

# Sémantique, analyse statique et typage

**David Monniaux**

`http://www.di.ens.fr/~monniaux`

*Centre National de la Recherche Scientifique*

École Normale Supérieure

Département d'Informatique

45, rue d'Ulm

75230 Paris cedex 5

# Sémantique ?

# Syntaxe vs sémantique

## Syntaxe

code source du programme (arbre syntaxique)

## Sémantique d'un programme

définition mathématique des comportements possibles

## Sémantique d'un langage

définition de la sémantique des programmes à partir de leur syntaxe

# Applications

Preuve de programmes « le programme fait ce qu'on lui demande »  
par rapport à une spécification mathématique  
ou critères simples (« ne plante pas »)

Optimisation optimisations demandent de vérifier des conditions

ex : omettre un test si on reste dans un tableau

# Syntaxe

## Syntaxe concrète

suite de caractères (non utilisée ici)

## Syntaxe abstraite

on part de l'arbre syntaxique

## Choix du langage

while-language

# While language

## Expressions arithmétiques

$+$ ,  $-$ ,  $*$ , variables, constantes entières

## Expressions booléennes

comparaisons  $=$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$

opérations  $\wedge$ ,  $\vee$ ,  $\neg$

Instructions skip ne rien faire

$X := a$  affecter  $a$  dans  $X$

$c_0; c_1$  faire  $c_0$  ensuite  $c_1$

# Contrôle de flot

## Test

if  $b$  then  $c_0$  else  $c_1$  : si  $b$  vrai alors  $c_0$  sinon  $c_1$

## Boucle

while  $b$  do  $c$  : si  $b$  est vrai, faire  $c$  et recommencer

for, do...while sont émulées

# Sémantique opérationnelle

# Idée de base

Plotkin *structural operational semantics* = S.O.S.

[?]

Exécutions possibles = prouvables via système de règles

$$\frac{H_1 \quad \dots \quad H_n}{C} R$$

« La conclusion  $C$  se déduit des hypothèses  $H_1$  à  $H_n$  par la règle  $R$ . »

# Arbres de preuves

$$\frac{\frac{\overline{A_{1,1}} \quad A_1 \quad H_{1,2}}{L_1} \quad R_1 \quad H_2}{C} R_0$$

$C$  se déduit de  $L_1$  (lemme) et de  $H_2$  hypothèse par  $R_0$ .  
 $L_1$  se déduit de  $A_{1,1}$  axiome et de l'hypothèse  $H_{1,2}$ .

Axiome = règle sans hypothèses

# Sémantique opérationnelle : arithmétique

$\sigma$  : fonction partielle des variables dans les entiers

$\sigma(v)$  = valeur de la variable  $v$

$\langle e, \sigma \rangle \rightarrow n$  = « l'expression  $e$  s'évalue en l'entier  $n$  dans l'environnement  $\sigma$  »

$$\overline{\langle n, \sigma \rangle \rightarrow n}$$

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

# Sémantique opérationnelle : arithmétique

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

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

# Sémantique opérationnelle : booléens

$\langle e, \sigma \rangle \rightarrow b =$  « l'expression  $e$  s'évalue en le booléen  $b$  dans l'environnement  $\sigma$  »

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

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

# Sémantique opérationnelle : booléens

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{true}} \text{ si } n_0 \leq n_1$$

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

# Sémantique opérationnelle : booléens

$$\frac{\langle b_0, \sigma \rangle \rightarrow v_0}{\langle \neg b_0, \sigma \rangle \rightarrow \neg v_0}$$

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

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

# Sémantique opérationnelle : instructions

$\langle P, \sigma \rangle \rightarrow \sigma' =$  « le programme  $P$ , s'il part de l'état mémoire  $\sigma$ , peut arriver en l'état mémoire  $\sigma'$  en fin d'exécution »

$$\overline{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}$$

# Sémantique opérationnelle : instructions

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

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

# Sémantique opérationnelle : tests

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

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

# Sémantique opérationnelle : boucles

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma}$$

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

# Exercices

Donner une sémantique pour l'appel de procédures

Donner une sémantique pour l'appel de fonctions  
(ce que cela implique pour les expressions)

Réfléchir sur les opérations booléennes

Donner une sémantique du throw...catch

Sémantique à petits pas

# Sémantique opérationnelle à petits pas

# Petits pas : entiers

$\langle e, \sigma \rangle \rightarrow_1 \langle e', \sigma' \rangle =$  « l'expression  $e$ , dans l'environnement  $\sigma$ , se transforme en l'expression  $e'$  en un pas d'exécution, et l'environnement passe à  $\sigma'$  »

Si  $e'$  est un entier on a terminé pour cette expression.

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

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

# Petits pas : arithmétique

$$\frac{\langle a_0, \sigma \rangle \rightarrow_1 \langle n_0, \sigma' \rangle}{\langle a_0 + a_1, \sigma \rangle \rightarrow_1 \langle n_0 + a_1, \sigma' \rangle}$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow_1 \langle n_0, \sigma' \rangle}{\langle a_0 + a_1, \sigma \rangle \rightarrow_1 \langle n_0 + a_1, \sigma' \rangle}$$

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

# Petits pas : booléens $\neg$

$\langle e, \sigma \rangle \rightarrow_1 \langle e', \sigma' \rangle =$  « l'expression  $e$ , dans l'environnement  $\sigma$ , se transforme en l'expression  $e'$  en un pas d'exécution, et l'environnement passe à  $\sigma'$  »

Si  $e'$  est un booléen on a terminé pour cette expression.

$\langle e, \sigma \rangle \rightarrow_1 \langle e', \sigma' \rangle =$  « l'expression  $e$ , dans l'environnement  $\sigma$ , se transforme en l'expression  $e'$  en un pas d'exécution, et l'environnement passe à  $\sigma'$  »

# Petits pas : booléens $\neg$

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}{\langle \neg b, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}{\langle \neg b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma \rangle}$$

# Petits pas : booléens $\wedge$

Court-circuitée

$$\frac{\langle b_0, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow_1 \langle b_1, \sigma' \rangle}$$

$$\frac{\langle b_0, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}$$

# Petits pas : booléens $\vee$

Court-circuitée

$$\frac{\langle b_0, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}{\langle b_0 \vee b_1, \sigma \rangle \rightarrow_1 \langle b_1, \sigma' \rangle}$$

$$\frac{\langle b_0, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}{\langle b_0 \vee b_1, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}$$

# Petits pas : instructions

$\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle =$  « quand on exécute la commande  $c$ , dans l'environnement  $\sigma$ , après un pas d'exécution on se retrouve à exécuter  $c'$  dans l'environnement  $\sigma'$  »

Si  $c' = \text{skip}$  on a terminé pour cette expression.

$$\frac{\langle a, \sigma \rangle \rightarrow_1 \langle n, \sigma' \rangle}{\langle X := a, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma' [X \mapsto n] \rangle}$$

# Petits pas : séquence

$$\frac{\langle c_0, \sigma \rangle \rightarrow_1 \langle c'_0, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow_1 \langle c'_0; c_1, \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma' \rangle}$$

# Petits pas : test

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_1 \langle c_0, \sigma' \rangle}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma' \rangle}$$

# Petits pas : boucles

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle \text{false}, \sigma' \rangle}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma' \rangle}$$

$$\frac{\langle b, \sigma' \rangle \rightarrow_1 \langle \text{true}, \sigma' \rangle \quad \langle c; \text{while } b \text{ do } c, \sigma' \rangle \rightarrow_1 \langle \text{skip}, \sigma'' \rangle}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma'' \rangle}$$

# Remarque

En pratique, on se ramène souvent à état =  $(p, \sigma)$

$p$  = pointeur d'instruction

$\sigma$  = état mémoire

# Exercices

appels de procédure, return

appels de fonctions

pointeurs sur fonctions

exceptions

break, goto etc.

# Sémantique dénotationnelle

# Idées de base

## Compositionnelle

Sémantique du tout définie par sémantique des parties

## Identifie programmes équivalents

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

Relation entrée/sortie

?, Ch. 5

# Dénotationnelle simple : expressions

$\llbracket e \rrbracket_A = \{(\sigma, n) \mid e \text{ s'évalue en l'entier } n \text{ dans le contexte } \sigma\}$

$\llbracket \cdot \rrbracket$  standard pour sémantiques dénotationnelles

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

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

# Dénotationnelle simple : expressions

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

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

# Dénotationnelle simple : booléens

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

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

# Dénotationnelle simple : comparaisons

Rq : en flottants, NaN = NaN rend false, = non réflexif

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

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

# Dénotationnelle simple : comparaisons

Rq : en flottants, NaN  $\leq$  NaN rend false, = non réflexif

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

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

# Parenthèse sur les flottants

Le  $=$  et le  $\leq$  en flottants ne sont pas réflexifs.à cause du NaN.

Le  $+$ ,  $*$ , ne sont pas associatifs!!!

On écrira donc  $\oplus$ ,  $\ominus$ ,  $\otimes$ ...

$(x \oplus y) \ominus y$  pas forcément  $= x$

```
# (1.0 +. 1E20) -. 1E20 = 0.;;
```

```
- : bool = true
```

**Attention confusion** objets langage de programmation  $\neq$  objets mathématiques qui « ressemblent »

ex : non terminaison

# Dénotationnelle simple : booléens

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

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

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

# Dénotationnelle simple : instructions

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

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

# Dénotationnelle simple : séquence

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

avec la composée de deux relations  $A \subseteq X \times Y$  et  $B \subseteq Y \times Z$  :

$$\{(x, z) \in X \times Z \mid \exists y \in Y (x, y) \in A \wedge (y, z) \in B\}$$

# Dénotationnelle simple : test

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

# Ce que l'on voudrait

Pour tout programme  $P$  :

$$(\sigma, \sigma') \in \llbracket P \rrbracket_C \iff \langle \sigma, P \rangle \rightarrow \sigma'$$

Pour le while, suggère point fixe :

$$\begin{aligned} \llbracket \text{while } b \text{ do } c \rrbracket_C &= \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in \llbracket b \rrbracket_B\} \\ &\cup \{(\sigma, \sigma') \in \llbracket \text{while } b \text{ do } c \rrbracket_C \circ \llbracket c \rrbracket_C \mid (\sigma, \text{true}) \in \llbracket b \rrbracket_B\} \end{aligned}$$

# Théorèmes de point fixe

# Cpo's

Ordre partiel complet (cpo) = 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  $L$ .

Fonction continue = pour toute suite croissante  $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ ,  
 $f(\sup_n x_n) = \sup_n f(x_n)$

En général : cette définition =  $\omega$ -continue, continue = pour tout ensemble dirigé.

Toute fonction continue est croissante ; Réciproque vraie sur les domaines finis.

# Plus petit point fixe

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

Preuve  $\perp \sqsubseteq f(\perp)$  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  $\text{lfp } f \sqsubseteq x$ .

# Sémantique dénotationnelle simple : while

# Dérivation

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

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

$\phi(\emptyset)$  capture exactement

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma}$$

# Dérivation

$\phi^2(\emptyset)$  capture exactement la précédente règle +

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma_1 \quad \frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma_1 \rangle \rightarrow \sigma'}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'}$$

$\sigma_1$  = état après un tour d'exécution

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

# Point fixe

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

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

# Exercices

fonctions récursives

théorème de Rice-Shapiro

sémantique par continuations : call-cc, exceptions

goto et compagnie

# Sémantique dénotationnelle sur les cpo

# Arithmétique

$\llbracket n \rrbracket_A . \sigma =$  résultat de l'évaluation de  $n$  dans  $\sigma$

$$\llbracket n \rrbracket_a . \sigma = n$$

$$\llbracket X \rrbracket_a . \sigma = \sigma(X)$$

# Arithmétique

$$\llbracket a_0 + a_1 \rrbracket_a . \sigma \stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma + \llbracket a_1 \rrbracket_a . \sigma$$

$$\llbracket a_0 * a_1 \rrbracket_a . \sigma \stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma \times \llbracket a_1 \rrbracket_a . \sigma$$

# Booléens

$$\llbracket \text{true} \rrbracket_b . \sigma \stackrel{\text{def}}{=} \text{true}$$

$$\llbracket \text{false} \rrbracket_b . \sigma \stackrel{\text{def}}{=} \text{false}$$

$\llbracket a_0 = a_1 \rrbracket_b . \sigma \stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma =_B \llbracket a_1 \rrbracket_a . \sigma$  avec  $(v_0 =_B v_1) = \text{true}$  si  $v_0 = v_1$   
et  $(v_0 =_B v_1) = \text{false}$  sinon ;

$\llbracket a_0 \leq a_1 \rrbracket_b . \sigma \stackrel{\text{def}}{=} \llbracket a_0 \rrbracket_a . \sigma \leq_B \llbracket a_1 \rrbracket_a . \sigma$  avec  $(v_0 \leq_B v_1) = \text{true}$  si  $v_0 \leq v_1$   
et  $(v_0 \leq_B v_1) = \text{false}$  sinon ;

# Booléens

$$\llbracket b_0 \wedge b_1 \rrbracket_b . \sigma \stackrel{\text{def}}{=} \llbracket b_0 \rrbracket_b . \sigma \wedge \llbracket b_1 \rrbracket_b . \sigma$$

$$\llbracket b_0 \vee b_1 \rrbracket_b . \sigma \stackrel{\text{def}}{=} \llbracket b_0 \rrbracket_b . \sigma \vee \llbracket b_1 \rrbracket_b . \sigma$$

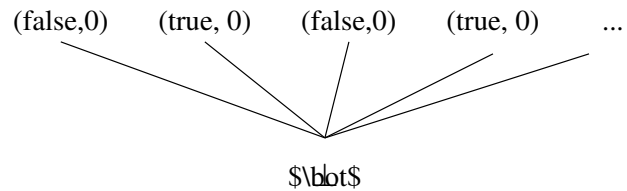
attention à  $\perp$  et court-circuit : strict vs non-strict

$$\llbracket \neg b \rrbracket_b . \sigma \stackrel{\text{def}}{=} \neg \llbracket b \rrbracket_b$$

# Cpo avec $\perp$

$$\llbracket P \rrbracket_c : \Sigma \rightarrow \Sigma_{\perp}$$

$$\Sigma_{\perp} = \Sigma \cup \{\perp\}, \text{ ordre } \perp < \sigma \text{ pour tout } \sigma \in \Sigma$$



$$\llbracket P \rrbracket_c . \sigma = \perp \text{ si commande ne termine pas}$$

# Instructions

$$\llbracket \text{skip} \rrbracket_c . \sigma \stackrel{\text{def}}{=} \sigma$$

$$\llbracket X := a \rrbracket_c . \sigma \stackrel{\text{def}}{=} \sigma[X \mapsto \llbracket a \rrbracket_a . \sigma]$$

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

avec  $g \circ f$  la composition stricte : si  $f(x) = \perp$ , alors  $g \circ f = \perp$

# Flot de contrôle

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

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

# Exercices

passage de fonctions en paramètres

fonctions récursives

sémantique par continuations : goto, exceptions, etc.

# Sémantique axiomatique à la Hoare

# Présentation

[?, Ch. 6]

Floyd-Hoare : C.A.R. Hoare = Tony Hoare +  
Robert Floyd

$\{A\}P\{B\} =$

si condition  $A$  valide, alors toute exécution de  $P$  qui termine donne un résultat vérifiant  $B$

$$\{A\}P\{B\} \iff \forall \sigma, \sigma', \sigma \models A \wedge \langle P, \sigma \rangle \rightarrow \sigma' \Rightarrow \sigma' \models B$$

# Correction partielle vs totale

En correction totale :  $[A] P [B]$

si condition  $A$  valide, alors toute exécution de  $P$  termine et donne un résultat vérifiant  $B$

Pour programmes déterministes :

$$\{A\}P\{B\} \iff \forall \sigma \ \sigma \models A \Rightarrow \exists \sigma' \ \langle P, \sigma \rangle \rightarrow \sigma' \wedge \sigma' \models B$$

# Instructions

$$\{A\}\text{skip}\{A\}$$
$$\{B[x \mapsto a]\}x := a\{B\}$$

# Flot de contrôle

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

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

$A$  est un **invariant de boucle** =  
vrai en tête de boucle à tous tours d'exécution

# Extension

$$\frac{\vdash A \implies A' \quad \{A'\}P\{B'\} \quad \vdash B' \implies B}{\{A\}P\{B\}}$$

$\vdash F \implies G$  = « on peut prouver  $F \implies G$  avec les règles de l'arithmétique »

# Complétude

Pour toutes instructions hors `while do` : si on a  $P$  et  $B$  alors on trouve un  $A$  qui convient (et est optimal) : *weakest precondition*.

Par exemple :

$$\frac{\{H_1\}c_0\{B\} \quad \{H_2\}c_1\{B\}}{\{(H_1 \wedge b) \vee (H_2 \wedge \neg b)\}\text{if } b \text{ then } c_0 \text{ else } c_1\{B\}}$$

C'est valable aussi pour les boucles !

# Théorème de Cook

Si  $B$  formule de l'arithmétique et  $P$  programme, alors on peut obtenir mécaniquement une formule  $A$  telle que

$$\{A\}P\{B\}$$

Et même :

$$\{\sigma \mid \sigma \models A\} = \llbracket P \rrbracket_{wp} \{\sigma' \mid \sigma' \models B\}$$

$$\llbracket P \rrbracket_{wp} (B) \stackrel{\text{def}}{=} \{\sigma \mid \forall \sigma' \langle \sigma, P \rangle \rightarrow \sigma' \implies \sigma' \in B\}$$

# Squelette de preuve

Pour opérations hors while : règles sont conditions nécessaires et suffisantes

# Bibliographie

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