Langages, Traducteurs et Sémantique, M1, 2006/2007
J.-C. Fernandez, Y. Lakhnech, L. Gonnord, F. Lagnier, C. Ené

LTS

Les exercices sont indépendants. Le barème n'est donné qu'à titre indicatif. Tous documents autorisés.

Exercice I (5 points) - Sémantique opérationnelle naturelle et axiomatique

Introduction d'une construction `case`

Partie A : Sémantique naturelle

On considère le langage `while` vu en cours. On étend les commandes en ajoutant la construction `case` :

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \mid \\ & & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\ & & \texttt{while } b \texttt{ do} S \mid \texttt{case } a \texttt{ of } LS \texttt{ endcase} \\ LS & ::= & n : S \mid n : S, LS \end{array}$$

Ainsi la construction `case` est constituée d'une expression arithmétique a et d'une liste de commandes étiquetées $n : S$ (avec $n \in \mathbb{N}$), le séparateur de liste est le symbole de ponctuation ",".

Une construction `case` est bien formée si toutes les étiquettes sont distinctes.

Question 1

On veut définir une fonction B_D qui, appliquée à une liste de commandes étiquetées, vérifie que toutes les étiquettes sont distinctes. Cette fonction prend une liste de commandes étiquetées en paramètre et rend l'ensemble des étiquettes en vérifiant la contrainte précédente. Dans le cas où il y a une double définition, l'ensemble retourné est {`erreur`}. Compléter la définition suivante de la fonction :

$$\begin{array}{lcl} B_D(n : S) & = & \{n\} \\ B_D(n : S, LS) & = & \cdots \end{array}$$

Dans ce qui suit, on ne considère que des constructions `case` bien formée.

La sémantique en français de la construction `case` a `of` LS `endcase` est donnée de la manière suivante :
– l'expression a doit délivrer un entier que l'on note v,
– lorsque cette valeur v correspond à une des étiquettes n_i, la commande associée S_i est exécutée et le résultat de cette exécution est le résultat de la construction `case` ; sinon, si aucune des étiquettes ne correspond à v, alors la construction `case` se comporte comme `skip`.
Dans l'exemple suivant,

```
x:= 3 ;
y :=2 ;
case x-y of
1 : x := x+y,
0 : x := 2,
3 : y := 0
endcase
```

c'est la commande étiquetée par 1 qui sera exécutée. Le résultat du programme sera $[x \mapsto 5, y \mapsto 2]$.

Question 2

Donner des règles de sémantique naturelle pour la construction case.

Question 3

Appliquer ces règles à l'exemple ci-dessus.

Partie B : Sémantique axiomatique

Question 4

Donner une règle de Hoare correcte pour la construction case bien formée, de la forme :

$$\frac{\bigwedge_{i=1}^{k} \{\cdots\} S_i \{\cdots\} \qquad \bigwedge_{i=1}^{k} a \neq n_i \Longrightarrow \cdots}{\{P\}\texttt{case}\ a\ \texttt{of}\ n_1 : S_1, \cdots n_k : S_k\ \texttt{endcase}\{Q\}}$$

Exercice II : Types (5 points)

Dans cet exercice, on s'intéresse au langage `while` vu en cours. L'idée est de considérer comme un système de types la règle des signes. Celle-ci indique par exemple que la somme de deux entiers positifs est un entier positif, que le produit de deux entiers négatifs est positif, etc. Par contre, la différence de deux entiers positifs est un entier. Nous rappelons tout d'abord la syntaxe du langage vu en cours, légèrement modifiée :

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \mid \\ & & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & & \texttt{while}\ b\ \texttt{do} S \\ a & := & p \mid n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\ b & := & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \end{array}$$

Nous avons donc trois catégories syntaxiques $Aexp$, $BExp$ et Stm respectivement pour les expressions arithmétiques, les expressions booléennes et les commandes.

La modification consiste à avoir remplacé une dénotation d'entier par une dénotation d'entier positif ou nul, notée p et une dénotation d'entier strictement négatif, notée n. On va définir 3 types `Pos`, `Neg`, `Int` qui désignent respectivement les entiers positifs ou nul, les entiers strictement négatifs et les entiers relatifs. Nous avons un ordre naturel sur les types `Pos` $<$ `Int` et `Neg` $<$ `Int`. On peut définir la borne supérieure $\sqcup$: $x \sqcup y = y$ ssi $x < y$ ou $x = y$.

On considère l'environnement de types qui à chaque variable associe un type :

$$\texttt{Env} = \texttt{Var} \longrightarrow \{\texttt{Pos}, \texttt{Neg}, \texttt{Int}\}.$$

La relation d'ordre et la borne supérieure sont étendues à Env de la manière suivante :

$(\Gamma \sqcup \Gamma')(x) = \Gamma(x) \sqcup \Gamma'(x)$

Les configurations sont données par
– $Env \times Aexp \cup \{\texttt{Pos}, \texttt{Neg}, \texttt{Int}\}$ pour les expressions arithmétiques,
– $Env \times Stm \cup Env$ pour les commandes,
– $Env \times Bexp \cup \{ok\}$ pour les expressions booléennes.
On note Γ un élément de l'environnement et $\tau, \tau_1, ...$ un type.

Partie A : Expressions arithmétiques

On donne ci-dessous les règles pour les constantes, les variables et l'addition.

$$(\Gamma, p) \longrightarrow \texttt{Pos} \qquad (\Gamma, n) \longrightarrow \texttt{Neg} \qquad (\Gamma, x) \longrightarrow \Gamma(x)$$

$$\frac{(\Gamma, a_1) \longrightarrow \texttt{Pos} \quad (\Gamma, a_2) \longrightarrow \texttt{Pos}}{(\Gamma, a_1 + a_2) \longrightarrow \texttt{Pos}} \qquad \frac{(\Gamma, a_1) \longrightarrow \texttt{Neg} \quad (\Gamma, a_2) \longrightarrow \texttt{Neg}}{(\Gamma, a_1 + a_2) \longrightarrow \texttt{Neg}}$$

$$\frac{(\Gamma, a_1) \longrightarrow \texttt{Neg} \quad (\Gamma, a_2) \longrightarrow \texttt{Pos}}{(\Gamma, a_1 + a_2) \longrightarrow \texttt{Int}}$$

On constate, à la lecture de ces règles que l'addition de deux entiers positifs (respectivement négatifs) a pour résultat un entier positif (respectivement négatif) mais que l'addition d'un positif et d'un négatif est un entier (cela traduit en quelque sorte une perte d'information).

Question 1

Donner les règles pour la soustraction et la multiplication.

Partie B : Commandes

On donne les règles pour l'affectation, la commande skip et l'itération.

$$\frac{(\Gamma, a) \longrightarrow \tau}{(\Gamma, x := a) \longrightarrow \Gamma[x \mapsto \tau]} \qquad (\Gamma, \mathit{skip}) \longrightarrow \Gamma \qquad \frac{(\Gamma, S) \longrightarrow \Gamma'}{(\Gamma, \texttt{while}\ b\ \texttt{do}\ S) \longrightarrow \Gamma \sqcup \Gamma'}$$

Question 2

Donner les règles pour la composition séquentielle et pour la conditionnelle.

Partie C : Preuve*

Cette partie difficile, n'est pas incluse directement dans le barème et est susceptible de donner des points supplémentaires. On veut montrer que, étant donné un programme S, le calcul de Γ, i.e. $(\Gamma, S) \longrightarrow \Gamma'$ donne une information qui est une approximation de la sémantique opérationnelle naturelle dans un sens que l'on va préciser.

Etant donné un environnement de types Γ et une mémoire σ, on définit la correspondance suivante :

$$(\Gamma, \sigma) \in \mathcal{R} \text{ ssi } \forall x \cdot [(\Gamma(x) = \texttt{Pos} \wedge \sigma(x) \geq 0) \vee (\Gamma(x) = \texttt{Neg} \wedge \sigma(x) < 0) \vee \Gamma(x) = \texttt{Int}]$$

Question 1

Monter que pour tout $(\Gamma, \sigma) \in \mathcal{R}$,

si $(\Gamma, a) \longrightarrow \texttt{Pos}$ alors $\mathcal{A}[a]\sigma \geq 0$ et si $(\Gamma, a) \longrightarrow \texttt{Neg}$ alors $\mathcal{A}[a]\sigma < 0$

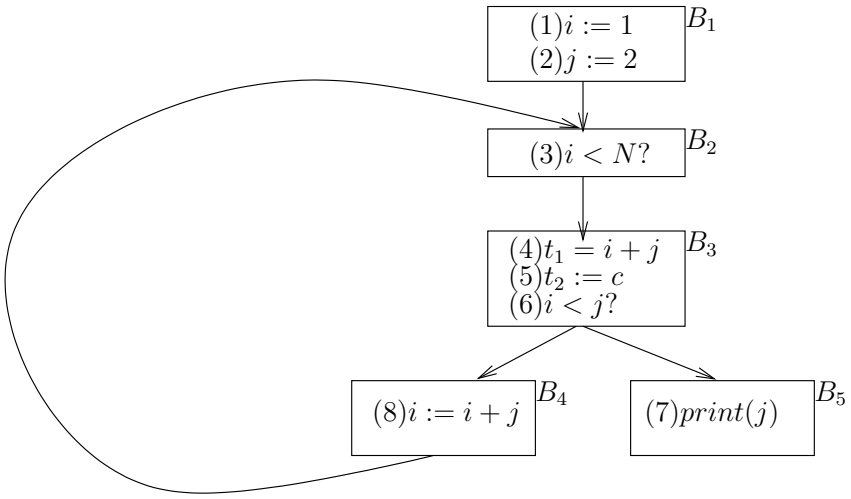
Question 2

Montrer, par induction que si

$$(\Gamma, \sigma) \in \mathcal{R} \text{ et } (S, \sigma) \longrightarrow \sigma' \quad \text{ alors } \exists \Gamma' \cdot (\Gamma, S) \longrightarrow \Gamma' \text{ et } (\Gamma', \sigma') \in \mathcal{R}$$

Exercice III : Optimisation (5 points)

On considère le graphe de flot de contrôle suivant :



Variables actives

1. Calculer les ensembles $Gen(b)$ et $Kill(b)$ pour chaque bloc de base b.
2. Calculer les ensembles $In(b)$ et $Out(b)$ pour chaque bloc de base b.
3. Suppression des instructions inutiles qui sont les affectations $x := e$ telles que x est inactive en fin de bloc et n'est pas utilisée dans le bloc après l'instruction.

Exercice IV : Génération de code (5 points)

On considère d'une part le langage d'assemblage vu en cours et, d'autre part, le programme suivant :

```
main() {
  int x ;
  int P1() {
    int y ;
    void Q2 () {
      int z ;
      z=2;
      x = y+x+z;
    }
    void P2() {
      y=3;
      Q2 ();
    } /*  end P2 */
    P2();
    return (x+10);
  } /* end P1 */
  x=2+ P1();
}/* end main */
```

1. Dessiner la pile lors de l'exécution de `Q2`.
2. Dans la procédure `P2`, donner la séquence d'instructions associée à `y=3`.
3. Dans la procédure `P2`, donner la séquence d'instructions associée à l'appel de `Q2`.
4. Dans la procédure `Q2`, donner la séquence d'instructions associée à `x=y+x+z`.
5. Dans la procédure `P1`, donner la séquence d'instructions associée à `return(x+10)`.
6. Dans la procédure `main`, donner la séquence d'instructions associée à `x=2+ P1()`.