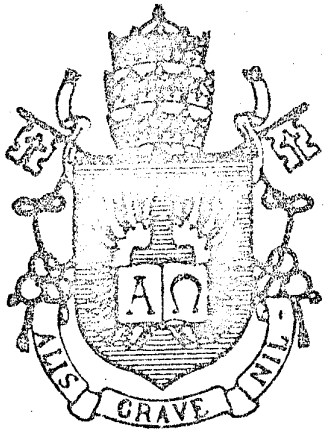

PUC

Series: Monografias em Ciência da Computação

No. 20/89

SOME EXTENSIONS TO WARREN'S PLAN-GENERATION ALGORITHM

Antonio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453

RIO DE JANEIRO - BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciencia da Computação, No 20/89

Editor: Paulo A. S. Veloso October, 1989

SOME EXTENSIONS TO WARREN'S PLAN-GENERATION ALGORITHM *

Antonio L. Furtado

* Research partly sponsored by FINEP

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

ABSTRACT

An extended version of the WARPLAN plan-generation algorithm of D. S. D. Warren is presented. The extensions cover a few cases outside the scope of the original version. An example, adopting the abstract data type paradigm, and the complete listing of the ARITY Prolog implementation are included.

KEYWORDS

Plan-generation, Prolog, databases, integrity constraints, abstract data types.

RESUMO

Uma versão ampliada do algoritmo WARPLAN para geração de planos devido a D. S. D. Warren é apresentada. Nesta versão podem ser tratados alguns casos que estavam além do escopo da versão original. Um exemplo, dentro do paradigma de tipos abstratos de dados, e a listagem completa da implementação em ARITY Prolog estão incluidos.

PALAVRAS-CHAVE

Geração de planos, Prolog, bancos de dados, restrições de integridade, tipos abstratos de dados.

1. A brief presentation of the original WARPLAN

The WARPLAN algorithm was formulated in Prolog by D. S. D. Warren in 1974 [Wa]. In this section we review its main features. The reader is referred to [Wa] or [CCP] for more details.

In general, the plan-generation problem is: given a goal statement, derive a plan to transform the current state of the world into a state where the goal is achieved. In WARPLAN, the state of the world is mainly characterized by a set of facts, expressed as ground (i.e. variable-free) positive literals; the goal is then expressed as a conjunction of facts and the plan as a sequence of applications of operations whose effect is to add and/or delete facts.

A remarkable aspect of WARPLAN is that the facts in the goal conjunction are allowed to be mutually interfering, in the sense that it may not be possible to satisfy them in the order they appear in the conjunction. Having achieved part of the facts in the conjunction, the algorithm proceeds to satisfy the next fact X by adding an appropriate operation-application U by one of the following methods:

a. extension - The new operation-application U is examined. It should preserve P, the part already achieved. Also the preconditions C to apply U should be consistent with P. If these conditions are met, the partial plan developed so far can be recursively expanded to produce a state where both P and C are attained. U is then appended to the plan.

b. insertion - If the above method fails, it is checked whether the last operation-application V in the current partial plan would not delete X. If so, an attempt is made to insert U somewhere before V.

The frame problem [Ni] is handled through the meta-predicate "holds". A fact X is considered to hold if it is "given" in the initial state, denoted by the operation "start"; otherwise X is considered to hold in the state resulting from a sequence S = V1=>V2=>...=>Vn-1=>Vn of operation-applications, where V1 is "start" and Vn is the last operation-application, if either Vn is able to add X or Vn preserves X and X held in the partial plan V1=>V2=>...=>Vn-1, which is checked recursively. This last proviso formalizes the assumption that an operation-application does not modify the world except with respect to the facts that the operation is explicitly declared to be able to add or delete. As a consequence, the combinatorial explosion of rules to say that each other fact remains undisturbed is avoided.

Operations are characterized by their effects, in terms of facts added and/or deleted, and by preconditions, which are logical

expressions including facts that should hold when the operation is applied. As explained for the extension method, the previous achievement of such facts becomes a sub-goal that may cause the plan to be extended backwards so that an intermediate state is reached where the operation can be applied.

Provision is also made for requirements that must remain invariant through all states, by indicating facts that are always true (positive requirements) or expressions that are impossible (negative requirements).

Special care is taken to ensure that variables are instantiated when needed, either by taking values from the correct domains or by being unified with dummy values generated by the system.

The extensions introduced by the present work will be briefly described in the next section.

2. Extensions to WARPLAN

Negative literals are allowed in the goal conjunction, indicating facts to be deleted. This was easily implemented by including a "not_holds" meta-predicate.

Besides enforcing consistency on applying an operation we also require that the application be fully "productive", in the sense that none of the facts it is supposed to add currently holds and all facts it should delete do hold. Although this may seem too strong, it proved useful to further reduce the need for explicit preconditions; for example, in an operation to transfer a student S from a course A to a course B we do not have to require that A and B be different, since if they were the same the operation would not be productive in any situation.

As in WARPLAN, we characterize each operation by its effects ("added" and "deleted" clauses) and preconditions ("precond"). However our clauses have an extra parameter to denote the state reached so far, which provides values for some parameters and allows to determine preconditions with respect to this state, rather than only with respect to the initial state.

Since the "precond" clauses can have antecedents, we can generate conjunctions of arbitrary length of positive and/or negative facts to be taken as sub-goals, thus leading to some degree of iteration in forming a plan. For example, to cancel a course a precondition may be to first require that each student then taking the course should not take it.

We further characterize operations by "operation" clauses, giving the number of parameters and possibly indicating their domains,

as a device to ensure that variables be instantiated.

Both the original and the revised versions of WARPLAN are able, on backtracking, to generate alternative plans, some of which are simply permutations of the same operation-appplications. When displaying each alternative plan, we recognized the convenience to also show in what they differ in the non-trivial cases: if different operations are present they may add and/or delete different facts to achieve their preconditions, besides of course performing the additions and deletions required by the goal expression.

As the alternative plans and their net-effects are exhibited, we disposed that the user can select one of them and have it executed.

Finally, our "plans(Goal,Plan)" meta-predicate is invertible. If "Plan" is given and "Goal" is a variable, it will be unified with the conjunction of net effects of "Plan". We feel that this illustrates nicely the duality between the concepts of facts and operations, as denotations of states.

In the next section an example application is presented. The program itself in ARITY Prolog [Mal] is given in the Appendix.

3. An example application

Our example, already used in [VF], refers to an academic database involving courses and students. Facts are of two kinds:

- offered(C) - course C is offered
- takes(S,C) - student S takes course C

The database conforms to the integrity constraint that a student can only be taking a course that is being offered. Also, the number of courses taken by a student cannot decrease (during the term). Only the operations below are provided; note that the requirement on the number of courses is trivially preserved since there is no operation to simply drop a course:

- offer(C)
  effects: offered(C) is added
  preconditions: none

- cancel(C)
  effects: offered(C) is deleted
  preconditions: not takes(S,C) is required for each S taking C

- enroll(S,C)
  effects: takes(S,C) is added

```
  preconditions: offered(C)

- transfer(S,C1,C2)
  effects: takes(S,C1) is deleted and takes(S,C2) is added
  preconditions: offered(C2)
```

In this simple-minded example the domain of courses is {a,b,c} and the domain of students is {u,v}. The Prolog clauses are:

```
fact(offered(C)).
fact(takes(S,C)).

operation(offer(C)) :-
  on(C,[a,b,c]).
operation(cancel(C)) :-
  on(C,[a,b,c]).
operation(enroll(S,C)) :-
  on(S,[u,v]),
  on(C,[a,b,c]).
operation(transfer(S,C1,C2)) :-
  on(S,[u,v]),
  on(C1,[a,b,c]),
  on(C2,[a,b,c]).

added(offered(C),offer(C),_).
added(takes(S,C),enroll(S,C),_).
added(takes(S,C2),transfer(S,C1,C2),_).

deleted(offered(C),cancel(C),_).
deleted(takes(S,C1),transfer(S,C1,C2),_).

precond(offer(C),true,_).
precond(cancel(C),X,St) :-
  sub_goals(X,not takes(_,C),St).
precond(enroll(S,C),offered(C),_).
precond(transfer(S,C1,C2),offered(C2),_).
```

The reader will notice that we took advantage of the extra state parameter in the "precond" clause for "cancel".

The example fits with the abstract data type paradigm, in that the database can only be updated by the operations supplied, whose preconditions were carefully chosen. A benefit of this orientation is that, starting from a state where all integrity constraints hold, it can be shown that any state produced by executing a plan formed by the algorithm will also be valid.

To test the Prolog program, the initial state below can be given:

```
offered(a).
offered(c).
takes(u,a).
takes(v,a).
takes(u,c).
```

and the reader may want to try:

```
:- plans((takes(u,b), takes(v,c)), P).
:- plans(not offered(a),P).
:- plans(G,start=>offer(b)=>transfer(u,a,b)).
:- pl(takes(u,b)).
```

The "pl" meta-predicate is the one that displays net effects and allows to execute one of the alternative plans generated. We trace it next. The ";" prompt typed by the user causes the interpreter to backtrack, thus producing another alternative. Our own ">" prompt selects an alternative for execution.

```
:- pl(takes(u,b)).

start=>offer(b)=>enroll(u,b)
+ [offered(b),takes(u,b)]
- [];

start=>offer(b)=>transfer(u,a,b)
+ [offered(b),takes(u,b)]
- [takes(u,a)]>

:- listing(offered), listing(takes).

offered(a)
offered(c)
offered(b)
takes(v,a)
takes(u,c)
takes(u,b)
```

Because we deliberately limited our example, some convenient features of the algorithm were deferred to the next section. These have to do with the meaning of expressions given as goals and sub-goals (in the "precond") clauses, as well as with the invariants that can be separately declared as in WARPLAN.

4. Restrictions and propagations

As seen, the head of "precond" clauses contains, besides the template of the operation O involved, a parameter L consisting of a logical expression and a parameter S to transmit the state reached so far. L is regarded as a sub-goal to be attained before the operation O is applied.

Thus, if L is the atom "true", O can be applied immediately. L is handled as a non-trivial sub-goal, however, if it consists of one literal or a conjunction of literals that can be affected by the operations supplied. This policy differs in a subtle way from what is usually understood in the database area; for example, if we establish that students taking a course that is cancelled must be transferred to other courses, this side-effect is accomplished

- before cancelling the course, as a sub-goal, by the plan-generation method;
- after cancelling the course, through the so-called trigger or propagation mechanisms, by database systems.

The database policy then allows temporary inconsistent states, to be immediately fixed by the propagated actions. The temporary inconsistency is sometimes masked by regarding the operation together with these actions as a single operational unit, called a transaction. From a practical standpoint, the two policies lead to the same end results. Note in passing that there are situations where the transaction concept is inevitable, as happens with well-known accounting examples where two operations O1 and O2 "compensate" each other, so that any order of execution passes through an inconsistent state.

Another usual database policy is to restrict the application of O. In the academic example we would then reject an attempt to cancel a course taken by one or more students. In plan-generation, this is implemented by using antecedents in the "precond" clauses:

```
precond(cancel(C),true,St) :- not holds(takes(_,C),St).
```

If some student is taking C as state St is reached then "precond" fails and "cancel" cannot be applied. A stronger requirement can be imposed by

```
precond(cancel(C),true,_) :- not holds(takes(_,C),start).
```

since it refers to the state where the execution of the entire plan begins, so that even the goal below would fail in the example of the previous section:

```
:- plans((takes(v,b), takes(u,b), not offered(a)), P).
```

as "cancel(a)" is tried after generating

start=>offer(b)=>transfer(u,a,b)=>transfer(v,a,b)

Until now our examples had only literals that could be affected (added and/or deleted) by operations in our goal or sub-goal expressions. Actually, they can be quite general logical expressions, containing variables, built-in or user-defined predicates, etc. For instance:

:- plans((takes(u,C1), takes(v,C2), not C1 = C2), P).

calls for a state where each of the two students takes at least one different course.

Finally, to factor-out expressions that must remain invariant under any operation, even when involving literals of the types that could be affected, the "imposs" clauses of WARPLAN can be used. To say that in any state no student can take both courses a and b, we can write:

imposs((takes(S,a), takes(S,b))).

In our version we were careful to test for violation of such clauses, not only for goals and sub-goals, but also for additional effects of operations not directly mentioned in goal or sub-goal expressions. For instance, if "takes(u,a)" and "takes(u,c)" presently hold, we may consider the introduction of "transfer(u,c,b)" in a plan to accomplish the goal "not takes(u,c)", but the operation would also yield "takes(u,b)", which combined with "takes(u,a)", would violate the "imposs" clause above. On the other hand, we do not need the "always(P)" clauses of WARPLAN for facts P that are always true, their meaning being captured by representing P explicitly and by requiring "imposs(not P)".

In the database area, the "imposs" requirements correspond to the direct declaration of integrity constraints, instead of enforcing them by restricting the application of each relevant operation.

5. Conclusion

Although not even with the present extensions the algorithm covers more than rather simple planning problems, it is surprisingly powerful for its size and has proven to be quite useful in situations that occur in practical not too complex databases. In [Ch] the reader will find an extensive list of references on this kind of planning.

We are presently checking and evaluating our implementation. Another extension already introduced but not discussed here, is the treatment of facts residing in secondary storage under an SQL database management system. We arranged to handle both workspace and secondary storage facts uniformily in Prolog, letting calls to SQL facts be expressed as ordinary Prolog clauses that are automatically compiled into SQL commands and executed through the ARITY Prolog/SQL interface [Fu].

Our research on plan-generation is part of the NICE project towards cooperative environments for information systems usage.

References

[CCP] H. Coelho, J. C. Cotta and L. M. Pereira - "How to solve it with Prolog" - Laboratorio Nacional de Engenharia Civil - (1982).

[Ch] D. Chapman - "Planning for conjunctive goals" - Artificial Intelligence - 32, 3 (1987) 333-377.

[Fu] A. L. Furtado - "Treating wokspace and SQL facts uniformily in Prolog" - PUC/RJ technical report (1989).

[Ma] C. Marcus - "Prolog programming" - Addison-Wesley (1986).

[Ni] N. J. Nilsson - "Principles of artificial intelligence" - Springer (1982).

[VF] P. A. S. Veloso and A. L. Furtado - "Towards simpler and yet complete formal specifications" - in "Information systems: theoretical and formal aspects" - A. Sernadas, J. Bubenko Jr., and A. Olive (eds.) - North-Holand (1985) 175-189.

[Wa] D. S. D. Warren - "WARPLAN - a system for generating plans" - DAI, memo 76 - University of Edinburgh (1974).

Appendix

```
% plan generation

:- op(650,yfx,=>).

pl(C) :-
  plans(C,T),
  nl, write(T),
  s_added(T,A),
  nl, write($+ $), write(A),
  s_deleted(T,D),
  nl, write($- $), write(D),
  get0(K), nl,
  (K = 62, !,
   exec_plan1(T);
   not K = 59, !;
   fail).
pl(C).

plans(P,T)  :-
  var(P), !,
  net_effects(T,P),
  consistent(P,nil),
  chk(T).
plans(C,T) :-
  not consistent(C,true), !, fail.
plans(C,T) :-
  plan(C,true,start,T),
  net_effects(T,P),
  consistent(P,nil).

plan((X , C),P,T,T2) :- !,
  solve(X,P,T,P1,T1),
  plan(C,P1,T1,T2).
plan(X,P,T,T1) :-
  solve(X,P,T,P1,T1).

solve(not X,P,T,P,T) :-
  not fact(X), not X.
solve(not X,P,T,P1,T) :-
  fact(X),
  not_holds(X,T),
  and(not X,P,P1).
solve(not X,P,T,(not X , P),T1) :-
  fact(X),
  deleted(X,U,T),
  operation(U),
  achieve(not X,U,P,T,T1).
solve(not _,_,_,_,_) :- !, fail.
solve(X,P,T,P,T) :-
```

```
    not fact(X), X.
solve(X,P,T,P1,T) :-
    fact(X),
    holds(X,T),
    and(X,P,P1).
solve(X,P,T,(X , P),T1) :-
    fact(X),
    added(X,U,T),
    operation(U),
    achieve(X,U,P,T,T1).

achieve(X,U,P,T,T1 => U) :-
    precond(U,C,T),
    consistent(C,P),
    plan((true , C),P,T,T1),
    productive(U,T1),
    preserves(T1 => U,P).
achieve(X,U,P,T => V,T1 => V) :-
    retrace(P,V,P1,T),
    achieve(X,U,P1,T,T1),
    productive(V,T1),
    precond(V,C1,T1),
    preserves(T1,(true , C1)),
    preserved(X,T1 => V).

preserves(U,(X , C)) :- !,
    preserved(X,U),
    preserves(U,C).
preserves(U,X) :-
    preserved(X,U).

preserved(not X,V) :-
    holds(X,V), !, fail.
preserved(not X,V) :- !.
preserved(true,V) :- !.
preserved(X,V) :-
    not_holds(X,V), !, fail.
preserved(_,_).

retrace((not P , C),V,X,T) :-
    deleted(P,V,T), !,
    retrace(C,V,X,T).
retrace((P , C),V,X,T) :-
    added(P,V,T), !,
    retrace(C,V,X,T).
retrace((P , C),V,(P , C1),T) :-
    retrace(C,V,C1,T).
retrace(true,V,true,T).

holds(X,T => V) :-
    added(X,V,T),
```

```
  operation(V).
holds(X,T => V) :- !,
  holds(X,T),
  not deleted(X,V,T).
holds(X,T) :-
  given(T,X).

not_holds(X,T => V) :-
  deleted(X,V,T),
  operation(V).
not_holds(X,T => V) :- !,
  not_holds(X,T),
  not added(X,V,T).
not_holds(X,T) :-
  not given(T,X).

given(start,X) :-
  fact(X),
  X.

productive(U,T) :-
  added(X,U,T),
  holds(X,T), !, fail.
productive(U,T) :-
  deleted(X,U,T),
  not_holds(X,T), !, fail.
productive(_,_).

consistent(C,P) :-
  imposs(S),
  not (not intersect(C,S)),
  (P == nil, implied1(S,C);
   not P == true, implied(S,(C,P))),
  !, fail.
consistent(C,P) :-
  elem(X,C),
  elem(not X,C),
  !, fail.
consistent(_,_).

implied((S1 , S2),C) :- !,
  implied(S1,C),
  implied(S2,C).
implied(not X,C) :-
  elem(X,C), !, fail.
implied(X,C) :-
  elem(not X,C), !, fail.
implied(X,C) :-
  elem(X,C).
implied(X,C) :-
  not fact(X),
```

```
  X.

implied1((S1 , S2),C) :- !,
  implied1(S1,C),
  implied1(S2,C).
implied1(not X,C) :-
  elem(X,C), !, fail.
implied1(X,C) :-
  elem(not X,C), !, fail.
implied1(X,C) :-
  elem(X,C).
implied1(X,C) :-
  X.

intersect(S1,S2) :- !,
  elem(X,S1),
  elem(X,S2).

and(X,P,P) :-
  elem(Y,P),
  X == Y, !.
and(X,P,(X , P)).

elem(X,(Y , C)) :-
  elem(X,Y).
elem(X,(Y , C)) :- !,
  elem(X,C).
elem(X,X).

sub_goals(X,not Y,S) :- !,
  xsetof(Y,holds(Y,S),Z),
  neg_conj_list(X,Z).
sub_goals(X,Y,S) :-
  xsetof(Y,holds(Y,S),Z),
  conj_list(X,Z).


% checking and determining the effects of plans

chk(start) :- !.
chk(T => U) :-
  precond(U,C,T),
  effects(T,E),
  consistent(C,E),
  forall(elem(F,C),
    (F == true; holds(F,T))),
  productive(U,T),
  chk(T).

sub_plan(S,P) :-
  exlp(P,P1),
```

```
  append(S1,_,P1),
  one(ex1p(S,S1)).

part_plan(S,T) :-
  (not var(S), p_pl(S,T);
   var(S),
   op_plan(L, T=>'$'),
   p_pl(S=>L,T=>'$')).

p_pl(O,T=>O) :-
  not O = _=>_.
p_pl(S=>O,T=>O) :- !,
  p_pl1(S,T).
p_pl(S,T=>O) :-
  p_pl(S,T).

p_pl1(O,T=>O) :-
  not O = _=>_.
p_pl1(S=>O,T=>O) :-
  p_pl1(S,T).

op_plan(O,T=>O).
op_plan(P,T=>O) :-
  op_plan(P,T).

effects(start,[]) :- !.
effects(T,P) :-
  xbagof(F, F1 ^ (s_add1(F,T) ;
                  (s_del1(F1,T), F = (not F1))),
         C),
  conj_list(P,C).

net_effects(T,P) :-
  xbagof(F, F1 ^ ((s_add1(F,T), not holds(F,start)) ;
                  (s_del1(F1,T), holds(F1,start), F = (not F1))),
         L),
  conj_list(P,L).

s_added(T,S) :-
  xsetof(A,s_add1(A,T),S).

s_add1(X,T => V) :-
  added(X,V,T),
  operation(V).
s_add1(X,T => V) :- !,
  s_add1(X,T),
  not deleted(X,V,T).

s_deleted(T,S) :-
  xsetof(A,s_del1(A,T),S).
```

```
s_del1(X,T => V) :-
  deleted(X,V,T),
  operation(V).
s_del1(X,T => V) :- !,
  s_del1(X,T),
  not added(X,V,T).


% plan execution

exec_plan(T) :-
  plans(_,T),
  exec_plan1(T).

exec_plan1(T) :-
  ex1p(T,V),
  ex2p(V).

ex1p(start,[]).
ex1p(Ts => T,V) :- !,
  ex1p(Ts,Vs),
  append(Vs,[T],V).

ex2p([]) :- !.
ex2p([F|R]) :-
  forall(added(P,F,_), assert(P)),
  forall(deleted(P,F,_), retract(P)),
  ex2p(R).


% general utilities

on(X,[X|_]).
on(X,[Y|R]) :-
  on(X,R).

item(E,[E|R],1).
item(E,[X|R],J) :-
  var(J),
  item(E,R,I),
  J is I + 1 .
item(E,[X|R],J) :-
  not var(J),
  not J == 1,
  I is J - 1,
  item(E,R,I).

card([],0).
card([A|R],N) :-
  card(R,M),
  N is M + 1 .
```

```
append([],X,X).
append([X|R],Y,[X|S]) :-
  append(R,Y,S).

conc(Ls,S) :-
  conc1(Ls,Ls1),
  concat(Ls1,S).

conc1([],[]).
conc1([X|R],[Y|S]) :-
  string_term(Y,X),
  conc1(R,S).

reverse(Xs,Ys) :-
  rev1(Xs,[],Ys).

rev1([X|Xs],A,Ys) :-
  rev1(Xs,[X|A],Ys).
rev1([],Ys,Ys).

copy(X,Y) :-
  listvar(X,L1),
  cop1(L1,L),
  cop2(X,L,Y), !.

cop1([],[]).
cop1([V|R],[[V,W]|S]) :-
  cop1(R,S).

cop2(X,L,Y) :-
  case([atomic(X) -> Y = X,
        functor(X,F,N) -> (functor(Y,F,N), cop3(0,N,X,L,Y)),
        var(X) -> (cop4(X,X1,L), Y = X1)]).

cop3(N,N,X,L,Y) :- !.
cop3(I,N,X,L,Y) :-
  J is I + 1,
  arg(J,X,A),
  arg(J,Y,B),
  cop2(A,L,B),
  cop3(J,N,X,L,Y).

cop4(X,X1,[[Y,X1]|R]) :-
  X == Y, !.
cop4(X,X1,[P|R]) :-
  cop4(X,X1,R).

v_on(X,[Y|Z]) :-
  var(Y),!,
  (X == Y ;
```

```
    v_on(X,Z)).
v_on(X,[X|_]).
v_on(X,[Y|Z]) :-
  v_on(X,Z).

no_dup([X|Xs],Ys) :-
  v_on(X,Xs), !,
  no_dup(Xs,Ys).
no_dup([X|Xs],[X|Ys]) :-
  not v_on(X,Xs),
  no_dup(Xs,Ys).
no_dup([],[]).

v_flat(A,B,[A|B]) :-
  var(A), !.
v_flat(A,B,B) :-
  atomic(A), !.
v_flat([A|R],B,C) :- !,
  v_flat(R,B,T),
  v_flat(A,T,C).
v_flat(A,B,C) :-
  A =.. [F|R], !,
  v_flat(R,B,C).

listvar(X,Y) :-
  v_flat(X,[],T1),
  reverse(T1,T2),
  no_dup(T2,T3),
  reverse(T3,Y).

name_var(F) :-
  listvar(F,L),
  name_var1(L,1).

name_var(P,P1) :-
  copy(P,P1),
  name_var(P1).

name_var1([],I).
name_var1([X|R],I) :-
  J is I + 1,
  int_text(I,K),
  concat($V$,K,X),
  name_var1(R,J).

one(P) :- P, !.

forall(X,Y) :-
  not (X, not Y).

xsetof(X,Y,Z) :-
```

```
  (setof(X,Y,Z), !; Z = []).

xbagof(X,Y,Z) :-
  (bagof(X,Y,Z), !; Z = []).

neg_conj_list(true,[]) :- !.
neg_conj_list(not X,[X]) :- !.
neg_conj_list((not X , C),[X|Z]) :-
  neg_conj_list(C,Z).

conj_list(true,[]) :- !.
conj_list(X,[X]) :- not X =.. [',',_,_], !.
conj_list((X , C),[X|Z]) :-
  conj_list(C,Z).
```