Université Grenoble Alpes **UFRIM2AG**
Master 1 **Année 2016-2017**

Sémantique des Langages de Programmation et Compilation

Examen du mercredi 7 décembre 2016

— **Durée :** 3 heures. **Tous documents autorisés. Les dispositifs électroniques sont interdits.**
— Les parties sont **indépendantes**.
— Le soin de la copie sera prise en compte : -1 point en cas de manque de soin.

Partie 1 : Typage ($\sim$ 4 points)

Dans cette partie, on s'intéresse à des fragments de langage permettant de décrire des programmes, sans déclaration de variables qui permettraient de décider du type d'une variable. Le type d'une variable ne peut être calculé que lors d'une affectation.

On considère un fragment du langage **While** vu en cours :

$$\begin{array}{lcl} e & ::= & n \mid x \mid e \odot e \mid \cdots \\ S & ::= & x := e \mid S; S \mid \texttt{if}\ e\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{fi} \mid \texttt{while}\ e\ \texttt{do}\ S\ \texttt{od} \end{array}$$

Pour les expressions, l'opérateur binaire $\odot$ désigne n'importe quel opérateur arithmétique ou logique. Concernant les instructions S, nous considérons l'affection, la conditionnelle, la séquence et l'itération :

— L'affectation de la valeur d'une expression à une variable. Une variable conserve la dernière valeur qui lui a été affectée et donc son type. Lorsqu'une variable est affectée plusieurs fois dans un programme le système ne conserve que la dernière valeur et donc le type de cette dernière valeur. Par exemple,
```
x:= 3;
x:= x=4
```
Dans cette séquence est associée le type `int` à x puis le type `Bool`.
— Une instruction conditionnelle. Les deux parties d'une instruction conditionnelle doivent conduire à la définition des mêmes variables avec des types cohérents pour chacune des deux branches. Ainsi, si les variables `a` et `b` ont été affectées précédemment, l'instruction `if a=b then x:= 4 else y:=5` sera incorrecte ; de même que `if a=b then x:= 4 else x:= 4=6`. Par contre, l'instruction `if a=b then (x:=4;y:=3) else (y:=1;x:=8)` est correcte.

Calcul des types On considère les ensembles :
— Noms qui contient l'ensemble des noms de variables,
— Types = {`Int`, `Bool`},
— Env = Noms $\to$ Types $\ni \Gamma$, Γ est une fonction qui à chaque nom de variable associe son type.

On considère les jugements suivants pour les instructions `S` et les expressions `e` :
— pour les instructions : $\Gamma \vdash S \mid \Gamma'$, qui signifie que dans l'environnement Γ, l'instruction S est correctement typée et produit l'environnement Γ'.
— pour les expressions : $\Gamma \vdash e : t$, qui signifie que dans l'environnement Γ l'expression e est bien typée et est de type t.

On donne les règles pour certaines expressions dans la Figure 1. Les exercices de cette partie consistent à modifier le système de typage vu en cours pour prendre en compte les nouvelles constructions.

Exercice 1 Affectation

1. Proposez une règle de typage pour l'affectation.
2. Donnez un programme correct vis-à-vis de cette règle.
3. Donnez un programme incorrect vis-à-vis de cette règle.

$\dfrac{\Gamma(x) = \texttt{Int}}{\Gamma \vdash x : \texttt{Int}}$	x est de type `Int` dans l'environment Γ si $\Gamma(x) = \texttt{Int}$.
$\dfrac{}{\Gamma \vdash n : \texttt{Int}}$	La dénotation n est de type `Int`.
$\dfrac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}}$	$e_1 + e_2$ est de type `Int` si e_1 et e_2 sont de type `Int`.
$\dfrac{\Gamma \vdash e_1 : \texttt{t} \quad \Gamma \vdash e_2 : \texttt{t}}{\Gamma \vdash e_1 = e_2 : \texttt{Bool}}$	$e_1 = e_2$ est de type `Int` si e_1 et e_2 sont de même type `t`, avec `t` `Bool` ou `Int`

FIGURE 1 – Règles pour les expressions

Exercice 2 Conditionnelle

1. Proposez une règle de typage pour la conditionnelle.
2. Donnez un programme correct vis-à-vis de cette règle.
3. Donnez un programme incorrect vis-à-vis de cette règle.

Exercice 3 Séquence

1. Proposez une règle de typage pour la séquence.
2. Donnez un programme correct vis-à-vis de cette règle.
3. Donnez un programme incorrect vis-à-vis de cette règle.

Exercice 4 Itération

1. Expliquez en une phrase le typage de l'itération et proposez une règle de typage pour l'itération.
2. Donnez un programme correct vis-à-vis de cette règle.
3. Donnez un programme incorrect vis-à-vis de cette règle.

Partie 2 : Sémantique opérationnelle ($\sim$ 7 points)

On s'intéresse dans cette partie à la notion de pointeur. On adopte une notation syntaxique "à la C". On modifie la syntaxe du langage **While** de la manière suivante :

$$
\begin{array}{lll}
g & ::= & x \mid *x \mid *(x+n) \\
e & ::= & n \mid g \mid \&x \mid e \odot e \mid \texttt{Null} \\
S & ::= & g := e \mid g := \texttt{malloc}(n) \mid \texttt{free}(x) \mid \texttt{begin}\ D_V\ S\ \texttt{end} \mid S; S \\
D_V & ::= & \texttt{int}\ x\ ;\ D_V \mid \texttt{int}^*\ x\ ;\ D_V \mid \epsilon
\end{array}
$$

Dans cette grammaire, la constante n désigne un entier naturel et `Null` une adresse particulière, dont le contenu est inaccessible. En plus de cette adresse `Null`, on considère deux espaces d'adressage distincts :

— la pile : pour la gestion des données locales des procédures, noté $\texttt{Adr}_S$,

— le tas : pour les données gérées dynamiquement par les fonctions `malloc` et `free`, noté $\texttt{Adr}_H$.

Nous avons $\texttt{Adr}_S \cap \texttt{Adr}_H = \emptyset$.

Par ailleurs, lors de la définition de la sémantique opérationnelle, on considère que la vérification de type a déjà été effectuée. Les types utilisés sont $\{\mathsf{Int}, \mathsf{Int}*\}$.

Ces types ont la même signification que dans le langage C :

— à une variable de type Int est associée une adresse dans la pile contenant une valeur,

— à une variable de type Int* est associée une adresse dans la pile contenant soit une autre adresse dans la pile, soit une adresse dans le tas.

Rappels du cours

Avant de décrire la sémantique, nous rappelons les éléments vus en cours lors de l'extension du langage `While` avec les blocs et les procédures.

— On considère l'environnement, $\texttt{Env}_V$ qui, à chaque variable associe l'adresse d'un emplacement mémoire (table de symboles) :

$$\texttt{Env}_V = \texttt{Var} \to \texttt{Loc} \ni \rho, \hat{\rho}$$

— On considère la mémoire `Store` qui associe une valeur au contenu d'une adresse d'un emplacement mémoire :

$$\texttt{Store} = \texttt{Loc} \to \mathbb{Z} \ni \sigma$$

— Pour la sémantique de l'affectation, nous avions vu :

$$(x := a, \hat{\rho}, \sigma) \to \sigma[\hat{\rho}(x) \mapsto \mathcal{A}[a](\hat{\rho}, \sigma)]$$

L'introduction des instructions `malloc` et `free` va entrainer des modifications de la sémantique, des configurations à la définition des règles de transition.

Nous apportons une modification aux notations du cours : l'espace d'adressage `Loc` est renommé en `Adr`.

Domaines sémantiques

Les domaines sémantiques vont être modifiés pour prendre en compte les deux espaces d'adressage.

Entiers	$\mathbb{Z}$
Booléens	$\mathbb{B}$
Valeurs	$\mathbb{Z} \cup \mathbb{B}$
Adresses	$\texttt{Adr} = \texttt{Adr}_S \cup \texttt{Adr}_H \cup \{\texttt{Null}\}$
Environnement	$\texttt{Env}_V = \texttt{Var} \longrightarrow \texttt{Adr}_S$
Mémoire	$\texttt{Mem} = \texttt{Adr} \setminus \{\texttt{Null}\} \longrightarrow \texttt{Val} \cup \texttt{Adr}$

Une adresse, élément de `Adr`, désigne un emplacement mémoire de la pile, du tas ou `Null`. À une variable est associée une adresse dans la pile. De plus, on suppose connue la fonction Γ qui, à chaque variable x associe son type $\Gamma(x)$. Cette fonction est déterminée lors du typage.

Configurations

Déclarations	$(\texttt{Dec}_V \times \texttt{Env}_V) \cup \texttt{Env}_V$
Instructions	$(\texttt{Stm} \times \texttt{Env}_V^* \times \texttt{Mem}) \cup \texttt{Mem}$

Notations

— On note $\rho, \rho', \rho_S, \cdots$ les éléments de $\texttt{Env}_V$, $\hat{\rho}, \cdots$ les éléments de $\texttt{Env}_V^*$. $\texttt{Env}_V^*$ désigne les piles d'environnements comme dans le cours.

— Enfin, on note σ, σ' les éléments de `Mem`. Notons que $\sigma(\texttt{Null})$ n'est pas défini. Autrement dit, le dé-référencement de `Null` est interdit.

Ainsi, les transitions entre configurations seront de la forme :

$$(S, \hat{\rho}, \sigma) \to \sigma'$$

Commentaires sur les domaines sémantiques Comme vu dans le cours, à un identificateur x, on associe une adresse $\hat{\rho}(x)$ d'un emplacement mémoire qui contiendra soit une valeur $v \in \mathbb{N}$, si x est de type Int ($\Gamma(x) = \mathsf{Int}$), soit une adresse si x est de type Int^* ($\Gamma(x) = \mathsf{Int}^*$).

Exemples Considérons le programme C suivant :

```
int x, *y ;
x = 1;
y = &x ;
```

Ici, $\hat{\rho}(x)$ et $\hat{\rho}(y)$ désignent des adresses dans la pile, i.e, $\hat{\rho}(x), \hat{\rho}(y) \in \texttt{Adr}_S$. On aura $\sigma(\hat{\rho}(x)) = 1$ et $\sigma(\hat{\rho}(y)) = \hat{\rho}(x)$.

Considérons le programme C suivant :

```
int  *y ;
y = malloc(...)
*y = 2;
```

On aura $\sigma(\hat{\rho}(y))$ qui est une adresse dans le tas, i.e. $\sigma(\hat{\rho}(y)) \in \texttt{Adr}_H$, et $\sigma(\sigma(\hat{\rho}(y))) = 2$.

Sous-Partie 1 : blocs et affectations

Dans les questions qui suivent, on demande de produire les règles de sémantique pour la construction bloc, les déclarations et la construction affectation.

Pour les déclarations, il faut initialiser les variables de type `Int` à 0 et les variables de type `Int`* à `Null`.

Pour la construction affectation, on va décomposer en plusieurs règles, en fonction du type de la partie gauche et du type de la partie droite de l'affectation.

Exercice 5

1. Nous nous intéressons à la sémantique du bloc. Donner la sémantique opérationnelle de la construction bloc en complétant la règle suivante :

$$\frac{(D_V, \emptyset, \sigma) \rightarrow_D (\rho_l, \sigma') \quad \cdots}{(\mathsf{begin}\ D_V\ S\ \mathsf{end}, \hat{\rho}, \sigma) \rightarrow \sigma'')}$$

2. Nous nous intéressons aux déclarations. Le compilateur dispose d'une fonction $\texttt{New}_S$ qui, à chaque variable associe une adresse dans la pile. Compléter les deux règles suivantes en se rappelant qu'il faut initialiser une variable de type `Int` à 0 et une variable de type `Int`* à `Null` :

$$\frac{a_x = \texttt{New}_S() \quad (D_V, \rho_l[\cdots], \sigma[\cdots]) \rightarrow (\rho_l', \sigma')}{(\texttt{int}\ x\ ; D_V, \rho_l, \sigma) \rightarrow (\rho_l', \sigma')} \qquad \frac{a_x = \texttt{New}_S() \quad (D_V, \rho_l[\cdots], \sigma[\cdots]) \rightarrow (\rho_l', \sigma')}{(\texttt{int}^*\ x\ ; D_V, \rho_l, \sigma) \rightarrow (\rho_l', \sigma')}$$

3. Nous nous intéressons à la sémantique de l'affectation $x := y$ avec $\Gamma(x) = \Gamma(y) = \mathsf{Int}$.
   Donner la règle de sémantique opérationnelle de cette construction. La sémantique informelle est la suivante : on détermine la valeur de y, qui est un entier, et elle devient la nouvelle valeur de x.
4. Nous nous intéressons à la sémantique de l'affectation $x := y$ avec $\Gamma(x) = \Gamma(y) = \mathsf{Int}*$. Donner la règle de sémantique opérationnelle de cette construction. La sémantique informelle est la suivante : on détermine la valeur de y, qui est une adresse, elle devient la nouvelle adresse associée à x.
5. Nous nous intéressons à la sémantique de l'affectation $x := *y$ avec $\Gamma(x) = \mathsf{Int}$ et $\Gamma(y) = \mathsf{Int}*$. Donner la règle de sémantique opérationnelle de cette construction.
6. Nous nous intéressons à la sémantique de l'affectation $x := \&y$ avec $\Gamma(x) = \mathsf{int}*$ et $\Gamma(y) = \mathsf{Int}$. Donner la règle de sémantique opérationnelle de cette construction.

Sous-Partie 2 : `malloc` et `free`

On s'intéresse aux primitives `malloc` et `free`. Pour cela, on modifie le domaine mémoire en associant à une adresse soit une adresse dans la pile, soit une adresse dans le tas et une taille.

On remplace $\texttt{Mem} = \texttt{Adr} \setminus \{\texttt{Null}\} \longrightarrow \texttt{Val} \cup \texttt{Adr}$ par

$$\texttt{Mem} = \texttt{Adr} \setminus \{\texttt{Null}\} \longrightarrow \texttt{Val} \cup \texttt{Adr}_S \cup (\texttt{Adr}_H \times \mathbb{N}) \cup \{\texttt{Null}\}$$

On suppose disposer d'un allocateur système `new` qui, appelé avec un paramètre entier naturel, retourne une adresse d'un emplacement libre dans le tas.

Exercice 6 Sémantique du `malloc`

On suppose $\Gamma(x) = \mathsf{int}*$.

1. Compléter la règle suivante :

$$\frac{v = \cdots \quad l_h = \mathsf{new}(v)}{(x = \texttt{malloc}(n), \hat{\rho}, \sigma) \to (\sigma[\hat{\rho}(x) \mapsto (l_h, v)])}$$

Exercice 7 Sémantique du `free`

On donne la sémantique de la fonction `free`, qui libère simplement un emplacement mémoire sur le tas.

$$\frac{}{(\mathsf{free}(x), \hat{\rho}, \sigma) \to \sigma}$$

On se propose de renforcer la sémantique pour faire en sorte qu'une zone libérée devienne inaccessible.

Supposons que les programmes `C` suivants (P1, ...,P4) ont les déclarations suivantes :

```
  P1                      P2          P3              P4
  --                      --          --              --
int *a ;                  int a ;     int a ;         int *a ;
int *b ;                  int *b ;    int *b ;        int *b ;
a=malloc(4*sizeof(int));                              a=malloc(4*sizeof(int));

&b=a;                     b=&a        b=&a+a          b=a+3;
free(b);                  free(b);    free(b);        free(b);
*a=2;                                                 *a=2;
```

Aucune erreur à la compilation pour ces 4 programmes.

1. L'exécution des programmes P2 et P3 produit une erreur. En examinant ces programmes, on constate que dans le cas de P2 et P3, on a une tentative de libérer de la mémoire située dans l'espace d'adressage de la pile $\texttt{Adr}_S$. Modifier la règle de l'instruction `free` pour qu'une erreur à l'exécution soit levée dans P2 et P3.
2. L'exécution des programmes P1 et P4 ne produit aucune erreur. En revanche, on accède à de la mémoire libérée, ce qui peut conduire à des situations incertaines. Renforcer la règle précédente pour qu'une erreur à l'exécution soit levée dans P1 et P4.

Exercice 8 Sémantique de l'affectation (suite)

1. Donner la sémantique opérationnelle pour l'instruction

$$*p := *q$$

2. Donner la sémantique opérationnelle pour l'instruction

$$*(p+n) := *(q+m)$$

n et m sont des constantes entières.

Exercice 9 Bonus : Analyse de programmes

On se propose de détecter une tentative d'accès à la mémoire au travers d'un pointeur `Null`. Dans cet exercice, un bloc de base est réduit à une seule instruction d'affectation. Aussi, on ne considère que des programmes sans conditionnelle ni itération.

Un pointeur x est égal à `Null` en un point `j` du programme

— soit parce qu'il existe un chemin allant du bloc initial à j contenant une affectation à `Null` et que cette affectation est la dernière affectation à x sur ce chemin,

— soit parce que sur ce chemin, il existe une affectation $x = y$ et que y est `Null`

On va propager en avant un ensemble de variables sur une séquence S_0 d'affectations. Initialement, cet ensemble est égal à $\emptyset$. On va définir pour chaque affectation son effet sur un ensemble de variables dont on sait qu'elles ont une adresse associée à `Null`. On définit une fonction de transfert qui prend en compte l'effet de chaque affectation sur l'ensemble. Considérons l'exemple suivant :

```
 1
x = Null
 2
y = x
 3
```

Au point de contrôle `1` (resp. `2`, `3`), l'ensemble est égal à $\emptyset$ (resp. {`x`}, {`x`, `y`}). On définit une fonction F à 2 arguments, le premier étant un ensemble de variables, le second une séquence d'affectations. Partant de $X = \emptyset$, on calcule

$$F(X, x := e; S) = F(X', S)$$

où X est le résultat de l'effet de l'affectation sur X. Ainsi dans l'exemple, on aura

$$F(\emptyset, \texttt{x} = \texttt{Null}; \texttt{y} = \texttt{x}) = F(\{x\}, \texttt{y} = \texttt{x}) = F(\{x, y\}, \epsilon) = \{x, y\}$$

1. Calculer $F(X, x = \texttt{Null}; S)$
2. Calculer $F(X, x = y; S)$
3. Détecter l'accès à des pointeurs `Null` sur les parties droites d'affectation.

Partie 3 : Optimisation ($\sim$ 5 points)

On considère le graphe de flot de contrôle de la Figure 2. Nous nous intéressons aux expressions disponibles et variables actives, au sens donné dans le cours.

Exercice 10 Expressions disponibles

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. Supprimer les calculs redondants.

Dans la suite, on considère le graphe modifié résultat de l'exercice précédent.

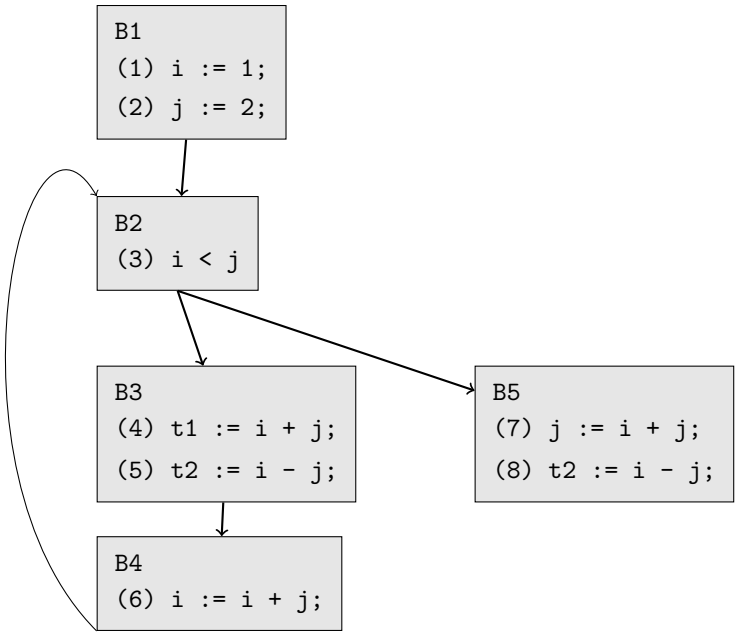

FIGURE 2 – Graphe de flot de contrôle initial

Exercice 11 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.

Partie 4 : Génération de code ($\sim$ 5 points)

Exercice 12 Génération de code

On considère le programme dans la Figure 3.

1. Dessiner la pile lors de l'exécution de g3 en représentant uniquement le chaînage dynamique et le chaînage statique.
2. On considère la procédure p1. Donner la séquence de code qui réalise `z = g2(2)`.
3. On considère la procédure p2. Donner la séquence de code qui réalise `z = y + p3(p)`.
4. On considère la procédure p3. Donner la séquence de code qui réalise `return p(x+x1+y)`.
5. On considère la procédure g2. Donner la séquence de code qui réalise `y = p2(x1,g3)`.

```
#include <stdio.h>
typedef int (*intprocint) ( int);
/* intprocint désigne le type procedure ayant un paramètre entier et un résultat entier */

typedef int (*intprocintint) (int,int);
typedef int (*intprocvoid) (void);

main(){
  int x1;
  int p1 () {
    int x ;
    int y ;
    int z ;
    int p2(int x,intprocint p) {
      int p3 (intprocint p){
        return p(x+x1+y);
      }
      z=11;
      z = y + p3(p);
      return z;
    }
    int g2(int x) {
      int g3 (int x){return (x-1);}
      if (x>0) y=p2(x1,g3);
    }
    y=22;
    z=g2(2);
    return z;
  }
  x1=0;
  x1=p1();
  printf("%d\n",x1);
}
```

FIGURE 3 – Programme pour la génération de code