 = (λf.λx.(f x)), l'expression se réduit en False (comme vu précédemment)
Hypothèse de récurrence : ∀ N > 0 et N < M, *λn.(n (λx.False) True) N → False*

Montrons alors : λn.(n (λx.False) True) M → False

Si N = λf.λx.(f^n x) alors M = λf.λx.(f (f^n x))
et donc :
λn.(n (λx.False) True) M = λn.(n (λx.False) True) (λf.λx.(f (f^n x)))
→ ((λf.λx.(f (f^n x))) (λx.False) True)
→ ((λx.((λx.False) ($(λx.False)^n$ x))) True) où ($(λx.False)^n$ x) = ((λx.False) (… ((λx.False) x))) [n fois]
or la sous-expression *((λx.False) ($(λx.False)^n$ x)) se réduit toujours en False*
donc l'expression globale se réduira en :
((λx.False) True) qui se réduit forcément en : False.

Donc ∀N > 0 , λn.(n (λx.False) True) N → False

Donc cette expression représente la fonction qui teste si son argument est 0