17

An Interactive Theorem-Proving Program

John Allen and David Luckham
Computer Science Department
Stanford University

Abstract

We present an outline of the principal features of an on-line interactive theorem-proving program, and a brief account of the results of some experiments with it. This program has been used to obtain proofs of new mathematical results recently announced without proof in the *Notices of the American Mathematical Society*.

1. INTRODUCTION

The construction of the theorem-proving program we are going to discuss here is by no means completed. Rather, the program exists in a state of continual revision and extension, and that part of it which is running and yielding results at the moment is intended to form the nucleus of a much larger and more extensive automatic deduction system in the future. The program is being developed with a number of different questions and applications in mind.

First of all, some of the present generation of deduction programs are already capable of proving the sort of theorem of an elementary mathematical theory that appears in the textbooks either as a basic theorem or standard exercise (sometimes as a 'starred' exercise), and might reasonably be classified as 'somewhat' tricky. We have in mind the theorems of algebra and number theory obtained by programs discussed by Wos, Robinson and Carson (1965), Wos *et al.* (1967), and Luckham (1968). In addition, it is reported by Guard *et al.* (1969) that an open problem in modular lattice theory was solved with the aid of an on-line program (and it is interesting to note that, with reference to the initial set of axioms and hypotheses, this seems not to be the most difficult theorem that has been proved by a program so far). We are thus motivated to ask if, by adding a flexible interactive facility to a good deduction program, it is possible to construct a system that would be useful in investigating some fairly basic mathematics. This could take the form of searching for interesting consequences of unusual sets of

axioms, or for alternative proofs of known results, or of checking the steps of informal mathematical proofs for correctness. The interactive facility is to allow the user to monitor and direct the progress of a proof search at run time.

So far, we have used the program described below (sections 2 and 3) in interactive mode to prove some mathematical results announced in a recent issue of the *Notices of the American Mathematical Society* (Chinthayamma 1969). These are results concerning questions of dependence in the axiomatization of Ternary Boolean Algebra (henceforth called TBA) and in degree of difficulty appear quite similar to trigonometric identities. Interestingly, certain of the more 'important looking' deductions that turned up in the course of a proof search and that we chose to use in proceeding with the search, were (as we later found out) standard theorems of TBA published by Grau (1947).

There then follows a number of questions bearing upon the problem of finding out how the existing logical rules of inference and proof search strategies are best put to good use. For example, the program under discussion here uses the Resolution Principle (Robinson 1965a) as the fundamental rule of inference for that part of it dealing with pure first-order logic. There are many refinements of this principle aimed at improving the efficiency of the proof search and, for the most part, they do not work uniformly well, but tend to help or hinder the search depending on the problem involved. The same thing is true of the equality rule (Paramodulation (Wos and Robinson 1968)) which the program uses for problems involving the identity relation, and doubtless we shall find that it also applies to our extended system for multi-sorted logic. We need to gain a good deal more information about the relative merits of the various search strategies than we have at present. Also, the use of these two rules of inference together poses some additional questions. In some problems a significantly large set of deductions is generated by both rules, so that methods for reducing this duplication of effort would definitely be useful. We also need to know how different formalizations of a problem affect the efficiency with which proofs are obtained; sometimes a formalization within applied logic with identity is easier for the program to deal with than a pure first-order logic formulation, but this is not always true. There are basic programming questions. We do not have any theoretical study to tell us the most efficient way to implement the basic operations at the lowest level of the program (e.g., the tests for unification, alphabetic variants and subsumptions) so that even this is, at present, a matter for experimentation. And there is the problem of allowing for the incorporation of new strategies and inference rules into the program as they are developed. This rather chaotic state of knowledge forces upon us a number of basic decisions about the construction of the program and the way it is used. Our program places nearly all efficiency strategies under user control. At the start, when the program is first called, the user can choose which

strategies he wishes to apply to a particular problem, and he may alter his choice at any time during the search for a proof. This facility allows the user to take advantage of his own experience with the class of problem he is working on, and it also gives us an easy way to compare the performances of various combinations of strategies. Further decisions are to write the program in LISP, and to base its construction on a very simple fundamental flow diagram in which the subroutines for different operations are kept independent of each other, and operations of different types (e.g., choice strategies, inference rules, editing strategies; see figure 1) are programmed in strictly separate boxes of the diagram. While both of these decisions involve sacrificing some computational speed and efficiency, they make it relatively easy to introduce new operations, change the implementation of existing ones, and to extend the options available to the user, all of which one is almost certainly going to want to do.

Finally, the possibility of applying the deduction program to areas other than mathematical theorem-proving, for example information retrieval and projects in artificial intelligence requiring a certain deductive capability, should be mentioned. Here, it is usually necessary for the program to be able to reconstruct detailed information about any proof it generates, and we have allowed for this, so that, in particular, the question-answering techniques outlined by Green and Raphael (1968) and Luckham and Nilsson (1969) can be added easily.

The discussion below is organized into three sections and from time to time it assumes the reader has a slight acquaintance with the sort of terminology that appears in Robinson (1965a) and Luckham (1967, 1969). We shall deal only with the program for first-order logic with identity. In section 2 we describe the basic flow structure of the program and give a brief survey of what is known about the search strategies it uses. Section 3 contains a description of the on-line facility and the way we tend to use it at present, and some practical comments about the various strategies. Section 4 contains an analysis of some proofs of theorems in group theory and TBA.

2. BASIC STRATEGIES

Originally, the problem of running out of memory space was the crucial stumbling block of all the early theorem-provers, and led to concentration upon the development of space-saving strategies. This, together with the increase in the size of computer memories, has placed us in a situation where the computation time required to find a proof is now an equally important problem, especially if one has on-line interactive applications in mind. Indeed, with some of the more sophisticated strategies currently available, it is possible for the program to spend most of its time saving space! We have tended therefore to experiment with space-saving strategies which may not be the most restrictive, but have relatively simple and fast implementations,

and stand a good chance of saving computation time as well (see example 5 in Luckham (1969)).

The fundamental flow diagram of the program is given in figure 1. Each cycle or loop of its computation may be considered as beginning with a CHOICE operation when two or more statements (or clauses) from the list of clauses already in memory are chosen to make deductions from next. These clauses are given to one or both of the rules of inference, Resolution (Robinson 1965a) and the equality rule – Paramodulation (Wos and Robinson 1968) – depending on the way the user has things set up. Deductions following from the rules are passed on to separate editing operations associated with each of the rules, and then (if any remain) to a common editing operation. If a proof has been found, this is recovered by PROOF; otherwise the remaining new deductions are added to the lists by BOOK-KEEP, and the cycle is repeated. The computation may be interrupted any time a Book-Keep operation has just been completed, by the UPDATE routine which provides the standard interactive facilities (section 3).

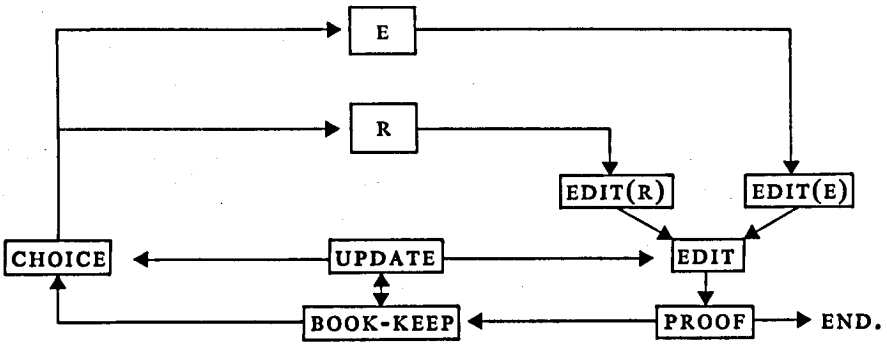

Figure 1

The strategies fall naturally into two classes, *choice* strategies and *editing* strategies, since they all operate either at the choice stage or the edit stage of the computation. At the moment, the editing strategies are, with one exception, the standard logical ones (see section 3). The principal choice strategies are refinements of the resolution principle in the sense of Luckham (1969) and they operate in the following way. Each refinement has associated with it a refining condition, $P(A, B)$, on pairs of clauses A, B (more generally, P operates on finite sets of clauses). A refinement allows only those resolvents C of Clauses A and B satisfying P to be generated. Thus,* if we let $R(A, B)$ denote the set of resolvents of A and B, if $R^n(S)$ denotes the set of all resolvents of level $i \leqslant n$ that can be generated starting from the initial set S of

* For further notation and terminology see Robinson (1965a) or Luckham (1968, 1969).

clauses, and if $\tilde{R}^n(S)$ denotes the subset of $R^n(S)$ that will be allowed by a refinement, we have,

$\tilde{R}^0(S) = S,$

$\tilde{R}^{n+1}(S) = \{C : C \in R(A, B) \;\&\; A, B \in \tilde{R}^n(S) \;\&\; P(A, B)\} \cup \tilde{R}^n(S).$ (1)

Trivially, $R^n(S)$ corresponds to the case where P is always true. The program includes the following refinements.

$\tilde{R}_1$: Resolution relative to a model

Let M be an interpretation or model over the domain of literals in the Herbrand expansion of S, $H(S)$ (*see* Luckham, 1969).

If each instance of clause A contains a literal occurring in M, we say M satisfies A (notation, $M \models A$). Then $\tilde{R}_1$ is defined by (1) with the condition,

$$P_1(A, B) =_{df} M \not\models A \vee M \not\models B.$$

Under this refinement, only resolvents of pairs of clauses, at least one of which is false of the model M are generated. The program allows any arbitrary LISP definition of a model to be given, but so far we have used only very simple models. The conditions, P, corresponding to some particularly simple (but nevertheless useful) models are programmed specially, e.g., the model consisting of all positive atoms, which yields the refinement called P_1-deduction in Robinson (1965b). The Set of Support (Wos, Robinson and Carson 1965) is a special case of $\tilde{R}_1$ (*see* Luckham 1969).

$\tilde{R}_2$: Resolution with merging

Andrews (1968) introduced the notion of a merge instance of a resolvent, C: if C is a resolvent of A and B, say $C = (A - \mathscr{L})\theta \cup (B - \mathscr{M})\theta$, and ρ is a substitution (possibly empty) such that $(A - \mathscr{L})\theta\rho \cap (B - \mathscr{M})\theta\rho \neq \varnothing$, then $C\rho$ is a merge instance of C, (or simply, a *merge*). Resolution with merging, $\tilde{R}_2$ is the refinement obtained from (1) by substituting the condition,

$$P_2(A, B) =_{df} A \in S \vee B \in S \vee A \text{ is a merge } \vee B \text{ is a merge.}$$

Thus $\tilde{R}_2$ omits resolvents of pairs of non-merge resolvents, and generates only those resolvents that follow from at least one axiom or one merge. This refinement only restricts the deductions at level 2 and beyond, and does not cut down the number of level 1 deductions, so it is important to use it in conjunction with something that restricts level 1, such as set of support. Also, the property of being a merge is a property of the deduction tree of a clause; a clause may occur first as a merge and later on as a non-merge, for example. This leads to some complications in implementing $\tilde{R}_2$ if one wants to take full advantage of the refinement. What we have done in this program is to implement a computationally slick but less restrictive refinement which we refer to simply as 'merging'.

$\tilde{R}_3$: Ancestry filter form

This refinement, originally introduced independently by Loveland (1969) and Luckham (1969) is obtained from (1) by the condition.

$$P_3(A, B) =_{df} A \in S \vee B \in S \vee A \in Tr(B) \vee B \in Tr(A),$$

where $Tr(A)$ denotes the deduction tree of A (cf. Andrews 1968, or Luckham 1969). Essentially, under this refinement, when it comes to choosing which clauses to resolve a clause A against, the ancestry tree of A is used to filter out the necessary clauses from the full list of all clauses in memory; A is resolved only with those clauses B that are axioms or occur in $Tr(A)$. The resulting proof trees generated in this way have a particularly simple structure (see figure 3) and are said to be in *Ancestry Filter Form* (AFF). The implementation of $\tilde{R}_3$ is very easy since it simply uses the mechanism for recovering proofs, which is already available in the program.

Each of these refinements has been proved to be logically complete. In fact, slightly stronger results are contained in the completeness proofs that have been given. Andrews (1968) gives a proof of the completeness of $\tilde{R}_2$ in conjunction with the set of support; the completeness proof for $\tilde{R}_1$ given in Luckham (1969) contains additional facts about the editing strategies; and proofs of the completeness of $\tilde{R}_3$ in conjunction with the set of support are given in Loveland (1969) and Luckham (1969). From a practical point of view, it is necessary to know as much as possible about the conditions under which the logical completeness of a refinement is maintained when it is used in conjunction with the standard editing strategies or with other refinements, for it is this knowledge that gives us some lines of possible further action when a proof search terminates without finding a proof. Essentially, the problem here is that one usually starts a proof search with a highly restrictive and logically incomplete combination of strategies. If this is unsuccessful, some of the parameters causing incompleteness are then relaxed and, of course, one has to know which these are. Some of the extra details that have proved useful may be summarized as follows.

Let us denote the strategy obtained by running the refinements $\tilde{R}_i$ and $\tilde{R}_j$ in conjunction, by $\tilde{R}_i \cap \tilde{R}_j$. It is proved in Luckham (1969) that both of the combinations $\tilde{R}_1 \cap \tilde{R}_2$ and $\tilde{R}_1 \cap \tilde{R}_3$ are *logically incomplete*. In other words, if one tries to extend the completeness results for $\tilde{R}_2$ or $\tilde{R}_3$ in conjunction with the set of support to the more restrictive refinement, $\tilde{R}_1$, one meets with failure. This is especially annoying since $\tilde{R}_1 \cap \tilde{R}_3$ has turned out to be quite useful (see section 4 and also example 5 in Luckham (1969)), and we do not have as yet any characterization of classes of problem for which this combination is complete. On the positive side, it has recently been proved by Kieburtz and Luckham (1969) that $\tilde{R}_2 \cap \tilde{R}_3$ is complete, and indeed a more restrictive strategy is also complete.

The interaction of the refinements with the editing strategies also needs careful scrutiny. For example, Loveland (1969) shows that if the initial set of hypotheses is not minimally inconsistent and the support set is chosen appropriately badly, then the conjunction of $\tilde{R}_3$ and set of support will generate a proof only if tautologies are admitted. It is also now well known that the various refinements will in general exclude the simplest proof trees from consideration, and admit proofs that are longer or contain longer

clauses or more complex functional terms. In this respect $\tilde{R}_1$ is well behaved. Corollaries of the proof of theorem 1 (Luckham 1969) are that $\tilde{R}_1$ remains complete with the elimination of tautologies, and that it does not increase the complexity of terms in the proof tree. The proof is also easily extended to show that $\tilde{R}_1$ remains complete when any clause which is subsumed by another is eliminated. Regarding $\tilde{R}_3$, it is shown by Kieburtz and Luckham (1969) that if the initial set of hypotheses satisfies some simple conditions, this refinement remains complete when both tautologies and clauses subsumed by an ancestor are eliminated. However, $\tilde{R}_3$ may increase the depth of nesting of function symbols in the terms occurring in the proof; that is, in order to find an $\tilde{R}_3$-proof from hypotheses S, the program may have to use terms more complex than those in the smallest set K such that $K(S)$ is inconsistent.

The refinements $\tilde{R}_1$ and $\tilde{R}_3$ are also usefully employed in the system for logic with identity to control the number of deductions generated by the equality rule. Here, some completeness results are known, but there are still many open questions.

3. THE INTERACTIVE FACILITIES

In this section, without going into fine details, we will attempt to give the reader an idea of how the program appears to a user and the sort of facilities that may be called upon in the course of a proof search. The program is written in LISP and currently runs on the PDP-10 time-sharing system at the Stanford Artificial Intelligence Project. (With very little effort the program can be altered so as to be compatible with a LISP System available on IBM 360-series Machines.) The basic input–output device for on-line use is an Information International Inc. display scope and keyboard (henceforth called a console). Teletypes are also available, but these tend to be far too slow for the volume of information that this program puts out; they can be used when the program is run in a sort of batch-processing mode as a background job to the system. Other peripheral equipment usually required during a run includes a line printer for printed output, and the disk which is used as a permanent storage device for the program and problems, and as temporary storage for output and checkpoint situations when the user wishes to store uncompleted computations in order to experiment with other strategies and directions of proof.

When the user first sits down at the console to start a proof, the initial dialogue with the program assigns values to a sequence of program variables. This determines which strategies the program will begin with; the settings of these variables may be changed at any time during a proof search thus changing the strategies. The sequence of variables appearing on the console is usually the following.

1. *Set of support* (Wos, Robinson and Carson 1965). The user specifies which axioms are to be supported. To reject a strategy, the user types, 'NIL'.

2. *Unit Preference* (Wos, Robinson and Carson 1965). The user specifies a bound on the level or depth to which the preference strategy is allowed to look ahead.

3. $\tilde{R}_3$*: Ancestry Filter Form* (Luckham 1969). If AFF is not selected, the program uses a level saturation scheme to generate resolvents and paramodulants, in which the order of generation is determined by the order in which clauses appear on the clause lists.

4. $\tilde{R}_1$*: Model – relative deduction* (Luckham 1969). If the user chooses $\tilde{R}_1$, he is asked to define the model M.

5. $\tilde{R}_2$*: Merging* (Andrews 1968). As previously mentioned, the merging strategy is less restrictive than Andrew's 'Resolution with Merging', but is easier to program and its execution is much faster.

6. *Ordering.* The user can ask the program to order the predicate letters. This imposes an ordering on the literals. The resolvents generated from two clauses may then be restricted to those in which only the literal highest in the ordering is eliminated from one of the clauses.*

7. *Equality Rule* (*Paramodulation* (Wos and Robinson) 1969). If a proof involving the equality rule is to be attempted, the user is asked to volunteer some further information:

(a) Which predicate letter is to be interpreted as representing the equality relation;

(b) A bound on the depth of nesting of function symbols to which the matching test involved in computing paramodulants is to be applied;

(c) A list of unit clauses, each consisting of a positive equality atom (called the demodulation list), to be used in the operation of *demodulation* (Wos *et al.* 1967). All equality atoms, $A_i=B_i$, are ordered so that A_i is at least as complex as B_i. The demodulation list, $\{A_i=B_i\}$ is then used as follows. If a clause C contains a (well-formed) subexpression which is an alphabetic variant of some A_i in this list, that subexpression is replaced by the corresponding variant of B_i. The procedure is iterated until no further variants of A's are found in the modified C. At present, this operation is applied to paramodulants, and it is the final descendant (or modification) of C that is retained (or more accurately, handed on to the other editing strategies). This user-controlled demodulation list has been a useful tool in reducing the number of trivial paramodulants.

The following editing strategies are also specified by the user at this time.

8. A bound on the *depth of nesting of function symbols* occurring in the terms of a clause. Any clause in which this bound is exceeded is rejected.

9. A bound on the *length* (number of literals) of a clause.

10. *Subsumptions.* The number of subsumption tests can be restricted in three ways. The test can be restricted to clauses of length less than a chosen bound, or so that only the newly generated clauses can be discarded (we call

* This is a variation of an idea of J.C.Reynolds.

this 'forward subsumptions') or so that it applies only to clauses of equal length (alphabetic variants). The implementation of the subsumption test is now sufficiently fast that our usual practice is not to restrict the test at all (except when AFF is employed; see section 2).

11. *Trivial Deductions.* The length-two clauses are treated specially. Let $\mathscr{L}$ be a list of unit clauses chosen by the user (e.g., a subset of the unit axioms is a common choice). Let C be a two-clause of the form $\{\neg l_1, l_2\}$. If there is a substitution σ_1 such that $\{l_1, \sigma_1\} \in \mathscr{L}$ and $\{l_2 \sigma_1 \sigma_2\} \in \mathscr{L}$ for some further σ_2, then C is rejected. We call this strategy, trivial deduction elimination. Note that the elimination of tautologies is a special case of this strategy if $\mathscr{L}$ is the set of all atoms in the Herbrand Universe. Since trivial deduction elimination is logically incomplete, we allow the user to choose the list $\mathscr{L}$. The strategy is an attempt to eliminate some of those clauses which, although not eliminable on purely logical grounds, are unlikely to contribute to a proof because the deductions following from them are already known. Obviously, we need more sophisticated (but fast) strategies to do this.

The value assigned to the final variable determines the amount of information the program gives the user about the deductions it makes; normally, the new resolvents and paramodulants are displayed on the scope as they occur, but this feature can be turned off. When the initial dialogue is completed and the strategies to be used have been chosen, the axioms and hypotheses are typed in from the console or read from a file on the disk, and the proof search begins. At any time, if a key is struck, the search will be interrupted and the program will wait for inquiries from the user. The current on-line command vocabulary allows the user the following options. (a) He can search the clause list, displaying each member either one at a time or in rapid succession, or he can display any particular clause if he knows its position in the list. (b) The ancestry tree (or proof) of any member of the clause list can be displayed. (c) The resolvents or paramodulants of any two chosen clauses can be computed and displayed. (d) Clauses can be deleted from or added to the clause list. (e) Finally, the user may enter a LISP READ–EVAL loop. In this state the settings of the strategy variables can be examined or redefined, the model (if $\tilde{R}_1$ is being used) can be changed, and in fact any arbitrary LISP computation (including one using the prover itself) can be executed.

Naturally, the user will attempt to find the most powerful combination of strategies for his problem. As a result of our own experience, we can make the following comments. The two most effective of our refinement strategies are $\tilde{R}_1$ and $\tilde{R}_3$. For almost all problems there seems to be a choice of model (and a very simple model) such that $\tilde{R}_1$ is of some significant help in getting a proof (*see also* Luckham 1968). $\tilde{R}_3$ has yielded some striking results. Although requiring the proof tree to satisfy the AFF condition almost always increases the ordinal depth of the tree by a significant factor (50 per cent is not uncommon), it has been the very simple examples with relatively short

proofs (level 5 or 6) that show unfavourable statistics for $\tilde{R}_3$. The AFF condition imposes a strong restriction on the growth of the number of deductions generated as a function of level. In many examples this seems to be linear. For more difficult problems it pays to search the extra levels with the AFF restriction. Even in cases where it turns out that more deductions are *retained* under the AFF condition, it often happens that far fewer deductions are *generated*, so that the deeper AFF proof-tree is obtained faster than the shorter trees because much less time is spent on editing computations. Also, many natural human proofs satisfy AFF; a point at which a resolvent of non-axiom clauses A and B is computed, and $A \in Tr(B)$, corresponds intuitively to an appeal to the lemma A. The combined strategy, $\tilde{R}_1 \cap \tilde{R}_3$, although incomplete, is very effective. If $\tilde{R}_3$ is used without $\tilde{R}_1$, it is important to add the set of support strategy. Merging, $\tilde{R}_2$, does not seem to be nearly as effective as the other two refinements, and it is almost always used in conjunction with at least one of them. The unit preference strategy appears to do more harm than good on the more difficult problems, usually because it leads to too many irrelevant clauses at the early levels. It is therefore used mainly to test for the end of a proof when the user is getting impatient. There are two 'end condition' strategies, which allow the user to search for unit proofs (unit preference) and vine-form proofs (i.e., proof trees in AFF and such that each deduction follows from an axiom). One way of using these strategies is to save the current state of the proof search, and then to introduce either strategy in an attempt to find a quick proof; if this is not successful, the user can continue with the original search. We hope to add some decision procedures for use in this way. Finally, the setting of the depth and length bounds (8 and 9 above) is usually crucial, since these essentially limit the subspace of the Herbrand expansion that will be searched for a proof. The user tends to start with these set as low as seems reasonable without excluding all proofs.

In practice we usually start a new problem with the $\tilde{R}_1 \cap \tilde{R}_2 \cap \tilde{R}_3$ combination, the model being chosen so as to generate a small number of level 1 deductions. As the search progresses, we try to keep track of any interesting deductions that may be relevant to the problem, and print their proofs on the line printer. These may be added as additional hypotheses when new searches are initiated. If a proof is found, the program displays it on the scope and stops. The search may fail to terminate successfully for three reasons. Too many clauses may be generated so that memory space is used up. This is unusual, but if it happens one generally tightens the bounds and tries again. The search may run out of time and the end condition strategies fail to produce a proof. If the maximal level searched is much larger than expected, this may mean that an incomplete combination of strategies is being used, so a different, possibly less restrictive, combination is tried. If, on the other hand, most of the time is spent in editing computations, one must use a more restrictive combination of strategies and bounds. Finally the program may stop

because it cannot make any more deductions. This case is treated as a sure sign of an incomplete strategy, and some of the bounds are relaxed or strategies turned off.

4. EXAMPLES OF PROOFS

In 1947 A.A.Grau (1947) presented a study of a ternary operation in Boolean Algebra, where 'ternary operation' was taken to mean a function of three variables defined on a set of elements and having values in the set. Grau defined the following algebraic system, K, which he called Ternary Boolean Algebra (TBA).

K is a system consisting of a set S and two operations under which the system is closed, one ternary, $(x\,y\,z)$, and the other unary, x', satisfying the axioms,

$$(x\,y\,(u\,v\,w)) = ((x\,y\,u)\,v\,(x\,y\,w)) \quad (1)$$
$$(y\,x\,x) = x \quad (2)$$
$$(x\,y\,y') = x \quad (3)$$
$$(x\,x\,y) = x \quad (4)$$
$$(y'\,y\,x) = x. \quad (5)$$

K is thus a homogeneous theory (all elements have equal significance) and has a realization within Boolean Algebra if the ternary operation, $(x\,y\,z)$, is interpreted as $(x \cap y) \cup (y \cap z) \cup (z \cap x)$. The axioms are therefore consistent. Furthermore, let p be a fixed element of S, and define $x \cap y = df.\ (x\,p\,y)$, and $x \cup y =_{df.} (x\,p'\,y)$. Then the system $\langle S, \cap, \cup, ' \rangle$ forms a Boolean Algebra with p as its null element and p' as its universe element.

A recent abstract in the *Notices of the American Mathematical Society* (Chinthayamma 1969) announces without proof that Grau's axioms 1–3 are sufficient to define a TBA, and that they are also independent, and some new sets of axioms are given. The program has been used to establish all of the dependence results announced by Chinthayamma between the various axiomatizations. Let us consider the dependence results concerning Grau's axioms.* Establishing axiom 4 from 1–3 is straightforward, and it takes the program only a few seconds to display a proof,

$$\begin{aligned}(x\,x\,y) &= (x\,x\,(y\,x\,x')), \text{ axiom 3}\\ &= ((x\,x\,y)\,x\,(x\,x\,x')), \text{ axiom 1}\\ &= ((x\,x\,y)\,x\,x), \text{ axiom 3}\\ &= x, \text{ axiom 2.}\end{aligned}$$

(The proof is actually displayed in a formal notation–see figure 2.)
The dependence of axiom 5 on 1–4 is a bit more problematic and was first obtained with a certain amount of interaction. While the theorem-prover was chewing on the axioms with the strategies set so that it was actually attempting a proof by contradiction, the user was also trying the problem. He found that he could in fact prove the result on the assumption, $y\,x\,y' = x$,

* At this point the reader might care to try these problems before reading on.

Axioms. (Note that the disjunction connective is omitted from the clauses. Interpret P(X, Y, Z, W) as (X Y Z)=W, and C(X) as X′.)

1. P(x8,x2,x9,x7) ¬P(x4,x5,x3,x9) ¬P(x4,x5,x1,x8) ¬P(x4,x5,x6,x7) ¬P(x1,x2,x3,x6)
2. P(x4,x5,x6,x7) ¬P(x8,x2,x9,x7) ¬P(x4,x5,x3,x9) ¬P(x4,x5,x1,x8) ¬P(x1,x2,x3,x6)
3. P(x2,x1,x1,x1,)
4. P(x1,x2,C(x2),x1)
5. P(x1,x1,x2,x1)
6. ¬P(C(B),B,A,A) (Negation of Grau Axiom 5)

PROOF:

SUPPORT=NIL
UNIT-PREFERENCE=NIL
MERGE=T
ORDER=NIL
ANCESTRY=T
DEPTH-BOUND=1
LENGTH-BOUND=4
MODEL=T
NIL
(P)
PARAMODULATE=NIL

NIL 1 2
1. P(C(x1),x1,x15,x15) 3 4
2. ¬P(C(B),B,A,A) AXIOM 6
3. P(C(x13),x11,x2,x2) ¬P(C(x13),x11,x13,x13) 5 6
4. P(x2,x1,x1,x1,) AXIOM 3
5. P(x8,x11,x1,x1) ¬P(x1,C(x2),x10,x8) ¬P(x10,x11,x2,x2) 7 8
6. P(x2,x1,x1,x1) AXIOM 3
7. P(x8,x11,x1,x7) ¬P(x1,C(x2),x10,x8) ¬P(x1,C(x2),x6,x7) ¬P(x10,x11,x2,x6) 9 10
8. P(x1,C(x2),x2,x1) 11 12
9. P(x1,C(x2),x2,x1) 13 14
10. P(x8,x2,x9,x7) ¬P(x4,x5,x3,x9) ¬P(x4,x5,x1,x8) ¬P(x4,x5,x6,x7) ¬P(x1,x2,x3,x6) AXIOM 1
11. P(x1,C(x2),x2,x7) ¬P(x1,x2,C(x2),x7) 15 16
12. P(x1,x2,C(x2),x1) AXIOM 4
13. P(x1,C,(x2),x2,x7) ¬P(x1,x2,C(x2),x7) 17 18
14. P(x1,x2,C(x2),x1) AXIOM 4
15. P(x8,x10,x1,x7) ¬P(x2,x1,x10,x8) ¬P(x2,x1,x10,x7) 19 20
16. P(x1,x2,C(x2),x1) AXIOM 4
17. P(x8,x10,x1,x7) ¬P(x2,x1,x10,x8) ¬P(x2,x1,x10,x7) 21 22
18. P(x1,x2,C(x2),x1) AXIOM 4
19. P(x8,x1,x9,x7) ¬P(x4,x5,x2,x9) ¬P(x4,x5,x1,x8) ¬P(x4,x5,x1,x7) 23 24
20. P(x2,x1,x1,x1,) AXIOM 3
21. P(x8,x1,x9,x7) ¬P(x4,x5,x2,x9) ¬P(x4,x5,x1,x8) ¬P(x4,x5,x1,x7) 25 26
22. P(x2,x1,x1,x1) AXIOM 3
23. P(x1,x1,x2,x1) AXIOM 5
24. P(x8,x2,x9,x7) ¬P(x4,x5,x3,x9) ¬P(x4,x5,x1,x8) ¬P(x4,x5,x6,x7) ¬P(x1,x2,x3,x6) AXIOM 1
25. P(x1,x1,x2,x1) AXIOM 5
26. P(x8,x2,x9,x7) ¬P(x4,x5,x3,x9) ¬(Px4,x5,x1,x8) ¬P(x4,x5,x6,x7) ¬P(x1,x2,x3,x6) AXIOM 1

QED

Figure 2

and could prove this if $x\,y'\,y=x$ were true. A quick check of the clause list found both $y\,x\,y'\neq x$ and $x\,y'\,y\neq x$ derived from the negation of axiom 5 (in other words, it had come to the same conclusions). The program was then given the two assumptions as well as axioms 1–4 and found a proof using only one of the assumptions, $x\,y'\,y=x$. This at least gave the user an alternative 'sub-goal'. However, an examination of this proof showed length and depth bounds of 4 and 1 respectively to be sufficient. Re-setting these parameters and attempting the original problem yielded a proof in 8 minutes; a translation into natural notation is the following.

$x=(x\,y\,y')$	axiom 3,
$=(x\,y\,(y'\,y'\,y))$	axiom 4,
$=((x\,y\,y')\,y'\,(x\,y\,y))$	axiom 1,
$=(x\,y'\,y)$,	axioms 3 and 2, (lemma)
$=(x\,y'\,(y'\,y\,y))$	axiom 2,
$=((x\,y'\,y')\,y\,(x\,y'\,y))$	axiom 1,
$=(y'\,y\,x)$	axiom 2 and lemma.

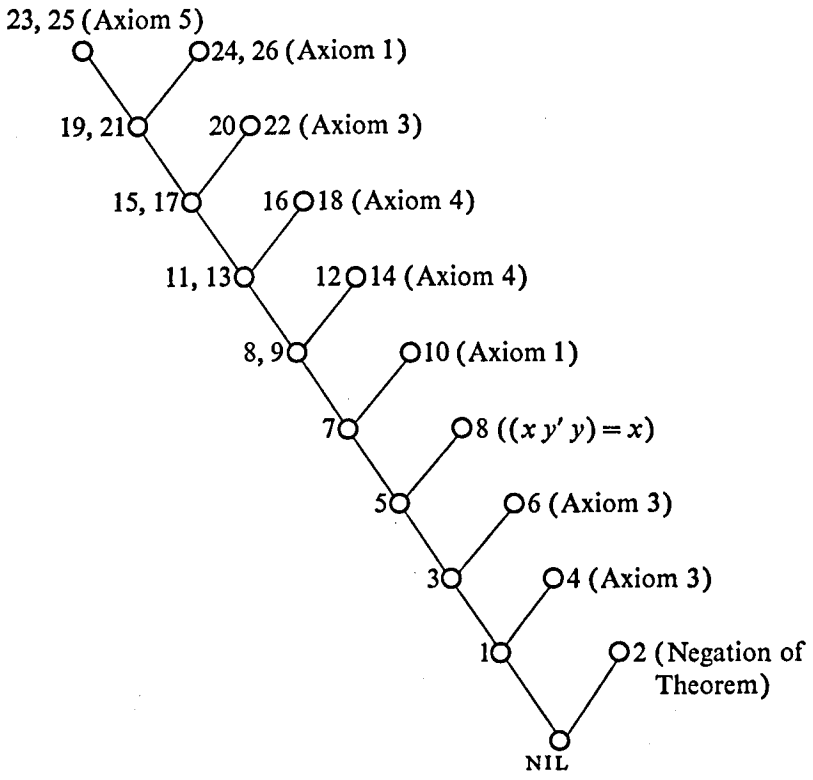

Figure 3. AFF proof-tree of proof in figure 2

These appear to us to be exactly the sort of problem mathematicians would much prefer to have machines to do. After all, it seems to have taken twenty years for these dependencies to become common knowledge, which is certainly a monument to the reluctance of people to negotiate such puzzles.

The formal proof of the second problem is given in figure 2, and its AFF proof tree is drawn in figure 3 (notice that the proof recovery outputs duplicate copies of isomorphic sub-trees; in figure 2, $Tr(8)=Tr(9)$).

Figure 4 contains some statistics comparing $\tilde{R}_1$ and $\tilde{R}_3$ for proofs of the lemma $x\,y'\,y=x$, from Grau's axioms 1–4. Here all strategies not mentioned were held constant at the settings indicated in figure 2; in each case when $\tilde{R}_1$ was used, the model was the set of all negative literals, and the set of support, when used, was equivalent to using this model at level 1. In all of these TBA problems, the Ancestry Filter proved to be a very powerful strategy.

	Clauses			
Choice Strategies	*Retained*	*Generated*	*Time*	*Proof Level*
$\tilde{R}_1 \cap \tilde{R}_2 \cap \tilde{R}_3$	153	711	174 sec.	5
$\tilde{R}_2 \cap \tilde{R}_3 \cap$ (Set of Support)	185	2244	280 sec.	4
(No Support) $\tilde{R}_2 \cap \tilde{R}_3$	185	2632	369 sec.	4
$\tilde{R}_1 \cap \tilde{R}_2$	236	17,289	30 min.	4 (No Proof, run stopped)

Figure 4

Finally, figure 5 is an example of a proof using both rules of inference. The problem is the following:

> Let K be a system consisting of a set S and a binary operation defined on S. If K is closed and associative and has an element e such that $e^2=e$, and every element S has a left inverse with respect to e and at most one right inverse with respect to e, then K is a group.*

The resolvents in the proof are marked, 'R', and the paramodulants are marked, 'E', or 'E, D' if they result from demodulation as well. We have used elementary algebra problems of this type (e.g., those considered in Wos, Robinson and Carson (1967) and Guard *et al.* (1969)) to compare the performances of our strategies on formulations in pure logic and in the first order theory of equality. Initial results suggest that the equality formulation proofs are shorter and more concise, but take longer to generate because of the amount of editing necessary on the paramodulants.

* This problem was suggested by Dr. L. Wos.

Axioms. (Interpret R(X,Y) as X=Y, F(X,Y) as the binary operation, and G(X) as the left inverse of X with respect to E).

1. R(F(F(X1,X2),X3),F(X1,F(X2,X3)))
2. R(F(E,E),E)
3. R(F(G(X1),X1)E)
4. R(X1,X1)
5. R(X2,X3) ¬R(F(X1,X3),E) ¬R(F(X1,X2),E)
6. ¬R(F(E,A),A)

SUPPORT=NIL
UNIT-PREFERENCE=NIL
MERGE=T
ORDER=NIL
ANCESTRY=T
DEPTH-BOUND=2
LENGTH-BOUND=2
MODEL=2
NIL
(R)
PARMODULATE=T
EQUAL-SYMBOL=R
PAR-DEPTH-BOUND=2

PROOF:

NIL 1 2
1. R(F(E,X2),X2) 3,4 E
2. ¬R(F(E,A),A)AXIOM 6
3. R(F(E,X2),F(X2,E)) 5,6 R
4. R(X1,F(X1,E)) 7,8 R
5. R(F(G(X2),F(E,X2)),E) 9,10 E,D(AXIOM 1)
6. R(X2,F(X1,E)) ¬R(F(G(X1),X2),E) 11,12 R
7. R(X2,F(X1,E)) ¬R(F(G(X1),X2),E) 13,14 R
8. R(F(G(X1),X1),E) AXIOM 3
9. R(X1,F(X1,E)) 15,16 R
10. R(F(G(X1),X1),E) AXIOM 3
11. R(F(G(X1),F(X1,E)),E) 17,18 E
12. R(X2,X3) ¬R(F(X1,X3),E) ¬(F(X1,X2),E) AXIOM 5
13. R(F(G(X1),F(X1,E)),E) 19,20 E
14. R(X2,X3) ¬R(F(X1,X3),E) ¬R(F(X1,X2),E) AXIOM 5
15. R(X2,F(X1,E)) ¬R(F(G(X1),X2),E) 21,22 R
16. R(F(G(X1),X1),E) AXIOM 3
17. R(F(G(X1),F(X1,X3)),F(E,X3)) 23,24 E
18. R(F(E,E),E) AXIOM 2
19. R(F(G(X1),F(X1,X3)),F(E,X3)) 25,26 E
20. R(F(E,E),E) AXIOM 2
21. R(F(G(X1),F(X1,E)),E) 27,28 E
22. R(X2,X3) ¬R(F(X1,X3),E) ¬R(F(X1,X2),E) AXIOM 5
23. R(F(G(X1),X1),E) AXIOM 3
24. R(F(F(X1,X2),X3),F(X1,F(X2,X3))) AXIOM 1
25. R(F(G(X1),X1),E) AXIOM 3
26. R(F(F(X1,X2),X3),F(X1,F(X2,X3))) AXIOM 1
27. R(F(G(X1),F(X1,X3)),F(E,X3)) 29,30 E
28. R(F(E,E),E) AXIOM 2
29. R(F(G(X1),X1),E) AXIOM 3
30. R(F(F(X1,X2),X3),F(X1,F(X2,X3))) AXIOM 1

QED

Figure 5

Acknowledgement

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Department of Defense (SD–183).

REFERENCES

Andrews, P.B. (1968) Resolution with merging. *J. Ass. comput. Mach.*, **15**, 367–81.

Chinthayamma (1969) Sets of independent axioms for a ternary Boolean algebra. *Notices of Amer. Math. Soc.*. **16**, 69T–A69, 654.

Grau, A.A. (1947) Ternary Boolean Algebra. *Bull. Amer. Math. Soc.*, **53**, 567–72.

Green, C. & Raphael, B. (1968) The use of theorem-proving techniques in question–answering, *Proc. 23rd National Conference ACM*. Washington, D.C.: Thompson Book Co.

Guard, J.R., Oglesby, F.C., Bennett, J.H. & Settle, L.G. (1969) Semi-automated mathematics. *J. Ass. comput. Mach.*, **16**, 49–62.

Kieburtz, R. & Luckham, D. (1969) *Compatibility of Refinements of the Resolution Principle* (in press).

Loveland, D. (1969) A linear format for resolution. *Proceedings IRIA Symposium on Automatic Demonstration.* Versailles, France, December 1968. Springer-Verlag (in press).

Luckham, D. (1967) The resolution principle in theorem-proving. *Machine Intelligence 1*, (eds Collins, N.L. & Michie, D.). Edinburgh: Oliver and Boyd.

Luckham, D. (1968) Some tree-paring strategies for theorem-proving. *Machine Intelligence 3*, pp. 95–112 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Luckham, D. (1969) Refinement theorems in resolution theory. *Proceedings IRIA Symposium on Automatic Demonstration.* Versailles, France, December 1968. Springer-Verlag (in press).

Luckham, D. & Nilsson, N. (1969) *On extracting information from resolution proof trees.* Stanford Artificial Intelligence Project Memo (in press).

Robinson, J.A. (1965a) A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, **12**, 23–41.

Robinson, J.A. (1965b) Automatic deduction with hyper-resolution. *International Journal of computer Mathematics*, **1**, 227–34.

Wos, L., Robinson, G., & Carson, D. (1965) Efficiency and completeness of the set of support strategy in theorem-proving. *J. Ass. comput. Mach.*, **12**, 536–41.

Wos, L., Robinson, G., Carson, D. & Shalla, L. (1967) The concept of demodulation in theorem-proving. *J. Ass. comput. Mach.*, **14**, 698–704.

Wos, L. & Robinson, G. (1969) Paramodulation and set of support. *Proceedings IRIA Symposium on Automatic Demonstration.* Versailles, France, December, 1968, Springer-Verlag (in press).