Definição do mecanismo de execução do Prolog

Definição

O mecanismo de execução do Prolog é obtido a partir do interpretador abstracto

- escolhendo o goal mais à esquerda em vez de um arbitrário
- substituindo a escolha não determinística de uma cláusula por uma pesquisa sequencial de uma cláusula unificável e por backtracking.

Familia Bíblica: código

Código

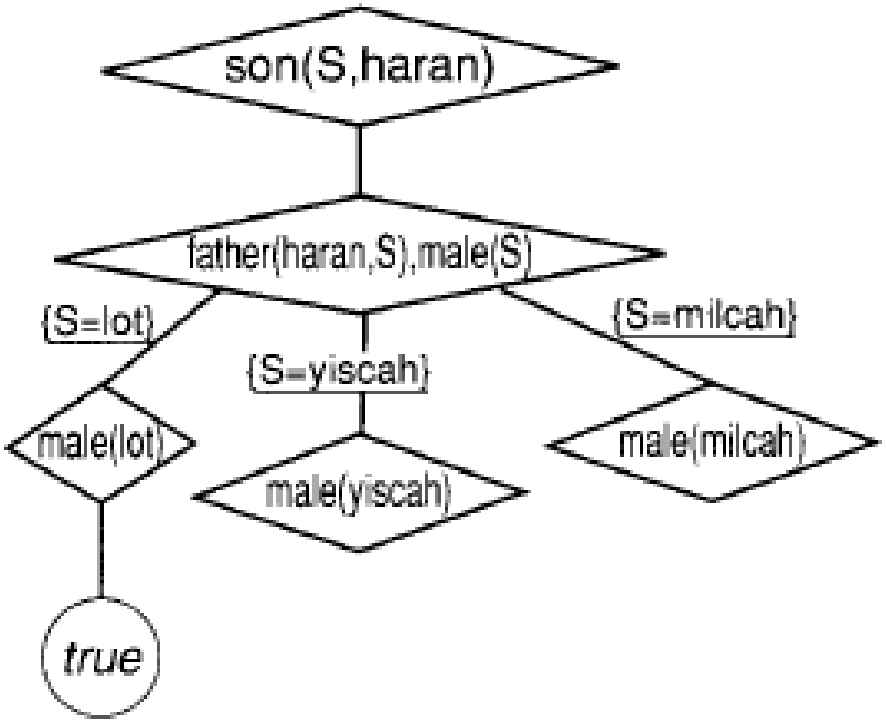
```
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).
father(abraham,isaac).

male(isaac).
male(lot).

female(yiscah).
female(milcah).

son(X,Y) :-
        father(Y,X),
        male(X).
```

Família Bíblica: árvore de pesquisa



Familia Bíblica: trace

```
son(X,haran)?
    father(haran,X)                          X=lot
    male(lot)
            true
        Output:  X=lot
            ;
    father(haran,X)                          X=milcah
    male(milcah)        f
    father(haran,X)                          X=yiscah
    male(yiscah)        f
                  no (more) solutions
```

Append: trace

```
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
append([ ],Ys,Ys).
```

```
append(Xs,Ys,[a,b,c])                                  Xs=[a|Xs1]
    append(Xs1,Ys,[b,c])                               Xs1=[b|Xs2]
        append(Xs2,Ys,[c])                             Xs2=[c|Xs3]
            append(Xs3,Ys,[ ])                         Xs3=[ ],Ys=[ ]
                    true
                Output: (Xs=[a,b,c],Ys=[ ])
                                ;
        append(Xs2,Ys,[c])                             Xs2=[ ],Ys=[c]
                true
            Output: (Xs=[a,b],Ys=[c] )
                                ;
    append(Xs1,Ys,[b,c])                               Xs1=[ ],Ys=[b,c]
            true
        Output: (Xs=[a],Ys=[b,c])
                                ;
append(Xs,Ys,[a,b,c])                                  Xs=[ ],Ys=[a,b,c]
        true
    Output: (Xs=[ ],Ys=[a,b,c])
                                ;
                    no (more) solutions
```

Código

```
parent(terach,abraham).
parent(isaac,jacob).
parent(abraham,isaac).
parent(jacob,benjamin).

ancestor(X,Y) :-
        parent(X,Y).
ancestor(X,Y) :-
        parent(X,Z),
        ancestor(Z,Y).
```

- Quais as soluções (e respectiva, ordem) da query `ancestor(terach,X)`?
  - `X=abraham; X=isaac; X=jacob; X=benjamin`
- Se alterarmos a ordem das cláusulas para `ancestor/2`, o que muda na alínea anterior?

Concatenação

Código

```
/* concatena(Xs,Ys,XsYs)
   XsYs e o resultado de concatenar
   as listas Xs e Ys */

concatena([],Ys,Ys).

concatena([X|Xs],Ys,[X|Zs]) :-
   concatena(Xs,Ys,Zs).
```

- Quais as soluções da query `concatena(Xs,[c,d],Ys)`?
  - `Xs = [], Ys = [c, d]; Xs = [_G263], Ys = [_G263, c, d] ;...`
- Se alterarmos a ordem das cláusulas, o que muda na alínea anterior?

Problema da terminação

- Prolog atravessa as árvores de pesquisa utilizando “depth-first”.
- Pode não encontrar a solução para um goal mesmo que exista uma computação finita.
- O Prolog não termina devido às regras recursivas.

Regras recursivas como causa de não terminação

Exemplo

Considerando o seguinte programa:

```
married(X,Y) :- married(Y,X).
married(abraham,sarah).
```

Qual a resposta a `married(abraham,sarah)`?
Ciclo infinito.
Como resolver?

```
are_married(X,Y) :- married(X, Y).
are_married(X,Y) :- married(Y, X).

married(abraham,sarah).
```

- A “recursão à esquerda” pode causar ciclos infinitos.
- Se possível evitar! Senão, ter os devidos “cuidados”.

Definições circulares como causa de não terminação

Exemplo

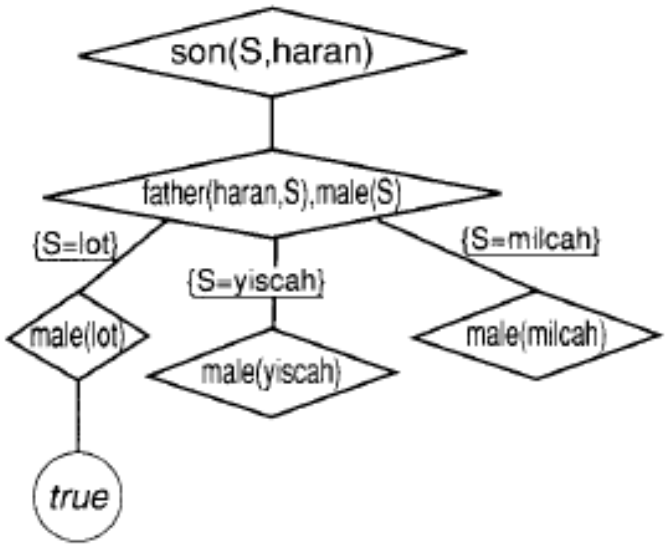
```
parent(X,Y) :-
        child(Y,X).

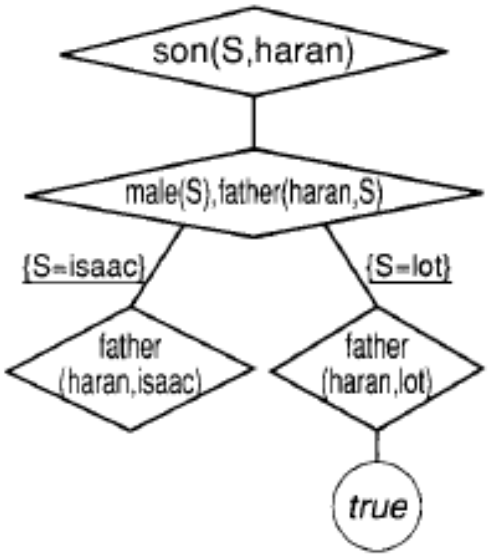
child(X,Y) :-
        parent(Y,X).
```

Ordem dos goals

- A ordem dos goals é a forma principal de especificar o controlo do fluxo (sequencial) nos programas Prolog.
- Diferente ordem dos goals pode levar a diferente ordem de soluções. Por exemplo, no programa da família bíblica, se alterarmos a cláusula de son/2 para `son(X,Y) :- male(X), father(Y,X).` a ordem das soluções à query `son(X,Y)?` vai ser diferente.

A ordem dos goals determina a árvore de pesquisa





A ordem dos goals determina se uma computação termina

Ancestor: recursão à direita

```
ancestor(X,Y) :-
        parent(X,Z),
        ancestor(Z,Y).
```

Ancestor: recursão à esquerda

```
ancestor(X,Y) :-
        ancestor(Z,Y),
        parent(X,Z).
```

Partition: teste primeiro

```
partition([X|Xs],Y,[X|Ls],Bs) :-
          X =< Y,
          partition(Xs,Y,Ls,Bs).
```

Partition: recursão primeiro

```
partition([X|Xs],Y,[X|Ls],Bs) :-
          partition(Xs,Y,Ls,Bs),
          X =< Y.
```

- Devemos colocar os testes aritméticos o quanto antes para falhar o mais cedo possível!

Evitando redundância

Mínimo com redundância

```
minimo(X,Y,X) :- X =< Y.
minimo(X,Y,Y) :- Y =< X.
```

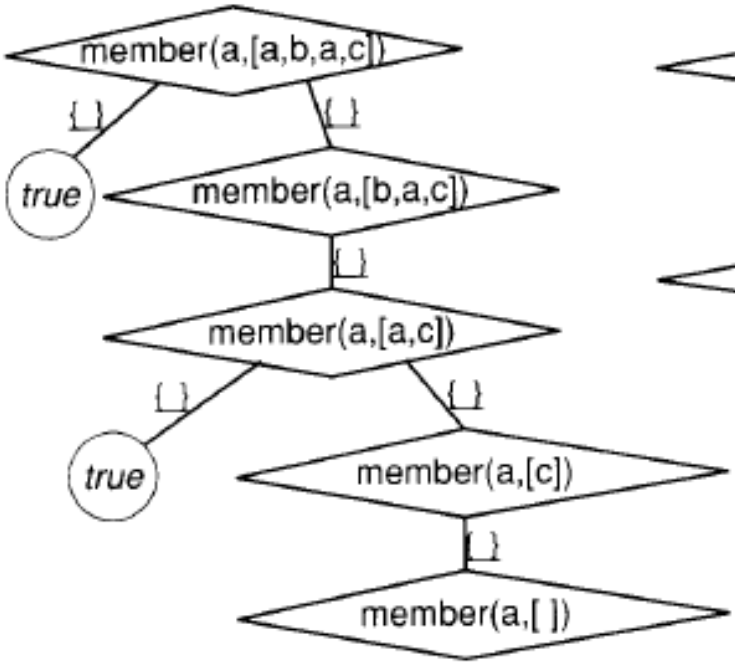
Mínimo sem redundância

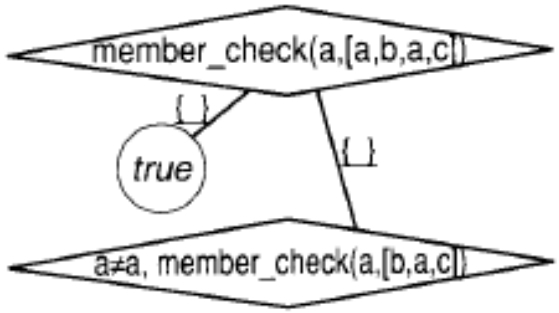
```
minimo(X,Y,X) :- X =< Y.
minimo(X,Y,Y) :- Y < X.
```

Member check

Código

```
/* member_check(X,Xs)
   X is a member of the list Xs
   */
member_check(X,[X|_]).
member_check(X,[Y|Ys]) :-
        X \= Y,
        member_check(X,Ys).
```





Não membro

Código

```
/* nonmember(X,Xs)
   X is not a member of the list Xs.
*/

nonmember(X,[Y|Ys]) :-
          X \= Y,
          nonmember(X,Ys).
nonmember(_,[]).
```

Novamente a ordem dos goals interessa. Porquê?

Tradução (*naive*)

Código

```
/* translate(Words,Mots) :-
    Mots is a list of French words that is the
   translation of the list of English words Words.
*/

translate([Word|Words],[Mot|Mots]) :-
        dict(Word,Mot),
        translate(Words,Mots).

translate([],[]).

dict(the,le).                    dict(dog,chien).
dict(chases,chasse).             dict(cat,chat).
```

Programa típico que efectua um *mapping*.

Remoção de duplicados

- Considere que se pretende definir um predicado `no_doubles(Xs,Ys)` em que a lista Ys é obtida removendo os elementos duplicados da lista Xs.
- Como seria definida a regra para `no_doubles([X|Xs],...)`?
- Neste caso podemos pensar de um modo diferente, em vez de determinar se um elemento já apareceu no output podemos determinar se ele irá aparecer.

Código

```
no_doubles([X|Xs],Ys) :-
        member(X,Xs),
        no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-
        nonmember(X,Xs),
        no_doubles(Xs,Ys).
no_doubles([],[]).
```

Construção de estruturas “bottom-up”

```
reverse([a,b,c],Xs)
  reverse([a,b,c],[ ],Xs)
    reverse([b,c],[a],Xs)
      reverse([c],[b,a],Xs)
        reverse([ ],[c,b,a],Xs)        Xs=[c,b,a]
          true
```

Figure 7.3 Tracing a `reverse` computation

Predicados do sistema para aritmética

- No Prolog os predicados de sistema para aritmética permitem aceder às capacidades aritméticas do computador de um modo directo.
- O predicado `is/2` (`is(Value,Expression)`):
  - é utilizado avaliação de expressões aritméticas
  - normalmente escrito na notação infixa, i.e. `Value is Expression`.
- Existem outros operadores (`>, <, ...`) e podemos inclusivamente definir os nossos utilizando o predicado `op/3`.
  - Na query `(A < B)?`, `A` e `B` são avaliadas como expressões aritméticas e depois o seu resultado é comparado.

Predicado `is/2`

- Um query da forma `Value is Expression` é intrepretada do seguinte modo: a expressão aritmética `Expression` é avaliada e o resultado é unificado com `Value`.
- Exemplos
  - `(X is 3+5)?` tem a solução `X=8`.
  - A query `(8 is 3+5)?` sucede
  - a query `(3+5) is (3+5) ?` falha na unificação
  - a query `(X is 3+a)?` tem um erro pois `3+a` não pode ser avaliado.
  - a query `(X is 3+Y)?` pode ter um erro.
- O predicado `is/2` não é a atribuição de outras linguagens.
  - `(N is N+1)?` falha sempre.

Máximo divisor comum

Código

```
/* greatest_common_divisor(X,Y,Z) :-
   Z is the greatest common divisor
   of the integers X and Y.
*/
greatest_common_divisor(I,0,I).
greatest_common_divisor(I,J,Gcd) :-
        J > 0,
        R is I mod J,
        greatest_common_divisor(J,R,Gcd).
```

Factorial (recursivo)

Código

```
/*
  factorial(N,F) :-
  F is the integer N factorial.
*/

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

Programa totalmente correcto

Definição

Um programa Prolog está *totalmente correcto* sobre um domínio `D` de goals se para todos os goals em `D` a computação:

- termina;
- não causa nenhum erro em tempo de execução;
- tem o significado correcto.

Exemplo

O programa do factorial acima está totalmente correcto sobre o domínio dos goals em que o primeiro argumento é um inteiro.

Factorial (iterativo)

Código

```
/*
   factorial(N,F) :- F is the integer
   N factorial.
*/
factorial(N,F) :- factorial(N,1,F).

factorial(0,F,F).

factorial(N,T,F) :-
    N > 0,
    T1 is T*N,
    N1 is N-1,
    factorial(N1,T1,F).
```

Não é *exactamente* iterativo mas pode ser quase tão eficiente (*Tail recursion optimization*).

Soma dos elementos de uma lista (recursivo)

Código

```
/*
    sumlist(Ls,Sum) :- Sum is the sum of
    the list of integers Ls.
*/

sumlist([],0).

sumlist([I|Ls],Sum) :-
  sumlist(Ls,LsSum),
  Sum is I+LsSum.
```

Soma dos elementos de uma lista (iterativo)

Código

```
/*
    sumlist(Ls,Sum) :- Sum is the sum of
    the list of integers Ls.
*/
sumlist(Ls,Sum) :-
  sumlist(Ls,0,Sum).

sumlist([],Sum,Sum).

sumlist([I|Ls],Temp,Sum) :-
  Temp1 is Temp+I,
  sumlist(Ls,Temp1,Sum).
```

Máximo dos elementos de uma lista (iterativo)

Código

```
/*
    maxlist(Xs,N) :- N is the maximum of the
   list of integers Xs.
*/
maxlist([X|Xs],M) :- maxlist(Xs,X,M).

maxlist([],M,M).
maxlist([X|Xs],Y,M) :-
   maximum(X,Y,Y1),
    maxlist(Xs,Y1,M).


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

Tamanho de uma lista

Código

```
% length(?Xs, +N) :- Xs is a list of length N.

length1([],0).
length1([X|Xs],N) :-
  N > 0, N1 is N-1, length1(Xs,N1).
```

Código

```
/* length(?Xs,?N) :- N is the length of
the list Xs. */

length2([],0).
length2([X|Xs],N) :-
  length2(Xs,N1), N is N1+1.
```

Predicados de tipo básicos

Predicado	True	False
integer(X)	integer(3).	integer(3.0).
float(X)	float(3.0).	float(3).
atom(X)	atom(a).	atom(3).
compound(X)	compound(p(a)).	compound(a).

Outros predicados de tipos

```
number(X) :- //...

atomic(X) :- // atom, integer, float e ...
```

“Aplanar” listas (versão 2xrecursiva)

```
flatten([[a],[b,[c,d]],e],[a, b, c, d, e]).
```

Código

```
/* flatten(Xs,Ys) :- Ys is a list of
   the elements of Xs.
*/

flatten([X|Xs],Ys) :-
  flatten(X,Ys1),
  flatten(Xs,Ys2),
  append(Ys1,Ys2,Ys).

flatten(X,[X]) :-
  atomic(X), // nao e lista
  X \== [].

flatten([],[]).
```

“Aplanar” listas (versão com stack)

Código

```
flatten(Xs,Ys) :- flatten(Xs,[],Ys).

flatten([X|Xs],S,Ys) :-
  list(X),
  flatten(X,[Xs|S],Ys).

flatten([X|Xs],S,[X|Ys]) :-
  atomic(X),
  X \== [],
  flatten(Xs,S,Ys).

flatten([],[X|S],Ys) :-
  flatten(X,S,Ys).

flatten([],[],[]).

list([X|Xs]).
```

Predicado `functor/3`

Definição

Predicado `functor/3` O predicado `functor(Termo,F,Aridade)` é verdadeiro se `Termo` é um termo de aridade `Aridade` cujo principal functor é `F`.

Exemplo

- ```
  ?- functor(a,a,0).
  true
  ```
- ```
  ?- functor(father(haran,lot),X,Y).
  X=father, Y=2
  ```
- ```
  functor(X,father,2).
  X = father(_G270, _G271).
  ```
  ```

Predicado `arg/3`

Definição

`arg(N,Termo,Arg)` `Arg` é o `N`-ésimo argumento do termo `Termo`.

Exemplo

- `?- arg(1,father(haran,lot),haran).`
  `true`
- `?- arg(2,father(haran,lot),X).`
  `X=lot`
- `?- arg(1,father(X,lot),haran).`
  `X=haran`

Subtermo: código

Código

```
/* subterm(Sub,Term) :- Sub is a subterm
   of the ground term Term.
*/
subterm(Term,Term).

subterm(Sub,Term) :-
   compound(Term),
   functor(Term,F,N),
   subterm(N,Sub,Term).

subterm(N,Sub,Term) :-
   N > 1,
   N1 is N-1,
   subterm(N1,Sub,Term).

subterm(N,Sub,Term) :-
   arg(N,Term,Arg),
   subterm(Sub,Arg).
```

Predicado =../2

Definição

Predicado =../2 O goal `Termo =.. Lista` sucede se `Lista` é uma lista que cuja cabeça é o nome do functor do termo `Termo` e o corpo é a lista dos argumentos de `Termo`.

Exemplo

- `?- father(haran,lot) =.. [father,haran,lot].`
  `true.`
- `?- X =.. [father,haran,lot].`
  `X = father(haran, lot).`
- `?- father(haran, lot) =.. X.`
  `X = [father, haran, lot].`

Subtermo (=..)

Código

```
subterm(Term,Term).

subterm(Sub,Term) :-
   compound(Term),
   Term =.. [F|Args],
   subterm_list(Sub,Args).

subterm_list(Sub,[Arg|Args]) :-
   subterm(Sub,Arg).

subterm_list(Sub,[Arg|Args]) :-
   subterm_list(Sub,Args).
```

Univ (construir a lista)

Código

```
/* univ(Term, List) :- List is a list
   containing the functor of Term followed
   by the arguments of Term.
*/
univ(Term, [F|Args]) :-
   functor(Term,F,N),
   args(0,N,Term,Args).

args(I,N,Term,[Arg|Args]) :-
   I < N,
   I1 is I+1,
   arg(I1,Term,Arg),
   args(I1,N,Term,Args).

args(N,N,Term,[]).
```

Univ (construir o termo composto)

Código

```
/*  univ(Term, List)  The functor of Term
  is the first element of the list List,
   and its arguments are the rest of List's
  elements. */

univ(Term, [F|Args]) :-
  length(Args,N),
  functor(Term,F,N),
  args(Args,Term,1).

args([Arg|Args],Term,N) :-
  arg(N,Term,Arg),
  N1 is N+1,
  args(Args,Term,N1).

args([],Term,N).
```

Predicado `var/1` e `nonvar/1`

- `var(Termo)` sucede se `Termo` é uma variável e falha caso contrário
  - `?- var(X).` sucede
  - `?- var(a)` e `var([X|Xs]` falham
- `nonvar(Termo)` falha se `Termo` é uma variável e sucede caso contrário

Exemplo (Versão mais eficiente `grandparent/2`)

```
/* grandparent(X,Z) :- X is the grandparent of Z. */
grandparent(X, Z) :-
        nonvar(X),
        parent(X, Y),
        parent(Y, Z).

grandparent(X, Z) :-
        nonvar(Z),
        parent(Y,Z),
        parent(X,Y).
```

Verificar se um termo é *ground*

Exemplo

```
/*  groundT(Term) :- Term is a ground term.
*/
groundT(Term) :-
   nonvar(Term),
   atomic(Term).

groundT(Term) :-
   nonvar(Term),
   compound(Term),
   functor(Term,F,N),
   groundT(N,Term).

groundT(N,Term) :-
   N > 0,
   arg(N,Term,Arg),
   groundT(Arg),
   N1 is N-1,
   groundT(N1,Term).

groundT(0,Term).
```

Comparando termos non ground

Definição (Predicado `==/2`)

A query `?- X == Y.` sucede se `X` e `Y` são

- "constantes" (atomic) idênticas
- variáveis idênticas
- estruturas cujo principal functor tem o mesmo nome e aridade e que `Xi == Yi` sucede recursivamente para todos os argumentos `Xi` e `Yi` correspondentes de `X` e `Y`

Unify with occurs check

- Na unificação normal, `?- A = f(A).` sucede.
- No entanto *unify with occurs check* do goal acima não sucede, isto é, uma variável só unifica com um termo se este termo não contiver tal variável.

Exemplo

```
/* unify(Term1,Term2) :- Term1 and Term2
   are unified with the occurs check.
*/
unify(X,Y) :- var(X), var(Y), X=Y.

unify(X,Y) :-
   var(X), nonvar(Y), not_occurs_in(X,Y), X=Y.

unify(X,Y) :-
   var(Y), nonvar(X), not_occurs_in(Y,X), Y=X.
```

Exemplo

```
unify(X,Y) :-
  nonvar(X), nonvar(Y), atomic(X), atomic(Y), X=Y.

unify(X,Y) :- nonvar(X), nonvar(Y), compound(X),
  compound(Y), term_unify(X,Y).

not_occurs_in(X,Y) :- var(Y), X \== Y.

not_occurs_in(X,Y) :- nonvar(Y), atomic(Y).

not_occurs_in(X,Y) :- nonvar(Y), compound(Y),
  functor(Y,F,N), not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :-
  N > 0, arg(N,Y,Arg), not_occurs_in(X,Arg),
  N1 is N-1, not_occurs_in(N1,X,Y).

not_occurs_in(0,X,Y).

term_unify(X,Y) :-
  functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).
```

Exemplo

```
unify_args(N,X,Y) :-
  N > 0, unify_arg(N,X,Y), N1 is N-1, unify_args(N1,X,Y).
unify_args(0,X,Y).

unify_arg(N,X,Y) :-
  arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
```

Meta-variáveis

Exemplo

```
/*
   ou(X,Y) <- X ou Y
*/

ou(X,Y) :-
   call(X).

ou(X,Y) :-
   call(Y).
```