
Projet Prolog 4IF

Hexanôme 4104 - Equipe Dragibus

5 novembre 2013

1 Généalogie

On créera une base de connaissances en spécifiant les relations "parent" directes. `parent(X, Y)` étant vrai si X est le parent de Y.

```
1 parent(john, martha).
2 parent(john, marc).
3 parent(rodriigo, john).
4 parent(jean, leo).
5 parent(louis, jean).
6 parent(louis, george).
7 parent(rodriigo, louis).
8 parent(ivoronne, rodriigo).
```

Définition du lien grand-parent par une propagation de la relation parent sur 2 niveaux. `grandParent(X, Y)` est vrai si X est le grand parent de Y.

```
1 grandParent(X, Y) :-
2     parent(X, Z),
3     parent(Z, Y).
4
5 % Test de la relation grand-parent
6 testGrandParent :-
7     grandParent(rodriigo, martha),
8     grandParent(rodriigo, marc),
9     grandParent(rodriigo, jean),
10    grandParent(ivoronne, john),
11    grandParent(rodriigo, marc),
12    \+ grandParent(jean, george),
13    \+ grandParent(ivoronne, martha),
14    \+ grandParent(john, martha),
15    \+ grandParent(martha, martha),
16    grandParent(louis, leo),
17    !.
```

Définition du lien ancêtre par une propagation récursive sur plusieurs niveaux, jusqu'à satisfaction. `ancetre(X, Y)` est vrai si X est un ancetre de Y.

```
1 ancetre(X, Y) :-
2     parent(X, Y).
3 ancetre(X, Y) :-
4     parent(X, Z),
5     ancetre(Z, Y).
6
7 % Test de la relation ancetre
8 testAncetre :-
9     ancetre(ivoronne, martha),
10    ancetre(rodriigo, marc),
11    ancetre(john, martha),
12    \+ ancetre(martha, martha).
```

Définition de la liste des ancêtres à partir d'un élément de la famille. On utilise `setof` car cette partie n'est pas sur les listes, et son utilisation est pratique pour éliminer les doublons dans les candidats à la relation ancêtre. On définit de la même façon pour récupérer la liste des descendants.

`ancetres(X, L)` est vrai si L est l'ensemble des ancetres de X.

`descendants(X, L)` est vrai si L est l'ensemble des descendants de X.

Dans ces deux prédicats, X ne peut pas être déterminée à partir de L, et donc ne peut pas être une

variable.

```
1 ancetres(X, L) :-  
2   setof(A, ancetre(A, X), L).  
3  
4 descendants(X, L) :-  
5   setof(A, ancetre(X, A), L).  
6  
7 ancetres(X, _) :- var(X), fail.  
8 descendants(X, _) :- var(X), fail.
```

Définition de la relation frère-soeur complète en partant du constat que celle-ci correspond au partage des parents. Cette relation n'est pas réflexive, une personne ne pouvant pas être son propre frère-soeur. `frereSoeur(X, Y)` est vrai si X et Y ont le meme parent (donc sont frère-soeur).

```
1 frereSoeur(X, Y) :-  
2   parent(Z, X),  
3   parent(Z, Y),  
4   X \= Y.  
5  
6 % Test de la relation frere-soeur  
7 testFrereSoeur :-  
8   frereSoeur(martha, marc),  
9   frereSoeur(marc, martha),  
10  \+ frereSoeur(marc, marc),  
11  \+ frereSoeur(rodriigo, marc),  
12  \+ frereSoeur(rodriigo, jean).
```

Définition de la relation oncle-tante par une relation frère-soeur pour un parent. `oncleTante(X, Y)` est vrai si X est l'oncle (ou la tante) de Y.

```
1 oncleTante(X, Y) :-  
2   frereSoeur(X, Z),  
3   parent(Z, Y).  
4  
5 % Test de la relation oncle-tante  
6 testOncleTante :-  
7   oncleTante(louis, marc),  
8   oncleTante(louis, martha),  
9   \+ oncleTante(rodriigo, martha),  
10  \+ oncleTante(marc, martha).
```

Définition de la relation cousin par une relation frère-sœur entre deux parents. `cousin(X, Y)` est vrai si X et Y sont cousin (càd que leur parent sont freres).

```
1 cousin(X, Y) :-  
2   parent(A, X),  
3   parent(B, Y),  
4   frereSoeur(A, B).  
5  
6 % Test de la relation cousin  
7 testCousin :-  
8   cousin(jean, martha),  
9   cousin(marc, jean),  
10  \+ cousin(marc, martha),  
11  \+ cousin(rodriigo, martha).
```

2 Listes

`element(X, L, LprivX)` est vrai si `L/X` est une liste composé des éléments de `L` privé une fois de `X`. L'ordre doit être préservé. Si `X` n'est pas dans `L`, alors le prédicat est faux. Les trois paramètres peuvent être des variables.

```
1 element(X, [X|Xs], Xs).
2 element(X, [T|As], [T|Bs]) :-
3     element(X, As, Bs),
4     X \== T.
5
6 testElement :-
7     \+ element(_, [], []),
8     element(b, [a, b, c], [a, c]),
9     element(a, [a, b, c], [b, c]),
10    element(c, [a, b, c], [a, b]),
11    \+ element(d, [a, b, c], _),
12    \+ element(a, [a, b], [a, b]),
13    \+ element(a, [c, b], [c, b]),
14    \+ element(b, [a, b, c], [a, b]),
15    % Il faut rajouter une coupure pour prouver les predicats
16    % suivants pour limiter l'espace des solutions :
17    % element(X, [X | L], L),
18    % element(X, [A, X | L], [A | L]),
19    % ...
20    true.
```

`extract(L1, L2)` est vrai si `L2` est un sous ensemble de `L1`. Les deux paramètres peuvent être des variables.

```
1 extract(_, []).
2 extract(L, [X|Xs]) :-
3     element(X, L, L2),
4     extract(L2, Xs).
5
6 testExtract :-
7     extract([], []),
8     extract([a], []),
9     extract([a], [a]),
10    extract([a, b], []),
11    extract([a, b], [a]),
12    extract([a, b], [b]),
13    extract([a, b], [b, a]),
14    extract([a, b, c], []),
15    extract([a, b, c], [b]),
16    extract([a, b, c], [c]),
17    extract([a, b, c], [a, b]),
18    extract([a, b, c], [a, c]),
19    extract([a, b, c], [b, a]),
20    extract([a, b, c], [a, b, c]),
21    extract([a, b, c], [a, c, b]),
22    extract([a, b, c], [b, a, c]),
23    \+ extract([], [_]),
24    \+ extract([a], [b]),
25    \+ extract([a, b, c], [a, b, d]),
26    % Toutes les permutations de L1 sont des sous ensembles de L2 :
27    permutation(L1, L2),
28    extract(L1, L2),
29    write(L1), write(' '), write(L2), write('\n').
```

`concat(L1, L2, L1nL2)` est vrai si `L1nL2` est la concaténation de `L1` et `L2`.

```
1 concat([], L2, L2).
2 concat([X|Xs], L2, [X|L]) :-
3   concat(Xs, L2, L).
4
5 testConcat :-
6   % On utilise append/3 de la lib standard pour tester
7   append(L1, L2, L1nL2),
8   concat(L1, L2, L1nL2),
9   write(L1), write(' + '), write(L2), write(' = '), write(L1nL2), write('\n').
```

`inv(L1, L2)` est vrai si `L1` est l'inverse de `L2`. On remarque que si `L1` n'est pas une variable et que `L2` en est une, le prédicat ne se termine pas.

```
1 inv([], []).
2 inv([X|Xs], L) :-
3   inv(Xs, L2),
4   concat(L2, [X], L).
5
6 testInv :-
7   % On utilise reverse/2 de la lib standard pour tester
8   reverse(L1, L2),
9   inv(L1, L2),
10  write(L1), write(' : '), write(L2), write('\n').
```

`subsAll(E, X, L1, L2)` est vrai si `L2` est la liste `L1` avec les éléments `E` remplacés par des `X`. Dans le cas où `E` est une variable, il prend la première valeur de `L1`.

```
1 subsAll(_, _, [], []).
2 subsAll(E, X, [T|Q], L) :-
3   T \= E,
4   subsAll(E, X, Q, Ls),
5   concat([T], Ls, L).
6 subsAll(E, X, [E|Q], L) :-
7   subsAll(E, X, Q, Ls),
8   concat([X], Ls, L).
9
10 testSubsAll :-
11   % On utilise select/4 de la lib standard.
12   select(X, L1, Y, L2),
13   subsAll(X, Y, L1, L2),
14   write(X), write(' => '), write(Y), write(' : '),
15   write(L1), write(' => '), write(L2), write('\n').
```

3 Arithmétique

`element(Idx, X, L)` est vrai si `X` est à la position `Idx` dans `L` (en commençant à compter à partir de 1). On doit passer par une fonction `element/4` pour intégrer un accumulateur en paramètre.

```
1 element(Idx, X, L) :-
2   element(Idx, 0, X, L).
3 element(Idx, _, _, _) :-
4   Idx < 1, !, fail.
5 element(Idx, Count, H, [H|_]) :-
6   Idx is Count + 1.
7 element(Idx, Count, Item, [_|T]) :-
8   Count1 is Count + 1,
9   element(Idx, Count1, Item, T).
10
11 testElement :-
12   \+ element(0, _, _),
13   element(1, X, [X]),
14   element(1, X, [X, _, _, _]),
15   \+ (X \= Y, element(2, X, [_ , Y, _, _])),
16   element(1, a, [a, b, a]),
17   \+ element(2, a, [a, b, a]),
18   element(3, a, [a, b, a]),
19   % Identique a nth1/3. On l'utilise alors pour prouver le code.
20   nth1(Idx, L, E),
21   element(Idx, E, L).
```

4 Ensembles

`list2ens(L, E)` est vrai quand `E` correspond à l'ensemble des éléments de `L`, sans doublon. L'ordre de la liste initiale est conservé. On doit alors définir un prédicat `list2ens/3` pour utiliser un accumulateur en paramètre. Également, on définit simplement un prédicat `element/2` qui est vrai lorsqu'un élément `X` est dans une liste `L`.

```
1 element(_, []) :- fail.
2 element(X, [_|_]).
3 element(X, [_|Q]) :- element(X, Q).
4
5 list2ens([], V, V).
6 list2ens([T|Q], V, E) :-
7     element(T, V),
8     list2ens(Q, V, E).
9 list2ens([T|Q], V, E) :-
10    \+element(T, V),
11    concat([T], V, V1),
12    list2ens(Q, V1, E).
13
14 list2ens(L, E) :-
15     list2ens(L, [], E1),
16     inv(E1, E),
17     !.
18
19 % Test de la conversion d'une liste en ensemble
20 testList2Ens :-
21     list2ens([], []),
22     list2ens([a, a], [a]),
23     list2ens([a, b, a], [a, b]),
24     \+ list2ens([a, b], [a]),
25     \+ list2ens([a, b, c], [a, b]).
```

`ensemble(L)` est vrai quand `L` est un ensemble, et donc ne contient pas de doublons. On réutilise le prédicat précédemment fait `list2ens`.

```
1 ensemble(L) :-
2     list2ens(L, L).
3
4 % Test de la verification d'un ensemble
5 testEnsemble :-
6     ensemble([]),
7     ensemble([a]),
8     ensemble([a, b]),
9     ensemble([a, b, c]),
10    \+ ensemble([a, a]),
11    \+ ensemble([a, b, a]),
12    \+ ensemble([a, b, b]),
13    \+ ensemble([c, c, b]).
```