Chapitre III.

Le contrôle du retour-arrière en Prolog (L'opérateur de coupure) et la négation par l'échec.

1. L'opérateur de coupure, le coupe-choix ou « cut » !

1.1. Introduction

Plusieurs clauses du programme peuvent être candidates à la résolution d'un but, l'interpréteur Prolog essaye toutes les clauses unifiables avec ce but, suivant leur ordre d'apparition dans le programme, en utilisant le mécanisme du retour-arrière. Or, il n'est pas toujours nécessaire d'essayer toutes les clauses. Par exemple, pour prouver que Ahmed est un père, il suffit de trouver un enfant X qui est un fils ou une fille de Ahmed, et il n'est pas nécessaire de trouver tous ses enfants.

L'opérateur de coupure est le principal outil de contrôle du langage Prolog. Son utilisation permet de couper certaines branches de l'arbre de recherche qu'on juge inutiles.

1.2. Signification opérationnelle de la coupure

La coupure notée « ! », est un prédicat prédéfini très important qui a pour effet de stopper la recherche de prolog dès qu'il aura trouvé une solution. Ce prédicat qui n'a aucune signification logique et qui est toujours prouvé avec succès, permet de contrôler l'espace de recherche et d'économiser le travail de Prolog en limitant l'exploration de l'arbre de recherche.

Remarques

Soit la clause a :-b1,b2,...,bm, !,c1,c2,...,cn.

- La satisfaction de la coupure, supprime tous les choix accumulés entre l'appel de la clause et l'appel de la coupure (choix sur a et sur b1,b2,...,bm).
- Toutes clauses susceptibles de réduire les buts situés après la coupure (c1,c2,...,cn) seront essayées. (voir exemple 1 et 2)

1.3. Exemple d'utilisation de la coupure

Exemple 1 : « Ahmed est-il un père ? »

```
%programme
père(ahmed, prénom1).
père(ahmed, prénom2).
père(ahmed, prénom3).
```

Que répond Prolog aux questions suivantes ?

```
?-père(ahmed,_).
?-père(ahmed,Prénom).
?-père(ahmed,_), !.
?-père(ahmed,Prénom), !.
```

Exemple 2 : Soit le programme Prolog suivant :

```
a :-b, !,c.
a :-j.
b :-f.
b :-g.
c :-d.
c :-e.
f. g.
j :-h.
j :-i.
```

Construire l'arbre de recherche de la question : **?-a.**

Exemple 3 : Soit le programme suivant :

```
s(a).                          (1)
s(b).                          (2)
t(a,b).                        (3)
t(b,a).                        (4)
p1(X,Y):- s(X),t(Y,X).         (5)
p1(a,a).                       (6)
p2(X,Y):- s(X),t(Y,X),!.       (7)
p2(a,a).                       (8)
p3(X,Y):- s(X),!,t(Y,X).       (9)
p3(a,a).                       (10)
p4(X,Y):- !,s(X),t(Y,X).       (11)
p4(a,a).                       (12)
```

Quel est l'arbre de recherche prolog à la question ?-p1(X,Y). , ?-p2(X,Y). , ?-p3(X,Y). et ?-p4(X,Y) ?

Exemple 4 : « affichage de tous les entiers de N à 1 par ordre décroissant »

Versions incorrectes sans coupure:

** afficher(N) :-writeln(N),K is N-1,afficher(K).

Pb : L'affichage ne s'arrête jamais.

Solution : ajouter une condition N>0

** afficher(N) :-N>0,writeln(N),K is N-1,afficher(K).

Pb : L'affichage s'arrête, mais Prolog répond par false en sortant.

Solution : utiliser la coupure !

Version correcte, en utilisant la coupure:

```
afficher(0) :- !. %ne fait rien et arrête la recherche en répondant true.
afficher(N) :-writeln(N),K is N-1,afficher(K). % appel recursive
```

Exemple 5 : « le maximum de deux nombres »

Sans coupure :

```
max(X,Y,X) :-X>=Y.
max(X,Y,Y) :-X<Y.
```

On remarque que ces deux clauses s'excluent mutuellement, c-à-d, si la 1ière réussit, l'autre va forcement échouer (il n'est pas nécessaire de l'essayer).

Pour cela, on utilise la coupure.

Avec coupure :

```
max(X,Y,X) :-X>=Y, !.
max(X,Y,Y).
```

Attention !

Prolog répond *true* à la question ?-max(5,2,2).

Puisque le but n'est pas unifiable avec la tête de la 1ière clause et l'est avec la seconde.

Donc on passe directement à la 2ième clause.

Version correcte avec la coupure :

```
max(X,Y,Z) :-X>=Y, !,Z=X.
max(X,Y,Y).
```

On a modifié la version précédente de manière à ce que tout but unifiable à la tête de la 2ième clause soit unifiable à la tête de la 1ière clause.

Dans cette version, Prolog répond *false* à la question ?-max(5,2,2).

1.4. Utilité de la coupure

- ☒ éliminer les points de choix menant à des échecs certains.
- ☒ supprimer certains tests d'exclusion mutuelle dans les clauses.
- ☒ permettre de n'obtenir que la première solution de la démonstration: exprimer le déterminisme.
- ☒ assurer la terminaison de certains programmes.
- ☒ contrôler et diriger la démonstration.
- ☒ définir la négation (à expliquer plus tard).
- ☒ exprimer le Si-alors-Si non.

1.5. Les dangers de la coupure

Considérez les programmes suivants :

1)Version 1 :

```
enfants(ahmed, 3).
enfants(ali, 1).
enfants(X, 0).
```

2)Version 2 :

```
enfants(ahmed, 3) :- !.
enfants(ali,1) :- !.
enfants(X, 0).
```

et considérez la question ?-enfants(ahmed, N). dans les deux cas.
1) N=3 et N=0
2) N=3 mais,
?-enfants(ahmed, 0). donne aussi « true ».
Le programme correct serait :

```
enfants(ahmed,N) :- !, N=3.
enfants(ali,N) :- !,N=1.
enfants(X, 0).
```

1.6.Expression du Si-alors-Si non

Il est possible en Prolog de mimer le Si-alors-Si non des langages de programmation impératifs. Pour cela, on peut définir une expression de la forme :

```
D :-A, !,B. % si A s'unifie, on essaie d'unifier B.
D :-C. % si non, on unifie C.
```

Ce qui revient à : « **si** A **alors** B, **si non** C »

Exercice :

Soit un programme contient des faits tels que :

```
fille(fille1).
fille(fille2).
...
```

Ajouter à ce programme la connaissance : « **si** un enfant est fille, il aime jouer aux poupées **si non** il aime jouer au ballon »

Solution incorrecte :

```
aime(X, poupées) :-fille(X), !.
aime(X, ballon).
%Pour la question ?-aime(fille1,ballon), Prolog répond true.
```

Solution correcte :

```
aime(X, Quoi) :-fille(X), !, Quoi=poupées.
aime(X, ballon).
```

2. La négation par l'échec

2.1. Hypothèse du monde clos

- Toute assertion qui n'est pas explicitement vraie est fausse.
- Une réponse *false* ne signifie pas forcément qu'une question Q est fausse, mais qu'il n'est pas possible de prouver Q à partir de la base de faits et de règles.

Exemple : soit la base de connaissances suivante :

```
humain (ahmed).
animal(chèvre).
humain(X) :-not(animal(X)).
```

A la question ?-animal(loup).
Prolog répondra *false*.

Cela ne veut pas dire que le loup n'est pas un animal mais que animal(loup) n'est pas démontrable à partir de la base de connaissances.

2.2. L'atome *fail*

fail est un atome réservé de Prolog qui n'est jamais démontrable, il provoque donc un échec de la démonstration où il figure, puisque il n'y a pas de clause dont la tête est unifiable à *fail*.

Ainsi, les réponses aux questions suivantes sont *false* :

?-fail.
?-A,fail.
?-a,fail.
?-2==2,fail.
?- fail,2==2.
...

2.3. La négation par l'échec en Prolog (notée *not* ou \+)

La négation en prolog peut être exprimée en utilisant l'opérateur de la coupure et l'atome *fail* de la façon suivante :

not (P) :- P, !, fail. % Sémantique procédurale : si P réussit, not P échoue ; si P échoue, not P réussit.
not (P).

- Pour démontrer not P, Prolog essaie de démontrer P
- S'il réussit, un coupe-choix élimine les points de choix éventuellement créés durant cette démonstration puis échoue.
- Si la démonstration de P échoue, Prolog utilise la deuxième règle qui réussit.

Exemple :

?- not(fail).
true
?- not(X=1).% X ne s'unifie pas à 1.
false
?- X=0, not(X=1).
X= 0
?- not(X=1), X=0.
false