

# Programmation Logique – TP 5

## Méta - Programmation

Narendra Jussien

2001 – 2002

### 1 Le méta interprète de base

**Rappel de cours :** Un méta interprète de Prolog en Prolog peut s'écrire de la façon suivante :

```
prouve( true ) :- !.
prouve( (A,B) ) :- !,
    prouve(A),
    prouve(B).
prouve( But ) :-
    predefini(But), !,
    But.
prouve( But ) :-
    clause(But,Corps),
    prouve(Corps).
```

Les cuts (!) ne sont *a priori* pas indispensables mais SICStus Prolog ne permet pas un appel au prédicat `clause/2` lorsque le but recherché est un prédicat prédéfini (une erreur est alors déclenchée).

D'autre part, les appels à `clause/2` ne peuvent concerner que des prédicats préalablement déclarés comme étant `dynamic`. Par exemple :

```
:- dynamic member/2. % le pr'\{e}dicat member d'arit'\{e} 2 est dynamique
member(X,[X|_]).
member(X,[_|Ys]) :-
    member(X,Ys).

:- dynamic fact/2.
fact(X,1) :- X = 0.
fact(N,F) :-
    N1 is N - 1,
    fact(N1,F1),
    F is N * F1.
```

#### 1.1 Une première amélioration

On veut pouvoir suivre de manière automatique la démonstration réalisée par le système Prolog. Pour cela, on se propose d'améliorer l'interprète de base en ajoutant un affichage de la démonstration en cours.

On cherche donc à obtenir le comportement suivant :

```
| ?- prouve_trace(fact(2,N)).
je cherche \{a} prouver : fact(2,_61)
pour cela je cherche \{a} prouver : 2=0
```

```

j'appelle le but pr\{e}d\{e}fini 2=0 -> il n'est pas v\{e}rifi\{e} -- backtrack --
  pour cela je cherche \{a} prouver : _444 is 2-1,fact(_444,_436),_61 is _436*2
j'appelle le but pr\{e}d\{e}fini _444 is 2-1 -> il est v\{e}rifi\{e}
je cherche \{a} prouver : fact(1,_436)
  pour cela je cherche \{a} prouver : 1=0
j'appelle le but pr\{e}d\{e}fini 1=0 -> il n'est pas v\{e}rifi\{e} -- backtrack --
  pour cela je cherche \{a} prouver : _1559 is 1-1,fact(_1559,_1551),_436 is _1551*1
j'appelle le but pr\{e}d\{e}fini _1559 is 1-1 -> il est v\{e}rifi\{e}
je cherche \{a} prouver : fact(0,_1551)
  pour cela je cherche \{a} prouver : 0=0
j'appelle le but pr\{e}d\{e}fini 0=0 -> il est v\{e}rifi\{e}
j'appelle le but pr\{e}d\{e}fini _436 is 1*1 -> il est v\{e}rifi\{e}
j'appelle le but pr\{e}d\{e}fini _61 is 1*2 -> il est v\{e}rifi\{e}

N = 2 ?

```

On affichera donc les développements réalisés sur le remplacement d'une tête de clause par le corps de clause correspondant et les appels aux buts prédéfinis qu'ils réussissent ou non.

On pourra utiliser le prédicat prédéfini `format/2`. Pour des informations sur l'utilisation de ce prédicat, on se référera au manuel en ligne de SICStus Prolog disponible à l'adresse suivante : <http://www.cling.gu.se/datorinfo/sicstus/sicstus3/sicstus3.html>.

## 1.2 Une deuxième amélioration

Le principe de la trace est intéressant mais elle n'est pour l'instant pas très lisible, on se propose donc d'améliorer l'interprète précédent en intégrant une visualisation du niveau courant de la démonstration.

Nous choisirons de représenter le niveau par des séries de 2 tirets (code 45 en Prolog). On obtiendra donc la trace suivante :

```

| ?- prouve_evolue(fact(2,N)).
  je cherche \{a} prouver : fact(2,_61)
-- je cherche donc \{a} prouver : 2=0
-- j'appelle le but pr\{e}d\{e}fini 2=0 -> il n'est pas v\{e}rifi\{e} -- backtrack --
-- je cherche donc \{a} prouver : _493 is 2-1,fact(_493,_485),_61 is _485*2
-- j'appelle le but pr\{e}d\{e}fini _493 is 2-1 -> il est v\{e}rifi\{e}
-- je cherche \{a} prouver : fact(1,_485)
---- je cherche donc \{a} prouver : 1=0
---- j'appelle le but pr\{e}d\{e}fini 1=0 -> il n'est pas v\{e}rifi\{e} -- backtrack --
---- je cherche donc \{a} prouver : _1697 is 1-1,fact(_1697,_1689),_485 is _1689*1
---- j'appelle le but pr\{e}d\{e}fini _1697 is 1-1 -> il est v\{e}rifi\{e}
---- je cherche \{a} prouver : fact(0,_1689)
----- je cherche donc \{a} prouver : 0=0
----- j'appelle le but pr\{e}d\{e}fini 0=0 -> il est v\{e}rifi\{e}
---- j'appelle le but pr\{e}d\{e}fini _485 is 1*1 -> il est v\{e}rifi\{e}
-- j'appelle le but pr\{e}d\{e}fini _61 is 1*2 -> il est v\{e}rifi\{e}

N = 2 ?

```

## 2 Un interprète avec continuation

Il est souvent intéressant au cours d'une démonstration Prolog de connaître précisément l'ensemble des buts restant à prouver pour obtenir une solution. L'interprète de base ne permet pas cela, puisqu'il *oublie* en quelque sorte ce qui lui reste à prouver (on peut s'en rendre compte au traitement du cas (A,B) ou la preuve de A se fait sans savoir qu'il restera à prouver B.

Un méta interprète qui conserve une telle information est appelé un *interprète avec continuation*. On va dans un tel interprète afficher à chaque instant de la résolution la liste de buts restant à prouver.

Pour l'écrire, on complètera le canevas suivant.

```
prouve_cont(true, ?? ).
prouve_cont(?? , ??) :-
    prouve_cont(??,??).
prouve_cont((A,B), ?? ) :-
    prouve_cont(A, ?? ).
prouve_cont( But,Cont ) :-
    predefini(But), !,
    But,
    prouve_cont(?? , ??).
prouve_cont(But, ??) :-
    clause(But,Corps),
    prouve_cont(??,??).
```

*Attention*, les mêmes problèmes que pour le méta interprète de base se présentent lorsque l'on cherche à appeler `clause/2` pour un prédicat prédéfini. Compléter en conséquence le canevas précédent avec des cuts judicieusement placés. On pourra aussi ajouter des informations imprimées à l'exécution pour visualiser plus précisément le fonctionnement.

Le fonctionnement attendu de l'interprète est alors le suivant :

```
| ?- prouve_continuation(fact(2,N)).
* pour prouver fact(2,1), il faut que je prouve 2=0,
  il me reste aussi \'{a} prouver : []
* 2=0 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a \{e}chou\{e} -- backtrack --
* pour prouver fact(2,_61), il faut que je prouve _355 is 2-1,fact(_355,_347),_61 is _347*2
  il me reste aussi \'{a} prouver : []
* _355 is 2-1 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a r\{e}ussi
* pour prouver fact(1,1), il faut que je prouve 1=0,
  il me reste aussi \'{a} prouver : [_61 is 1*2]
* 1=0 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a \{e}chou\{e} -- backtrack --
* pour prouver fact(1,_347), il faut que je prouve _1528 is 1-1,fact(_1528,_1520),_347 is _1520*1
  il me reste aussi \'{a} prouver : [_61 is _347*2]
* _1528 is 1-1 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a r\{e}ussi
* pour prouver fact(0,1), il faut que je prouve 0=0,
  il me reste aussi \'{a} prouver : [_347 is 1*_1,_61 is _347*2]
* 0=0 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a r\{e}ussi
* _347 is 1*1 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a r\{e}ussi
* _61 is 1*2 est un pr\{e}dicat pr\{e}d\{e}finsi -> il a r\{e}ussi

N = 2 ?
```

### 3 Un interprète en largeur d'abord

Supposons défini le prédicat suivant :

```
fact(N,F) :-
    N1 is N - 1,
    fact(N1, F1),
    F is F1 * N1.
fact(0,1).
```

La demande de résolution de `fact(2,N)` provoque, tout naturellement, une récursion infinie ce qui fait que le programme ne s'arrête jamais. C'est assez frustrant

puisqu'il existe une façon de prouver `fact(2,N)` (en sélectionnant la deuxième clause au bon moment).

Une façon de résoudre ce problème consiste à écrire un interprète Prolog *en largeur d'abord*, par opposition au fonctionnement classique de Prolog qui fonctionne en profondeur d'abord. Un tel interprète permet alors de s'affranchir des contraintes sur le placement des clauses les unes par rapport aux autres.

Le principe est ici de conserver tous les *chemins* partiels de démonstration et d'avancer d'une étape à la fois dans chacun de ces chemins. Pour cela, on utilisera une liste contenant les chemins courants et à chaque fois que l'on franchira une étape pour une liste donnée on mettra le chemin de démonstration à terminer résultant, à la fin de notre liste courante de traitement.

Dès qu'un chemin de résolution se réduit à `true` la démonstration est terminée. L'interprète que l'on cherche à écrire ressemble alors à quelque chose comme :

```
prouve_largeur( But ) :-
    exploration_simultanee([ [But] ] ).

exploration_simultanee([ [true] | _ ] ).
exploration_simultanee([ [true|Xs] | R] ) :-
    ... % \{a} compl\{e}ter !
exploration_simultanee([ [ (A,B) | Reste] | R] ) :-
    ... % \{a} compl\{e}ter !
exploration_simultanee([ [But|Reste] | R] ) :-
    predefini(But),
    ... % \{a} compl\{e}ter !
exploration_simultanee([ [But | Reste] | R] ) :-
    But \== true, not(predefini(But)),
    ... % \{a} compl\{e}ter !
```

On va dans un premier temps compléter le canevas précédent sachant qu'il manque la gestion d'un cas bien précis, qu'il faudra donc prendre en compte. D'autre part, peut-on ici utiliser le `cut` pour exprimer le non déterminisme des différentes possibilités ?

Pour écrire cet interprète, on pourra utiliser le prédicat `findall(f(X),predicat(X),L)` qui renvoie dans `L` la liste de tous les `f(X)` avec `X` permettant de conduire l'appel de `predicat(X)` vers un succès.

```
| ?- findall(X,member(X,[a,b,c]),L).
L = [a,b,c]
| ?- findall(couple(X,Y), append(X,Y,[a,b,c]), L).
L = [ couple([], [a,b,c]),
      couple([a], [b,c]),
      couple([a,b], [c]),
      couple([a,b,c], [])
    ]
```

**Attention :** l'interprète en largeur d'abord que nous écrivons ici ne fonctionne qu'avec des programmes Prolog dits *normalisés*. Nous dirons qu'un programme est normalisé si toutes les unifications dans les têtes de clauses sont explicites :

```
fact(N,F) :-
    N1 is N - 1,
    fact(N1, F1),
    F is F1 * N.
fact(0,1). % ici unification implicite

member(X,[X|_]). % ici aussi
```

```

member(X,[_|Ys]) :-
    member(X,Ys).

%% Version normalis\{e}e
fact_norm(N,F) :-
    N1 is N - 1,
    fact_norm(N1, F1),
    F is F1 * N.
fact_norm(N,F) :- N = 0, F = 1.          % ici unification explicite

member_norm(X,[Y|_]) :- X = Y.
member_norm(X,[_|Ys]) :-
    member_norm(X,Ys)

```

Voyez-vous pourquoi l'interprète ne fonctionne pas avec des programmes non normalisés? On pourra essayer avec `member` ou `fact`.

Pour finir, écrire un prédicat `normalise(Tete,TeteNorm,Unifications)` qui, à partir d'une tête de clause *i.e.* un appel de prédicat, renvoie une version normalisée de cette tête de clause (la nouvelle tête de clause et la liste des unifications à réinsérer théoriquement dans le corps de clause).

On peut décrire simplement les têtes de clauses normalisées, comme étant des appels de prédicats où tous les termes instanciés sont remplacés par des variables, et les variables identiques sont rendues uniques (les unifications ainsi perdues devant alors être réinjectées dans le corps de clause).

```

| ?- normalise(fact(0,1), Fnorm, Unif).
Unif = [_A=0,_B=1],
Fnorm = fact(_A,_B)

```