

Exercice I : Sémantique opérationnelle : extension du langage `while` avec des exceptions

On considère dans cet exercice la liaison dynamique pour les variables.

Introduction de l'instruction `break`

Nous rajoutons à la grammaire du langage `while` vue en cours sans les procédures mais avec les blocs, l'instruction `break` en étendant la syntaxe des commandes $S ::= \dots \mid \text{break}$. Cette instruction provoque l'arrêt de l'exécution de la boucle immédiatement englobante ou, à défaut du bloc immédiatement englobant. Le programme s'exécute ensuite normalement, le contrôle passant juste après l'instruction interrompue. Les configurations terminales sont alors $\mathbf{State} \cup \{\text{stop}_b\} \times \mathbf{State}$ (où $\mathbf{State} = \mathbf{Var} \rightarrow \mathbb{Z}$).

On rajoute, pour la construction `break`, la règle suivante : $\boxed{(\text{break}, \sigma) \longrightarrow (\text{stop}_b, \sigma)}$

Exemple

Considérons le fragment de programme `S` suivant :

```
begin var x := 2 ; while (x<10) do if x = 5 then break else skip;
      x := x+1 od ; x:=0;
```

A partir d'une configuration initiale (S, σ) , l'exécution de la boucle s'arrête dans la configuration $(\text{stop}_b, \sigma[x \mapsto 5])$, ce qui conduit à la configuration terminale pour `S`, $\sigma[x \mapsto 0]$.

Question 1 Modifier la règle de sémantique vue en cours, de la composition séquentielle en complétant les règles suivantes :

$$\frac{(S_1, \sigma) \longrightarrow (\text{stop}_b, \sigma')}{(S_1; S_2, \sigma) \longrightarrow \dots} \qquad \frac{(S_1, \sigma) \longrightarrow \sigma' \quad \dots}{(S_1; S_2, \sigma) \longrightarrow \dots}$$

Question 2 Modifier la règle de sémantique vue en cours, pour la commande `while`, sachant que, si le corps de la boucle contient une instruction `break`, son exécution est interrompue ; l'exécution continue normalement après l'instruction `while`.

Question 3 Modifier la règle de sémantique vue en cours pour le bloc, sachant que si la partie commande contient une instruction `break`, son exécution est interrompue ; l'exécution continue normalement après le bloc.

Introduction de la commande `exit`

Nous rajoutons à la grammaire ci-dessus l'instruction `exit` en étendant la syntaxe des commandes $S ::= \dots \mid \text{exit}$. Cette instruction provoque l'arrêt du programme en cours d'exécution. Les nouvelles configurations terminales sont alors

$\mathbf{State} \cup \{\text{stop}_b\} \times \mathbf{State} \cup \{\text{stop}_e\} \times \mathbf{State}$. On rajoute, pour la construction `exit`, la règle suivante :

$$\boxed{(\text{exit}, \sigma) \longrightarrow (\text{stop}_e, \sigma)}$$

Si on reprend l'exemple 1.1.1 en remplaçant l'instruction `break` par la construction `exit`, à partir de la configuration initiale (S, σ) , l'exécution de la boucle s'arrête dans la configuration $(\text{stop}_e, \sigma[x \mapsto 5])$, ce qui conduit à la configuration terminale pour S , $(\text{stop}_e, \sigma[x \mapsto 5])$.

Question 4 Modifier les règles vues en cours pour la composition séquentielle, le bloc et la commande `if`.

Introduction de la commande `raise`

Nous rajoutons la possibilité de déclarer des exceptions et de traiter les exceptions nommées. Une exception est définie dans une déclaration d'exceptions qui lui donne un nom. On rajoute, avant les déclarations de variables, les déclarations d'exceptions. Lors de l'exécution d'un programme, une exception peut être levée explicitement par l'instruction `raise` ou implicitement, par exemple lors d'une division par 0. Dans cet exercice, nous ne considérerons pas les exceptions implicites. Quand une exception est levée, le contrôle est transféré à un traitement d'exception fourni à la fin d'un bloc. Ce traitement termine l'exécution du bloc.

Nous modifions la syntaxe des blocs en rajoutant des déclarations d'exceptions, et des traitements d'exception. La syntaxe d'un bloc devient : $S ::= \text{begin } D_E; D_V; S; T \text{ end}$

Les déclarations d'exceptions se font de manière analogue aux déclarations de variables :

$$D_E ::= \text{exception } e; D_E \mid \epsilon$$

Un exception est provoquée par la commande `raise e`.

Le traitement d'exception, à la fin du bloc, prend la forme suivante :

$$T ::= \text{when } e \implies S; T \mid \epsilon.$$

La construction `when` concerne les exceptions explicites.

La syntaxe des commandes devient alors :

$$S ::= x := a \mid \text{skip} \mid S; S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{while } b \text{ do } S \text{ od} \mid \text{begin } D_E; D_V; S; T \text{ end} \mid \text{break} \mid \text{exit} \mid \text{raise } e$$

Exemple d'utilisation

```
begin
  exception trouve ;
  var x :=0 ;
  var res :=0 ;
```

```

x := 0 ;
while true do
  if x = 5 then raise trouve else skip
  x := x+1 ;
od
when trouve => res := 5 ; x:=-1
end

```

A partir d'une configuration initiale (S, σ, τ) , (on verra à la question 7 la forme des configurations et la signification de τ). l'exécution de la boucle s'arrête dans la configuration $(\mathbf{stop}_r, \sigma[x \mapsto 5], \tau[\text{trouve} \mapsto 1])$, ce qui conduit d'abord à la configuration $(\mathbf{stop}_r, \sigma[x \mapsto 5], \tau[\text{trouve} \mapsto 1])$ avant le traitement de l'exception, puis à la configuration terminale pour $S, \sigma[x \mapsto -1, \text{res} \mapsto 5], \tau[\text{trouve} \mapsto 1]$.

Question 5 On rappelle que $DV(D_V)$ dénote l'ensemble des variables déclarées dans D_V . De manière similaire, $DE(D_E)$ dénote l'ensemble des exceptions déclarées dans D_E . On suppose qu'une exception ne peut être levée et traitée que dans le bloc où elle est déclarée et pas dans les blocs internes. Exprimer de manière formelle, en donnant des règles de sémantique statique, le fait que tout nom d'exception e dans la construction **when** $e \dots$ ou dans la construction **raise** e est déclaré dans la partie D_E .

Question 6 Reprendre la question précédente en supposant qu'une exception peut être levée et traitée dans un bloc imbriqué.

Question 7 Dans cette question, nous nous intéressons à la sémantique opérationnelle de **raise** et d'un bloc. Pour cela, nous modifions les configurations de la manière suivante ;

$\Gamma = (\mathbf{Stm} \times \mathbf{State} \times \mathbf{Except}) \cup \mathbf{State} \cup \{\mathbf{stop}_b\} \times \mathbf{State} \cup \{\mathbf{stop}_e\} \times \mathbf{State} \cup \{\mathbf{stop}_r\} \times \mathbf{State} \times \mathbf{Except}$

L'ensemble $\mathbf{Except} = \text{NomE} \xrightarrow{\text{part.}} \{0, 1\}$ est l'ensemble des applications partielles des noms d'exception NomE vers l'ensemble $\{0, 1\}$.

Dans une configuration (S, σ, τ) , $\tau(e) = 1$ si e a été déclaré et si une instruction **raise** e est apparue. La règle de sémantique pour la commande **raise** est définie de la manière suivante ;

$\boxed{(\mathbf{raise} \ e, \sigma, \tau) \longrightarrow (\mathbf{stop}_r, \sigma, \tau[e \mapsto 1])}$

Modifier la règle de sémantique opérationnelle vue en cours pour les blocs.

Exercice 2 - Sémantique opérationnelle naturelle et typage

Extension du langage while avec la notion de tableau

On considère la syntaxe abstraite d'un fragment du langage **while** vu en cours en ne différenciant pas les expressions arithmétiques (qui étaient notées **a** dans le cours) des expressions booléennes (qui étaient notées **b** dans le cours) et en l'étendant avec les tableaux. Ici, les deux sortes d'expression sont notées **e**.

Par ailleurs, contrairement à ce que l'on a vu en cours, les déclarations de variables ne comportent pas d'initialisation et sont typées.

$$\begin{aligned}
 D_V & ::= \text{var } x : t; D_V \mid \text{var } x[n] : t; D_V \mid \epsilon \\
 S & ::= x := e \mid x[e] := e \mid \text{skip} \mid S_1; S_2 \mid \\
 & \quad \text{if } e \text{ then } S_1 \text{ else } S_2 \\
 & \quad \text{while } e \text{ do } S \mid \text{begin } D_V \text{ Send} \\
 e & ::= n \mid x \mid \text{true} \mid e + e \mid e = e \mid e \text{ and } e \mid x[e] \\
 t & ::= \text{Entier} \mid \text{Bool}
 \end{aligned}$$

Partie A : Typage

On définit l'ensemble `Type` de la manière suivante :

$$\begin{aligned}
 \text{TypeBase} & = \text{Entier}, \text{Bool} \\
 \text{Type} & = \text{TypeBase} \cup \{\text{Tab}\} \times \mathbb{N} \times \text{Type}
 \end{aligned}$$

Intuitivement, un type est soit un type de base, soit un tableau de n éléments d'un même type.

Lors d'une déclaration de tableau `var x[n] : t`, on associe au nom de variable `x` un triplet $(\text{Tab}, \mathcal{N}(n), t)$, indiquant que la variable `x` est un tableau de $\mathcal{N}(n)$ éléments de type t ; dans ce cas, les éléments du tableau sont $x[0], \dots, x[\mathcal{N}(n) - 1]$.

On rappelle que, pour le typage, l'environnement est une fonction qui à un nom de variable associe un type. L'environnement est construit en analysant les déclarations, utilisé pour la vérification de type dans les expressions, et utilisé pour montrer qu'une commande est bien typée.

Question 1

Donnez formellement la règle qui permet de modifier l'environnement lors de la déclaration `var x[n] : t`.

Question 2

Donnez un exemple de programme incorrect vis-à-vis de cette règle.

Question 3

Proposez en français une règle décrivant l'accès à un élément de tableau `x[e]`. Donnez formellement la règle de typage correspondante.

Question 4

Donnez un exemple de programme incorrect vis-à-vis de cette règle.

Question 5

Proposez en français une règle décrivant l'affectation `x[e1] := e2`. Donnez formellement la règle de typage correspondante.

Question 6

Donnez un exemple de programme incorrect vis-à-vis de cette règle.

Partie B : Sémantique naturelle

Dans cette partie, on considère la sémantique décrivant la liaison statique pour les variables : L'environnement Env décrit une association entre un nom de variable et un emplacement mémoire, i.e. Loc , la mémoire Sto associe à un emplacement mémoire son contenu :

$$\begin{aligned} Env &= \text{Nom} \mapsto \text{Loc} \\ Sto &= \text{Loc} \mapsto \mathbb{Z} \oplus \mathbb{B} \end{aligned}$$

Comme vu en cours, ces fonctions sont partielles et l'ensemble des adresses des emplacements mémoire est assimilé à \mathbb{N} . On considère que le domaine d'un élément $sto \in \text{Loc} \mapsto \mathbb{Z} \oplus \mathbb{B}$ est un intervalle. On considère la fonction $\mathbf{new} : \text{Sto} \times \mathbb{N} \mapsto \text{Sto} \times \mathbb{N}$.

$$\begin{aligned} \mathbf{new}(sto_1, k_2) &= (sto_2, k_1) \text{ tel que} \\ \text{Dom}(sto_1) &= [0 \cdots k_1 - 1] \\ \text{Dom}(sto_2) &= [0 \cdots k_1 + k_2 - 1] \end{aligned}$$

Autrement dit, si k_1 est le plus petit entier tel que $sto_1(k_1)$ n'est pas définie, la fonction \mathbf{new} alloue k_2 emplacements mémoires contigus à partir de l'adresse k_1 .

Par ailleurs, on peut utiliser la fonction \mathbf{size} qui à un type associe la taille d'un élément de ce type.

Lors de la déclaration de tableau $\mathbf{var} x[n] : t$, on alloue n emplacements contigus de taille $\mathbf{size}(t)$. On suppose ici que l'information sur les types, i.e. la fonction Γ qui à chaque variable associe son type est connue (cf partie A).

Question 1

Proposez une règle de sémantique naturelle pour la construction $\mathbf{var} x : t; D_V$.

Question 2

Proposez une règle de sémantique naturelle pour la construction $\mathbf{var} x[n] : t; D_V$.

Question 3

Proposez une règle de sémantique naturelle pour la construction $\mathbf{x}[e_1] := e_2$

Exercice 3 - Correction de la fonction de génération de code intermédiaire

Code 3 adresses : syntaxe

- Variables : $x \in \text{Var}$
- Temporaires : $t, t_i \in \text{Tmp}$
- Noms α une variable ou un temporaire
- Expressions **Exp**
 - $e \in \text{Exp}$
 - $e ::= n \mid \alpha \mid \alpha \text{ op } \alpha \quad \text{op} \in \{+, -, *, /\}$
- Instructions **Inst**
 - $i \in \text{Inst}$
 - $i ::= t := e \mid x := e \mid \text{if } \alpha \text{ oprel } \alpha \text{ goto } l \mid \text{goto } l \quad \text{oprel} \in \{=, <, >, \leq, \geq\}$

Configurations

Les configurations sont des quadruplets (pc, c, σ, τ) où pc, c, σ, τ désignent respectivement le compteur programme, le code, la mémoire et la valeur de chaque temporaire :

c	$\mathbb{N} \mapsto \text{Inst}$	le code
σ	$\text{Var} \mapsto \mathbb{Z}$	la mémoire
τ	$\text{Tmp} \mapsto \mathbb{Z}$	la mémoire temporaire

Sémantique des expressions

On définit la sémantique des expressions de la manière suivante :

$$\begin{aligned}
 \mathcal{E}[n]_{(\sigma, \tau)} &= \mathcal{N}[n] \\
 \mathcal{E}[x]_{(\sigma, \tau)} &= \sigma(x) \\
 \mathcal{E}[t]_{(\sigma, \tau)} &= \tau(t) \\
 \mathcal{E}[e_1 \text{ op } e_2]_{(\sigma, \tau)} &= \mathcal{E}[e_1]_{(\sigma, \tau)} \text{ op } \mathcal{E}[e_2]_{(\sigma, \tau)}
 \end{aligned}$$

Sémantique des instructions

$$\begin{aligned}
 (pc, c, \sigma, \tau) \triangleright (pc + 1, c, \sigma, \tau[t \mapsto \mathcal{E}[e]_{(\sigma, \tau)}]) & \text{ si } c(pc) = t := e \\
 (pc, c, \sigma, \tau) \triangleright (pc + 1, c, \sigma[x \mapsto \mathcal{E}[e]_{(\sigma, \tau)}], \tau) & \text{ si } c(pc) = x := e \\
 (pc, c, \sigma, \tau) \triangleright (l, c, \sigma, \tau) & \text{ si } c(pc) = \text{goto } l \text{ et} \\
 (pc, c, \sigma, \tau) \triangleright (pc + 1, c, \sigma, \tau) & \text{ si } c(pc) = \text{if } \alpha \text{ oprel } \alpha' \text{ goto } l \text{ et } \neg(\mathcal{E}[\alpha]_{(\sigma, \tau)} \text{ oprel } \mathcal{E}[\alpha']_{(\sigma, \tau)}) \\
 (pc, c, \sigma, \tau) \triangleright (l, c, \sigma, \tau) & \text{ si } c(pc) = \text{if } \alpha \text{ oprel } \alpha' \text{ goto } l \text{ et } \mathcal{E}[\alpha]_{(\sigma, \tau)} \text{ oprel } \mathcal{E}[\alpha']_{(\sigma, \tau)}
 \end{aligned}$$

Génération de code pour les expressions arithmétiques

On considère des tableaux à une dimension et N éléments allant de 0 à N-1 et des tableaux à deux dimensions à NxM éléments (N lignes, M colonnes). La taille d'un élément est T. On peut généraliser sans problème. Les tableaux sont rangés en colonne, ce qui fait que $T[i, j] = N * T * j + i * T$.

$\text{GenCodeAExp}(x)$	=	(ε, x)
$\text{GenCodeAExp}(n)$	=	(ε, n)
$\text{GenCodeAExp}(\text{tab}[i, j])$	=	Soit $t_1 = \text{nouveauTemp}, t_2 = \text{nouveauTemp}$ $t_3 = \text{nouveauTemp}, t_4 = \text{nouveauTemp}$ $t_5 = \text{nouveauTemp}$ dans $(t_1 := T * i \parallel$ $t_2 := N * T \parallel$ $t_3 := t_2 * j \parallel$ $t_4 := t_1 + t_3 \parallel$ $t_5 := \text{tab}[t_4, t_5])$
$\text{GenCodeAExp}(a_1 + a_2)$	=	Soit $(C_1, t_1) = \text{GenCodeAExp}(a_1),$ $(C_2, t_2) = \text{GenCodeAExp}(a_2),$ $t = \text{nouveauTemp}$ dans $(C_1 \parallel C_2 \parallel t := t_1 + t_2, t)$

Génération de code pour les expressions booléennes

$\text{GenCodeBExp}(a_1 < a_2, \text{lvrai}, \text{lfaux})$	=	Soit $(C_1, t_1) = \text{GenCodeAExp}(a_1),$ $(C_2, t_2) = \text{GenCodeAExp}(a_2),$ dans $C_1 \parallel C_2 \parallel \text{if } t_1 < t_2 \text{ goto } \text{lvrai} \parallel \text{goto } \text{lfaux}$
$\text{GenCodeBExp}(b_1 \wedge b_2, \text{lvrai}, \text{lfaux})$	=	Soit $l = \text{nouvelleEtiq}()$ dans $\text{GenCodeBExp}(b_1, l, \text{lfaux}) \parallel$ $l :$ $\text{GenCodeBExp}(b_2, \text{lvrai}, \text{lfaux})$
$\text{GenCodeBExp}(\neg b, \text{lvrai}, \text{lfaux})$	=	$\text{GenCodeBExp}(b, \text{lfaux}, \text{lvrai})$

Génération de code pour les instructions de contrôle

A chaque nœud de l'arbre abstrait, on associe du code de la manière suivante :

<code>GenCodeInst (x := a)</code>	=	Soit $(C,t)=\text{GenCodeAExp}(a)$ dans $C \parallel x := t$
<code>GenCodeInst (S₁ ; S₂)</code>	=	Soit $C_1 = \text{GenCodeInst}(S_1)$, $C_2 = \text{GenCodeInst}(S_2)$ dans $C_1 \parallel C_2$
<code>GenCodeInst (while b do S od)</code>	=	Soit $l\text{debut}=\text{nouvelleEtiq}()$, $l\text{vrai}=\text{nouvelleEtiq}()$, $l\text{faux}=\text{nouvelleEtiq}()$ dans $l\text{debut} : \parallel$ $\text{GenCodeBExp}(b,l\text{vrai},l\text{faux}) \parallel$ $l\text{vrai} : \parallel$ $\text{GenCodeInst}(S) \parallel$ $\text{goto } l\text{debut} \parallel$ $l\text{faux} :$
<code>GenCodeInst (if b then S₁ else S₂)</code>	=	Soit $l\text{suiant}=\text{nouvelleEtiq}()$, $l\text{vrai}=\text{nouvelleEtiq}()$, $l\text{faux}=\text{nouvelleEtiq}()$ dans $\text{GenCodeBExp}(b,l\text{vrai},l\text{faux}) \parallel$ $l\text{vrai} :$ $\text{GenCodeInst}(S_1) \parallel$ $\text{goto } l\text{suiant} \parallel$ $l\text{faux} : \parallel$ $\text{GenCodeInst}(S_2) \parallel$ $l\text{suiant} :$

Correction de la génération

Démontrer les propriétés suivantes ;

- La relation \triangleright est déterministe.
- On note $\#C$ la taille du code, c'est-à-dire le nombre d'instructions. Soit $(C,t)=\text{GenCodeAexp}(a)$.
Montrer que

$$\forall \sigma \tau \cdot \exists \tau' (0, C, \sigma, \tau) \triangleright^* (\#C, C, \sigma, \tau') \text{ ssi } \mathcal{A}[a]_\sigma = \mathcal{E}[t]_{(\sigma, \tau')}$$

- Soit $C=\text{GenCodeInst}(S)$. Montrer que

$$\forall \sigma \sigma' \tau \exists \tau' \cdot (0, C, \sigma, \tau) \triangleright^* (\#C, C, \sigma', \tau') \text{ ssi } (S, \sigma) \longrightarrow \sigma'$$