RA

stichting
mathematisch
centrum

REKENAFDELING NR 22/71 NOVEMBER

RA

G.H.A. KOK, J.M. VAN VAALEN

AN AUTOMATIC THEOREM-PROVER

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

Table of contents

	page
Introduction	1
1. Names and notions	2
1.1. Clauses	2
1.2. Example	3
2. Various proof procedures using the resolution rule	5
2.1. Inference rules and completeness	5
2.2. Search strategies and completeness	10
2.3. Edit strategies	12
3. The program	14
3.1. Description	14
3.2. The text of the ALGOL 60 program	19
3.3. List of error messages	35
3.4. Examples	35
4. Possible extensions to the program	39

Introduction

In the course of the seminar on automatic theorem-proving, which took place at the Mathematical Centre from October '70 till July '71, we made a theorem-proving-program. In this report we briefly review the theory and publish the program, together with some results. The program can be used to prove theorems expressed in the first order predicate calculus. We assume that the reader is familiar with this subject and also that he has some knowledge of model theory. For both subjects we refer to Mendelson [1].

Reference

[1] Elliot Mendelson, Introduction to mathematical logic.
D. van Nostrand Company, Inc. Princeton, 1964.

1. Names and notions

Let some axioms $A_1, \ldots, A_n$ and an alleged theorem B be given. We want to get an answer to the question: does B follow from $A_1 \wedge A_2 \wedge \ldots \wedge A_n$. We try to settle this by showing that $\neg(A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B)$ is a contradiction, or, equivalently, that $(A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge \neg B)$ is unsatisfiable (i.e. false for every interpretation).
The formulae have to be given in a special subset of the predicate calculus, to wit as a conjunction of clauses.

1.1. Clauses

A clause is a disjunction of literals. A literal is a predicate (also called: atomic formula, atom), possibly preceded by a negation symbol. It can be proved [1], [2] that for every formula of the predicate calculus there exists a conjunction of clauses, preceded by a universal quantifier for each variable that occurs in that conjunction, which is true iff that formula is.

We now give an outline of the construction of that conjunction of clauses. Let the original formula be F.

The first step is rewrite F in prenex normal form [3]. That means, finding a formula F_1 which holds iff F holds, and, moreover, has the structure $Q_1x_1\ Q_2x_2, \ldots, Q_nx_n\ A$, in which $Q_1, \ldots, Q_n$ are quantifiers, A is a formula which contains no quantifiers and $x_1, \ldots, x_n$ are all the variables that occur on A.

The second step consists of rewriting A as a conjunction of disjunctions of literals.

The third step is to eliminate the existential quantifiers from the sequence $Q_1x_1, \ldots, Q_nx_n$. This is achieved in the following way:

Let i be the smallest index such that Q_i is an existential quantifier. If there is none, the third step is finished; else we introduce a Skolem function [2] with parameters $x_1, \ldots, x_{i-1}$ and substitute it for every occurrence of x_i; if i = 1 then the Skolem function has no parameters. Now we delete Q_ix_i from the prenex, and start again.

As every clause is now always preceded by a universal quantifier for each variable we need no longer write these quantifiers but simply remember their implied presence.

1.2. Example

Say we have a set V and a multiplication operator *, and the following axioms:

axiom 1:	$\forall x,y\ \exists z$	$x*y = z$	(product in V),
axiom 2:	$\forall x,y\ \exists z$	$z*x = y$	(left solution),
axiom 3:	$\forall x,y\ \exists z$	$x*z = y$	(right solution),
axiom 4:	$\forall x,y,u$	$(x*y)*u = x*(y*u)$	(associativity),
theorem:	$\exists x \forall y$	$y*x = y$	(identity element).

In order to translate this into predicate calculus we introduce the predicate $P(x,y,z)$, which can be interpreted as $x*y = z$.

axiom 1:	$\forall x,y\ \exists z$	$P(x,y,z)$,
axiom 2:	$\forall x,y\ \exists z$	$P(z,x,y)$,
axiom 3:	$\forall x,y\ \exists z$	$P(x,z,y)$,
axiom 4:	$\forall x,y,z,u,v,w$	$(P(x,y,z) \wedge P(y,u,v) \wedge P(z,u,w) \rightarrow P(x,v,w)) \wedge$ $(P(x,y,z) \wedge P(y,u,v) \wedge P(x,v,w) \rightarrow P(z,u,w))$,
negated theorem:	$\forall x\ \exists y$	$\neg P(y,x,y)$.

If we now perform our three steps, where F is taken to be the conjunction of the above axioms and negated theorem, we obtain the following result (which, by convention, is denoted as a set of clauses rather than as the conjunction of these):

axiom 1 : $P(x,y,f(x,y))$,

axiom 2 : $P(g(x,y),x,y)$,

axiom 3 : $P(x,h(x,y),y)$,

axiom 4a: $\neg P(x,y,z) \vee \neg P(y,u,v) \vee \neg P(z,u,w) \vee P(x,v,w)$,

axiom 4b: $\neg P(x,y,z) \vee \neg P(y,u,v) \vee \neg P(x,v,w) \vee P(z,u,w)$,

negated theorem : $\neg P(j(x),x,j(x))$.

In this report we shall sometimes consider the clauses as a set of literals; an important role is played by the "null clause" which is the empty set of literals, denoted by $\square$.

References

[1] M. Davis and H. Putnam, A computing procedure for quantification theory, J. Assoc. Comp. Mach. 7 (1960), 201-215.

[2] M. Davis, Eliminating the irrelevant from mechanical proofs, Proc. Sympos. Appl. Math., Vol. 18, Amer. Math. Soc., Providence, R.I., 1963, 15-30.

[3] Elliot Mendelson, Introduction to mathematical logic, D. van Nostrand, Inc. Princeton, 1964.

2. Various proof procedures using the resolution rule

A proof procedure (T,Σ) consists of an inference system T, which determines a search space, and a search strategy Σ, which provides the way of searching this space.
An inference system T can be complete or incomplete; similarly, a search strategy Σ for a given complete T may or may not be complete for obtaining proofs constructable in T [1],[2].

2.1. Inference rules and completeness

In resolution theory, T consists in general of one rule of inference: the resolution rule or some refinement of this rule [3], [4].
First we have to explain the so-called unification algorithm. Two literals k and l are called unifiable if there exists a substitution τ such that $k\tau = l\tau$; $k\tau$ is then an instance of both k and l.
If there is such a substitution then there is a most general substitution σ such that, if τ is a substitution such that $k\tau = l\tau$, then τ is a composition of σ and another substitution, say λ; thus, $\tau = \sigma\lambda$. The instance $k\sigma$ is then called the most general instance of k and l.
The unification algorithm can be described as follows:

1. let the given literals be k and l; set $j = 0$ and $\theta_0 = \varepsilon$ (the identity substitution);
2. if $k\theta_j = l\theta_j$, then stop and θ_j is the most general substitution;
3. scan $k\theta_j$ and $l\theta_j$ in parallel from left to right and locate the leftmost position in which they do not agree; let t1 and t2 be the terms which begin at that position; if neither t1 nor t2 is a variable or either t1 or t2 is a variable which is properly contained in the other, then stop: k and l are not unifiable;
4. if t1 is a variable then set $\theta_{j+1} = \theta_j \{t2|t1\}$, else set $\theta_{j+1} = \theta_j \{t1|t2\}$, add 1 to j and goto step 2.

We can now define the resolution rule.

The resolution rule: If A and B are clauses (without any variables in common) containing literals k and l respectively such that k and l are "opposite in sign" but $|k|$ and $|l|$ (where $|k|$ is meant to be the literal k with its negation sign, if any, deleted) have a most general common instance m and σ is the most general substitution with $m = |k|\sigma = |l|\sigma$, then infer from A and B the clause $C = (A-\{k\})\sigma \cup (B-\{l\})\sigma$.

C is called a resolvent of A and B.

R(A,B) denotes all resolvents that can be derived from A and B; note that R(A,B) = R(B,A).

Some examples

1. From the clauses $P(x) \vee \neg Q(x)$ and $\neg P(x) \vee \neg P(y) \vee R(x,y)$ we can derive two new clauses: one by unifying $P(x)$ and $\neg P(x)$:

$$\neg Q(x) \vee \neg P(y) \vee R(x,y)$$

and one by unifying $P(x)$ and $\neg P(y)$:

$$\neg Q(y) \vee \neg P(x) \vee R(x,y).$$

2. From the clauses $P(x,f(x,y),z) \vee Q(x,z)$ and $\neg P(h(x),y,g(x,y)) \vee R(x,y)$ we derive the clause:

$$Q(h(x),g(x,f(h(x),t))) \vee R(x,f(h(x),t)).$$

This we shall show step by step:
First, take care that the variables differ:

$$P(s,f(s,t),v) \vee Q(s,v)$$

$$\neg P(h(x),y,g(x,y)) \vee R(x,y)$$

$j = 0$ and $\theta_0 = \varepsilon$
scan from left to right:

$$\theta_1 = \{h(x)|s\} \quad j = 1$$

$$\theta_2 = \{h(x)|s,\ f(h(x),t)|y\} \quad j = 2$$

$$\theta_3 = \{h(x)|s,\ f(h(x),t)|y,\ g(x,f(h(x),t))|v\}$$

$$j = 3 \text{ and } P(s,f(s,t),v)\theta_3 =$$

$$P(h(x),y,g(x,y))\theta_3 =$$

$$P(h(x),f(h(x),t),g(x,f(h(x),t))).$$

3. An example of two literals which are not unifiable:

$$P(x, f(x)) \text{ and } P(y, g(y))$$

or

$$P(x, f(x)) \text{ and } P(s, s).$$

What can we do with this inference rule?
Given some axioms $A_1, \ldots, A_n$ and an alleged theorem B, we want to get an answer to the question: does B follow from $A_1 \wedge A_2 \wedge \ldots \wedge A_n$.
We transform this question into: is the set containing $\{A_1, \ldots, A_n, \neg B\}$ unsatisfiable.
We suppose $A_1, \ldots, A_n, \neg B$ to be in clause form; we can then derive new clauses from this set of clauses.
The completeness of the resolution rule means that if $A_1, \ldots, A_n, \neg B$ is an unsatisfiable set of clauses, then we can derive in a finite number of steps the null clause (denoted by $\square$), and if we can derive the null clause the original set was unsatisfiable.
If the initial set of clauses is S (the axioms and the negated theorem), the resolution rule defines the search space as follows:

$$R^{\circ}(S) = S$$

$$n \geq 0: R^{n+1}(S) = \{C \mid C \in R(A,B) \ \& \ A,B \in R^{n}(S)\} \cup R^{n}(S).$$

Completeness now means:
S is unsatisfiable iff there is an n such that

$$\square \in R^{n}(S).$$

Each refinement of the resolution rule has associated with it a refining condition P(A,B) on pairs of clauses A,B: a refinement allows only the resolvents of clauses A,B satisfying P to be generated.

If $\tilde{R}^{n}(S)$ denotes the subset of $R^{n}(S)$ that will be generated by a refinement, then:

$$\tilde{R}^{O}(S) = S$$

$$n \geq 0: \tilde{R}^{n+1}(S) = \{C \mid C \in R(A,B) \ \& \ A,B \in \tilde{R}^{n}(S) \ \& \ (P_n(A,B) \vee P_n(B,A))\} \cup \tilde{R}^{n}(S).$$

The completeness again states that S is unsatisfiable iff there is an n such that $\Box \in \tilde{R}^n(S)$.

If, for a clause C, $C \in R^n(S) - R^{n-1}(S)$, where $R^{-1}(S)$ is taken to be $\emptyset$, we say that C is a clause of level n. The level of a clause is not uniquely determined because the same clause can be constructed in several ways. The level of $\Box$ is called the level of the proof. In the sequel the phrase "l is the level of C" means l is equal to the smallest n such that $C \in R^n(S) - R^{n-1}(S)$.

We shall mention only such refinements as we have implemented.

1. Unrestricted binary resolution, mentioned above, $P_n(A,B) \equiv$ <u>true</u>.

2. Plus p-resolution, allowing only to make so-called +p deductions. A positive clause is a clause which contains no negative literals; a negative clause contains only negative literals.
   Condition for plus p-resolution:

$$P_n(A,B) \equiv \text{df } A \text{ is a positive clause.}$$

3. Minus-p-resolution; condition:

$$P_n(A,B) \equiv \text{df } A \text{ is a negative clause.}$$

4. Resolution with set of support.
   A subset of S is chosen as a support set T.

$$P_n(A,B) \equiv \text{df } A \in R^n(S) - (S-T) \text{ or } B \in R^n(S) - (S-T)$$

   for n = 1 this gives: $A \in T$; thus only such clauses are derived in whose derivation at least one clause is used which is taken from the support set.
   This refinement yields a complete inference system iff S − T is a satisfiable subset of S (for example the axioms).

5. Linear resolution with set of support.
   Let $R^{-1}(S)$ be the set S − T where T is the support set:

$$P_n(A,B) \equiv \text{df } (A \in S \vee A \in Tr(B)) \ \& \ (B \in \tilde{R}^n(S) - \tilde{R}^{n-1}(S)).$$

   $A \in Tr(B)$ means: A belongs to the deduction tree of B.

Completeness

The shortest proofs for completeness are given by Anderson and Bledsoe [5].

Essential in those proofs is the so-called lifting lemma (Robinson) proved in [6].

A ground clause is a clause which contains no variables.

Very informally described this lemma states that when R denotes the resolution operation and P denotes some very specific instantiation (replacing of variables by constants) then $R(P(S)) \subseteq P(R(S))$; as a generalization the relation $R^n(P(S)) \subseteq P(R^n(S))$ holds.

This means that if we have proved the completeness of the resolution rule for ground clauses, implying a proof of $\exists n \ \Box \in R^n(P(S))$, we can conclude that $\Box \in R^n(S)$ which completes the proof of the general completeness.

We will therefore prove completeness of unrestricted resolution for ground clauses according to [5] as an example.

Define $k(S) = (\sum_{C \in S} |C|) - |S|$ where $|S|$ is the number of clauses in S and $|C|$ the number of literals in the clause C.

Theorem. If S is an unsatisfiable set of ground clauses then $\Box$ can be deduced by resolution.

Proof. By induction on $k(S)$.

Step 1. If $k(S) = 0$ then either $\Box \in S$ or S consists only of unit clauses. The only way in which a set of unit clauses can be unsatisfiable is for two of those unit clauses to be negations of each other. Using these two clauses will immediately produce $\Box$.

Step 2. Suppose a) S is an unsatisfiable set of clauses with $k(S) = N>0$ and b) for any unsatisfiable S' of clauses with $k(S') < N$ there is a deduction of $\Box$ from S by resolution.

If $\Box \in S$, we have finished, so suppose $\Box \notin S$; since $k(S) > 0$ there is at least one clause of the form $A \vee L$ where L is a literal, so $S = S' \cup \{A \vee L\}$.

Consider the sets $S_1 = S' \cup \{A\}$ and $S_2 = S' \cup \{L\}$.

Note that both sets are unsatisfiable and $k(S_1) < N$ and $k(S_2) < N$. From the induction hypothesis it follows that there is both a deduction of $\square$ from S_1 and a deduction of $\square$ from S_2 by resolution; in other words: for some m and n: $\square \in R^m(S_1)$ and $\square \in R^n(S_2)$. If $\square$ can be deduced by resolution from S_1 then either $\square$ or $\{L\}$ can be deduced from S by simply performing the same resolution steps.
When $\{L\}$ is produced, all clauses of $S_2 = S' \cup \{L\}$ are produced from S by resolution (since surely those of S' are); so $S_2 \subseteq R^m(S)$, hence $R^n(S_2) \subseteq R^{m+n}(S)$: so $\square$ is produced by resolution from S. This completes the proof.

2.2. Search strategies and completeness

The search space can be searched in several ways. In our program two strategies can be chosen:

1. A complexity saturation strategy,

or

2. A diagonal search strategy given in [1] and applied for example in [7].

We have to define complexity of a clause. This can be done in several ways. In the program the level of a clause is chosen as the complexity of that clause; a different definition of complexity would have been the number of resolution steps taken before finding this clause (number of edges in the deduction tree).
A complexity saturation search strategy first generates all clauses of complexity 0 and will generate all clauses of complexity f before generating any clause of complexity f + 1.
When, in our program, this strategy is chosen first all resolvents of level 1 are generated, then those of level 2 and so on.
One restriction was incorporated: upon finding a unit clause the program tries to unify this clause with all unit clauses with opposite sign generated before, in an attempt to derive the null clause.
The completeness of this strategy follows from the completeness of the inference system.
Saturation strategies are inefficient but complete and terminate for

any unsatisfiable S with a refutation (deduction of □) of least complexity.

A diagonal search strategy generated clauses in order of cost, where the cost of a clause is defined as the sum of the complexity g of the clause and its 'heuristic value' h.
This heuristic value can be for example the number of literals of the clause, which is an estimate (specifically, a lower bound) of the path that must be covered before reaching the null clause.
This search strategy is called diagonal as clauses of the same cost (g+h) are on the same diagonal when plotted in a two dimensional array with coordinates g and h. (fig. 1).

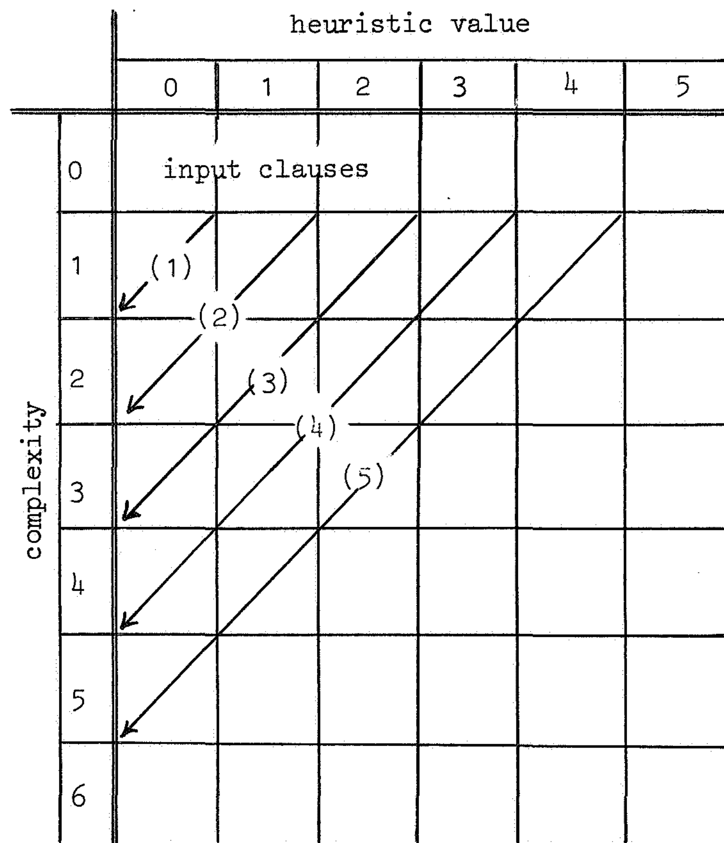


fig. 1. Order of generating clauses by using a diagonal search strategy.

Input clauses have complexity zero.
In the program we have chosen 'level' as complexity and 'number of literals' as heuristic value of a clause.
If S has a refutation of least complexity k then both complexity saturation and diagonal search terminate with a simplest proof.
Saturation search generates all clauses generated by diagonal search (with the same function as complexity) but will generate in addition all clauses of complexity less than k with cost greater than k.
So diagonal search is always at least as efficient as complexity saturation search.
For some examples see section 3.4.

2.3. Edit strategies

A generated clause can be redundant; an edit strategy tries to delete some of these redundant clauses.
An edit strategy can be compatible or incompatible with the proof procedure used [2].

1) Deletion of clauses which are alphabetic variants of clauses already in memory.

2) Deletion of tautologies (clauses of the form $P \vee \neg P$).

3) Clauses of the form $P \vee P \vee Q$ are transformed to

$$P \vee Q.$$

These three strategies are compatible with all complete proof procedures using a resolution rule.

4) An important edit strategy, not implemented in the program, is the deletion of so-called subsumed clauses.
For example: if we have generated a clause A and we now generate a clause $A \vee B$ or vice versa, this clause $A \vee B$ gives us no more information and may be deleted.

5) The user gives a limit to the number of literals of the generated clauses; no clauses with a greater number of literals will be generated. Also, a limit to the depth of nesting of function

symbols may be prescribed. These latter strategies are, of course, incompatible with any proof procedure, but sometimes they may help us to find a proof more quickly.

It is clear that we should try to avoid generating redundant clauses in the first place in stead of deleting them afterwards.

References

[1] D. Luckham, The resolution principle in theorem proving. Machine Intelligence I (eds. Collins and Michie) (1967), 47-63.

[2] R. Kowalski, Search strategies for theorem proving. Machine Intelligence V (eds. Meltzer and Michie) (1970), 181-202.

[3] D. Luckham, Refinement theorems in resolution theory. Symp. on Aut. Dem. Lecture notes in Mathematics 125 (1970), 163-190.

[4] J. Allen and D. Luckham, An interactive theorem proving program. Machine Intelligence V (eds. Meltzer and Michie) (1970), 321-337.

[5] R. Anderson and W. Bledsoe, A linear format for resolution with merging and a new technique for establishing completeness. JACM July 1970, 525-534.

[6] J.A. Robinson, A machine oriented logic based on the resolution principle. JACM January 1965, 23-41.

[7] R. Kowalski and D. Keuhner. Linear resolution with selection function. Memo 34, October 1970. Metamathematics Unit. Edinburgh University.

3. The program

3.1. Description

The program is organized in such a way that it can be used both off line and on line. The version published here is off line. For the on line version the Boolean *on line* must be set *true* and the system procedures *resym*, *prsym*, *printtext*, *nlcr* and *new page* (see [1]) require an environment in which they work through a teletype.

input description for the theorem-prover.

```
<theory> ::= <heading> <list of axioms> <bar> <negated theorem>.

<heading> ::= <thinking time>, <strategy letter>, <inf system letter>,
              <maximal number of literals>,
              <maximal depth>,

<strategy letter> ::= s|d
<inf system letter> ::= u|p|m|s|l
<list of axioms> ::= <list of clauses>
<negated theorem> ::= <list of clauses>
<list of clauses> ::= <clause>|<list of clauses>, <clause>
<clause> ::= <clause number>:<list of literals> <comment>
<list of literals> ::= <literal>|<list of literals>V<literal>
<literal> ::= <predicate>|¬<predicate>
<predicate> ::= <predicate letter> <idgit> <parameter part>
<predicate letter> ::= k|l|m|n|o|p|q|r
<idgit>      ::= <empty>|0|1|2|3
<parameter part> ::= <empty>|(<list of terms>)
<list of terms> ::= <term>|<list of terms>, <term>
<term>       ::= <constant>|<function>|<variable>
<constant> ::= <constant letter> <idgit>|<unsigned integer>
<constant letter> ::= a|b|c|d|e
<function> ::= <function letter> <idgit> <parameter part>
<function letter> ::= f|g|h|i|j
<variable> ::= <variable letter> <idgit>
<variable letter> ::= s|t|u|v|w|x|y|z
<comment> ::= '<string not containing '>'|<empty>
```

<inf system letter>;

u unrestricted resolution,

p plus P resolution,

m minus P resolution,

s set of support, with the negated theorem as a support set,

l linear resolution with set of support.

<thinking time>; to be expressed in seconds.

<clause number>; all clauses of a theory are numbered, starting with 1 for the first clause.

<idgit>; the idgits zero and empty are considered identical.

Capitals and their corresponding small letters are considered identical.

Example: existence of the left-inversion.

axioms: $x.x^{-1} = e$,

$x.e = x$,

$x.(y.u) = (x.y).u$,

theorem: ${}^{-1}x.x = e$.

input for the theorem-prover:

100, d, p, 8, 8,

1: $P(x,g(x),e)$,

2: $P(x,e,x)$,

3: $\neg P(y,u,v) \vee \neg P(x,v,w) \vee \neg P(x,y,z) \vee P(z,u,w)$,

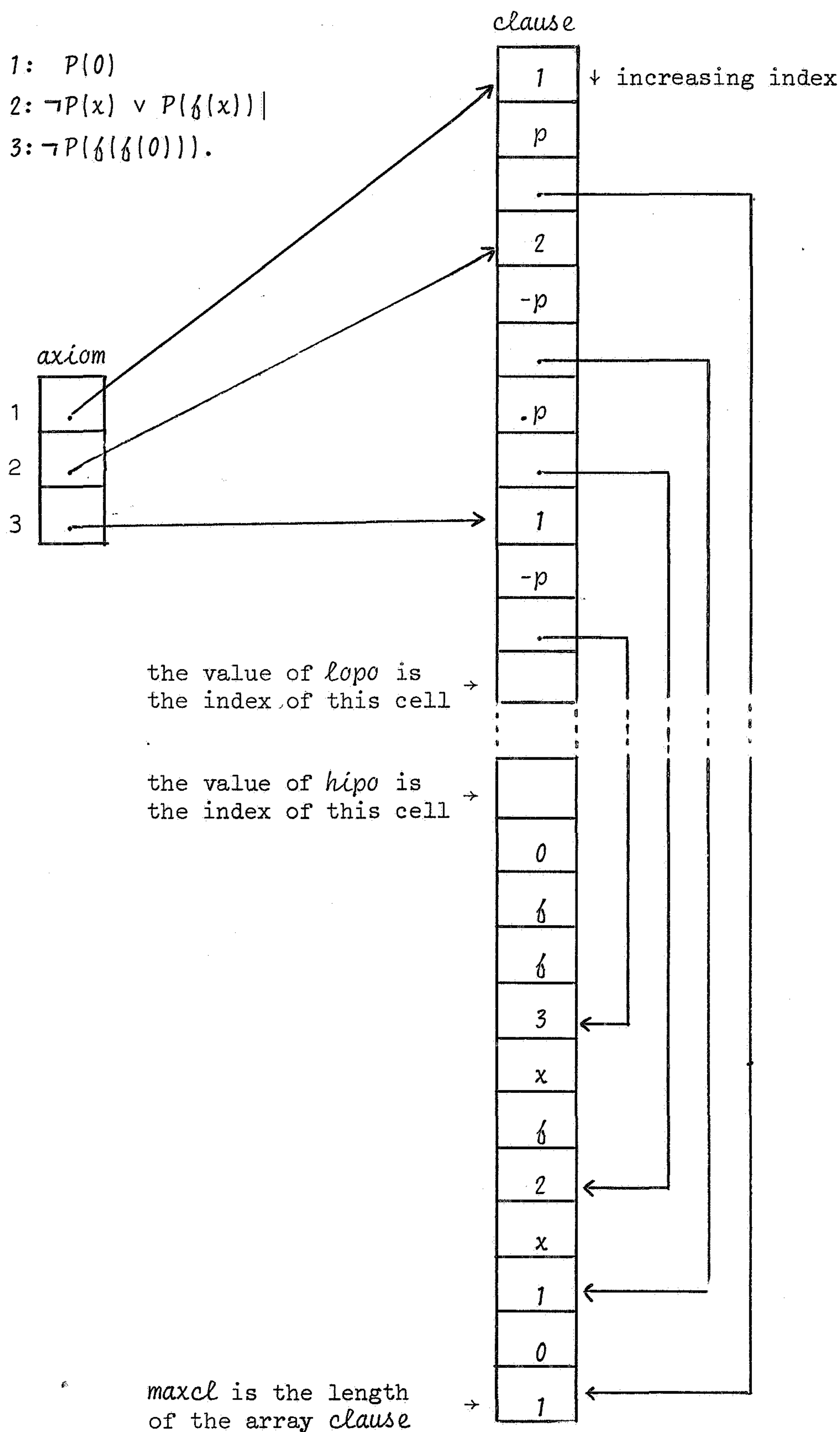
4: $\neg P(x,y,z) \vee \neg P(z,u,w) \vee \neg P(y,u,v) \vee P(x,v,w)|$

5: $\neg P(x,a,e)$.

If a theory which is used as input is not in accordance with the input description, it is skipped by the off line version. The one line version will skip the theory if the error is in the heading, but if there is an error in a clause, only that clause is skipped and asked for again. Skipping in the on line version is done by setting the integer *isstockp* equal to zero.

All initial and deduced clauses are stored in the array *clause*, references to these clauses are in *axiom* [ ]. The figure shows the organization

of *axiom* and *clause* for the situation of the simple theory:



For each existing function or predicate the number of parameters is stored in the array *nopar*. In the array *substitution* there is a cell for each variable. If there are no substitutions for a certain variable, its cell contains zero, else its cell contains a reference to a complete term in the array *clause*. In this way there can be some kind of recursion in the substitution; e.g., the cell of x refers to $f(v)$ while the cell of v refers to w.

Each cycle of computation may be considered beginning with a choice operation where two clauses from the list of clauses already in memory are chosen to make the next deduction, according to the search strategy (procedure *search* if complexity saturation is required and procedure *search diagonal* if the diagonal search strategy is used) and the inference system (the Boolean procedure *cond*(i,j) delivers the value *true* only if the two clauses with number i and j satisfy the condition given by the inference system) chosen by the user.
Then the procedure *resolve* is called with the numbers of the two clauses as parameters.
Resolve begins to check if there is any time left to go on at all; if there is, it verifies whether the new clause will have too many literals (edit strategy); if so, we return and choose the next two clauses, otherwise we search for two complementary literals in the two clauses and try to unify those two.

The implementation of the unification algorithm.
We wrote an integer procedure *subsym*, one of whose parameters is a predicate. The first call of *subsym* delivers the predicate identifier and subsequent calls deliver one after the other the symbols of its parameter list while any defined substitutions are being performed. After the last symbol has been delivered *subsym* yields the value 1000 upon the next call. With the help of *subsym* we wrote *subsym1* (*pred 1*) and *subsym2* (*pred 2*) which make it possible to deliver two predicates simultaneously.

The Boolean procedure *unifiable* (*pred 1*, *pred 2*) checks if *pred 1* and *pred 2* are unifiable and fills the array *substitution*. Local to *unifiable* is the procedure *substitution control* which makes sure that the variable for which we need a substitution does not occur in the parameterlist of the function that is to be substituted. As a side effect a correct substitution is noted.

We now give a simplified description of *unifiable:*

```
L: t1:= subsym 1 (pred 1); t2:= subsym 2 (pred 2);
   if t1 = 1000 ∧ t2 = 1000 then unifiable := true
   else
   if t1 = t2 then goto L
   else
   if (variable (t1) ∨ variable (t2)) ∧
       substitution control then goto L
   else
   unifiable := false
```

When we have found the clauses to be unifiable we can construct the resolvent by using *subsym* (procedure *make resolvent*). If the resolvent is a null clause we print the proof; if it is a unit clause we look ahead if the null clause can be generated directly; else we first examine whether the clause just generated is a redundant one (edit strategy). If the new clause (number = nocl) is a tautology, the procedure *notaut* delivers the value *false* and the clause is skipped; when it is a alphabetic variant of some clause already in memory, *notdupl* delivers the value *false* and we also skip this clause and choose the next two clauses to be resolved.
If the clause is not deleted we fill the array *history*: *history* [*nocl*] becomes the number of the first parent * *maax* (maximum of the number of clauses that can be stored in the memory) + the number of the second parent. This array *history* is used to print out the proof when the null clause has been found.

```
begin comment 3.2. The text of the ALGOL 60 program;
  boolean on line;
  integer isstockp;
  on line:= false;
  begin integer left bracket, right bracket, comma, negsym, orsym,
    colon, period, bar, apos, new line, tabsym, space sym, letter
    d, letter s, letter u, letter l, letter p, letter m, maxcl, maax;
    maxcl:= 20000; maax:= 1000; comma:= 87; colon:= 90;
    negsym:= if on line then 65 else 76;
    orsym:= if on line then 64 else 79; period:= 88;
    left bracket:= 98; right bracket:= 99; new line:= 119;
    tabsym:= 118; spacesym:= 93; apos:= 120;
    bar:= if on line then 67 else 127; letter s:= 28; letter d:= 13;
    letter u:= 30; letter p:= 25; letter m:= 22; letter l:= 21;
    begin integer think time, strategy, litno, depth, start, level,
      proof time, inf syst, noax, noth, nocl, lopo, hipo, sym,
      bracket counter, address1, address2, last term or pred1, last
      term or pred2, stackp1, stackp2, number of theories, i;
      integer array axiom[1:maax], clause[1:maxcl], history[1:maax],
      substitution[280:359], nopar[150:279], substack1,
      substack2[0:80,1:2];
      boolean execute, on;

      procedure read heading;
      begin

        procedure seperator; if sym = comma then readsym else
        error(‹›, er(15), exit);

        sym:= 0; bracket counter:= 0;
        think time:= read1(read sym, sym);
        if think time < 0 then exit; seperator; strategy:= sym;
        readsym; seperator; inf syst:= sym; readsym; seperator;
        litno:= read1(readsym, sym); seperator;
        depth:= read1(readsym, sym); if sym ≠ comma then error(
        ‹›, er(15), exit);
        if inf syst ≠ letter u ∧ inf syst ≠ letter s ∧ inf syst ≠
        letter p ∧ inf syst ≠ letter m ∧ inf syst ≠ letter l then
        error(‹›, er(17), exit);
        if strategy ≠ letter s ∧ strategy ≠ letter d then error(
        ‹›, er(16), exit); sym:= 0
      end read heading;


      integer procedure restart;
      begin nlcr; nlcr; printtext(
        ‹if you want to try it with another strategy›); nlcr;
        printtext(‹then give a new heading else print 0,›); nlcr;
        read heading; nocl:= noax + noth;
        lopo:= axiom[nocl] + 2 × clause[axiom[nocl]];
        hipo:= clause[lopo] - clause[clause[lopo]] - 1; restart:= 0;
        goto cs
      end;
```

```
integer procedure additional time;
begin printtext(⟨how much more time do you want to spend ⟩);
  nlcr; proof time:= proof time + think time; sym:= 0;
  think time:= read1(read sym, sym);
  if think time < 0 then restart else start:= time;
  additional time:= 0
end;


procedure time control;
if think time < time - start then error(
⟨⟩, er(102), if on line then additional time else exit);


boolean procedure constant(s); value s; integer s;
comment a0,....e9,0,1,...;
constant:= s ≤ 10 ∨ s ≥ 100 ∧ s < 150;


boolean procedure function(s); value s; integer s;
comment f0,....j9;
function:= s ≥ 150 ∧ s < 200;


boolean procedure predicate(s); value s; integer s;
comment k0,....r9;
predicate:= s ≥ 200 ∧ s < 280;


boolean procedure variable(s); value s; integer s;
comment s0,....z9;
variable:= s ≥ 280 ∧ s < 360;


integer procedure lopoplus1;
begin lopoplus1:= lopo:= lopo + 1; if lopo = hipo then
  error(⟨⟩, er(101), if on line then restart else exit)
end;


integer procedure hipomin1;
begin hipomin1:= hipo:= hipo - 1; if lopo = hipo then
  error(⟨⟩, er(101), if on line then restart else exit)
end;


procedure deaf;
begin on:= on line; on line:= false end;


procedure hear; if on then on line:= true;


integer procedure readsym;
```

```
begin boolean comment;
  comment:= false;
1: sym:= resym; deaf; prsym(sym); hear; if sym = apos then
  begin comment:= ¬comment; goto 1 end;
  if comment ∨ sym = new line ∨ sym = tabsym ∨ sym = space sym
  then goto 1;
  if sym = left bracket then bracket counter:= bracket counter
  + 1 else if sym = right bracket then bracket counter:=
  bracket counter - 1;
  if (sym = colon ∨ sym = period ∨ sym = orsym) ∧ bracket
  counter ≠ 0 then
  begin bracket counter:= 0; error(⟨⟩, er(12), nothing) end;
  if sym > 36 ∧ sym < 63 then sym:= sym - 27;
  readsym:= sym
end read sym;


procedure error(diagnosis, action, termination);
string diagnosis; integer action, termination;
begin
  integer procedure create(expr); integer expr;
  comment create is used to bring expr to life;
  create:= expr;

  nlcr; printtext(diagnosis); create(action); nlcr;
  create(termination)
end error;


integer procedure exit;
begin exit:= 0;
  if on line then isstockp:= 0 else if sym ≠ period then
  for sym:= readsym while sym ≠ period do ; nlcr; goto ex
end;


integer procedure print(x); value x; integer x;
begin integer i, j, xd;
  integer array h[1:8];
  print:= x; if x < 0 then prsym(65); x:= abs(x); xd:= x div 10;
  for i:= 1, i + 1 while x > 0 do
  begin j:= i; h[i]:= x - xd × 10; x:= xd; xd:= x div 10 end;
  for i:= j step - 1 until 1 do prsym(h[i])
end print;


integer procedure er(n); value n; integer n;
comment for use as 'action' in 'error';
begin nlcr; tab; prsym(sym); tab; printtext(⟨error ⟩);
  er:= print(n); nlcr
end er;


integer procedure nothing; comment for use in 'error';
nothing:= 0;
```

```
procedure read theory;
begin integer axi, olopo, ohipo;

  procedure initialize;
  begin integer i;
    olopo:= lopo:= 0; ohipo:= hipo:= maxcl + 1; execute:= true;
    noax:= noth:= nocl:= - 1;
    for i:= 150 step 1 until 279 do nopar[i]:= - 1
  end initialize;


  integer procedure lopo plus1;
  begin lopo plus1:= lopo:= lopo + 1;
    if lopo ≥ hipo then error(⟨⟩, er(1), exit)
  end;


  integer procedure hipo min1;
  begin hipo min 1:= hipo:= hipo - 1;
    if lopo ≥ hipo then error(⟨⟩, er(1), exit)
  end;


  integer procedure no execution(but go on, if); label but go on;
  boolean if; comment for use as 'termination' in 'error';
  begin execute:= false; no execution:= 0;
  l: if if then goto but go on else
    begin read sym; goto l end
  end no execution;


  integer procedure no execution on line version(but go on, if,
  next clause); label but go on, next clause; boolean if;
  begin comment if an error is found in the on line input: the
    last clause is read again;
    no execution on line version:= 0; if on line then
    begin axi:= axi - 1; nlcr; lopo:= olopo; hipo:= ohipo;
      bracket counter:= 0; printtext(⟨last clause again⟩);
      nlcr; isstockp:= sym:= 0; goto next clause
    end
    else no execution(but go on, if)
  end no execution on line version;


  integer procedure read clause;
  comment gives the address of a clause in clause[ ];
  begin integer nol, rc;

    integer procedure no execution(but go on, if);
    label but go on; boolean if;
    no execution:= no execution on line version(but go on,
    if, next clause);
```

```
comment now both versions look the same;

integer procedure read literal;
begin comment gives the address of a literal;
  boolean negation;
  integer predsym, pp;

  integer procedure read identifier;
  begin comment gives the code of an identifier;
    integer identifier;
    if sym < 10 then
    begin read identifier:= - read1(readsym, sym);
      goto identifier read
    end;
    if sym < 10 ∨ sym > 35 then error(
    <>, er(2), no execution(if sym = orsym then next
    literal else if sym = colon then next clause else
    execution, sym = orsym ∨ sym = colon ∨ sym = period));
    read identifier:= identifier:= 10 × sym; read sym;
    if sym < 10 then
    begin if sym < 4 then read identifier:= identifier +
      sym else
      begin error(<>, er(3), no execution(here, true));
      here: read identifier:= identifier + sym
      end;
      read sym
    end;
  identifier read:
  end read identifier;


  integer procedure read term;
  begin comment gives the address of a term;
    integer term;
    term:= read identifier;
    if ¬(constant(term) ∨ function(term) ∨ variable(term))
    then error(
    <>, er(4), no execution(if sym = orsym then next
    literal else if sym = colon then next clause else
    execution, sym = orsym ∨ sym = colon ∨ sym = period));
    read term:= hipo min 1; clause[hipo]:= term;
    if function(term) then read parameters(term);
  end read term;


  procedure read parameters(pred or fun); value pred or fun;
  integer pred or fun;
  begin integer pari;
    pari:= 0; if sym = left bracket then
    begin read sym;
    next parameter: read term; pari:= pari + 1;
      if sym = comma then
      begin read sym; goto next parameter end
      else if sym ≠ right bracket then error(
      <>, er(5), no execution(if sym = orsym then next
```

```
        literal else if sym = colon then next clause else
        execution, sym = orsym ∨ sym = colon ∨ sym =
        period)) else read sym
      end;
      if nopar[pred or fun] = − 1 then nopar[pred or fun]:=
      pari else if nopar[pred or fun] ≠ pari then error(
      ⌘, er(6), no execution(if sym = orsym then next
      literal else if sym = colon then next clause else
      execution, sym = orsym ∨ sym = colon ∨ sym = period))
    end read parameters;

    read literal:= 0;
    if sym ≠ colon ∧ sym ≠ orsym then error(
    ⌘, er(10), no execution(if sym = colon ∨ sym = orsym
    then here else execution, sym = colon ∨ sym = orsym
    ∨ sym = period)) else read sym;
  here: if sym = negsym then
    begin negation:= true; read sym end
    else negation:= false; predsym:= read identifier;
    if ¬predicate(pred sym) then error(
    ⌘, er(7), no execution(if sym = orsym then next literal
    else if sym = colon then next clause else
    execution, sym = orsym ∨ sym = colon ∨ sym = period));
    read literal:= lopo plus 1;
    clause[lopo]:= if negation then − pred sym else pred sym;
    pp:= hipo min 1; read parameters(abs(pred sym));
    lopo plus 1; clause[lopo]:= pp; clause[pp]:= pp − hipo
  end read literal;

  if sym = comma then readsym else if sym = bar then
  begin noax:= axi − 1; readsym end
  else if sym = period then
  begin if noax = − 1 then error(⌘, er(14), exit) else
    begin nocl:= axi − 1; noth:= nocl − noax; goto execution
    end
  end;
  if sym < 10 then
  begin if axi ≠ read1(read sym, sym) then error(
    ⌘, er(8), no execution(if sym = colon then here else
    execution, sym = colon ∨ sym = period))
  end
  else error(
  ⌘, er(9), no execution(if sym = colon then here else
  execution, sym = colon ∨ sym = period)); olopo:= lopo;
  ohipo:= hipo;
here: read clause:= rc:= lopo plus1; nol:= 0;
next literal: read literal; nol:= nol + 1;
  if sym = orsym then goto next literal; clause[rc]:= nol;
next clause:
end read clause;

initialize;
for axi:= 1 step 1 until maax do
begin history[axi]:= 0; olopo:= lopo; ohipo:= hipo;
  axiom[axi]:= read clause
```

```
  end;
execution: if sym ≠ period then error(
  ⟨⟩, er(13), exit) else if ¬execute then error(
  ⟨⟩, er(11), exit)
end read theory;


integer procedure print identifier(i); value i; integer i;
if i < 0 then print( - i) else
begin comment i is the code of an identifier;
  integer s, d;
  print identifier:= i; s:= i div 10; d:= i - s × 10; prsym(s);
  if d > 0 then prsym(d); prsym(93)
end print identifier;


integer procedure print term(t); value t; integer t;
begin comment t is the address of a term, this procedure
  delivers the highest address lower than t which does not
  belong to the complete term;
  integer term;
  term:= clause[t]; print identifier(term);
  print term:= if function(term) then print parameter
  list(term, t - 1) else t - 1
end print term;


integer procedure print parameter list(pred or fun, pl);
value pred or fun, pl; integer pred or fun, pl;
if nopar[pred or fun] > 0 then
begin comment pred or fun is the code of an identifier of a
  predicate or function,pl is the place in clause[] where its
  parameter list begins. as in print term the procedure
  delivers the first value which does not belong to the
  parameter list;
  integer i, npf;
  npf:= nopar[pred or fun]; prsym(left bracket);
  for i:= 1 step 1 until npf - 1 do
  begin pl:= print term(pl); prsym(comma) end;
  print parameter list:= print term(pl); prsym(right bracket)
end print parameter list
else print parameter list:= pl;

integer procedure print literal(l); value l; integer l;
begin comment l is the address of a literal. print literal
  delivers the first value bigger than l which does not
  belong to the literal.that means: if l is not the last
  literal of the clause then print literal delivers the
  address of the next one;
  integer pred;
  pred:= abs(clause[l]); if clause[l] < 0 then prsym(negsym);
  print identifier(pred);
  print parameter list(pred, clause[l + 1] - 1);
  print literal:= l + 2
end print literal;
```

```
procedure print clause(address in clause);
value address in clause; integer address in clause;
begin integer i, n of lit;
  n of lit:= clause[address in clause] - clause[address in
  clause] div 100 × 100; if n of lit = 0 then
  begin prsym(100); prsym(101); goto clause printed end;
  address in clause:= address in clause + 1;
  for i:= 1 step 1 until n of lit - 1 do
  begin address in clause:= print literal(address in clause);
    prsym(orsym)
  end;
  print literal(address in clause);
clause printed:
end print clause;


procedure print axiom(address in axiom); value address in axiom;
integer address in axiom;
begin nlcr; print(address in axiom); prsym(colon);
  print clause(axiom[address in axiom])
end;


integer procedure last of(term); value term; integer term;
if function(clause[term]) then
begin if variable(clause[term]) ∨ constant(clause[term]) then
  last of:= term
end
else
begin comment last of gives the last address which still
  does belong to term;
  integer i, lo, nopart;
  nopart:= nopar[clause[term]]; lo:= term;
  for i:= 1 step 1 until nopart do lo:= last of(lo - 1);
  last of:= lo
end last of;


integer procedure subsym(term or pred, last term or pred,
stackp, address, substack); value term or pred;
integer term or pred, last term or pred, stackp, address;
integer array substack;
begin integer t, clt;

  procedure recursion(term); value term; integer term;
  begin stackp:= stackp + 1; substack[stackp,1]:= term;
    substack[stackp,2]:= last of(term)
  end recursion;

  if term or pred ≠ last term or pred then
  begin stackp:= - 1; last term or pred:= term or pred;
    if stackp = - 1 then
    begin if predicate(abs(clause[term or pred])) then
```

```
        begin stackp:= stackp + 1;
          substack[stackp,1]:= clause[term or pred + 1] - 1;
          substack[stackp,2]:= clause[term or pred + 1] -
          clause[clause[term or pred + 1]];
          address:= term or pred;
          clt:= subsym:= abs(clause[term or pred]); goto ready
        end else recursion(term or pred)
      end
    end;

re: if substack[stackp,1] < substack[stackp,2] then
    begin stackp:= stackp - 1; if stackp > - 1 then goto re end;
    if stackp = - 1 then
    begin address:= address - 1; clt:= - 1000; subsym:= 1000;
      goto ready
    end;
k: address:= t:= substack[stackp,1]; substack[stackp,1]:= t - 1;
    clt:= clause[t]; if variable(clt) then
    begin if substitution[clt] > 0 then
      begin recursion(substitution[clt]); goto k end
    end;
    subsym:= clt;
ready:
end subsym;


integer procedure subsym1(term or pred); value term or pred;
integer term or pred;
subsym1:= subsym(term or pred, last term or pred1, stackp1,
address1, substack1);


integer procedure subsym2(term or pred); value term or pred;
integer term or pred;
subsym2:= subsym(term or pred, last term or pred2, stackp2,
address2, substack2);


boolean procedure unifiable(pred1, pred2); value pred1, pred2;
integer pred1, pred2;
begin last term or pred1:= last term or pred2:= 0;
  begin integer t1, t2, j;

    boolean procedure substitution control(var, i); value var;
    integer var, i;
    begin comment delivers true if a substitution is still
      possible else false, a side effect is that the
      substitution is noted down in substitution[];
      integer parcount, u, subvar;
      subvar:= if i = 1 then address1 else address2;
      if ¬function(if i = 1 then t1 else t2) then
      begin substitution control:= true;
        substitution[var]:= subvar; goto ready
      end;
      parcount:= nopar[if i = 1 then t1 else t2];
```

```
      if parcount = 0 then goto true;
    l: u:= if i = 1 then subsym1(pred1) else subsym2(pred2);
      parcount:= parcount - 1 + (if function(u) then nopar[u]
      else 0);
      if u = var then substitution control:= false else if
      parcount > 0 then goto l else
    true:
      begin substitution control:= true;
        substitution[var]:= subvar
      end;
    ready:
    end substitution control;

    for j:= 280 step 1 until 359 do substitution[j]:= 0;
  l: t1:= subsym1(pred1); t2:= subsym2(pred2);
    if t1 = 1000 ∨ t2 = 1000 then
    begin if t1 = 1000 ∧ t2 = 1000 then unifiable:= true  end
    else if t1 = t2 then goto l else if (if variable(t1) then
    substitution control(t1, 2) else if variable(t2) then
    substitution control(t2, 1) else false) then goto l else
    unifiable:= false
  end
end unifiable;


procedure tell history(cl); value cl; integer cl;
if history[cl] > 0 then
begin integer prcl1, prcl2, prcl;
  prcl1:= history[cl] div maax;
  prcl2:= history[cl] - prcl1 × maax; if prcl1 < prcl2 then
  begin prcl:= prcl1; prcl1:= prcl2; prcl2:= prcl end;
  tell history(prcl1); tell history(prcl2); print axiom(cl);
  printtext(⊰:from⊱); print(prcl1); prsym(93); print(prcl2);
  history[cl]:= 0;
end tell history;


procedure line print(nocl, c1, c2); integer nocl, c1, c2;
begin deaf; print axiom(nocl); printtext(⊰:from⊱); print(c1);
  prsym(93); print(c2); hear
end line print;


procedure lookahead;
begin integer i;
  for i:= 1 step 1 until nocl do if clause[axiom[i]] -
  clause[axiom[i]] div 100 × 100 = 1 then
  begin if clause[axiom[nocl] + 1] = - clause[axiom[i] + 1]
    then
    begin if unifiable(axiom[i] + 1, axiom[nocl] + 1) then
      begin nocl:= nocl + 1; deaf; nlcr; printtext(⊰level= ⊱);
        absfixt(4, 0, level + 1); hear;
        axiom[nocl]:= lopoplus1; clause[lopo]:= 0;
        line print(nocl, i, nocl - 1);
        history[nocl]:= i × maax + nocl - 1; nlcr; nlcr;
```

```
        printtext(⟨level=  ⟩); print(level + 1); printtext(
        ⟨    proof time=  ⟩); print(proof time + time - start);
        nlcr; tell history(nocl);
        if on line then restart else exit
      end
    end
  end
end lookahead;


procedure take last;
begin lopo:= axiom[nocl] - 1; nocl:= nocl - 1;
  hipo:= clause[lopo] - clause[clause[lopo]] - 1
end take last;


procedure resolve(c1, c2); value c1, c2; integer c1, c2;
begin integer adc1, adc2, lit1, lit2, i, j, k;
  boolean first;
  time control; adc1:= axiom[c1]; adc2:= axiom[c2];
  lit1:= clause[adc1] - clause[adc1] div 100 × 100;
  lit2:= clause[adc2] - clause[adc2] div 100 × 100;
  first:= true; if lit1 + lit2 - 2 > litno then goto no;
  for i:= 1 step 1 until lit1 do
  for j:= 1 step 1 until lit2 do if clause[adc1 + 2 × i - 1] =
  - 1 × clause[adc2 + 2 × j - 1] then
  begin if first then
    begin make diff(adc1, adc2); first:= false end;
    if unifiable(adc1 + 2 × i - 1, adc2 + 2 × j - 1) then
    begin nocl:= nocl + 1; if nocl = maax then
      begin nlcr; print(lopo); print(hipo); error(
        ⟨⟩, er(101), if on line then restart else exit);
      end;
      if lit1 + lit2 = 2 then
      begin axiom[nocl]:= lopoplus1; clause[lopo]:= 0;
        line print(nocl, c1, c2);
        history[nocl]:= c1 × maax + c2; nlcr; nlcr; printtext(
        ⟨level=  ⟩); print(level); printtext(
        ⟨    proof time=  ⟩); print(proof time + time - start);
        nlcr; tell history(nocl);
        if on line then restart else exit
      end;
      lopoplus1; axiom[nocl]:= lopo;
      clause[lopo]:= lit1 + lit2 - 2 + level × 100;
      for k:= 1 step 1 until lit1 do if k ≠ i then
      begin if make resolvent(adc1 + 2 × k) then goto no end;
      for k:= 1 step 1 until lit2 do if k ≠ j then
      begin if make resolvent(adc2 + 2 × k) then goto no end;
      if lit1 + lit2 = 3 then
      begin if notdupl then
        begin line print(nocl, c1, c2);
          history[nocl]:= c1 × maax + c2; lookahead
        end;
        goto no
      end;
```

```
          if notaut then
          begin if notdupl then
            begin line print(nocl, c1, c2);
              history[nocl]:= c1 × maax + c2;
            end
          end
        end
      end;
   no:
   end resolve;


   procedure make diff(c1, c2); value c1, c2; integer c1, c2;
   begin integer i, j, lit, ad, adr, cl, k, sub;
     for i:= 280 step 1 until 359 do substitution[i]:= 0;
     lit:= clause[c1] - clause[c1] div 100 × 100;
     for i:= 1 step 1 until lit do
     begin adr:= clause[c1 + 2 × i]; ad:= clause[adr];
       for j:= 1 step 1 until ad do if variable(clause[adr - j])
       then substitution[clause[adr - j]]:= 1
     end;
     lit:= clause[c2] - clause[c2] div 100 × 100;
     for i:= 1 step 1 until lit do
     begin adr:= clause[c2 + 2 × i]; ad:= clause[adr];
       for j:= 1 step 1 until ad do
       begin cl:= clause[adr - j]; if variable(cl) then
         begin sub:= substitution[cl];
           if sub = 0 then substitution[cl]:= 2 else if sub < 0
           then clause[adr - j]:= - sub else if sub = 1 then
           for k:= 280 step 1 until 359 do if substitution[k] = 0
           then
           begin substitution[k]:= 1; clause[adr - j]:= k;
             substitution[cl]:= - k; goto next
           end
         end;
       next:
       end
     end
   end make diff;


   boolean procedure make resolvent(pred); value pred; integer pred;
   begin integer address, newnot, fudepth, t, parcount;
     make resolvent:= false; parcount:= fudepth:= newnot:= 0;
     address:= pred - 1; clause[lopoplus1]:= clause[address];
     clause[lopoplus1]:= hipomin1; last term or pred1:= 0;
     subsym1(address);
     for t:= subsym1(address) while t ≠ 1000 do
     begin newnot:= newnot + 1; clause[hipomin1]:= t;
       if function(t) then
       begin fudepth:= fudepth + 1; parcount:= parcount + nopar[t];
         if fudepth > depth then
         begin take last; goto no end
       end
       else parcount:= parcount - 1;
```

```
    if parcount = 0 then fudepth:= 0;
  end;
  clause[clause[lopo]]:= newnot; make resolvent:= true;
no:
end make resolvent;


boolean procedure notdupl;
begin integer i, j, k, n, lit, ad1, ad2, no1, no2, no;
  boolean yes;
  notdupl:= true; ad1:= axiom[nocl] - 1;
  lit:= clause[ad1 + 1] - clause[ad1 + 1] div 100 × 100;
  nocl:= nocl - 1;
  for i:= 1 step 1 until nocl do if clause[axiom[i]] -
  clause[axiom[i]] div 100 × 100 = lit then
  begin ad2:= axiom[i] - 1;
    for j:= 1 step 1 until lit do
    begin yes:= true;
      for k:= 1 step 1 until lit do
      begin if clause[ad1 + 2 × j] = clause[ad2 + 2 × k] then
        begin no1:= clause[ad1 + 2 × j + 1];
          no2:= clause[ad2 + 2 × k + 1]; no:= clause[no1];
          for n:= 0 step 1 until no do if clause[no1 - n] ≠
          clause[no2 - n] then goto noo; yes:= false;
          goto next
        end;
      noo:
      end;
    next: if yes then goto nex
    end;
    goto dupl;
  nex:
  end;
  nocl:= nocl + 1; goto ne;
dupl: notdupl:= false; nocl:= nocl + 1; take last;
ne:
end notdupl;


boolean procedure notaut;
begin integer i, j, address, lit, term1, term2, ad, k;
  notaut:= true; address:= axiom[nocl];
  lit:= clause[address] - clause[address] div 100 × 100;
  for i:= 1 step 1 until lit do
  begin for j:= i + 1 step 1 until lit do if abs(clause[address
    + 2 × j - 1]) = abs(clause[address + 2 × i - 1]) then
    begin term1:= clause[address + 2 × j];
      term2:= clause[address + 2 × i]; ad:= clause[term1];
      for k:= 0 step 1 until ad do if clause[term1 - k] ≠
      clause[term2 - k] then goto no;
      if clause[address + 2 × j - 1] = - clause[address + 2 ×
      i - 1] then
      begin notaut:= false; take last end
      else
      begin lit:= lit - 1;
```

```
        clause[address]:= clause[address] - 1;
        for k:= address + 2 × j + 1 step 2 until lopo do
        begin clause[k - 1]:= clause[k + 1] + ad + 1;
          clause[k - 2]:= clause[k]
        end;
        lopo:= lopo - 2;
        for k:= term1 - ad - 1 step - 1 until hipo do clause[k
        + ad + 1]:= clause[k]; hipo:= hipo + ad + 1
      end
    end;
  no:
  end
end notaut;


boolean procedure bel(ad1, ad2);
begin integer his1, his2;
  his1:= history[ad2] div maax;
  his2:= history[ad2] - his1 × maax;
  bel:= if his1 = ad1 ∨ his2 = ad1 then true else if his1 = 0
  then false else if ¬bel(ad1, his1) then bel(ad1, his2)
  else true
end bel;


boolean procedure belong(address1, address2);
begin integer lev1, lev2;
  lev1:= clause[axiom[address1]] div 100;
  lev2:= clause[axiom[address2]] div 100;
  belong:= if lev1 > lev2 then bel(address2, address1) else
  if lev1 < lev2 then bel(address1, address2) else false
end belong;


procedure fill box(h, f); value h, f; integer h, f;
begin integer i, j, nol, nor, nol1, nor1, nool;
  level:= h; nool:= nocl; deaf; nlcr; printtext(⟨level= ⟩);
  print(level); hear;
  for i:= 1 step 1 until nool do
  begin nor:= clause[axiom[i]]; nol:= nor - nor div 100 × 100;
    nor:= nor div 100;
    for j:= i + 1 step 1 until nool do
    begin nor1:= clause[axiom[j]];
      nol1:= nor1 - nor1 div 100 × 100; nor1:= nor1 div 100;
      if nol + nol1 = f + 2 ∧ (nor = h - 1 ∨ nor1 = h - 1)
      then
      begin if cond(i, j) then resolve(i, j) end
    end
  end
end fill box;


procedure search diagonal;
begin integer diag, h;
  for diag:= 1 step 1 until 100 do
```

```
  begin deaf; nlcr; printtext(⪡diagonal= ⪢); print(diag); hear;
    for h:= diag - 1, diag - 2 step - 1 until 1 do fill
    box(diag - h, h)
  end;
  printtext(⪡all resolvents until diag 100 are made⪢)
end search diag;

boolean procedure cond(address1, address2);
cond:= if inf syst = letter u then true else if inf syst =
letter s then (if level = 1 then (address1 < noax + noth ∧
address1 > noax) ∨ (address2 < noax + noth ∧ address2 > noax)
else true) else if inf syst = letter p then
plusp(address1) ∨ plusp(address2) else if inf syst = letter m
then minp(address1) ∨ minp(address2) else if level = 1
then (address1 < noax + noth ∧ address1 > noax) ∨
(address2 < noax + noth ∧ address2 > noax) else if address1 >
noax + noth ∧ address2 > noax + noth then belong(address1,
address2) else true;


procedure search;
begin integer i, j, noold, nonew;
  nonew:= 0;
  for level:= 1 step 1 until 100 do
  begin deaf; nlcr; printtext(⪡level= ⪢); absfixt(4, 0, level);
    hear; noold:= nonew; nonew:= nocl;
    if noold = nonew then error(
    ⪡⪢, er(103), if on line then restart else exit);
    for i:= 1 step 1 until nonew do
    for j:= (if i < noold then noold + 1 else i + 1) step 1
    until nonew do if cond(i, j) then resolve(i, j)
  end;
  printtext(⪡ all resolvents until level 100 are made ⪢)
end search;


boolean procedure minp(address in axiom);
integer address in axiom;
begin integer address, number, i;
  minp:= false; address:= axiom[address in axiom];
  number:= 2 × (clause[address] - clause[address] div 100 ×
  100);
  for i:= 1 step 2 until number do if clause[address + i] > 0
  then goto no; minp:= true;
no:
end minp;


boolean procedure plusp(address in axiom);
integer address in axiom;
begin integer address, number, i;
  plusp:= false; address:= axiom[address in axiom];
  number:= 2 × (clause[address] - clause[address] div 100 ×
  100);
```

```
      for i:= 1 step 2 until number do if clause[address + i] < 0
      then goto no; plusp:= true;
    no:
    end plusp;

    number of theories:= if on line then 1 000 000 else read; i:= 0;
  l: i:= i + 1; printtext(≮theory number≯); print(i); nlcr;
    isstockp:= 0; read heading; read theory;
  cs: start:= time; proof time:= 0;
    if strategy = letter s then search else search diagonal;
  ex: if i < number of theories then
    begin new page; goto l end;

     end
   end
 end
```

3.3. List of error messages

Errors in the heading:

15: Heading incorrect.

16: Unknown strategy letter.

17: Unknown inference system letter.

Errors found during the translation of the theory:

1: Memory exhausted.

2: No predicate or term where expected.

3: Digit > 3 used as *idgit*.

4: No term where expected.

5: Parameter list not concluded with).

6: Number of parameters of a predicate or function does not agree with former occurrences.

7: No predicate where expected.

8: Clause number is incorrect.

9: Clause number is missing.

10: Literal is not preceded by a colon or by an v.

11: No execution because there are errors in the input.

12: -

13: Period is missing.

14: Bar is missing.

Errors found during the execution:

101: Memory exhausted.

102: Time exhausted.

103: All resolvents are made but the empty clause was not generated.

3.4. Examples

As a comparison between different inference systems and search strategies we give here all results of one example.

Theory in clause form (see section 1).

1: $P(g(x,y),x,y)$,

2: $P(x,h(x,y),y)$,

3: $P(x,y,f(x,y))$,
4: $\neg P(x,y,z) \vee \neg P(y,u,v) \vee \neg P(z,u,w) \vee P(x,v,w)$,
5: $\neg P(x,y,z) \vee \neg P(y,u,v) \vee \neg P(x,v,w) \vee P(z,u,w)$|
6: $\neg P(j(x),x,j(x))$ *'this is the negated theorem'*.

Proofs: (all of level 4)

inf syst.	search strategy	proof time sec.	number of generated clauses
u	*s*	no proof	704
u	*d*	53	109
p	*s*	38	81
p	*d*	38	81
m, s, l	*s*	11	38
m, s, l	*d*	6	26

1. proof using unrestricted resolution or plus p resolution:

   7: $\neg P(s,s1,z) \vee \neg P(s,v,w) \vee P(z,h(s1,v),w)$: *from 5 and 2*
   8: $\neg P(g(s1,z),v,w) \vee P(z,h(s1,v),w)$: *from 7 and 1*
   9: $P(w,h(v,v),w)$: *from 8 and 1*
   10: □: *from 9 and 6.*

2. proof using minus p, set of support or linear resolution with set of support:

   7: $\neg P(s,s1,j(s2)) \vee \neg P(s1,s2,s3) \vee \neg P(s,s3,j(s2))$: *from 6 and 5*
   8: $\neg P(s1,s2,v) \vee \neg P(g(s1,j(s2)),v,j(s2))$: *from 7 and 1*
   9: $\neg P(v,s2,v)$: *from 8 and 1*
   10: □: *from 9 and 2.*

Mendelson [2] gives on page 40, a theory L1 (an axiom system for the propositional calculus). $\vee$ and $\neg$ are the primitive connectives. $A \supset B$ is used as an abbreviation for $\neg A \vee B$. There are four axioms:

1. $A \vee A \supset A$,
2. $A \supset A \vee B$,
3. $A \vee B \supset B \vee A$,
4. $(B \supset C) \supset (A \vee B \supset A \vee C)$.

The only rule of inference is modus ponens. As an exercise there is asked to prove in L1: A ∨ ¬A. We did this exercise with the theorem-prover (in 228 seconds).

Input:

```
300, D, P, 5, 3,
 1:  P(H(F(X,X),X)) 'AXIOM1',
 2:  P(H(X,F(X,Y))) 'AXIOM2',
 3:  P(H(F(X,Y),F(Y,X))) 'AXIOM3',
 4:  P(H(H(Y,Z),H(F(X,Y),F(X,Z)))) 'AXIOM4',
 5: ¬P(H(X,Y)) v P(F(G(X),Y)) 'H(X,Y) IS SHORT',
 6: ¬P(F(G(X),Y)) v P(H(X,Y)) 'FOR F(G(X),Y)',
 7: ¬P(H(X,Y)) v ¬P(X) v P(Y) 'MODUS PONENS'/
 8: ¬P(F(A,G(A))) 'NEGATED THEOREM'.
```

proof:

```
 9: ¬P(H(S,S1)) v P(H(F(S3,S),F(S3,S1))): FROM 7 and 4
10:  P(H(F(S2,F(Z,Z)),F(S2,Z))): FROM 9 and 1
11:  P(F(G(S2),F(S2,Y))): FROM 5 and 2
12: ¬P(H(F(G(S),F(S,S1)),S3)) v P(S3): FROM 11 and 7
13:  P(F(G(Z),Z)): FROM 10 and 12
14: ¬P(F(S,S1)) v P(F(S1,S)): FROM 7 and 3
15:  P(F(Z,G(Z))): FROM 13 and 14
16:  □: FROM 15 and 8.
```

We used the theorem prover to prove that for every set x, $x \cap x = x$. As axioms we took:

1.	$\forall x, y\ \exists z$	$x \cap y = z$,
2.	$\forall x, y\ \exists z$	$x \cup y = z$,
3.	$\forall x, y$	$x \cap y = y \cap x$,
4.	$\forall x, y$	$x \cup y = y \cup x$,
5.	$\forall x, y, z$	$x \cap (y \cap z) = (x \cap y) \cap z$,
6.	$\forall x, y, z$	$x \cup (y \cup z) = (x \cup y) \cup z$,
7.	$\forall x, y$	$x \cap (x \cup y) = x$,
8.	$\forall x, y$	$x \cup (x \cap y) = x$.

In the input $P(x,y,z)$ could be interpreted as: $x \cap y = z$, and $Q(x,y,z)$ as $x \cup y = z$.

Input:

100, d, p, 6, 3,
1: $P(x,y,f(x,y))$,
2: $Q(x,y,g(x,y))$,
3: $\neg P(x,y,z) \vee P(y,x,z)$,
4: $\neg Q(x,y,z) \vee Q(y,x,z)$,
5: $\neg P(x,u,w) \vee \neg P(y,z,u) \vee \neg P(x,y,v) \vee P(v,z,w)$ *'axiom 5a',*
6: $\neg P(v,z,w) \vee \neg P(x,y,v) \vee \neg P(y,z,u) \vee P(x,u,w)$ *'axiom 5b',*
7: $\neg Q(x,u,w) \vee \neg Q(y,z,u) \vee \neg Q(x,y,v) \vee Q(v,z,w)$ *'axiom 6a',*
8: $\neg Q(v,z,w) \vee \neg Q(x,y,v) \vee \neg Q(y,z,u) \vee Q(x,u,w)$ *'axiom 6b',*
9: $\neg Q(x,y,z) \vee P(x,z,x)$ *'axiom 7',*
10: $\neg P(x,y,z) \vee Q(x,z,x)$ *'axiom 8'|*
11: $\neg P(a,a,a)$ *'negated theorem'.*

proof: (in 3 seconds)

12: $Q(s,f(s,s1),s)$: *from 1 and 10*
13: $P(s,s,s)$: *from 12 and 9*
14: $\square$: *from 13 and 11.*

References

[1] D. Grune, Handleiding milli-systeem van de EL X8, Mathematisch Centrum.

[2] Elliot Mendelson, Introduction to mathematical logic, D. van Nostrand Company, Inc. Princeton, 1964.

4. Possible extensions to the program

1. Instead of restricting the inference system to a resolution rule only, one can add another inference rule called factoring. If a clause A contains a literal l and a literal k of the same sign and l and k have a most general unifier σ, then one can infer as factor a new clause $(A-\{l\})\sigma$. Factoring combined with a resolution rule gives a complete inference system.

2. We might implement more advanced search strategies. For example the upward diagonal search strategy. This is a diagonal search strategy i.e., all clauses of cost g are generated before generating any clause of cost $g + 1$. But the order of generating clauses of the same cost (diagonal) can be changed.
   In section 2.figure 1 we draw up all clauses of complexity 1 and cost g first, then those of complexity 2 and so on. Upward diagonal search generates clauses of let us say complexity g and cost g first, if possible; if we then generate a clause of complexity f and cost g $(g>f)$ we must thereafter generate all clauses of cost g and complexity $> f$. (See [2] and [1].)

3. The program either yields the answer yes or it does not stop. If the answer is yes the way the null clause was found is printed. Sometimes (for example in applications to question-answering systems) we want to know more: namely, we want to know which substitutions were made:
   Example:
   Theorem: $\exists x R(x)$; negation: $\forall x \neg R(x)$.
   If we know the substitutions performed during the proof we can know what x satisfies the condition R.
   Green [3] has suggested to replace the negated theorem T by $T \vee ANSWER(\ldots)$ where the parameters of $ANSWER$ are all the variables occurring in T. Instead of deriving the null clause we then derive a clause that only contains the predicate $ANSWER$: the terms of $ANSWER$ then display the substitutions performed on the respective variables.

Another way to get information consists of extracting the information from the resolution tree after the null clause has been found; this has been suggested by Nilsson and Luckham [4]:
In the resolution tree, at every place where the negated theorem occurs, it is replaced by a tautology; if the negated theorem, for example, is $P(x,y)$, we replace it by $P(x,y) \vee \neg P(x,y)$.
Instead of deriving the null clause, then the theorem is derived that has been proved, where Skolem functions are handled in a specific way. The advantage of this method to that of Green is that we do not need to take the literal *ANSWER* with us during the whole search and yet we find the same or even more information.

4. Before using resolution rules we can split the theorem into a number of separate subtheorems which are independent and each of which is easier to prove. This we call the use of subgoals as mentioned by Ernst [5].

5. One can think of implementing a special rule for easier handling of the equality symbol. Something has been done using paramodulation [6] or E-resolution but not many practical results have been obtained.

6. Apart from the theorem proving program we can think of a program which translates sentences in first order predicate logic into clause form or even from a subset of Dutch into first order logic and then into clauses.

7. We want to apply other artificial intelligence techniques to theorem proving: for example the and-or-tree strategy of Slagle [7] (see also Kowalski [1]) and a learning system mentioned by Waterman [8].

References

[1] R. Kowalski and D. Kuehner. Linear resolution with selection functions. Memo 34, October 1970. Metamathematics Unit. Edinburgh University.

[2] R. Kowalski. Search strategies for theorem proving. Mach. Int. V (eds. Meltzer and Michie) (1970), 181-202.

[3] C. Green. The application of theorem proving to question-answering systems. June 1969. Art. int. group techn. note 8, SRI project 7494, Stanford.

[4] D. Luckham and N.J. Nilsson. Extracting information from resolution proof trees. Artificial Intelligence vol. 2, (Spring '71), 27-54.

[5] G.W. Ernst. The utility of independent subgoals in theorem proving. Information and Control 8, (1971), 237-251.

[6] L. Wos and G. Robinson. Paramodulation and set of support. Symp. on Aut. Dem. Lecture notes in mathematics 125. (1970), 163-191.

[7] J.R. Slagle and C.D. Farrell. Experiments in automatic learning for a multipurpose heuristic program. CACM vol. 14, 91-99.

[8] D. Waterman. Generalization learning techniques for automating the learning of heuristics. Artificial Intelligence vol. 1, (1970), 121-170.