Sémantiques mécanisées: quand la machine raisonne sur ses langages

Introduction

Xavier Leroy

2019-11-28

Collège de France, chaire de sciences du logiciel

La sémantique d'un langage de programmation

Donner un sens aux programmes

Assigning meaning to programs (Floyd, 1967)

Plus modestement : savoir répondre à la question

≪Que fait ce programme, au juste ?≫

Que fait ce programme, au juste ?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>1/2)*
(D[I-1+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

(Raymond Cheong, IOCCC 2001)

Que fait ce programme, au juste ?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>1/2)*
(D[I-1+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

(Raymond Cheong, IOCCC 2001)

(Il calcule des racines carrées en précision arbitraire.)

Et celui-ci ?

```
#define _ F-->00 || F-OO--;
long F=00,OO=00;
main()F_OO();printf("%1.3f\n", 4.*-F/OO/OO);F_OO()
{
            _-_-_-_
        _-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_
   _-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
   _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_
            _-_-_-_
}
```

(Brian Westley, IOCCC 1988)

Et celui-ci ?

```
#define _ F-->00 || F-OO--;
long F=00,OO=00;
main()F_OO();printf("%1.3f\n", 4.*-F/OO/OO);F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_
           _-_-_-_-_
}
```

(Brian Westley, IOCCC 1988)

(Il calcule une valeur approchée de π)

Et celui-là ?

```
 #define crBegin static int state=0; switch(state) { case 0:
 #define crReturn(x) do { state=__LINE__; return x; \
                          case __LINE__:; } while (0)
 #define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham,

auteur du logiciel PuTTY)

Et celui-là ?

```
 #define crBegin static int state=0; switch(state) { case 0:
 #define crReturn(x) do { state=__LINE__; return x; \
                          case __LINE__:; } while (0)
 #define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham,
auteur du logiciel PuTTY)

(C’est un décompresseur pour le *run-length encoding* en style ≪co-routine≫.)

Trois degrés de sémantique

Sémantique intuitive :
un code bien écrit dans un langage adapté raconte une histoire et doit se lire aisément.

Sémantique précise :
manuels de référence, standards ISO, autres textes normatifs.

Sémantique formelle : (l'objet de ce cours)
décrire ce que fait un programme avec la précision absolue des mathématiques.

Une brève histoire des langages de programmation et de leur sémantique

La préhistoire : le langage machine

≪L'informatique c'est rien que des zéros et des uns!≫

```
10111000 00000001 00000000 00000000 00000000
10111010 00000010 00000000 00000000 00000000
00111001 11011010 01111111 00000110
00001111 10101111 11000010
01000010 11101011 11110110
11000011
```

(code machine x86 pour la fonction factorielle)

L'Antiquité (1949) : langages d'assemblage

Une représentation textuelle du langage machine, avec des mnémoniques pour les instructions, des étiquettes pour les points de code, et des commentaires pour les humains.

Exemple : la fonction factorielle en assembleur x86

```
; Entrée: argument N dans registre EBX
; Sortie: factorielle de N dans registre EAX
factorielle:
        mov eax, 1      ; résultat initial = 1
        mov edx, 2      ; indice i = 2
L1:     cmp edx, ebx    ; tant que i <= N ...
        jg L2
        imul eax, edx   ; multiplier résultat par i
        inc edx         ; incrémenter i
        jmp L1          ; fin tant que
L2:     ret             ; fin fonction
```

Une sémantique très précise!

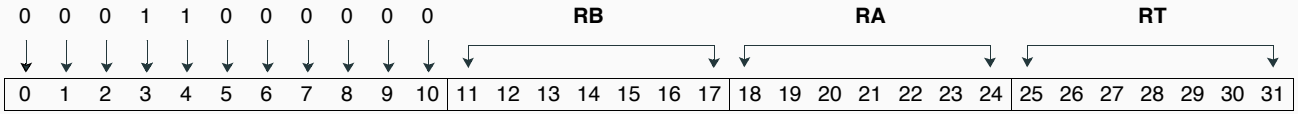
Exprimée comme l'effet de chaque instruction sur l'état du processeur. Pas/peu d'ambiguïtés si on connaît bien l'architecture du matériel.

Instruction Set Architecture

Synergistic Processor Unit

Add Word **Required** **v 1.0**

a **rt,ra,rb**



For each of four word slots:

- The operand from register RA is added to the operand from register RB.
- The 32-bit result is placed in register RT.
- Overflows and carries are not detected.

La Renaissance (1957) : Fortran

Des expressions arithmétiques qui se rapprochent des formules mathématiques usuelles :

```
D  = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Une construction de contrôle structurée : la boucle

```
    DO 10 I=1,N
    ...
10  CONTINUE
```

(Plus `GO TO` et `IF` comme en assembleur.)

Une syntaxe et une sémantique moins claires

Conventions lexicales peu lisibles :

`DO10I=1,20`	boucle de 1 à 20
`DO10I=1.20`	affectation de 1.20 à la variable `DO10I`

Priorités et associativités des opérateurs :

`A + B * C` est `A + (B * C)` et non pas `(A + B) * C`
`A - B - C` est `(A - B) - C` et non pas `A - (B - C)`

Le compilateur peut ≪associer≫ `A + B + C` comme `(A + B) + C` ou comme `A + (B + C)` ou comme `(A + C) + B`.
En virgule flottante, ça calcule des valeurs différentes.

Les Lumières (1960) : Algol

Expressions arithmétiques + contrôle structuré
(avec des mots-clés qui racontent une petite histoire en anglais :
`begin...end`, `if...then...else`, `for...do`, etc).

Les procédures et des fonctions pour rendre le code réutilisable :

```
procedure quadratic(x1, x2, a, b, c);
     value a, b, c; real a, b, c, x1, x2;
begin
  real d;
  d := sqrt(b * b - 4 * a * c);
  x1 := (-b + d) / (2 * a);
  x2 := (-b - d) / (2 * a)
end;
```

Quelle(s) sémantique(s) pour les appels de fonctions ?

Algol 60 offre deux sémantiques pour le passage des arguments aux fonctions, les deux sémantiques les plus simples à exprimer à l'époque :

- passage par valeur pour les paramètres marqués `value`
  ($\approx$ Lisp, C, Java, Caml, …)
  ($\approx$ λ-calcul en appel par valeur)
- substitution du paramètre par l'expression argument pour les autres paramètres, aussi appelée *copy rule*
  ($\approx$ macros en Lisp)
  ($\approx$ λ-calcul en appel par nom).

Le mélange *copy rule* + affectations se révèle explosif !

Grandeur de la *copy rule*

Une fonction de sommation très générale :

```
real procedure Sum(k, l, u, ak)
      value l, u;  integer k, l, u;  real ak;
   begin
      real s;
      s := 0;
      for k := l step 1 until u do
         s := s + ak;
      Sum := s
   end;
```

Somme des carrés : `Sum(i, 1, n, i*i)`

Somme de la matrice `A` : `Sum(i, 1, m, Sum(j, 1, n, A[i,j]))`

Misère de la *copy rule*

```
procedure swap(a, b)
    integer a, b;
    begin
        integer temp;
        temp := a;
        a := b;
        b := temp;
    end;
```

Cette procédure n'échange pas toujours ses arguments !

Par exemple, `swap(i, A[i])` s'expanse en

`temp := i; i := A[i]; A[i] := temp`.

Les Lumières (1958) : Lisp

Le premier des langages fonctionnels :

- Structuré autour d'expressions et de fonctions récursives.
- Syntaxe minimaliste et non ambiguë (S-expressions).
- Sémantique qui se veut mathématique dès le début : liens explicites avec la théorie des fonctions récursives.

  (J. McCarthy, *Towards a Mathematical Science of Computation*, IFIP Congress 1962.)

La sémantique des fonctions se révèle plus subtile que prévu…

Portée des liaisons de variables

```
(let  ((x 1))                    ; première liaison de x
(flet ((f (y) (+ x y)))          ; f est une fonction utilisant x
(let  ((x "foo"))                ; nouvelle liaison de x
   (f 0))))                      ; appel de f
```

Que vaut `x` dans le corps de `f` lorsqu'on calcule `f 0`?

- Portée lexicale : la valeur de `x` au moment de la définition de la fonction `f`, c.à.d. `1`. C'est ce que prédit le λ-calcul.
- Portée dynamique : la valeur de `x` au moment de l'appel, c.à.d. `"foo"`. C'est le choix des premiers Lisp mais est considéré comme une erreur historique.

Point d'étape

Vers 1965, plusieurs centaines de langages de programmation existent déjà. (P. J. Landin, *The next 700 programming languages*, 1966.)

On sait déjà formaliser leur syntaxe, avec des formalismes grammaticaux comme la forme de Backus-Naur (BNF).

Le besoin de formaliser leur sémantique se fait sentir : plus les langages deviennent de haut niveau, plus leur sémantique intuitive ou précise est surprenante !

Une brève histoire des sémantiques formelles

Trois styles de sémantique formelle

Sémantique opérationnelle

Décrire formellement les étapes de l'exécution du programme.

P.ex. par réductions successives (réécriture) de termes (syntaxiques).

Exemple : la simplification des expressions arithmétiques

$$(1 + 2) \times (3 + 4) \to 3 \times (3 + 4) \to 3 \times 7 \to 21$$

Exemple : le λ-calcul et sa β-réduction

$$(\lambda x.\, M)\, N \to M\{x \leftarrow N\}$$

Trois styles de sémantique

Sémantique opérationnelle

Sémantique dénotationnelle

Associer à chaque élément syntaxique du programme un objet mathématique qui capture son sens — sa *dénotation*.

Exemples de dénotations :

Objet syntaxique	Dénotation
Expression sans variables	Sa valeur (un nombre)
Expression avec variables	Fonction valeur des variables $\mapsto$ valeur de l'expression
Commande sans boucles	Fonction valeur des variables «avant» $\mapsto$ valeur des variables «après»

Trois styles de sémantique

Sémantique opérationnelle

Sémantique dénotationnelle

Sémantique axiomatique

Décrit la sémantique d'un fragment de programme par les assertions logiques (*préconditions*, *postconditions*, *invariants*) qu'il satisfait.

Premières sémantiques opérationnelles

Peter J. Landin, *The mechanical evaluation of expressions*, The Computer Journal 6(4), 1964.

Un langage ≪applicatif≫ basé sur le λ-calcul

($\approx$ Lisp en portée statique)

Modèle d'exécution : une *machine abstraite* appelée SECD.

Peter J. Landin, *Correspondence between ALGOL 60 and Church's Lambda-notation*, Comm. ACM 8(2), 8(3), 1965.

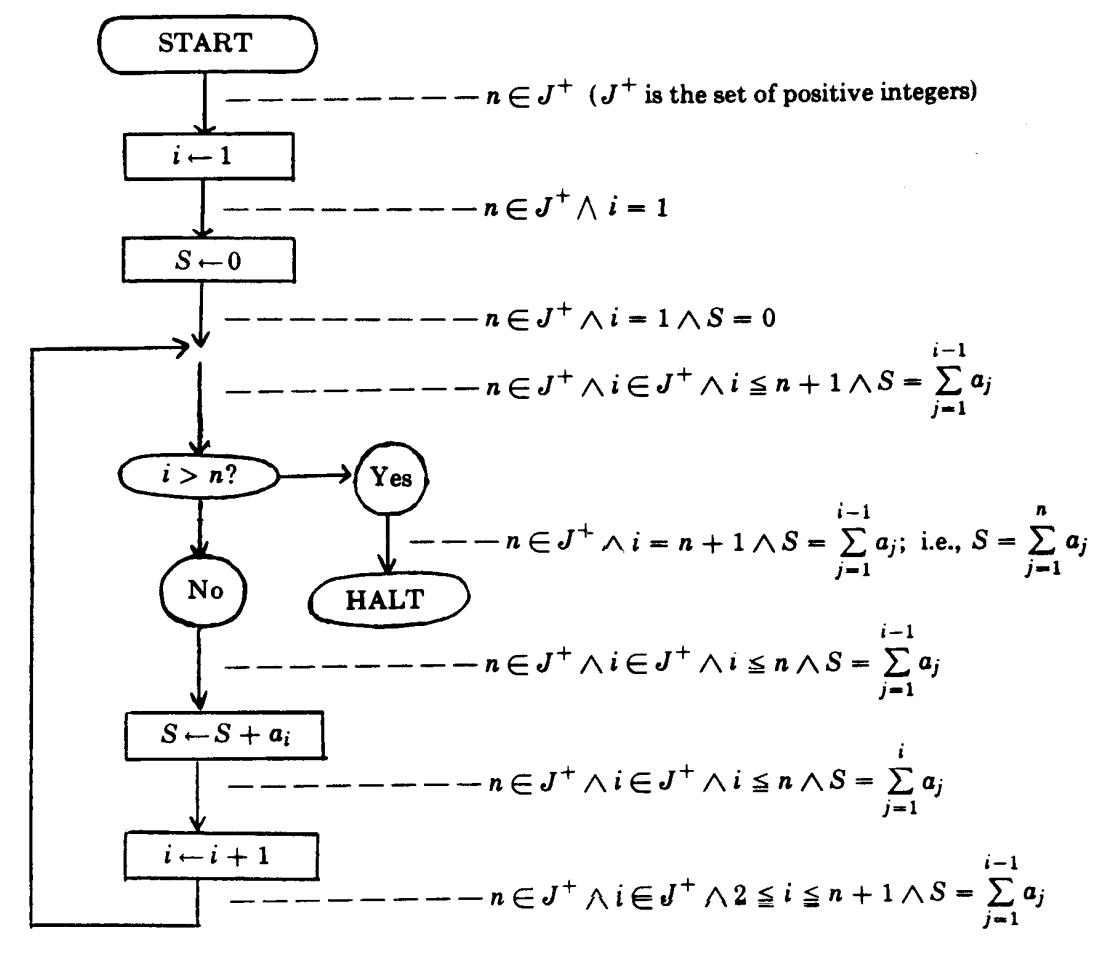
Esquisse d'une traduction d'Algol 60 vers son langage applicatif + données mutables + continuations ($\approx$ Scheme).

Ne convainc pas : trop complexe, pas assez mathématique.

Naissance de la sémantique axiomatique

Robert Floyd, *Assigning meaning to programs*, 1967

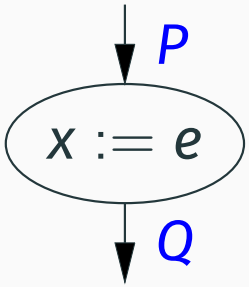
Redécouvre une idée de Turing (1949) : pour prouver un programme, il suffit d'annoter son organigramme avec des assertions logiques, et vérifier la cohérence de ces assertions.



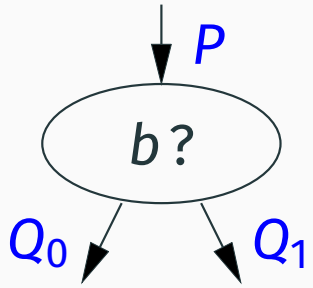
Naissance de la sémantique axiomatique

Robert Floyd, *Assigning meaning to programs*, 1967

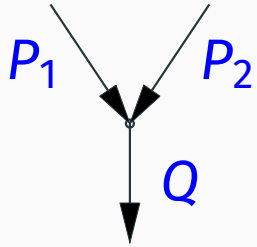
Formalise les règles logiques qui relient les préconditions P et les postconditions Q des noeuds d'un organigramme :



$$P \Rightarrow Q\{x \leftarrow e\}$$



$$P \wedge \neg b \Rightarrow Q_0$$
$$P \wedge b \Rightarrow Q_1$$



$$P_1 \vee P_2 \Rightarrow Q$$

Observe que ces règles suffisent à définir mathématiquement la sémantique de tout organigramme.

1969–1980 : l'âge d'or de la sémantique axiomatique

Comme outil pour la preuve de programmes : logique de Hoare (1969), calcul des *weakest preconditions* (Dijkstra, 1975).

Comme méthodologie de développement par raffinements successifs (Wirth, 1971), *guarded commands* (Dijkstra, 1975).

Comme guide pour la conception de langages structurés :

- commandes à une seule sortie ; pas de `break`, de `return` (Pascal)
- fonctions pures vs. procédures avec effets (preliminary Ada)

La sémantique dénotationnelle naïve

Christopher Strachey, *Towards a formal semantics*, 1964, 1966.

Ce texte et d'autres notes de Strachey introduisent le style de sémantique où des fonctions associent une dénotation à chaque construction syntaxique.

Expressions : $\mathcal{E} : \mathit{expr} \to \mathit{env} \to \mathit{val}$

$$\mathcal{E}\ x = \lambda e.\ e(x)$$

$$\mathcal{E}\ (a_1\ +\ a_2) = \lambda e.\ \mathcal{E}\ a_1\ e + \mathcal{E}\ a_2\ e$$

Commandes : $\mathcal{C} : \mathit{cmd} \to \mathit{env} \to \mathit{env}$

$$\mathcal{C}\ \texttt{skip} = \lambda e.\ e$$

$$\mathcal{C}\ (x\ :=\ a) = \lambda e.\ e\{x \leftarrow \mathcal{E}\ a\ e\}$$

$$\mathcal{C}\ (c_1;\ c_2) = \mathcal{C}\ c_2 \circ \mathcal{C}\ c_1$$

La sémantique dénotationnelle naïve

≪*The approach was deliberately informal and, as subsequent events proved, gravely lacking in rigour.*≫ (Strachey, cité par Scott)

Circularités dans les équations pour les boucles ou la récursion :

$$\mathcal{C}\,(\texttt{while}\ b\ \texttt{do}\ c) = \lambda e.\ \begin{cases} e & \text{si } \mathcal{B}\ b\ e = \texttt{false} \\ \mathcal{C}\,(\texttt{while}\ b\ \texttt{do}\ c)\,(\mathcal{C}\ c\ e) & \text{si } \mathcal{B}\ b\ e = \texttt{true} \end{cases}$$

Ensembles de dénotations mal définis :
si D est l'ensemble des dénotations de lambda-termes purs, on a envie d'interpréter $\lambda x.M$ comme une fonction $D \to D$,
mais $D \approx D \to D$ est impossible (cardinalité).

La théorie des domaines

Dana Scott, *Outline of a mathematical theory of computation*, 1970

Dana Scott, *Data types as lattices*, 1975.

Ensembles partiellement ordonnés, du moins défini ($\bot$) au plus défini, dotés d'une structure topologique (limites, continuité).

Répondent aux besoins de la sémantique dénotationnelle :

- Sémantique des boucles et de la récursion générale comme plus petits points fixes.
- Raisonnements précis sur la divergence (non terminaison).
- Constructions de domaines ≪circulaires≫ tels que $D_\infty \approx D_\infty \to_{cont} D_\infty$.

1975–1990 : l'âge d'or de la sémantique dénotationnelle

Extension de ≪l'approche Scott-Strachey≫ à presque tous les traits des langages de programmation connus.
(Notamment le contrôle non structuré via des continuations.)

Le formalisme sémantique le plus utilisé au congrès *Principles of Programming Languages* jusque 1990 environ.

Formalisation de quelques ≪vrais≫ langages de programmation, dont Ada séquentiel (V. Donzeau-Gouge, J. Storbank Petersen).

Le retour de la sémantique opérationnelle

Gordon Plotkin, *Call-by-name, call-by-value and the lambda-calculus*, 1975
Robin Milner, *A calculus of communicating systems*, 1980
Gordon Plotkin, *A structural approach to operational semantics*, 1981
Gilles Kahn, *Natural semantics*, STACS, 1987
Matthias Felleisen, Daniel Friedman, *Control operators, the SECD-machine, and the λ-calculus*, 1987

Généralisation de l'approche ≪par réductions≫ du lambda-calcul à de nombreux autres langages (Plotkin, Felleisen)

Utilisation de systèmes déductifs (règles d'inférence) pour la sémantique opérationnelle (Kahn).

Les *Labeled Transition Systems* comme première sémantique satisfaisante des calculs de processus (Milner).

1990-2010 : l'âge d'or de la sémantique opérationnelle

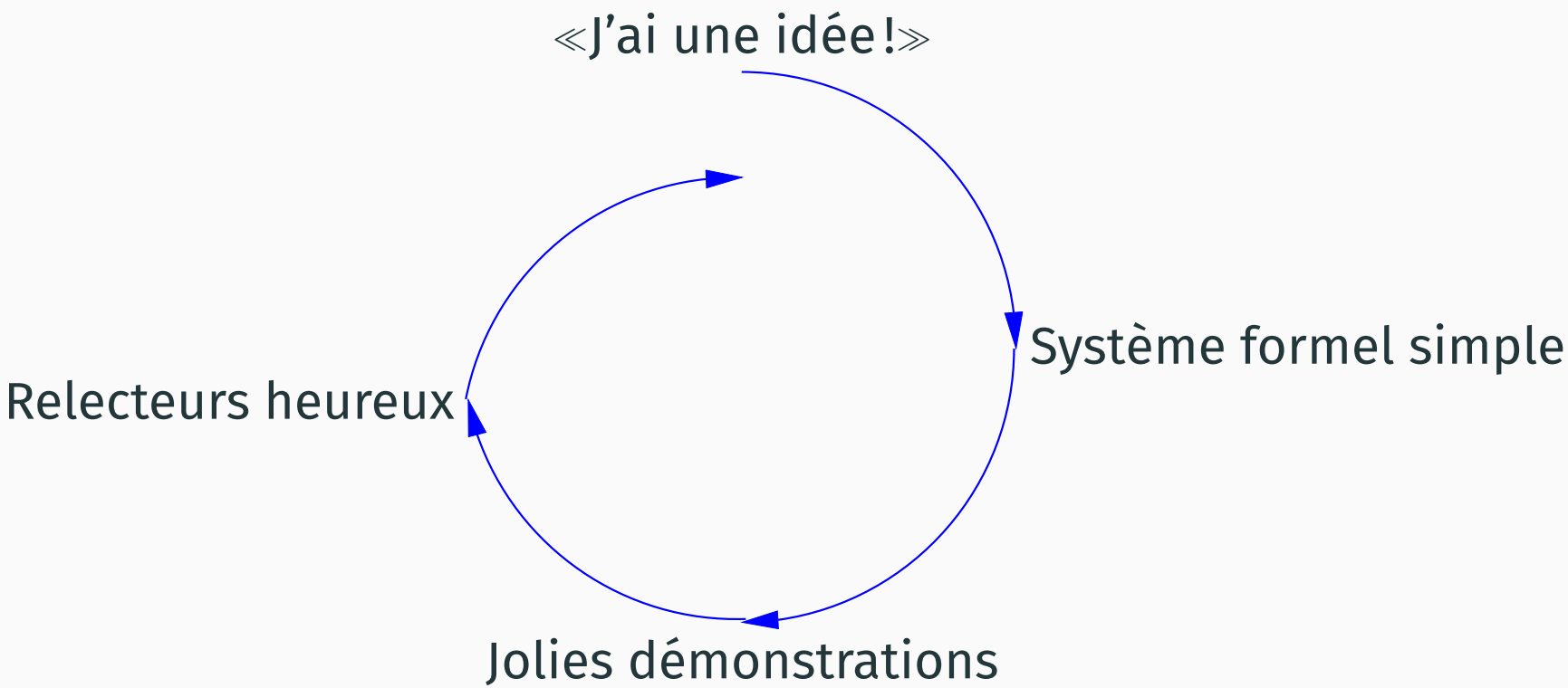
Approche très utilisée en recherche en langages de programmation, majoritaire dans les publications POPL.

Utilisation pour formaliser de ≪vrais≫ langages :

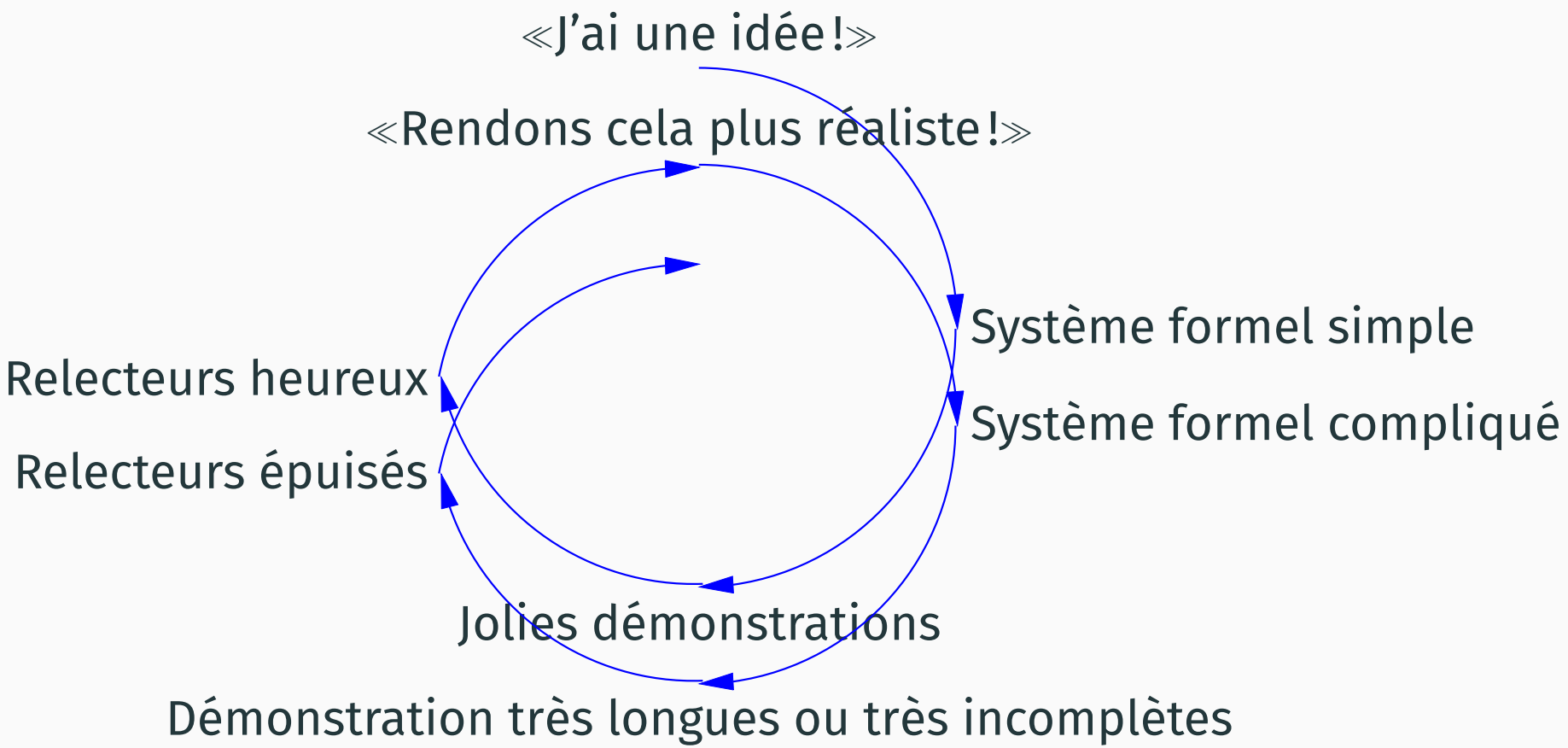
- Sur le papier : *The Definition of Standard ML* (Milner, Tofte, Harper, 1990, 1997).
- Sur machine : Java (Klein & Nipkow), C (Norrish, Leroy), Javascript (Gardner et al), etc.

Mécaniser la sémantique

Le cercle vicieux de la formalisation



Le cercle vicieux de la formalisation



Quand les démonstrations perdent leur intérêt

Proofs written by computer scientists are boring :
they read as if the author is programming the reader.

(John C. Mitchell)

The proofs of the remaining 18 cases are similar and make extensive use of the hypothesis that [...]

(auteur anonyme)

Les assistants à la démonstration

Implémentations informatiques de logiques mathématiques.

Fournissent un langage de spécification (un ≪vernaculaire mathématique≫) pour écrire les définitions et énoncer les théorèmes.

Fournissent des moyens pour construire les démonstrations, automatiquement ou en interaction avec l'utilisateur.

Vérifient que les démonstrations sont correctes et exhaustives.

Exemples : ACL2, Agda, Coq, HOL, Isabelle, Lean, PVS.

Un aperçu de Coq

Définition des nombres premiers :

```
Definition divise (n m: nat) : Prop :=
  exists k, m = k * n.

Definition premier (n: nat) : Prop :=
  n > 1 /\ forall i, divise i n -> i = 1 \/ i = n.
```

Il n'y a pas de plus grand nombre premier :

```
Theorem Euclide:
  ~ exists N, forall p, premier p -> p <= N.
Proof.
  ...
Qed.
```

Mécaniser les sémantiques avec un assistant à la démonstration

Les sémantiques pour des langages réalistes sont des systèmes formels ≪larges≫ (beaucoup de cas à considérer)
et ≪plats≫ (peu d'empilements de concepts).

Les assistants à la démonstration sont une grande aide pour

- gérer cette complexité ≪en largeur≫ ;
- trouver les erreurs élémentaires (omissions, erreurs de types) ;
- vérifier l'exhaustivité des démonstrations ;
- analyser l'impact lorsqu'on fait évoluer le langage ;
- rendre exécutables certaines définitions.

Plan du cours

Sémantiques mécanisées

Ce cours est une introduction aux sémantiques formelles des langages de programmation et à leur utilisation pour construire et valider des outils de programmation et de vérification :

- systèmes de types ;
- logiques de programmes ;
- analyses statiques ;
- compilateurs.

Présentation unifiée sur deux petits langages :
principalement IMP (impératif), un peu STLC (fonctionnel).

Toutes les définitions, propriétés, et démonstrations sont mécanisées avec l'assistant Coq.

Faut-il connaître Coq pour suivre ce cours ?

Non, pas nécessaire pour comprendre les définitions et les principaux résultats. (Souvent donnés deux fois, en notation mathématique usuelle puis en notation Coq.)

Oui, si vous souhaitez rejouer et modifier les démonstrations, et faire quelques exercices.

Supports de cours

Vidéos et transparents sur le site Web du Collège de France.

Sources Coq commentés sur Github :
`https://github.com/xavierleroy/cdf-sem-meca`

Plan du cours

28/11	Des expressions et des commandes : la sémantique d'un langage impératif
05/12	Cours reporté au 06/02
12/12	*Traduttore, traditore* : vérification formelle d'un compilateur
19/12	Compiler mieux : optimisations, analyses statiques, et leur vérification
09/01	Des logiques pour raisonner sur les programmes
16/01	Un art abstrait : l'analyse statique par interprétation abstraite
30/01	L'éternité, c'est long : divergence, théorie des domaines, approches coinductives
06/02	Des fonctions et des types : la sémantique d'un langage fonctionnel
13/02	Coq en Coq ? Mécaniser la logique d'un assistant à la démonstration

Programme du séminaire

05/12 Séminaire reporté au 13/02

12/12 *Lambda, the ultimate teaching assistant (Agda version)*
Philip Wadler (U. Edinburgh)

19/12 L'arithmétique des ordinateurs et sa formalisation
Sylvie Boldo (Inria)

09/01 Sémantique formelle de JavaScript
Alan Schmitt (Inria)

16/01 Logique de séparation en Coq : théorie et pratique
Arthur Charguéraud (Inria)

30/01 Interpréteurs abstraits mécanisés
David Pichardie (ENS Rennes)

06/02 *Understanding and evolving the Rust language.*
Derek Dreyer (MPI SWS)

13/02 What's in a name ? Représenter les variables et leurs liaisons
Xavier Leroy

Bibliographie

Bibliographie

Introduction aux sémantiques des langages de programmation :

- H. R. Nielson et F. Nielson, *Semantics with Applications : an appetizer*, Sprnger, 2007.

Pour apprendre Coq :

- En anglais : Pierce et al, *Software foundations, vol 1 : Logical foundations*, `https://softwarefoundations.cis.upenn.edu/`
- En français : Bertot et Castéran, *Le Coq'Art*, `https://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf`.
  Le cours en vidéo de Bertot :
  `http://www-sop.inria.fr/members/Yves.Bertot/videos-coq/`

Sémantiques mécanisées, premier cours

Des expressions et des commandes: la sémantique d'un langage impératif

Xavier Leroy

2019-11-28

Collège de France, chaire de sciences du logiciel

Échauffement :
les expressions arithmétiques

Les expressions arithmétiques

Un langage d'expressions comprenant

- Des constantes entières 0, 1, -5, ..., N
- Des variables x, y, z, ...
- Les opérations ≪plus≫ et ≪moins≫ : $e_1 + e_2$ et $e_1 - e_2$ où e_1 et e_2 sont elles-mêmes des expressions.

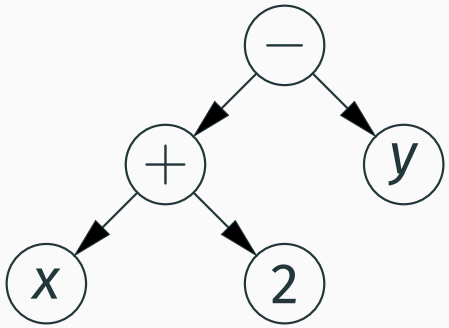
Syntaxe concrète

La notation algébrique usuelle, décrite par une grammaire BNF :

$$
\begin{aligned}
\mathit{expr} &::= \mathit{terme} \mid \mathit{expr}\ \texttt{+}\ \mathit{terme} \mid \mathit{expr}\ \texttt{-}\ \mathit{terme} \\
\mathit{terme} &::= \mathit{const} \mid \mathit{var} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \\
\mathit{const} &::= \texttt{-}? \ [\texttt{0} - \texttt{9}]+ \\
\mathit{var} &::= [\texttt{a} - \texttt{z}\ \texttt{A} - \texttt{Z}]+
\end{aligned}
$$

Note : grammaire non ambiguë : `A+B-C` est bien lu comme `(A+B)-C` et non comme `A+(B-C)`.

Syntaxe abstraite, sous forme d'arbre



$$x + 2 - y$$
$$(x + 2) - y$$
$$x + 2 - (y)$$

Aux feuilles : constantes et variables.

Aux noeuds : opérateurs $+$, $-$

Syntaxe abstraite, dans les articles de recherche

Une espèce de grammaire pour les arbres de syntaxe abstraite :

Expressions arithmétiques :

$$
\begin{array}{rll}
a ::= & x & \text{variables} \\
| & N & \text{constantes entières} \\
| & a_1 + a_2 & \text{somme de deux expressions} \\
| & a_1 - a_2 & \text{différence de deux expressions}
\end{array}
$$

Syntaxe abstraite sous forme de types inductifs

La représentation naturelle des arbres de syntaxe abstraite dans les langages fonctionnels et les assistants à la démonstration est un type inductif.

En OCaml :

```
type aexp =
  | CONST of int
  | VAR of string
  | PLUS of aexp * aexp
  | MINUS of aexp * aexp
```

En Coq :

```
Inductive aexp : Type :=
  | CONST (n: Z)
  | VAR (x: ident)
  | PLUS (a1: aexp) (a2: aexp)
  | MINUS (a1: aexp) (a2: aexp).
```

Syntaxe abstraite sous forme de types inductifs

```
Inductive aexp : Type :=
  | CONST (n: Z)
  | VAR (x: ident)
  | PLUS (a1: aexp) (a2: aexp)
  | MINUS (a1: aexp) (a2: aexp).
```

Définit 4 fonctions pour construire des valeurs de type `aexp` :

```
CONST: Z -> aexp
VAR: ident -> aexp
PLUS: aexp -> aexp -> aexp
MINUS: aexp -> aexp -> aexp
```

Syntaxe abstraite sous forme de types inductifs

```
Inductive aexp : Type :=
  | CONST (n: Z)
  | VAR (x: ident)
  | PLUS (a1: aexp) (a2: aexp)
  | MINUS (a1: aexp) (a2: aexp).
```

Toute valeur de type `aexp` est finiment engendrée par ces 4 fonctions $\Rightarrow$ analyse par cas + récursion structurelle.

```
Fixpoint F (a: aexp) :=
  match a with
  | CONST n => ...
  | VAR x => ...
  | PLUS a1 a2 => ... F a1 ... F a2 ...
  | MINUS a1 a2 => ... F a1 ... F a2 ...
  end.
```

Sémantique dénotationnelle des expressions

Une expression arithmétique dénote une fonction
valeurs des variables $\to$ valeur de l'expression.

Les valeurs des variables sont données par un état mémoire
(*store*) $\quad s$: nom de variable $\to$ valeur de la variable.

Sur papier, la sémantique dénotationnelle se présente comme une liste d'équations :

$$[\![x]\!]\, s = s(x)$$
$$[\![N]\!]\, s = N$$
$$[\![a_1 + a_2]\!]\, s = [\![a_1]\!]\, s + [\![a_2]\!]\, s$$
$$[\![a_1 - a_2]\!]\, s = [\![a_1]\!]\, s - [\![a_2]\!]\, s$$

(Note : $+$ et $-$ ont des significations différentes à gauche et à droite.)

Mécanisation de cette sémantique dénotationnelle

Sur machine, cette sémantique dénotationnelle se présente comme une fonction récursive définie par cas sur la forme de l'expression.

```
Definition store : Type := ident -> Z.

Fixpoint aeval (a: aexp) (s: store) : Z :=
  match a with
  | CONST n => n
  | VAR x => s x
  | PLUS a1 a2 => aeval a1 s + aeval a2 s
  | MINUS a1 a2 => aeval a1 s - aeval a2 s
  end.
```

Utilisations de cette sémantique dénotationnelle

Comme une calculatrice (un interpréteur pour le langage) :

Si `x` *vaut 10, alors* `2 + x - 1` *vaut 19.*

Pour simplifier des expressions :

$[\![\texttt{x} + (\texttt{10} - \texttt{1})]\!]\, s = s(x) + 9$

Pour démontrer des propriétés algébriques des expressions :

$[\![\texttt{x} + \texttt{1}]\!]\, s > [\![\texttt{x}]\!]\, s$ *pour tout* s

Pour démontrer des ≪méta≫ propriétés de la sémantique :

Si $s(x) = s'(x)$ *pour tout* x *libre dans* a*, alors*
$[\![a]\!]\, s = [\![a]\!]\, s'$.

Extensions et variantes

Étendre le langage d'expressions :

- avec des formes dérivées (p.ex. $-x \stackrel{def}{=} 0 - x$)
- avec d'autres formes primitives (p.ex. la multiplication).

Modifier la sémantique :

- Entiers «machine» au lieu d'entiers exacts $\mathbb{Z}$.
- Traiter les cas d'erreurs : débordements, division par 0, variable non définie, ...

Modulariser la sémantique dénotationnelle avec des monades

(Eugenio Moggi, *Notions of computations and monads*, 1989, 1991)

$$
\begin{aligned}
[\![N]\!] &= \mathtt{inj}(N) \\
[\![x]\!] &= \mathtt{get}(x) \\
[\![e_1 + e_2]\!] &= \mathtt{bind}\ [\![e_1]\!]\ (\lambda v_1.\ \mathtt{bind}\ [\![e_2]\!]\ (\lambda v_2.\ v_1 \oplus v_2)) \\
[\![e_1 - e_2]\!] &= \mathtt{bind}\ [\![e_1]\!]\ (\lambda v_1.\ \mathtt{bind}\ [\![e_2]\!]\ (\lambda v_2.\ v_1 \ominus v_2))
\end{aligned}
$$

Paramétrée par une monade d'environnement M et une interprétation V des valeurs entières :

$$
\begin{aligned}
\mathtt{ret} &: \forall \alpha.\ \alpha \to M\ \alpha & \qquad \mathtt{inj} &: \mathbb{Z} \to M\ V \\
\mathtt{bind} &: \forall \alpha, \beta.\ M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta & \qquad \cdot \oplus \cdot &: V \to V \to M\ V \\
\mathtt{get} &: \mathit{ident} \to M\ V & \qquad \cdot \ominus \cdot &: V \to V \to M\ V
\end{aligned}
$$

Modulariser la sémantique dénotationnelle avec des monades

Quelques choix possibles pour V :

$V = \mathbb{Z}$	arithmétique exacte
$V = [-2^{63}, 2^{63}[$	arithmétique ≪machine≫ 64 bits signée

Quelques choix possibles pour la monade M :

$M\,\alpha = (\mathit{ident} \to V) \to \alpha$	monade d'environnement
$M\,\alpha = (\mathit{ident} \to \texttt{option}\,V) \to \texttt{option}\,\alpha$	monade d'erreur et d'environnement

(Voir aussi le cours 2018-2019 ≪Peut on changer le monde ? Programmation impérative, effets monadiques, effets algébriques≫.)

Le langage IMP
et sa sémantique à réductions

Le langage IMP

Un langage minimal représentatif des langages impératifs à contrôle structuré.

Expressions arithmétiques :

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2$$

Expressions booléennes :

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

Commandes (*statements*) :

$$\begin{aligned} c ::= {} & \texttt{skip} && \text{(ne rien faire)} \\ \mid {} & x := a && \text{(affectation)} \\ \mid {} & c_1; c_2 && \text{(séquencement)} \\ \mid {} & \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 && \text{(conditionnelle)} \\ \mid {} & \texttt{while}\ b\ \texttt{do}\ c && \text{(boucle)} \end{aligned}$$

Un programme IMP

Division euclidienne, par soustractions successives.

```
// entrée: dividende dans a, diviseur dans b

r := a;
q := 0;
while b <= r do
    r := r - b;
    q := q + 1
done

// sortie: quotient dans q, reste dans r
```

Sémantiques des expressions booléennes

La routine : une sémantique dénotationnelle sous forme d'une fonction à valeurs dans `bool`.

$$\texttt{beval} : \texttt{bexp} \to \texttt{store} \to \texttt{bool}$$

De nombreuses formes dérivées :

$$a_1 \neq a_2 \qquad a_1 < a_2 \qquad a_1 \geq a_2 \qquad a_1 > a_2 \qquad a_1 \texttt{ or } a_2$$

Sémantique dénotationnelle des commandes

Tentons l'approche dénotationnelle naïve : la sémantique d'une commande est une fonction
état mémoire «avant» $\to$ état mémoire «après».

$$[\![\texttt{skip}]\!]\, s = s$$

$$[\![x := a]\!]\, s = s\{x \leftarrow [\![a]\!]\, s\}$$

$$[\![c_1; c_2]\!]\, s = [\![c_2]\!]\, ([\![c_1]\!]\, s)$$

$$[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!]\, s = \begin{cases} [\![c_1]\!]\, s & \text{si } [\![b]\!]\, s = \texttt{true} \\ [\![c_2]\!]\, s & \text{si } [\![b]\!]\, s = \texttt{false} \end{cases}$$

$$[\![\texttt{while } b \texttt{ do } c]\!]\, s = \begin{cases} s & \text{si } [\![b]\!]\, s = \texttt{false} \\ [\![\texttt{while } b \texttt{ do } c]\!]\, ([\![c]\!]\, s) & \text{si } [\![b]\!]\, s = \texttt{true} \end{cases}$$

Sémantique dénotationnelle des commandes

$$[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\ s = [\![\texttt{while}\ b\ \texttt{do}\ c]\!]\ ([\![c]\!]\ s) \quad \text{si } [\![b]\!]\ s = \texttt{true}$$

Cette équation est circulaire et ne définit pas l'état mémoire «après» l'exécution d'une boucle `while`.

Cet état est d'ailleurs non défini si la boucle ne termine pas ! (p.ex. `while true do skip`)

La fonction Coq correspondante est rejetée comme non structurellement récursive.

Sémantique dénotationnelle des commandes

Pourrions-nous changer le type de la fonction de dénotation en $\texttt{com} \to \texttt{store} \to \texttt{option store}$, de sorte que

$[\![c]\!]\, s = \texttt{Some}\ s'$ signifie que c termine avec l'état mémoire s'
$[\![c]\!]\, s = \texttt{None}$ signifie que c diverge ?

En logique classique : oui.

En théorie des types (Coq, Agda, etc) : non, car

- toutes les fonctions définissables sont calculables ;
- la fonction dénotation déciderait le problème de l'arrêt pour IMP ;
- IMP est Turing-complet.

Sémantique des commandes par réductions

Plan B : une sémantique opérationnelle par réductions successives, dans le style du lambda-calcul et sa beta-réduction.

On réduit des configurations c/s composées d'une commande c et de l'état mémoire courant s :

$$c/s \quad \to \quad c'/s'$$

c : commande — une étape — c' : commande résiduelle

s : état mémoire «avant» — de calcul — s' : état mémoire «après»

Règles de réduction

Affectations :

$$(x := a)/s \to \texttt{skip}/s\{x \leftarrow [\![a]\!]\, s\}$$

Séquences :

$$(c_1; c_2)/s \to (c_1'; c_2)/s' \quad \text{si } c_1/s \to c_1'/s'$$
$$(\texttt{skip}; c_2)/s \to c_2/s$$

Exemple :

$$(x := 1; y := 2)/s \to (\texttt{skip}; y := 2)/s' \to (y := 2)/s' \to \texttt{skip}/s''$$

avec $s' = s\{x \leftarrow 1\}$ et $s'' = s'\{y \leftarrow 2\}$.

Règles de réduction

Conditionnelles :

$$(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)/s \to c_1/s \qquad \text{si } [\![b]\!]\, s = \texttt{true}$$
$$(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)/s \to c_2/s \qquad \text{si } [\![b]\!]\, s = \texttt{false}$$

Boucles :

$$(\texttt{while } b \texttt{ do } c)/s \to \texttt{skip}/s \qquad \text{si } [\![b]\!]\, s = \texttt{false}$$
$$(\texttt{while } b \texttt{ do } c)/s \to (c; \texttt{while } b \texttt{ do } c)/s \quad \text{si } [\![s]\!]\, b = \texttt{true}$$

Sémantique à réductions sous forme de règles d'inférence

$$(x := a)/s \to \texttt{skip}/s[x \leftarrow [\![a]\!]\, s])$$

$$\frac{c_1/s \to c_1'/s'}{(c_1; c_2)/s \to (c_1'; c_2)/s'} \qquad (\texttt{skip}; c)/s \to c/s$$

$$(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)/s \to \begin{cases} c_1/s & \text{si } [\![\mathrm{b}]\!]\, \mathrm{s} = \texttt{true} \\ c_2/s & \text{si } [\![\mathrm{b}]\!]\, \mathrm{s} = \texttt{false} \end{cases}$$

$$\frac{[\![b]\!]\, s = \texttt{true}}{(\texttt{while } b \texttt{ do } c)/s \to (c; \texttt{while } b \texttt{ do } c)/s}$$

$$\frac{[\![b]\!]\, s = \texttt{false}}{(\texttt{while } b \texttt{ do } c)/s \to \texttt{skip}/s}$$

Exprimer des règles d'inférences en Coq

Étape 1 : écrire chaque règle comme une formule logique usuelle.

$$x := a/s \to \texttt{skip}/s[x \leftarrow [\![a]\!]\, s])$$

$$\frac{c_1/s \to c_1'/s'}{(c_1; c_2)/s \to (c_1'; c_2)/s'}$$

```
forall x a s,
    red (ASSIGN x a, s) (SKIP, update x (aeval s a) s)

forall c1 c s1 c2 s2,
    red (c1, s1) (c2, s2) ->
    red (SEQ c1 c, s1) (SEQ c2 c, s2)
```

Étape 2 : donner un nom à chaque règle et en faire les cas d'un prédicat inductif.

```
Inductive red: com * store -> com * store -> Prop :=
  | red_assign: forall x a s,
      red (ASSIGN x a, s) (SKIP, update x (aeval s a) s)
  | red_seq_done: forall c s,
      red (SEQ SKIP c, s) (c, s)
  | red_seq_step: forall c1 c s1 c2 s2,
      red (c1, s1) (c2, s2) ->
      red (SEQ c1 c, s1) (SEQ c2 c, s2)
  | red_ifthenelse: forall b c1 c2 s,
      red (IFTHENELSE b c1 c2, s)
          ((if beval s b then c1 else c2), s)
  | red_while_done: forall b c s,
      beval s b = false ->
      red (WHILE b c, s) (SKIP, s)
  | red_while_loop: forall b c s,
      beval s b = true ->
      red (WHILE b c, s) (SEQ c (WHILE b c), s).
```

Utiliser un prédicat inductif

Chaque cas de la définition est un théorème qui permet de conclure $\texttt{red}\ (c, s)\ (c', s')$ pour certains c, s, c', s'.

De plus, la proposition $\texttt{red}\ (c, s)\ (c', s')$ est vraie seulement si elle a été déduite en appliquant ces théorèmes un nombre fini de fois.

$\Rightarrow$ principes de raisonnement : par récurrence sur la dérivation et par cas sur la dernière règle utilisée.

(Pour mieux comprendre les bases de l'approche, voir le cours 2018-2019 ≪Des armes de construction massive : types inductifs et prédicats inductifs≫.)

Suites de réductions

Le comportement d'une commande c s'obtient en formant des suites de réductions commençant par c/s.

- Terminaison avec l'état final s' : suite finie de réductions vers $\texttt{skip}/s'$.

$$c/s \to c_1/s_1 \to \cdots \to \texttt{skip}/s'$$

- Divergence : suite infinie de réductions

$$c/s \to c_1/s_1 \to \cdots \to c_n/s_n \to \cdots$$

- Erreur à l'exécution : suite finie de réductions vers un état irréductible autre que $\texttt{skip}$ (ne se produit jamais en IMP)

$$c/s \to c_1/s_1 \to \cdots \to c'/s' \not\to \qquad c' \neq \texttt{skip}$$

Autres formes de sémantique opérationnelle : sémantique naturelle, interpréteurs de référence

La sémantique naturelle

Un autre style de sémantique opérationnelle, intermédiaire entre sémantique à réductions et fonction d'évaluation.

Aussi appelé *big-step semantics*, les *small-step semantics* étant les sémantiques à réductions.

Intuition de la sémantique naturelle

Si la commande $c; c'$ termine, sa suite de réductions a une forme très particulière :

$$(c; c')/s \to (c_1; c')/s_1 \to \cdots \to (\texttt{skip}; c')/s_2 \to c'/s_2 \to \cdots \to \texttt{skip}/s_3$$

Cette suite montre que la commande c termine depuis s sur un état intermédiaire s_2, et que la commande c' termine depuis s_2 sur s_3

$$c/s \to c_1/s_1 \to \cdots \to \texttt{skip}/s_2$$
$$c'/s_2 \to \cdots \to \texttt{skip}/s_3$$

Intuition de la sémantique naturelle

Idée : définir un prédicat $c/s \Downarrow s'$ qui veut dire
≪dans l'état initial s, la commande c termine avec l'état final s'≫,
sous forme de règles d'inférence
qui capturent cette structure des exécutions qui terminent.

Exemple : nous avons vu que $(c; c')$ démarré dans s termine sur s' ssi c démarré dans s termine sur s_2 et c' démarré dans s_2 termine sur s', pour un certain état intermédiaire s_2. D'où la règle

$$\frac{c_1/s \Downarrow s_2 \qquad c_2/s_2 \Downarrow s'}{c_1; c_2/s \Downarrow s'}$$

Les règles de la sémantique naturelle d'IMP

$$\texttt{skip}/s \Downarrow s \qquad\qquad x := a/s \Downarrow s[x \leftarrow [\![a]\!]\, s]$$

$$\frac{c_1/s \Downarrow s' \qquad c_2/s' \Downarrow s''}{c_1; c_2/s \Downarrow s''} \qquad\qquad \frac{\begin{array}{c} c_1/s \Downarrow s' \text{ if } [\![b]\!]\, s = \texttt{true} \\ c_2/s \Downarrow s' \text{ if } [\![b]\!]\, s = \texttt{false} \end{array}}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2/s \Downarrow s'}$$

$$\frac{[\![b]\!]\, s = \texttt{false}}{\texttt{while } b \texttt{ do } c/s \Downarrow s}$$

$$\frac{[\![b]\!]\, s = \texttt{true} \qquad c/s \Downarrow s' \qquad \texttt{while } b \texttt{ do } c/s' \Downarrow s''}{\texttt{while } b \texttt{ do } c/s \Downarrow s''}$$

Équivalence avec la sémantique à réductions

Un résultat bien connu :

$$\mathtt{c}/\mathtt{s} \Downarrow \mathtt{s}' \quad \text{si et seulement si} \quad \mathtt{c}/\mathtt{s} \xrightarrow{*} \mathtt{skip}/\mathtt{s}'$$

On peut donc utiliser l'une ou l'autre sémantique pour raisonner sur les exécutions qui terminent.

La sémantique naturelle fournit un principe de récurrence (sur les dérivations de $c/s \Downarrow s'$) plus fort et très utile pour les preuves de compilateurs (3e cours) et de logiques de programmes (5e cours).

Un interpréteur de référence

Il s'est révélé impossible de définir la sémantique d'une commande par une fonction
état mémoire ≪avant≫ $\longrightarrow$ état mémoire ≪après≫
puisque cette fonction serait partielle (non-terminaison).

Nous pouvons cependant définir une approximation de cette fonction en bornant *a priori* la profondeur de récursion avec un paramètre `fuel` de type `nat`.

```
Fixpoint cexec_f (fuel: nat) (s: store) (c: com)
                                          : option store :=
  match fuel with
  | O => None
  | S fuel' => ... cexec_f fuel' s' c' ...
  end.
```

Un interpréteur de référence

```
Fixpoint cexec_f (fuel: nat) (s: store) (c: com)
                                        : option store :=
  ...
```

Un résultat `Some s'` signifie que `c` termine sur `s'` à coup sûr.

Un résultat `None` est non concluant : ou bien `c` diverge, ou bien il faut plus de fuel pour terminer le calcul de `c`.

Très utile pour tester la sémantique sur des exemples de programmes.

Point d’étape

Point d'étape

Le langage IMP = expressions + commandes impératives.

Sémantiques : dénotationnelle naïve, opérationnelles
(à réductions, ou naturelle, ou par interpréteur borné).

Formalisation Coq : types inductifs, fonctions récursives,
prédicats inductifs.

Premières démonstrations : équivalences entre sémantiques.