PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 8 /72

AUTOMATIC PROOFS FOR THEOREMS ON PREDICATE CALCULUS

by

Sueli M. Santos
and
Marília R. Millan

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

AUTOMATIC PROOFS FOR THEOREMS ON PREDICATE CALCULUS

Suelî M. Santos

and

Marîlîa R. Mîllan

Assistant Professors
Computer Science Department
PUC / RJ

This paper was published in the International Congress of Cybernetics and Systems - Oxford - August 28 - September 1st 1972

Series Editor: Prof. A. L. Furtado November /1972

ABSTRACT

The attempt to use a computer to prove propositions of formalized theories has characterized in the last few years a special field of Computer Science, artificial intelligence. Our aim in this paper is to prove theorems of propositional and pure predicate calculus in a automatic way using the programming language SNOBOL-4. We choose this language because it is appropriate for handling structures such as arrays and trees.

The formal system we use is the one developed by M. Smullyan. This system constructs proofs for expressions in the propositional, as well as in the predicate calculus, in the form of trees, or analytic tableaux as Smullyan calls them.

The proofs for propositional calculus being straightforward, have the advantage of being very simple and elegant. It is even more striking that those properties are also present in proofs of expressions in the predicate calculus for which we cannot provide a decision procedure. Then it is not a trivial task to find an automatic way of constructing proofs for formulas in a subset of its theorems.

1. INTRODUCTION

Most of the recent research in the area of automatic theorem proving has been done using Resolution based theorem provers[3,4] and investigating new heuristics to improve this method. Attempts at introducing different methods to prove theorems on a computer also seem reasonable. This paper reports on one such attempt.

Our theorem proving method is based upon the logical system of "analytic tableaux", developed by R. Smullyan[1]. It is a variant of the semantic tableaux of Beth[2]. Smullyan's tableaux are simpler than Beth's semantic tableaux because they utilize only one tree instead of two. As it is a very simple and natural method of proving theorems automatically, it seems to us to be a good method for a first approach to theorem proving. Its simplicity is a more evident and immediate advantage of the method, although we hope that, with further research, we may find applications, in question-answering, automatic programming, and so on.

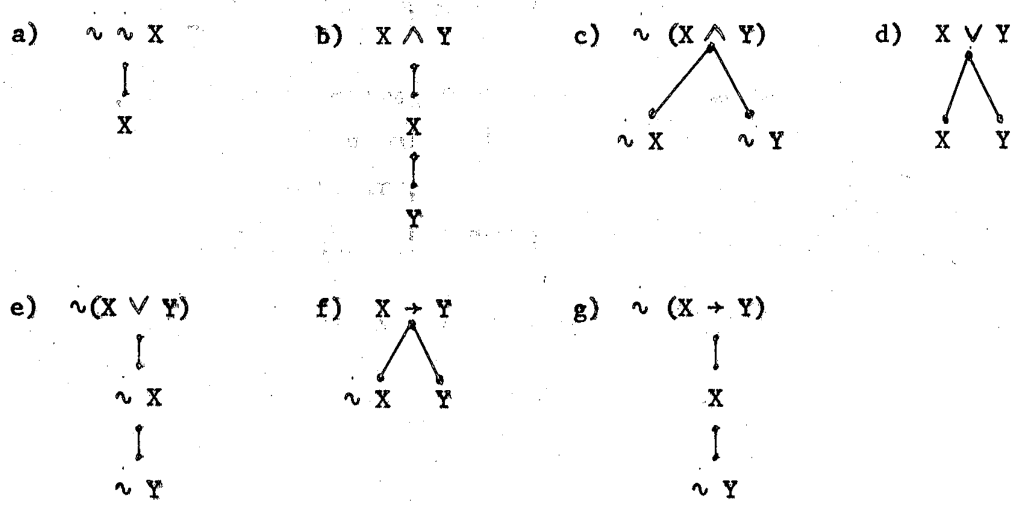
The logical system is presented in the next section. Proofs of the consistency and completeness of the system are omitted since they are found in the reference[1]. In (3), we discuss programmable algorithms for proving theorems in the propositional and predicate calculus. The algorithms are presented in appendix I and II. In the summary we only mention the design of the actual implementation using the SNOBOL 4 programming language[6].

2. SMULLYAN'S LOGICAL SYSTEM

In the ensuing discussion we use the following vocabulary: propositional connectives: $\sim$, $\vee$, $\wedge$, $\rightarrow$; quantifiers: $\forall$, $\exists$; auxiliary symbols: {, } , [,] , (,) .
We have an infinite set of: variables: x, y, z,...; predicates: P,Q,R,...; variables for formulas: A,B,C,...; parameters (constants): a,b,c,...

Formulas and predicates are defined in the usual way, as well as the notion of binary tree. Tree vocabulary, such as, node , leaf and path from one node to another, is used without further explanation (see, for example, Knuth[5]). We borrow Smullyan's definition of branch to be a path whose last node is a leaf or a path that is infinite.

The theorems are proved using a refutation method. If we want to prove that a formula X in the propositional calculus is a theorem, we construct a binary tree whose root is labelled by the negation of X. Other nodes of the tree are generated by the following rules:

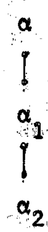


The formulas in the above rules are of two types:

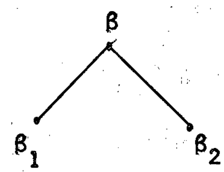
(i) those whose roots have only one immediate sucessor ((a), (b), (e) and (g)) are called <u>α - formulas</u>; (ii) those whose roots have two successors ((c), (d) and (f)) are called <u>β - formulas</u>.

These rules can be represented in α synthetic way as:

(i)



(ii)



α_1 and α_2 (β_1 and β_2) will be called <u>components</u> of an α - formula (β - formula).

An <u>analytic tableau</u> for the formula X is a binary tree T whose nodes are labelled by formulas in the theory. Nodes and their labels are generated as follows: Step (1) - Label the root of T by X ; Step (2) - Let Y be the label of a leaf n of T. We may extend T by one of the two operations.

(i) If an α - formula occurs as the label of a node in the path from the root to the leaf n, we adjoin as its immediate successor, a node labelled by α_1 (or α_2) and then to this new node adjoin its immediate successor labelled by α_2 (or α_1);

(ii) If a β - formula occurs as the label of a node in the path from the root to the leaf n, we simultaneously adjoin the left successor of n labelled by β_1 and the titht successor of n labelled by β_2.

Operation (i) ((ii)) will be called <u>development</u> of an α - formula (β - formula).

A <u>branch</u> of a tableau T is said to be <u>closed</u> if and only if it contains two nodes such that one is labelled by a formula and the other by its negation. T is called <u>closed</u> if and only if every branch of T is closed. By a <u>proof of a formula X</u> is meant a closed tableau whose root is labelled by the negation of X. A branch Θ of a tableau is said to be <u>complete</u> if for every α - formula which occurs in Θ, both α_1 and α_2 occur in Θ, and for every β - formula which occurs in Θ, at least one of β_1, β_2 occurs in Θ. A tableau T is <u>completed</u> if every branch of T is either complete or closed.

It has been proven by Smullyan that the system is complete, consistent and decidable[1]. The above analysis is sufficient for proving formulas in the propositional calculus as in the simple example given in Figure 1.

(0) $\sim [(p \rightarrow q) \rightarrow ((p \wedge r) \rightarrow q)]$

(1) $p \rightarrow q$ from (0), by rule (g)

(2) $\sim ((p \wedge r) \rightarrow q)$ from (0), by rule (g)

(3) $p \wedge r$ from (2), by rule (g)

(4) $\sim q$ from (2), by rule (g)

(5) p from (3), by rule (b)

(6) r

(7) $\sim p$ from (1), by rule (f)   (8) q from (1), by rule (f)

Figure 1. The tableau for a formula in the propositional calculus.

For the predicate calculus we add the rules:

j) $(\forall x)\ A$ | A^x_a

k) $\sim(\exists x)\ A$ | $\sim A^x_a$, where a is any parameter

l) $(\exists x)\ A$ | A^x_a

m) $(\forall x)\ A$ | $\sim A^x_a$, provided that a has not occurred in the labels of any nodes in the tree so far constructed (i.e., a is new)

A^x_a denotes the result of the substitution of the parameter a for all free occurrences of x in A. The constants a,b,c,... are elements of a universe or a domain U that is by hypothesis the domain of an interpretation of the formulas of the predicate calculus.

Universally quantified formulas ((j) and (k)) are called <u>γ - formulas</u>; existencially quantified formulas ((l) and (m)) are called <u>δ - formulas</u>. The above rules can be represented in a succint form:

(i) γ
|
γ(a), where a is any parameter

(ii) δ
|
δ(a), provided that a is a new parameter.

An analytic tableau for a formula X in the predicate calculus can be defined as a simple extension of the two steps given earlier for the propositional case. To Step (2), we append: (iii) If a γ - formula occurs as the label of a node in the path from the root the leaf n, we adjoin as its only immediate successor a node labelled by γ(a), where a is any parameter; (iv) If a δ - formula occurs as the label of a node in the path from the root to the leaf n, we adjoin as its only immediate successor, a node labelled by δ(a), where a is a new parameter.

For any formula X, X the label of a node on a branch Θ of T, define Θ to be <u>fulfilled</u> on Θ if either: (i) X is an α - formula and α_1, α_2 are both labels of nodes in Θ; (ii) X is a β - formula and at least one of β_1, β_2 is a label of a node in Θ; (iii) X is a γ-formula and, for every $a \in U$, γ(a) is a label of a node in Θ; (iv) X is a δ - formula and, for at least one $a \in U$, δ(a) is a label of a node in Θ.

By *finished tableau* is meant a tableau which is either infinite[1], or is finite but all its formulas are fulfilled. This is a simple modification of Smullyan's definition of finished systematic tableau.

The notions of closed branch, closed tableau and proof for a formula are the same as in the case of propositional calculus. The system for predicate calculus was also proven by Smullyan to be complet and consistent, but, of course, it is not decidable.

In the propositional calculus, if we have a complete, but open tableau for a formula X, we can conclude that the negation of X is not a theorem. We can always complete the tableau so the method constitutes a decision procedure for the propositional calculus.

In the predicate calculus, we may have an open tableau which is not fulfilled. As there may be no way to finish the tableau in a finite number of steps, we cannot conclude that either X or the negation of X is a theorem. When X is satisfiable in a finite domain, we can conclude that the negation of X is not a theorem. In such a case, the method may also provide a model for X.

Take as an example to illustrate the use of the rules in the predicate calculus a tableau for the negation of the formula $[(\forall x)(\forall y)(Px \vee Qy)] \rightarrow [(\forall x)\ Px \vee (\forall y)\ Qy]$ as appears in Figure 2.

(0) $\sim\{[(\forall x)(\forall y)(Px \vee Qy)] \rightarrow [(\forall x)Px \vee (\forall y)Qy]\}$

(1) $(\forall x)(\forall y)(Px \vee Qy)$ from (0), by rule (g)

(2) $\sim((\forall x)\ Px \vee (\forall y)Qy))$ from (0), by rule (g)

(3) $\sim (\forall x)Px$ from (2), by rule (e)

(4) $\sim (\forall y)Qy$ from (2), by rule (e)

(5) $\sim P\ a$ from (3), by rule (m)

(6) $\sim Q\ b$ from (4), by rule (m)

(7) $(\forall y)(Pa \vee Qy)$ from (1), by rule (j)

(8) $Pa \vee Qb$ from (7), by rule (j)

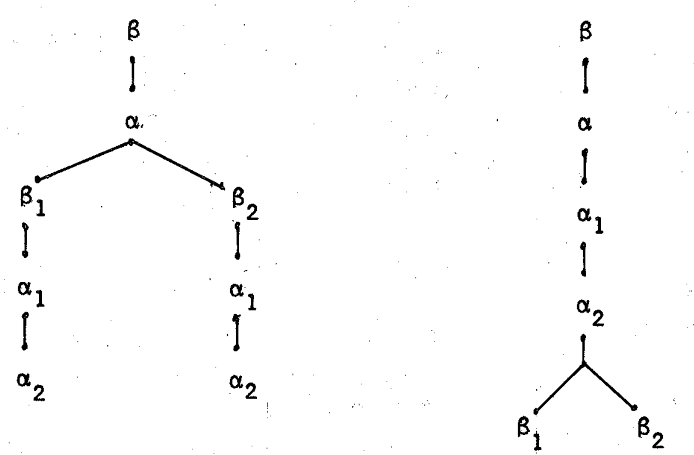
(9) Pa from (8), by rule (d)
X

(10) Qb from (8), by rule (d)
X

Figure 2. The tableau for a formula in the predicate calculus.

3. THE ALGORITHM

Smullyan[1] suggests a systematic way for constructing the analytic tableau for a formula in the propositional or predicate calculus. In the propositional case, he suggests that all possible developments of nodes labelled by α - formulas should be carried our first. This is a way of avoiding repetitions in tree of the components of α - formulas. The difference is illustrated in Figure 3. The suggestion helps to make the automatic construction of the tree more efficient.



(a) Development of a β-formula without using Smullyan's suggestion

(b) Using Smullyan's suggestion

Figure 3

In developing a programmable algorithm, we decided on the following order in which to examine the nodes: begin by leaf of the left most path, working along the path towards the root, and developing all α - formulas before β - formulas. Our algorithm references the following program variables: LEAF - the set of the leaves of the tree; OR_i - if i counsts the number of β-formulas developed along a single path, then

OR_i is the set of nodes labelled by β - formulas occurring in the path between the i th and the (i-1)st β - formula development. That is OR_i is the set of β - formulas along the path marked in the diagram of the Figure 4. $MORE_i$ - set of nodes labelled by components of the ith β - formula developed; ADD - set of undeveloped nodes; k - a node in T; Φ - the root of T; ℓ(k) - label of a node k; left(k)(right(k)) - left(right) successor of k; top(name of set) - the last item added to the set:

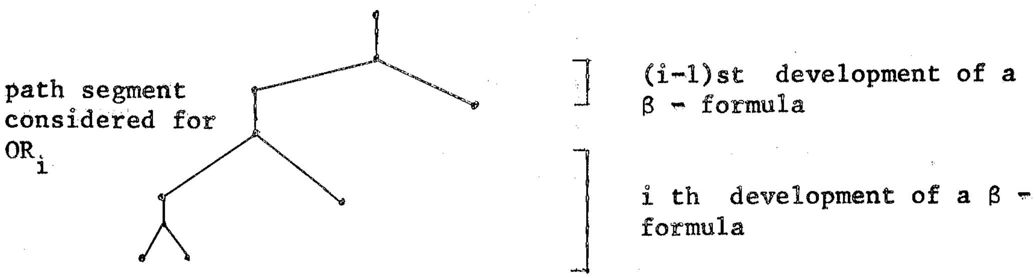


Figure 4. Order of development of β-formulas.

Our algorithm and the description of the function it uses are given in Appendix I using an ALGOL - like language. The condition of halting for the algorithm is given by one of two responses: <u>X is a theorem</u> or <u>X is not a theorem</u>.

A formula in conjunctive normal form that is typically troublesome for some mechanical theorem provers is: $(p \vee q) \wedge (p \vee \sim q) \wedge (\sim p \vee q) \wedge (\sim p \vee \sim q)$. A proof for this formula is shown in Figure 5. In this case, we had to carry out an exhaustive search. The size of the tree increases rapidly due to consecutive applications of rules to β - formulas. However, if we do the type of analysis we proposed in the previous section we need store only a fraction of the nodes (those on open branches, in each step).

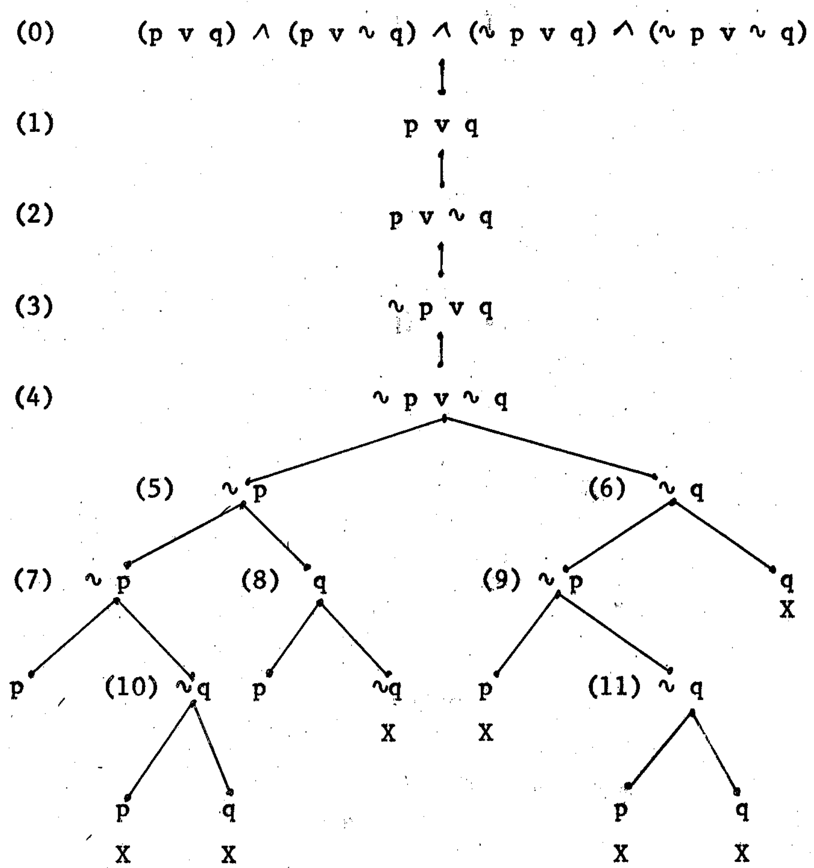


Figure 5. Tableau for a formula in conjuctive normal form.

In more complex expressions the storage requirements may be worse. We need to find heuristics to control the explosive nature of the exhaustive search. Initially, we are using a preprocessor to transform the conjuctive (or disjunctive) normal forms into equivalent forms (such as a conditional) and testing its advantages. Preferably, we would like to habe better heuristics for ordering the development of the formulas.

For the predicate calculus Smullyan suggest developing α - formulas first- then δ - formulas, β - formulas and, finally, γ - formulas. The delicate problem here is the one of instantiation of quantified variables. For the δ - formulas, if we strictly obey the proviso of rule (ℓ) and (m) in Section 2, we have to use a new parameter for each δ - formula. Smullyan[1] suggests a liberalization of this restriction for oneplace predicates only, allowing instantiation with a parameter already used. As he proves this is permissable, we included it in our algorithm. We suggest a further liberalization for cases of several-place predicates. The strategy would be to instantiate δ - formulas by constants already used in the instantiation γ - formulas. If a contradiction related to such a "liberal" instantiation is found, we repeat the instantitation of the δ - formula, this time using an altogether new constant. But, if there is no contradiction, we have a model for the negation of the formula. This case is illustrated by the following example. Suppose our problem is to find out whether or not $\sim[(\forall y)(\exists x)\, Rxy \wedge \sim (\exists x)\, Rxx]$ is a theorem. We construct the tableau in Figure 6.

(0) $\sim\{\sim [(\forall y)\ (\exists x)\ Rxy \land \sim (\exists x)\ Rxx]\}$

(1) $(\forall y)(\exists x)\ Rxy \land \sim (\exists x)\ Rxx$

(2) $(\forall y)(\exists x)\ Rxy$

(3) $\sim (\exists x)\ Rxx$

(4) $(\exists x)\ Rxa$

(5) Rba

(6) $\sim Raa$

(7) $(\exists x)\ Rxb$

(8) Rab

(9) $\sim Rbb$

Figure 6. A liberalized tableau.

At level (7) of the tree we have an example of the liberalization for some - place predicates. If we used a new constant for instantiating ($(\exists x)$ Rxb) we would continue introducing new constants without finding a finite model for the formula. We cannot solve our problem. This liberalization is not included in the algorithm presented below. If it were, we would be able to find a model for the example (Figure 6).

The algorithm for the predicate calculus requires further variables. We have

ANY – set of undeveloped nodes labelled by γ – formulas; SOME – set of undeveloped node labelled by δ – formulas; AVAIL – a finite set of available constants; SOMELIST(ANYLIST) – set of constants which were used in instantiating δ – formulas (γ – formulas).

To guarantee that an open branch in which γ – formula occurs as the label of a node, is fulfilled, we have to instantiate the γ – formula using all constants (without repetition) that occur in SOMELIST. To keep account of which constants have already been used , the algorithm references: LIST(k) – set of constants which were used for instantiating a γ – formula (δ–formula) that labels node k. For a δ – formula the set is unitary and it is useful to a heuristic function for choosing the constant to the instantiation of a γ – formula: MOREANY – set of node labelled by γ – formulas that must be instantiated, using constant not in LIST(k).

The algorithm with the description of the functions it uses is given in Appendix II.

The CHOICE function used in the algorithm is very important from the point of view of the efficiency of the method. It is our intention to develop further the function by introducing new heuristics to find instantiations for γ – formulas that will let us close (or fulfill) a branch, when possible, in an efficient way. For now, we present only a rough outline of a CHOICE function. Two sets are needed: AIDLIST and TEMPLIST. AIDLIST stores the constants that were used in instantiating δ – formulas occurring along a given path. This is useful in picking, from SOMELIST, a good constant for instantiating the next γ – formula in that path. TEMPLIST is just temporary storage for constants picked from SOMELIST and which do not appear in AIDLIST. This may be used at a later time.

Since the predicate calculus is undecidable besides those described for the propositional case we have a third halt condition in our algorithm. In the present situation, if the AVAIL set is exhausted before either all the branches are closed or fulfilled then the algorithm halts with no decision. One possible improvement is the algorithm's recognition of an infinite loop condition implied by the necessity of creating new constants for the instantiation of δ - formulas. This occurs for formulas that are neither theorems nor are their negations satisfiable in any finite domain.

It is obvious that the algorithm without the last halt condition does not compromise the completeness of Smullyan's system. The last halt condition imposes an upper bound on the size of AVAIL set. It may be the case that we need more than this finite number of parameters in order to close the tableau for the negation of a theorem. This limitation as well as the one implied by an upper bound on the length of the formulas we work with, is related to the implementation of the algorithm. However, these limitations do not compromise Smullyan's theoretical results.

4. SUMMARY

We are using the SNOBOL-4 programming language for two main reasons: first, it easily let us represent the tree (tableau) for formulas by means of what is called "programmer - defined data types". Second the pattern features provides a synthetic manner in which to program the searches for the components of formulas, searches that are necessary for the implementation of the algorithm. Smullyan's logical system leads itself to automation. It is an especially fruitful area for investigating heuristics for solving problems of instantiation of quantified variables.

Appendix I - The algorithm for the propositional calculus.

```
Step 1  -  comment: initialization
           T ← φ; ℓ(Φ) ← ∿ X; i ← 0; LEAF ← ADD ← OR_i ← MORE_i ← φ;
           LEAF ← LEAF  {Φ}; k ← Φ;

Step 2  -  if ℓ(k)  is atomic then go to step 3;
           else if ℓ(k) is a β - formula then begin OR_i ← OR_i ∪ {k};
                else go to Step 4;

Step 3  -  if  ADD ≠ φ  then begin  k ← top(ADD);ADD←ADD-{k};
           go to Step 2; end;
           else go to Step 5;

Step 4  -  comment: development of the  α - formula ℓ(k)
           if LEAF = φ then X is a theorem ;
           else for each j such that  j ε LEAF;
                do if k is in the path from  Φ to j then
                      begin;
                      ℓ(left(j)) ← α_1; ℓ(left(left(j)) ← α_2(*);LEAF←LEAF-{j};
                      if CHECK(left(j)) fails then
                         if CHECK(left(left(j))) fails then
                            begin LEAF ← LEAF ∪ {left(left(j))};
                            ADD ← ADD ∪ {left(j), left(left(j))}; end;
                      end;
                end; go to Step 3;

Step 5  -  if  OR_i ≠ φ  then go to Step 6;
           else if  MORE_i = φ  then if i = 0  then go to Step 7;
                                     else begin i ← i-1; go to Step 5; end;
                else begin k ← top(MORE_i);MORE_i← MORE_i-{k};go to Step 2;end;
```

(*) In the case of rule (a), section 2 we do not need to have left(left(j)).

```
Step 6  -  comment: development of the  β - formula ℓ(k)
           if  LEAF = φ  then  X is a theorem;
           else begin  k ← top(OR_i); OR_i ← OR_i - {k}; i ← i+1;
           for each  j  such that  j ε LEAF;
           do if k is in the path from  Φ  to j then
                 begin ℓ(left(j)) ← β_1;ℓ(right(j)) ← β_2; LEAF∪LEAF-{j};
                 if CHECK(right(j)) fails then
                    begin MORE_i← MORE_i ∪ right(j)};LEAF←LEAF∪{right(j)};end;
                 else if CHECK(left(j)) fails then
                         begin MORE_i←MORE_i∪{left(j)};LEAF←LEAF∪{left(j)};end;
                       else;
                 end;
              else;
           end; go to Step 3; end;
Step 7  -  if LEAF = φ  then X is a theorem; else  X is not a theorem;
  CHECK is the following boolean function:
  BOOLEAN FUNCTION CHECK(m)
         comment: m is a node of T.
         if both ℓ(m) and the complement of ℓ(m) occurs in the
            path from  Φ to m then SUCCESS; else FAILURE;
```

Appendix II - The algorithm for the predicate calculus;

```
Step 1 - comment: initialization
         T ← φ; LEAF ← ADD ← OR_i← MORE_i← ANY ← SOME ← φ; ℓ(Φ) ← ∿ X;
         i = 0; LEAF ← LEAF ∪ {Φ}; k ← Φ;
```

```
Step 2 - if ℓ(k) is atomic then go to Step 3;
         else if ℓ(k) is a γ - formula then begin ANY ← ANY ∪ {k};
                                              go to Step 3; end;
              else if ℓ(k) is a β - formula then begin OR_i ← OR_i ∪ {k};
                                                   go to Step 3; end;
                   else if ℓ(k) is a δ-formula then begin SOME←SOME∪{k};
                                                     go to Step 3; end;
                        else go to Step 4;

Step 3 - if ADD ≠ φ then begin k ← top(ADD); ADD ← ADD - {k};
                        go to Step 2; end;
         else go to Step 8;

Step 4 - comment: development of the α - formula ℓ(k),
                  (same as Step 4 in Appendix I)

Step 5 - (same as Step 5 in Appendix I)

Step 6 - comment: development of the β - formula ℓ(k)
                  (same as Step 6 in Appendix I)

Step 7 - if ANY ≠ φ then go to Step 10;
         else if MOREANY = φ then go to Step 14;
              else go to Step 11;

Step 8 - if SOME ≠ φ then go to Step 9;
         else go to Step 5;

Step 9 - comment: development of the δ - formula ℓ(k).
         if LEAF = φ then X is a theorem;
         else begin k ← top(SOME); SOME ← SOME -{k}; δ(a)←SOMECOMPONENT(k);
              for each j such that j ε LEAF;
              do if k is in the path from Φ to j then
                    begin ℓ(left(j)) ← δ(a); LEAF ← LEAF - {j};
                    if CHECK(left(j)) fails then
                    begin LEAF ← LEAF∪{left(j)}; ADD←ADD∪{left(j)}; end;
                 end;
              end; go to Step 3;
         end;
```

```
Step 10 - comment: development of the γ - formula ℓ(k)
          if LEAF = ϕ then X is a theorem;
          else begin k←top(ANY); ANY←ANY - {k}; MOREANY→MOREANY∪{k};
               γ(a) ← ANYCOMPONENT(k);
               for each j such that j ε LEAF;
               do if k is in the path from Φ to j then
                    begin (left(j)) ← γ(a); LEAF ← LEAF - {j};
                    if CHECK(left(j)) fails then
                       begin ADD ← ADD∪{left(j)};LEAF←LEAF∪{left(j)};end;
                    else if ANY ≠ ϕ then go to Step 10;
                       else;
                    end;
                  else;
               end; go to Step 3;
          end;

Step 11 - if SOMELIST ≠ ϕ then go to Step 12;
          else go to Step 14;
Step 12 - if LEAF = ϕ then X is a theorem;
          else for each k such that k ε MOREANY;
          do if CHOICE(k,a,SOMELIST) succeeds then go to Step 13;
             else if ANYLIST ≠ ϕ and top(ANYLIST) ∉ LIST(k) then
                    begin a ← top(ANYLIST); go to Step 13; end;
                 else;
          end; X is not a theorem;
```

```
Step 13 - LIST(k) ← LIST(k)∪{a}; x←BOUNDVAR(k); s←SCOPE(k);γ(a)←REPLACE(s,x,a);
          for each j such that j ε LEAF;
          do if k is in the path from Φ to j then
                begin ℓ(left(j)) ← γ(a); LEAF ← LEAF - {j};
                if CHECK(left(j)) fails then
                  begin LEAF←LEAF∪{left(j)}; ADD←ADD ∪ {left(j)}; end;
                end;
          end; go to Step 3;
Step 14 - if LEAF = ϕ  then  X is a theorem;
          else  X is not a theorem;
```

Functions (ii), (iii), (iv) below are very simple. So we are not going to describe them formally;

(i) FUNCTION CHECK(m) - see Appendix I.

(ii) FUNCTION BOUNDVAR(m) - returns as value the variable bound by the most external quantifier in ℓ(m).

(iii) FUNCTION SCOPE(m) - returns as value the scope of the most external quantifier in ℓ(m).

(iv) FUNCTION REPLACE(f,x,a) - returns as value the formula which is the result of the substitution of parameter a for all free occurrences of x in the formula f.

(v) FUNCTION SOMECOMPONENTS(m)

comment: m is a node labelled by a δ - formula; the function returns as value the component of ℓ(m).

```
Step 1 - comment: choice of a parameter for instantiating ℓ(m) using li-
         beralization.
         x ← BOUNDVAR(m); s ← SCOPE(m);
```

```
      if ANYLIST = φ then go to Step 2;
      else begin  a ← top(ANYLIST) ;
           if a ε SOMELIST or a occurs in s then go to Step 2;
           else if exists b, b ε SOMELIST and b occurs in s then go to Step2;
                else go to Step 3;
           end;

Step 2 - if AVAIL = φ then no decision;
      else begin a ← top(AVAIL); AVAIL ← AVAIL - {a}; end;

Step 3 - SOMECOMPONENT ← REPLACE(s,x,a); SOMELIST ← SOMELIST∪{a};
      LIST(m) ← LIST(m)∪ {a};
```

(vi) FUNCTION ANYCOMPONENT(m)

comment: m is a node whose label is a γ - formula; the function returns as value the component of ℓ(m).

```
Step 1 - BOUNDVAR(m); s ← SCOPE(m);
      if SOMELIST = φ  then
         if ANYLIST = φ  then if AVAIL = φ then no decision;
                              else begin a←top(AVAIL);AVAIL←AVAIL -{a};
                                   ANYLIST←ANYLIST∪{a}; go to Step 2;
                                   end;
         else begin a ← top(ANYLIST); go to Step 2; end;
      else CHOICE(m,a,SOMELIST);

Step 2 - ANYCOMPONENT ← REPLACE(s,x,a); LIST(m) ← LIST(m)∪ {a};
```

(vii) BOOLEAN PROCEDURE CHOICE (m,b,X)

comment: m is a node labelled by a γ-formula; X is a set of parameters; b is an element of X; the function returns SUCCESS or FAILURE. In the first case also returns b as a value.

```
Step 1 - comment: initialization
      TEMPLIST ← φ;
```

```
Step 2 - for each b such that b ε X;
        do if b ∉ LIST(m) then
           begin for each j such that j ε LEAF;
                do if m is in the path from Φ to j then
                      begin AIDLIST ← φ;
                      for each n such that ℓ(n) is a δ - formula
                      and n is in the path from m to j;
                      do AIDLIST ← AIDLIST ∪ LIST(n) ; end;
                         if b ε AIDLIST then success; b is the choice;
                end;
           TEMPLIST ← TEMPLIST {b};
           end;
        end;

Step 3 - if TEMPLIST = φ then FAILURE; else go to Step 4;

Step 4 - for each b such that b ε TEMPLIST;
        do for each j such that j ε LEAF;
           do if m is in the path from Φ to j then
                begin AIDLIST ← φ;
                for each n such that ℓ(n) is a  δ - formula
                and n is in the path from Φ to j;
                do AIDLIST ← AIDLIST ∪ LIST(n); end;
                   if b ε AIDLIST then SUCCESS; b is the choice;
           end;
        end;

Step 5 - b ← top(TEMPLIST); SUCCESS; b is the choice;
```


Acknowledgments

The authors express their gratitude do Adele Jean Goldberg who helped with valious suggestions in the outline of the algorithms , read the manuscript and suggested numerous improvements.

BIBLIOGRAFIA

1. R.M. Smullyan - First-Order Logic, Springer-Verlag, New York INC.1968.

2. E.W. Beth - The Foundations of Mathematics, North Holland, 1959.

3. J.A. Robinson - "A Machine - oriented Logic based on the resolution principle", J. ASS. comput. Mach., 12, pp. 23-41, 1965.

4. D. Luckham - "The Resolution Principle in Theorem Proving", Machine Intelligence I, pp. 47-61, Edinburgh: Oliver & Boyd, 1967.

5. D. E. Knuth - The Art of Computer Programming, Vol. I, Addison-Wesley Publishing Company, 1969.

6. R. E. Griswold; J. F. Poage, I.P. Polonsky - The SNOBOL 4 Programming Language, Prentice-Hall, 1971.