Coupure et négation

1. PILES DE RESOLUTION

1.1. Introduction

Nous introduisons l'apprentissage du fonctionnement du langage Prolog : comment résolut-il une question à partir de fait, …?

La résolution d'une question peut, globalement, être décrite comme deux étapes successives : l'unification (la recherche dans les fais d'une tête de clause correspondant à la question) et la résolution des conditions. Nous allons ici, plus particulièrement nous intéresser à la deuxième partie de cette méthode et aux effets qu'elle inclue dans la programmation même en langage Prolog. Comme décrit lors de la résolution "à la main" des exemples précédents, à chaque étape de la résolution d'un problème, Prolog effectue un "choix" séquentiel dans les clauses résultantes de l'unification de la question et de données. Par exemple :

Si les faits sont définis comme suit :

```
enfant(lisa, lucile).

enfant(alice, lucile).

enfant(lisa, mathieu).

enfant(leo, sophie).

enfant(marc, morgane).

enfant(francis, anne).
```

Et la question "quels sont les enfants de lucile" est posé à prolog :

```
?- enfant(X, lucile).
```

Le résultat de l'unification de la question avec les données est composé de deux clauses : {enfant(lisa, lucile), enfant(alice, lucile)}. A partir de ce résultat, prolog retourne la première réponse (`X=lisa`) puis, en cas de demande (avec " `;`"), la seconde (`X=alice`). On peut ainsi dire qu'il conserve les trois choix possible puis, selon la suite de l'exécution, passe du premier au second puis au troisième et ainsi de suite jusqu'à la fin des faits. Cette méthode est mise en œuvre pour la résolution de chaque question et chaque condition lors de l'exécution d'un programme prolog.

1.2. Notion de coupure (ou blocage du retour en arrière)

Il est possible, dans un programme prolog, d'influence la méthode de résolution utilisée par le langage et plus particulièrement de modifier la pile de résolution. Ceci en utilisant la coupure (le *cut*) permettant d'annuler les choix restant en cours dans la pile de résolution. Ainsi dans l'exemple précédent, l'adjonction d'une coupure après le prédicat enfant dans la question a

comme effet de restreindre les choix possible à une seule réponse qui sera alors la première trouvée. En fait, on peut donner la définition possible de la coupure : tous les choix restés en suspend à gauche de la coupure dans une clause sont annihilés et toutes les clauses suivant la clause contenant une coupure sont élagués. Le cut "!" est un mécanisme pour empêcher le retour en arrière (*back-tracking*).

```
p(X) :- a(X), !, b(X).
```

Une fois un cut franchi :

Il coupe touts les clause en dessous de lui, !

coupe tous les buts à sa gauche,

par contre il y a possibilité de retour arrière sur les sous buts à la droite du cut.

L'effet principal de l'utilisation de la coupure est double : au niveau de la rapidité d'exécution d'un programme, du fait de l'élagage de choix dans la pile, la vitesse est accrue. Au niveau de la syntaxe, et comme nous allons le voir, l'écriture de certains devient inutile.

Les conséquences d'utilisation du cut sont d'améliorer l'efficacité et d'exprimer le déterminisme, mais il revient à faire fonctionner prolog comme un langage procédural et il peut introduire des erreurs de programmation vicieuses, car il modifie le comportement d'un programme dans sa manière de rechercher des solutions en cas d'échec, il faut donc l'utiliser avec précaution.

Le cut supprime tous les points de choix placés entre le début de résolution de la clause et le moment où il est résolu. Le cut coupe toutes les branches ET qui sont à sa gauche, et qui dépendent du même nœud que lui, et toutes les branches OU au même niveau que lui.

Regardez l'exemple qui suit : considérons le programme suivant pour lequel Z est le maximum de {X, Y} quand on a max (X,Y,Z).

```
max(X, Y, Z) :- inferieur(X, Y), !, Z=Y.
max(X, Y, Z) :- Z=X.
inferieur(X, Y) :- X=1, Y=2.
inferieur(X, Y) :- X=1, Y=1.
inferieur(X, Y) :- X=2, Y=2.
```

Observez les réponses :

```
1 ?- max(1,1,Z).
Z = 1.

2 ?- max(1,2,Z).
Z = 2.

3 ?- max(X,Y,Z).
X = 1,
Y = 2,
Z = 2.
```

Analysons la deuxième question et réponse : le moteur commence par unifier la question `max(1,2,Z)` avec la tête de la clause `max(X,Y,Z) :- inferieur(X,Y), ! Z=Y.`

Il unifie alors X à 1 et Y à 2. Le moteur résout le premier sous-but `inferieur(1,2).` Il a trois possibilités, il choisit la première qui réussit.

Le moteur exécute alors le cut !. La coupure coupe la branche ET correspondant à inferieur(1,2). Et touts les branches qui se trouvent en-dessous, coupant ainsi les deux autres choix possibles non encore testés pour inferieur. Le cut coupe aussi toues les branches OU du même niveau que lui, il coupe donc la branche qui conduit à Z=1.

Le moteur exécute alors le troisième atome, Z=2.

Si l'on force le back-tracking après avoir envoyé Z=2, le moteur échoue parce que tous les points de choix ont été coupés.

Exemple la fusion de deux listes triées est bien déterministe : on peut donc éliminer les possibilités de retour en arrière :

Fusion

```
fusion([], L, L).
fusion(L, [], L).
fusion([X|Xs], [Y|Ys], [X|Zs]):- X < Y, !, fusion(Xs, [Y|Ys], Zs).
fusion([X|Xs], [X|Ys], [X, X|Zs]):- !, fusion(Xs, Ys, Zs).
fusion([X|Xs], [Y|Ys], [Y|Zs]):- X > Y, !, fusion([X|Xs], Ys, Zs).
```

Les trois dernières règles du prédicat ‘fusion’ sont mutuellement exclusives, cela a comme conséquence qu’une seule parmi elles réussira. Nous savons que lorsqu’une d’elles est vérifiée, essayer les autres est inutile car elles échoueront forcément. C’est pour cela que la coupure est juste après le test de supériorité, égalité ou infériorité si un test est vrai les autres tests qui fort partie des autres règles de fusion ne seront pas exécutés.

```
?- fusion([1,3,5], [4,5,6], X).
X = [1, 3, 4, 5, 5, 6]
true
```

Règles d'utilisation :

La bonne utilisation est l'utilisation des "cuts verts", car ils ne modifient pas le sens d'un programme. Par exemple la fusion décrite ci-dessus. Les "cuts rouges" (ou paresseux) sont une mauvaise utilisation, par exemple le minimum :

```
min(X, Y, X) :- X < Y, !.
min(X, Y, Y).
```

Ce prédicat conduit bien au calcul du minimum pour la première solution. Le cut est nécessaire pour éviter de produire une seconde solution au cours d'un retour en arrière. Si l'on a la règle :

```
P:- B1, …, Bi-1, min(2, 3, Z), Bi, …, Bn.
```

A la première exécution, Z = 2. Si l'on revient en arrière à cause d'un échec des but, B1, …, Bn, et si min/3 ne comporte pas de cut, on obtient la valeur Z= 3 car min/3 aurait pu produire deux solutions. On repartirait alors avec cette valeur pour trouvers but B1, …, Bn.

Autre cut rouge :

```
ifthenelse(P, Q, R):- P, !, Q).
ifthenelse(P, Q, R):- R.
```

1.3. Exemples avec coupure

1) Augmentation de la vitesse :

La coupure permet, par exemple, de limiter les choix en cas de recherche d'un élément dans une liste et ainsi d'augmenter la rapidité de l'exécution d'un programme utilisant cette recherche.

```
recherche(X, [X|R]):- ! /* L'élément est trouvé, supp des autres choix ***/.
Recherche(X, [Y|R]):- recherche(X, R).
```

L'effet secondaire de l'utilisation de la coupure dans ce programme est le suivant: considérons la recherche de l'atome a dans la liste [c,a,b,d,a], il s'y trouve à deux reprises.

Lors de la résolution de l'appel :

```
?- recherche(a,[c,a,b,d,a]).
```

true.

La réponse est yes: l'élément est trouvé, l'autre choix : non, la recherche est coupée.

2) Boucle infinie :

On peut simuler les boucles, par exemple pour la saisie de données, par l'utilisation de la coupure : on définit pour cela le prédicat répétition de la manière suivante :

```
repetion.
repetion :-repetition.
```

Ce prédicat est toujours vrai et il permet de construire une boucle :

```
repetition.
repetition:- repetition.

saisie(X):- display('entrez un nombre > 0 et <100).
repetition,
read(X),
X>0, X<100, !, traitement(X).

traitement(R): X is R*R.
```

Exercice :

En utilisant le prédicat append et le cut, écrivez un programme prolog pour le tri à bulles.

2.1. Négation

Le symbole de la négation est "\+" (anciennement *not*).

Un programme logique exprime ce qui est vrai. La négation, c'est donc ce que l'on ne peut pas trouvé :

Si Clause1 réussi =>\+ Clause1 échoue

Si Clause 2, échoue => \+ Clause2 réussit.

Le prédicat not se définit par:

```
not(P) :- P, !, fail.
not(P) :- true.
```

Avec la négation il vaut mieux que les variables soient instanciées.

```
?- \+member(X, [a, b]).
false.
```

Le programme identifie X à une des variables, réussit et le not le fait échouer. De même dans une règle, il faut veiller à l'instanciation des variables. Avec le programme :

```
marie(francois).
etudiant(paul).
etudiant_celibataire(X) :- \+marie(X), etudiant(X).
```

La requête suivant échoue :

```
?- etudiant_celibataire(X).
false.
```

Car X s'unifie à François puis le not fait échouer. Alors que :

```
?- etudiant_celibataire(paul).
true.
```

Cette requête réussit car X = paul donc \+marie(paul) est vrai. Pour que la règle produise les résultats attendus, il faut inverser les sous buts :

```
etudiant_celibataire(X) :- etudiant(X), \+marie(X).
```

De cette façon X sera instancié avant d'être soumis au not.

Exercice :

Pouvez-vous modifier le programme sur l'énigme d'einstein en y ajoutant des coupures ou des négations ?