Introduction à la Sémantique Opérationnelle

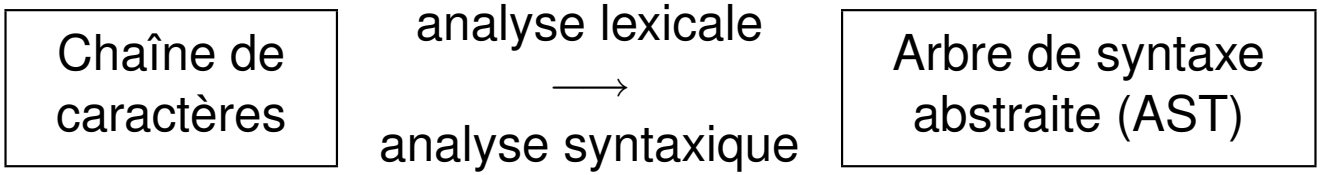
(à grands pas)

M.V. Aponte

Année 2007/2008

Sémantique : Qu'est-ce ?

"Sémantique : Etude du sens des unités linguistiques et de leur composition." [Petit Larousse, 1994]



Associer une "signification" à un programme à partir de son AST.

Sémantique : Qu'est-ce ?

Sémantique d'un langage : définition précise, non ambiguë et indépendante de l'implantation, de la "signification" des constructions de ce langage

- quelle valeur nous décidons de donner à une expression ?
- quel effet nous décidons d'attribuer à une instruction ?

Sémantique formelle d'un langage : exprimée dans un formalisme mathématique

Syntaxe vs Sémantique

- Exemple de la logique:
  - Syntaxe: $\forall x \in N$ *successeur*$(x) \not= 0$
  - Sémantique: *tous les nombres naturels ont un successeur différent de zéro*

- Exemple d'un langage de programmation:
  - Syntaxe:
    `z:=x; x:=y; y:=z;`
    est formé de 3 instructions d'affectation séparées par des ;
  - Sémantique:
    échanger les valeurs des variables x et y en donnant à z la dernière valeur de y

Sémantique : Pourquoi ?

Garantir certaines propriétés vérifiées par les programmes ... augmenter la confiance que l'on peut avoir dans les programmes.

- spécification d'un compilateur pour un langage donné ...
- raisonnement sur les propriétés attendues du langage – comme par exemple le déterminisme de l'exécution des instructions

Sémantique : Pourquoi ?

- raisonnement sur les propriétés des programmes:
  - Equivalence de programmes ... utile pour transformer un programme en un programme équivalent mais plus efficace
  - Terminaison de programmes
  - Non-modification par un programme des valeurs contenues à des adresses sensibles de la mémoire
  - ...
- génération d'outils

Sémantique : Comment ? (1)

Sémantique dynamique : description du comportement (i.e., l'exécution) de tous les programmes, y compris ceux dont l'exécution provoque une "erreur".

Plusieurs manières de formaliser sa description:

- *sémantique opérationnelle*
- *sémantique dénotationnelle*
- *sémantique axiomatique*

Sémantique opérationnelle

Le sens d'une construction est donné par les calculs induits lors de son exécution par une machine. On s'intéresse à la *manière* dont se font les calculs.

$\Rightarrow$ *Le sens d'un programme est donné par la séquence d'états succéssifs d'une machine qui l'exécute.*

Programme = Système de transitions entre états

Sémantique opérationnelle(2)

Exemple:

P1: `a:=1; b:=0;`

P2: `a:=1 /* un commentaire */; b:=0;`

P3: `b:=0; a:=1;`

P1 et P2 sont (du point de vue de la sémantique opérationnelle) sémantiquement équivalents.
P1 n'est pas équivalent à P3.

Remarque: Deux programmes qui ne sont pas opérationnellement équivalents peuvent aboutir au même état final.

Sémantique Dénotationnelle

(vision fonctionnelle)

Le sens d'une construction est modélisé par un objet mathématique qui représente l'effet de son exécution sur l'état du programme.
programme = fonction mathématique

On s'intéresse à l'effet d'un programme et non à la manière de l'exécuter.
non développé ici

Sémantique Dénotationnelle(2)

Exemple: P1, P2 et P3 comme avant.

P4: `a:=1; b:=1;`

P1, P2 et P3 sont dénotationnellement équivalents, mais ils ne sont pas équivalents à P4.

Exercice: Quelles équivalences pour ces trois programmes?

P1: `z:=x; x:=y; y:=z;`
P2: `z:=y; y:=x; x:=z;`
P3: `z:=y; y:=x; x:=z; z:=y;`

Sémantique Axiomatique

Des propriétés spécifiques portant sur l'effet d'exécuter un programme sont inserées sous forme d'assertions.

Certains aspects de l'exécution sont déliberemment ignorés: on s'intéresse aux *propriétés de correction partielle* par rapport à des *préconditions* et *post-conditions*:

si l'état initial d'un programme satisfait la précondition, la propriété de correction partielle garantit que l'état final satisfait la postcondition.

Sémantique Axiomatique(2)

programme = “transformateur” de propriétés logiques

$$\{P\}\texttt{Prog}\{Q\}$$

Exemple:

$\{\mathit{vrai}\}\texttt{a} := 1;\texttt{b} := 0;\{a \geq 0 \vee b \geq 0\}$
$\{\mathit{vrai}\}\texttt{b} := 0;\texttt{a} := 1;\{a \geq 0 \vee b \geq 0\}$
$\{\mathit{vrai}\}\texttt{a} := 1;\texttt{b} := 1;\{a \geq 0 \vee b \geq 0\}$

sont tous équivalents.

Sémantiques Opérationnelles

Deux sortes de formulations:

- Sémantique opérationnelle à petits pas: description assez fine de l'exécution. Non étudiée ici.

- Sémantique naturelle ou à grands pas: cache un peu plus de détails de l'exécution.

Composantes d'une Sémantique Opérationnelle

(a) *Une syntaxe*: ce sont les constructions textuelles auxquelles on attachera un sens.

(b) *Une propriété sémantique particulière*: que l'on veut caractériser pour la syntaxe.

(c) *Un calcul*: une description de la manière de calculer la propriété sémantique pour n'importe quelle construction de la syntaxe.

Composantes: une syntaxe

Les constructions textuelles auxquelles on veut attacher un sens.

Exemple: le langage des expressions arithmétiques sur les constantes et variables entières: 3, x+7, ...

$$
\begin{array}{lcl}
eA & := & num_n \mid \mathrm{x} \\
 & \mid & eA_1 + eA_2 \mid eA_1 - eA_2 \\
 & \mid & eA_1 * eA_2 \mid eA_1 / eA_2
\end{array}
$$

num_n et x sont des *méta-variables* pour les numéraux et les variables. Autrement dit, num_n représente n'importe quel numéral du langage décrit par la grammaire de numéraux, alors que x représente un identificateur quelconque décrit par la grammaire des variables.

Composantes: une propriété sémantique

C'est la propriété particulière que l'on veut caractériser pour la syntaxe.

Exemples:

- *Valeur*: d'une expression arithmétique quelconque.
- *Type*: d'une expression arithmétique quelconque.
- *Nombre d'accès mémoire*: d'une expression arithmétique quelconque.
- etc.

Composantes: un calcul

Une description de la manière de calculer la propriété sémantique pour n'importe quelle construction de la syntaxe.

Exemples:

- *Valeur*:
  - Valeur d'un numéral num_n: le nombre entier n correspondant,
  - Valeur d'une variable x: sa valeur en mémoire,
  - etc..
- *Type* d'une constante num_n: int, etc...
- *Nombre d'accès mémoire*: d'une constante n: 0, etc...

Pour que cette description soit complète, on doit prévoir au moins un cas de calcul de la propriété, pour chaque cas de la syntaxe.

Systèmes d'inférence

Dans une sémantique opérationnelle, les "calculs" des propriétés sont décrits avec des *systèmes d'inférence.*

Ils servent à déduire une propriété particulière pour les constructions syntaxiques d'un langage. Les déductions dans ces systèmes sont appelées *jugements.* Ils correspondent au résultat déduit pour la propriétée étudiée.

Un jugement

Un jugement a la forme:

$$< C, \Gamma > \rightsquigarrow r$$

C est la construction syntaxique traitée, Γ est un ensemble d'hypothèses sur les variables dans C, et r est le résultat obtenu pour la propriété étudiée.

On peut lire ce juegement: *le résultat de la propriétée étudiée pour la construction C et suivant les hypothèses dans* Γ *est r.*

Jugements

Exemples:
Si la syntaxe étudiée est celle des expressions arithmétiques, et la propriété à caractériser, leur valeur, la valeur de la constante 3 est établie par le jugement:

$$< 3, \Gamma > \rightsquigarrow 3$$

Exemples de jugements

Supossons que Γ représente l'ensemble des valeurs en mémoire pour chacune des variables dans l'expression étudiée. Si $\Gamma = [x \mapsto 5]$, alors

$$< x + 3, \Gamma > \rightsquigarrow 8$$

Exemple: Si la propriété étudié est le type, on pourrait établir des jugements tels que:

$$< 3, \Gamma > \rightsquigarrow \mathit{int}$$

ou

$$< x + 3, [x \mapsto \mathit{float}] > \rightsquigarrow \mathit{float}$$

Systèmes d'inférence

Composé de:

- *Un ensemble de règles d'inférence*: ce sont des jugements avec premises: ils sont vrais si toutes les premises sont vraies.

- *Un ensemble d'axiomes*: ce sont des jugements sans premises: ils sont toujours vrais.

Règles d'inférence

Ce sont des jugements avec premises: ils sont vrais si toutes les premises sont vraies.

$$(R)\frac{j_1 \quad j_2 \quad \cdots \quad j_n}{j}$$

R est le nom de la règle
$\{j_1, \cdots, j_n\}$: *prémises* qui doivent être "satisfaites" pour que le *jugement conclusion* j soit satisfait.

Axiomes

Ce sont des jugements sans premises: ils sont toujours vrais.

$$(A)\ \frac{}{j}$$

A est le nom de l'axiome
Le jugement j est toujours satisfait.

Exemple: sémantique des expressions arithmétiques

La syntaxe: (rappel)

$$\begin{array}{lcl} eA & := & num_n \mid \mathrm{x} \\ & \mid & eA_1 + eA_2 \mid eA_1 - eA_2 \\ & \mid & eA_1 * eA_2 \mid eA_1 / eA_2 \end{array}$$

E_A est l'ensemble des termes de ce langage.

Les jugements (en francais):

le "sens" d'une expression eA *est celle de son résultat numérique. Plus précisement, du résultat des opérations qu'elle énonce, en tenant compte des valeurs des variables en mémoire au moment de l'évaluation*.

évaluer *eA* dans l'état courant de la mémoire $\rightsquigarrow$ n

Un état de la mémoire

Il donne pour chaque variable, sa valeur courante. On le modélise par une fonction (partielle) des variables vers leurs valeurs:

$$\sigma : \mathit{Var} \to \mathcal{Z}$$

La valeur d'une variable x dans un état σ est noté $\sigma(x)$.

Lorsque seules quelques variables nous intéressent, nous noterons $\sigma = [x \mapsto 5, y \mapsto 2]$ pour l'état σ où $\sigma(x) = 5$, et $\sigma(y) = 2$, et où la valeur pour tout autre variable nous est indifférente.

Jugement pour l'évaluation d'une expression

Evaluer une expression $a \in E_A$ dans l'état mémoire σ donne le résultat r:

$$\langle a, \sigma \rangle \rightsquigarrow r$$

Exemple:

$$\langle x + 2, [x \mapsto 5] \rangle \rightsquigarrow 7$$

Le couple $\langle a, \sigma \rangle$ formé d'une construction de la syntaxe et d'un état d'évaluation est nommé *configuration*.

Règles d'évaluation

$$(A_n)\ \frac{}{\langle num_n, \sigma\rangle \rightsquigarrow n}\ (n \in \mathbb{Z}) \qquad (V_{ar})\ \frac{}{\langle x, \sigma\rangle \rightsquigarrow \sigma(x)}\ (x \in dom(\sigma))$$

$$(A_{dd})\ \frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 + a_2, \sigma\rangle \rightsquigarrow n_1 + n_2} \qquad (M_{ul})\ \frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 \times a_2, \sigma\rangle \rightsquigarrow n_1 \times n_2}$$

$$(S_{ous})\ \frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 - a_2, \sigma\rangle \rightsquigarrow n_1 - n_2} \qquad (D_{iv})\ \frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 / a_2, \sigma\rangle \rightsquigarrow n_1 / n_2}$$

$a_1, a_2, \ldots \in E_A$

Condition dans toutes les règles: $(n, n_1, n_2 \ldots \in \mathbb{Z})$

Meta-variables et instances

Dans

$$(A_{dd})\frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 + a_2, \sigma\rangle \rightsquigarrow n_1 + n_2}$$

a_1 et a_2 sont des *méta-variables* pouvant être remplacées par des expressions quelconques du langage E_A. Idem pour n_1, σ, ...

Lorsque l'on remplace les méta-variables d'une règle par des termes du langage, on obtient une *instance* de la règle.

$$(A_{dd})\frac{\langle x, [x \mapsto 5]\rangle \rightsquigarrow 5 \quad \langle 2, [x \mapsto 5]\rangle \rightsquigarrow 2}{\langle x + 2, [x \mapsto 5]\rangle \rightsquigarrow 5 + 2}$$

est une instance de la règle Add.

Interprétation des numéraux et symboles

Dans les règles:

$$(A_n)\frac{}{\langle num_n, \sigma\rangle \rightsquigarrow n}\ (n \in \mathbb{Z}) \qquad (A_{dd})\frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 + a_2, \sigma\rangle \rightsquigarrow n_1 + n_2}$$

num_n et $a_1 + a_2$ sont des morceaux de syntaxe, alors que n, n_1, n_2 sont des valeurs entières. La différence est importante.

D'un coté, nous avons du texte, de l'autre, des objets mathématiques correspodant au *sens, ou interprétation* pour ce texte.

Interprétation des numéraux et symboles(2)

Il est possible de rendre compte de cette distinction de manière explicite au sein de nos règles.

La *fonction d'interprétation* $[\![]\!]$ prend en argument un morceau de la syntaxe et donne en résultat l'objet mathématique qui correspond à son interprétation:

- $[\![num_n]\!]$ donne en résultat l'entier n correspondant au numéral num_n,
- $[\![+]\!]$ désigne l'opérateur d'addition habituel sur les entiers,
- ...

Interprétation des numéraux et symboles(3)

Nous pouvons alors énoncer des règles plus précises:

$$(A_n)\frac{[\![num_n]\!] = n}{\langle num_n, \sigma\rangle \rightsquigarrow n}\ (n \in \mathbb{Z}) \qquad (A_{dd})\frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma\rangle \rightsquigarrow n_2}{\langle a_1 + a_2, \sigma\rangle \rightsquigarrow n_1 [\![+]\!] n_2}$$

Cela alourdit considérablement les notations. En général, après avoir sousligné la différence entre un morceau de syntaxe '+' et son interprétation $[\![+]\!]$, on préfère laisser cette distinction implicite dans les notations.

Nous ne changeons doc pas nos règles...

Exemple d'évaluation

Avec ce système d'inférence, et sachant que σ est t.q. $\sigma(x) = 2$, pouvons nous conclure $\langle (2 \times 3) + x, \sigma \rangle \rightsquigarrow 8$?

Nous allons construire un *arbre de dérivation* pour ce jugement:

- la racine: est le jugement que nous souhaitons obtenir en conclusion $\langle (2 \times 3) + x, \sigma \rangle \rightsquigarrow 8$,
- les noeuds: sont les jugements premisses, ainsi que les instances des règles nous permettant d'obtenir chaque conclusion intermédiaire ou finale.
- les feuilles: sont nécessairement des instances d'axiomes.

Arbre de dérivation d'une évaluation

σ : état tel que $\sigma(x) = 2$
$\langle (2 \times 3) + x, \sigma \rangle \rightsquigarrow 8$?

$$(A_{dd})\dfrac{(M_{ul})\dfrac{(A_n)\dfrac{}{\langle 2, \sigma \rangle \rightsquigarrow 2} \quad (A_n)\dfrac{}{\langle 3, \sigma \rangle \rightsquigarrow 3}}{\langle 2 \times 3, \sigma \rangle \rightsquigarrow 6} \quad (V_{ar})\dfrac{}{\langle x, \sigma \rangle \rightsquigarrow 2}}{\langle (2 \times 3) + x, \sigma \rangle \rightsquigarrow 8}$$

Chaque noeud (jugement) a pour fils les jugments qui lui sont nécessaires en tant que premisses. Il est étiquetté par le nom de la règle ayant permi d'obtenir ce jugement. Les feuilles sont des instances d'axiomes.

Arbre de dérivation = Preuve

Cet arbre de dérivation est une

preuve de validité du jugement $\langle (2 \times 3) + x, [x \mapsto 2] \rangle \rightsquigarrow 8$

dans le système d'inférence que nous avons énoncé.

Conclusion : $\langle (2 \times 3) + x, [x \mapsto 2] \rangle \rightsquigarrow 8$ est un théorème pour la sémantique donnée.

Et les jugements qui ne sont pas des théorèmes? $\Rightarrow$ Ceux pour lesquels il n'existe pas d'arbre de dérivation permettant de les conclure. Exemple: $\langle (2 + 3), \sigma] \rangle \rightsquigarrow 8$

Sémantique opérationnelle d'évaluation à grands pas

Décrire le processus d'évaluation à l'aide de systèmes d'inférence dont les règles portent sur des jugements exprimant qu'une valeur $v \in \mathbb{Z}$ est le résultat de l'évaluation d'une expression $a \in E_A$ étant donnée une valuation $\sigma \in \mathcal{V}[\mathbb{Z}]$:

$$\langle a, \sigma \rangle \rightsquigarrow v$$

Caractériser un sous-ensemble des jugements de la forme $\langle a, \sigma \rangle \rightsquigarrow v$ qui sont "corrects d'un point de vue sémantique" : il s'agit de l'ensemble des théorèmes d'un système d'inférence.

Exemples

$\langle 3 + 2, \sigma \rangle \rightsquigarrow 5$ est un théorème

$\langle 3 + 2, \sigma \rangle \rightsquigarrow 1$ n'est pas un théorème, il s'agit d'un jugement "syntaxiquement" correct mais "sémantiquement" erroné

Notre sémantique opérationnelle: bilan

Ce que nous avons fait:

- caractériser les *configurations* pour les expressions arithmétiques “sémantiquement correctes du point de vue de leur évaluation”, i.e., pouvant être évaluées,
- caractérisé leur valeur,
- imposé un ordre d’évaluation pour leurs sous-expressions

Nous obtenons une description précise de la valeur de n’importe quelle configuration d’expression arithmétique sémantiquement correcte. Nous pouvons en déduire un algorithme d’évaluation.

Notre sémantique opérationnelle: bilan (2)

Ce que nous n'avons pas fait:

- caractériser de facon précise les configurations érronées: que se passe-t-il si nous ne sommes pas capables d'obtenir une dérivation, est-ce parce qu'elle n'existe pas, ou parce que nous ne savons pas faire?
- caractérisér le "résultat obtenu" si l'on tente d'évaluer une configuration érronée.

Notre algorithme sait-il décider si une configuration n'est pas sémantiquement correcte? Que se passe-t-il si cela survient au milieu d'une évaluation?

$\Rightarrow$ Nous devrons ajouter les erreurs!

Une propriété dans notre sémantique: l'équivalence

Caractériser les expressions arithmétiques sémantiquement équivalentes: qui s'évaluent à la même valeur.

$$a_1 \sim a_2 \Leftrightarrow (\forall v \in \mathbb{VA}\ \forall \sigma \in \mathcal{V}[\mathbb{Z}] \quad \langle a_1, \sigma \rangle \rightsquigarrow v \Leftrightarrow \langle a_2, \sigma \rangle \rightsquigarrow v)$$

a_1 et a_2 sont équivalentes si indépendamment du contexte d'évaluation elles s'évaluent dans une même valeur.

Exemple: $x + x \sim 2 \times x$

Technique de preuve

Comme pour toute équivalence, nous devons montrer deux énoncés: si nous supposons valide chaque membre de l'équivalence, cela nous permet d'en déduire l'autre.

Nous devons donc montrer que pour tout état σ les deux énoncés suivants sont valides:

(1) Si $\langle x + x, \sigma \rangle \rightsquigarrow n$ alors $\langle 2 \times x, \sigma \rangle \rightsquigarrow n$,

(2) Si $\langle 2 \times x, \sigma \rangle \rightsquigarrow n$, alors $\langle x + x, \sigma \rangle \rightsquigarrow n$.

Technique de preuve (2)

Pour montrer (1) Si $\langle x + x, \sigma \rangle \rightsquigarrow n$ alors $\langle 2 \times x, \sigma \rangle \rightsquigarrow n$,

- nous supposons valide $\langle x + x, \sigma \rangle \rightsquigarrow n$,
- nous construisons un arbre de preuve pour ce jugement,
- nous réutilisons des sous-arbres de cette dérivation afin d'obtenir une preuve (dérivation) pour $\langle 2 \times x, \sigma \rangle \rightsquigarrow n$
- et nous laissons la preuve du réciproque en exercice...

Expressions arithmétiques équivalentes

Si (1) : $\langle x + x, \sigma \rangle \rightsquigarrow n$, alors:

$$(A_{dd})\frac{(V_{ar})\dfrac{}{\langle x, \sigma \rangle \rightsquigarrow \sigma(x)} \quad (V_{ar})\dfrac{}{\langle x, \sigma \rangle \rightsquigarrow \sigma(x)}}{\langle x + x, \sigma \rangle \rightsquigarrow \sigma(x) + \sigma(x)}$$

où $\sigma(x) + \sigma(x) = n$. En réutilisant le sous-arbre (Var), on peut construire l'arbre :

$$(M_{ul})\frac{(A_n)\dfrac{}{\langle 2, \sigma \rangle \rightsquigarrow 2} \quad (V_{ar})\dfrac{}{\langle x, \sigma \rangle \rightsquigarrow \sigma(x)}}{\langle 2 \times x, \sigma \rangle \rightsquigarrow 2 \times \sigma(x)}$$

qui est une preuve pour $\langle 2 \times x, \sigma \rangle \rightsquigarrow 2 \times \sigma(x)$. Mais, puisque $2 \times \sigma(x) = \sigma(x) + \sigma(x) = n$, on a bien une preuve de $\langle 2 \times x, \sigma \rangle \rightsquigarrow n$.

La réciproque (2) est laissée en exercice.

Un mini-langage impératif: While

Syntaxe:

$$c \;:=\; \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } (b) \text{ do } c$$

$$a \;:=\; num_n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid (a_1)$$

$$b \;:=\; \text{true} \mid \text{false} \mid a_1 \leq a_2 \mid a_1 = a_2 \mid \text{not } b_1 \mid b_1 \text{ or } b_2 \mid b_1 \text{ and } b_2$$

$a, a_1 \ldots \in E_A$ désignent les expressions arithmétiques
$b, b_1 \ldots \in E_B$ ensembles des expressions boléennes
$c, c_1 \ldots \in E_C$ ensemble des instructions du langage While

Exemples pour la syntaxe

Instruction de calcul du PGCD:

while $(\text{not } (x = y))$ do if $x \leq y$ then $y := y - x$ else $x := x - y$

Expression boléenne: $\text{not } (x = 0 \text{ and } x \leq (7/z))$

Sémantique des expressions booléennes

Même démarche qu'avec les expressions arithmétiques.

Les jugements: $\langle b, \sigma \rangle \Rightarrow t$ où:

- $b \in E_B$ est une construction syntaxique pour les expressions boléennes,
- $t \in \mathcal{B}$ est une valeur de vérité dans l'ensemble $\mathcal{B} = \{true, false\}$

et qu'on lit: *une expression booléenne b s'évalue en une valeur $t \in \mathcal{B}$ étant donnée une valuation σ*.

Interprétation des constantes et opérateurs

Interpréter une expression booléenne c'est lui donner une valeur appartenant à $\mathcal{B}$ avec $\mathcal{B} = \{\text{true}, \text{false}\}$.

$t \in \mathbb{B}$ est interprété par lui-même : $[\![\text{true}]\!] = \text{true}$ et $[\![\text{false}]\!] = \text{false}$

$\{=, \leq\}$ interprétés par les opérateurs habituels.

$[\![\text{ or }]\!]_{=} \wedge$
$[\![\text{ and }]\!]_{=} \vee$
$[\![\text{not }]\!]_{=} \neg$

Système d'inférence

$$(B_=)\frac{\langle a_1,\sigma\rangle \rightsquigarrow n_1 \quad \langle a_2,\sigma\rangle \rightsquigarrow n_2}{\langle a_1 = a_2,\sigma\rangle \Rightarrow (n_1 = n_2)} \qquad (B_\leq)\frac{\langle a_1,\sigma\rangle \rightsquigarrow n_1 \quad \langle a_2,\sigma\rangle \rightsquigarrow n_2}{\langle a_1 \leq a_2,\sigma\rangle \Rightarrow (n_1 \leq n_2)}$$

$$(B_c)\frac{}{\langle t,\sigma\rangle \Rightarrow t} \quad t \in \mathcal{B} \qquad (B_\wedge)\frac{\langle b_1,\sigma\rangle \Rightarrow v_1 \quad \langle b_2,\sigma\rangle \Rightarrow v_2}{\langle b_1 \text{ and } b_2,\sigma\rangle \Rightarrow v_1 \wedge v_2}$$

$$(B_\vee)\frac{\langle b_1,\sigma\rangle \Rightarrow v_1 \quad \langle b_2,\sigma\rangle \Rightarrow v_2}{\langle b_1 \text{ or } b_2,\sigma\rangle \Rightarrow v_1 \vee v_2} \qquad (B_\neg)\frac{\langle b,\sigma\rangle \Rightarrow t}{\langle \text{not } b,\sigma\rangle \Rightarrow (\neg t)}$$

Pour toutes les règles avec occurrences de n_1, n_2, : $n_1, n_2 \in \mathbb{Z}$

Expressions et changements d'états

Nous avons défini les états:

$$\sigma : \mathit{Var} \rightarrow \mathcal{Z}$$

afin de donner des valeurs aux variables entières présentes dans une expression:

Soit $\sigma = [x \mapsto 5]$ alors $\langle x + 2, \sigma \rangle \rightsquigarrow 7$

Si nous retraçons en rouge l'évolution des états:

$$[x \mapsto 5] \Rightarrow x + 2 \rightsquigarrow 7 \Rightarrow [x \mapsto 5]$$

- *l'état avant évaluation* de l'expression est $\sigma = [x \mapsto 5]$
- *l'état après évaluation* ne change pas.

Expressions, instructions et changements d'état

Effet d'une instruction: Un programme impératif réalise des affectations: les valeurs des variables en mémoire sont modifiés:

(état avant) $\sigma = [x \mapsto 5] \Rightarrow x := x+1 \Rightarrow [x \mapsto 6] = \sigma_1$ (état après)

$\Rightarrow$ état avant évaluation $\not=$ état après évaluation

Effet d'une expression: ne change pas l'état courant de la mémoire: *elle l'inspecte*.

(état avant) $[x \mapsto 5] \Rightarrow x + 1 \rightsquigarrow 7 \Rightarrow [x \mapsto 5]$ (état après)

Instructions et changements d'état

Le rôle d'un programme c est de modifier l'état courant σ_1 de la mémoire en un état σ_2 :

$$\langle c, \sigma_1 \rangle \rightarrow \sigma_2$$

L'exécution de l'instruction c dans l'état σ_1 conduit à l'état σ_2.

Nous décrivons l'exécution d'une instruction *par le changement d'état qu'elle provoque*.

Changement d'état: $\sigma[x \leftarrow n](y) = \begin{cases} n & \text{si } y = x \\ \sigma(y) & \text{sinon} \end{cases}$

Exemple: $\langle x := 0, \sigma \rangle \rightarrow \sigma[x \leftarrow 0]$

Sémantique opérationnelle de While: jugements

- *le sens d'une expression* est celui de caulculer une valeur. Pour cela on peut avoir besoin d'inspecter l'état des variables,

- *le sens* d'une instruction de While est de (possiblement) modifier l'état courant de la mémoire.

Les jugements pour une sémantique de While ne cherchent donc pas à calculer une valeur, mais *un nouvel état*.

$$\langle c, \sigma_1 \rangle \rightarrow \sigma_2$$

où c est une instruction de While.

Règles pour une sémantique de While

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

$$(C_3)\ \frac{\langle c_1, \sigma\rangle \to \sigma_1 \quad \langle c_2, \sigma_1\rangle \to \sigma_2}{\langle c_1 ; c_2, \sigma\rangle \to \sigma_2}$$

$$(C_4)\ \frac{\langle b, \sigma\rangle \Rightarrow \text{true} \quad \langle c_1, \sigma\rangle \to \sigma_1}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \to \sigma_1}$$

$$(C_5)\ \frac{\langle b, \sigma\rangle \Rightarrow \text{false} \quad \langle c_2, \sigma\rangle \to \sigma_2}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \to \sigma_2} \qquad (C_6)\ \frac{\langle b, \sigma\rangle \Rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma\rangle \to \sigma}$$

$$(C_7)\ \frac{\langle b, \sigma\rangle \Rightarrow \text{true} \quad \langle c, \sigma\rangle \to \sigma_1 \quad \langle \text{while } b \text{ do } c, \sigma_1\rangle \to \sigma_2}{\langle \text{while } b \text{ do } c, \sigma\rangle \to \sigma_2}$$

Sémantique opérationnelle à grands pas : exemple

$$(C_3)\dfrac{(C_2)\dfrac{(A_n)\dfrac{}{\langle 0,\sigma\rangle \rightsquigarrow 0}}{\langle x := 0,\sigma\rangle \to \sigma_1}\quad (C_7)\dfrac{D_1 \quad D_2 \quad D_3}{\langle \text{while } x \leq 0 \text{ do } x := x+1, \sigma_1\rangle \to \sigma_2}}{\langle x := 0 \text{ ; while } x \leq 0 \text{ do } x := x+1, \sigma\rangle \to \sigma_2}$$

où $\sigma_1 = \sigma[x \leftarrow 0]$, $\sigma_2 = \sigma_1[x \leftarrow 1]$. D_1 et D_2 sont des arbres de dérivation pour $\langle x \leq 0, \sigma_1\rangle \Rightarrow \text{true}$ et $\langle x := x+1, \sigma_1\rangle \to \sigma_2$.
D_3 est l'arbre :

$$(C_6)\dfrac{(B_{\leq})\dfrac{(V_{ar})\dfrac{}{\langle x, \sigma_2\rangle \rightsquigarrow 1} \quad (A_n)\dfrac{}{\langle 0, \sigma_2\rangle \rightsquigarrow 0}}{\langle x \leq 0, \sigma_2\rangle \Rightarrow \text{false}}}{\langle \text{while } x \leq 0 \text{ do } x := x+1, \sigma_2\rangle \to \sigma_2}$$

Effets de bords lors de l'évaluation d'expressions

La règle:

$$(C_4)\frac{\langle b, \sigma\rangle \Rightarrow \text{true} \quad \langle c_1, \sigma\rangle \rightarrow \sigma_1}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \sigma_1}$$

ainsi que (C_5) ne sont correctes que si l'évaluation d'une expression booléenne ne modifie pas l'état dans lequel s'effectue l'évaluation.

Est-ce le cas avec le langage C? Non:

```
if (i++ > 0) i=i-1 else i=i+1;
```

... même remarque pour toutes les règles nécessitant l'évaluation d'une expression.

Programmes équivalents

Deux programmes c_1, c_2 sont équivalents, noté $c_1 \sim c_2$ si leur exécution à partir d'un même état initial aboutît dans un état final identique.

$\forall c_1, c_2 \in E_C$:

$$c_1 \sim c_2 \Leftrightarrow (\forall \sigma, \sigma' \in \mathcal{V}[\mathbb{Z}] \quad \langle c_1, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle c_2, \sigma \rangle \rightarrow \sigma')$$

Exemples :

- if b then I else $I \sim I$
- if b then I else *skip* $\not\sim I$

Programmes équivalents: exemples(1)

$c; (\text{if } b \text{ then } c' \text{ else } c'') \overset{?}{\sim} \text{if } b \text{ then } (c; c') \text{ else } (c; c'')$

Non. Voici un contre-exemple:

c	$x := 0$
c'	$x := x + 1$
c''	$x := x + 5$
b	$1 \leq x$
σ	$\sigma(x) = 2$

$\langle c; (\text{if } b \text{ then } c' \text{ else } c''), \sigma \rangle \rightarrow \sigma'$	$\sigma'(x) = 5$
$\langle \text{if } b \text{ then } (c; c') \text{ else } (c; c''), \sigma \rangle \rightarrow \sigma''$	$\sigma''(x) = 1$

Programmes équivalents : exemples (2)

Si:

$$c_1 : \quad (\text{if } b \text{ then } c \text{ else } c'); c''$$
$$c_2 : \quad \text{if } b \text{ then } (c; c'') \text{ else } (c'; c'')$$

$c_1 \overset{?}{\sim} c_2$

Pour σ, σ' quelconques, nous construisons un arbre d'inférence de $\langle c_1, \sigma \rangle \to \sigma'$ à partir d'un arbre de d'inférence de $\langle c_2, \sigma \rangle \to \sigma'$ et *vice versa*.

Programmes équivalents : exemples (3)

Si on dispose d'un arbre d'inférence de $\langle(\text{if } b \text{ then } c \text{ else } c'); c'', \sigma\rangle \to \sigma'$, il a forcément été obtenu à partir de la règle C_3 :

$$(C_3)\dfrac{(C_i)\dfrac{\vdots}{\langle \text{if } b \text{ then } c \text{ else } c', \sigma\rangle \to \sigma_1} \quad (C_j)\dfrac{\vdots}{\langle c'', \sigma_1\rangle \to \sigma'}}{\langle(\text{if } b \text{ then } c \text{ else } c'); c'', \sigma\rangle \to \sigma'}$$

On distingue alors deux cas (résultat de l'évaluation de b).

Programmes équivalents : exemples (4)

(1) Si b s'évalue à true, alors, à partir de l'arbre :

$$(C_3)\dfrac{(C_4)\dfrac{(B_i)\dfrac{\vdots}{\langle b,\sigma\rangle \rightsquigarrow \text{true}}\quad (C_j)\dfrac{\vdots}{\langle c,\sigma\rangle \rightarrow \sigma_1}}{\langle \text{if } b \text{ then } c \text{ else } c',\sigma\rangle \rightarrow \sigma_1}\quad (C_k)\dfrac{\vdots}{\langle c'',\sigma_1\rangle \rightarrow \sigma'}}{\langle (\text{if } b \text{ then } c \text{ else } c');c'',\sigma\rangle \rightarrow \sigma'}$$

on peut obtenir l'arbre :

$$(C_4)\dfrac{(B_i)\dfrac{\vdots}{\langle b,\sigma\rangle \rightsquigarrow \text{true}}\quad (C_3)\dfrac{(C_j)\dfrac{\vdots}{\langle c,\sigma\rangle \rightarrow \sigma_1}\quad (C_k)\dfrac{\vdots}{\langle c'',\sigma_1\rangle \rightarrow \sigma'}}{\langle c;c'',\sigma\rangle \rightarrow \sigma'}}{\langle \text{if } b \text{ then } (c;c'') \text{ else } (c';c''),\sigma\rangle \rightarrow \sigma'}$$

et *vice versa*.

Programmes équivalents : exemples (5)

(2) Si b s'évalue à false, alors, à partir de l'arbre :

$$(C_3)\dfrac{(C_5)\dfrac{(B_i)\dfrac{\vdots}{\langle b,\sigma\rangle \rightsquigarrow \text{false}}\quad (C_j)\dfrac{\vdots}{\langle c',\sigma\rangle \to \sigma_1}}{\langle \text{if } b \text{ then } c \text{ else } c',\sigma\rangle \to \sigma_1}\quad (C_k)\dfrac{\vdots}{\langle c'',\sigma_1\rangle \to \sigma'}}{\langle (\text{if } b \text{ then } c \text{ else } c');c'',\sigma\rangle \to \sigma'}$$

on peut obtenir l'arbre :

$$(C_5)\dfrac{(B_i)\dfrac{\vdots}{\langle b,\sigma\rangle \rightsquigarrow \text{false}}\quad (C_3)\dfrac{(C_j)\dfrac{\vdots}{\langle c',\sigma\rangle \to \sigma_1}\quad (C_k)\dfrac{\vdots}{\langle c'',\sigma_1\rangle \to \sigma'}}{\langle c';c'',\sigma\rangle \to \sigma'}}{\langle \text{if } b \text{ then } (c;c'') \text{ else } (c';c''),\sigma\rangle \to \sigma'}$$

et *vice versa*.

Boucles et Terminaison

L'instruction while permet d'écrire des programmes dont l'exécution ne termine pas.

Les arbres d'inférence étant des objets finis, la sémantique opérationnelle à grands pas n'est pas en mesure de rendre compte de l'exécution des programmes qui ne terminent pas ... contrairement à la sémantique opérationnelle à petits pas.

Exemple: while true do skip

$$(C_7)\dfrac{(B_c)\dfrac{}{\langle \text{true}, \sigma\rangle \rightsquigarrow \text{true}}\ (C_1)\dfrac{}{\langle \text{skip}, \sigma\rangle \rightarrow \sigma}\ (C_7)\dfrac{\vdots}{\langle \text{while true do skip}, \sigma\rangle \rightarrow \sigma'}}{\langle \text{while true do skip}, \sigma\rangle \rightarrow \sigma'}$$

Il n'existe pas d'état σ' tel que $\langle \text{while true do skip}, \sigma\rangle \rightarrow \sigma'$.