

# Interprétation et analyse de programmes

Didier Buchs

Université de Genève

23 avril 2018

1/38



## Vers un interpréteur

- Il s'agit de produire un programme prenant en entrée le langage à interpréter et produisant l'état du programme après l'exécution.
- Prolog est un bon langage pour réaliser cette tâche, néanmoins un certain nombre de définitions dans la sémantique ne sont pas assez opérationnelles ni suffisamment concrètes. ?
- L'implémentation des règles sert à l'animation et même éventuellement à l'évaluation de propriétés du langage. / programme
- Dans la suite, Prolog sera employé mais les mêmes principes peuvent être utilisés pour d'autres langages au prix de certaines difficultés par rapport à la gestion des termes.

Règles

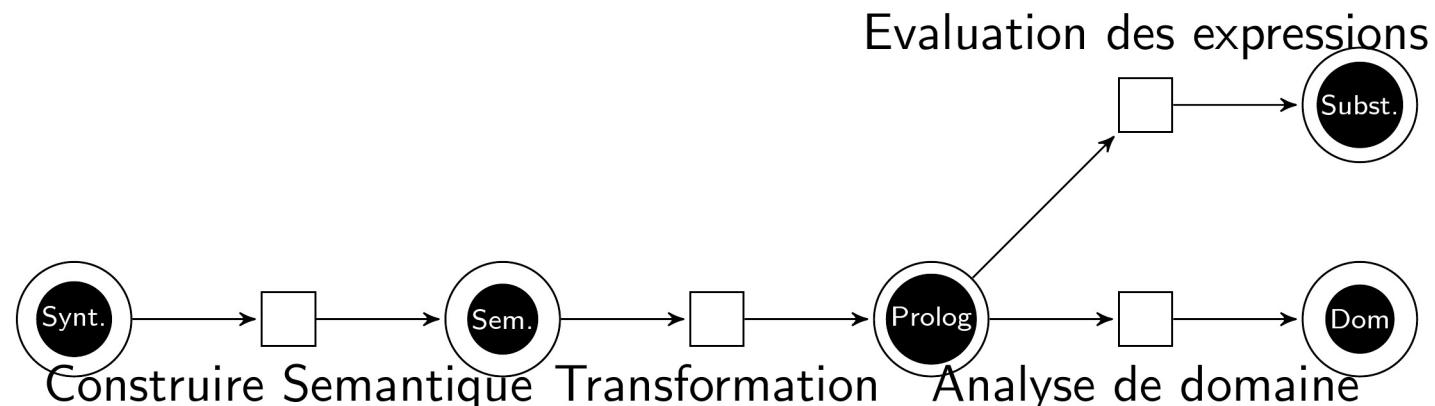
$P_1 \wedge P_2 \wedge P_3$   
cas

$\Leftarrow$

Conc ! -  $P_1, P_2, P_3, \dots$

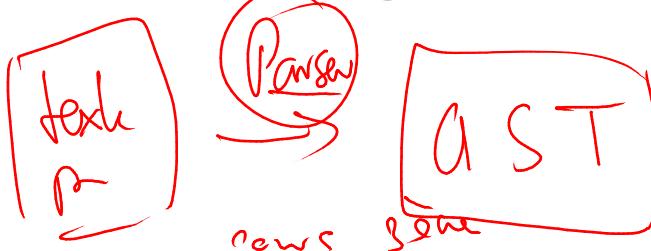
Prolog

# Processus pour l'obtention d'un interpréteur



## Principes à appliquer :

- Etablir la structure syntaxique du langage en prolog
- Définir la notion d'état d'un programme
- Définir quelques procédures de manipulation (substitutions, listes, ...)
- Traduire en prolog les règles sémantiques sans en changer le sens
  - Filtrer la règle applicable de manière unique
  - Evaluer les préconditions
  - Etablir l'effet de la règle
  - Equations algébriques à résoudre
- Pièges : non-déterminisme, problèmes de négation.



abstract syntax tree

# Syntaxe en Prolog

- Deux solutions :
  - Utiliser les grammaires DCG pré-implémentée en Prolog
  - Utiliser pragmatiquement, termes, opérateurs et foncteurs pour encoder les unités syntaxiques
- Grammaires DCG (Direct Clause Grammar) présenté dans le cours de compilation.  $g \dashrightarrow a, b, \dots, d$ .
- Opérateurs en Prolog

```
:op(50, xfx, -->).  
:-op(500, yfx, +).  
:-op(900, fy, not).  
:-op(700, xfx, =).
```

# Syntaxe du Langage minimal et Prolog

- Opérateur  $_ + _$  :  $1 + y * 3 - 4 * 7 + x$
- Opérateur  $_ < _$  :  $sq(y) < x$
- Expressions :
  - Variable atome prolog =  $atom(x)$
  - Termes, foncteurs pour appel de fonction
- Instructions :
  - $_ := _$  :  $y := y + 1$
  - $if\_then\_else(_, _, _)$
  - $while(_, _)$
- Programmes :
  - Séquence  $_ ; _$
  - Fonctions :  $func(_, _)$  :  
 $func(sqrt(x), y := 1; while(sq(y) < x, y := y + 1); return y)$

## Exemple

- ?- eval([func(sqrt(x),y:=1;while(y\*y<x,y:=y+1); return x:=sqrt(65536),S].
- Contexte des fonctions :

func(sqrt(x),y:=1;while(y\*y<x,y:=y+1); return y)

pour simplifier dans l'implémentation qui suit, la dernière instruction est un 'return' !!

func(sqrt(x),y:=1;while(y\*y<x,y:=y+1), y)

- Le corps du programme :

x:=sqrt(65536)

- S, substitution = ensemble de couples
- Résultat :  $S = [(x, 256)];$

# Comment construire l'interpréteur : fonctions d'évaluations majeures



- $\text{bigstep}_e$  : calcule l'expression entière ( $e$ ) pour une substitution donnée ( $S$ ) et un ensemble de fonctions ( $\text{PROG}$ ) :  $\text{env}(\text{PROG}, S), e -> N$
- $\text{bigstep}_c$  : calcule les conditions ( $e$ ) pour une substitution donnée ( $S$ ) et un ensemble de fonctions ( $\text{PROG}$ ) :  $\text{env}(\text{PROG}, S), e -> N$
- $\text{bigstep}_i$  : calcule ( $S'$ ) l'effet d'une instruction sur une substitution ( $S$ ) connaissant un ensemble de fonctions ( $\text{PROG}$ ) :  $\text{PROG}, i, S -> S'$
- $\text{bigstep}_b$  : calcule ( $S'$ ) l'effet d'une séquence d'instructions ( $p$ ) :  $\text{PROG}, p, S -> S'$
- $\text{eval}(ENV, P, S) : - \text{bigstep}_b(ENV, P, [], S).$

# Substitutions

Trouver la valeur définie pour une variable

```
/* subsextract([],V,0) :- fail */
subsextract([(V,N)|L],V,N).
subsextract([(V,NN)|L],VV,N) :-
    V \== VV, subsextract(L,VV,N).
```

ajouter une valeur à une variable dans la substitution (propriétés telles que unicité et non-ordre prise en compte)

```
subsadd([],V,N,[ (V,N) ]).
subsadd([(V,NN)|L],V,N,[ (V,N) | L ]).
subsadd([(V,NN)|L],VV,N,[ (V,NN) | LL ]):- V \== VV,
    subsadd(L,VV,N,LL).
```

## Expressions et relations

Se basent sur l'implémentation de prolog des entiers :

```
bigstep_c(ENV,E < EE) :-  
    bigstep_e(ENV,E,N), bigstep_e(ENV,EE,NN), N < NN.  
bigstep_c(ENV,E > EE) :-  
    bigstep_e(ENV,E,N), bigstep_e(ENV,EE,NN), N > NN.  
bigstep_c(ENV,E = EE) :-  
    bigstep_e(ENV,E,N), bigstep_e(ENV,EE,NN), N = NN.  
bigstep_c(ENV,E \== EE) :-  
    bigstep_e(ENV,E,N), bigstep_e(ENV,EE,NN), N \== NN.
```

## Expressions : opérations

```
bigstep_e(ENV,E + EE,NS) :-  
    bigstep_e(ENV,E,N),bigstep_e(ENV,EE,NN),NS is N + NN.  
bigstep_e(ENV,E * EE,NS) :-  
    bigstep_e(ENV,E,N),bigstep_e(ENV,EE,NN),NS is N * NN.  
bigstep_e(ENV,E / EE,NS) :-  
    bigstep_e(ENV,E,N),bigstep_e(ENV,EE,NN),NS is N / NN.  
bigstep_e(ENV,E - EE,NS) :-  
    bigstep_e(ENV,E,N),bigstep_e(ENV,EE,NN),NS is N - NN.  
  
bigstep_e(ENV,N,N) :- integer(N).
```

## Traduction

Definition (Sémantique d'évaluation )

$e \in ExpVar = T_{\{+, -, *, /\}}(\mathbb{N} \cup V)$  et  $n \in \mathbb{N}$ ,  $s \in Subs$   
 $+_{\mathbb{N}}, *_{\mathbb{N}}, -_{\mathbb{N}}, /_{\mathbb{N}}$  sont les fonctions sur  $\mathbb{N}$

$$R \text{ Constante : } \frac{}{S \vdash n \Rightarrow n}$$
$$R_- : \frac{S \vdash e \Rightarrow n, S \vdash e' \Rightarrow n'}{S \vdash e - e' \Rightarrow n -_{\mathbb{N}} n'}$$

Devient :  
 $bigstep\_e(ENV, V, N) :- atom(V), subexchange(ENV, V, N).$

$bigstep\_e(ENV, N, N) :- integer(N).$

$bigstep\_e(ENV, E - EE, NS) :-$   
 $bigstep\_e(ENV, E, N), bigstep\_e(ENV, EE, NN), NS \text{ is } N - NN.$

## Expressions : variables

```
bigstep_e(env(PROG,S),V,N) :- atom(V), subsextract(S,V,N).
```

Les variables sont des atomes !

Contexte d'évaluation : env(def fonctions, substitution)

```
env([], [(x, 20), (y, 2)])
```

Exemple :

```
?- bigstep_e(env([], [(x, 20), (y, 2)]), (1+y*3-4*7+x), N).
```

N = -1

## Expressions : appel de fonctions

Les fonctions sans paramètres ne sont pas autorisées (sinon confusion avec les variables !!) Appel de fonctions par 'functors'

*substitution de l'appel*

```
bigstep_e(env(PROG,S),FCTCALL,N) :-  
    FCTCALL =.. [NAME|PARAM], length(PARAM,M), M>0,  
    findfunction(PROG,NAME,  
        func(NAMEPARAMFORMAL,FBODY,ERETURN)),  
    NAMEPARAMFORMAL =.. [NAME|PARAMFORMAL],  
    bigstep_le(env(PROG,S),PARAM,LN), substitution du  
    bindparam(LN,PARAMFORMAL,SS), le programme  
    bigstep_p(PROG,FBODY,SS,SSS),  
    bigstep_e(env(PROG,SSS),ERETURN,N).
```

$$S \cap SS = \emptyset$$

## Expressions : appel de fonctions Variante 1

Variables globales qui sont supposées sans collision de nom,  
attention au paramètre de retour de substitution (effet de bord)

```
bigstep_e(env(PROG,S),FCTCALL,N,S5) :-  
    FCTCALL =.. [NAME|PARAM], length(PARAM,M), M>0,  
    findfunction(PROG,NAME,  
                 func(NAMEPARAMFORMAL,FBODY,ERETURN))  
    NAMEPARAMFORMAL =.. [NAME|PARAMFORMAL],  
    bigstep_le(env(PROG,S),PARAM,LN),  
    bindparam(LN,PARAMFORMAL,SS),  
    merge(S,SS,S4),  
    bigstep_p(PROG,FBODY,S4,SSS),  
    bigstep_e(env(PROG,SSS),ERETURN,N),  
    extract(SSS,S5).
```

forise  
en  
couple  
des  
variables globales

(pas dans la 'syntaxe de'  
la langue)

15/38

## Expressions : appel de fonctions

```
?- bigstep_b([func(sqrt(X), Y := 1;  
              while(Y * Y < X, Y := Y + 1), Y)],  
             X := sqrt(65536), [], S).  
  
S = [ (X, 256)] ;
```

## Instructions : if then else

```
bigstep_i(PROG,if_then_else(COND,P,PP),S,SS) :-  
    bigstep_c(env(PROG,S),COND),  
    bigstep_p(PROG,P,S,SS).
```

```
bigstep_i(PROG,if_then_else(COND,P,PP),S,SS) :-  
    not(bigstep_c(env(PROG,S),COND)),  
    bigstep_p(PROG,PP,S,SS).
```

## Instructions : while

```
bigstep_i(PROG,while(COND,P),S,SSS):-  
    bigstep_c(env(PROG,S),COND),  
    bigstep_p(PROG,P,S,SS),  
    bigstep_p(PROG,while(COND,P),SS,SSS).  
  
bigstep_i(PROG,while(COND,P),S,S):-  
    not(bigstep_c(env(PROG,S),COND)).
```

## Instructions : affectation

```
bigstep_i(PROG, (V:=E), S, SS) :-  
    bigstep_e(env(PROG, S), E, N), subsadd(S, V, N, SS).
```

rappel de la règle originale :

Definition (Sémantique d'évaluation : Règle affectation )

$e \in ExprVar_V$  et  $v \in V$ ,  $S, S', S'' \in Subs$

$$Raffection : \frac{S \vdash e \implies n}{S \vdash v := e \implies, S/[v = n]}$$

## Blocs

La sémantique d'un programme comme une liste d'instructions

```
bigstep_p(PROG,(I;P),S,SSS) :- !,  
    bigstep_i(PROG,I,S,SS),  
    bigstep_p(PROG,P,SS,SSS).
```

cut, pour éviter  
d'exécuter l'insL

cut pour sélectionner seulement une instruction.

```
bigstep_p(PROG,I,S,SS) :- bigstep_i(PROG,I,S,SS).
```

## Exercice

sachant qu'un langage de manipulation binaire à la syntaxe abstraite suivante :

$exp = exp + exp$	union bit par bit
$exp = bin$	nombre binaires
$exp => exp$	shift right
$exp = < exp$	shift left
$bin = digit bin   digit$	ex : 1010101
$digit = 0   1$	

En utilisant les règles produites pour ce langage, donner un programme Prolog 'interprétant' ce langage.

# Exercice : (syntaxe)

## Exercice : (sémantique)

23/38



# Analyse de programmes

- La sémantique de notre langage impératif peut être interprétée pour analyser un programme et en étudier le comportement
- Types d'analyse possibles :
  - Trouver les chemins d'exécution possibles
  - Déetecter des anomalies telles que variables non initialisées, code mort, ....
- Comment procéder ?
  - Utiliser l'interprète Prolog généré à partir de la sémantique :
    - Exécution directe
    - Exécution symbolique
    - Analyse statique
  - Execution symbolique en Prolog
    - Posséder une axiomatisation des types de données (Peano pour les entiers)
    - Avoir un mécanisme d'évaluation plus 'intelligent' que la résolution SLD

$$0 \leq y + x \leq 3$$

# Evaluation des expressions

## Definition

$e \in ExprVar_V$  et  $v \in V$ ,  $Var(e) = \{v\}$ ,  $S \in Subs$

$$Dom_e(v) = \{< k, n > \in Num \times Num | S/[v = n] \vdash e \implies k\}$$

Exemple :  $e = \text{sqr}(v)$

$$Dom_e(v) = \{< 1, 1 >, < 1, 2 >, \dots, < 2, 4 >, \dots, \}$$

Comment faire mieux ?

$\nexists Dom_e(v) = \infty$

# Evaluation des expressions

Avec l'usage de la résolution les solutions seront 'généralisées' par l'usage de variables. (identique à la lazy evaluation)

## Definition

$e \in ExprVar_V$  et  $v \in V$ ,  $Var(e) = \{v\}$ ,  $S \in Subs$

$$SDom_e(v) = \{ < k, e' > \in Num \times ExprVarV \mid S/[v = e'] \vdash e \implies k \}$$

Exemple :  $e = \text{pair}(v)$

$$Dom_e(v) = \{ < \text{false}, 1 >, < \text{true}, 2 >, \cancel{< \text{true}, \text{succ}(\text{succ}(0)) >} \\ , \dots, \cancel{< \text{true}, \text{succ}(\text{succ}(0)) * x >} , \dots, \}$$

Ce qui serait en intension :

$$\underline{\underline{Dom}}_{pair(x)}(v) = \{ \langle \underline{\underline{false}}, \underline{\underline{2 * x + 1}} \rangle, \langle \underline{\underline{true}}, \underline{\underline{2 * x}} \rangle \mid x \in Num \}$$

## Marquage des choix

Il s'agit de 'monitorer' les choix des règles pour pouvoir décider de critères tels que la couverture ...

- Couverture des branches
- Couverture des conditions ...

Pour cela il faut complètement axiomatiser la sémantique et améliorer les preuves :

- Pas de structures non connues ( $apply_{\mathbb{N}}(+)$ )
- Attention à la résolution SLD, en changer pour atteindre la complétude.

## Pourquoi changer la représentation des types de données

- Solution utilisant les entiers de Prolog

- $N \text{ is } M_1 + M_2$
- Si  $M_1$  et  $M_2$  sont définis alors  $N$  est défini
- Si  $N$  défini impossible de trouver  $M_1$  et  $M_2$  car 'is' est une fonction stricte.

- Solution correcte, rendre inversible les opérations

- Représentation de tous entiers par les générateurs :
- 0 et s (successeur)

Exemple : le nombre 3 est : s s s 0

$$x + y$$

$$x + 0 = x$$

$$x + s(z) = s(x+z)$$

$$\begin{aligned} x * 0 &= 0 \\ x * s(z) &= x * (x * z) \end{aligned}$$

$$\begin{aligned} s(x) > 0 &= \text{true} \\ 0 > s(x) &= \text{false} \\ s(x) > s(y) &= x > y \end{aligned}$$

$$\begin{aligned} 0 > 0 &= \text{false} \\ \} &\text{ Peano} \end{aligned}$$

$$x - 0 = x$$

$$0 - s(y) = 0$$

$$s(x) - s(y) = x - y$$

$$x > y$$

$$x > s(x) = \text{false}$$

## Analyse des chemins (if then else)

Exemple (X est une variable prolog définissant un domaine, x est une variable du langage) : *variable Prolog*

```
?- bigstep_b([],x:=s(X);if_then_else(x< s(s(s(0))),  
x:=x+s(0),x:=x-s(0)),[],R).
```

X = 0

R = [ (x, s(s(0)))] ;

X = s(0)

R = [ (x, s(s(s(0))))] ;

X = s(s(0))

R = [ (x, s(s(0)))] ; *Variable*

X = s(s(s(\_G320)))

R = [ (x, s(s(s(s(\_G320)))))] ;

No

## Axiomatisation des naturels (Peano)

Générateurs

```
0 : -> nat;  
S _ : nat -> nat
```

Typage par adjonction de l'arité à chaque termes

Axiomes

$$\begin{aligned}0 + N &= N \\ S(N) + N' &= s(N+N')\end{aligned}$$

# Opérations en Prolog

Le typage est attaché à chaque termes.

```
peanoadd(0:(nat),NN,NN).  
peanoadd(s(N):(nat->nat),NN,s(NNN):(nat->nat)):-  
    peanoadd(N,NN,NNN).  
  
peanomul(0:(nat),NN,0:(nat)).  
peanomul(s(N):(nat->nat),NN,NNNN):-  
    peanomul(N,NN,NNN),peanoadd(NN,NNN,NNNN).
```

## Opérations en Prolog (2)

```
peanodiv(N,NN,s(NS)):(nat->nat)):-  
    peanoGT(N,NN,t),peanosub(N,NN,NNN),peanodiv(NNN,NN,NS)  
peanodiv(N,NN,0:(nat)):-peanoLT(N,NN,t).  
peanodiv(N,NN,s(0:(nat))):-peanoEQ(N,NN,t).  
  
peanosub(0:(nat),0:(nat),0:(nat)).  
peanosub(s(N):(nat->nat),0:(nat),s(N):(nat->nat)).  
peanosub(s(N):(nat->nat),s(NN):(nat->nat),NNN):-peanosub(N,
```

## relations

```
peanoLT(0:(nat),s(NN):(nat->nat),t:bool).  
peanoLT(0:(nat),0:(nat),f:bool).  
peanoLT(s(NN):(nat->nat),0:(nat),f:bool).  
peanoLT(s(NN):(nat->nat),s(N):(nat->nat),R):-peanoLT(NN,N,I  
  
peanoGT(N,NN,R):-peanoLT(NN,N,R).  
  
peanoEQ(0:(nat),s(NN):(nat->nat),f:bool).  
peanoEQ(0:(nat),0:(nat),t:bool).  
peanoEQ(s(NN):(nat->nat),0:(nat),f:bool).  
peanoEQ(s(NN):(nat->nat),s(N):(nat->nat),R):-peanoLT(NN,N,I  
  
peanoNEQ(N,NN,f:bool):-peanoEQ(N,NN,t:bool).  
peanoNEQ(N,NN,t:bool):-peanoEQ(N,NN,f:bool).
```

## Typage, synthèse des valeurs

```
isnat(N,N):-var(N),!.
```

```
isnat(0,0:nat).
```

```
isnat(s(N),s(NN):(nat->nat)):-isnat(N,NN).
```

```
isnat(NN):-isnat(N,NN).
```

} *générateur*

Ce prédictat produit les valeurs typées depuis des valeurs non typées connues pour un type connu.

Il peut servir de générateur de valeurs.

## Variables libres (couverture des branches)

```
?- eval(x:=s(Y);b:=B;if_then_else(b,z:=s(0)+x,z:=s(0)-x)).
```

```
x:nat = s(_G157)
b:bool = t
z:nat = s(s(_G157))
```

```
Y = _G157
B = _G163 ;
```

```
x:nat = s(0)
b:bool = f
z:nat = 0
```

```
Y = 0:nat
B = _G163 ;
```

35/38

## Couverture minimale ? (strategie et axiomatisation)

Réécriture équivalente des expressions implique un changement de comportement!!!

```
?- eval(x:=s(Y);b:=B;if_then_else(b,z:=x+s(0),z:=x-s(0))).
```

```
x:nat= s(0)
```

```
b:bool= t
```

```
z:nat= s(s(0))
```

```
Y = 0:nat
```

```
B = _G163 ;
```

```
x:nat = s(s(0))
```

```
b:bool = t
```

```
z:nat = s(s(s(0)))
```

```
x:nat = s(s(s(0)))
```

```
b:bool = t
```

36/38

# Exploration de domaines

```
?-loadfun([sqrt,sq],C),bigstep_p(C,x:=sqrt(s(s(s(0)))),[],R),
          subsprint(R).
x:nat' = 's(s(0))

?- loadfun([sqrt,sq],C),bigstep_p(C,x:=sqrt(s(s(s(X)))),[],R),
          subsprint(R).
x:nat' = 's(s(0))
X = 0:nat

x:nat' = 's(s(0))
X = s(0:nat): (nat->nat)

x:nat' = 's(s(s(0)))
X = s(s(0:nat): (nat->nat)): (nat->nat)

x:nat' = 's(s(s(0)))
X = s(s(s(0:nat): (nat->nat)): (nat->nat)):
```

37/38



## Intérêts de l'exploration de domaines

- Vérification par test

- Création de données de test (domaines d'entrées et hypothèses de réduction)
- évaluation de la couverture avec instrumentation (marquage des choix) (*chemin d'exécution, couverture des tests...*)

- Analyse de programmes

- détection de comportements aberrants (ajout des exceptions dans la sémantique, par complémentation des comportements corrects)
- détection de violation d'invariants
- accès interdits à la mémoire (sémantique des pointeurs et de la mémoire)

memory alias  
↓  
data race  
conflict

En règle générale, usage de méthodes d'approximations avec faux positifs (preuves) (ou négatifs ! tests)

- dangling reference  
- memory leak

$$f(x, y, z) \rightarrow \mathbb{R}$$

$x, y, z \in \mathbb{N}^3$

$$\text{Dom}(x, y, z) \subseteq \mathbb{N}^3$$

# Tests

- tests Ponchonnel
    - Entrée ↪ Entrée
    - $f(\text{ent}) \rightarrow \boxed{QR}$
  - test de robustesse
    - (cas non prévu)
    - ex : attaque SQL
  - test de régression
    - logiciel  $\frac{A}{T_G} \rightsquigarrow A'$   $\frac{G_I}{T_I}$
- hypothèses  
Rédaction  
? Pile  
- un permis  
par rapport aux  
valeurs mises dans  
la pile  
- regularité  $\leq n$
- Oracle
- ↓
- Test