Acta Universitatis Palackianae Olomucensis. Facultas Rerum Naturalium. Mathematica

Jan Štěpán

Propositional calculus proving methods in Prolog

Acta Universitatis Palackianae Olomucensis. Facultas Rerum Naturalium. Mathematica, Vol. 29 (1990), No. 1, 301--321

Persistent URL: http://dml.cz/dmlcz/120239

ACTA UNIVERSITATIS PALACKIANAE OLOMUCENSIS
FACULTAS RERUM NATURALIUM
1990 MATHEMATICA XXIX VOL. 97

Katedra výpočetní techniky
přírodovědecké fakulty Univerzity Palackého v Olomouci
Vedoucí katedry: Doc.RNDr.Drantišek Koliba, CSc.

PROPOSITIONAL CALCULUS PROVING METHODS IN PROLOG

JAN ŠTĚPÁN

(Received March 31, 1989)

Two methods of proving the theorems of the propositional calculus are described in this paper - Wang's algorithm (acc. [1]) and the method of analytical tables (acc. [3]). Two programs in Prolog are quotated to Wang's algorithm (from [1] and [2]), for the method of analytical tables author's program is presented. Efficiency of the programs is demonstrated on examples. Further, the practical and didactic value of presented methods and programs is discussed.

I. WANG'S METHOD

Wang's method lies in the transformation of given formula to the sequence of formulas, in which the relation of inference is held. Further, this sequence is simplified by transformation of its constituents (formulas). During these modifications either inference relation among these formulas is confirmed or it is evident that given formula is not provable. Proper process is following:

(Denotation: symbols ~, & , v, ⇒ , <=> denote negation, conjuction, disjunction, implication and equivalence respectively; → is the symbol of inference - let´s expression A → B, where A, B are sets (of formulas), call sequent, and the set of formulas A antecedent and set B succedent of the sequent A → B.

1. We express the given formula into the form of sequence, in which premises are on the left of the symbol → (separated by comma) and assertion is on the right of the symbol → , for instance
   premise1, premise2, ..., premisen → assertion;
   any of sequences separated by → can be empty.

2. The transformations of partial formulas are performed by these rules:
   - if the formula is negation, i.e. ~A, we erase it and on the other side of the sequent according to symbol → we add the formula A - for instance

     p v q, ~(r & s), ~q, p v r → s,~p

     we change to sequence

     p v q, p v r, p → s, r & s, q;
   - if the formula is conjuction in antecedent or disjunction in succedent, we replace these connectives by comma, for instance

     p & q, r & (~p v s) → ~q v ~r

     we change to sequence

     p, q, r, ~p v s → ~q, ~r;
   - if the formula is disjunction in antecedent or conjunction in succedent, we decompose considerated sequent to two sequents - the first one contains one argument of the formula, the second one contains remaining argument; each of these sequents must be further transformated separately; for instance

```
                     r, ~p v s -> ~q,   ~r & s
                      /                    \
      r, ~p -> ~q, ~r & s                r, s -> ~q, ~r & s
        /           \                      /           \
r,~p -> ~q,~r    r,~p -> ~q,s     r,s -> ~q,~r     r,s -> ~q, s
```

- if the formula is implication or equivalence, we replace it on the principle of the schemas:

  A => B ... ~A v B, or

  A <=> B ... (A => B) & (B => A), event.

  A <=> B ... (~A & ~B) v (A & B).

3. If there is the same formula both in the antecedent and succedent, the given formula is a theorem. If it is not more possible to apply any of the rules mentioned in 2 (i.e. both antecedent and succedent are sequences of atomic formulas and no of them occurs in the antecedent and the succedent at the same time), the given formula is not provable.

Note: In both following programs based on Wang´s method there is the symbol of inference used as the symbol for implication, which is not quite correct. It is motivated by technical reasons, the function of proper programs is not influenced and it is always evident from the documentation of the proofs which sense of the symbol => is considered.

II. ALGORITHMS IN PROLOG TO WANG´S METHOD

Algorithm 1 - comes from [1], where it is published with errors. In the same way it is accepted even in [2]. I bring up the original version here, incorrect clause is mentioned lately.

Logic program for algorith 1:

```
/* Operations */
:-op(700,xfy, <=> ).                    /* equivalence */
:-op(650,xfy,=> ).                      /* implication */
:-op(600,xfy,v).                        /* disjunction */
:-op(550,xfy,&).                        /* conjunction */
:-op(500,fy,~ ).                        /* negation    */
```

```
/* Read in and try to prove formula;
   write 'valid' or 'not valid' accordingly */
formulas:- repeat,write('Formula:'),nl,
           read(T),(T==stop;theorem(T),fail).
theorem(T):- nl,nl,
             (prove([ ] & [ ]=> [ ] & [T]),!,nl,
             write('Formula is valid');
             nl,write('Formula is not valid')),nl,nl.
to_prove(T):- write('prove,'),nl,write(T),nl,nl,
              prove(T).
prove(E1):- rule(E1,E2,Rule),!,
            write(E2),by_rule(Rule),nl,
            prove(E2).
/* Case for v on l.h.s. */
prove(L & [H v I | T] => R):- !,
           first_branch,to_prove(L & [H|T] => R),
           branch_proved,
           second_branch,to_prove(L & [I|T] => R),
           branch_proved.
/* Case for v on r.h.s. */
prove(L & [H & I|T] => R):- !,
           first_branch,to_prove(L => R & [H|T]),
           branch_proved,
           second_branch,to_prove(L => R & [I|T]),
           branch_proved.
/* Case for atom */
prove(L & [H|T] => R):- !,prove([H|L] & T => R).
prove(L => R & [H|T]):- !,prove(L => [H|R] & T).
/* Finally, check whether tautology */
prove(T):- tautology(T),write('Tautology.'),nl.
prove(_):- write('This branch is not provable.'),fail.
/* Case where => appears in one of the sides */
rule(L & [H => I|T] => R,
     L & [~H v I|T] => R, rule_5).
rule(L => R & [H => I|T],
     L => R & [~H v I|T], rule_6).
```

```
/* Cases where <=> appears in one of the sides */
rule(L & [H <=> I|T] => R,
     L & [(H => I) & (I => H)|T] => R, rule_7).
rule(L => R & [H <=> I|T],
     L => R & [(H => I) & (I => H)|T], rule_8).
/* Case where ~ appears */
rule(L & [~H|T] => R & R2,
     L & T => R & [H|R2], rule_2).
rule(L1 & L2 => R & [~H & T],
     L1 & [H|L2] => R & T, rule_2).
/* Case for & on l.h.s. */
rule(L & [H & I|T] => R,
     L & [H,I|T] => R, rule_3).
/* Case for v on r.h.s. */
rule(L => R & [H v I|T],
     L => R & [H,I|T], rule_3).
tautology(L & [ ] => R & [ ]):- member(M,L),
                                member(M,R).
branch_proved:- write('this branch has been proved.'),nl.
first_branch:- nl,write('First branch:').
second_branch:- nl,write('Second branch:').
by_rule(R):- write('      by '),write(R),nl,nl.
member(H,[H|_]).
member(I,[_|T]):- member(I,T).
```

Examples of algorithm 1 performance (by |1|):

```
Formula:
a => a.
[ ] & [ ] => [ ] & [~a v a]                      by rule_6
[ ] & [ ] => [ ] & [~a,a]                        by rule_3
[ ] & [a] => [ ] & [a]                           by rule_2
Tautology.
Formula is valid

Formula:
(a => b) & (b => c) => (a => c).
```

```
[ ]&[ ] => [ ]& [~((a => b)&(b => c)) v (a => c)]          by rule_6
[ ]&[ ] => [ ]& [~((a => b)&(b => c)),(a => c)]            by rule_3
[ ]&[(a => b)&(b => c)) => [ ]&[a => c]                    by rule_2
[ ]&[(a => b)&(b => c)] => [ ]&[ a v c]                    by rule_6
[ ]&[a => b,b => c] => [ ]&[ a v c]                        by rule_3
[ ]&[~a v b,b => c] => [ ]&[ a v c]                        by rule_5
[ ]&[~a v b,b => c] => [ ]&[ a,c]                          by rule_3
[ ]&[a,~a v b,b => c] => [ ]&[c]                           by rule_2
First branch: prove,
[a]&[ a,b => c] => [ ]&[c]
[a]&[b => c] => [ ]&[a,c]                                  by rule_2
[a]&[~b v c] => [ ]&[a,c]                                  by rule_5
First branch: prove,
[a]&[~b] => [ ]&[a,c]
[a]&[ ] => [ ]&[b,a,c]                                     by rule_2
Tautology.
This branch has been proved.
Second branch: prove,
[a]&[c] => [ ]&[a,c]
Tautology.
This branch has been proved.
Second branch: prove,
[a]&[b,b => c] => [ ]&[c]
[b,a]&[~b v c] => [ ]&[c]                                  by rule_5
First branch: prove,
[b,a]&[~b] => [ ]&[c]
[b,a]&[ ] => [ ]&[b,c]                                     by rule_2
Tautology.
This branch has been proved.
Second branch: prove,
[b,a]&[c] => [ ]&[c]
Tautology.
This branch has been proved.
Formula is valid.
```

These examples are quotated as it was mentioned above. But this program evaluates proper formulas as improvable, because it contains an error. Apart from that algorithm 1 causes runaway for the (improvable) formula

(~p v q)&(~q v r)&(~r v s)&(~u v s) => (~p v u)   (*)

which is recommended to verification of this program in |1| and |2|. The first deffect can be remedied by changing the second of rules labelled as "rule 2" to
"rule(L1 L2 = R | H T|, L1 |H L2| = R T, rule 2).".
The second deffect can be put away by suitable location of cut in the clauses "prove" (separating branches). Then we can introduce the proofs of other theorems for comparison.

Formula:
p v ~p.
[ ]&[ ] => [ ]&[p,~p]   by rule_3
[ ]&[p] => [p]&[ ]   by rule_2
Tautology.
Formula is valid.

Formula:
~(p&~p).
[ ]&[p&~p] => [ ]&[ ]   by rule_2
[ ]&[p,~p] => [ ]&[ ]   by rule_3
[p]&[ ] => [ ]&[p]   by rule_2
Tautology.
Formula is valid.

Formula:
~p&p.
First branch: prove,
[ ]&[ ] => [ ]&[~p]
[ ]&[p] => [ ]&[ ]   by rule_2
This branch is not provable.
Formula is not valid.

Algorithm 2 is founded on Wang´s method as well. It differs from algorithm 1 especially by technique of programming and more over by documentation of proof. The proof of (nonvalid) formula

(p => q) => ((p => r) =>(q => r))

following program leads to runaway, for quotated formula (*) as well. So it provides only partial decision of given formula provability.

Logic program of algorithm 2:

```
/* Wang's algorithm */
:-op(700,xfy,<=>).              /* equivalence */
:-op(600,xfy,=>).               /* implication */
:-op(500,xfy,v).                /* disjunction */
:-op(400,xfy,&).                /* conjunction */
:-op(300,fy,~).                 /* negation    */
wang:- nl,nl,write('Formula:'),nl,read(T),
       (T==stop,!;prove(T),wang).
prove(L => R):-
   nl,theorem(L& true => R v false),!,  /* procedure theorem */
   write('Formula is a theorem');       /* requires this */
   nl,write('Formula is not a theorem').
prove(T):- prove(true => T).
theorem(T):- nl,write('Prove:'),write(T),(tautology(T));
             perpartes(T);transf(T,T1),theorem(T1)).
tautology(L => R):- conmember(Exp,L),dismember(Exp,L),!,
                    nl,write('   This is tautology'),nl.
perpartes(L => R):- conconc(L1,(E1 v E2)& L2,L),
                    conconc(L1,L2,LL),
                    nl,write('First branch:'),nl,
                    theorem(E1& LL => R),
                    nl,write('Second branch:'),nl,
                    theorem(E2& LL => R).
perpartes(L => R):- disconc(R1,(E1& E2) v R2,R),
                    disconc(R1,R2,RR),theorem(L => E1 v RR),
                    theorem(L => E2 v RR).
transf(L => R,LL => Exp v R):-        /* negation */
   conconc(L1,~Exp& L2,L),
   conconc(L1,L2,LL).
```

```
transf(L => R,Exp&L => RR):- disconc(R1,~Exp v R2,R),
                             disconc(R1,R2,RR).
transf(L => R,LL => R):- conconc(L1,(A&B)&L2,L),
                         conconc(L1,A&(B&L2),LL).
transf(L => R,L => RR):- disconc(R1,(A v B) v R2,R),
                         disconc(R1,A v (B v R2),RR).
transf(L => R,LL => R):- conconc(L1,Exp&L2,L),rule(Exp,Expl),
                         conconc(L1,Expl&L2,LL).
transf(L => R,L => RR):- disconc(R1,Exp v R2,R),rule(Exp,Expl),
                         disconc(R1,Expl v R2,RR).
/* Rules */
rule(A => B,~A v B).
rule(A <=> B,(~A & ~B) v (A&B)).
conconc(true,Exp,Exp).
conconc(Term&Expl,Exp2,Term&Exp3):-
                        conconc(Expl,Exp2,Exp3).
conmember(Term,Exp):- conconc(Expl,Term&Exp2,Exp).
disconc(false,Exp,Exp).
disconc(Term v Expl,Exp2,Term v Exp3):-
                         disconc(Expl,Exp2,Exp3).
dismember(Term,Exp):- discont(Expl,Term v Exp2,Exp).
/*Empty expression on the left is true, on the right is false*/
```

Examples of algorithm 2 performance:

```
Formula:
p => p.
Prove: p&true => p v false
   This is tautology
Formula is a theorem


Formula:
p v ~p.
Prove: true&true => (p v ~p) v false
Prove: true&true => p v ~p v false
Prove: p&true&true => p v false
   This is tautology
Formula is a theorem
```

```
Formula:
~(p & ~p).
Prove: true & true => ~(p & ~p) v false
Prove: (p & ~p) & true & true => false
Prove: p & ~p & true & true => false
Prove: p & true & true => p v false
   This is tautology
Formula is a theorem
Formula:
~p & p.
Prove: true & true => ~p & p v false
Prove: true & true => ~p v false
Prove: p & true & true => false
Formula is not a theorem


Formula:
 (p => q) & (q => r) => (p => r).
 Prove: ((p => q) & (q => r)) & true => (p => r) v false
 Prove: (p => q) & (q => r) & true => (p => r) v false
 Prove: (~p v q) & (q => r) & true => (p => r) v false
 First branch:
 Prove: ~p & (q => r) & true => (p => r) v false
 Prove: (q => r) & true => p v (p => r) v false
 Prove: (~q v r) & true => p v (p => r) v false
First branch:
 Prove: ~q & true => p v (p => r) v false
 Prove: true => q v p v (p => r) v false
 Prove: true => q v p v (~p v r) v false
 Prove: true => q v p v ~p v r v false
 Prove: p v true => q v p v r v false
     This is tautology
Second branch:
 Prove: r & true => p v (p => r) v false
 Prove: r & true => p v (~p v r) v false
 Prove: r & true => p v ~p v r v false
    This is tautology
```

Second branch:
Prove: q & (q => r) & true => (p => r) v false
Prove: q & (~q v r) & true => (p => r) v false
First branch:
Prove: ~q & q & true => (p => r) v false
Prove: q & true => q v (p => r) v false
This is tautology
Second branch:
Prove: r & q & true => (p => r) v false
Prove: r & q & true => (~p v r) v false
Prove: r & q & true => ~p v r v false
This is tautology
Formula is a theorem

The form of algorithm 2 listings is better arranged than that of algorithm 1. The protocol of proof is closely related to logical symbolics, it is not necessary to differentiate two meanings of the symbol => , it can be considered only as implication. Then it is evident, that given formula is transformed to form (of tautology)

X & A => X v B,

where X, A, B are arbitrary formulas. The front memeber is considered as a conjunction, the back one as a disjunction of certain expressions - subformulas of given formula.

III. METHOD OF ANALYTICAL TABLES

Method of analytical tables is founded on decomposition of formula to simpler components - subformulas of considered formula. The models of the decomposition are rules for construction of tables. The analytical table of the formula X is taken as a dyadic tree (graph), the nodes of which are occurences of the formulas, and which is constructed by following way - by the help of two-type rules:

- conjunctive of form $\dfrac{K}{\begin{array}{c}K1\\K2\end{array}}$ and disjunctive of form $\dfrac{D}{D1 \mid D2}$

The process of construction:

1. the root of the tree is formula X;
2. let formula Y be terminal node of the given tree
   - if there - on the way from X to Y - occurs a formula K, then any of formulas K1 or K2 as the only successor of node Y can be added - we usually add step by step firstly K1, secondly K2 (the tree in considered branch develops linearly);
   - if there - on the way from X to Y - occurs a formula D, then formula D1 can be added as left successor and D2 as right successor of formula Y (the tree in the node Y develops into two branches).

The branch of given tree is said to be closed, if it contains a formula and its negation. Analytical table (tree) is called to be closed, if every of its branches is closed. The proof of the formula X is then understood as a closed table for formula ~X. Such accepted proof seems to demonstrate that every branch of decomposition of formula ~X forms inconsistent set of formulas. That is why the formula ~X inconsistent, hence formula X is a tautology or theorem.

Decompositional rules, which may be used in above mentioned process, are according types:

- conjunctive rules with two successors

$$\dfrac{X \,\&\, Y}{\begin{array}{c}X\\Y\end{array}} \qquad \dfrac{\sim(X \vee Y)}{\begin{array}{c}\sim X\\\sim Y\end{array}} \qquad \dfrac{\sim(X \Rightarrow Y)}{\begin{array}{c}X\\\sim Y\end{array}}$$

- conjunctive rules with one successor

$$\dfrac{\sim\sim X}{X} \qquad \dfrac{X \Leftrightarrow Y}{(X \Rightarrow Y) \,\&\, (Y \Rightarrow X)} \qquad \dfrac{\sim(X \Leftrightarrow Y)}{\sim X \Leftrightarrow Y}$$

- disjunctive rules

$$\dfrac{\sim(X \,\&\, Y)}{\sim X \mid \sim Y} \qquad \dfrac{X \vee Y}{X \mid Y} \qquad \dfrac{X \Rightarrow Y}{\sim X \mid Y}$$

Example: proof of formula (p => q) => (~q => ~p)

1.	~((p => q) => (~q => ~p))		
2.	(p => q) & ~(~q => ~p)		(1)
3.	p => q		(2)
4.	~(~q => ~p)		(2)
5.	~p	q	(3)
6.	~q	~q	(4)
7.	~~p	~~p	(4)
8.	p	p	(7)

In this proof on the left there are lines numbered, on the right there is the source of formula, which occurs here, introduced by a line number. Both branches of proof are closed. In the left branch there is a contradiction between formulas in lines 5 and 7 or 5 and 8, in the right one there is a contradiction between formulas in lines 5 and 6. There is no need to continue in decomposition of given branch when contradiction appears. In this proof there are redundant the formulas of line 8 and in the right branch that of line 7. The proof can be shortened by the preferring of the conjunctive type rules applications.

IV. ALGORITHM TO METHOD OF ANALYTICAL TABLES IN PROLOG

Algorithm 3 is written in Prolog-8Ø, that is why here are some differences from algorithms 1 a 2. Especially, there differs priorities of "operations" - logical connectives, but only by numerical values. Usual convention of descending prioriry of sequence of conectives ~, &, v, => , <=> is respected. It appears in the proof protocol - the brackets are omitted always there, where the order of operations is given by implicit relationship.

Optimizing of proof construction is not applied, because it would make computation longer.

In proof protocol the branches are signed only as the first one and the second one. Corresponding assignment is realized on the principle of LIFO.

Algorithm works as follows - if it finds out the first branch, in which there is no contradiction, the computation is finished, because the formula cannot become a theorem.

Logic program of algorithm 3:

```
/* Operations - logical connectives */
:- op(210,xfy,<=>)                    /* equivalence */
:- op(180,xfy,=>)                     /* implication */
:- op(150,xfy,v)                      /* disjunction */
:- op(120,xfy,&)                      /* conjunction */
:- op(90,xfy,~)                       /* negation    */
/* Organisation of reading and proving of formula */
formula:- repeat,nl,nl,write('Formula:'),nl,
          read(F),(F==stop;theorem(~F),fail).
theorem(T):- nl,nl,write('Proof of inconsistency of formula:'),
             nl,write(T),nl,nl,write('Main branch:'),
             (seq([T],[T]),!,nl,nl,write('formula is theorem');
             nl,nl,write('formula is not theorem')).
/* Decomposition of formula and branching */
seq([X|Y],Z):- nl,write(X),fail.
/* Conjunctive rules */
seq([~~X1|X2],Y):- append([X1],Y,T),
                   append(X2,[X1],Z),!,
                   seq(Z,T).
seq([X1 & X2|X3],Y):- append([X1,X2],Y,T),
                      append(X3,[X1,X2],Z),!,
                      seq(Z,T).
seq([~(X1 v X2)|X3],Y):- append([~X1 & ~X2],Y,T),!,
                         seq([~X1 & ~X2|X3],T).
seq([~(X1 => X2)|X3],Y):- append([X1 & ~X2],Y,T),!,
                          seq([X1 & ~X2|X3],T).
seq([X1 <=> X2|X3],Y:- append([(X1 => X2) & (X2 => X1)],Y,T),!,
                       seq([(X1 => X2) & (X2 => X1)|X3], T).
seq([~(X1 <=> X2)|X3],Y:- append([~X1 <=> X2],Y,T),!,
                          seq([~X1 <=> X2|X3],T).
```

```
/* Disjunctive rules */
seq([~(X1 & X2)| X3],Y):- append([~X1],Y,T1),
                          append([~X2],Y,T2),!,
                          v1,seq([~X1|X3],T1),!,
                          v2,seq([~X2|X3],T2).
seq([X1 v X2|X3],Y):- append([X1],Y,T1),
                      append([X2],Y,T2),!,
                      v1,seq([X1|X3],T1),!,
                      v2,seq([X2|X3],T2).
seq([X1 => X2|X3],Y):- append([~X1],Y,T1),
                       append([X2],Y,T2),!,
                       v1,seq([~X1| X3],T1),!,
                       v2,seq([X2|X3],T2).
/* Atomic formula */
seq([_|X],Y):- seq(X,Y).
/* End of decomposition */
seq([ ],X):- scontr(X,X).
append([ ],L,L).
append([H|T],L[H|U]):- append(T,L,U).
/* Searching of contradiction in actual branch */
scontr([ ],_):- fail.
scontr([X|Y],Z):- (contr(X,Y),nl,write(´branch closed´));
                  scontr(Y,Z).
contr(X,[~X| _]).
contr(~X,[X|_]).
contr(X,[_|Y]):- contr(X,Y).
v1:- nl,nl,write(´1. branch´).
v2:- nl,nl,write(´2. branch´).
```

Examples of algorithm 3 performance:

Formula:

p => p.

Proof of inconsistency of formula:

~(p => p)

Main branch:
~(p => p)
p & ~p
p
~p
branch closed
formula is theorem

Formula:
p v p.
Proof of inconsistency of formula:
(p v ~p)
Main branch:
~(p v ~p)
~p & ~~p
~p
~~p
p
branch closed
formula is theorem

Formula:
~(p & ~p).
Proof of inconsistency of formula:
~~(p & ~p)
Main branch:
~~(p & ~p)
p & ~p
p
~p
branch closed
formula is theorem

Formula:
~p & p.
Proof of inconsistency of formula:
~(~p & p)

Main branch:
~(~p & p)
1. branch
~~p
p
formula is not theorem

Formula:
(p => q) & (q => r) => (p => r).
Proof of inconsistency of formula:
~((p => q) & (q => r) => (p => r))
Main branch:
~((p => q) & (q => r) => (p => r))
(p => q) & (q => r)
~(p => r)
p & ~r
p => q

1. branch	2. branch
~p	q
q => r	q => r
1. branch	1. branch
~q	~q
p	p
~r	~r
branch closed	branch closed
2. branch	2. branch
r	r
p	p
~r	~r
branch closed	branch closed

formula is theorem

V. COMPARISON OF ALGORITHMS

Let´s choose the law of implication transitivity as representative - it is more complicated formula, individual proofs are regardless of used algorithm approximately of the same length and in all proofs multiple branching is used.

Firstly we can assume that documentation of proofs at all algorithms is badly arranged as soon as the proof "length" overpasses screen range. It seems to be a serious didactic deffect, if uwer wants to understand more complicated proofs.

If we eliminate algorithm 1, which has not a character of logic program (see [2]), there are algorithms 2 and 3 left to evaluation. The length of the proof made by algorithm 2 will be usually little bit less than that of algorithm 3. Essential advantage of algorithm 3 is the fact that

- during realization the proof the formulas become more and more simple, so the proof is clearer than at the other algorithms,
- formulas are in usual syntactic form and so it is easy to find the reason of contradiction in actual branch (closed branch).

SOUHRN

METODY DOKAZOVÁNÍ TEORÉMŮ VÝROKOVÉHO POČTU V PROLOGU

JAN ŠTĚPÁN

V článku jsou popsány dva algoritmy důkazu teorémů výrokového počtu - Wangova metoda a metoda analytických tabulek. Wangova metoda je doložena dvěma programy v Prologu převzatými z [1] a [2]. Pro metodu analytických tabulek je předložen autorův program. Efektivnost programů je demonstrována na příkladech. Dále je diskutována praktická a didaktická hodnota uvedených metod a programů.

РЕЗЮМЕ

МЕТОДЫ ДОКАЗАТЕЛЬСТВА ТЕОРЕМОВ ПРОПОЗИЦИОНАЛЬНОГО ИСЧИСЛЕНИЯ В ПРОЛОГЕ

Я. ШТЕПАН

В этой статье описаны два алгорифма доказательства теоремов пропозиционального исчисления - метод Ванга и метод аналитичных таблиц. Метод Ванга является основанием двух программ, которые приняты из /1/ и /2/. Для метода аналитичных таблиц здесь показана программа автора. Действенность этих программ показана на примерах. Далее здесь обсуждена практическая и учебная ценность этих алгорифмов и программ.

REFERENCES

[1] C o e l h o, H. - C o t t a, J.C. - P e r e i r a, L.M.: How to solve it with Prolog. Lisboa, LNEC 1985.

[2] C o e l h o, H. - C o t t a, J.C.: Prolog by example. Springer-Verlag 1988.

[3] S m u l l y a n, R.M.: First order logic. Bratislava, ALFA 1979.

Author´s address:

RNDr.PhDr.Jan Štěpán, CSc.

katedra výpočetní techniky
přírodovědecké fakulty UP

771 46 Olomouc

Czechoslovakia