

...suite chap3

## Structures de données | Opérateur de coupure !

### 4. Stratégie de résolution SLD.

Le langage prolog est basé sur une implémentation du principe de résolution SLD (Sélection Linéaire Défini)

#### 4.1. Définition :

La résolution SLD est un procédé récursif qui est réalisé en deux points :

##### *1. la stratégie de sélection :*

Consiste à résoudre d'abord le but le plus à gauche, la liste des (sous)butts est gérée par une pile.

##### *2. la stratégie de recherche :*

Explore (une branche de) l'arbre de dérivation en profondeur par le backtracking et par le choix des clauses selon leur ordre d'apparition.

Les points de choix sont les clauses non encore essayées lors de la résolution d'un but.

#### 4.2. L'arbre de résolution SLD.

Cet arbre est utilisé pour représenter la résolution SLD ou les dérivations. Il est construit de manière récursive comme suit :

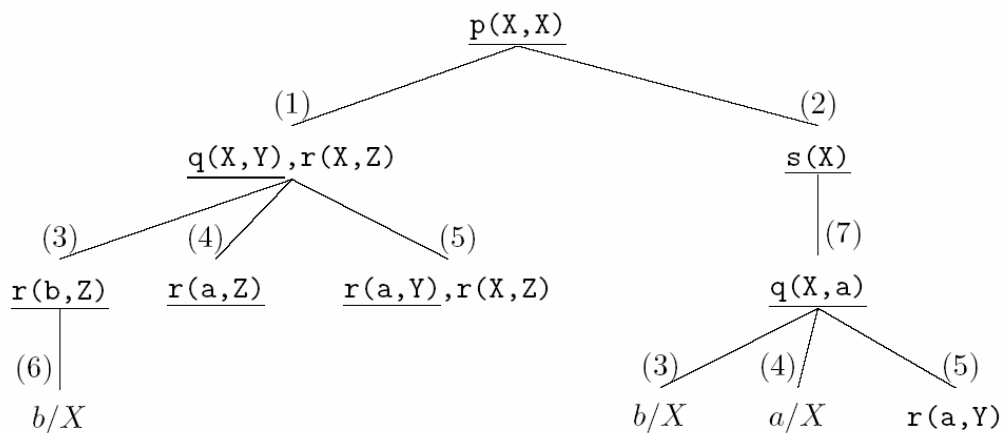
- 0) La racine contient le but initial ;
- 1) Chaque nœud représente la liste des (sous) butts courants ;
- 2) Les fils d'un nœud : on choisit le prédicat le plus à gauche dans la liste des buts courants, on examine le programme pour trouver les clauses définies dont la tête peut être unifiée au prédicat.
- 3) On construit un fils, dont la liste des buts courants est celle du nœud père, duquel on a éliminé les prédicats unifiés ;
- 4) On numérote la branche avec la clause définie utilisée ; s'il n'y a pas de clause alors la branche de l'arbre est un échec ;
- 5) Si dans un nœud, la liste des buts courants est vide, le nœud réussit !

#### 4.3. Exemple d'arbre SLD.

Pour le programme prolog suivant :

- (1) :  $p(X,X) :- q(X,Y), r(X,Z).$
- (2) :  $p(X,X) :- s(X).$
- (3) :  $q(b,a).$
- (4) :  $q(a,a).$
- (5) :  $q(X,Y) :- r(a,Y).$
- (6) :  $r(b,Z).$
- (7) :  $s(X) :- q(X,a).$

L'arbre SLD du but  $p(X, X)$  est le suivant :



### 5. L'opérateur de coupure : Cut ! Ou comment contrôler le retour arrière !

La stratégie de recherche standard Prolog, peut être modifiée par un prédicat opérateur de coupure : le Cut et noté !

#### 5.1. Définition.

L'effet de Cut dans la clause :

$$A :- A1, A2, \dots, Am, !, B1, \dots, Bn .$$

Abandonne les points de choix restants sur A, A1, ... , Am lorsque ! est selectionné ;

Il s'utilise comme un prédicat et modifie la recherche SLD, afin de ne pas « backtrack » sur les règles suivantes pour un but donné.

#### 5.2. Fonctionnement du Cut sur exemple.

- (1) : a :- u , v .  
 (2) : a :- b1 , b2 , ! , c1 , c2 .  
 (3) : a :- b , i .  
 (4) : a :- t .

- Pour réduire un but a prolog essaie (1) ;
- Lors du retour arrière, il essaie (2) ;
- Dans la clause (2), prolog essaie de résoudre b1 , b2 , et
- Si les buts b1 , b2 échouent , alors le ! ne sera pas atteint et il y aura un retour arrière pour essayer (3) puis (4) ;
- Si les (sous) buts b1 et b2 réussissent, le prochain (sous)but à résoudre afin de réduire le but a sera le cut !
- ! est réduit par convention à la « clause vide » et sa réduction fait que lors du retour arrière, Prolog n'essaie ni (3) , ni (4) .
- La satisfaction de ! dans a :- b1 , ..., bm , ! , c1 , ... , cn . supprime tous les choix accumulés entre l'appel de la clause et l'appel de coupure !

### 5.3. Utilisations courantes du Cut.

- En plaçant un Cut , on exprime à Prolog qu'on n'a trouvé la règle exacte pour un but précis « en arrivant là, on a pris la règle qu'il faut pour ce but »
- On veut faire échouer immédiatement un but sans chercher d'autres solutions, on ajoute Cut ! avec fail « en arrivant là, il faut arrêter de satisfaire ce but »
- On veut arrêter la recherche d'autres solutions par le retour arrière « on a trouvé la solution, inutile d'en chercher d'autres ».

**Remarque :** la sémantique d'une coupure dépend de l'ordre des clauses.

### 5.4. Exemple.

Insertion dans une liste d'entiers ordonnés.  
 Soit le paquet de clauses **insérer** suivant :

- (1) : insérer ( X, [], [X] ) :- !  
 (2) : insérer( X, [ Y | Ys ] , [ Y | Zs ] ) :- X > Y , ! , insérer( X, Ys, Zs ).  
 (3) : insérer( X, [ Y | Ys ] , [ X , Y | Ys ] ) :- X <= Y .

Soit le but G1 ?- insérer ( 5, [ 2 , 3 , 6 ] , U ) pour le réduire , clause (2) avec  
 { U = [ 2 | U1 ] } et  
 G2 = 5 > 2 , ! , insérer ( 5 , [ 3 , 6 ] , U1 ) puis (3)  
 G3 = ! , insérer ( 5 , [ 3 , 6 ] , U1 )

La réduction de ! est la clause vide, sa réduction supprime tous les choix depuis le but principal qui a appelé la clause contenant le Cut ! c'ad G1 jusqu'à G3.  
Le seul point de choix pour G1 étant (3) ce choix est supprimé.

On poursuit la résolution par

$G4 = \text{insérer}(5, [3, 6], U1) \quad \{ U1 = [3 | U2] \} \quad (2)$

$G5 = 5 > 3, !, \text{insérer}(5, [6], U2).$

$G6 = !, \text{insérer}(5, [6], U2).$

.....

$G7 = !, \text{insérer}(5, [6], U2) \quad \{ U2 = [6 | U3] \} \quad (2)$

**$G8 = 5 > 6, !, \text{insérer}(5, [], U3)$  le but  $5 > 6$  échoue donc**

Prolog fait retour arrière et retourne au but le plus récent, ayant des points de choix, soit le but G7, et essaye la clause (3) d'où :

$G7 = !, \text{insérer}(5, [6], U2) \quad \{ U2 = [5, 6] \} \quad (3)$

**$G8 = 5 \leq 6$**

**En remontant aux  $U_i$ , on obtient  $U = [2 | U1] = [2 | [3 | U2]] = [2 | [3 | [5, 6]]]$   
 $\{U = [2, 3, 5, 6]\} \rightarrow$  la solution à la requête du but principal G1**

**Résultat : L'élément 5 a été inséré à la bonne position !**

**Discussion de l'Exemple :**

Si le Cut, n'avait pas été utilisé dans la clause (2) :

(2) :  $\text{insérer}(X, [Y | Ys], [Y | Zs]) :- X > Y, !, \text{insérer}(X, Ys, Zs).$

Prolog aurait cherché une autre solution à G1, par retour arrière et donc par la clause suivante du programme, à savoir la clause (3) ; ce qui revient à dire que :

Prolog aurait tenté d'insérer l'élément 5 dans la liste donnée [2, 3, 6], par une autre manière, alors que cette opération n'a pas besoin d'être répétée puisque ayant déjà réussi (et sans tenir compte de l'échec ou la réussite de cette autre tentative)

Mais dans notre cas, ce point de choix est supprimé et la recherche d'une autre solution à G1 est éliminée !

**5.5. Conclusion :**

Le Cut permet d'écrire des programmes plus efficaces en temps calcul et permet un enrichissement du pouvoir d'expression de prolog.

Une forme déclinée de la négation dans la connaissance de type **si – alors – sinon.**

*« L'erreur est humaine, le système d'exploitation ne pardonne pas ! »*