REFERENCES

1. Bratko, I. Prolog programming for artificial intelligence. Great Britain: Addison Wesley, 1990.
2. Idestam-Almquist, P. Generalization of clauses. Doctoral dissertation, Stockholm University and the Royal Institute of Technology, 1993.
3. _____. Generalization under implication by recursive anti-unification. International conference on machine learning 10 (1993): 151-158.
4. Le, T. V. Techniques of prolog programming. Singapore: John Wiley and Sons. Inc., 1993.
5. Lloyd, J. W. Foundations of logic programming. Germany: Springer- Verlag,1984.
6. Muggleton, S. Inverting Implication. Artificial Intelligence Journal. 1993.
7. Nienhuys-Cheng, S.-H., and de Wolf, R. M. The subsumption theorem in inductive logic programming: facts and fallacies. Workshop on Inductive Logic Programming (ILP-95). Bad Honnef, Germany, 1995.
8. Plotkin, G. D. Automatic Methods of Inductive Inference. Doctoral dissertation, Edinburgh University, 1971.
9. Robinson, J. A. A machine-oriented logic based on the resolution principle. Journal of the ACM 12, 1 (1965): 23-41.

APPENDIX
PROLOG PROGRAMME

In this appendix we will show a programme which implements the *J*-algorithm. This programme computes indirect roots of clauses, even if they contain cross connections .The output from the programme is a generalization under implication of the input.

The Main Programme

```
/* lgg_j(+D,?C):-                    [pro_m.pro]
        The clause C is an indirect root of the clause D by the J-algorithm. */

Note that an argument which is preceeded by the + notation is an input and an
argument which is preceeded by the ? notation is an output.

:- [pro_11].          /* Load and compile the subprogrammes. */
:- [pro_21].
:- [pro_41].
:- [pro_51].

lgg_j(D,G):-
        length_neg(D,M),              /* See the 4th subprogramme. */
        lggj_1(D,M,G).
```

```
lggj_1(D,M,G):-
/* Check the value of M, if M equals one, then the output is the input. */
        M==1, !, G = D.
lggj_1(D,M,G):-
/* Check the value of M, if M does not equal one, then do the following things. */
        original_clause(D,C),                   /* See the 1st subprogramme. */
        C1 = C, C2 = C,                         /* Let C1 and C2 equal to C. */
        resol(D,C1,C2,M,C,G).

resol(D,C1,C2,M,C,G):-
/* Resolve C1 with C2, and check the value of M and N. */
        resolution(C1,C2,Cs),                   /* See the 2nd subprogramme. */
        length_neg(Cs,N),                       /* See the 4th subprogramme. */
        check_length(D,Cs,M,N,C,G).

check_length(D,Cs,M,N,C,G):-
/* Compare the values of M and N, if M equals N, then compare clauses D and Cs.*/
        M==N, !,
        equal_length(D,Cs,M,N,C,G).
check_length(D,Cs,M,_,C,G):-
/* If M does not equal N, then do the following thing. */
        C1 = C, C2 = Cs,                        /* Let C1 equal C, and C2 equal Cs. */
        resol(D,C1,C2,M,C,G).
```

```
equal_length(D,Cs,M,N,C,G):-
/* Compare clauses D and Cs, if we can find a substitution T such that CsT = D, do
the following thing. */
        compare_t([],HVars,Cs,D,[],T),      /* See the 3rd subprogramme. */
        check_p(T,C),                       /* See the 3rd subprogramme. */
        subst_in_1(C,T,Cc),                 /* Replace C with CT. */
        C1 = Cc, C2 = Cc,                   /* Let C1 and C2 equal Cc. */
        resol(D,C1,C2,M,Cc,G).
equal_length(D,Cs,M,N,C,G):-
/* Compare clauses D and Cs, if we can obtain D from Cs by substituting for some
variables which do not occur in C, then finish, with the output C. */
        compare_t([],HVars,Cs,D,[],T),
        not check_p(T,C),
        check_vars(D,C),
        G = C.
```

The 1st Subprogramme

```
/* original_clause(+C1,?C2):-                    [pro_11.pro]
        The clause C2 has the same positive and negative predicates as clause C1, but
no negative predicate is repeated. Otherwise, the terms in the negative predicates in C2
are replaced with new variables. */

original_clause(C1,C2):-
        check(C1,C3),                  /* The clause C3 has the same positive and
negative predicates as clause C1, but no negative predicate is repeated. */
        choose(C3,[],_,C2).            /* The terms in the negative predicates in C2 are
replaced with new variables. */

check([X|Xs],[X|Z]):- check_neg(Xs,Z).
check_neg([],[]).
/* Check that the clause has no negative predicate which is repeated. */
check_neg([X|Xs],Y):-
        check_neg(Xs,Y),
        in_list(X,Y), !.
check_neg([X|Xs],[X|Y]):-
        check_neg(Xs,Y).

in_list(X,[Z|Zs]):-
/* Check that the 1st clause is in the 2nd clause. */
        nonvar(X), nonvar(Z),
        arg(1,X,X1), arg(1,Z,Z1),
        functor(X1,F,N), functor(Z1,F,N), !.
in_list(X,[_|Zs]):- in_list(X,Zs).
```

```
choose([],S,S,[]).
choose([X|Xs],S1,S3,[X|Zs]):-
/* Not change the terms in the positive predicate. */
        X \= not(Y), !,
        choose(Xs,S1,S3,Zs).

choose([X|Xs],S1,S3,[Z|Zs]):-
/* Change the terms in the negative predicate to the new variables. */
        X = not(Y),
        anti_u(X,Z),
        nonvar(Z), !,
        choose(Xs,S2,S3,Zs).

anti_u(X,Z):-
        nonvar(X), functor(X,not,1), !,
        functor(Z,not,1), arg(1,X,X1),
        anti_u(X1,Z1), arg(1,Z,Z1).
anti_u(X,Z):-
        nonvar(X), functor(X,F,N), !,
        functor(Z,F,N),
        anti_u_args(N,X,Z).

anti_u_args(0,_,_):- !.
anti_u_args(N,X,Z):-
        N>0,
        arg(N,Z,ZN), N1 is N-1,
        anti_u_args(N1,X,Z).
```

The 2nd Subprogramme

```
/* resolution(+C1,+C2,?C):-                    [pro_21.pro]
        The clause C comes from resolving the clause C1 with the clause C2. */

resolution(C1,C2,C):-
        reverse(C1,C3),
        new_vars(C2,C4),
        resol_1(C3,C4,C).

reverse([],[]).                          /* The 2nd list is the reverse of the 1st list. */
reverse([X|Xs],Y):- reverse(Xs,Ys), append(Ys,[X],Y).

append([],Ys,Ys).
/* The 3rd list is the concatenation of the 1st list and the 2nd list. */
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).

new_vars(C,Cp):-
/* Introduce new variables into the clause Cp to make the variables in Cp disjoint from
those in C. */
        asserta($tmp(C)),              /* Reserve the memory. */
        retract($tmp(Cp)).             /* Clear the momery. */

select(X,[X|Xs],Xs).
/* The element X is selected and removed from the 1st list obtaining the 2nd list. */
select(X,[Y|Ys],[Y|Zs]):- select(X,Ys,Zs).
```

```
resol_1([],_,[]).
/* Resolve the 1st clause with the 2nd clause to get the 3rd clause. */
resol_1([(not X)|Xs],[Y|Ys],Z):-
        unify(X,Y,[],S2,_),
        subst_in(Xs,S2,Xsp),
        subst_in(Ys,S2,Ysp),
        reverse(Xsp,Xspp),
        append(Xspp,Ysp,Z).

unify(X,Y,S,S,X):- X==Y, !.
/* Find the substitution S such that never replace a variable in 1st clause with one of
the new variables introduced into the 2nd clause. */
unify(X,Y,S1,[(X)/Y|S1],X):- var(Y), !.
unify(X,Y,S1,[(Y)/X|S1],Y):- var(X), !.
unify(X,Y,S1,S2,Z):-
        functor(X,F,N), functor(Y,F,N), !, functor(Z,F,N),
        unify_args(N,X,Y,S1,S2,Z).

unify_args(0,_,_,S,S,_):- !.
unify_args(N,X,Y,S1,S2,Z):-
        N>0, arg(N,X,XN), arg(N,Y,YN),
        unify(XN,YN,S1,S3,ZN), arg(N,Z,ZN),
        subst_term(X,S3,X1), subst_term(Y,S3,Y1),
        N1 is N-1, unify_args(N1,X,Y,S3,S2,Z).

subst_in([],_,[]).
subst_in([X|Xs],S,[Z|Zs]):-
/* The last clause comes from substituting the 1st clause with substitution S. */
        subst_term(X,S,Z),
        subst_in(Xs,S,Zs).
```

```
subst_term(X,S,Z):-
        nonvar(X), functor(X,F,N), !,
        functor(Z,F,N),
        subst_term_args(N,X,S,Z).
subst_term(X,S,X2):-
        var(X), select((X2)/V,S,_),
        X==V, !.
subst_term(X,S,X):- var(X).

subst_term_args(0,_,_,_):- !.
subst_term_args(N,X,S,Z):-
        N>0, arg(N,X,XN),
        subst_term(XN,S,ZN),
        arg(N,Z,ZN), N1 is N-1,
        subst_term_args(N1,X,S,Z).

subst_in_1([],_,[]).
subst_in_1([X|Xs],S,[Z|Zs]):-
        subst_term_1(X,S,Z),
        subst_in_1(Xs,S,Zs).

subst_term_1(X,S,Z):-
        nonvar(X), functor(X,F,N), !,
        functor(Z,F,N),
        subst_term_args_1(N,X,S,Z).
subst_term_1(X,S,X2):-
        var(X), select(V/(X2),S,_),
        X==V, !.
subst_term_1(X,S,X):- var(X).
```

```
subst_term_args_1(0,_,_,_):- !.
subst_term_args_1(N,X,S,Z):-
        N>0, arg(N,X,XN),
        subst_term_1(XN,S,ZN),
        arg(N,Z,ZN), N1 is N-1,
        subst_term_args_1(N1,X,S,Z).
```

The 3rd Subprogramme

```
/* compare_t(+HVarsIn,+HVarsOut,+Cs,+D,+[],?T):-          [pro_41.pro]
        Compare clauses D and Cs and find a substitution T such that CsT ⊆ D. */

compare_t(HVars,HVars,[],_,S,S).
compare_t(HVarsIn,HVarsOut,[X|Xs],Ys,S1,S3):-
        select(Y,Ys,_),
        put_u(HVarsIn,HVarsTmp,X,Y,S1,S2),
        delete(Y,Ys,NYs),
        compare_t(HVarsTmp,HVarsOut,Xs,NYs,S2,S3).

put_u(HVarsIn,HVarsOut,X,Y,S1,S2):-
        nonvar(X), nonvar(Y),
        functor(X,F,N), functor(Y,F,N),!,
        put_u_args(HVarsIn,HVarsOut,N,X,Y,S1,S2).
put_u(HVarsIn,HVarsOut,X,Y,S1,S2):-
        var(X),
        check_subst(HVarsIn,HVarsOut,X/Y,S1,S2).

put_u_args(HVars,HVars0,_,_,S,S):- !.
put_u_args(HVarsIn,HVarsOut,N,X,Y,S1,S3):-
        N>0, arg(N,X,XN), arg(N,Y,YN),
        put_u(HVarsIn,HVarsTmp,XN,YN,S1,S2),
        N1 is N-1, put_u_args(HVarsTmp,HVarsOut,N1,X,Y,S2,S3).

check_subst(HVars,HVarsX1/T1,[X2/T2|S],[X2/T2|S]):-
        X1==X2, !, T1==T2.
/* Check that if the 1st terms of 2 substitutions equal each other, then the 2nd terms
must be equal, too.*/
```

```
check_subst(HVarsIn,HVarsOut,X1/T1,[X2/T2|S1],[X2/T2|S2]):-
        check_subst(HVarsIn,HVarsOut,X1/T1,S1,S2).
check_subst(HVars,[X|HVars],X/T,[],[]):- X==T, !.
/* Check that the 2nd term of the substitution does not equal the 1st term of it.*/
check_subst(HVars,HVarsX/T,[],[X/T]):-
        not member_v(X,HVars).

delete(X,[Y|Ys],Ys):- X == Y, !.
/* The element X is removed from the 1st list, obtaining the 2nd list. */
delete(X,[Y|Ys],[Y|Zs]):- delete(X,Ys,Zs).

check_p(T,C):-
        not not check_prime(T,C).

check_vars(D,C):-
        not not check_vars_1(D,C).

check_vars_1(D,C):-
/* Check that the variables in C are also in D. */
        vars_1(C,VarC),
        vars_1(D,VarD),
        check_member(VarC,VarD).

check_member([],_).
/* Check that the variables in the 1st list are also in the 2nd list. */
check_member([C|Cs],D):-
        member_v(C,D),
        check_member(Cs,D).
```

```
check_prime(T,C):-
/* Check that the 1st variable of the substituions in T is in C. */
        vars_1(C,VarC),
        check_p1(T,VarC).

vars_1(C,VarC):-
/* VarC is the list of all variables in clause C. */
        vars(C,Var),
        in_list_c(Var,VarC).

vars([],[]).
vars([X|Xs],VarC):-
        nonvar(X), !,
        functor(X,F,N),
        vars_args(N,X,VarC1),
        vars(Xs,VarC2),
        append(VarC1,VarC2,VarC).
vars([X|Xs],[X|VarC]):-
        var(X), !,
        vars(Xs,VarC).

vars_args(0,X,[]):- !.
vars_args(N,X,Z):-
        N>0, arg(N,X,XN),
        vars([XN],Z1),
        N1 is N-1, vars_args(N1,X,Z2),
        append(Z1,Z2,Z).
```

```
check_p1([X|Xs],VarC):-
/* Check that the 1st variable of the substitution in the 1st list is also in the 2nd list. */
        X = A/B,
        member_v(A,VarC), !.
check_p1([X|Xs],VarC) :-
        check_p1(Xs,VarC).

in_list_c([],[]).
/* Check that the terms in the 1st list are also in the 2nd list, if not, put them in the 2nd
list. */
in_list_c([X|Xs],Z):-
        member_v(X,Xs), !,
        in_list_c(Xs,Z).
in_list_c([X|Xs],[X|Z]):-
        in_list_c(Xs,Z).

member_v(X,[Y|Ys]) :- Y == X, !.    /* Check that X is a member in the 2nd list. */
member_v(X,[_|Xs]):-
        member_v(X,Xs).
```

The 4th Subprogramme

```
/* length_neg(+C1,?N):-                    [pro_51.pro]
        The length N is the maximum number of times that a single negative predicate
occurs in the clause C1. */

length_neg(C1,N):-
        check_neg2(C1,Z),find_max(Z,N).

check_neg2([],[]).
/* Find the number of times that a single negative predicate occurs in the 1st clause. */
check_neg2([X|Xs],Y):-
        X \= not(Z), !,
        check_neg2(Xs,Y).
check_neg2([X|Xs],Y1):-
        X = not(Z),
        check_neg2(Xs,Y),
        in_list3(X,Y,Y1), !.
check_neg2([X|Xs],[(X,1)|Y]):-
        X = not(Z),
        check_neg2(Xs,Y).

in_list3(X,[(Z,N)|Xs],[(Z,N1)|Xs]):-
/* Check the negative predicate in the 1st clause, if it is repeated, add one in N1. */
        nonvar(X), nonvar(Z),
        arg(1,X,X1), arg(1,Z,Z1),
        functor(X1,F,M), functor(Z1,F,M), !, N1 is N+1.
in_list3(X,[X|Xs],[X|Xsp]):- in_list3(X,Xs,Xsp).
```

```
find_max([],0).                    /* Find the maximum number of times that a
single negative predicate occurs in the 1st clause. */
find_max([(_,N)|X],N):-
        find_max(X,M), M < N, !.
find_max([_|X],N):-
        find_max(X,N).
```

VITA

Name	Miss Chotiros Surapholchai
Degree	B.Sc. (Mathematics), 1992 Chulalongkorn University Bangkok, Thailand.
Position	Instructor in the Department of Mathematics Faculty of Science, Chulalongkorn University Bangkok, Thailand.