

Interprétation et analyse de programmes

Didier Buchs

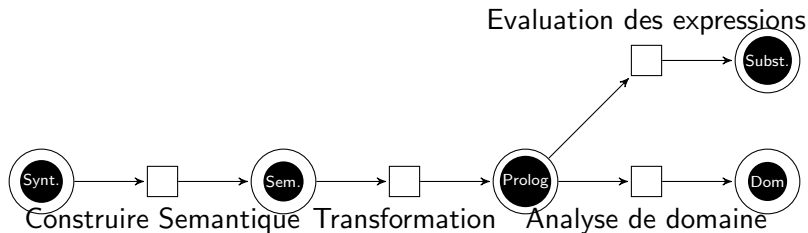
Université de Genève

23 février 2018

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.
- Dans la suite, Prolog sera employé mais les mêmes principes peuvent être utilisé pour d'autres langages au prix de certaines difficultés par rapport à la gestion des termes.

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.

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. $g \text{ -- } > a, b, ., d$.
- 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

- *bigstep_e* : calcule l'expression entière (e) pour une substitution donnée (S) et un ensemble de fonctions ($PROG$) : $env(PROG, S), e \rightarrow N$
- *bigstep_c* : calcule les conditions (e) pour une substitution donnée (S) et un ensemble de fonctions ($PROG$) : $env(PROG, S), e \rightarrow N$
- *bigstep_i* : calcule (S') l'effet d'une instruction sur une substitution (S) connaissant un ensemble de fonctions ($PROG$) : $PROG, i, S \rightarrow S$
- *bigstep_b* : calcule (S') l'effet d'une séquence d'instructions (p) : $PROG, p, S \rightarrow S$
- $eval(ENV, P, S) : -bigstep_b(ENV, P, [], S)$.

Substitutions

Trouver la valeur définie pour une variable

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

ajouter une valeur a 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).
```

Definition (Sémantique d'évaluation)

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

$$\begin{array}{l} \text{R Constante : } \frac{}{S \vdash n \Longrightarrow n} \\ R- : \frac{S \vdash e \Longrightarrow n, S \vdash e' \Longrightarrow n'}{S \vdash e - e' \Longrightarrow n -_{\mathbb{N}} n'} \end{array}$$

Devient :

```
bigstep_e(ENV,N,N):-integer(N).
```

```
bigstep_e(ENV,E - EE,NS):-  
    bigstep_e(ENV,E,N),bigstep_e(ENV,EE,NN),NS is N - NN.
```

Expressions : variables

```
bigstep_e(env(PROG,S),V,N):-atom(V),subextract(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'

```
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) ,  
    bindparam(LN,PARAMFORMAL,SS) ,  
    bigstep_p(PROG,FBODY,SS,SSS) ,  
    bigstep_e(env(PROG,SSS) ,ERETURN,N) .
```

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).
```

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)

$e \in \text{ExprVar}_V$ et $v \in V$, $S, S', S'' \in \text{Subs}$

$$\text{Raffectedation : } \frac{S \vdash e \Longrightarrow n}{S \vdash v := e \Longrightarrow_I S/[v = n]}$$

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 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 :

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

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

Exercice : (syntaxe)

Exercice : (sémantique)

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
 - Détecter 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
- Exécution 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

Evaluation des expressions

Definition

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

$$Dom_e(v) = \{ \langle k, n \rangle \in Num \times Num \mid S/[v = n] \vdash e \implies k \}$$

Exemple : $e = sqr(v)$

$$Dom_e(v) = \{ \langle 1, 1 \rangle, \langle 1, 2 \rangle, \dots, \langle 2, 4 \rangle, \dots, \}$$

Comment faire mieux ?

Evaluation des expressions

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

Definition

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

$$SDom_e(v) = \{ \langle k, e' \rangle \in \text{Num} \times \text{ExprVar}_V \mid S/[v = e'] \vdash e \implies k \}$$

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

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

Ce qui serait en intension :

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

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 is $M1 + M2$
 - Si $M1$ et $M2$ sont définis alors N est défini
 - Si N défini impossible de trouver $M1$ et $M2$ 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 : $s\ s\ s\ 0$

Analyse des chemins (if then else)

Exemple (X est une variable prolog definissant un domaine, x est une variable du langage) :

```
?- 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)))] ;

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

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

No

Axiomatisation des naturels (Peano)

Générateurs

$0 : \rightarrow \text{nat};$

$S _ : \text{nat} \rightarrow \text{nat}$

Typage par adjonction de l'arité à chaque termes

Axiomes

$0 + N = N$

$S(N) + N' = s(N+N')$

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,
```



```
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,R).

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,R).

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).
```

Ce prédicat 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 ;
```

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
```

Exploration de domaines

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

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

```
?- loadfun([sqrt,sq],C),bigstep_p(C,x:=sqrt(s(s(s(X)))),[],R),  
    subspint(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)): (nat->nat)
```