Cours de programmation fonctionnelle et logique

Cours 10 : Contrôle d'exécution et coupure

Table des matières

1 Motivation de la coupure

1.1 Premier exemple : appartenance à une liste

Soit la fonction qui vérifie qu'un élément est bien dans une liste.

Première solution :

```
membre1(X,[Y|L]) :- X = Y.
membre1(X,[Y|L]) :- X \= Y, membre1(X,L).
```

Deuxième solution :

```
membre2(X,[X|L]).
membre2(X,[Y|L]) :- membre2(X,L).
```

L'idée ici, c'est que le cas d'égalité a été traité avec la première règle et que l'on n'a pas à en tenir compte lors de l'écriture de la deuxième règle.

Évaluation du travail effectué

Conclusions :

- L'écriture du programme a de grosses conséquences sur son exécution.
- Seule la première occurence de X dans L nous intéressait ici. Nus aurions aimé pouvoir spécifier que le programme devait s'arrêter dès la première occurence trouvée.

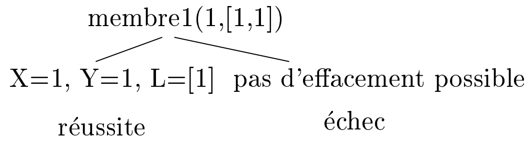

FIG. 1: Évaluation avec le prédicat `membre1`.

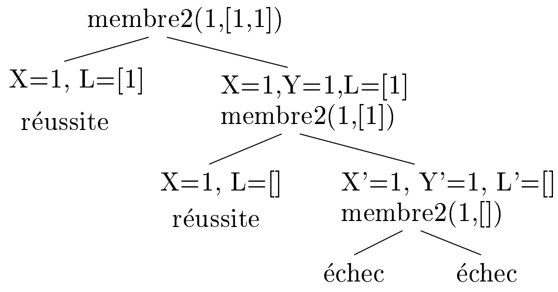

FIG. 2: Évaluation avec le prédicat `membre2`.

1.2 Deuxième exemple : fonctions booléennes

On s'intéresse ici au codage des fonctions booléennes. On veut associer une valeur booléenne, 0 ou 1, à chaque formule.

Soit une formule `p`. Si `p` s'efface, `p` est vraie, d'où :`valeur_booléenne(p,1) :- p.`

Comment faire pour associer la valeur 0 à `p` quand `p` ne s'efface pas ? Première idée :

```
valeur_booléenne(P,1) :- P.
valeur_booléenne(P,0).
```

```
?- valeur_booléenne(membre1(2,[1]),Y).

Y = 0 ;

No
?- valeur_booléenne(membre1(1,[1]),Y).
```

```
Y = 1 ;

Y = 0 ;

No
```

Si p ne s'efface pas, la valeur 0 lui est associée, et tout va pour le mieux. Pas contre, si p s'efface, les valeurs 0 et 1 lui sont asociées ! On voudrait que si p s'efface, la seule valeur associée à p soit 1. Solution : la coupure !

2 La coupure

Notation : « ! » (point d'exclamation). La coupure permet d'oublier des points de choix lors de l'effacement.

Solution de notre problème de valeurs booléennes :

```
valeur_booléenne(P,1) :- P, !.
valeur_booléenne(P,0).
```

```
?- valeur_booléenne(membre1(2,[1]),Y).

Y = 0 ;

No
?- valeur_booléenne(membre1(1,[1]),Y).

Y = 1 ;

No
```

Interprétation

Nous avons deux cas à consuidérer :

1. p s'efface (p peut être évalué à vrai, cf. figure 3). Une fois que p a été évalué à vrai, on « évalue » la coupure (« ! »). Celle-ci supprime le *backtracking* sur les choix mis en attente entre le moment où elle a été introduite et celui où elle a été effacée.

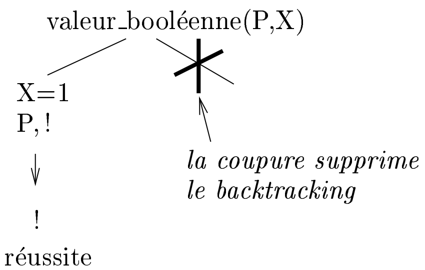

FIG. 3: Coupure : cas où p s'efface.

2. p ne s'efface pas (p ne peut pas être évalué à vrai, cf. figure 4). Come p ne s'efface pas, la coupure n'est pas atteinte et ily a *backtracking*.

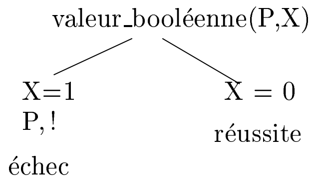

FIG. 4: Coupure : cas où p ne s'efface pas.

Dans la recherche d'appartenance d'un élément à une liste, la coupure permet d'arrêter le programme lors de la découverte de la première solution possible :

```
membre3(X,[X|L]) :- !.
membre3(X,[Y|L]) :- membre2(X,L).
```

3 Vue théorique

Soit $b_1, ..., b_n$ la suite de buts à effacer à un instant donné, et soit $c_1, ..., c_m$ la suite des choix en attente à cet instant, du plus ancien au plus récent. $c_1, ..., c_m$ sont donc des buts déjà effacés. *Notation* : $[c_1, ..., c_m]b_1, b_2, ..., b_n$.

Supposons que pour effacer b_1 on utilise la règle $p : -q_1, ..., q_i, !, q_{i+1}, ..., q_l$, avec la contrainte $b_1 = p$. ON obtient donc le nouvel état :

$$[c_1, ..., c_m, b_1]q_1, ..., q_i, !, q_{i+1}, ..., q_l, b_2, ..., b_n$$

Les effacement successifs donnent :

$$[c_1, ..., c_m, b_1, q_1]q_2, ..., q_i, !, q_{i+1}, ..., q_l, b_2, ..., b_n$$

.

$$[c_1, ..., c_m, b_1, q_1, ..., q_i]!, q_{i+1}, ..., q_l, b_2, ..., b_n$$

.

$$[c_1, ..., c_m]q_{i+1}, ..., q_l, b_2, ..., b_n$$

. L'effacement de la coupure supprime donc tous les choix (quelque soit la règle concernée) mis en attente, depuis le plus récent et jusqu'à celui dont l'effacement a provoqué l'apparition de la coupure. Ici, la suite des choix en attente est rétablie à ce qu'elle était juste avant l'effacement de b_1.

Autre exemple : la conditionnelle

On veut réaliser ici une conditionnelle *if C then A else B*.

```
ifthenelse(C,A,B) :- C, !, A.
ifthenelse(C,A,B) :- B.
```

Si `C` s'évalue à vrai on s'interdit d'avoir recours à la deuxième règle et on évalue `A`. Sinon, on évalue `B`.

Attention : les règles doivent être écrites dans cet ordre !

Retour sur les fonctions booléennes

```
et(P,Q,1) :- P, Q, !.
et(P,Q,0).
ou(P,Q,1) :- P, !.
ou(P,Q,1) :- Q, !.
ou(P,Q,0).
```

4 Les dangers de la couprue

Il faut toujours prouver qu'introduire la coupure ne va pas nous faire croire, à tort, qu'un problème n'a pas de solutions.

```
homme(pierre).
homme(jacques).
grand(jacques).
?- homme(X),grand(X).

X = jacques ;

No

?- homme(X),!,grand(X).

No
```

5 La négation

`non(P) :-` si `P` est vrai alors renvoyer faux... Comment coder une telle propriété ? On utilise le prédicat `fail` qui est toujours faux.

```
non(P) :- P, fail.
non(P).
```

Si `p` est vrai alors faux, sinon vrai. Le problème ici, c'est que si `p` est vrai, `non(P)` vaut simultanément vrai et faux. On doit donc utiliser la coupure :

```
non(P) :- P, !, fail.
non(P).
```

On effectue ce que l'on appelle une « négation par l'échec ». La coupure oblige la première régle à échouer en interdisant le *backtracking*.

Signification théorique de la négation

`p` est vrai, ou `p` s'efface, s'il existe **une** utilisation qui permet d'effacer `p`. `p` est vrai s'il existe au moins une instanciation des vraiables qui le composen qui lui permette de s'évaluer à vrai. Donc `non(P)` est vrai si et seulement si `p` ne s'évalue **jamais** à vrai.

On suppose que l'on dispose d'un prédicat `chiffre` qui s'évalue à vrai sur les éléments de l'ensemble $\{0,1,2,3,4,5,6,7,8,9\}$.

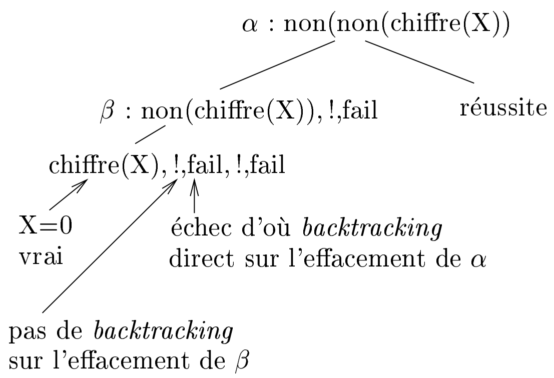

FIG. 5: Signification de la double négation.

Dans l'exemple de la figure 5, on a vérifié que `chiffre(X)` pouvait s'effacer, mais on n'a pas créé de conditions sur `X`, contrairement à un simple appel à `chiffre(X)`. Donc ici : `non(non(chiffre(X)))` signifie « exise-t-il une solution ? », alors que `chiffre(X)` signifie « qu'est-ce qu'il existe comme solution ? ».

```
?- chiffre(X).

X = 0 ;

X = 1 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

X = 6 ;

X = 7 ;

X = 8 ;

X = 9 ;
```

```
No
?- non(non(chiffre(X))).

X = _G219 ;

No
```

6 Exemple : résolution des trinômes du second degré

On s'intéresse donc aux équations de la forme : $a.x^2 + b.x + c$.

```
résoudre(0,0,0,X) :-
   !, string_to_list(X,"indéfini").
résoudre(0,0,_,X) :-
   !, string_to_list(X,"aucune solution").
résoudre(0,B,C,X) :-
   !, X is -C/B.
résoudre(A,B,C,X) :-
   D is B*B - 4*A*C, solutions(A,B,D,X).
solutions(A,B,0,S) :-
   X is - B/(2*A), !,
   string_to_list(T,"Solution double : "),
   string_to_atom(SX,X),
   string_concat(T,SX,S).
solutions(A,B,D,S) :-
   D < 0, !, X is -B/(2*A),
   E is sqrt(-D), Y is E/(2*A),
   string_to_list(P," + i*("),
   string_to_list(N, " - i*("),
   string_to_list(PAR1,"), "),
   string_to_list(PAR2,")"),
   string_to_atom(SX,X),
   string_to_atom(SY,Y),
   string_concat(SX,P,PP),
   string_concat(PP,SX,PPP),
   string_concat(PPP,PAR1,PPPP),
   string_concat(SY,N,NN),
   string_concat(NN,SY,NNN),
   string_concat(NNN,PAR2,NNNN),
   string_concat(PPPP,NNNN,S).
solutions(A,B,D,S) :-
   E is sqrt(D), X is (-B+E)/(2*A),
   Y is (-B-E)/(2*A),
   string_to_atom(SX,X),
   string_to_atom(SY,Y),
   string_to_list(ESP,", "),
   string_concat(SX,ESP,SS),
   string_concat(SS,SY,S).
```