ENSIIE 3A - Sémantique des langages - janvier 2011
Durée : 2h - Avec documents (sauf ouvrage) - calculette sans objet - ordinateur interdit Correction

Les exercices sont indépendants et peuvent être traités dans n'importe quel ordre.

Exercice 1 (**Logique de Hoare**)

On considére le programme *Prog* suivant :

```
while y ≠ x do
   x:=x-1;
   y:=y-2;
end;
```

Question 1

Démontrer que le triplet de Hoare $\{x \leq y\}$ *Prog* $\{x = y\}$ est valide. Vous dessinerez pour cela un arbre de dérivation qui utilise les règles de la logique de Hoare.

Pour montrer que I est un invariant, selon la règle du while de la logique de Hoare, il faut démontrer le triplet de Hoare suivant $\{y \neq x \ \wedge \ I\}\ C\ \{I\}$ avec C désignant le corps de la boucle. Prendre pour I la formule $x \leq y$.

On démontre aisément $\{x - 1 \leq y - 2\}\ x := x - 1; y := y - 2\ \{x \leq y\}$ (on calcule la plus faible précondition associée à la post-condition $x \leq y$ en remontant le corps de la boucle). Comme on a $(y \neq x \ \wedge \ x \leq y) \Rightarrow x - 1 \leq y - 2$ on en déduit, en utilisant la règle de la conséquence logique gauche, le triplet $\{y \neq x \ \wedge \ x \leq y\}\ x := x - 1; y := y - 2\ \{x \leq y\}$. Et donc en utilisant la règle du while on obtient $\{\ x \leq y\}$ *Prog* $\{x \leq y \ \wedge \ \neg(y \neq x)\}$.

De plus comme on a $x \leq y \ \wedge \ \neg(y \neq x) \Rightarrow x = y$, en utilisant la règle de conséquence droite, on obtient le triplet demandé à savoir $\{\ x \leq y\}$ *Prog* $\{x = y\}$.

Question 2

On a donné en cours les règles de la logique de Hoare pour la correction partielle. Modifiez les règles pour qu'elles traitent de la correction totale. Le triplet sera alors noté $[P]\ c\ [Q]$ et signifie que si on exécute c dans un état qui satisfait P alors c termine dans un état qui satisfait Q.

On introduit un variant. Il faut traduire qu'il s'agit d'une quantité positive et qu'elle décroît strictement au cours d'iune itération.

$$\frac{[I \wedge e \wedge V = n]\ c\ [I \wedge V \geq 0 \wedge V < n]}{[I]\ \mathit{while}\ e\ \mathit{do}\ c\ [I \wedge \neg e]}$$

Question 3

On s'intéresse maintenant à la correction totale du programme *Prog*. Démontrez que le triplet $[x \leq y]$ *Prog* $[x = y]$ est valide. Vous mettrez en évidence les nouvelles parties de la démonstration.

Prendre pour variant l'expression $y - x$.

Exercice 2 (**Sémantique opérationnelle mini-langage impératif**)

On considère le langage impératif étudié en cours muni de sa sémantique opérationnelle à grand pas.

1. Équivalence de programmes

   Soit c_1 le programme `while 0 ≤ x do y := 5 ; x :=a od;`
   Soit c_2 le programme `y:= 5; while 0 ≤ x do x :=a od;`
   Dans les deux programmes précédents, `a` désigne une expression arithmétique quelconque (mais c'est la même dans les deux cas).

   Question 4

Les deux programmes c_1 et c_2 sont-ils équivalents (au sens de la relation $\equiv$ définie en cours) ?

Non ils ne le sont pas. Si l'on part d'un état initial σ tel que $\sigma(x) < 0$ et $\sigma(y) \neq 5$ on obtient des états différents. En effet dans ce cas, l'exécution de c_1 laisse l'état inchangé : on ne rentre pas dans la boucle. L'exécution de c_2 résulte dans un état $\sigma' = \sigma[y \to 5]$: en effet on exécute la première commande (`y:=5`) mais on ne rentre pas dans la boucle.

On définit une nouvelle relation d'équivalence un peu moins forte : on dira que deux programmes c_1 et c_2 sont S-équivalents (où S est un ensemble d'états) et on écrira $c_1 \equiv_S c_2$ si *pour état σ de S*, si l'on exécute c_1 (resp. c_2) dans σ et qu'il termine dans un état σ' alors c_2 (resp. c_1) exécuté dans σ termine également dans l'état σ'.

Question 5

Pourquoi cette relation d'équivalence est-elle moins forte que la relation $\equiv$ définie en cours ?

L'ensemble des états est restreint. Dans la définition donnée en cours, on considère l'ensemble de tous les états alors que dans cette nouvelle définition on ne prend en compte que les états de S.

Question 6

Donnez l'exemple de deux programmes qui ne sont pas équivalents mais qui sont S-équivalents pour un S que vous caractériserez.

On peut prendre comme exemple les deux programmes C_1 et C_2 définis dans l'exercice. On prend pour S l'ensemble $\{\sigma | (\sigma(x) < 0 \ et \ \sigma(y) = 5) \ ou \ \sigma(x) \geq 0\}$.

2. Extension du langage avec l'instruction `break`

On ajoute la commande break au langage impératif étudié en cours :

```
c := ...   | break
```

La commande `break` a pour effet de causer la terminaison abrupte du programme en cours d'exécution. L'état final est alors celui dans lequel le `break` a été exécuté.

Dans la sémantique formelle, afin de distinguer terminaison normale et terminaison abrupte, on doit modifier le jugement d'exécution : il prend désormais la forme $\langle \sigma_1, c \rangle \to o, \sigma_2$ où σ_1 et σ_2 sont des états (valuations), c une commande et o un indicateur de terminaison valant N si la terminaison est normale, B si la terminaison a été causée par l'exécution de `break`.

Le jugement d'évaluation des expressions reste identique et a la forme $< e, \sigma > \rightsquigarrow v$ où e est une expression, σ une valuation et v une valeur entière ou booléenne.

Question 7

Définir la sémantique opérationnelle à grands pas du langage étendu.

Les nouvelles règles

$$(C_0)\ \frac{}{\langle \mathsf{break}, \sigma\rangle \to B, \sigma}$$

$$(C_1)\ \frac{}{\langle \mathsf{skip}, \sigma\rangle \to N, \sigma} \qquad (C_2)\ \frac{\langle a, \sigma\rangle \rightsquigarrow n}{\langle x := a, \sigma\rangle \to N, \sigma[x \leftarrow n]} \quad n \in \mathbb{Z}$$

$$(C_3)\ \frac{\langle c_1, \sigma\rangle \to N, \sigma_1 \quad \langle c_2, \sigma_1\rangle \to o, \sigma_2}{\langle c_1 ; c_2, \sigma\rangle \to o, \sigma_2} \qquad (C'_3)\ \frac{\langle c_1, \sigma\rangle \to B, \sigma_1}{\langle c_1 ; c_2, \sigma\rangle \to B, \sigma_1}$$

$$(C_4)\ \frac{\langle b, \sigma\rangle \rightsquigarrow \mathsf{true} \quad \langle c_1, \sigma\rangle \to o, \sigma_1}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \to o, \sigma_1} \qquad (C_5)\ \frac{\langle b, \sigma\rangle \rightsquigarrow \mathsf{false} \quad \langle c_2, \sigma\rangle \to o, \sigma_2}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \to o, \sigma_2}$$

$$(C_6)\ \frac{\langle b, \sigma\rangle \rightsquigarrow \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \to N, \sigma}$$

$$(C_7)\ \frac{\langle b, \sigma\rangle \rightsquigarrow \mathsf{true} \quad \langle c, \sigma\rangle \to N, \sigma_1 \quad \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma_1\rangle \to o, \sigma_2}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \to o, \sigma_2} \qquad (C'_7)\ \frac{\langle b, \sigma\rangle \rightsquigarrow \mathsf{true} \quad \langle c, \sigma\rangle \to B, \sigma_1}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \to B, \sigma_1}$$

Question 8

On considère le programme suivant C

```
x := 2;
while 0 ≤ x do
   if x = 1 then x := 2 ; break; x := 0
   else x := x - 1
   fi
od
```

Soit σ un état quelconque. Caractérisez σ' et o de manière à ce que l'on ait $\langle \sigma, C\rangle \to o, \sigma'$. Vous dessinerez l'arbre de dérivation le plus clairement possible (avec sa conclusion en bas et les noms des règles clairement mentionnés). Vous ne développerez pas les sous-arbres correspondant à l'évaluation des expressions arithmétiques et booléennes.

On termine de manière anormale (B) dans l'état qui associe x à la valeur 1. Il fallait dessiner l'arbre de dérivation. Attention dans cet arbre de dérivation ne laissez pas des variables non définies. Un o par exemple : il doit valoir B ou N.

Exercice 3 (**Le choix non déterministe**)

On étend le langage impératif du cours avec une commande de choix non déterministe notée c_1 [] c_2 (cette commande a été rencontrée en B).

```
c := ... | c [] c
```

La sémantique informelle de la commande c_1 [] c_2 exprime que l'on exécute de manière non déterministe c_1 ou c_2 (qui peuvent être des commandes complexes).

Question 9

Étendre la sémantique à grands pas du langage de manière à prendre en compte cette nouvelle commande.

On ajoute les deux règles suivantes :

$$(D_1)\ \frac{\langle c_1, \sigma\rangle \to \sigma_1}{\langle c_1 \ [] \ c_2, \sigma\rangle \to \sigma_1}$$

$$(D_2)\ \frac{\langle c_2, \sigma\rangle \to \sigma_1}{\langle c_1 \ [] \ c_2, \sigma\rangle \to \sigma_1}$$

Question 10

Étendre la sémantique à petits pas du langage de manière à prendre en compte cette nouvelle commande.

On ajoute les deux règles suivantes :

$$(D_1)\ \frac{}{\langle c_1 \ [] \ c_2, \sigma\rangle \hookrightarrow \langle c_1, \sigma\rangle}$$

$$(D_2)\ \frac{}{\langle c_1 \ [] \ c_2, \sigma\rangle \hookrightarrow \langle c_2, \sigma\rangle}$$

Question 11

Étendre la sémantique axiomatique du langage (les règles de Hoare) de manière à prendre en compte cette nouvelle commande. Étendre de même le générateur de conditions de vérification (la fonction VC).

La nouvelle règle : Puisque l'exécution peut suivre le chemin i_1 ou i_2, les deux chemins doivent assurer le contrat.

$$\frac{\{P\}\ i_1\ \{Q\} \quad \{P\}\ i_2\ \{Q\}}{\{P\}\ i_1\ []\ i_2\ \{Q\}}$$

Et la fonction VC est complétée avec la ligne suivante : $VC(i_1\ []\ i_2, Q) = VC(i_1, Q) \wedge VC(i_2, Q)$.

Exercice 4 (**Langage fonctionnel**)

On considère le langage fonctionnel suivant λ^I :

Syntaxe : les expressions sont des expressions formées à partir de constantes entières, d'additions, de variables, d'abstractions et d'applications :

$$e ::= n | \lambda x.e | x | e\ e | e + e'$$

Sémantique : voir les règles rappelées à la figure 1.

Types : les types sont formés à partir du type de base int, des variables de type et du constructeur $\rightarrow$:

$$\tau ::= int | \alpha | \tau \rightarrow \tau$$

Règles de typage : on considère un typage monomorphe. Les règles de typage sont données à la figure 2.

• On appelle λ^E le langage λ^I augmenté de la séquence de 2 expressions : si e et e' sont deux expressions de λ^E alors $e; e'$ est une expression de λ^E.

La sémantique de la séquence est formalisée par la nouvelle règle (SEQ) suivante. Elle exprime qu'évaluer la séquence $e; e'$ consiste à évaluer l'expression e puis évaluer l'expression e'. La valeur de l'expression e est *jetée*.

$$\boxed{(\text{SEQ})\ \frac{< e, \sigma > \rightsquigarrow v \quad < e', \sigma > \rightsquigarrow v'}{< e; e', \sigma > \rightsquigarrow v'}}$$

La règle de typage de la séquence est la suivante :

$$\boxed{(\text{SEQ})\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e; e' : \tau'}}$$

On veut montrer que la séquence peut se définir avec les autres contructions de λ^I. La séquence $e; e'$ peut se définir comme $(\lambda x.e')\ e$ avec x une variable non libre de e' . Par exemple $3; 4$ peut se traduire par $(\lambda x.4)\ 3$.

Question 12

Donner la traduction de l'expression $x + 1; x$.

$(\lambda y.x)\ (x + 1)$

Question 13

Montrer formellement que l'évaluation de l'expression obtenue conduit bien au même résultat que la séquence initiale, dans une valuation σ telle que $\sigma(x) = 0$.

Il suffit de montrer en utilisant les règles de la sémantique que les deux expressions s'évaluent en la même valeur, à savoir 0. Pour le démontrer formellement, on dessine les deux arbres de dérivation.

Evaluation de $x + 1; x$

$$\dfrac{\dfrac{\dfrac{}{< x, \sigma >\rightsquigarrow 0} \quad < 1, \sigma >\rightsquigarrow 1}{< x + 1, \sigma >\rightsquigarrow 1} \quad \dfrac{}{< x, \sigma >\rightsquigarrow 0}}{< x + 1; x, \sigma >\rightsquigarrow 0}$$

Evaluation de $(\lambda y.x)\ (x + 1)$

$$\dfrac{\dfrac{}{< \lambda y.x, \sigma >\rightsquigarrow << \lambda\ y.x, \sigma >>} \quad \dfrac{\dfrac{}{< x, \sigma >\rightsquigarrow 0} \quad \dfrac{}{< 1, \sigma >\rightsquigarrow 1}}{< x + 1, \sigma >\rightsquigarrow 1} \quad \dfrac{}{< x, \sigma[y \leftarrow 1] >\rightsquigarrow 0}}{< (\lambda y.x)\ (x + 1), \sigma >\rightsquigarrow 0}$$

Question 14

Montrer que le type de l'expression obtenue est le même que celui de la séquence initiale, pour un contexte Γ tel que $\Gamma(x) = int$. Vous dessinerez l'arbre de dérivation.

On démontre en dessinant les deux arbres de dérivation que les deux expressions ont le type int.

Typage de $x + 1; x$

$$\dfrac{\dfrac{\dfrac{x : int \in \Gamma}{\Gamma \vdash x : int} \quad \Gamma \vdash 1 : int}{\Gamma \vdash x + 1 : int} \quad \dfrac{}{\Gamma \vdash x : int}}{\Gamma \vdash x + 1; x : int}$$

Typage de $(\lambda y.x)\ (x + 1)$

$$\dfrac{\dfrac{\dfrac{x : int \in (\Gamma \oplus y : int)}{\Gamma \oplus y : int \vdash x : int}}{\Gamma \vdash \lambda y.x : int \rightarrow int} \quad \dfrac{\dfrac{x : int \in \Gamma}{\Gamma \vdash x : int} \quad \dfrac{}{\Gamma \vdash 1 : int}}{\Gamma \vdash x + 1 : int}}{\Gamma \vdash (\lambda y.x)\ (x + 1) : int}$$

$$(\text{CONST}) \quad < n, \sigma > \rightsquigarrow n \qquad\qquad (\text{ID}) \quad < x, \sigma > \rightsquigarrow \sigma(x)$$

$$(\text{ABS}) \quad < \lambda\ x.e, \sigma > \rightsquigarrow << \lambda\ x.e, \sigma >>$$

$$(\text{ADD}) \quad \frac{< e, \sigma > \rightsquigarrow v \qquad < e', \sigma > \rightsquigarrow v' \qquad v'' = v +_N v'}{< e + e', \sigma > \rightsquigarrow v''}$$

$$(\text{APP}) \quad \frac{< e, \sigma > \rightsquigarrow << \lambda\ x.e_f, \sigma_f >> \qquad < e', \sigma > \rightsquigarrow v \qquad < e_f, \sigma_f[x \leftarrow v] > \rightsquigarrow v'}{< e\ e', \sigma > \rightsquigarrow v'}$$

Figure 1: Sémantique opérationnelle de λ^I

Question 15

Donner la traduction de l'expression $((\lambda y.y + 1)\ x); x$.

Par exemple $(\lambda z.x)((\lambda y.y + 1)\ x)$ ou encore $(\lambda y.x)((\lambda y.y + 1)\ x)$.

Question 16

Définir la fonction *trad* de traduction d'une expression de λ^E dans λ^I. La fonction *trad* doit transformer toutes les séquences selon le schéma donné.

La fonction se définit par induction sur l'expression à traduire. Ainsi :
$trad(x) = x$
$trad(n) = n$
$trad(e_1 + e_2) = trad(e_1) + trad(e_2)$
$trad(e_1\ e_2) = trad(e_1)\ trad(e_2)$
$trad(e_1; e_2) = (\lambda y.trad(e_2))\ trad(e_1)$ avec y variable non libre dans e_2

Question 17

Énoncer formellement le théorème qui établit que si une expression e de λ^E s'évalue en une valeur alors sa traduction dans λ^I s'évalue en la même valeur.

Indiquer comment démontrer ce théorème.

Le théorème à démontrer par induction sur e :

$$\forall e, v, \sigma, < e, \sigma > \rightsquigarrow v \Rightarrow < trad(e), tradenv(\sigma) > \rightsquigarrow tradval(v)$$

Il faut aussi traduire la valeur v car celle-ci peut être une fermeture et contenir des séquences. De même pour l'environnement. La fonction *tradval* se définit ainsi :
$tradval(n) = n$
$tradval(<< \lambda\ x.e, \sigma >>) = << \lambda\ x.trad(e), tradenv(\sigma) >>$
avec
$tradenv(\emptyset) = \emptyset$
$tradenv(x : v; \sigma) = x : tradval(v); tradenv(\sigma)$

$$(\textsc{const})\ \ \Gamma \vdash n : int \qquad\qquad (\textsc{id})\ \ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$(\textsc{abs})\ \ \frac{\Gamma \oplus x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\, x.e : \tau \to \tau'} \qquad (\textsc{add})\ \ \frac{\Gamma \vdash e : int \quad \Gamma \vdash e' : int}{\Gamma \vdash e + e' : int}$$

$$(\textsc{app})\ \ \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

Figure 2: typage de λ^I