Sémantique des langages de programmation première partie: sémantique naturelle

Yves Bertot

Mars 2011

1 Pourquoi étudier la sémantique des langages de programmation?

L'objectif de la sémantique des langages de programmation est de fournir des méthodes pour raisonner sur les programmes et de permettre la programmation sans erreurs. La sémantique des langages de programmation doit permettre d'éviter l'introduction d'erreur, en se reposant sur deux techniques:

- Grâce à la sémantique du langage de programmation, on pourra décrire précisément le comportement du programme et le comparer à un comportement attendu, par exemple sous la forme de relations logiques entre valeurs d'entrée et valeurs de sortie.
- La sémantique permettra également de définir des disciplines de programmation et de montrer que ces disciplines permettent d'éviter certaines classes d'erreurs de programmation.

Cette matière jouit d'un intérêt économique réel. L'informatique embarquée, utilisée de plus en plus fréquemment dans tous les appareils de la vie moderne, requiert un niveau de qualité bien plus important que les application traditionnelles de l'informatique. De nombreuse applications requièrent maintenant un niveau de qualité des programmes que seules des méthodes formelles de développement et de vérification peuvent assurer. Les domaines les plus demandeurs actuellement sont ceux du transport (Matra Transport, Alsthom, Dassault-Aviation), des télécommunication (France-Télécom) et le domaine des transactions commerciales (Gemalto, Schlumberger).

Ce cours est conçu comme une introduction aux différentes techniques utilisées en sémantique des langages de programmation. Il s'inspire des premiers chapitres d'un livre écrit en 1993 par Glynn Winskel, *The formal semantics of programming languages, an introduction*, et publié par MIT Press dans la série *Foundations of Computing*. Nous tâcherons de présenter les différents aspects suivants:

1. La sémantique naturelle: présenter l'exécution des programmes comme un système logique,
2. Preuves par récurrence: applications aux langages de programmation,
3. L'exécution de spécifications sémantiques,
4. Les preuves en sémantique.

Tout au long de notre travail, nous appuierons notre étude sur l'exemple d'un tout petit langage.

2 Description des structures de données

Nous allons volontairement réduire au maximum le langage que nous étudions. L'objectif est de faciliter notre étude en ne voyant qu'une fois chacun des concepts qui seraient répétés pour un langage plus réaliste.

2.1 Expressions et instructions

La première étape de traitement d'un programme est l'analyse syntaxique. Le résultat de cette analyse est un *terme de syntaxe abstraite*. La définition du langage passe par la description de l'ensemble des termes qui seront admis. Voici la définition du langage.

- Soit V un ensemble de variables, les éléments de V seront notés v, v', v_i, w, x, ...
- le langage des expressions arithmétiques, $aexp$, dont les éléments seront notés e, e', etc, est l'ensemble des termes de la forme v, n (ou n est un nombre entier), ou $e_1 + e_2$. Parfois, pour bien distinguer l'addition de deux expressions arithmétique (qui est un fragment de programme qui parle d'addition) de l'addition de deux nombre entiers, nous utiliserons une notation différente pour la première $plus(e_1, e_2)$. Dans la littérature sur les langages de programmation, on présente habituellement cette définition par la ligne suivante:
$$e ::= v \mid n \mid e_1 + e_2$$
- le langage des expressions booléennes, $bexp$, dont les éléments seront notés b, b', etc, est l'ensemble des termes de la forme $blt(b, b')$,
- le langage des instructions, $instr$, dont les éléments seront notés I, I', etc, est l'ensemble des termes de la forme $v := e \mid (I_1; I_2) \mid$ `while` b `do` I `done`.

Dans ce langage on peut représenter le fragment de programme suivant:

```
v1 := 0;
v2 := 0;
while v3 > v2 do
  v1 := v2;
  v2 := v2 + 1
done
```

Dans la suite du cours, nous utiliserons la représentation des termes sous leur forme "mathématique" ou dans leur syntaxe "concrète", suivant ce qui est le mieux adapté pour nos besoins.

Exercices

1. *Ecrire un terme pour un programme qui échange les valeurs de deux variables v_1 et v_2, en utilisant une variable intermédiaire v_3.*

2. *Ecrire un terme pour un programme qui met dans une variable v_2 la factorielle de la valeur d'une variable v_1.*

2.2 Encodage en Coq

Pour représenter les programmes nous utiliserons trois types de données différents de Coq, tous définis comme des types inductifs.

```
Require Export ZArith String.
Require Export List.
Open Scope Z_scope.
Open Scope string_scope.

Inductive aexpr : Type :=
  avar (s:string) | anum (x:Z) | aplus (e1 e2 : aexpr).

Inductive bexpr : Type :=
  blt (e1 e2 : aexpr).

Inductive instr : Type:=
  skip | assign (s:string) (e:aexpr)
| sequence (i1 i2 : instr) | while (b : bexpr) (i:instr).
```

L'expression `assign "x" (aplus (avar "x") (anum 1)` représente donc l'instruction $x := x + 1$.

3 Etat des machines

Le comportement d'un programme impératif s'exprime par la modification d'un état. Il faut déterminer comment nous allons le représenter. Pour le langage qui nous intéresse, il suffit de modéliser cet état par une correspondance entre les

variables du programme et des valeurs. Par nature, il n'existe qu'un nombre fini de variables dans le programme et nous pourrons nous contenter de représenter l'état comme une liste de couples dont le premier élément est une variable et le second est une valeur. Cette liste de couples sera également un terme, dans un langage défini de la façon suivante:

- le langage des couples est constitué des expressions de la forme $pair(v, n)$ où v est une variable et n est un nombre
- le langage des états machines, *state*, dont les éléments seront notés s, s', etc, est l'ensemble des termes nil ou $cons(pair(v, n), s)$.

Pour une écriture plus concise, nous écrirons souvent

$$(v, n) \cdot s$$

au lieu de

$$cons(pair(v, n), s).$$

3.1 représentation des états en Coq

Pour les états nous utiliserons le type prédéfini des listes et le type prédéfini des couples. Le type des états sera en fait appelé `env` (pour "environnement").

```
Definition env := list (string * Z).
```

4 Descriptions sémantiques

4.1 Jugements

Dans notre travail sur la description des langages, nous allons raisonner sur des énoncés de la forme *l'exécution de l'instruction i à partir de l'état s retourne le nouvel état s'* ou bien *l'évaluation de la variable v dans l'état s retourne la valeur numérique n*. Pour chacun de ces énoncés, nous utiliserons une notation abrégée, dont le style est emprunté à la théorie de la preuve.

Dans cette section, nous allons énumérer les différents types d'énoncés qui apparaitront dans notre description du langage de programmation, en tâchant de démystifier les notations utilisées.

Pour chacun de ces énoncés nous utilisons des notations avec différents types de flèches, comme $\rightarrow$, $\mapsto$ ou $\leadsto$. La signification attachée à chacune de ces flèches est complètement arbitraire: nous n'utilisons ces flèches que pour disposer d'une notation compacte.

4.1.1 Evaluation des expressions

Pour déterminer la valeur d'une variable dans un état machine donné, nous utiliserons un énoncé de la forme *dans l'état s l'expression e a la valeur n*. Nous écrirons cet énoncé sous la forme suivante:

$$s \vdash e \to n$$

Il ne faut pas chercher à attribuer une signification précise aux symboles $\vdash$ et $\to$. Nous sommes simplement en présence d'une relation entre trois objets s, v, et n, et les symboles $\vdash$ et $\to$ sont utilisés pour séparer les différents objets mis en relation. Le fait que cette relation soit ternaire et que ces symboles soient utilisés, dans cet ordre, pour représenter cette relation est purement conventionnel. Le symbole $\vdash$ est utilisé communément en théorie de la preuve et est souvent appelé "thèse".

Nous utiliserons la même notation pour représenter l'évaluation d'expressions booléennes.

4.1.2 Mise à jour de l'état

L'affectation provoque le changement de la valeur associée à une variable. Dans notre représentation logique cela se traduit par la création d'un nouvel état qui ne diffère de l'ancien que par la valeur associée à la variable. Nous utiliserons donc un énoncé de la forme *dans l'état initial s, la mise à jour de la variable v avec la valeur n retourne un nouvel état s'*. Nous écrirons cet énoncé de la façon suivante:

$$s \vdash v, n \mapsto s'$$

Ici encore, l'utilisation des symboles $\vdash$ et $\mapsto$ est purement conventionnelle.

4.1.3 Exécution d'une instruction

Lorsque l'on exécute une instruction, on modifie l'état de la machine. Nous utiliserons donc des énoncés de la forme *dans l'état s l'exécution d'une instruction I retourne le nouvel état s'*. Nous écrirons cet énoncé de la façon suivante:

$$s \vdash I \rightsquigarrow s'.$$

Lorsque l'on écrit ce jugement, on exprime également le fait que l'exécution de I termine.

4.1.4 Encodage des jugements en Coq

En Coq, nous utiliserons un prédicat à trois arguments pour $s \vdash e \to n$, c'est à dire une fonction `aeval` qui prend trois arguments et retourne une propriété. Le résultat est donc de type `Prop`.

```
aeval : env -> aexpr -> Z -> Prop
```

Par exemple, le jugement $s \vdash e \to n$ sera représenté en Coq par `aeval` s e n.

Nous serons également amenés à utiliser la notation $s \vdash e \to v$ lorsque e est une expression booléenne et v est une valeur booléenne. Dans ce cas nous ne pouvons pas utiliser le même prédicat, le nouveau prédicat s'appellera `beval`.

```
beval : env -> bexpr -> bool -> Prop
```

Pour la mise à jour de l'état, c'est un prédicat à quatre arguments que nous utilisons.

```
update : env -> string -> Z -> env -> Prop
```

Enfin pour l'exécution d'une instruction nous utiliserons un prédicat de la forme suivante:

```
exec : env -> instr -> env -> Prop
```

Nous verrons plus loin comment ces prédicats sont définis.

4.2 Formules logiques auxiliaire

Nous serons également amenés à traiter des énoncés simples sur les entiers naturels, de la forme $n_1 < n_2$, $n_1 \leq n_2$ ou bien $n_1 + n_2 = n_3$. La signification de ces énoncés sera toujours directe, étant donné que n_1, n_2, et n_3 sont censées représenter des valeurs entières connues.

En Coq, ces formules logiques seront représentées directement par des formules logiques de Coq.

4.3 Schémas de raisonnements: les règles d'inférence

Suivant l'approche décrite dans ce chapitre, décrire la sémantique du langage de programmation, c'est tout simplement donner les principes qui permettent d'affirmer qu'un énoncé est vrai. Ces principes prennent la forme d'étapes de raisonnement mettant en jeu plusieurs énoncés.

La notation est encore inspirée de la théorie de la preuve. On représente les étapes élémentaires de raisonnement logique par des règles d'inférence, qui ont toutes la forme *si les propositions P_1, ..., P_k sont prouvées, alors on peut prouver la proposition Q*. La tradition est de représenter ces règles d'inférence avec une barre horizontale de la manière suivante:

$$\frac{P_1 \dots P_k}{Q}$$

Les proposition P_i sont généralement appelées les *prémisses* de la règle, tandis que Q est appelée la conclusion.

Les raisonnements représentés sont schématiques au sens où ce sont des schémas qui peuvent se reproduire dans plusieurs situations différentes. Plus précisément, les propositions P_1, ..., P_k, Q peuvent contenir des variables, éventuellement partagées entre plusieurs propositions. Un nombre fini de règles permet donc de représenter l'ensemble (infini) des exécutions possibles de programmes.

Pour Coq, les barres de fractions utilisées dans les règles d'inférences représentent simplement des implications et nous utiliserons directement des flêches comme pour les raisonnements de Coq.

Nous allons maintenant énumérer l'ensemble des règles d'inférences qui permettent de décrire la sémantique de notre langage d'exemple.

4.4 Evaluation des variables

Pour connaitre la valeur d'une variable v dans un état donné s, il faut que la valeur de cette variable soit spécifiée dans l'état. Si l'état est effectivement représenté par une liste de couples, deux cas peuvent se produire, suivant que le premier couple fait référence à la variable v ou non.

Si la variable v apparait au début de la liste de couples, l'état s a alors la forme $(v, n) \cdot s'$ et la valeur de l'expression est alors n. Nous exprimons ceci par la règle d'inférence suivante:

$$\frac{}{(v, n) \cdot s' \vdash v \to n} \tag{1}$$

Si la variable v a une valeur n dans un état s, elle a encore la même valeur dans un environnement où l'on ajoute un autre couple associant une variable à une valeur. Nous exprimons ceci par la règle d'inférence suivante:

$$\frac{s \vdash v \to n \quad v \neq v'}{(v', n') \cdot s \vdash v \to n} \tag{2}$$

4.5 Evaluation des expressions

4.5.1 Valeurs numériques immédiates

En dehors des variables, on est aussi amené à évaluer des valeurs numériques et des expressions d'additions. Pour les valeurs numériques, c'est simple: leur évaluation retourne leur valeur dans n'importe quel état:

$$\frac{}{s \vdash n \to n} \tag{3}$$

4.5.2 Addition

Si l'évaluation de deux expressions e_1 et e_2 dans un même état s retourne deux valeurs numériques n_1 et n_2, alors l'évaluation de l'expression $plus(e_1, e_2)$ dans le même état s retourne la somme de n_1 et n_2. Ceci s'exprime de la façon suivante:

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2}{s \vdash plus(e_1, e_2) \to n_1 + n_2} \tag{4}$$

4.5.3 Encodage en Coq

Nous donnons toutes les règles d'évaluation d'expressions arithmétiques d'un seul coup. Ceci définit en même temps le prédicat `aeval` et quatre théorèmes qui permettent de prouver certaines instance de ce prédicat.

```
Inductive aeval : env -> aexpr -> Z -> Prop :=
  ae_num : forall r n, aeval r (anum n) n
```

```
| ae_var1 : forall r x v, aeval ((x,v)::r) (avar x) v
| ae_var2 : forall r x y v v' , x <> y -> aeval r (avar x) v' ->
              aeval ((y,v)::r) (avar x) v'
| ae_plus : forall r e1 e2 v1 v2,
              aeval r e1 v1 -> aeval r e2 v2 ->
              aeval r (aplus e1 e2) (v1 + v2).
```

Cette façon de décrire `aeval` exprime aussi qu'il n'y a pas d'autre moyen de prouver des instances de ce prédicat que d'utiliser les quatre règles `ae_num`, `ae_var1`, `ae_var2`, `ae_plus` un nombre fini de fois.

4.6 Evaluation des expressions booléennes

4.6.1 Comparaisons

On peut être amené à avoir plusieurs règles pour le même opérateur. Ainsi, la comparaison se traite de façon similaire à l'addition, mais une règle de raisonnement représente le cas ou la valeur de la comparaison est le booléen `true` et une règle de raisonnement représente le cas où la valeur de la comparaison est le booléen `false`:

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2 \quad n_1 < n_2}{s \vdash blt(e_1, e_2) \to true} \tag{5}$$

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2 \quad n_2 \leq n_1}{s \vdash blt(e_1, e_2) \to false} \tag{6}$$

4.6.2 Encodage en Coq

```
Inductive beval : env -> bexpr -> bool -> Prop :=
| be_lt1 : forall r e1 e2 v1 v2,
            aeval r e1 v1 -> aeval r e2 v2 ->
            v1 < v2 -> beval r (blt e1 e2) true
| be_lt2 : forall r e1 e2 v1 v2,
            aeval r e1 v1 -> aeval r e2 v2 ->
            v2 <= v1 -> beval r (blt e1 e2) false.
```

4.7 Modifier la valeur associée à une variable

4.7.1 Description des règles d'inférence

Si l'on veut associer la variable v à la valeur n dans l'état de la machine, on va produire un nouvel état. La modification est simple à décrire si la variable v apparait dans le premier couple de l'état: la valeur dans le premier couple est modifiée, mais le reste de l'état mémoire reste inchangé. Ceci s'exprime par la règle d'inférence suivante.

$$\frac{}{(v, n') \cdot s \vdash v, n \mapsto (v, n) \cdot s} \tag{7}$$

Si l'on sait mettre à jour l'état s pour la variable v et la valeur n en obtenant un nouvel état s', alors la mise à jour de l'état $(v', n') \cdot s$ pour cette variable et cette valeur fournit l'état $(v', n') \cdot s'$. Notez que la valeur associée à la variable v' n'est alors pas changée. Ce schéma de raisonnement est valable si v et v' sont des variables distinctes. Ceci s'exprime par la règle d'inférence suivante:

$$\frac{s \vdash v, n \mapsto s'}{(v', n') \cdot s \vdash v, n \mapsto (v', n') \cdot s'} \quad (v \neq v') \tag{8}$$

4.7.2 Encodage en Coq

```
Inductive update : env->string->Z->env->Prop :=
| s_up1 : forall r x v v', update ((x,v)::r) x v' ((x,v')::r)
| s_up2 : forall r r' x y v v', update r x v' r' ->
            x <> y -> update ((y,v)::r) x v' ((y,v)::r').
```

Exercices

3. *On ajoute au langage des expressions booléennes $true$ et $false$, Ecrivez les règles d'inférence qui décrivent l'évaluation de ces expressions (s'inspirer des valeurs numériques).*

4. *On ajoute au langage une expression $minus$ pour représenter la soustraction. Ecrivez les variantes possibles de règles d'inférence qui décrivent l'évaluation des expression contenant des soustractions.*

5. *Même question avec un opérateur d'égalité.*

4.8 Exécution des instructions

4.8.1 Affectation

Si l'évaluation d'une expression e dans un état s retourne la valeur n et si la mise à jour de l'état pour la variable v et la valeur n à partir de l'état s retourne le nouvel état s', alors l'exécution de l'affectation $assign(v, e)$ dans l'état s termine et retourne le nouvel état s'.

$$\frac{s \vdash e \to n \quad s \vdash v, n \mapsto s'}{s \vdash assign(v, e) \rightsquigarrow s'} \tag{9}$$

4.8.2 Séquence

Si l'exécution d'une instruction I_1 à partir d'un état s retourne un état s' et l'exécution d'une instruction I_2 à partir de s' retourne s'', alors l'exécution de l'instruction $sequence(I_1, I_2)$ à partir de l'état s termine et retourne l'état s''.

$$\frac{s \vdash I_1 \rightsquigarrow s' \quad s' \vdash I_2 \rightsquigarrow s''}{s \vdash sequence(I_1, I_2) \rightsquigarrow s''} \tag{10}$$

4.8.3 Boucle

Si dans l'état s l'expression e s'évalue à $true$ et l'exécution de l'instruction I termine en retournant le nouvel état s' et si dans l'état s' l'exécution de l'instruction $while(e, I)$ termine en retournant un nouvel état s'' alors dans l'état s l'exécution de l'instruction $while(e, I)$ termine en retournant l'état s''.

$$\frac{s \vdash e \to true \quad s \vdash i \rightsquigarrow s' \quad s' \vdash while(e,i) \rightsquigarrow s''}{s \vdash while(e,i) \rightsquigarrow s''} \tag{11}$$

Une autre règle d'inférence permet de décrire le comportement de l'instruction de boucle lorsque l'expression s'évalue à $false$. Dans ce cas, l'exécution s'arrête sans changement d'état:

$$\frac{s \vdash e \to false}{s \vdash while(e,I) \rightsquigarrow s} \tag{12}$$

4.8.4 Encodage en Coq

```
Inductive exec : env->instr->env->Prop :=
| SN1 : forall r, exec r skip r
| SN2 : forall r r' x e v,
   aeval r e v -> update r x v r' -> exec r (assign x e) r'
| SN3 : forall r r' r'' i1 i2,
    exec r i1 r' -> exec r' i2 r'' ->
    exec r (sequence i1 i2) r''
| SN4 : forall r r' r'' b i,
    beval r b true -> exec r i r' ->
    exec r' (while b i) r'' ->
    exec r (while b i) r''
| SN5 : forall r b i,
    beval r b false -> exec r (while b i) r.
```

Exercices

6. *Ajoutez dans le langage une instruction $if(e, i_1, i_2)$ et la ou les règles d'inférences qui sont adaptées.*

7. *En supposant que le comportement de l'instruction if est déjà spécifié, écrire une règle pour la boucle qui se contente de réutiliser la sémantique de if.*

5 Utiliser la spécification sémantique

Les règles d'inférences sont faites pour s'emboiter, en utilisant d'autres règles pour prouver les prémisses d'une règle.

5.1 Construction de dérivations sous forme de figures

Considérons la séquence d'instructions suivante.

```
x := x + y;
y := x + (- 4);
x := x + (- 3);
```

Nous voulons décrire l'exécution de cette séquence d'instructions dans l'environnement où `y` vaut initialement 4 et `x` vaut initialement 3. Prenons les notations suivantes:

$$\begin{array}{lcllcl} P_2 & = & \text{y := x + (- 4); x := x + (- 3)} & P_1 & = & \text{x := x + y; } P_2 \\ s_0 & = & (\text{“x”}, 3) \cdot (\text{“y”}, 4) \cdot \emptyset & s_1 & = & (\text{“x”}, 7) \cdot (\text{“y”}, 4) \cdot \emptyset \\ s_2 & = & (\text{“x”}, 7) \cdot (\text{“y”}, 3) \cdot \emptyset & s_3 & = & (\text{“x”}, 4) \cdot (\text{“y”}, 3) \cdot \emptyset \\ s_4 & = & (\text{“y”}, 4) \cdot \emptyset & & & \end{array}$$

Nous allons construire une dérivation pour l'énoncé suivant:

$$s_0 \vdash P_1 \rightsquigarrow s_3$$

Le squelette de la dérivation a la forme suivante:

$$\dfrac{\dfrac{D_1}{s_0 \vdash \texttt{x:=x+y} \rightsquigarrow s_1} \qquad \dfrac{\dfrac{D_2}{s_1 \vdash \texttt{y:=x + (-4)} \rightsquigarrow s_2} \qquad \dfrac{D_3}{s_2 \vdash \texttt{x:=x + (-3)} \rightsquigarrow s_3}}{s_1 \vdash P_2 \rightsquigarrow s_3}}{s_0 \vdash P_1 \rightsquigarrow s_3}$$

Dans cette figure, D_1, D_2 et D_3 représentent des sous-dérivations, chacune pour une affectation. La dérivation D_1 est donnée dans la figure suivante:

$$\dfrac{D_4 \qquad \overline{(\text{“x”}, 3) \cdot s_4 \vdash \text{“x”}, 7 \mapsto (\text{“x”}, 7) \cdot s_4}}{s_0 \vdash \texttt{x:=x+y} \rightsquigarrow s_1}$$

La dérivation D_4 est donnée dans la figure suivante:

$$\dfrac{\overline{(\text{“x”}, 3) \cdot (\text{“y”}, 4) \cdot s \vdash \text{“x”} \to 3} \qquad \dfrac{\overline{(\text{“y”}, 4) \cdot s \vdash \text{“y”} \to 4} \quad \text{“x”} \neq \text{“y”}}{(\text{“x”}, 3) \cdot (\text{“y”}, 4) \cdot s \vdash \text{“y”} \to 4}}{s_0 \vdash \texttt{x+y} \to 7}$$

La preuve que “x” $\neq$ “y” ne repose pas sur l'utilisation d'une des règles logiques ajoutées pour décrire le langage de programmation, mais doit normalement être faite.

Exercices

8. *Construire les dérivations pour D_2 et D_3.*

9. *Construire la dérivation pour l'énoncé suivant:*

$$(\text{“x”}, 1) \cdot s \vdash \texttt{while(2>x)\{x:= x+x\}} \rightsquigarrow (\text{“x”}, 2) \cdot s$$

10. *On considère que la taille d'une expression arithmétique est le nombre d'opérateurs qui entre dans sa fabrication, en assimilant les variables et les valeurs numériques à des expressions de taille 1. l'expression* $plus(x, 1)$ *est de taille 3. Pour les dérivations, on considère que la taille est le nombre de règles qui interviennent dans leur construction (c'est la même chose, n'est-ce pas?). Quel est la taille minimale de la dérivation pour une expression arithmétique de taille* n*. Si l'on considère que l'évaluation a lieu dans un état de taille* p*, existe-t-il une taille maximale pour les dérivations sur les expressions de taille* n*?*

11. *Quelle est la taille de la dérivation décrivant l'exécution de l'expression* $while(x > y, assign(y, y + 1))$ *dans l'état* $(x, n) \cdot (y, 0) \cdot s$*?*

12. *Quelle est la taille de la dérivation décrivant l'exécution de l'expression*

    $$while(x > y, sequence(while(x > z, assign(z, z + 1)), assign(y, y + 1)))$$

    dans l'état $(x, n) \cdot (y, 0) \cdot (z, 0) \cdot s$*?*

13. *Ecrire un programme qui prend en entrée des représentations d'un environnement et d'une instruction et retourne en sortie la taille de la dérivation calculant l'exécution de cet instruction dans cet environnement lorsque cette taille existe (le programme pourra ne peut pas terminer s'il n'existe pas une telle dérivation). Tant qu'à faire, il ne sera pas plus difficile que ce programme retourne aussi l'environnement final. Le langage de programmation est laissé au choix de l'étudiant, mais Ocaml est conseillé.*

5.2 Représentation des dérivations en Coq

En Coq, les dérivations sont des preuves d'énoncé logiques. Ce qui en fait des objets particuliers est que ces preuves sont obtenues par composition des constructeurs des définitions inductives. Si l'on observe à nouveau les dérivations construites à la section précédente, elles utilisent les règles d'inférences pour l'évaluation de l'addition, de variables, de nombres entiers. Par exemple, la dérivation D_4 est représentée par la preuve Coq suivante.

```
Lemma D4 :
  aeval (("x", 3)::("y", 4)::nil)(aplus (avar "x")(avar "y")) 7.
Proof.
apply ae_plus with (v1 := 3) (v2 := 4).
  apply ae_var1.
apply ae_var2.
  discriminate.
apply ae_var1.
Qed.
```

On voit bien apparaître l'utilisation des trois constructeurs de la définition `aeval`, `ae_plus` (pour la règle d'addition), `ae_var1` et `ae_var2`. L'énoncé initial

est la formule prouvée par la dérivation, les formules intermédiaires apparaissent comme buts intermédiaires. L'expression `"x" <> "y"` est démontrée par la tactique `discriminate`.

La dérivation D_1 peut être modélisée par la preuve Coq suivante, en réutilisant la preuve D_4.

```
Lemma D1 :
  exec (("x", 3)::("y", 4)::nil)
       (assign "x" (aplus (avar "x")(avar "y")))
       (("x", 7)::("y", 4)::nil).
apply SN2 with (v := 7).
  exact D4.
apply s_up1.
Qed.
```

5.3 Raisonnement sur les dérivations

5.3.1 Simple raisonnement par cas

Lorsque l'on veut prouver un énoncé de la forme $s \vdash e \to n \Rightarrow P(s, e, n)$, nous pouvons utiliser la connaissance que l'énoncé $s \vdash e \to n$ a forcément été démontré à l'aide des règles d'inférences. On peut donc faire un traitement par cas, dans lequel on tire de l'information de la forme des règles utilisées.

Par exemple, on peut démontrer la valeur associée la variable "y" dans l'environnement $(\text{“x”}, 1) \cdot (\text{“y”}, 2) \cdot \emptyset$ est forcément 2. Le raisonnement fonctionne ainsi:

1. pour prouver $(\text{“x”}, 1) \cdot (\text{“y”}, 2) \cdot \emptyset \vdash \text{“y”} \to n$, on n'a pas pu utiliser la règle pour les entiers (3) ou la règle pour les additions (4). Parmi, les règles pour les variables, on n'a pas pu utiliser non plus la règle (1), parce que cette règle impose que la variable que l'on trouve en tête de l'environnement soit la même que celle que l'on veut évaluer. Ici, la variable en tête de l'environnement est "x" et la variable que l'on veut évaluer est "y". La seule règle que l'on a pu utiliser est donc forcément la règle (2) présentée vers la page 7. Puisque cette règle a été appliquée, on peut en déduire que la prémisse $(\text{“y”}, 2) \cdot \emptyset \vdash \text{“y”} \to n$ a forcément été prouvée également.

2. Pour prouver $(\text{“y”}, 2) \cdot \emptyset \vdash \text{“y”} \to n$, on sait que les règles (2), (3), et (4) n'ont pas pu être utilisées. En particulier, la règle (2) n'a pu s'appliquer parce que cette règle exige que la variable en tête de l'environnement (ici "y") soit différentes de la variable dont on veut déterminer la valeur (ici c'est aussi "y"). C'est donc seulement la règle (1) qui a pu être appliquée ici:

   $$(x, n) \cdot s \vdash x \to n$$

   Cette règle impose que la valeur retournée est la même que celle trouvée en deuxième composante du couple en tête de l'environnement. Donc la valeur de n est nécessairement 2.

5.3.2 Raisonnement par récurrence sur les dérivations

Un autre critère que l'on peut utiliser pour raisonner sur les dérivations est leur taille et on peut utiliser cette taille pour faire un raisonnement par récurrence. Dans le cas d'une dérivation d'évaluation, on suppose que l'on dispose d'une dérivation d prouvant un énoncé $s \vdash e \to n$, on peut alors chercher à prouver une propriété $P(s, e, n)$. Si l'on décide de faire une preuve par récurrence sur la dérivation d, cela signifie que l'on s'autorise à utiliser une hypothèse de récurrence qui indique que pour toute dérivation d' de taille plus petite que d, si d' prouve $s' \vdash e' \to n'$. alors la propriété $P(s', e', n')$ est déjà satisfaite. Pour utiliser cette notion de récurrence, il n'est pas vraiment nécessaire de définir ce qu'est la taille d'une dérivation, car très souvent on n'utilise l'hypothèse de récurrence que pour des dérivations qui sont des sous-dérivation de la dérivation d.

Par exemple, nous allons démontrer que l'évaluation d'une expression arithmétique dans un environnement donné retourne toujours le même résultat.
Théorème *si* $s \vdash e \to n$ *et* $s \vdash e \to n'$ *alors* $n = n'$.
Démonstration. Considérons une dérivation D quelconque et démontrons par récurrence sur la taille de cette dérivation l'énoncé suivant: pour tout environnement s, toute expression arithmétique e et toute valeur entière n, si D prouve $s \vdash e \to n$, alors pour toute valeur n' telle que $s \vdash e \to n'$ soit également prouvable on a $n = n'$.

Le fait que l'on fasse une preuve par récurrence s'exprime de la façon suivante: pour toute dérivation δ plus petite que D, si δ prouve $s_\delta \vdash e_\delta \to n_\delta$ et si l'on sait prouver par ailleurs $s_\delta \vdash e_\delta \to n'_\delta$, alors on peut en déduire $n_\delta = n'_\delta$.

La règle à la racine de la dérivation D est forcément l'une des quatre règles de la définition sémantique. Nous pouvons utiliser cette information pour raisonner par cas:

1. Si D est construite avec la règle (1), alors les variables s, et e sont forcément de la forme suivante:

$$\begin{aligned} s &= (v, n) \cdot s' \\ e &= v \end{aligned}$$

   Dans ce cas la seconde dérivation prouvant $s \vdash e \to n'$ peut seulement utiliser la règle (1) également. En effet, les autres cas sont impossibles: la règle (2) ne s'applique que si la variable en première composante du couple en tête de l'environnement (ici v) est différente de l'expression évaluée, mais ici l'expression évaluée est v; la règle (4) n'est applicable que si l'expression évaluée est une addition (mais ici c'est une variable); la règle (3) n'est applicable que si l'expression évaluée est un entier. Comme la règle utilisée pour prouver $(v, n) \vdash v \to n'$ est forcément la règle (1), on a forcément $n = n'$.

2. Si D est construite avec la règle (2), alors les variables s et e sont forcément

de la forme suivante:

$$\begin{aligned} s &= (v', n_1) \cdot s' \\ e &= v \end{aligned}$$

et il existe forcément des dérivations pour les prémisses de cette règle. Donc il existe une dérivation D' pour prouver $s' \vdash v \to n$ et on sait également que $v \neq v'$ est vrai. A cause de cela, l'énoncé $s \vdash e \to n'$ est également de la forme $(v', n_1) \cdot s' \vdash v \to n'$. Cet énoncé ne peut pas avoir été prouvé par la règle (1) car il faudrait alors que l'on ait $v = v'$. Donc seulement la règle 2 peut avoir été utilisée et il existe forcément une preuve de $s' \vdash v \to n'$.

Notons que la dérivation D' est une sous-dérivation de la dérivation D, c'est donc une dérivation de taille plus petite, qui prouve $s' \vdash v \to n$, par ailleurs nous savons qu'il existe une preuve de $s' \vdash v \to n'$. En appliquant l'hypothèse de récurrence, nous pouvons en déduire que $n = n'$.

3. Si la dérivation D est obtenue en appliquant la règle (3), alors l'expression e est l'entier n. La preuve de $s \vdash e \to n'$ s'écrit également $s \vdash n \to n'$ et peut seulement avoir été prouvée avec la règle (3), donc $n = n'$.

4. S la dérivation D est obtenue en appliquant la règle (4), alors il existe forcément deux expressions e_1 et e_2 telles que $e = plus(e_1, e_2)$ et deux nombres n_1 et n_2 tels que $n = n_1 + n_2$ et les hypothèses de la règle $s \vdash e_1 \to n_1$ et $s \vdash e_2 \to n_2$ doivent avoir été prouvées par deux dérivations D_1 et D_2. Par ailleurs, puisque e est alors une addition, l'énoncé $s \vdash e \to n'$ peut seulement avoir été prouvé par la règle 4 et il doit exister deux nombres n'_1 et n'_2 tels que $n' = n'_1 + n'_2$ et tels que les énoncés $s \vdash e_1 \to n'_1$ et $s \vdash e_2 \to n'_2$ soient prouvables. On peut alors utiliser l'hypothèse de récurrence deux fois, une fois pour D_1 et la preuve de $s \vdash e_1 \to n'_1$ pour déduire que $n_1 = n'_1$ et une fois pour D_2 et la preuve de $s \vdash e_2 \to n'_2$ pour déduire que $n_2 = n'_2$. On peut donc conclure que $n = n'$ dans ce cas également. Ceci termine notre preuve.

5.4 Raisonnement sur les dérivation en Coq