Sémantique des Langages de Programmation.

Yassine Lakhnech
lakhnech@imag.fr
http://www-verimag.imag.fr/~lakhnech

Cours en Master Sciences, Technologies & Santé
Mention Mathématiques, Informatique
M1 majeure Informatique
UFR IMA
Université Joseph Fourier
Yassine Lakhnech

Motivation

Pourquoi étudier la sémantique d'un LP?
La sémantique est essentielle pour :

- comprendre les programmes
- tester et vérifier les programmes
- écrire et comprendre des spécifications
- écrire des compilateurs
- classifier les langages de programmation

Pourquoi formaliser la sémantique?

Exemple : Liens statiques vs. liens dynamiques

Program Static_Dynamic
var a;
proc $p(x)$;
  var a;
  begin
    $a := x + 1; write(x)$
  end;
  begin
    $a := 0; p(a)$
  end;

Exemple : Liens statiques vs. liens dynamiques

Program Static_Dynamic
var a;
proc $p(x)$;
  var a;
  begin
    $a := x + 1; write(x)$
  end;
  begin
    $a := 0; p(a)$
  end;

Quelle valeur est imprimée?

Exemple : Passage de paramètres

Program value_reference
var a;
proc $p(x)$;
  begin
    $a := x + 1; write(a); write(x)$
  end;
begin
  $a := 2; p(a); write(a)$
end;

Exemple : Passage de paramètres

Program value_reference
var a;
proc $p(x)$;
  begin
    $x := x + 1; write(a); write(x)$
  end;
begin
  $a := 2; p(a); write(a)$
end;

Quelles valeurs sont imprimées?

Exemple : Passage de paramètres

Program value_reference
var a;
proc $p(x)$;
  begin
    $x := x + 1; write(a); write(x)$
  end;
begin
  $a := 2; p(a); write(a)$
end;

2 3 2 , si appel par valeur

3 3 3, si appel par référence

Contenu du cours

- Style de sémantique :
  - Sémantique opérationnelle naturelle
  - Sémantique opérationnelle structurelle
  - Sémantique axiomatique : Logique de Hoare
  - Sémantique dénotationnelle et calcul approché de propriétés.
- Langages considérés :
  - impératif
  - fonctionnel

Le langage **While**

x : variable
S : commande
a : expression arithmétique
b : expression booléenne

$$S \quad ::= \quad x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

Catégories syntaxiques 1

- Chiffres

$$n \in \textbf{Num} = \{0, \cdots, 9\}^+$$

- Variables

$$x \in \textbf{Var}$$

- Expressions arithmétiques

$$a \quad \in \quad \textbf{Aexp}$$
$$a \quad := \quad n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$

Catégories syntaxiques 2

- Expressions booléennes

$$b \quad \in \quad \textbf{Bexp}$$
$$b \quad := \quad \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

- Commandes

$$S \quad \in \quad \textbf{Stm}$$
$$S \quad ::= \quad x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

Syntaxe concrète vs. syntaxe abstraite

- Le terme $S_1; S_2$ représente l'arbre de racine ;, fils gauche l'arbre de S_1 et fils droite celui de S_2.
- Nous utlisons des parenthèses pour lever des ambiguités. Exemple :

$$S_1, (S_2; S_3) \text{ et } (S_1; S_2); S_3$$

Domaines sémantiques

Entiers relatifs : $\mathbb{Z}$

Booléens : $\mathbb{B}$

États :

$$\textbf{State} = \textbf{Var} \to \mathbb{Z}$$

$\sigma[y \mapsto v]$ dénote l'états σ' tel que:

$$\sigma'(x) = \begin{cases} \sigma(x) & \text{si } x \neq y \\ v & \text{sinon} \end{cases}$$

Fonctions sémantiques 1

- Chiffres : entiers relatifs

$$\mathcal{N} : \textbf{Num} \to \mathbb{Z}$$

- Variables : valeur associé par un état

$$\sigma \in State = \textbf{Var} \to \mathbb{Z}$$

- Expressions arithmétiques : dans chaque état une valeur dans $\mathbb{Z}$

$$\mathcal{A} : \textbf{Aexp} \to (\textbf{State} \to \mathbb{Z})$$

Fonctions sémantiques 2

- Expressions booléennes : dans chaque état une valeur dans $\mathbb{B}$

$$\mathcal{B} : \textbf{Bexp} \to (\textbf{State} \to \mathbb{B})$$

- Commandes :

$$\mathcal{S} : \textbf{Stm} \to (\textbf{State} \xrightarrow{part.} \textbf{State})$$

Sémantique des expressions arithmétiques

$$\mathcal{N}(n_1 \cdots n_k) = \Sigma_{i=1}^{k} n_i \cdot 10^{k-i}$$

$$\begin{aligned}
\mathcal{A}[n]\sigma &= \mathcal{N}[n] \\
\mathcal{A}[x]\sigma &= \sigma(x) \\
\mathcal{A}[a_1 + a_2]\sigma &= \mathcal{A}[a_1]\sigma +_I \mathcal{A}[a_2]\sigma \\
\mathcal{A}[a_1 * a_2]\sigma &= \mathcal{A}[a_1]\sigma *_I \mathcal{A}[a_2]\sigma \\
\mathcal{A}[a_1 - a_2]\sigma &= \mathcal{A}[a_1]\sigma -_I \mathcal{A}[a_2]\sigma
\end{aligned}$$

La sémantique des expressions arithmétiques est définie inductivement sur la structure des expressions. C'est une sémantique compositionnelle.

Sémantique des expressions booléennes

Exercice : Définir la sémantique des expressions booléennes.

Sémantique naturelle 1

Idée : Décrire comment le résultat de l'exécution d'un programme est obtenu.
Sémantique décrite par un système d'inférence : axiomes et règles.

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

$$(\text{skip}, \sigma) \to \sigma$$

$$\frac{(S_1, \sigma) \to \sigma', \quad (S_2, \sigma') \to \sigma''}{(S_1; S_2, \sigma) \to \sigma''}$$

Sémantique naturelle 2

Si $\mathcal{B}[b]\sigma = \mathbf{tt}$ alors

$$\frac{(S_1, \sigma) \to \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \to \sigma'}$$

Si $\mathcal{B}[b]\sigma = \mathbf{ff}$ alors

$$\frac{(S_2, \sigma) \to \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \to \sigma'}$$

Si $\mathcal{B}[b]\sigma = \mathbf{tt}$ alors

$$\frac{(S, \sigma) \to \sigma', \quad (\text{while } b \text{ do } S, \sigma') \to \sigma''}{(\text{while } b \text{ do } S, \sigma) \to \sigma''}$$

Sémantique naturelle 3

Si $\mathcal{B}[b]\sigma = \mathbf{tt}$ alors

$$\frac{(S, \sigma) \to \sigma', \quad (\text{while } b \text{ do } S, \sigma') \to \sigma''}{(\text{while } b \text{ do } S, \sigma) \to \sigma''}$$

Si $\mathcal{B}[b]\sigma = \mathbf{ff}$ alors

$$(\text{while } b \text{ do } S, \sigma) \to \sigma$$

Arbre de dérivation

- Les feuilles sont des axiomes
- Les nœuds sont des règles de la sémantique.

Exemple : Calculons la sémantique de:

1. $x := 2;$ while $x > 0$ do $x := x - 1$
2. $x := 2;$ while $x > 0$ do $x := x + 1$

La sémantique naturelle est déterministe

Théorème Pour toute commande $S \in$ **Stm**, pour tout états σ, σ' et σ'' :

1. Si $(S, \sigma) \to \sigma'$ et $(S, \sigma) \to \sigma''$ alors $\sigma' = \sigma''$.
2. Si $(S, \sigma) \to \sigma'$ alors il n'existe pas d'arbre de dérivation infini.

Preuve Preuve par induction sur la structure de l'arbre de dérivation. □

La fonction sémantique $\mathcal{S}_{ns}$

$$\mathcal{S}_{ns}[S]\sigma = \begin{cases} \sigma' & ; \text{ Si } (S, \sigma) \to \sigma' \\ \text{undef} & ; \text{ sinon} \end{cases}$$

Sémantique opérationnelle structurelle: SOS

Une sémantique opérationnelle définit un système de transition.
Un système de transition est donné par:

$$(\Gamma, T, \to) \text{ où}$$

- Γ est l'ensemble de configurations.
- $T \subseteq \Gamma$ est l'ensemble des configurations finales.
- $\to \subseteq \Gamma \times \Gamma$ est la transition de transition.

Dans le cas de la sémantique opérationnelle on a :

1. $\Gamma = (\textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.
2. $T = \textbf{State}$
3. $\to$ définie par des arbres d'inférence.

Sémantique opérationnelle structurelle: SOS

Le système de transition sera tel que par :

1. $\Gamma = (\textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.
2. $T = \textbf{State}$
3. $\rightarrow$ définie par des séquences d'inférence.

Sémantique opérationnelle structurelle 1

Idée : Décrire comment le résultat de l'exécution d'un programme est obtenu.
Sémantique décrite par un système d'inférence : axiomes et règles.

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

$$(\text{skip}, \sigma) \Rightarrow \sigma$$

$$\frac{(S_1, \sigma) \Rightarrow \sigma'}{(S_1; S_2, \sigma') \Rightarrow (S_2, \sigma')} \qquad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{(S_1; S_2, \sigma') \Rightarrow (S_1'; S_2, \sigma')}$$

Sémantique opérationnelle structurelle 2

Si $\mathcal{B}[b]\sigma = \textbf{ff}$ alors

$$(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \Rightarrow (S_2, \sigma)$$

Si $\mathcal{B}[b]\sigma = \textbf{tt}$ alors

$$(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \Rightarrow (S_1, \sigma)$$

$$(\text{while } b \text{ do } S, \sigma) \Rightarrow (\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, \sigma)$$

Séquence d'inférence

- Séquence fini :

  $\gamma_1, \gamma_1, \cdots, \gamma_k$ où

  - $\gamma_i \Rightarrow \gamma_{i+1}$, pour $i \in [1, k-1]$ et
  - $\gamma_k \not\Rightarrow$ c.a.d. qu'il n'existe aucune configuration γ avec $\gamma_k \Rightarrow \gamma$. Dans ce cas, si γ_k n'est pas une configuration finale, on l'appelle *configuration bloquante*.
- Séquence infini :

  $\gamma_1, \gamma_1, \cdots$ où

  $\gamma_i \Rightarrow \gamma_{i+1}$, pour $i \geq 1$

Donc, on ne considère que des séquence maximale.

Discussion sur les deux sémantiques opérationnelles

Comment modélisent-elles le fait qu'un programme divèrge?

- Sémantique naturelle :
  Un programme S divèrge dans l'états σ, si (S,σ) n'a pas de configuration successeur: $(S,\sigma) \not\rightarrow$

$$\nexists \gamma \cdot (S,\sigma) \rightarrow \gamma$$

- Sémantique structurelle :
  Un programme S divèrge dans l'états σ, s'il y a une séquence infini qui commence avec (S,σ).

Equivalence sémantique

- Sémantique naturelle :
  S_1 et S_2 sont *sémantiquement équivalents*, si pour tout états σ et σ':

$$(S_1,\sigma) \rightarrow \sigma' \text{ ssi } (S_2,\sigma) \rightarrow \sigma'$$

- Sémantique structurelle :
  S_1 et S_2 sont *sémantiquement équivalents*, si pour tout états σ
  - Pour toute configuration γ finale ou bloquante:

$$(S_1,\sigma) \Rightarrow^* \gamma \text{ ssi } (S_2,\sigma) \Rightarrow^* \gamma$$

  - Il existe une séquence inifni qui commence avec (S_1,σ) ssi il existe une séquence inifni qui commence avec (S_2,σ).

La fonction sémantique $\mathcal{S}_{sos}$

$$\mathcal{S}_{sos}[S]\sigma = \begin{cases} \sigma' & ; \text{ Si } (S,\sigma) \Rightarrow^* \sigma' \\ \text{undef} & ; \text{ sinon} \end{cases}$$

Question
a-t-on $\mathcal{S}_{ns} = \mathcal{S}_{sos}$?

Lemme Pout tout S, σ et σ'

Si $(S,\sigma) \rightarrow \sigma'$ alors $(S,\sigma) \Rightarrow^* \sigma'$.

□

Lemme Pout tout S, σ et σ'

Si $(S,\sigma) \Rightarrow^k \sigma'$ alors $(S,\sigma) \rightarrow \sigma'$.

□

Styles de sémantique et principes de preuve associés

- Sémantique inductive: ex. $\mathcal{A}$.
  Preuve par induction structurelle sur les expressions arithmétiques.
- Sémantique opérationnelle naturelle.
  Relation de transition définie par les arbres d'inférence.
  Preuve sur la structure de l'arbre d'inférence.
- Sémantique opérationnelle structurelle.
  Relation de transition définie par les séquences d'inférence.
  Preuve par récurrence sur la longueur de la séquence d'inférence.

Lemmes intérmidiaires

Lemme Pout tout S, σ et σ'

Si $(S,\sigma) \to \sigma'$ alors $(S,\sigma) \Rightarrow^* \sigma'$.

□

Preuve On montre d'abord

Si $(S_1,\sigma) \Rightarrow^k \sigma'$ alors $(S_1;S_2,\sigma) \Rightarrow^k (S_2;\sigma')$

Preuve de Lemme par induction sur la structure de l'arbre de dérivation de $(S,\sigma) \to \sigma'$. □

Lemmes intérmidiaires 2

Lemme Pout tout S, σ et σ'

Si $(S,\sigma) \Rightarrow^k \sigma'$ alors $(S,\sigma) \to \sigma'$.

□

Preuve On montre d'abord Si $(S_1;S_2,\sigma) \Rightarrow^k \sigma''$ alors il existe σ' et k_1 avec :
$(S_1,\sigma) \Rightarrow^{k_1} \sigma'$ et $(S_2,\sigma') \Rightarrow^{k-k_1} \sigma''$.
Preuve de Lemme par reccurrence sur k : $(S,\sigma) \Rightarrow^k \sigma'$. □

Extensions du langage **While**

- La commande abort. La configuration (abort,σ) n'a pas de sucesseur. C'est le blocage.
- La commande or : $S_1 \text{or} S_2$.
  On dénote le langage obtenu par **While**^{or}.
  Configuration :

$$\{(S,\sigma) \mid S \in \textbf{While}^{or},\ \sigma \in \textbf{State}\} \cup \textbf{State}.$$

Règles de transition pour or

- Règles de transition pour SN :

$$\frac{(S_1,\sigma) \to \sigma'}{(S_1 \text{or} S_2,\sigma) \to \sigma'}$$

$$\frac{(S_2,\sigma) \to \sigma'}{(S_1 \text{or} S_2,\sigma) \to \sigma'}$$

- Règles de transition pour SOS :

$$(S_1 \text{or} S_2,\sigma) \Rightarrow (S_1,\sigma)$$

$$(S_1 \text{or} S_2,\sigma) \Rightarrow (S_2,\sigma)$$

Discussion

Dans la sémantique naturelle le or "cache" la non-terminaison.
Exemple Soit $S_1 =$ while true do skip et $S_2 =$ while false do skip.

$$(S_1 \text{or} S_2, \sigma) \to \sigma$$

Mais $(S_1 \text{or} S_2, \sigma)$ aura toujours une séquence infini dans SOS. □

La mise en parallèle

On ajoute la commande $\|$: $S_1 \| S_2$.
On dénote le langage obtenu par **While**$^{\|}$.
Configuration : $\{(S,\sigma) \mid S \in$ **While**$^{\|}$, $\sigma \in$ **State**$\} \cup$ **State.**
Règles de transition pour or

- Règles de transition pour SN :

$$\frac{(S_1,\sigma)\to\sigma'}{(S_1 \| S_2,\sigma)\to\sigma'} \qquad \frac{(S_2,\sigma)\to\sigma'}{(S_1 \| S_2,\sigma)\to\sigma'}$$

- Règles de transition pour SOS :

$$\frac{(S_1,\sigma) \Rightarrow (S_1',\sigma')}{(S_1 \parallel S_2,\sigma) \Rightarrow (S_1' \parallel S_2,\sigma')}$$

$$\frac{(S_2,\sigma) \Rightarrow (S_2',\sigma')}{(S_1 \parallel S_2,\sigma) \Rightarrow (S_1 \parallel S_2',\sigma')}$$

Discussion concernant le $\|$

La sémantique naturelle ne permet pas d'exprimer une sémantique d'entrelacement (interleaving semantics).

La sémantique structurelle le permet.

Conclusion

Sémantique naturelle

- ne distingue pas entre blocage et non-terminaison
- non-determinisme "cache" la non-terminaison
- ne permet pas d'exprimer la sémantiqued'entrelacement

Sémantique structurelle

- distingue entre blocage et non-terminaison
- non-determinisme ne "cache" pas la non-terminaison
- permet d'exprimer la sémantiqued'entrelacement

Blocs et procédures

Blocs et déclarations de variables

$$
\begin{array}{lll}
S & \in & \textbf{Stm} \\
S & ::= & x := a \mid \text{skip} \mid S_1; S_2 \mid \\
 & & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & & \text{while } b \text{ do } S \mid \text{begin } D_V; S \text{ end}
\end{array}
$$

La catégorie syntaxique $\textbf{Dec}_V$

$$D_V ::= \text{var } x := a;\ D_V \mid \epsilon$$

Exemple

begin var $y := 1$;
($x := 1$;
begin var $x := 2$; $y := x + 1$ end
$x := y + x$)
end

Exemple

begin var $y := 1$;
($x := 1$;
begin var $x := 2$; $y := x + 1$ end
$x := y + x$)
end

Questions :

1. Les déclarations sont-elles déja actives pour les dclarations?
2. Quel odre choisir pour les déclarations?
3. Comment réstorer l'état?

Sémantique opérationnelle naturelle

Notation :

- DV(D_V) dénote l'ensemble de variables déclarées dans D_V.
- $s'[X \mapsto \sigma] = \lambda x \cdot$ if $x \in X$ then $\sigma(x)$ else $\sigma'(x)$.

Pour définir la sémantique, on définit un système de transitions pour les déclarations et un système pour les commandes.

Déclarations :

Configurations de la form (D_v, σ) ou s.

$$\frac{(D_V, \sigma[x \mapsto \mathcal{A}[a]\sigma]) \to_D \sigma'}{(\text{var } x := a;\ D_V, \sigma) \to_D \sigma'}$$

$$(\epsilon, \sigma) \to_D \sigma$$

Règle de transition pour les blocs

$$\frac{(D_V, \sigma) \to_D \sigma' \quad (S, \sigma') \to \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \to \sigma''[\text{DV}(D_V) \mapsto \sigma]}$$

Règle de transition pour les blocs

$$\frac{(D_V, \sigma) \to_D \sigma' \quad (S, \sigma') \to \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \to \sigma''}$$

Procédure

S	$\in$	**Stm**
S	::=	$x := a$ | skip | $S_1; S_2$ |
		if b then S_1 else S_2
		while b do S | begin D_V D_P; S end | call p
D_V	::=	var $x := a;\ D_V$ | ϵ
D_P	::=	proc p is $S;\ D_P$ | ϵ

Les déclarations de procédures forment la catégorie syntaxique

Dec$_P$.

Exemple

```
begin  var x := 0;
       proc p is x := x * 2;
       proc q is call p;
       begin var x := 5;
             proc p is x := x + 1;
             call q; y := x;
       end;
end
```


Exemple 2

Liens dynamiques pour les variables et les procédures.

```
begin  var x := 0;
       proc p is x := x * 2;
       proc q is call p;
       begin var x := 5;
             proc p is x := x + 1;
             call q; y := x;
       end;
end
```


Exemple 2

Liens dynamiques pour les variables et les procédures.

```
begin  var x := 0;
       proc p is x := x * 2;
       proc q is call p;
       begin var x := 5;
             proc p is x := x + 1;
             call p; y := x;
       end;
end
```


Exemple 2

Liens dynamiques pour les variables et les procédures.

```
begin  var x := 0;
       proc p is x := x * 2;
       proc q is call p;
       begin var x := 5;
             proc p is x := x + 1;
             x := x + 1; y := x;
       end;
end
```

Exemple 3

Liens dynamiques pour les variables et liens statiques pour les procédures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call q; y := x;
        end;
end
```


Exemple 3

Liens dynamiques pour les variables et liens statiques pour les procédures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call p; y := x;
        end;
end
```

Yassine Lakhnech, Sémantique Start C3 C4 – p.46/100

Exemple 3

Liens dynamiques pour les variables et liens statiques pour les procédures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              x := x * 2; y := x;
        end;
end
```

Yassine Lakhnech, Sémantique Start C3 C4 – p.46/100

Exemple 4

Liens statiques pour les variables et les procédures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call q; y := x;
        end;
end
```

Exemple 4

Liens statiques pour les variables et les procédures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call p; y := x;
        end;
end
```


Exemple 4

Liens statiques pour les variables et les procédures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              x := x * 2; y := x;
        end;
end
```

Yassine Lakhnech, Sémantique Start C3 C4 – p.47/100

Sémantique : liens dynamiques

Un état associe une valeur, un entier, à une variable.
Un *environnement* associe une valeur à une procédure.

$$\textbf{Env}_P = \textbf{Pname} \xrightarrow{part.} \textbf{Stm}$$

Les configurations : $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.

Règles de transitions

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (\mathsf{upd}(env, D_P), S, \sigma') \rightarrow \sigma''}{(env, \text{begin } D_V\ D_P; S \text{ end}, \sigma) \rightarrow \sigma''[\mathsf{DV}(D_V) \mapsto \sigma]}$$

où

- $\mathsf{upd}(env, \epsilon) = env$ et
- $\mathsf{upd}(env, \text{proc } p \text{ is } S; D_P) = \mathsf{upd}(env[p \mapsto S], D_P)$.

$$\frac{(env, env(p), \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'}$$

Sémantique : liens statiques pour les procédures

$$\textbf{Env}_P = \textbf{Pname} \xrightarrow{part.} \textbf{Stm} \times \textbf{Env}_P$$

Les configurations : $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.

- upd$(env, \epsilon) = env$ et
- upd$(env,$ proc p is $S; D_P) =$ upd$(env[p \mapsto (S, env)], D_P)$.

```
begin   var  x := 2;
        proc p is x := 0;
        proc q is begin x := 1; (proc p is call p); call p end;
        call q
end
```


Sémantique : liens statiques pour les procédures

$$\textbf{Env}_P = \textbf{Pname} \xrightarrow{part.} \textbf{Stm} \times \textbf{Env}_P$$

Les configurations : $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.

- upd$(env, \epsilon) = env$ et
- upd$(env,$ proc p is $S; D_P) =$ upd$(env[p \mapsto (S, env)], D_P)$.

```
begin   var  x := 2;
        proc p is x := 0;
        proc q is begin x := 1; (proc p is call p); call p end;
        call q
end
```

Yassine Lakhnech, Sémantique Start C3 C4 – p.50/100

Sémantique : liens statiques pour les procédures

$$\textbf{Env}_P = \textbf{Pname} \xrightarrow{part.} \textbf{Stm} \times \textbf{Env}_P$$

Les configurations : $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.

- upd$(env, \epsilon) = env$ et
- upd$(env,$ proc p is $S; D_P) =$ upd$(env[p \mapsto (S, env)], D_P)$.

```
begin   var  x := 2;
        proc p is x := 0;
        proc q is begin x := 1; (proc p is call p); call p end;
        call q
end
```

Yassine Lakhnech, Sémantique Start C3 C4 – p.50/100

Règles de transitions

Deux alternatives :

$$[\text{call}] \quad \frac{(env', S, \sigma) \to \sigma'}{(env, \text{call } p, \sigma) \to \sigma'} \text{ où } env(p) = (S, env').$$

$$[\text{call}_{rec}] \quad \frac{(env'[p \mapsto (S, env')], S, \sigma) \to \sigma'}{(env, \text{call } p, \sigma) \to \sigma'} \text{ où } env(p) = (S, env').$$

Sémantique : liens statiques

On remplace létat par une table des symboles et la mémoire :

- une table des symboles associe à une variable (un identificateur) une adesse dans la mémoire.
- La mémoire associe à une adresse une valeure.

Table des symboles : environnement des variables :

$$\textbf{Env}_V = \textbf{Var} \xrightarrow{part.} \textbf{Loc} = \mathbb{Z}$$

Mémoire : $\textbf{Store} = \textbf{Loc} \xrightarrow{part.} \mathbb{Z}$
On dénote par $\mathsf{new}(sto)$ le plus petit entier n tel que $sto(n)$ n'est pas défini.
Intuition : l'état correspond à $sto \circ env_V$.

Configurations

- Déclarations de variables :
  $(\textbf{Dec}_V \times Envv \times \textbf{Store}) \cup (Envv \times \textbf{Store})$.
- $\textbf{Env}_P = \textbf{Pname} \xrightarrow{part.} \textbf{Stm} \times \textbf{Env}_V \times \textbf{Env}_P$.
  - $\mathsf{upd}(env_V, env_P, \epsilon) = env_P$ et
  - $\mathsf{upd}(env_V, env_P, \mathsf{proc}\ p\ \mathsf{is}\ S; D_P) = \mathsf{upd}(env_V, env_P[p \mapsto (S, env_V, env_P)], D_P)$.
- Commandes : $(\textbf{Env}_V \times \textbf{Env}_P \times \textbf{Stm} \times \textbf{Store}) \cup (\textbf{Env}_V \times \textbf{Store})$.

Règles sémantiques pour déclaration de variables

$$\frac{(D_V, env_V[x \mapsto \mathsf{new}(sto)], st[\mathsf{new}(sto) \mapsto v]) \to_D (env'_V, sto')}{(\mathsf{var}\ \ x := a;\ D_V, env_V, sto) \to_D (env'_V, sto')}$$

où $v = \mathcal{A}[a](st \circ env_V)$.

$$(\epsilon, env_V, sto) \to_D (env_V, sto)$$

Règles sémantiques pour les commandes

$$(env_V, env_P, x := a, sto) \to (env_V, sto[env_V(x) \mapsto \mathcal{A}[a](sto \circ env_V)])$$

$$[\mathsf{call}] \quad \frac{(env'_V, env'_P, S, sto) \to (env'_V, sto')}{(env_V, env_P, \mathsf{call}\ p, sto) \to (env'_V, sto')}.$$

$$[\mathsf{call}_{rec}] \quad \frac{(env'_V, env'_P[p \mapsto (S, env'_V, env'_P)], S, sto) \to (env'_V, sto')}{(env_V, env_P, , \mathsf{call}\ p, sto) \to (env'_V, sto')}$$

où $env(p) = (S, env'_V, env'_P)$.

Génération de code correcte

Nous allons voir :

- comment on définit une sémantique opérationnelle d'une machine abstraite : une machine à pile.
- comment on peut spécifier un générateur de code pour le langage **While** par induction sur la structure des programmes.
- comment on peut utiliser la sémantique opérationnelle du langage et de la machine abstraite pour montrer que la spécification du générateur de code est correcte.

La machine abstraite AM

La machine AM est définit par un système de transition dont les configurations sont des triplets :

- Une liste $instr_1, \cdots, instr_n$ d'instruction. C'est le code qui reste à exécuter.
- Une pile qui est utilisée pour évaluer les expressions.
- La mémoire de la machine qui est décrite par un état; donc une fonction des variables vers $\mathbb{Z}$.

Nous allons définir l'ensemble des instructions et la relation de transition

$$(c, p, m) \triangleright (c' p', m')$$

La machine AM n'a donc ni registres et ni accumulateur. C'est le top de la pile qui joue le role d'accumulateur et le reste de la pile ceui de registres.

Les instructions

Instructions	Effet
push-n,True,False	empiler la constante n,**tt**,**ff**
fetch(x) store(x)	empiler la valeur de x dans l'état actuel dépiler le top de la pile et l'affecter à x
add	remplacer les deux plus hauts éléments de la pile par leur somme
sub,mult.and,le,equal,neg	similaire
brunch(c_1,c_2)	si le top est **tt** exécuter c_1 s'il est **ff** alors c_2, sinon blocage
loop(c_1,c_2)	exécuter c_1, si le top de la pile est **tt**, exécuter c_2 suivie de loop(c_1,c_2)) si c'est **ff** s'arreter
noop	skip

La relation de transition-sémantique des instructions

On définit les progammes cibles comme des mots sur l'alphabet des d'instructions. L'ensemble des programmes cibles et dénoté **Code**.
Une configuration de AM est donc (c, p, m) où c est un programme cible, p est le contenu de la pile qui est un mot sur $\mathbb{Z} \cup \mathbb{B}$ et m est un état dans **State**.
La relation $\triangleright$ est inductivement définit par :

$$
\begin{array}{lll}
(\text{push-n} \cdot c, p, m) & \triangleright (c, n \cdot p, m) & \\
(\text{True} \cdot c, p, m) & \triangleright (c, \mathbf{tt} \cdot p, m) & \\
(\text{False} \cdot c, p, m) & \triangleright (c, \mathbf{ff} \cdot p, m) & \\
(\text{fetch}(x) \cdot c, p, m) & \triangleright (c, m(x) \cdot p, m) & \\
(\text{store}(x) \cdot c, p, m) & \triangleright (c, v \cdot p, m[x \mapsto v]) & \text{si } v \in \mathbb{Z} \\
(\text{add} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 + v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\
(\text{sub} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 - v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z}
\end{array}
$$

La relation de transition-sémantique des instructions

:

$$
\begin{array}{lll}
(\text{mult} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 * v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\
(\text{le} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 \leq v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\
(\text{equal} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 = v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\
(\text{and} \cdot c, b_1 \cdot b_2 \cdot p, m) & \triangleright (c, (b_1 \wedge b_2) \cdot p, m) & \text{si } b_1, b_2 \in \mathbb{B} \\
(\text{neg} \cdot c, b \cdot p, m) & \triangleright (c, (\neg b) \cdot p, m) & \text{si } b \in \mathbb{B} \\
(\text{brunch}(c_1, c_2) \cdot c, \textbf{tt} \cdot p, m) & \triangleright (c_1 \cdot c, p, m) & \\
(\text{brunch}(c_1, c_2) \cdot c, \textbf{ff} \cdot p, m) & \triangleright (c_2 \cdot c, p, m) & \\
(\text{loop}(c_1, c_2) \cdot c, p, m) & \triangleright & \\
 & (c_1 \cdot \text{brunch}(c_2 \cdot \text{loop}(c_1, c_2), \text{noop}) \cdot c, p, m) & \\
(\text{noop} \cdot c, p, m) & \triangleright (c, p, m) &
\end{array}
$$

Quelques propriétés de AM

- La relation de transition $\triangleright$ est déterminsite :

  $(c,p,m) \triangleright (c_1,p_1,m_1) \wedge (c,p,m) \triangleright (c_2,p_2,m_2) \Rightarrow (c_1,p_1,m_1) = (c_2,p_2,$

  Ceci peut être démontré sur la longueur de c.
- Extensabilité du code et de la pile :

  $(c_1,p_1,m_1) \triangleright (c_2,p_2,m_2) \Rightarrow (c_1 \cdot c, p_1 \cdot p, m_1) \triangleright (c_2 \cdot c, p_2 \cdot p, m_2)$
- Code composabilité :
  Si $(c_1 \cdot c_2, p, m) \triangleright^k (\epsilon, p_2, m_2)$ alors il existe $k' \in \mathbb{N}$ et une configuration (ϵ, p', m') tels que $(c_1, p, m) \triangleright^{k'} (\epsilon, p', m')$ et $(c_2, p', m') \triangleright^{k-k'} (\epsilon, p_2, m_2)$.

La sémantique d'un programme cible

On définit la fonction sémantique

$$\mathcal{M} : \textbf{Code} \rightarrow (\textbf{State} \rightarrow \textbf{State}).$$

$$\mathcal{M}[c]m = \begin{cases} m' & ,(c,\epsilon,m) \triangleright^* (\epsilon,p,m') \\ \text{undef, sinon} & \end{cases}$$

Génération de code : le problème

Nous voulons définir trois fonctions :

1. $\mathcal{CA}$: **Aexp** $\rightarrow$ **Code**
2. $\mathcal{CB}$: **Bexp** $\rightarrow$ **Code**
3. $\mathcal{S}$: **Stm** $\rightarrow$ **Code**

tels que pour tout programme $c \in$ **Stm** on a

$$\mathcal{S}_{ns}[\,] = \mathcal{M} \circ \mathcal{CS}$$

Nous allons exiger que $\mathcal{CA}$, $\mathcal{CB}$ et $\mathcal{CS}$ satisfassent les propriétés suivantes :

1. $(\mathcal{CA}[a], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$
2. $(\mathcal{CB}[b], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[b]\sigma, \sigma)$
3. $(\mathcal{CS}[S], \epsilon, \sigma) \triangleright^* (\epsilon, p, \sigma')$ ssi $(S, \sigma) \rightarrow \sigma'$

Génération de code

Quelques exemples de clauses pour définir $\mathcal{CA}$, $\mathcal{CB}$ et $\mathcal{CS}$:

- $\mathcal{CA}[n] =$ push-n.
- $\mathcal{CA}[x] =$ fetch(x)
- $\mathcal{CA}[a_1 + a_2] = \mathcal{CA}[a_2] \cdot \mathcal{CA}[a_1] \cdot$ add
- $\mathcal{CB}[\text{true}] =$ True
- $\mathcal{CS}[x := a] = \mathcal{CA}[a] \cdot$ store(x)
- $\mathcal{CS}[S_1; S_2] = \mathcal{CS}[S_1] \cdot \mathcal{CS}[S_2]$

Exemple

Sémantique Axiomatique : Logique de Hoare

Correction partielle et correction totale

Le but est de spécifier la relation "entrées/sorties" que doit satisfaire le programme. Exemple :

Fact:
$y := 1;$
while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$

Correction partielle
Si la valeur de x est initialement $n > 0$ et si le programme términe alors la valeur finale de y est $n!$.
Correction totale

Si la valeur de x est initialement $n > 0$ alors le programme términe et la valeur finale de y est $n!$.

Vérification sémantique

Fact:
$y := 1;$
while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$
$(\text{Fact}, \sigma) \to \sigma'$
$\Downarrow$
$\sigma'(y) = \sigma(x)!$ et $\sigma(x) > 0$

Trois étapes:

1. Correction du corps de la boucle.
2. Correction de la boucle.
3. Correction du programme.

Dans les trois cas on étudie l'arbre de dérivation.

Triplet de Hoare

Exemple :

$\{x = n \wedge n > 0\}$
$y := 1;$
while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$
$\{y = n! \wedge n > 0\}$

- Precondition: $\{x = n \wedge n > 0\}$
- Postcondition: $\{y = n! \wedge n > 0\}$.

Nous utiliserons la logique de premier ordre pour décrire les pre- et postconditions.

Corréction partielle et totale

Le triplet

$$\{P\}S\{Q\}$$

est valide, noté : $\models \{P\}S\{Q\}$*, ssi pour tout états* σ, σ'*:*

- *Si* $\sigma \models P$ *et* $(S, \sigma) \to \sigma'$ *alors*
- $\sigma' \models Q$.

On dit que S est *partiellement correcte* par rapport à P et Q.
Le triplet

$$[P]S[Q]$$

est valide, noté : $\models [P]S[Q]$*, ssi pour tout états* σ*:*

- *Si* $\sigma \models P$ *alors le programme términe et*
- *pour tout état* σ'*: si* $(S, \sigma) \to \sigma'$ *alors* $\sigma' \models Q$.

On dit que S est *totalement correcte* par rapport à P et Q.

Calcul de Hoare

Axiomes:

$$\{P[a/x]\}x := a\{P\}$$

$$\{P\}\text{skip}\{P\}$$

Règles d'inférence:
Composition:

$$\frac{\{P\}S_1\{Q\}\ \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

Conditionnelle:

$$\frac{\{b \wedge P\}S_1\{Q\}\quad \{\neg b \wedge P\}S_2\{Q\}}{\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q\}}$$

La logique de Hoare pour la correction partielle

While:

$$\frac{\{b \wedge P\} S \{P\}}{\{P\} \text{while } b \text{ do } S \{\neg b \wedge P\}}$$

Conséquence:

$$\frac{\{P'\} S \{Q'\}}{\{P\} S \{Q\}} \text{ Si } \models P \Rightarrow P' \text{ et } \models Q' \Rightarrow Q$$

Notation:
On note

$$\vdash \{P\} S \{Q\}$$

quand on peut déduire $\{P\} S \{Q\}$ avec les axiomes et les règles.

L'exemple Fact

On peut montrer:

- $\{x = n \wedge n > 0\} y := 1 \{x = n \wedge n > 0 \wedge y = 1\}$
- $\{x = n \wedge n > 0 \wedge y = 1\}$
  while $\neg(x = 1)$ do $y := y * x; x := x - 1$
  $\{y = n! \wedge n > 0\}$.

et conclure
$\{x = n \wedge n > 0\}$
$y := 1;$ while $\neg(x = 1)$ do $y := y * x; x := x - 1$
$\{y = n! \wedge n > 0\}$

L'exemple PGCD

Soit pgcd le programme suivant :
Exemple :

while $\neg(x = y)$ do (if $x > y$ then $x := x - y$ else $y := y - x$);
$z := x$

On veut montrer

$$\{x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0\} \text{pgcd} \{z = \text{pgcd}(n, m)\}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{pre\}\text{pgcd}\{z = pgcd(n, m)\}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{pre \Rightarrow I \quad \{I\}\text{pgcd}\{z = \text{pgcd}(n, m)\}}{\{pre\}\text{pgcd}\{z = pgcd(n, m)\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.
$pre \Rightarrow I \quad \{I\}\text{pgcd}\{z = \text{pgcd}(n, m)\}$
$\{pre\}\text{pgcd}\{z = pgcd(n, m)\}$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{I\}\text{pgcd}\{z = \text{pgcd}(n, m)\}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I\}S'\{x = \text{pgcd}(n, m)\} \quad \{x = \text{pgcd}(n, m)\}z := x\{z = \text{pgcd}(n, m)\}}{\{I\}S'; z := x\{z = \text{pgcd}(n, m)\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I\}S'\{x = \text{pgcd}(n, m)\} \quad \{x = \text{pgcd}(n, m)\}z := x\{z = \text{pgcd}(n, m)\}}{\{I\}S'; z := x\{z = \text{pgcd}(n, m)\}}$$

Yassine Lakhnech, Sémantique Start C3 C4 – p.75/100

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{I\}S'\{x = \text{pgcd}(n, m)\}$$

Yassine Lakhnech, Sémantique Start C3 C4 – p.75/100

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I \wedge x \neq y\}S''\{I\} \quad I \wedge x = y \Rightarrow x = \text{pgcd}(n, m)}{\{I\}S'\{x = \text{pgcd}(n, m)\}}$$

Yassine Lakhnech, Sémantique Start C3 C4 – p.75/100

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I \wedge x \neq y\} S'' \{I\} \quad I \wedge x = y \Rightarrow x = \text{pgcd}(n, m)}{\{I\} S' \{x = \text{pgcd}(n, m)\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{I \wedge x \neq y\} \text{if } x > y \text{ then } x := x - y \text{ else } y := y - x \{I\}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I \wedge x \neq y \wedge x > y\} x := x - y \{I\} \quad \{I \wedge x \neq y \wedge \neg(x > y)\} y := y - x \{I\}}{\{I \wedge x \neq y\} \text{if } x > y \text{ then } x := x - y \text{ else } y := y - x \{I\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge x > y) \Rightarrow I[x - y/x]}{\{I \wedge x \neq y \wedge x > y\} x := x - y \{I\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge x > y) \Rightarrow I[x - y/x]}{\{I \wedge x \neq y \wedge x > y\} x := x - y \{I\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge \neg(x > y)) \Rightarrow I[y - x/x]}{\{I \wedge x \neq y \wedge \neg(x > y)\} y := y - x \{I\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge \neg(x > y)) \Rightarrow I[y - x/x]}{\{I \wedge x \neq y \wedge \neg(x > y)\} y := y - x \{I\}}$$

La preuve dans la logique de Hoare

Soient S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S''
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{pre\}\text{pgcd}\{z = pgcd(n, m)\}$$

Correction et complétude

Correction:

$$\text{Si } \vdash \{P\}S\{Q\} \text{ alors } \models \{P\}S\{Q\}$$

on peut déduire que des triplets valides.

Complétude

$$\text{Si } \models \{P\}S\{Q\} \text{ alors } \vdash \{P\}S\{Q\}$$

on peut déduire tous les triplets valides.

Correction de la logique de Hoare

Théorème La logique de Hoare pour la correction partielle est correcte:

$$\text{Si } \vdash \{P\}S\{Q\} \text{ alors } \models \{P\}S\{Q\}$$

La preuve est par induction sur la structure de l'arbre de déduction.

Complétude de la logique de Hoare

Théorème La logique de Hoare pour la correction partielle est complette:

$$\text{Si } \models \{P\}S\{Q\} \text{ alors } \vdash \{P\}S\{Q\}$$

Plan de la preuve

On introduit $\mathsf{wlp}(S, Q)$ avec :

- $\models \{P\}S\{Q\}$ ssi $\models P \Rightarrow \mathsf{wlp}(S, Q)$ et
- $\vdash \{\mathsf{wlp}(S, Q)\}S\{Q\}$

$\mathsf{wlp}(S, Q)$ décrit l'ensemble:

$$\{\sigma \mid \forall \sigma' \cdot (S, \sigma) \to \sigma' \Rightarrow \sigma' \models Q\}$$

Pour l'instant, on suppose qu'on peut donner une formule de premier-ordre qui exprime $\mathsf{wlp}(S, Q)$.
La preuve du premier point est directe.
Le deuxième est induction structurelle sur la syntaxe des programmes.

Cas du While

$$S \equiv \text{while } b \text{ do } S'$$

Pour montrer

$$\vdash \{\text{wlp}(S,Q)\}S\{Q\}$$

il suffit de montrer:

1. $\models \neg b \wedge \text{wlp}(S,Q) \Rightarrow Q$
2. $\models b \wedge \text{wlp}(S,Q) \Rightarrow \text{wlp}(S', \text{wlp}(S,Q))$.

car

$$\dfrac{\dfrac{\dfrac{2.\quad \{\text{wlp}(S', \text{wlp}(S,Q))\}S'\{\text{wlp}(S,Q)\}\ (I.H.)}{\{b \wedge \text{wlp}(S,Q)\}S'\{\text{wlp}(S,Q)\}}}{\{\text{wlp}(S,Q)\}S\{\neg b \wedge \text{wlp}(S,Q)\}} \quad 1.}{\{\text{wlp}(S,Q)\}S\{Q\}}$$

expressivité et décidabilité

Nous avons fait l'hypothèse qu'on peut exprimer $\text{wlp}(S,Q)$ comme une formule de premier-ordre. Mais cette hypothèse est elle vraie?
Elle ne l'est que si notre logique de premier-ordre contient l'arithmétique de Peano c.a.d. $(\mathbb{N}, +, *)$.
Raison: Il faut côder des séquences finis d'entiers par des entiers.
C'est la Goedelisation que vous verrez en calculabilité.
Conséquence: indécidabilité de $\{P\}S\{Q\}$.
L'ensemble complément de

$$\{\{P\}S\{\text{false}\} \mid \models \{P\}S\{Q\}\}$$

est récursivement énumérable (re) mais pas récursif.

Preuve de terminaison

Rappel: *Le triplet*

$$[P]S[Q]$$

est valide, noté : $\models [P]S[Q]$*, ssi pour tout états* σ*:*

- *Si* $\sigma \models P$ *alors le programme termine et*
- *pour tout état* σ'*: si* $(S,\sigma) \to \sigma'$ *alors* $\sigma' \models Q$.

On dit que S est *totalement correcte* par rapport à P et Q.
$\models [P]S[Q]$ ssi $\models \{P\}S\{Q\}$ et $\models [P]S[true]$ en d'autres mots :
Correction totale = Correction partielle $\wedge$ terminsaison.

Nous n'allons pas donner une logique de Hoare pour la correction totale. Nous allos juste montrer comment on montre qu'une boucle termine.

Ordre bien-fondées

Rappel: Soit $\leq$ une relation sur A.
$\leq$ est un ordre sur A ssi

1. $\leq$ est reflexive: $\forall a \in A \cdot a \leq a$.
2. $\leq$ est anti-symétrique: $\forall a, b \in A \cdot a \leq b \wedge b \leq a \Rightarrow a = b$.
3. $\leq$ est transitive: $\forall a, b, c \in A \cdot a \leq b \wedge b \leq c \Rightarrow a \leq c$.

Un ordre est total (ou linéaire), si on a $a \leq b$ ou $b \leq a$, pour tout $a, b \in A$.
On définit: $a < b$ ssi $a \leq b \wedge a \neq b$.

Un ordre $\leq$ est dit *bien-fondé* s'il n'y pas de chaîne décroissante infinie $a_0 > a_1 > a_2 \cdots$.

Exemples

- $(\mathbb{N}, \leq)$ est un ordre bien-fondé.
- $(\Sigma^*, \preceq)$ prefixe sur les mots finis est un ordre bien-fondé.
- $\mathbb{N}^*$ doté de l'odre lexigrographique est bien-fondé.
- $(2^A, \subseteq)$ est bien-fondé.

Terminaison de boucles

Théorème

$[P]$while b do $S'[true]$ est vrai

ssi il existe un ordre bien-fondé $(W, \leq)$, une fonction f : **State** $\rightarrow W$ (rang de terminaison) et un prédicat I tels que

1. $b \wedge P \Rightarrow I$ et $\{I \wedge B\}S'\{I\}$ c.a.d. I est un invariant de boucle et
2. $[b \wedge I \wedge f = z]S'[f < z]$ c.a.d. le rang de terminaison décroit à chaque itération.

Exemple

while $\neg(x = y)$ do (if $x > y$ then $x := x - y$ else $y := y - x$);
$z := x$

Nous avons montré:

$\{x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0\}$pgcd$\{z = \text{pgcd}(n, m)\}$.

Et la terminaison?

Sémantique Dénotionnelle

Motivation

La sémantique opérationnelle nous décrit comment un programme est exécuté. Elle nous sert par exemple à écrire un générateur de code.
La sémantique axiomatique nous permet de prouver des propriétés de nos programmes.
La sémantique dénotationnelle nous permettra de décrire l'effet d'un programme sur un état sans dire comment le programme est exécuté.

Un autre aspect important est la *compositionalité*. La sémantique d'un programme composé est une fonction de la sémantique des composants.

Les clauses

$$\mathcal{S}_d : \textbf{Stm} \to (\textbf{State} \xrightarrow{part.} \textbf{State})$$

$\mathcal{S}_d[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[a]\sigma]$
$\mathcal{S}_d[\![\text{skip}]\!] = id$
$\mathcal{S}_d[\![S_1; S_2]\!] = \mathcal{S}_d[\![S_2]\!] \circ \mathcal{S}_d[\![S_1]\!]$
$\mathcal{S}_d[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] = \text{cond}(\mathcal{B}[b], \mathcal{S}_d[\![S_1]\!], \mathcal{S}_d[\![S_2]\!])$ où

$$\text{cond}(p, f, g)\sigma = \begin{cases} f(\sigma); \text{ si } p(\sigma) = \textbf{tt} \\ g(\sigma); \text{ sinon} \end{cases}$$

Sémantique d'une boucle

On devrait avoir

$$\mathcal{S}_d[\![\text{while } b \text{ do } S]\!] = \text{cond}(\mathcal{B}[B], \mathcal{S}_d[\![\text{while } b \text{ do } S]\!] \circ \mathcal{S}_d[\![S]\!], id)$$

Donc $\mathcal{S}_d[\![\text{while } b \text{ do } S]\!]$ doit satisfaire l'équation:

$$f = \text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id)$$

Soit $F(f) = \text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id)$.
Mais cette équation a-t-elle une solution? peut être plusieurs?
Nous verrons qu'il existe une solution et que c'est la plus *petite* qui nous interesse.
En effet, on peut montrer que pour tout f tel que $f = F(f)$, on a:

$$\mathcal{S}_d[\![\text{while } b \text{ do } S]\!]\sigma = \sigma' \Rightarrow f(\sigma) = \sigma'.$$

L'existence d'un point fixe

L'outil pour répondre à cette question est
la théorie des points fixes (fixed point theory)

Minorants, majorants et limites

Soit $(D, \sqsubseteq)$ un ensemble ordonné et soit $X \subseteq D$.

- d est un *minorant* de X, si $d \sqsubseteq x$, pour tout $x \in X$.
- d est un *majorant* de X, si $x \sqsubseteq d$, pour tout $x \in X$.
- d est une *limite inférieure* de X, noté $\sqcap X$, si d est un minorant de X et pour tout minorant y: $y \sqsubseteq x$.
- d est une *limite supérieure* de X, noté $\sqcup X$, si d est un minorant de X et pour tout majorant y: $x \sqsubseteq y$.

On dénote par bot la limite $\sqcap D$ et par $\top$ $\sqcup D$ quand ils existent.
Exemple:

Considérons $(2^A, \subseteq)$. Alors, $\sqcap X = \bigcap_{x \in X} x$ et $\sqcup X = \bigcup_{x \in X} x$.

Treillis

Un ensemble $(D, \sqsubseteq)$ est un *treillis*, si $\sqcap X$ et $\sqcup X$ existent pour tout $X \subseteq D$ fini.
C'est un *treillis complet*, si $\sqcap X$ et $\sqcup X$ existent pour tout $X \subseteq D$.
Exemples:

1. $(2^A, \subseteq)$ est un treillis complet.
2. $(\{X \mid X \subseteq A, X \text{ fini}\}, \subseteq)$ est un treillis qui n'est pas complet, si A est infini.
3. Soit Σ un alphabet fini. Alors $(\Sigma^*, \preceq)$ est un treillis qui n'est pas complet.

Fonctions continues

Soit $(D, \sqsubseteq)$ un ensemble ordonné. Une *chaîne* (au fait ω-chaîne) est un sous-ensemble $X \subseteq D$ dénombrable ou fini sur lequel $\sqsubseteq$ est total c.a.d. une chaîne peut être donnée par $x_0 \sqsubseteq x_1 \cdots$.
Definition: Soient $(D, \sqsubseteq)$ et $(D', \sqsubseteq')$ deux ensembles ordonnés. Une fonction $f : D \to D'$ est *continue*, si:

1. elle est monotone: $d_1 \sqsubseteq d_2$ implique $f(d_1) \sqsubseteq' f(d_2)$ et
2. elle est $\sqcup$-additive: $f(\sqcup X) = \sqcup' f(X)$, pour toute chaîne X.

Les théorèmes de Knaster-Tarski et Kleene

Deux résultats fondamentaux:
Théorème Soit $(D, \sqsubseteq)$ un treillis complet et $f : D \to D$.

- Knaster-Tarski Si f est monotone alors $\sqcap\{x \mid f(x) \sqsubseteq x\}$ est le plus petit point fixe de f.
- Kleene Si f est continue alors $\bigsqcup_{i \geq 0} f^i(\bot)$ est le plus petit point fixe avec $\bigsqcup_{i \geq 0} f^i(\bot) = \bigsqcup\{f^i(\bot) \mid i \geq 0\}$.

Preuve du Théorème de Knaster-Tarski

Soit $(D, \sqsubseteq)$ un treillis complet et $f : D \to D$ monotone.
Soit $A = \{x \mid f(x) \sqsubseteq x\}$ et $x_0 = \sqcap A$.

1. x_0 est un point fixe.
   (a) On montre que si $x \in A$ alors $f(x) \in A$. Si $x \in A$ alors $f(x) \sqsubseteq x$. Alors par monotonie de f, $f(f(x)) \sqsubseteq f(x)$. Donc, $f(x) \in A$.
   (b) On montre $x_0 \in A$. Par défintion de x_0, pout tout $x \in A$ on a $x_0 \sqsubseteq x$. Donc, par monotonie de f, pout tout $x \in A$ on a $f(x_0) \sqsubseteq f(x) \sqsubseteq x$. Donc, $f(x_0)$ est un minorant de A. Comme x_0 est le plus grand minorant de A, on a $f(x_0) \sqsubseteq x_0$. Donc $x_0 \in A$.
   (a) et (b) implique $f(x_0) \in A$. Donc, $x_0 \sqsubseteq f(x_0)$ et $f(x_0) = x_0$.
2. x_0 est le plus petit point fixe. Car tout point fixe de f est dans A et x_0 est un minorant de A.

Preuve du Théorème de Kleene

Soit $(D, \sqsubseteq)$ un treillis complet et $f : D \to D$ continue.

1. $\bigsqcup_{i \geq 0} f^i(\bot)$ est un point fixe.

$$f(\bigsqcup_{i\geq 0} f^i(\bot)) = \bigsqcup_{i\geq 0} f^{i+1}(\bot) = \bigsqcup_{i\geq 1} f^i(\bot) = \bot \sqcup \bigsqcup_{i\geq 1} f^i = \bigsqcup_{i\geq 0} f^i(\bot)$$

2. $\bigsqcup_{i \geq 0} f^i(\bot)$ est plus que tout point fixe de f.
   Soit x un point fixe. On montre par récurrence: $\forall i \geq 0 \cdot f^i(\bot) \sqsubseteq x$. Ce qui implique $\bigsqcup_{i\geq 0} f^i(\bot) \sqsubseteq x$.

Ordre sur les états

Considérons $(\textbf{State} \overset{part.}{\to} \textbf{State}, \sqsubseteq)$ où $f \sqsubseteq g$ ssi pour tout σ, si $f(\sigma)$ est défini alors $g(\sigma)$ est défini et $f(\sigma) = g(\sigma)$.
Alors, $\sqcap X$ est toujours défini mais pas $\sqcup X$.
Pour cette raison, nous considérons:

$$\textbf{State}_\bot^\top = \textbf{State} \cup \{\top\} \cup \{\bot\}$$

avec l'ordre suivant: $\sigma \sqsubseteq \sigma'$ ssi

1. $\sigma' = \top$ ou
2. $\sigma = \bot$ ou
3. $\sigma = \sigma'$

Nous pouvons montrer:

$$(\textbf{State}_\bot^\top, \sqsubseteq) \text{ est un treillis complet.}$$

Ordre sur les fonctions

Soit $\mathcal{D}$ l'ensemble des fonctions totales et continues de $\textbf{State}_\bot^\top$ vers $\textbf{State}_\bot^\top$.
On définit l'ordre suivant sur $\mathcal{D}$:

$f \sqsubseteq g$ ssi $f(\sigma) \sqsubseteq g(\sigma)$, pour tout état σ.

Théorème $(\mathcal{D}, \sqsubseteq)$ est un treillis complet.

Sémantique de boucles

Notation: Etant donné une fonction $f : A \to A$. On dénote par μf, resp. νf, le plus petit point fixe, resp. le plus grand point fixe de f, s'il existe.
On défini

$$\mathcal{S}_d[\![\text{while } b \text{ do } S]\!] = \mu(\lambda f.\text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id)).$$

D'après la théorie des points fixes, $\mathcal{S}_d[\![\text{while } b \text{ do } S]\!]$ est bien défini, si $\lambda f.\text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id))$ est monotone.

Nous montrons qu'elle est même continue.