Prolog Project

Simon Munch
simun15@student.sdu.dk
DM552

November 15, 2016

Contents

1 Methodology

As one of ProLogs most powerfull mechanism is unification, this is the tool i was going to use most in this project. Other mechanisms that i use is negation as failure \+ . I also use cuts ! ,although i tried to keep theese to a minimum as I was warned they could ruin backtracking.

Besides the primary predicates asked for me to define in the project, I also needed some auxiliary predicates to ensure that i would be able to handle particular scenarios so i could get the solutions i wanted. Most importantly, i would need lists and some functions that would allow me to manipulate them. Luckily, Swi-Prolog comes with some functions by default such as sort/2 and findall/3, which allows me to sort a list and remove duplicates, and produce a list of all solutions to a function. I still had to make some myself to find certain properties of lists, such as no_match/2 which fails if an element in one list is also present in the other, using nonmember/2, which fails if a element is present in a list.

2 Implementation

Most of my solutions will be dependent on satisfies/2, so it was important that I got this one correct. I noticed that i had a tendency to repeat answers, and i eliminated those using cuts. I was a bit hesitant as i know that cuts might ruin backtracking, so I later returned to it, and tested what find_val_tt/2 would output with and without cuts in satisfies/2.

As the cuts worked as i wanted them to (removing duplicates), i had them in my code as follows: that whenever a function had two options to turn out true, i had a cut, as that way i wouldnt get to the same solution from two different directions. As an exmaple, my or(X,Y) looks like this with cuts:

```
satisfies(V,or(X,_)) :- satisfies(V,X), !.
satisfies(V,or(_,Y)) :- satisfies(V,Y), !.
```

In which case if a solution can be found which satisfies the first clause, only that one is returned. This is helpfull, as it would otherwise also look into the other clause, and perhaps finding that the second clause would also satisfy the same solution, thus creating repeat answers. cuts avoid this.

A notable development in my implementation is also that when I had to define taut_tab/1 , I got a hint that setting neg in front of a tautology would make it mimick a unsatisfiable function, and thus i could reuse unsat_tab/1

to find tautologies. I first learned of this trick after i had already made sat_tt/1, which is why the two different ways i find tautologies, satisfiably and unsatisfiably is different in my two implimentations.

3 Testing

3.1 find_val_tt/2 & satisfies/2

As i was very afraid that cuts would make my code run in ways i didn,t want it to, i wanted to test satisfies/2 with and without cuts. As such, i made a formula which would produce a good amount of solutions, and then compare what find_val_tt/2 returned with and without the cuts in satisfies/2.

```
?- find_val_tt(or(or(neg(p(1)),p(1)),and(or(p(3),p(4)),p(2))),V).
without cuts:
V = [1, 2, 3, 4] ;
V = [1, 2, 3, 4] ;
V = [1, 2, 3, 4] ;
V = [1, 2, 3] ;
V = [1, 2, 3] ;
V = [1, 2, 4] ;
V = [1, 2, 4] ;
V = [1, 2] ;
V = [1, 3, 4] ;
V = [1, 3] ;
V = [1, 4] ;
V = [1] ;
V = [2, 3, 4] ;
V = [2, 3, 4] ;
V = [2, 3, 4] ;
V = [2, 3] ;
V = [2, 3] ;
V = [2, 4] ;
V = [2, 4] ;
V = [2] ;
V = [3, 4] ;
V = [3] ;
V = [4] ;
V = [] ;

with cuts:
V = [1, 2, 3, 4] ;
V = [1, 2, 3] ;
V = [1, 2, 4] ;
```

```
V = [1, 2] ;
V = [1, 3, 4] ;
V = [1, 3] ;
V = [1, 4] ;
V = [1] ;
V = [2, 3, 4] ;
V = [2, 3] ;
V = [2, 4] ;
V = [2] ;
V = [3, 4] ;
V = [3] ;
V = [4] ;
V = [].
```

As the only solutons removed were duplicates, i could feel assured that the function behaved as i wanted it to.

3.2 tableau/2

I wanted to see if tableau/2 would give the same results on impl(equiv(p(1),p(2)),and(p(3),or(p(1),neg(p(3))))) as the project description.

```
?- tableau(impl(equiv(p(1),p(2)),and(p(3),or(p(1),neg(p(3))))),V).
V = [p(1), neg(p(2))] ;
V = [neg(p(1)), p(2)] ;
V = [p(3), p(1)] ;
V = [p(3), neg(p(3))] ;
```

As this is the same output as on the picture in the project description, i am satisfied.

4 Code

```
/** task 1.1
* Looking through the formula in an attempt to reach any p(N).
*/
wff(p(N)) :- integer(N).
wff(neg(X)) :- wff(X).
wff(impl(X,Y)) :- wff(X) , wff(Y).
wff(and(X,Y)) :- wff(X) , wff(Y).
wff(or(X,Y)) :- wff(X) , wff(Y).
wff(equiv(X,Y)) :- wff(X) , wff(Y).
wff(xor(X,Y)) :- wff(X) , wff(Y).

/** task 1.2
* Writing the rules for propositional symbols, with the
* appropiate cuts and cases. Can have strange behaviour on
* empty list and backtracking due to a combination of cuts
* and negations. */
satisfies(V,p(N)) :- member(V,N).
satisfies(V,neg(X)) :- \+satisfies(V,X).
satisfies(V,impl(X,Y)) :- satisfies(V,X) , satisfies(V,Y), !.
satisfies(V,impl(X,_)) :- \+satisfies(V,X), !.
satisfies(V,and(X,Y)) :- satisfies(V,X) , satisfies(V,Y).
satisfies(V,or(X,_)) :- satisfies(V,X), !.
satisfies(V,or(_,Y)) :- satisfies(V,Y), !.
satisfies(V,equiv(X,Y)) :- satisfies(V,X) , satisfies(V,Y) , !.
satisfies(V,equiv(X,Y)) :- \+satisfies(V,X) , \+satisfies(V,Y) , !.
satisfies(V,xor(X,Y)) :- satisfies(V,X) , \+satisfies(V,Y) , !.
satisfies(V,xor(X,Y)) :- \+satisfies(V,X) , satisfies(V,Y) , !.

/** task 1.3
* A list is made containing all the N's from p(N) in a formula.
* Duplicates are removed all possible subset is found.
* Unification is then made between the subsets and satisfies(V,F)
* to only give back relevant ones. Relevant valuations is dependent
* on the amount of p(N) in the formula. no need to have 6 in V
* if no p(6) exists. */
find_val_tt(F,V) :- get_v_list(F,Vlist) , sort(Vlist,Vclean) ,
                    subset(Vclean,V) ,   satisfies(V,F).

/* give a list of all N's in p(N) from F unsorted and with duplicates */
get_v_list(p(X),V) :- append([],X,V).
get_v_list(neg(X),V) :- get_v_list(X,V).
```

```
get_v_list(impl(X,Y),V) :- get_v_list(X,V1), get_v_list(Y,V2),
                                        concatenate(V1,V2,V).
get_v_list(and(X,Y),V) :- get_v_list(X,V1), get_v_list(Y,V2),
                                        concatenate(V1,V2,V).
get_v_list(or(X,Y),V) :- get_v_list(X,V1), get_v_list(Y,V2),
                                        concatenate(V1,V2,V).
get_v_list(equiv(X,Y),V) :- get_v_list(X,V1), get_v_list(Y,V2),
                                        concatenate(V1,V2,V).
get_v_list(xor(X,Y),V) :- get_v_list(X,V1), get_v_list(Y,V2),
                                        concatenate(V1,V2,V).

/** task 1.4
* If no solution from find_val_tt(F,V), it must be unsatisfiable.
* If not unsatisfiable, but instance of failure can be found,
* it bust be satisfiable, but not a tautology
* If not unsatisfiable nor satisfiable, it must be a tautology. */
unsat_tt(F) :- \+find_val_tt(F,_).
sat_tt(F) :- \+unsat_tt(F) , get_v_list(F,Vlist) , sort(Vlist,Vclean) ,
                    subset(Vclean,V) , findall(X,find_val_tt(F,X),Z) ,
                                                nonmember(V,Z).
taut_tt(F) :- \+unsat_tt(F) , \+sat_tt(F).

/** task 2.5
* Writing in the rules for tableaux with the appropiate first cases.
* In the case where a descendant is multiple values, a list is used.
*/
tableau(p(N),p(N)).
tableau(neg(p(N)),neg(p(N))).
tableau(neg(neg(X)),L) :- tableau(X,L).
tableau(impl(X,_),L) :- tableau(neg(X),L).
tableau(impl(_,X),L) :- tableau(X,L).
tableau(neg(impl(X,Y)),V) :- tableau(X,X1) , tableau(neg(Y),Y1) ,
                                                flatten([X1,Y1],V).
tableau(and(X,Y),V) :- tableau(X,X1) , tableau(Y,Y1) ,
                                                flatten([X1,Y1],V).
tableau(neg(and(X,_Y)),X1) :- tableau(neg(X),X1).
tableau(neg(and(_X,Y)),Y1) :- tableau(neg(Y),Y1).
tableau(or(X,_Y),X1) :- tableau(X,X1).
tableau(or(_X,Y),Y1) :- tableau(Y,Y1).
tableau(neg(or(X,Y)),V) :- tableau(neg(X),X1) , tableau(neg(Y),Y1) ,
                                                flatten([X1,Y1],V).
tableau(equiv(X,Y),V) :- tableau(X,X1) , tableau(Y,Y1) ,
                                                flatten([X1,Y1],V).
tableau(equiv(X,Y),V) :- tableau(neg(X),X1) , tableau(neg(Y),Y1) ,
```

```
                                                    flatten([X1,Y1],V).
tableau(neg(equiv(X,Y)),V) :- tableau(X,X1) , tableau(neg(Y),Y1) ,
                                                    flatten([X1,Y1],V).
tableau(neg(equiv(X,Y)),V) :- tableau(neg(X),X1) , tableau(Y,Y1) ,
                                                    flatten([X1,Y1],V).
tableau(xor(X,Y),V) :- tableau(X,X1) , tableau(neg(Y),Y1) ,
                                                    flatten([X1,Y1],V).
tableau(xor(X,Y),V) :- tableau(neg(X),X1) , tableau(Y,Y1) ,
                                                    flatten([X1,Y1],V).
tableau(neg(xor(X,Y)),V)  :- tableau(X,X1) , tableau(Y,Y1) ,
                                                    flatten([X1,Y1],V).
tableau(neg(xor(X,Y)),V)  :- tableau(neg(X),X1) , tableau(neg(Y),Y1) ,
                                                    flatten([X1,Y1],V).

/** task 2.6
* For all the leaves given by tablaeu/2, a list containing the neg N's
* and a list containing the normal N's are made. If a N appears
* in both lists, then the leaf is not valid, otherwise its a solution.
*/
find_val_tab(F,Vclean) :- tableau(F,Leaf) , leaf_to_list(Leaf,Listp),
                                            leaf_to_list_neg(Leaf,Listn) ,
                                            no_match(Listp,Listn) ,
                                            sort(Listp,Vclean).

/* make a leaf into a list with the p(N)'s N value(s) */
leaf_to_list(p(N),V) :- append([],N,V).
leaf_to_list(neg(p(_N)),[]).
leaf_to_list([],[]).
leaf_to_list([p(N)|L],V) :- append([],N,List1) , leaf_to_list(L,List2) ,
                                            concatenate(List1,List2,V).
leaf_to_list([neg(p(_N))|L],V) :- leaf_to_list(L,V).
leaf_to_list_neg(neg(p(N)),V) :- append([],N,V).
leaf_to_list_neg(p(_N),[]).
leaf_to_list_neg([],[]).
leaf_to_list_neg([neg(p(N))|L],V) :- append([],N,List1) ,
                                     leaf_to_list_neg(L,List2) ,
                                     concatenate(List1,List2,V).
leaf_to_list_neg([p(_N)|L],V) :- leaf_to_list_neg(L,V).

/** task 2.7
* Same as task 1.4, except i attempt to find the tautology by
* seeing of putting a neg/1 in front of it makes find_val_tab/2 fail.
*/
taut_tab(F) :- unsat_tab(neg(F)).
```

```
unsat_tab(F) :- \+find_val_tab(F,_).
sat_tab(F) :- \+taut_tab(F) , \+unsat_tab(F).

/**lists , some functions defined by default in swi-prolog.*/
/* length already defined by default */
/* sort defined by default */
/* findall defined by default */
append([],Y,[Y]).
append([X|L],Y,[X|W]) :- append(L,Y,W).
member([X|_L],X).
member([_X|L],Y) :- member(L,Y).
remove([],_Y,[]).
remove([X|L],X,L).
remove([_|L],Y,W) :- remove(L,Y,W).
concatenate([],L,L).
concatenate([X|L],W,[X|Z]) :- concatenate(L,W,Z).
subset([], []).
subset([X|L], [X|Lnew]):- subset(L, Lnew).
subset([_|L], Lnew):- subset(L, Lnew).
nonmember(E,[E|_]) :- ! , fail.
nonmember(E,[_|L]) :- ! , nonmember(E,L).
nonmember(_,[]).
flatten(L,F) :- flatten(L,[],F1), !, F = F1.
flatten(X,L,[X|L]) :- var(X), !.
flatten([],L,L) :- !.
flatten([X|L],T,List) :- !, flatten(X,L1,List), flatten(L,T,L1).
flatten(X,L,[X|L]).
no_match([],_).
no_match([X|L], List2) :- nonmember(X, List2) , no_match(L,List2).
```