

1 Sémantique opérationnelle

1.1 À grands pas

[Winskel, 1993, Ch. 2] Évaluation $\langle a, \sigma \rangle \rightarrow n$: l'expression a s'évalue en le nombre n dans l'état σ .

Évaluation des expressions entières :

Constantes $\frac{}{\langle n, \sigma \rangle \rightarrow n}$

Variables $\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}$

Arithmétique $\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1}$
 $\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 * a_1, \sigma \rangle \rightarrow n_0.n_1}$

Évaluation des expressions booléennes :

Constantes $\frac{}{\langle \text{true}, \sigma \rangle \rightarrow \text{true}} \quad \frac{}{\langle \text{false}, \sigma \rangle \rightarrow \text{false}}$

Comparaisons $\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{true}} \text{ si } n_0 \leq n_1$
 $\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{false}} \text{ si } \neg(n_0 \leq n_1)$

Opérateurs logiques $\frac{\langle b_0, \sigma \rangle \rightarrow v_0}{\langle \neg b_0, \sigma \rangle \rightarrow \neg v_0}$
 $\frac{\langle b_0, \sigma \rangle \rightarrow v_0 \quad \langle b_1, \sigma \rangle \rightarrow v_1}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow v_0 \wedge v_1}$
 $\frac{\langle b_0, \sigma \rangle \rightarrow v_0 \quad \langle b_1, \sigma \rangle \rightarrow v_1}{\langle b_0 \vee b_1, \sigma \rangle \rightarrow v_0 \vee v_1}$

Assignment $\frac{\langle a, \sigma \rangle \rightarrow v}{\langle X := a, \sigma \rangle \rightarrow \sigma[X \mapsto v]}$

Skip $\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}$

Séquence $\frac{\langle c_0, \sigma \rangle \rightarrow \sigma' \quad \langle c_1, \sigma' \rangle \rightarrow \sigma''}{\langle c_0; c_1, \sigma \rangle \rightarrow \sigma''}$

Test $\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow \sigma'}$
 $\frac{\langle b, \sigma \rangle \rightarrow \text{false} \quad \langle c_1, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow \sigma'}$
 $\frac{}{\langle b, \sigma \rangle \rightarrow \text{false}}$

Boucle $\frac{}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma}$
 $\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma' \quad \langle \text{while } b \text{ do } c, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma''}$

rq : voir ça comme relation ensembliste entrée/sortie

Équivalence sémantique :

$$c_0 \sim c_1 \iff \forall \sigma, \sigma' (\langle c_0, \sigma \rangle \rightarrow \sigma' \iff \langle c_1, \sigma \rangle \rightarrow \sigma') \quad (1)$$

exercice : donner sémantique de l'appel de procédures
 faire une sémantique pour le cas où l'on a une fonction random()
 donner sémantique de de l'appel de fonctions
 introduire modification de l'état dans les expressions
 introduire cas des fonctions qui plantent dans certains cas
 observer ce que ça donne sur booléens

On pourrait faire une évaluation court-circuitée :

$$\frac{\langle b_0, \sigma \rangle \rightarrow \text{false}}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow \text{false}}$$

$$\frac{\langle b_0, \sigma \rangle \rightarrow \text{true} \quad \langle b_1, \sigma \rangle \rightarrow v_1}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow v_1}$$

$$\frac{\langle b_0 \vee b_1, \sigma \rangle \rightarrow \text{true}}{\langle b_0, \sigma \rangle \rightarrow \text{true}}$$

$$\frac{\langle b_0, \sigma \rangle \rightarrow \text{true} \quad \langle b_1, \sigma \rangle \rightarrow v_1}{\langle b_0 \vee b_1, \sigma \rangle \rightarrow v_1}$$

exemple :

```
i=0;
while (t[i] < 5 && i < n) {
  i=i+1;
}
```

exercices : donner une sémantique opérationnelle pour le throw...catch

1.2 À petit pas

Évaluation des expressions entières :

Constantes $\overline{\langle n, \sigma \rangle \rightarrow_1 \langle n, \sigma \rangle}$

Variables $\overline{\langle x, \sigma \rangle \rightarrow_1 \langle \sigma(x), \sigma \rangle}$

Arithmétique $\overline{\langle a_0, \sigma \rangle \rightarrow_1 \langle a'_0, \sigma' \rangle}$
 $\overline{\langle a_0 + a_1, \sigma \rangle \rightarrow_1 \langle a'_0 + a_1, \sigma' \rangle}$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_1 \langle a'_1, \sigma' \rangle}{\langle n_0 + a_1, \sigma \rangle \rightarrow_1 \langle n_0 + a'_1, \sigma' \rangle}$$

$$\overline{\langle n_0 + n_1, \sigma \rangle \rightarrow_1 \langle n, \sigma \rangle} \quad n = n_0 + n_1$$

Négation $\overline{\langle b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}$
 $\overline{\langle -b, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}$

$$\overline{\langle b, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}$$

$$\overline{\langle -b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}$$

$$\begin{array}{c}
\frac{\langle b_0, \sigma \rangle \rightarrow \text{true}\sigma'}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow b_1\sigma'} \\
\text{Et logique } \frac{\langle b_0, \sigma \rangle \rightarrow \text{false}\sigma'}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow \text{false}\sigma'} \\
\frac{\langle b_0, \sigma \rangle \rightarrow \text{false}\sigma'}{\langle b_0 \vee b_1, \sigma \rangle \rightarrow b_1\sigma'} \\
\text{Ou logique } \frac{\langle b_0, \sigma \rangle \rightarrow \text{true}\sigma'}{\langle b_0 \vee b_1, \sigma \rangle \rightarrow \text{true}\sigma'} \\
\frac{\langle a, \sigma \rangle \rightarrow_1 \langle n, \sigma' \rangle}{\langle X := a, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma'[X \mapsto n] \rangle} \\
\text{Assignment } \frac{\frac{\sigma}{c_0 c_0' \sigma'}}{\langle c_0; c_1, \sigma \rangle \rightarrow_1 \langle c_0'; c_1, \sigma' \rangle} \\
\text{Séquence } \frac{\frac{\frac{\sigma}{c_0 \text{skip} \sigma'}}{\langle c_0; c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma' \rangle}}{\langle b, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle} \\
\text{Test } \frac{\frac{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_1 \langle c_0, \sigma' \rangle}{\langle b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma' \rangle} \\
\text{Boucle } \frac{\frac{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma' \rangle}{\langle b, \sigma' \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle} \quad \langle c; \text{while } b \text{ do } c, \sigma' \rangle \rightarrow_1 \langle \text{skip}, \sigma'' \rangle}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma'' \rangle}
\end{array}$$

rq : voir ça comme relation de pas à pas sur le compteur ordinal du programme + état mémoire, système de transitions sur $\Pi \times \Sigma$

2 Induction

Induction bien fondée

Cas le plus simple : “mesure” $m(x)$ dans \mathbb{N} associé à chaque objet x (exemple : hauteur de la preuve, longueur du mot etc.) et $a < b \iff m(a) < m(b)$

Plus général que simplement associer une “mesure” dans \mathbb{N} ! “Mesure” donne : pour tout élément a , toute chaîne partant de a est de longueur $< m(a)$.

Exemple qui ne passe pas : $\mathbb{N} \cup \{+\infty\}$ ordonné canoniquement, alors partant de $+\infty$ on a des chaînes descendantes $\infty \succ n \succ n-1 \succ \dots \succ 1 \succ 0$ pour tout n .

3 Dénotationnelle

[Winskel, 1993, Ch. 5]

3.1 Sauf le while

$$\llbracket n \rrbracket_A \stackrel{\text{def}}{=} \{(\sigma, n) \mid \sigma \in \Sigma\}$$

$$\llbracket X \rrbracket_A \stackrel{\text{def}}{=} \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}$$

$$\llbracket a_0 + a_1 \rrbracket_A \stackrel{\text{def}}{=} \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A, (\sigma, n_1) \in \llbracket a_1 \rrbracket_A\}$$

$$\llbracket a_0 * a_1 \rrbracket_A \stackrel{\text{def}}{=} \{(\sigma, n_0.n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A, (\sigma, n_1) \in \llbracket a_1 \rrbracket_A\}$$

$$\llbracket \text{true} \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \text{true})\} \quad \llbracket \text{false} \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \text{false})\}$$

$$\llbracket a_0 = a_1 \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \text{true}) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A, (\sigma, n_1) \in \llbracket a_1 \rrbracket_A, n_0 = n_1\}$$

$$\llbracket a_0 \neq a_1 \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \text{false}) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A, (\sigma, n_1) \in \llbracket a_1 \rrbracket_A, n_0 \neq n_1\}$$

$$\llbracket a_0 \leq a_1 \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \text{true}) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A, (\sigma, n_1) \in \llbracket a_1 \rrbracket_A, n_0 \leq n_1\}$$

$$\llbracket a_0 < a_1 \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \text{false}) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A, (\sigma, n_1) \in \llbracket a_1 \rrbracket_A, n_0 \not\leq n_1\}$$

remarque : sens de = peut être différent dans le langage de programmation et en maths (exemple : = sur flottants IEEE n'est pas réflexif à cause de NaN); +, -, × etc en flottants remplacés par ⊕, ⊖, ⊗ pour éviter la confusion

$$\llbracket b_0 \wedge b_1 \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, v_0 \wedge v_1) \mid (\sigma, v_0) \in \llbracket b_0 \rrbracket_B, (\sigma, v_1) \in \llbracket b_1 \rrbracket_B\}$$

$$\llbracket b_0 \vee b_1 \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, v_0 \vee v_1) \mid (\sigma, v_0) \in \llbracket b_0 \rrbracket_B, (\sigma, v_1) \in \llbracket b_1 \rrbracket_B\}$$

$$\llbracket \neg b \rrbracket_B \stackrel{\text{def}}{=} \{(\sigma, \neg \llbracket b \rrbracket_B) \mid (\sigma, b) \in \llbracket b \rrbracket_B\}$$

$$\llbracket \text{skip} \rrbracket_C \stackrel{\text{def}}{=} \{(\sigma, \sigma)\}$$

$$\llbracket X := a \rrbracket_C \stackrel{\text{def}}{=} \{(\sigma, \sigma[X \mapsto n]) \mid (\sigma, n) \in \llbracket a \rrbracket_A\}$$

$$\llbracket c_0; c_1 \rrbracket_C \stackrel{\text{def}}{=} \llbracket c_1 \rrbracket_C \circ \llbracket c_0 \rrbracket_C$$

en définissant la composée de deux relations $A \subseteq X \times Y$ et $B \subseteq Y \times Z$ par $\{(x, z) \in X \times Z \mid \exists y \in Y (x, y) \in A \wedge (y, z) \in B\}$.

$$\llbracket \text{if } b \text{ then } c_0 \text{ else } c_1 \rrbracket_C \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid (\sigma, \text{true}) \in \llbracket b \rrbracket_B \wedge (\sigma, \sigma') \in \llbracket c_0 \rrbracket_C \vee (\sigma, \text{false}) \in \llbracket b \rrbracket_B \wedge (\sigma, \sigma') \in \llbracket c_1 \rrbracket_C\}$$

remarque : sémantique d'une expression booléenne sans effets de bord peut être le domaine où elle est vraie

Ce que l'on voudrait :

Theorème 1. *Pour tout programme P , alors $(\sigma, \sigma') \in \llbracket P \rrbracket_C$ si et seulement si $\langle \sigma, P \rangle \rightarrow \sigma'$.*

C'est assez évident pour toutes les constructions sauf le while.

3.2 Aparté : théorèmes de points fixe

Définition 2. Un ordre partiel complet (cpo) est un ensemble ordonné (L, \sqsubseteq) tel que toute suite croissante $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ a une borne supérieure $\sup_n x_n$ dans L .

Définition 3. Une fonction f d'un cpo L dans lui-même est dite continue si pour toute suite croissante $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, $f(\sup_n x_n) = \sup_n f(x_n)$.

Rq : on dit aussi ω -continue.

Remarque 4. *Toute fonction croissante est continue.*

Theorème 5. *Soit L un cpo avec un plus petit élément \perp , soit f une fonction continue. Alors f a un plus petit point fixe, noté $\text{lfp } f = \sup_n f^n(\perp)$.*

Démonstration. $\perp \sqsubseteq f(\perp)$ donc par récurrence, avec f monotone, $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$.

$$f(\sup_n f^n(\perp)) = \sup_n f \circ f^n(\perp) = \sup_{n>0} f^n(\perp) = \text{lfp } f.$$

Soit x un point fixe de f . $\perp \sqsubseteq x$, donc par récurrence $f^n(\perp) \sqsubseteq f^n(x) = x$.
Donc $\text{lfp } f \sqsubseteq x$. □

Généralisation :

Définition 6. Soit (L, \sqsubseteq) un ensemble ordonné. Un sous-ensemble D de L est dit *dirigé* si pour tout $x, y \in D$ alors il existe un $z \in D$ tel que $x \sqsubseteq z$ et $y \sqsubseteq z$.

Définition 7. Un ensemble ordonné L est dit un cpo si tout ensemble dirigé admet une borne supérieure.

Définition 8. Une fonction f du cpo L au cpo L' est dite *continue* si pour tout ensemble dirigé D , alors $f(\sup D) = \sup f(D)$.

Remarque 9. *Ceci correspond à la notion de continuité par rapport à la topologie suivante : un ensemble O est ouvert si tout ensemble dirigé D tel que $\sup D \in O$ est tel que $D \cap O \neq \emptyset$.*

3.3 While

Suivant la profondeur n de la preuve de $\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow$.

Soit $\phi(T) = \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in \llbracket b \rrbracket_B\} \cup \{(\sigma, \sigma') \in T \circ \llbracket c \rrbracket_C \mid (\sigma, \text{true}) \in \llbracket b \rrbracket_B\}$.

$\phi(\emptyset)$ capture exactement $\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma}$.

$\phi^2(\emptyset)$ capture exactement la précédente règle $\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma_1 \rangle \rightarrow \sigma_1}$
 $\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma_1}$

Plus généralement : $\phi^n(\emptyset)$ capture exactement toutes les dérivations de $\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow$ de taille $\leq n$.

Donc $\bigcup_n \phi^n(\emptyset) = \text{lfp } \phi$ capture exactement les dérivations de profondeur quelconque.

$\llbracket \text{while } b \text{ do } c \rrbracket_C \stackrel{\text{def}}{=} \text{lfp } (T \mapsto \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in \llbracket b \rrbracket_B\} \cup \{(\sigma, \sigma') \in T \circ \llbracket c \rrbracket_C \mid (\sigma, \text{true}) \in \llbracket b \rrbracket_B\})$

exercices :

théorème de Rice-Shapiro

sémantique par continuations : goto et compagnie

call-cc

sémantique en arrière

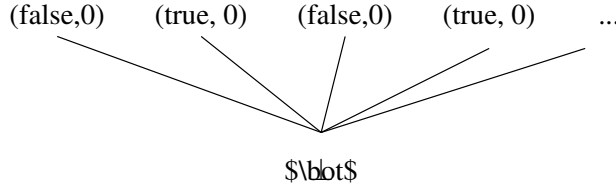


FIG. 1 – Cpo plat construit sur $\{\text{false}, \text{true}\} \times \mathbb{N}$.

4 Dénotationnelle en cpo

$$\begin{aligned}
\llbracket n \rrbracket_a . \sigma &= n \\
\llbracket X \rrbracket_a . \sigma &= \sigma(X) \\
\llbracket a_0 + a_1 \rrbracket_a . \sigma &\stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma + \llbracket a_1 \rrbracket_a . \sigma \\
\llbracket a_0 * a_1 \rrbracket_a . \sigma &\stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma \times \llbracket a_1 \rrbracket_a . \sigma \\
\llbracket \text{true} \rrbracket_b . \sigma &\stackrel{\text{def}}{=} \text{true} \quad \llbracket \text{false} \rrbracket_b . \sigma \stackrel{\text{def}}{=} \text{false} \\
\llbracket a_0 = a_1 \rrbracket_b . \sigma &\stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma =_B \llbracket a_1 \rrbracket_a . \sigma \text{ avec } (v_0 =_B v_1) = \text{true si } v_0 = v_1 \text{ et} \\
&(v_0 =_B v_1) = \text{false sinon;} \\
\llbracket a_0 \leq a_1 \rrbracket_b . \sigma &\stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma \leq_B \llbracket a_1 \rrbracket_a . \sigma \text{ avec } (v_0 \leq_B v_1) = \text{true si } v_0 \leq v_1 \text{ et} \\
&(v_0 \leq_B v_1) = \text{false sinon;}
\end{aligned}$$

remarque : sens de $=$ peut être différent dans le langage de programmation et en maths (exemple : $=$ sur flottants IEEE n'est pas réflexif à cause de NaN); $+$, $-$, \times etc en flottants remplacés par \oplus , \ominus , \otimes pour éviter la confusion

$$\begin{aligned}
\llbracket b_0 \wedge b_1 \rrbracket_b . \sigma &\stackrel{\text{def}}{=} \llbracket b_0 \rrbracket_b . \sigma \wedge \llbracket b_1 \rrbracket_b . \sigma \\
\llbracket b_0 \vee b_1 \rrbracket_b . \sigma &\stackrel{\text{def}}{=} \llbracket b_0 \rrbracket_b . \sigma \vee \llbracket b_1 \rrbracket_b . \sigma \\
\llbracket \neg b \rrbracket_b . \sigma &\stackrel{\text{def}}{=} \neg \llbracket b \rrbracket_b \\
\llbracket P \rrbracket_c : \Sigma &\rightarrow \Sigma_{\perp} \\
\Sigma_{\perp} &= \Sigma \cup \{\perp\}, \text{ ordre } \perp < \sigma \text{ pour tout } \sigma \in \Sigma \\
\llbracket \text{skip} \rrbracket_c . \sigma &\stackrel{\text{def}}{=} \sigma \\
\llbracket X := a \rrbracket_c . \sigma &\stackrel{\text{def}}{=} \sigma[X \mapsto \llbracket a \rrbracket_a . \sigma] \\
\llbracket c_0 ; c_1 \rrbracket_c &\stackrel{\text{def}}{=} \llbracket c_1 \rrbracket_c \circ \llbracket c_0 \rrbracket_c \text{ avec } g \circ f \text{ la composition stricte : si } f(x) = \perp, \text{ alors} \\
&g \circ f = \perp.
\end{aligned}$$

$$\llbracket \text{if } b \text{ then } c_0 \text{ else } c_1 \rrbracket_c . \sigma \stackrel{\text{def}}{=} \text{si } \llbracket \cdot \rrbracket_b \sigma = \text{true alors } \llbracket c_0 \rrbracket_c . \sigma \text{ sinon } \llbracket c_1 \rrbracket_c . \sigma$$

$$\llbracket \text{while } b \text{ do } c \rrbracket_c . \sigma \stackrel{\text{def}}{=} \text{lfp}(\phi \mapsto \sigma \mapsto \text{si } \llbracket \cdot \rrbracket_b \sigma = \text{true alors } \phi \circ \llbracket c \rrbracket_c . \sigma \text{ sinon } \sigma). \sigma$$

extension à fonctions récursives et passage de fonctions : environnements ordonnés coordonnée par coordonnée, scalaires non comparables, fonctions $A \rightarrow B_{\perp}$ ordonnées coordonnée par coordonnée

note : $x \mapsto y$ parfois noté $\lambda x.y$, $\text{lfp}(f \mapsto e)$ parfois noté $\mu f.e$

5 Sémantique axiomatique à la Hoare

[Winskel, 1993, Ch. 6] Floyd-Hoare C.A.R. Hoare (photo) Robert Floyd
Correction *partielle*

Invariant à fournir pour les boucles

Skip $\{A\}\text{skip}\{A\}$

Affectation $\{B[x \mapsto a]\}x := a\{B\}$

$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$

Séquence $\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$

$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\text{if } b \text{ then } c_0 \text{ else } c_1\{B\}}$

Test $\frac{\{A\}\text{if } b \text{ then } c_0 \text{ else } c_1\{B\}}{\{A \wedge b\}c\{A\}}$

$\frac{\{A \wedge b\}c\{A\}}{\{A\}\text{while } b \text{ do } c\{A \wedge \neg B\}}$

While $\frac{\{A \wedge b\}c\{A\}}{\{A\}\text{while } b \text{ do } c\{A \wedge \neg B\}}$

incomplétude de Gödel : le $\models A$ non testable

plus directement : pas de système de preuve effective ; tester si $\{\text{true}\}c\{\text{false}\}$

revient à tester si c boucle tout le temps, ce qui est impossible via Rice

complétude relative : si $\models \{A\}c\{B\}$ alors $\vdash \{A\}c\{B\}$

En exos : correction totale ? Ranking functions, montrer qu'une fois la ranking function fixée c'est une propriété d'accessibilité.

Remarques ranking function :

- si choix non déterministe infini : possibilité d'arbre bien fondé mais de profondeur infini

- si choix non déterministe fini : arbre bien fondé \Rightarrow de profondeur finie

preuve : si arbre de profondeur infinie, construction d'une trace infinie à chaque fois, prendre une sous-branche de profondeur infinie

Si pas de non-déterminisme : nombre total d'exécutions du programme dans le futur calculable à partir de tout état (lancer la machine et compter)

Si non-déterminisme borné : également (lancer la machine pour toutes les possibilités et prendre le max)

Nous définissons la précondition libérale la plus faible $P_{wp}(A)$ d'un prédicat $A \subseteq Y$ par rapport à une relation de transition $P \subseteq X \times Y$ par :

$$P_{wp}(A) = \{x \in X \mid \forall y \in Y (x, y) \in P \Rightarrow y \in A\} \quad (2)$$

Soit $P \subseteq X \times Y$ et $Q \subseteq Y \times Z$, définissons :

$$Q \circ P = \{(x, z) \in X \times Z \mid \exists y \in Y (x, y) \in P \wedge (y, z) \in Q\} \quad (3)$$

Lemme 10. Soit $P \subseteq X \times Y$, $Q \subseteq Y \times Z$, $A \subseteq Z$, alors $(Q \circ P)_{wp}(A) = P_{wp} \circ Q_{wp}(A)$

5.1 Gödel

Lemme 11. Il existe une fonction β calculable à partir des opérateurs $+$ et mod , telle que pour tous $a_0, \dots, a_k \in \mathbb{N}$, il existe m et n tels que $\beta(n, m, i) = a_i$.

L'équation 8 définit une telle fonction et le lemme 19 prouve la propriété désirée. Nous codons ainsi les suites finies d'entier naturels avec les couples d'entiers. Cependant, nous aurons également besoin de coder les suites finies d'entiers relatifs. Nous y parvenons en codant les entiers positifs ou nuls dans les entiers naturels pairs, et les entiers strictement négatifs dans les entiers naturels impairs.

Lemme 12. *Il existe un prédicat $\beta^\pm(n, m, i, v)$ construit à partir des opérateurs usuels tel que :*

- pour toute suite finie $a_0, \dots, a_k \in \mathbb{Z}$, il existe m et n tels que pour tout $0 \leq i \leq k$, $\beta^\pm(n, m, i, a_i)$;
- réciproquement, pour tout couple (n, m) , et tout i , il existe un a_i tel que $\beta^\pm(n, m, i, a_i)$.

Démonstration. Soit $R(x, y)$ ($x \in \mathbb{Z}$ et $y \in \mathbb{N}$) le prédicat $(x < 0 \wedge y = 1 - 2x) \vee (x \geq 0 \wedge y = 2x)$. Le résultat vient avec $\beta^\pm(n, m, k, v)$ défini comme $\exists v' \geq 0 R(v, v') \wedge v' = \beta(n, m, k)$. \square

Ce prédicat β^\pm nous sera très utile pour transformer des quantifications $\exists k \in \mathbb{N} \exists a_0, \dots, a_k \in \mathbb{Z} P(k, (a_i)_{0 \leq i \leq k})$, qui n'appartiennent pas au langage de l'arithmétique de Péano, en quantifications équivalentes $\exists n, m \in \mathbb{N} Q(m, n)$. En particulier, considérons le problème de la clôture transitive d'un prédicat d'avancement P : caractériser l'ensemble des couples (état initial, état final) liés par zéro ou plus transitions, sachant que chaque transition vérifie P .

Lemme 13. *Soit L un ensemble de variables libres. Soit $V = \{v_1, \dots, v_{|V|}\}$ un ensemble fini de variables disjoint de L , $V' = \{v'_1, \dots, v'_{|V|}\}$ une copie de cet ensemble. Soit P un prédicat sur $L \cup V \cup V'$. On peut construire un prédicat P^* sur $L \cup V$ tel que l'ensemble de ses modèles est exactement (l, σ_d, σ_f) tel que :*

$$\exists k \in \mathbb{N} \exists \sigma_0, \dots, \sigma_k \in \mathbb{Z}^V \quad \sigma_d = \sigma_0 \wedge \sigma_f = \sigma_k \wedge \forall 0 \leq i < k \quad (l, \sigma_i, \sigma_{i+1}) \models P \quad (4)$$

Démonstration. Nous allons remplacer, pour chaque variable v_j , la suite $\sigma_i(v_j)$ de ses valeurs au cours du temps par un couple (n_j, m_j) tel que, pour tout i , $\beta^\pm(m_j, n_j, i, \sigma_i(v_j))$:

$$\begin{aligned} P^* &\stackrel{\text{def}}{=} \exists n_1, m_1, \dots, n_{|V|}, m_{|V|} \\ &\quad (\beta^\pm(n_1, m_1, 0, w_1) \wedge \dots \wedge \beta^\pm(n_{|V|}, m_{|V|}, 0, w_{|V|})) \wedge \\ &\quad \exists k \left(\beta^\pm(n_1, m_1, k, w'_1) \wedge \dots \wedge \beta^\pm(n_{|V|}, m_{|V|}, k, w'_{|V|}) \wedge \right. \\ \forall i < k &\Rightarrow \left(\exists v_1, \dots, v_{|V|}, v'_1, \dots, v'_{|V|} \beta^\pm(n_1, m_1, i, v_1) \wedge \dots \wedge \beta^\pm(n_{|V|}, m_{|V|}, i, v_{|V|}) \wedge \right. \\ &\quad \left. \left. \beta^\pm(n_1, m_1, i+1, v'_1) \wedge \dots \wedge \beta^\pm(n_{|V|}, m_{|V|}, i+1, v'_{|V|}) \wedge P \right) \right) \quad (5) \end{aligned}$$

Ce prédicat a pour variables libres $L \cup \{w_1, \dots, w_{|V|}, w'_1, \dots, w'_{|V|}\}$; ses modèles $(l, w, w') \models P^*$ sont exactement ceux demandés. \square

Lemme 14. Soit L un ensemble de variables libres. Soit $V = \{v_1, \dots, v_{|V|}\}$ un ensemble fini de variables disjoint de L , $V' = \{v'_1, \dots, v'_{|V|}\}$ une copie de cet ensemble. Soit P un prédicat sur $L \cup V \cup V'$, soient A, B deux prédicats sur $L \cup V$. On peut construire un prédicat $wpIter(P, A, B)$ sur $L \cup V$ tel que l'ensemble de ses modèles est exactement (l, σ_d) tel que :

$$\begin{aligned} \forall k \in \mathbb{N} \forall \sigma_0, \dots, \sigma_k \in \mathbb{Z}^V \quad & (\sigma_d = \sigma_0 \wedge \sigma_k \models \neg A \wedge \\ & \forall 0 \leq i < k \quad (l, \sigma_i) \models A \wedge (l, \sigma_i, \sigma_{i+1}) \models P) \implies (l, \sigma_k) \models B \end{aligned} \quad (6)$$

Ceci correspond à $(P \circ \phi_A)_{wp}^*(B \setminus A)$.

Démonstration. Nous allons remplacer, pour chaque variable v_j , la suite $\sigma_i(v_j)$ de ses valeurs au cours du temps par un couple (n_j, m_j) tel que, pour tout i , $\beta^\pm(m_j, n_j, i, \sigma_i(v_j))$:

$$\begin{aligned} \forall n, m, k \forall v_1, \dots, v_{|V|} \quad & \left((\beta^\pm(n, m, 0, w_1) \wedge \dots \wedge \beta^\pm(n, m, 0, w_{|V|})) \right. \\ \wedge \quad & (\forall i \forall v_1, \dots, v_{|V|}, v'_1, \dots, v'_{|V|} \quad (i < k \wedge \beta^\pm(n, m, i, v_1) \wedge \dots \wedge \beta^\pm(n, m, i, v_{|V|}) \wedge \\ & \beta^\pm(n, m, i+1, v'_1) \wedge \dots \wedge \beta^\pm(n, m, i+1, v'_{|V|})) \implies P) \\ & \left. \wedge \beta^\pm(n, m, k, v_1) \wedge \dots \wedge \beta^\pm(n, m, k, v_{|V|}) \wedge \neg A \right) \implies B \end{aligned} \quad (7)$$

Ce prédicat a pour variables libres $L \cup \{w_1, \dots, w_{|V|}\}$; ses modèles $(l, w) \models P^*$ sont exactement ceux demandés. \square

Lemme 15. Si W est un prédicat de la logique sur les états, alors pour tout programme P , $\llbracket P \rrbracket_{wp}.W$ est aussi un prédicat de la logique sur les états.

Squelette : : facile pour composition, tests etc. Pour boucle : par le lemme précédent $\llbracket (\llbracket c \rrbracket_C \circ \phi_{\llbracket b \rrbracket_B})^* \rrbracket_{wp} (W \setminus \llbracket b \rrbracket_B)$ est représentée par $A = wpIter(\llbracket c \rrbracket_C, \llbracket b \rrbracket_B, W)$.

Theorème 16. Pour tous prédicats A et B , si $\{A\}P\{B\}$ est vrai, alors on peut le démontrer sous réserve que l'on puisse démontrer les implications de l'arithmétique.

Squelette : : facile pour composition, tests etc. Pour boucle :

Par définition de $\llbracket (\llbracket c \rrbracket_C \circ \phi_{\llbracket b \rrbracket_B})^* \rrbracket_{wp} (B)$, $\{A \wedge b\}c\{A\}$, et il existe une preuve cette assertion, par hypthèse de récurrence. Donc nous avons une preuve de $\{A\}while \ b \ do \ c\{A \wedge \neg b\}$. Par ailleurs, par définition de $\llbracket (\llbracket c \rrbracket_C \circ \phi_{\llbracket b \rrbracket_B})^* \rrbracket_{wp} (B)$, $A \subseteq B$. On peut donc montrer $\{A\}while \ b \ do \ c\{B\}$ sous réserve que l'on sache montrer $A \implies B$.

Remarque : on peut coder de la même façon “il existe une trace finie de longueur k ” donc : “pour toute longueur k il existe une trace finie de longueur k ” c’est-à-dire “il existe des traces de longueur non bornée”. = “profondeur infinie”

Sur un système à non-déterminisme fini, c’est équivalent à “il existe une trace infinie”.

Preuve :

- s’il existe une trace infinie alors profondeur infinie ;
- si la profondeur est infinie : si on a un ensemble borné de choix, un de ces choix mène sur une sous-branche de profondeur infinie, prendre celle-ci ; on construit par récurrence une trace infinie.

exercices :

sémantique dénotationnelle définie par formules de l’arithmétique liant entrée et sortie ; complétude relative

cas des programmes à variables réelles : trouver un exemple de programme dont entrée et sortie non liés par théorie des corps réels clos

quid de la complexité du problème “j’ai n booléens en entrée et une sortie booléenne” par rapport à n ?

quid de la calculabilité si on remplace les variables entières par des booléennes ?

quid de la garbage collection ?

6 Analyse statique

problème de l’arrêt

Théorème de Rice

dataflow analysis

système d’équation

interprétation abstraite

élargissement

flottants

7 Typage

Mitchell [1996]

$x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash t : \tau$ veut dire “ t a le type τ dans un contexte où x_1 a le type σ_1, \dots, x_n a le type σ_n ”.

$$\frac{\Gamma, t : \tau \vdash t : \tau}{\Gamma} \text{ var}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma, x : \sigma \vdash t : \tau} \text{ add var} \quad \text{si } x \notin \Gamma.$$

$$\frac{\Gamma \vdash (\lambda x : \sigma. t) : \sigma \rightarrow \tau}{\Gamma \vdash a : \sigma \rightarrow \tau \quad \Gamma \vdash b : \sigma} \rightarrow \text{ intro}$$

$$\frac{\Gamma \vdash a : \sigma \rightarrow \tau \quad \Gamma \vdash b : \sigma}{\Gamma \vdash (a \text{ } b) : \tau} \rightarrow \text{ elim}$$

$$\text{Ajout des types produits (} n\text{-uplets)} \quad \frac{\Gamma \vdash a : \sigma \quad \Gamma \vdash b : \tau}{\Gamma \vdash (a, b) : \sigma \times \tau} \times \text{ intro}$$

$$\frac{\Gamma \vdash \text{proj}_1^{\sigma, \tau} : \sigma}{\Gamma \vdash a : \sigma \times \tau} \times \text{ elim}_1$$

$$\frac{\Gamma \vdash \text{proj}_2^{\sigma, \tau} : \tau}{\Gamma \vdash a : \sigma \times \tau} \times \text{ elim}_2$$

Ajout des types sommes

```

type t =
  A of a
  | B of b


$$\frac{\Gamma \vdash \sigma : a}{\Gamma \vdash \text{inj}_1^{\sigma, \tau}(a) : \sigma + \tau} + \text{intro}_1$$


$$\frac{\Gamma \vdash b : \tau}{\Gamma \vdash \text{inj}_2^{\sigma, \tau} : \sigma + \tau} + \text{intro}_2$$


match v with
  A a -> f a
  | B b -> g b


$$\frac{\Gamma \vdash v : \sigma + \tau \quad \Gamma \vdash f : \sigma \rightarrow \rho \quad \Gamma \vdash g : \sigma \rightarrow \rho}{\Gamma \vdash \text{case}^{\sigma + \tau \rightarrow \rho} v f g : \rho} + \text{elim}$$

polymorphisme : prédictatif et imprédicatif
synthèse de types

# let f x = x;;
val f : 'a -> 'a = <fun>

système F /  $\lambda$ -calcul du second ordre  $\Lambda T.\lambda x.x$ 
sous-typage : types produits avec variables de rangée
et pour les langages impératifs?
cas de Caml : typage des références

# let f =
  let r = ref None in
  function toto ->
    let old = !r in r := toto; old;;
val f : '_a option -> '_a option = <fun>
# f (Some "toto");;
- : string option = None
# f;;
- : string option -> string option = <fun>

si on lit : contravariance, si on écrit, covariance; lire = paramètre
en Java, type checking passe

class PolyArrays {
  public static void main(String[] args) {
Integer[] ia = new Integer[10];
ia[0] = new Integer(15);
Object[] oa = ia;
oa[0] = "toto";
Integer x = ia[0];
System.out.println(x);
  }
}

```

```
$ java PolyArrays
Exception in thread "main" java.lang.ArrayStoreException: java.lang.String
    at PolyArrays.main(PolyArrays.java:6)
```

typage : simuler le let rec à l'aide des références et à l'aide des types inductifs
à occurrences négatives

8 Compilation

vectorisation
common subexpression elimination
reaching definitions
live variables \Rightarrow coloriage
strength reduction (calcul de deltas entre itérations)
constant propagation

A Arithmétique

Lemme 17. *Soient a et b deux entiers premiers entre eux. Soient u et v deux entiers. Alors il existe n tel que $n \equiv u \pmod{a}$ et $n \equiv v \pmod{b}$.*

Démonstration. Par l'algorithme d'Euclide étendu, on peut calculer un couple de Bézout (c, d) tel que $ac + bd = 1$. $n = acu + bdv$ convient. \square

Corollaire immédiat par récurrence :

Corollaire 18. *Soient a_1, \dots, a_n des entiers premiers entre eux deux à deux. Soient u_1, \dots, u_n des entiers. Alors il existe n tel que pour tout i , $n \equiv u_i \pmod{a_i}$.*

$$\beta(n, m, i) = n \bmod (1 + m(i + 1)) \quad (8)$$

Lemme 19. *Soient $a_0, \dots, a_k \in \mathbb{N}$. Alors il existe n, m tels que pour tout $0 \leq i \leq k$, $\beta(n, m, i) = a_i$.*

Démonstration. Soit $m = \max(k, a_0, \dots, a_k)!$. Soit $u_i = 1 + m(i + 1)$.

Soit p un nombre premier. Si $p \mid u_i$, alors $p \nmid m$. Si $p \mid u_i$ et $p \mid u_j$, alors $p \mid u_i - u_j = m(i - j)$; comme $p \nmid m$ alors $p \mid i - j$. Mais $|i - j| \leq k \leq \max(k, a_0, \dots, a_k)$ donc $i - j \mid m$; mais alors $p \mid m$ ce qui est absurde. Donc, u_i et u_j sont premiers entre eux.

Par le lemme chinois (corollaire 18), il existe un n tel que pour tout i , $n \equiv a_i \pmod{u_i}$. Or, $a_i < u_i$, donc $a_i = n \bmod u_i$. \square

Références

John C. Mitchell. *Foundations for programming languages*. Foundations of Computing. MIT Press, 1996.

Glynn Winskel. *The Formal Semantics of Programming Languages : An Introduction*. Foundations of Computing. MIT Press, 1993. ISBN 0-262-23169-7.