MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A.I. LABORATORY

Artificial Intelligence
Memo No. 268

October 1972

A HUMAN ORIENTED LOGIC FOR AUTOMATIC THEOREM PROVING

Arthur J. Nevins

A HUMAN ORIENTED LOGIC FOR AUTOMATIC THEOREM PROVING*

by Arthur J. Nevins

1. Introduction

The automation of first order logic has received comparatively little attention from researchers intent upon synthesizing the theorem proving mechanism used by humans. The dominant point of view [15], [18] has been that theorem proving on the computer should be oriented to the capabilities of the computer rather than to the human mind and therefore one should not be afraid to provide the computer with a logic that humans might find strange and uncomfortable. The preeminence of this point of view is not hard to explain since until now the most successful theorem proving programs have been machine oriented.

Nevertheless, there are at least two reasons for being dissatisfied with the machine oriented approach. First, a mathematician often is interested more in understanding the proof of a proposition than in being told that the proposition is true, for the insight gained from an understanding of the proof can lead to the proof of additional propositions and the development of new mathematical concepts. However, machine oriented proofs can appear very unnatural to a human mathematician thereby providing him with little if any insight. Second, the machine oriented approach

---

*This work was conducted at The George Washington University Program in Logistics and supported under Contract N00014-67-A-0214, Task 0001, Project NR 347 020, U.S. Office of Naval Research. The author also is grateful to The George Washington University and The University of Maryland for the use of their computational facilities.

has failed to produce a computer program which even comes close to equaling a good human mathematician in theorem proving ability; this leads one to suspect that perhaps the logic being supplied to the machine is not as efficient as the logic used by humans.

The approach taken in this paper has been to develop a theorem proving program as a vehicle for gaining a better understanding of how humans actually prove theorems. The computer program which has emerged from this study is based upon a logic which appears more "natural" to a human (i.e., more human oriented). While the program is not yet the equal of a top flight human mathematician, it already has given indication (evidence of which is presented in section 9) that it can outperform the best machine oriented theorem provers.

Some work was begun in [2], [3], [7], [10] and [17] directed toward the introduction of a reasoning by cases mechanism into automatic theorem proving. One can give many examples where humans use such a mechanism. Thus, in proving set A is identical to set B, usually one will attempt to prove two cases: (1) A is a subset of B and (2) B is a subset of A. In proving that a system is a group, usually one will attempt to prove two cases: (1) if two elements are in the group, then the product is in the group and (2) if an element is in the group, then its inverse is in the group. In proving a theorem by induction, one will prove a basis case (i.e., that it is true for say n=1) and also an induction case (i.e. if it is true for n, then it must be true for n+1).

However, some serious obstacles have prevented the effective use of reasoning by cases by previous automatic theorem provers. These

obstacles relate to the overall (i.e., global) organization of these programs. Any problem solving program must have an executive routine which controls the allocation of its computational resources. The executive routine must have a procedure for (1) determining when and how to create a goal and (2) preventing an explosion of goals from taking place. Previous theorem proving executives [3], [10], [17] made the mistake of associating a goal with every new formula that got generated.[1] By contrast, the present program first makes an intensive attempt to solve a goal (which typically will generate a number of new formulas for further processing under the control of a local executive) before deciding to generate a new goal. As viewed by the global executive, new goals get created only when a goal gets split into subgoals (where the logical basis for the split is a reasoning by cases argument). The rationale behind this is that the resources for solving a goal without splitting are considerable (i.e., repeated use of modus ponens, the equality inference rule, rules for simplifying formulas, etc.) and therefore one should provide substantial opportunity for solving the goal before introducing additional goals.

---

[1] This mistake was not made in [2] and [7] where procedures were presented for breaking a theorem into component cases which then are attacked by standard theorem proving techniques. However, these procedures do not provide the means by which the application of the standard techniques can interact with the mechanism that splits the theorem into cases. Thus, if the standard techniques are inadequate for solving a case, the information generated during the unsuccessful attempt is lost and will not contribute to the process of creating new subcases.

Nevertheless, the splitting of goals into subgoals is often necessary and if not controlled it could overwhelm the executive with more goals than it could handle. The key to controlling this growth (and it has demonstrated its value in the present program) stems from the recognition of the fact that it is not always necessary to solve each subgoal created by a split in order to solve an ancestor goal. For example, suppose the program knows that it can prove a goal G by proving A and B. So it first attempts to prove A and generates a number of new formulas in this attempt. However, suppose that, although it still cannot prove A, it nevertheless knows that it also could prove G by proving C and D. The program then would attempt to prove C and D and would view these attempts as subgoals of A. The reason C and D are regarded as subgoals of A is that their proofs could depend upon information derived from A. In this case, the solution of C and D would mean that we still would have to prove B in order to conclude G. However, if the proofs of C and D did not utilize information derived from A, then we would not have to prove B and could bring the proof of the ancestor goal G to an immediate conclusion. Similarly, if the proof of C did not depend upon information derived from C, we could skip over the proof of D.

From the viewpoint of human problem solving, we can regard these splits as "strategies" by which <u>some</u> ancestor goal is to be solved (it need not be the immediate ancestor). The program (and also a human) may have a number of different strategies under consideration <u>simultaneously</u>. In the course of implementing these strategies, the program discovers which strategies are of assistance to other strategies, which strategies must be completed, and which strategies can be discarded.

2. Preliminary Concepts

The reader is referred to [11] for a rigorous treatment of the basic definitions and ground rules used by most programs which prove theorems in the first order predicate calculus. An informal description is presented in this section in order to make sure that the reader is acquainted with the basic ideas.

We use the four logical connectives $\sim$ ("not"), $\supset$ ("implies"), $\wedge$ ("and"), $\vee$ ("or") as well as the universal quantifier $\forall$ ("for all") and the existential quantifier $\exists$ ("there exists"). We assume an infinite supply of variables, constants, functions and predicates.

Terms: A variable is a term, a constant is a term, and a function $f(t_1,\ldots,t_n)$ is a term provided that its arguments $t_1,\ldots,t_n$ are all terms.

Atomic formulas: A predicate $P(t_1,\ldots,t_n)$ is an atomic formula provided that its arguments $t_1,\ldots,t_n$ are all terms.

Literals: An atomic formula A is a literal and the negation of an atomic formula $\sim A$ is a literal.

Formulas: An atomic formula is a formula. If A and B are formulas, then $\sim A$, $A \supset B$, $A \wedge B$, and $A \vee B$ are formulas. If A(x) is a formula that may depend upon some variable x, then $\forall x A(x)$ and $\exists x A(x)$ are formulas.

Given the desire to prove that the formula A_{n+1} logically follows from the formulas $A_1,A_2,\ldots,A_n$, we will assume that the formulas $A_1,A_2,\ldots,A_n$, $\sim A_{n+1}$ are all true and then attempt to derive a contradiction. The first

step is to convert each of the formulas $A_1, A_2, \ldots, A_n, \sim A_{n+1}$ to prenex form using a standard procedure [9]. A formula which is in prenex form has all its quantifiers (if any) at the front of the formula. The next step is to remove first all the existential quantifiers and then all the universal quantifiers. An existential quantifier is removed by replacing the existential variable it quantifies by a function of those universal variables whose quantifiers appear to the left of the existential quantifier in question [11]. For example, in the formula $\forall u \forall v \exists w \forall x P(u,x,w,v)$, the existential quantifier which quantifies the variable w would be removed and w would be replaced by a function of u and v, say f(u,v), since these variables are universally quantified to the left of w. The resulting formula therefore would be $\forall u \forall v \forall x\ P(u,x,f(u,v),v)$. Universal quantifiers then are dropped with the understanding that the variables in question are to be given universal interpretations (i.e. if A(x) is assumed to be true, then it is assumed to be true for all possible values of the variable x).

At the heart of all current theorem proving programs is the use of "matching" routines. For example, suppose the program established that the formulas A(x,a,x) and $\sim$A(b,y,b) both follow from the assumptions of the problem where x and y are variables and a and b are constants. A matching routine then would determine for the program that variable x should be set equal to b and variable y should be set equal to a in order to eliminate the sources of difference between the formula A(x,a,x) and the formula A(b,y,b). Since A(x,a,x) must be true for x=b (as it is true for all values of x) and $\sim$A(b,y,b) must be true for y=a (as it is true for all values of y), this means that the formula A(b,a,b) would contradict the formula $\sim$A(b,a,b) and the program would have obtained a proof.

3. The Logical Deduction Rules

The use of matching routines is implicit in the rules of inference to be described in this section. Thus, when expressions E and E' appear in the statement of an inference rule, it should be understood that E and E' represent expressions which have been made identical by means of a matching routine. For example, the application of $A(b,y,b)$, $A(x,a,x) \supset B(x)$ to rule R6 would mean that $A(b,y,b)$ would be made identical to $A(x,a,x)$ by setting $x=b$, $y=a$ and this would result in the output $B(b)$ where $A(x,a,x)$, $A(b,y,b)$, $B(x)$ and $B(b)$ play the role of A, A', B, and B' respectively.

As described in section 2, we begin with a set of formulas all of which are assumed to be true. New formulas are established with the help of rules R2 through R12. Rule R1 determines when a problem has been solved.

R1. A problem is solved when it has been established that both the literals A' and $\sim A$ are true.

R2. Replace formula $\sim\sim A$ by A.

R3. Replace formula $A \wedge B$ by A,B.

R4. Replace formula $\sim(A \vee B)$ by $\sim A$, $\sim B$ (i.e. if one wishes to prove $A \vee B$, then either prove A or prove B).

R5. Replace formula $\sim (A \supset B)$ by A, $\sim B$ (i.e. if one wishes to prove $A \supset B$, then assume A and prove B).

R6. (Modus ponens) If it has been established that A' and $A \supset B$ are true where A is a literal, then add B' to the set of true formulas.

R7. (Modus ponens) If it has been established that $\sim B'$ and $A \supset B$ are true where B is an atomic formula, then add $\sim A'$ to the set of true formulas.

R8. (Modus ponens) If it has been established that B' and $A \supset \sim B$ are true where B is an atomic formula, then add $\sim A'$ to the set of true formulas.

R9. (Reasoning by cases) Split $A \vee B$ into case A and case B as shown in section 4.

R10. (Reasoning by cases) Split $\sim(A \wedge B)$ into case $\sim A$ and case $\sim B$ as shown in section 4.

R11. (Equality relation) If it has been established that r=t and A(r') are true where A(r') is a literal that depends upon the term r', then add A(t') to the set of true formulas. See section 7 for a more complete discussion of this rule. Also, see section 8 for the treatment of functions that are either both associative and commutative or just associative as the program has special routines which incorporate such functions into the equality relation.

R12. If it has been established that $P(t_1,\ldots,t_n)$ and $\sim P(t_1',\ldots,t_n')$ are true literals where P is not the equality predicate and if t_i has been made identical to t_i' by means of a match for all $i \neq j$ but this match fails for i=j, then add $\sim(t_j=t_j')$ to the set of true formulas.

Of the machine oriented approaches to automatic theorem proving, by far the most important have been those based upon the resolution principle [15]. Although resolution is based upon a single inference rule for generating new formulas, it is convenient to regard it as two rules -- called unit and non-unit resolution respectively. Unit resolution says that if it has been established that A_1' and $\sim A_1 \vee A_2 \vee \ldots\ldots A_r$ are

true where $A_1, A_2, \ldots, A_r$ are literals, then we can add $A_2' \vee A_3' \vee \ldots . A_r'$ to the set of true formulas.[2] From the standpoint of rule R9, the formula $\sim A_1 \vee A_2 \vee \ldots . A_r$ can be thought of as representing r cases given by the r literals $\sim A_1, A_2, \ldots, A_r$. The application of unit resolution to the formula $\sim A_1 \vee A_2 \vee \ldots . A_r$ then has the worthwhile property that it produces as output a formula with one less case (i.e. $A_2' \vee A_3' \vee \ldots . A_r'$ has only r-1 cases). On the other hand, non-unit resolution takes as input the two formulas $\sim A_1 \vee A_2 \vee \ldots . A_r$ and $A_1' \vee B_2 \vee \ldots . B_t$ and generates as output $A_2' \vee \ldots . A_r' \vee B_2' \vee \ldots . B'_t$ which is a formula which rule R9 would regard as representing r+t-2 cases. Since $r \geqq 2$ and $t \geqq 2$, it follows that $r+t-2 \geqq \max(r,t)$. This means that the output formula of a non-unit resolution will have at least as many and often more cases than either of its input formulas. Since the ultimate objective is to eliminate all cases from some formula, it is understandable that non-unit resolution would be much less effective than unit resolution and indeed researchers soon gave preference to unit resolution when generating new formulas [19], [4]. Now, since $\sim A \vee B$ is logically equivalent to $A \supset B$, unit resolution at least is related to modus ponens (see rules R6 through R8) which is a common form of human reasoning whereas non-unit resolution appears very unnatural to a human. The suspicion is quite strong that humans do not use non-unit

---

[2] As in the other deduction rules described in this section, we are adopting the convention which considers expressions written as E and E' as having been made identical by means of a matching routine.

resolution when making deductions but do use a form of unit resolution precisely because non-unit resolution is so much less efficient than unit resolution.

There is no mechanism in resolution for breaking a difficult problem into two or more simpler subproblems. Yet this is a common feature of human problem solving. The value of such a mechanism is that it is generally easier to solve a number of simple problems than it is to solve a single hard problem. The present program uses reasoning by cases as the logical basis for generating a goal-subgoal hierarchy and this is described in section 4. However, the following simple example can help to illustrate the main ideas.

Example: We wish to obtain a contradiction from the following six axioms (A1 through A6) where x, y, and z represent variables and a, b, and c represent constants.

A1. $x*(y*z) = (x*y)*z$

A2. $P(x*y) \vee \sim P(x) \vee \sim P(y)$

A3. $P(a)$

A4. $P(b)$

A5. $P(c)$

A6. $\sim P(a*(b*c))$

The program first attempts to find a contradiction by repeated use of all the deduction rules except R9 and R10. At this stage, the only new formula it can generate is A7 which is obtained by substituting A1 into A6 using rule R11.[3]

---

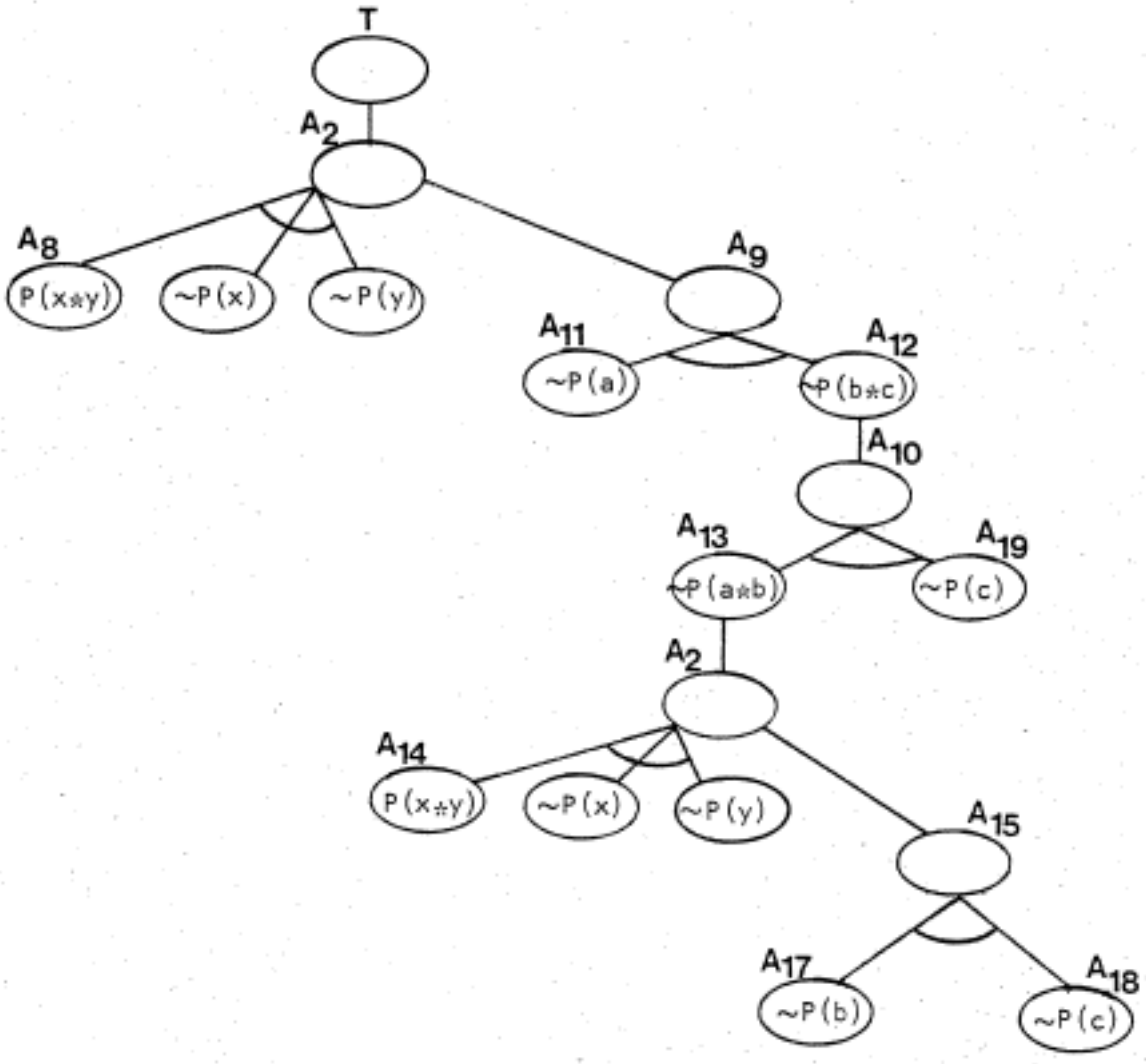
[3]Strictly speaking, we are substituting the right side of A1 into A6 after the match between $x*(y*z)$ and $a*(b*c)$ produced x=a, y=b, and z=c.

A7. $\sim P((a*b)*c)$

Actually, the program would not have considered A7 as distinct from A6 if it had been told that the symbol * satisfies the associative law A1 (see section 8). However, for purposes of exposition, we have assumed in this example that the program has no explicit knowledge that * is associative.

The remainder of this proof can be understood best by referring to Figure 1 which describes the goal-subgoal hierarchy that was generated from this example.

Figure 1



The node labeled T in Figure 1 represents the top level goal which consists of the desire to obtain a contradiction from the axioms A1 through A6. Since after generating the formula A7 the program finds that it has exhausted its resources without obtaining a contradiction, it now turns to a reasoning by cases argument (i.e. rules R9 or R10). Axiom A2 is chosen to be split into cases which become subgoals of T as shown in Figure 1. This generates A8.

A8. $P(x*y)$. Case 1 of A2.

Once again, the program attempts to find a contradiction using all the resources at its disposal except rules R9 and R10. This attempt discovers two distinct contradictions. The first contradiction is between A6 and A8 for $x=a$, $y=b*c$ and generates the formula A9.

A9. $\sim P(a) \vee \sim P(b*c)$. Cases 2 and 3 of A2 for $x=a$, $y=b*c$.

The second contradiction is between A7 and A8 for $x=a*b$, $y=c$ and generates the formula A10.

A10. $\sim P(a*b) \vee \sim P(c)$. Cases 2 and 3 of A2 for $x=a*b$, $y=c$.

After no more new formulas can be generated nor new contradictions obtained, the program discharges all formulas that resulted from the attempt to solve case 1 of A2 but formulas A9 and A10 now replace cases 2 and 3 of A2. This means that A1 through A7 together with A9 and A10 are the only formulas under consideration by the program once case 1 of A2 has been discharged. The program now chooses to split A9 into cases as shown in Figure 1. This generates A11.

A11. $\sim P(a)$. Case 1 of A9.

A contradiction is obtained between A3 and A11. This results in A11 being

discharged and A12 generated.

A12. ~P(b*c). Case 2 of A9.

Again, an attempt is made to solve this last case (i.e. "prove P(b*c)") without further use of a reasoning by cases argument (i.e. rules R9 or R10). This attempt fails. In fact, it even fails to generate any new formulas. However, rather than abandon its attempt to solve this case, the program splits A10 and considers the formulas A13 and A19 that result from the split as subgoals of A12 as shown in Figure 1. Although the solution of A13 and A19 would solve T, they are treated as subgoals of A12 since the program has no a priori way of knowing whether the information derived from A12 will be needed in the solutions to A13 and A19 . The program now attempts to solve A13.

A13. ~P(a*b). Case 1 of A10.

The attempt to solve A13 also fails so an attempt is made to split A2. Although the solution of the goals created by the split of A2 would solve T, these goals are attached as subgoals of A13 as shown in Figure 1 since information provided by their ancestor goals might help in their solution. Indeed, it should be recalled that A2 was split once before and resulted in the formulas A9 and A10. The reason another attempt is made to split A2 is that new formulas (i.e. A12 and A13) now are available which were not available during the previous attempt.

The program now attempts to solve A14.

A14. P(x,y). Case 1 of A2.

This second attempt to split A2 results in two new contradictions. The first contradiction is between A12 and A14 for x=b, y=c and generates the formula A15.

A15. $\sim P(b) \vee \sim P(c)$. Cases 2 and 3 of A2 for x=b, y=c.

The second contradiction is between A13 and A14 for x=a, y=b and generates the formula A16.

A16. $\sim P(a) \vee \sim P(b)$. Cases 2 and 3 of A2 for x=a, y=b.

After no more new formulas can be generated nor new contradictions obtained, the program discharges all formulas that resulted from the attempt to solve case 1 of A2 but formulas A15 and A16 now replace cases 2 and 3 of A2. The program now chooses to split A15 into subgoals of A13 as shown in Figure 1.

A17. $\sim P(b)$. Case 1 of A15.

A contradiction is obtained between A4 and A17. This results in A17 being discharged and A18 generated.

A18. $\sim P(c)$. Case 2 of A15.

A contradiction is obtained between A5 and A18. Now since both the generation of A17 and A18 as well as the solution of these goals in no way depended upon the parent goal A13, the program can conclude the proof of A12 immediately without attempting the proof of A19.[4] By contrast, the solution of A12 did involve information derived from A12 since its solution depended upon A15 which was derived from A12. However, since A12 is the last goal in the split of A9, this allows the program to conclude that it has a solution to the top level goal T.

---

[4]Although in this simple example the proof of A19 would have been trivial, one cannot expect to be so fortunate in general.

The above example by its very simplicity failed to emphasize the role played by the equality inference rule and modus ponens in the present theorem proving program. A very important feature of the program is its mechanism for controlling the number of formulas generated by the equality inference rule and this is described in section 7. Also, a formula coded as $A \supset B$ is used with modus ponens as a means for solving a case whereas the same formula coded as $\sim A \vee B$ would be used in generating the separate cases $\sim A$ and B. One encounters a number of formulas such as $x=y \supset x*z = y*z$ whose usefulness lies in the help they provide in solving a case rather than as a means for generating separate cases. The program has facilities for exploiting such formulas and these facilities are described in section 5.

4. Reasoning by Cases

The initial data given to the program is the list (L_0, L_1) where L_1 is a list of formulas from which a contradiction is to be found and L_0 is an empty list. A formula F is removed from L_1 and prepared as a possible input to one or more of the deduction rules. F is not allowed to be of the type $A \vee B$ or $\sim(A \wedge B)$ since we wish to make a concerted attempt to solve the problem before splitting it into subproblems. If F can be applied successfully to a rule which requires only one input (i.e. rule R2, R3, R4 and R5), then F is discarded and the output from this application of the deduction rule is placed on L_1. For example, if F is the formula $\sim(A \supset B)$, then F would be discarded and instead the formulas A and $\sim B$ would be inserted on L_1.

If F cannot be applied successfully to a one input deduction rule, then an attempt is made to apply it to a two input rule by using a formula from L_0 as the second input. A successful application to such a rule would

mean that the output would be placed on L_1 unless the rule is R1 in which case a contradiction would be found. After every formula on L_0 has been paired with F as the second input to each deduction rule, the formula F gets placed on L_0.[5] A new formula F now is removed from L_1 and the procedure repeats itself until either a contradiction has been obtained, the list L_1 has been exhausted except for formulas of the type $A \vee B$ or $\sim(A \wedge B)$, or a time limit has been exceeded.[6]

If a contradiction cannot be found in the above manner, then an appeal must be made to rules R9 or R10. Rule R9 splits a disjunction of the form $B_1 \vee B_2 \vee B_3 \ldots \vee B_r$ whereas rule R10 splits a negative conjunction of the form $\sim(B_1 \wedge B_2 \wedge B_3 \ldots \wedge B_r)$. If at this stage there are no disjunctions or negative conjunctions on L_1, then the program must admit failure. Otherwise, a disjunction or negative conjunction is taken from L_1 and a list K_1 is constructed as follows. K_1 is a list of four elements. $K_{1,1}$ (i.e. the first element on list K_1) initially represents the list $(B_1, B_2, B_3, \ldots, B_r)$. $K_{1,2}$ is either the symbol $\vee$ or the symbol $\wedge$ depending upon whether an application of rule R9 or R10 is being made. $K_{1,3}$ initially is an empty list. $K_{1,4}$ initially is a list of the variables appearing in any of the formulas $B_1, B_2, B_3, \ldots, B_r$. Although the following discussion will assume that rule R9 is being applied, the treatment of rule R10 is very similar by virtue of the logical equivalence of $\sim(A \wedge B)$ with $\sim A \vee \sim B$.

---

[5] As will be shown in section 7, a successful application of rule R11 to F often will cause F to be discarded immediately in favor of the resulting output which is placed on L_1.

[6] The action taken by the program if the time limit is exceeded is described at the end of the present section.

The state of affairs is summarized now by the list (L_0, K_1, L_1, L_2) where L_2 is a list consisting of the first formula on list $K_{1,1}$ (which in this case is B_1) together with all the disjunctions and negative conjunctions of L_1. As before, a proof is attempted first without using rules R9 or R10. Only this time, the output of a deduction rule is placed on L_2 instead of on L_1, a formula F which is removed from L_2 and applied to the deduction rules is either eventually discarded or transferred to L_1 instead of to L_0, and the deduction rules which require two inputs will pair the formula F with formulas from either L_1 or L_0 instead of just from L_0.

Suppose a contradiction is obtained. How should we proceed? An obvious method would be to (1) remove B_1 from the list $K_{1,1}$ of cases which have not yet been solved and place it on the list $K_{1,3}$ of cases which have been solved already, (2) reestablish L_1 as it had been just prior to the creation of K_1, (3) erase L_2 and replace it with a new list L_2 consisting of the next case B_2(which is now the first formula on list $K_{1,1}$) together with all the disjunctions and negative conjunctions of L_1, and (4) search for a contradiction in a similar manner as in case 1 when B_1 had been the controlling hypothesis.

The difficulty with the above four step method is that, in the solution to case 1, a variable x appearing in B_1 may have been set equal to some term t; if this variable x also appears in one of the formulas $B_2, B_3, \ldots\ldots, B_r$, then this specification that x should equal t must prevail also in at least one of these subsequent cases. In order to insure that x in fact does equal t in this latter case, the variables of

B_1, B_2,....., B_r are classified into types according to whether the variable is to be given a "universal" or "existential" interpretation.

Normally, all the variables would have universal interpretations (i.e., if formula A(x) is true, then it is true for all values of the variable x), since all the existential variables already were eliminated by the procedure described in Section 2. However, in attacking Case 1, we now place an existential interpretation on those variables which appear in both B_1 and one or more of the formulas B_2, B_3,....., B_r (i.e., if $A(x) \vee B(x)$ is assumed to be true, then we must find some value of x which will permit a solution to each case). For this purpose, we will say that an existential variable has been specified if it is set equal to a term t where t is not a universal variable.[7]

Now in first attacking Case 1, we seek applications of the deduction rules (except rules R9 and R10) and subject these rules to the restrictions that no existential variable may be specified unless the specification occurs during a successful application of rule R1 (i.e., we specify an existential variable only when so doing will assure the proof of a subproblem thereby providing a valuable restriction on the generation of subproblems).

If a successful solution is found which does not specify any existential variables, then we proceed directly to Case 2 by the four

---

[7] Note that if an existential variable has been set equal to a universal variable, then the existential variable is still free to assume any value which can be assumed by the universal variable. However, since a universal variable by definition can assume any value, this means that this existential variable has not really been "tied down" or "specified" upon being set equal to the universal variable.

step method described above (i.e., we assume B2 and discard all formulas that were generated during case 1 when B1 was the hypothesis of the case). The reason we go directly to case 2 when no existential variable was specified is that the solution of case 1 would not have committed any variable appearing in any of the subsequent cases.

However, suppose the successful solution necessitated the specification of at least one existential variable. Then we would not proceed directly to case 2. Instead, the use of existential variables enables the program to find a number of different solutions to case 1 during the same attempt at solving the case; this avoids the duplication of effort that would appear if the program were to proceed directly to case 2 as soon as it found a single solution only to find later that it must generate still more solutions to case 1 in order to solve the original problem. In particular, let x represent the vector of existential variables. Each time a solution to case 1 is obtained which resulted in a different specification t of the vector x, we generate the disjunction of the remaining cases for this specification (i.e., we generate the formula $B_2(t) \vee B_3(t) \ldots\ldots \vee B_r(t)$).[8] After no more solutions can be found, the program discharges B1 and all formulas generated from B1 except those disjunctions $B_2(t) \vee B_3(t) \ldots\ldots \vee B_r(t)$ each of which represents the remaining cases associated with a different solution vector t. Rather than consider any of the remaining cases 2 through r, the program later

---

[8] Any existential variable which was left unspecified by this solution would be replaced by a new universal variable in the formula $B_2(t) \vee B_3(t) \ldots\ldots \vee B_r(t)$.

will apply a reasoning by cases analysis to one or more of these disjunctions $B_2(t) \vee B_3(t) \ldots\ldots \vee B_r(t)$. However, the original disjunction $B_1 \vee B_2 \ldots\ldots \vee B_r$ still would be retained for a possible future application of rule R9. The reason for this retention is that subsequent applications of rules R9 and R10, by generating a goal-subgoal hierarchy such as was illustrated in Figure 1, could provide additional formulas which might form the basis for new solutions to case 1 of this disjunction.

So far we have assumed that we did not need an additional reasoning by cases analysis in order to solve case 1. However, if another application of rule R9 and R10 is needed, the program must decide whether the application should be treated as a subgoal of case 1 or whether the case analysis of $B_1 \vee B_2 \ldots\ldots \vee B_r$ should be abandoned altogether. The heuristic that is used to determine whether case 1 is worth pursuing is the presence or absence of existential variables in B1. Thus, case 1 would be abandoned if and only if an existential variable appears in B1. This heuristic also simplifies the programming since it means that we do not have to compare existential variables that originated from different disjunctions. In any event, the abandonment of case 1 is not necessarily permanent. Case 1 is just postponed in favor of an attempt to solve the cases of some other disjunction or negative conjunction. This attempt could provide the material with which to achieve solutions to case 1 of $B_1 \vee B_2 \ldots\ldots \vee B_r$ when and if this disjunction is reactivated at a later date.[9]

---

[9] If there were no more disjunctions or negative conjunctions, then not only would the abandonment of case 1 become permanent but the program would terminate its attempt to solve the original problem as well.

In general, the state of the system is described by a list $(L_0, K_1, L_1, K_2, L_2, \ldots\ldots, K_n, L_n, L_{n+1})$ where K_i contains the information controlling the reasoning by cases analysis of some specific disjunction or negative conjunction and is defined in a manner similar to K_1. Thus, $K_{i,1}$ represents the list of cases which have not yet been solved, $K_{i,2}$ is the symbol $\vee$ or $\wedge$ depending upon whether an application of rule R9 or R10 is being made, $K_{i,3}$ is the list of cases which have been solved already, and $K_{i,4}$ is the list of existential variables appearing in formulas on list $K_{i,1}$. The first formula on list $K_{i,1}$ represents the case currently under attack and $K_{i+1}, K_{i+2}, \ldots\ldots K_n$ were generated in an attempt to solve this case. For $1 \leq i \leq n$, formulas appearing in L_i already have been processed by the deduction rules and are under the immediate control of K_i. Formulas appearing in L_0 were processed prior to the application of any reasoning by cases analysis. Formulas appearing in L_{n+1} have not yet been processed by the deduction rules.

A new case gets initiated at what is then the lowest level of the goal-subgoal hiearchy. If this lowest level is n, then the empty list L_{n+1} would be created. The first formula on list $K_{n,1}$ would be placed on L_{n+1} if $K_{n,2} = \vee$; if $K_{n,2} = \wedge$, then the negation of this formula would be placed on L_{n+1}. An attempt first is made to solve this case using the deduction rules (except rules R9 and R10) and the output of these rules is placed on L_{n+1}. If a formula F, which appears on L_{n+1}, is applied to a two input deduction rule, then the second input would come from one of the lists $L_0, L_1, \ldots\ldots, L_n$. If F is neither

a disjunction nor a negative conjunction, then it eventually would be either discarded or transferred from L_{n+1} to L_n.

Since the application of $B_1 \vee B_2 \ldots\ldots \vee B_r$ to rule R9 means that the program must find r solutions instead of just one solution, it is a matter of great concern whether $B_1 \vee B_2 \ldots\ldots \vee B_r$ is really needed for the proof; for if it is not needed, then its application to rule R9 could be a great waste of effort. Furthermore, if for each of these r cases, the program should choose unnecessarily a formula $C_1 \vee C_2 \ldots\ldots \vee C_t$ for application to rule R9, then it would have to find rt solutions when only one was really needed. Clearly, the computational effort could snowball if the program is not careful about how it applies formulas to rules R9 and R10. This is not just a theoretical possibility. In the course of searching for a proof, it is not unusual to generate many irrelevant disjunctions and negative conjunctions. Which of the many disjunctions and negative conjunctions is the program to choose for applications to rules R9 and R10?

One way of attacking this problem is to let the decision to select a particular disjunction or negative conjunction represent the node of a goal tree. Although the use of goal trees is a common approach in artificial intelligence research, it will not work here because a good method for evaluating these nodes is not readily available.

Let us summarize the situation which has just been described. Reasoning by cases offers the opportunity to decompose a problem by considering each case separately and automatically erasing all formulas

that were generated during the attempt to solve a case before proceeding to any of the subsequent cases. On the other hand, reasoning by cases could lead to a disastrous explosion of subproblems under consideration by the theorem prover. How then do we prevent such an explosion from taking place?

The escape from our dilemma turns out to be surprisingly simple and one which is likely used by human theorem provers as well. The basic idea is to determine whether the solution to a case depends upon the hypothesis of the case; for if no such dependence is found, then none of the subsequent cases need be considered.

Recall that the system is described by a list $(L_0, K_1, L_1, K_2, L_2, \ldots\ldots, K_n, L_n, L_{n+1})$ where K_i controls the case analysis at the i^{th} level of the goal-subgoal hierarchy. We say that formula F depends upon K_i if either F is the hypothesis of one of the cases of K_i or else the derivation of F utilized this hypothesis at least once as an input to one of the deduction rules. We then define D(F) = dependence of formula F = the set of those K_i upon which F depends. We next define $D_{i,t}$ = dependence of the <u>solution</u> to case t of K_i = the union of all D(F) taken over all formulas F that appeared in the solution to case t of K_i. In particular, suppose the solution to case t of K_i itself utilized a reasoning by cases argument. Then this additional case analysis must have been solved under the control of some K_{i+1}. Letting n_{i+1} represent the number of cases that were solved in the case analysis of K_{i+1} and letting $D_{i+1,0}$ represent the dependence of the particular disjunction or negative conjunction which generated K_{i+1}, we would

obtain $D_{i,t}$ as the union of all $D_{i+1,j}$ taken over all integers j for which $0 \leq j \leq n_{i+1}$

Upon obtaining a solution to case t of K_i, the program asks "Is K_i a member of $D_{i,t}$?" If the answer is yes, the program goes directly to the next case of K_i. However, if the answer is no, then it skips over all the remaining cases of K_i and instead immediately concludes that it has solved the currently active case of K_{i-1}.

Furthermore, before a case of K_i actually is attempted, the program first examines the lists $K_{j,3}$ for all $j \leq i$ in order to determine whether the case had been solved already. If the answer is yes, the program assigns the same dependence to the solution of this case as prevailed for its previously solved duplicate and then skips over this case by proceeding directly to the next case of K_i.

The program also would not attempt to split a formula F into cases if one of the cases of F represented a specialization of a literal already appearing on some L_i (i.e., there is no point in examining any of the cases of F unless each of these cases is supplying us with some new information). For this reason, a further restriction is placed upon the specification of an existential variable. A specification of an existential variable is allowed by K_n only if it does not transform a subsequent case of K_n into a specialization of a literal already appearing on an L_i for some $i < n$.

We have said nothing as yet about the order by which disjunctions and negative conjunctions get activated by rules R9 and R10. Associated with every goal G is the list S(G) of those disjunctions and negative conjunctions that were available to G at the time of its activation. The attempt at solving G without utilizing rules R9 and R10 results in a new list T(G) consisting of the elements of S(G) followed by those disjunctions and negative conjunctions that were created during this attempt. If it is decided to sprout subgoals from G, then the first available formula F on list T(G) becomes the instrument for the split. Suppose G' is one of the cases obtained from the split of F. If no two cases of F have the same variable in common, then S(G') would be T(G) with formula F removed as there then would be no need to split F more than once. However, if the same variable appears in more than one case of F, then we may wish to seek additional splits of F at a later date and so S(G') would be T(G) with formula F transferred from the first to the last element in the list.[10]

---

[10] If the attempt to solve G' without R9 and R10 fails to find a single new solution and an existential variable appears in G', then F is abandoned and a special mark is placed on F to inform the program that F no longer is available for a split. However, if the attempt to solve G' without R9 and R10 fails to find a solution but no existential variable appears in G', then G' would become a parent goal whereupon this special mark would be removed from all formulas of T(G). The rationale for removing these marks (and thereby providing new opportunities for splits) is that the use of G' as a parent goal would provide new information that had not been available when these marks originally were inserted.

There is still one loose end that needs to be tied. We have said that in attempting to solve a case we first seek a solution using all the deduction rules except R9 and R10 but impose a maximum limit on the time spent looking for such a solution. The reason for this maximum time limit is that we do not wish to commit too much of our computational resources in this attempt when additional use of rules R9 or R10 may be necessary. Therefore, if the time limit expires before a solution can be found and the program does not wish to abandon the goal, it would remove from the system those formulas $F_1, F_2, \ldots\ldots, F_t$ which have not yet been processed by the deduction rules and combine then into a single compound formula $\sim(a = a) \vee (F_1 \wedge F_2 \ldots\ldots \wedge F_t)$. This compound formula then is placed at the bottom of the list of disjunctions and negative conjunctions that are associated with the current goal where it would be applied to rule R9 only after those formulas which precede it on the list. This allows the program to continue work on a goal by utilizing a reasoning by cases argument without waiting to process all the formulas $F_1, F_2, \ldots\ldots, F_t$. If it should turn out that one of the formulas $F_1, F_2, \ldots\ldots, F_t$ is necessary for the solution to the goal, then eventually the disjunction $\sim(a = a) \vee (F_1 \wedge F_2 \ldots\ldots \wedge F_t)$ would get split by rule R9. In that event, case 1 consisting of the hypothesis $\sim(a = a)$ would be solved trivially and $F_1 \wedge F_2 \ldots\ldots \wedge F_t$ (the hypothesis of case 2) would be decomposed by rule R3 thereby reestablishing the formulas $F_1, F_2, \ldots\ldots, F_t$.

5. Some Facilities for Representing Procedural Information

When presented with a new axiom, a human often will have some ideas about how the axiom is to be used. A program that can absorb these ideas has an advantage when it comes to solving actual problems. The present program has three main channels through which such information can be received and utilized.

First, as mentioned in Section 3, the procedure by which an axiom $A \supset B$ is used in a problem is different from the procedure associated with the logically equivalent axiom $\sim A \vee B$; the first formulation is used to help solve a case, whereas the second formulation is used to split a problem into separate cases.

Second, a routine, associated with an input F to a two input deduction rule, can decide whether a particular formula should be paired with F as the second input to the rule. Third, descriptive information, associated with one of the inputs, can be passed along to the output of a deduction rule; this descriptive information might be a factor later in deciding whether this output formula should be accepted as a second input to a particular two input rule.

For example, before the formulas $A \supset B$ and A' are applied to rule R6, the program checks to see whether a special attribute appears with $A \supset B$. If this attribute is not present, then the program would continue its attempt to apply $A \supset B$ and A' to rule R6. However, if this attribute does appear with $A \supset B$, it would have some IPL-V routine R associated with it; the program then would execute the routine R using the formula

A' as input data to the routine. Under these circumstances, the decision to continue (with this application of $A \supset B$ and A' to rule R6) would be made on the basis of the result obtained from this execution of routine R. Thus, a routine R associated with the formula $x \varepsilon G \supset x^{-1} \varepsilon G$ might reject $(a^{-1})^{-1} \varepsilon G$ as the second input to rule R6 in order to avoid an endless application of rule R6 to $x \varepsilon G \supset x^{-1} \varepsilon G$. Another way to handle this example stems from the fact that any attribute appearing with formula B gets transmitted to the output formula B' upon the successful application of $A \supset B$ and A' to rule R6. Thus, the routine R might reject A' if it determined that the creation of A' occurred as the output from a previously successful application of rule R6 to $x \varepsilon G \supset x^{-1} \varepsilon G$; routine R could make such a determination merely by checking to see whether a special attribute that appears with $x^{-1} \varepsilon G$ appears also with A'.

There is one attribute, known as the "expansion" attribute, which is processed by the program in a special way. Thus, if the expansion attribute appears with a literal, then the program does not allow the literal to be used as an input to any of the deduction rules except rule R11 and then only in the role of A(r'). After each previously generated formula of the type r = t has had a chance to be paired with this expansion literal for a possible application to R11, the expansion literal is removed from the system. For example, the placement of the expansion attribute with $x*z > y*z$ in the formula $x > y \supset x*z > y*z$ would cause this latter formula to generate expansions. Thus, formulas $b > c$ and $x > y \supset x*z > y*z$ applied to rule R6 would generate the

expansion b*c > c*z. Similarly, the application of ∿(b = c) with the cancellation law x*z > y*z ⊃ x = y to rule R7 would generate ∿(b*z = c*z) as an expansion.[11]

6. The Problem Solving Executive

Any problem solving program must have some overall scheme for allocating its computational resources. This "global" allocation already has been described for the present program in Section 4 in connection with the implementation of rules R9 and R10. However, the present program also must conduct "local" searches which are characterized by an attempt to find a contradiction without further use of rules R9 or R10. These local searches must likewise have a procedure that controls and guides the computational effort; we describe now this local allocation.

A non-literal is given priority ahead of a literal when it comes to deciding the next formula to be removed from L_{n+1} and applied to the deduction rules. Among non-literals, the order is first come first served. Among literals, priority is determined on the basis of a lexicographic ordering which chooses the literal which (1) depends upon the fewest number of the K_i (i.e., so that a solution obtained from this literal would have a better chance of not necessitating the solution of too many additional cases), and in the event of a tie

---

[11] However, we would not generate the expansion if either b or c were of the form u*v where u and v were existential variables since we wish to avoid the effort of trying to create a split by solving for one existential variable in terms of the other.

attempts to choose a literal which (2) does not possess the expansion attribute, and in case of still another tie chooses the literal which (3) has the least complexity where complexity is measured by the storage space taken up by the literal.

However, in trying to find a direct solution to a case involving existential variables, we do not process any expansions if a solution already has been obtained. The reason for this is that the processing of expansions is too expensive to justify their use when looking for additional splits since we can do so at a later date if the current split should prove insufficient.

7. The Equality Relation

It should be noted that unlike the treatment of equality by resolution based theorem provers [13], rule R11 does not permit the replacement of a term in a formula unless that formula is a literal nor does it allow this replacement to be made on the basis of an equality r = t unless the truth of r = t already has been established. The reason we can do this is that the reasoning by cases mechanism serves to detach the individual literals from a formula as it analyzes the separate cases so that eventually these literals will be available for use by rule R11. However, the advantage in postponing these replacements is that it keeps apart those formulas used in solving one case from formulas used in solving a later case with the result that formulas stemming from different cases do not interact with each other to produce additional formulas. Also, since the program does not always have to consider

all cases arising from a split (as was discussed in Section 4), this postponement enables the program to avoid the necessity of generating formulas from a later case if the current case should prove to be irrelevant to the solution of a higher level goal.

In rule R11 we impose the requirement that neither r nor r' can be variables; for if either r or r' were a variable, then the match of r with r' always would be trivially satisfied.[12] The significance of this restriction is that it limits the application of rule R11 to situations where its successful application would provide us with some "information" in the sense of [16] (i.e., the success of a rule gives us no "information" if its success is a foregone conclusion). The practical usefulness of this restriction is that it greatly reduces the number of formulas generated by rule R11 with little risk that one of the discarded formulas will be necessary for the solution.

In applying the equality b = c as an input to rule R11, the program will identify b with r and c with t (i.e., it will replace b by c rather than replace c by b) on the basis of the following eight conditions, where condition i takes priority over condition j for i < j:

(1) c appears as part of term b,

(2) more variables appear in b than in c,

(3) c is a constant which appears in a special list provided to the program by the user (so far, this list has consisted only of identity

---

[12] This assumes of course that r does not appear as a subelement of r' and vice versa.

symbols such as 0 and 1),

(4) a special attribute appears with the equality b = c which tells the program that the right side of the equality should be substituted for the left side (so far, this attribute has been used just once and that was to denote that b had significance only in its capacity as the definition of c),

(5) b, but not c, represents an associative product (i.e., b is of the form r*t where * obeys the associative law; see Section 8 for discussion of associativity),

(6) neither b nor c represent an associative product, but b is a function of more arguments than c,

(7) both b and c represent associative products such as $a_1*a_2.....*a_n$ where n, the number of terms in the product, is greater for b than it is for c, and

(8) b is of greater "complexity" than c where, as in Section 6, the complexity of an expression is measured by the storage space it occupies.

If neither b nor c can be identified with r on the basis of the above eight conditions, then an arbitrary choice is made. If a decision is made to identify b with r but an attribute appearing with formula b = c indicates a desire that both sides of the equality be given this opportunity, then c also would be matched with r and if successful the output A(t') would be designated as an expansion.

If in rule R11 (1) one of the above eight conditions does prevail for r = t, (2) A(r') is not an expansion, and (3) the matching of r to r' does not reduce the generality of r', then the generation of A(t') by rule R11 allows us with reasonable confidence to eliminate A(r') from further consideration provided that the new formula A(t') does not depend upon any of the K_i not already depended upon by A(r'). The program takes advantage of this by employing rule R11 as the first deduction rule to be applied to a literal; the literal is given the role of A(r') in rule R11 and different equalities r = t are paired with it in the hope that one of these equalities will lead to a quick elimination of the literal. Indeed, if the application of one of these equalities to rule R11 generated the literal L' from the literal L without eliminating L but a subsequent application of a different equality to rule R11 did eliminate L, then the program would eliminate the literal L' as well.

We have already restricted the application of rule R11 by not allowing the specification of an existential variable (i.e., we required in Section 4 that an existential variable could be specified only if the specification occurred during an application of rule R1). We now place a further restriction on rule R11 by not allowing any variable from A(r') to be specified. (i.e., the match of r to r' in rule R11 is not allowed if it reduces the generality of r') unless A(r') is either an equality b = c, or its negation ∿(b = c), or an expansion. In order to compensate for these restrictions on the treatment of literals which

do not involve the equality predicate, we included R12 as a rule of inference as this rule takes two such literals as input and produces the negation of an equality as output. However, since one of the motivations for rule R12 was its compensation for the restriction of substitutions into variables, we will require that at least one of the two input literals to rule R12 must possess a variable.

8. Associativity and Commutativity

Functions which are either associative and commutative or just associative play a fundamental role in mathematical reasoning. For example, addition and multiplication in ordinary arithmetic each satisfy both the associative and commutative laws. On the other hand, the multiplication of operators (such as found in matrix multiplication) provide important examples of functions which satisfy the associative but not the commutative law. In view of the great importance of associativity and commutativity, special routines were built into the program in order to provide a more accurate simulation of human problem solving as well as to exploit better the power which is available whenever it is known that a particular function satisfies either both the associative and commutative laws or just the associative law.

An associative function f is one which depends on two arguments and satisfies the relationship f(x, f(y,z)) = f(f(x,y),z) for all x, y, and z. It follows from this that the expressions f(a,f(b,f(c,d))), f(f(a,b),f(c,d)), and f(f(f(a,b),c),d) are all equivalent if f is associative. Using the more familiar product symbol * in place of f,

the above expression can be written as a*(b*(c*d)), (a*b)*(c*d), and ((a*b)*c)*d respectively. Clearly, the key feature of an associative product is that it is independent of the way the parentheses are grouped. A human who uses an associative product acknowledges this fact by writing the above expressions as a*b*c*d; at one stroke he therefore saves processing time as well as memory by avoiding the necessity of treating these equivalent expressions as distinct entities. Although an associative product is defined formally as a function of two arguments, it is used by humans informally as if it were a function of an <u>indeterminate</u> number of arguments $s_1, s_2, \ldots, s_m$ where m can be <u>any</u> integer greater than 1. The same point of view is adopted by the program which, for an associative function f, strips the parentheses from different expressions involving f by reducing them to the canonical form $f(s_1, s_2, \ldots, s_m)$. Thus, the program immediately would reduce $f(f(s_4, f(s_5, s_3)), f(f(s_1, s_2), s_6))$ to $f(s_4, s_5, s_3, s_1, s_2, s_6)$ if it knew f to be associative. Throughout the remainder of this section it will be assumed that the function f is associative.

Standard match routines, such as described in Section 2, can reduce and eventually eliminate the differences between two expressions A and A' only if A and A' have the same structure (i.e., only if in those places where the two expressions A and A' differ, a variable appears in one expression which can be equated to the term appearing in the corresponding part of the other expression). However, for dealing with associativity (and especially for commutativity) a more generalized method of matching is useful (such as the pattern matching of [8])

which can rearrange the position of terms within a structure as well as determine values for variables located at fixed positions.

The present program utilizes a routine MATCHA which can bring into correspondence two associative functions $f(s_1,\ldots,s_m)$ and $f(t_1,\ldots,t_n)$ even though m and n may not be equal to each other. The execution of MATCHA($f(s_1,\ldots,s_m)$, $f(t_1,\ldots,t_n)$) operates as follows. For purposes of exposition, we extend the definition of f to allow it to depend on only a single argument by defining $f(s_1) = s_1$. With no loss of generality, we assume $m \leq n$. If $m = 1$, an attempt is made to bring s_1 into correspondence with $f(t_1,\ldots,t_n)$, perhaps by a substitution of certain terms for variables, after which an exit is made from routine MATCHA. Suppose $m > 1$. We first attempt to find a substitution which will make s_1 identical to t_1 and if successful, we then execute MATCHA($f(s_2,\ldots,s_m)$, $f(t_2,\ldots,t_n)$) for this substitution. After MATCHA($f(s_2,\ldots,s_m)$, $f(t_2,\ldots,t_n)$) has been executed, we undo any substitution of a term for a variable that might have been needed to make s_1 identical to t_1. At this point, if $m = n$ or s_1 is not a variable, we exit from the routine MATCHA. Otherwise, beginning with $r = 2$, we set the variable s_1 equal to the term $f(t_1,\ldots t_r)$ (provided of course that s_1 does not appear in $f(t_1,\ldots t_r)$) and then execute MATCHA($f(s_2,\ldots s_m)$, $f(t_{r+1},\ldots t_n)$) for this substitution; after this has been tried for all integers r such that $r \geq 2$ and $m - 2 \leq n - (r+1)$, an exit is made from the routine MATCHA. For example, if x and y are variables, then the execution of

MATCHA(f(x,y),f(a,b,c)) would produce two successful matches corresponding to x = a, y = f(b,c) and x = f(a,b), y = c. Similarly, the execution of MATCHA(f(x,a,x,a,b),f(b,c,a,b,c,y)) would produce only one successful match (i.e., for x = f(b,c), y = f(a,b)).

The replacement mechanism in rule R11 is designed to take advantage of knowledge that a function f is associative. Thus, suppose r and r' in rule R11 are terms of the form $f(r_1,\ldots,r_m)$ and $f(r'_1,\ldots,r'_n)$ respectively where $m \leq n$. For <u>each</u> integer j such that $0 \leq j \leq n - m$, the program would attempt to find a substitution which would make r_i identical to r'_{j+i} for all $i = 1,2,\ldots,m$. If the program is successful for some j, then the output of rule R11 <u>for this j</u> would be of the form A(t') if m = n, $A(f(t',r'_{m+1},\ldots r'_n))$ if $0 = j < n - m$, $A(f(r'_1,\ldots,r'_j,t',r'_{j+m+1},\ldots r'_n))$ if $0 < j < n - m$, and $A(f(r'_1,\ldots r'_j,t'))$ if $0 < j = n - m$. For example, f(x,x) = e would cause A(f(a,b,b,c)) to be replaced by A(f(a,e,c)) where m = 2, n = 4 and j = 1.

An additional replacement routine is available for use when m < n and some r_i is a universal variable. Thus, for each integer j such that $0 \leq j < n - m$, the program would execute a routine $MATCHB(f(r_1,\ldots r_m),f(r'_{j+1},\ldots r'_n))$. The routine MATCHB is the same as routine MATCHA except that (1) it does not allow a substitution which reduces the generality of $f(r'_{j+1},\ldots r'_n)$ and (2) it proceeds differently when it reaches the point where, for some $p \geq j + m - 1$, it must execute $MATCHB(f(r_m),f(r'_{p+1},\ldots r'_n))$. Instead of executing $MATCHB(f(r_m),f(r'_{p+1},\ldots r'_n))$ by trying to bring r_m into corres-

pondence with $f(r'_{p+1},\ldots,r'_n)$ (as MATCHA would have done) it tries to bring r_m into correspondence with $f(r'_{p+1},\ldots,r'_q)$ for some integer q such that $p < q \leq n$. If the program is successful for some j and q, then the output of rule R11 would be $A(f(t'))$ if $0 = j < q = n$, $A(f(t',r'_{q+1},\ldots,r'_n))$ if $0 = j < q < n$, $A(f(r'_1,\ldots,r'_j, t', r'_{q+1},\ldots, r'_n))$ if $0 < j < q < n$, and $A(f(r'_1,\ldots r'_j,t'))$ if $0 < j < q = n$. For example, $f(x,x) = e$ would cause $A(f(a,b,c,b,c,a))$ to be replaced by $A(f(a,e,a))$, where $m = 2$, $n = 6$, $j = 1$, $p = 3$ and $q = 5$.

For the remainder of this section it will be assumed that the function f also satisfies the commutative law (i.e., $f(x,y) = f(y,x)$ for all x and y). The program uses the routine MATCHC to bring into correspondence two functions $f(s_1,\ldots,s_m)$ and $f(t_1,\ldots,t_n)$ when it is known that f is commutative as well as associative. The execution of $MATCHC(f(s_1,\ldots,s_m), f(t_1,\ldots t_n))$ operates as follows. With no loss of generality, we assume $m \leq n$. If $m = 1$, an attempt is made to bring s_1 into correspondence with $f(t_1,\ldots t_n)$ after which an exit is made from routine MATCHC. Suppose $m > 1$. We first attempt to find an s_i (giving priority to those s_i which are <u>not</u> variables) for which a substitution can be found that makes s_i identical to t_j for some t_j. If we are unsuccessful, we exit from the routine MATCHC. However, if we are successful for some s_i and t_j, we execute MATCHC $(f(s_1,\ldots,s_{i-1}, s_{i+1},\ldots s_m), f(t_1,\ldots t_{j-1}, t_{j+1},\ldots t_n))$ for this substitution and then exit from the routine MATCHC.

The execution of rule R11, when it is known that f is commutative as well as associative, is governed by the routine REPLACE. Thus,

suppose r and r' in rule R11 are terms of the form $f(r_1,\ldots,r_m)$ and $f(r'_1,\ldots,r'_n)$ respectively where $m \leq n$. The execution of REPLACE $(f(r_1,\ldots,r_m), f(r'_1,\ldots,r'_n))$ operates as follows. Beginning with $j = 1$, the program attempts to find a substitution which would make r_1 identical to r'_j. If the program is successful and $m > 1$, it would execute REPLACE$(f(r_2,\ldots,r_m), f(r'_1,\ldots,r'_{j-1}, r'_{j+1},\ldots r'_n))$ for this substitution. If the program is successful and $m = 1$, then the output of rule R11 for this j would be of the form $A(f(t', r'_1,\ldots, r'_{j-1}, r'_{j+1},\ldots,r'_n))$. The program carries out this procedure for each integer j such that $1 \leq j \leq n$ (unless a j is found which produces an output for rule R11 that allows A(r') to be eliminated). For example, f(x,x) = e would cause A(f(b,a,b,c)) to be replaced by A(f(e,a,c)).

9. Computational Experience

The theorem proving program described in this paper was written in IPL-V and run on IBM 360/50 and 370/145 computers. The maximum partition available to the program was 250,000 bytes of core storage. This amounted to 22,000 IPL-V words (after loading the IPL-V interpreter) of which 7,000 were consumed in loading the program leaving 15,000 IPL-V words for actual work space. Although the available memory was not large by current standards, computing time was more of a limiting factor than memory, for IPL-V is an interpretive language and therefore very slow in execution. The program might well have run an order of magnitude faster if it had been written in a language that was capable of execution in a compiled form. The computing times of the examples

reported in this section are for the 370/145.

Among the theorems proved by the program were all nine problems from group theory and number theory which were reported in [4]. It is also capable of solving much more difficult problems than these as evidenced by the examples to be described in this section. It accomplished this without the use of any bounds on substitutions.

The following interpretations are used with the examples of this section:

(1) $x \in y$ means "x is a member of set y",

(2) $x \subset y$ means "x is a subset of y",

(3) $x \equiv y$ means "set x is identical to set y",

(4) $x \cup y$ means " the union of sets x and y",

(5) $x \cap y$ means "the intersection of sets x and y",

(6) $x - y$ means "the set obtained by taking set x and removing from it all elements that appear in set y",

(7) $\cup$ means " the universal set consisting of all elements",

(8) SB(x) means "the set of all subsets of x",

(9) $*$ means " the associative product for the group",

(10) e means " the identity element for the group",

(11) s(x) means "x is a subgroup of the group"

(12) p(X,Y) menas "the product set XY consisting of all elements $x*y$ such that $x \in X$ and $y \in Y$",

(13) prime(x) means "x is a prime number",

(14) $x|y$ means "x divides y",

(15) rational(x) means "x is a rational number", and

(16) sqrt(x) means "the square root of x".

Example 1: The set of all subsets of A intersected with the set of all subsets of B is identical to the set of all subsets of A intersection B.

A1. $\Big(\big(f(x,y)\ \varepsilon x \supset f(x,y)\ \varepsilon y\big) \wedge \big(f(x,y)\ \varepsilon y \supset f(x,y)\ \varepsilon x\big)\Big) \supset x \equiv y$

A2. $x \equiv y \supset \Big((z\varepsilon x \supset z\varepsilon y) \wedge (z\varepsilon y \supset z\varepsilon x)\Big)$

A3. $(z\varepsilon x \wedge z\varepsilon y) \supset z\varepsilon x \cap y$

A4. $z\varepsilon x \cap y \supset (z\varepsilon x \wedge z\varepsilon y)$

A5. $(z \subset x \wedge z \subset y) \supset z \subset x \cap y$

A6. $z \subset x \cap y \supset (z \subset x \wedge z \subset y)$

A7. $(z \subset x \wedge z\varepsilon \cup) \supset z\varepsilon\ SB(x)$

A8. $z\varepsilon\ SB(x) \supset (z \subset x \wedge z\varepsilon \cup)$

A9. $\sim\big(SB(A) \cap SB(B) \equiv SB(A \cap B)\big)$

The program proved example 1 in one minute from the above set of initial axioms and generated 36 new formulas in the process. Although this theorem had been proved in [2], that program utilized routines that were especially designed for set theory. However, this has not been an easy theorem for general purpose theorem provers. What is perhaps remarkable about the effort of the present program is that it did not generate a single formula that was not necessary for the proof (i.e., each of the 36 additional formulas generated belonged to the 36 step proof produced by the program).

Example 2: The square root of every prime number is irrational.

A1. $x*(y*z) = (x*y)*z$

A2. $x*y = y*x$

A3. $x = y \supset x*z = y*z$

A4. $x*z = y*z \supset x = y$

A5. $e*x = x$

A6. $y*x = x \supset y = e$

A7. $sqrt(x)*sqrt(x) = x$

A8. $y/y*x$

A9. $y/x \supset x = y*h(x,y)$

A10. $prime(x) \supset \left(\sim(x/y*z) \vee x/y \vee x/z\right)$

A11. $\sim prime(e)$

A12. $rational(x) \supset \left(f(x) = x*g(x) \wedge \left(\sim\left(y/f(x)\right) \vee \sim\left(y/g(x)\right) \vee y = e\right)\right)$

A13. $\sim \left(prime(a) \supset \sim rational\left(sqrt(a)\right)\right)$

The program proved example 2 in 26 minutes and generated 536 new formulas in the process. Although example 2 has been the object of considerable attention in the literature [14], [5], [12], the present program is the first to prove this theorem without the aid of special hints that reflected a previous knowledge of the proof. The program produced a 30 step proof which is reproduced below. Since the basic operations for equality, such as its reflexivity x = x, are implicit in the operation of the program, they are not mentioned directly in the proof. Also, no direct mention is made of A1 and A2 since the effect of these axioms is implicit in the choice of match and replace routines

used by the program as described in Section 8.

Proof of Example 2:

A14. prime(a) Rule R5 applied to A13.

A15. $\sim\sim$ rational(sqrt(a)) Rule R5 applied to A13.

A16. rational(sqrt(a)) Rule R2 applied to A15.

A17. $\sim$(a/y*z) $\vee$ a/y $\vee$ a/z Rule R6 applied to A10 and A14.

A18. $(f(\text{sqrt}(a)) = \text{sqrt}(a)*g(\text{sqrt}(a))) \wedge (\sim(y/f(\text{sqrt}(a))) \vee \sim(y/g(\text{sqrt}(a))) \vee y = e)$ Rule R6 applied to A12 and A16.

A19. $f(\text{sqrt}(a)) = \text{sqrt}(a)*g(\text{sqrt}(a))$ Rule R3 applied to A18.

A20. $\sim(y/f(\text{sqrt}(a))) \vee \sim(y/g(\text{sqrt}(a))) \vee y = e$ Rule R3 applied to A18.

A21. sqrt(x)*sqrt(x)*z = x*z Rule R6 applied to A3 and A7.

A22. $\text{sqrt}(a)*g(\text{sqrt}(a))*z = f(\text{sqrt}(a))*z$ Rule R6 applied to A3 and A19.

A23. $\text{sqrt}(a)*f(\text{sqrt}(a)) = a*g(\text{sqrt}(a))$ Substitution of A19 into A21 using R11.

A24. $a*g(\text{sqrt}(a))*g(\text{sqrt}(a)) = f(\text{sqrt}(a))*f(\text{sqrt}(a))$ Substitution of A23 into A22 using R11.

A25. $\sim$(a/y*z) Case 1 of A17.

A26. $\sim(a*x = y*z)$ where y and z are the same variables that appeared in A25 (i.e., y and z have been given an existential interpretation). Rule R12 applied to A8 and A25. A contradiction is obtained between A24 and A26 for $y = z = f(\text{sqrt}(a))$ and $x = g(\text{sqrt}(a))*g(\text{sqrt}(a))$. This causes A25 and A26 to be replaced by A27.

A27. $a/f(sqrt(a)) \vee a/f(sqrt(a))$ Cases 2 and 3 of A17 for y = z = $f(sqrt(a))$.

A28. $\sim(y/f(sqrt(a)))$ Case 1 of A20.

A29. $\sim(y*x = f(sqrt(a)))$ where y is the same variable that appeared in A28. Rule R12 applied to A8 and A28. A contradiction is obtained between A19 and A29 for y = sqrt(a) and x = $g(sqrt(a))$. This causes A28 and A29 to be replaced by A30.

A30. $\sim(sqrt(a)/g(sqrt(a))) \vee (sqrt(a) = e)$ Case 2 and 3 of A20 for y = sqrt(a).

A31. $a/f(sqrt(a))$ Case 1 of A27.

A32. $f(sqrt(a)) = a*h(f(sqrt(a)),a)$ Rule R6 applied to A9 and A31.

A33. $a*h(f(sqrt(a)),a)*z = f(sqrt(a))*z$ Rule R6 applied to A3 and A32.

A34. $a*h(f(sqrt(a)),a)*sqrt(a) = a*g(sqrt(a))$ Substitution of A23 into A33 using rule R11.

A35. $h(f(sqrt(a)),a)*sqrt(a) = g(sqrt(a))$. Rule R6 applied to A4 and A34.

A36. $\sim(sqrt(a)/g(sqrt(a)))$ Case 1 of A30.

A37. $\sim(sqrt(a)*x = g(sqrt(a)))$ Rule R12 applied to A8 and A36. A contradiction is obtained between A35 and A37. This causes A36 and A37 to be replaced by A38.

A38. $sqrt(a) = e$ Case 2 of A30.

A39. $e*sqrt(a) = a$ Substitution of A38 into A7 using rule R11.

A40. $sqrt(a) = a$ Substitution of A5 into A39 using Rule R11.

A41. e = a Substitution of A38 into A40 using rule R11.

A42. prime(e) Substitution of A41 into A14 using rule R11.

A contradiction is obtained between A11 and A42. This causes A31 through A42 to be replaced by A43.

A43. a/f(sqrt(a)) Case 2 of A27. The program solved this last case merely by noting that it is the same as a case which had been solved previously (i.e., it is the same as A31). The proof of the theorem now is complete since all outstanding cases have been solved.

Example 3: Grau's three axioms are sufficient to define a ternary boolean algebra.

The following five axioms define a ternary boolean algebra.

A1. $f(f(x,y,u), v, f(x,y,w)) = f(x,y,f(u,v,w))$

A2. $f(y,x,x) = x$

A3. $f(x,y,g(y)) = x$

B1. $f(x,x,y) = x$

B2. $f(g(y),y,x) = x$

The object is to show that axioms B1 and B2 both follow from axioms A1 through A3. That this in fact could be done was announced in the mathematical literature [6] but no proof was presented. It was proved subsequently by an interactive theorem proving program [1] which utilized a human participant in the proof finding process. The following 18 step proof is quite different from the one in [1] and did not involve any human participation. The program is presented with axioms

A1 through A3 as well as the denial of B2. It was not necessary to deny B1 since B1 was produced in the course of proving B2. The proof took 110 minutes during which 245 new formulas were created.

A4. $\sim\big(f\big(g(a),a,b\big) = b\big)$ Denial of B2.

A5. $f\big(y,v,f(x,y,w)\big) = f\big(x,y,f(y,v,w)\big)$ Substitution of A2 into left side of A1 using rule R11.

A6. $f(y,v,y) = f\big(x,y,f(y,v,y)\big)$ Substitition of A2 into left side of A5 using rule R11.

A7. $f\big(y,v,f(x,y,v)\big) = f(x,y,v)$ Substitution of A2 into right side of A5 using rule R11.

A8. $f\big(f(v,w,u),\ v,f(v,w,w)\big) = f(u,v,w)$ Substitution of A7 into right side of A1 using rule R11.

A9. $f\big(f(v,w,u),\ v,w\big) = f(u,v,w)$ Substitution of A2 into A8 using rule R11.

A10. $f(x,x,y) = f\big(g(y),x,y\big)$ Substitution of A3 into left side of A9 using rule R11.

A11. $x = f\big(g\big(g(x)\big),x,g(x)\big)$ Substitution of A3 into left side of A10 using rule R11.

A12. $g\big(g(x)\big) = x$ Substitution of A3 into A11 using rule R11.

A13. $f\big(x,g(y),y\big) = x$ Substitution of A12 into A3 using rule R11.

A14. $f\big(y,g(z),\ f(x,y,z)\big) = f(x,y,y)$ Substitution of A13 into right side of A5 using rule R11.

A15. $f\big(y,g(z),\ f(x,y,z)\big) = y$ Substitution of A2 into A14 using rule R11.

A16. $f\big(y,g\big(g(y)\big),x\big) = y$ Substitution of A3 into A15 using rule R11.

A17. $f(x,x,y) = x$ Substitution of A12 into A16 using rule R11. This

proves axiom B1.

A18. $f(x,y,x) = x$ Substitution of A6 into A17 using rule R11.

A19. $f(g(y),x,y) = x$ Substitution of A17 into A10 using rule R11.

A20. $f(y,x,g(y)) = x$ Substitution of A12 into A19 using rule R11.

A21. $f(x,y,x) = f(g(y),y,x)$ Substitution of A20 into left side of A9 using rule R11.

A22. $f(g(y),y,x) = x$ Substitution of A18 into A21. This completes the proof since A22 contradicts A4.

Example 4: In a group, if $x*x*x = e$ and $f(x,y) = x*y*x^{-1}*y^{-1}$ for all x and y, then $f(f(a,b),b) = e$

A1. $x*(y*z) = (x*y)*z$

A2. $x = y \supset x*z = y*z$

A3. $x = y \supset z*x = z*y$

A4. $x*z = y*z \supset x = y$

A5. $z*x = z*y \supset x = y$

A6. $x*e = x$

A7. $e*x = x$

A8. $x*x^{-1} = e$

A9. $x^{-1}*x = e$

A10. $x*x*x = e$

A11. $f(x,y) = x*y*x^{-1}*y^{-1}$

A12 $\sim(f(f(a,b),b) = e)$

Example 4 was discussed in the appendix to [13] in which it was shown that a paramodulation proof of this theorem could be found which

took only 47 steps whereas a comparable proof using ordinary resolution took 136 steps. However, these proofs were not obtained from an actual procedure for finding proofs but were the product of an inspired human effort. By contrast, the first computer produced proof of this theorem was obtained by the present program. This proof was 44 steps long and took 30 minutes during which 415 new formulas were created.

Example 5: Let K be a subgroup of group G and Kog be the right coset of K in G for some $g \in G$. Then Kog is identical to K if and only if g is a member of K.

A1. $x*(y*z) = (x*y)*z$

A2. $x = y \supset x*z = y*z$

A3. $x = y \supset z*x = z*y$

A4. $x*z = y*z \supset x = y$

A5. $z*x = z*y \supset x = y$

A6. $x*e = x$

A7. $e*x = x$

A8. $x*x^{-1} = e$

A9. $x^{-1}*x = e$

A10. $(x^{-1})^{-1} = x$

A11. $[f(x,y) \in x \supset f(x,y) \in y] \supset x \subset y$

A12. $x \subset y \supset (z \in x \supset z \in y)$

A13. $(z \in x \vee z \in y) \supset z \in x \cup y$

A14. $z \in x \cup y \supset (z \in x \vee z \in y)$

A15. $(z \in x \wedge z \in y) \supset z \in x \cap y$

A16. $z \in x \cap y \supset (z \in x \wedge z \in y)$

A17. $\left(z \in x \wedge \sim(z \in y)\right) \supset z \in x - y$

A18. $z \in x - y \supset \left(z \in x \wedge \sim(z \in y)\right)$

A19. $(x \subset y \wedge y \subset x) \supset x \equiv y$

A20. $x \equiv y \supset (x \subset y \wedge y \subset x)$

A21. $\left(\begin{array}{c}\left(\left(g(x) \in x \wedge h(x) \in x\right) \supset g(x) * h(x) \in x\right) \\ \wedge \left(g(x) \in x \supset \left(g(x)\right)^{-1} \in x\right)\end{array}\right) \supset s(x)$

A22. $s(z) \supset \left(e \in z \wedge (x \in z \supset x^{-1} \in z) \wedge (x^{-1} \in z \supset x \in z)\right)$

A23. $s(z) \supset \left(x * y \in z \vee \sim(x \in z) \vee \sim(y \in z)\right)$

A24. $s(K)$

A25. $x \in Koz \supset \left(x = m(x,z) * z \wedge m(x,z) \in K\right)$

A26. $x * z \in Koz \vee \sim(x \in K)$

A27. $\sim\left((Kog \equiv K \supset g \in K) \wedge (g \in K \supset Kog \equiv K)\right)$

The program found a 46 step proof to example 5 in 8 minutes during which 219 new formulas were generated[13].

Example 6: If H and K are subgroups of group G, then the product set HK is a subgroup of G if and only if HK is identical to KH.

A1-A22. Same as in Example 5.

A23. $s(x) \supset \left(s(y) \supset (z \in p(x,y) \supset (z = m(x,y,z) * n(x,y,z) \wedge m(x,y,z) \in x \wedge n(x,y,z) \in y))\right)$

---

[13] Axioms A2 through A10 were not allowed to interact directly with each other since the remaining initial axioms were placed in a set of support [20]. The purpose was only to save a little computer time as these nine axioms were very familiar and their initial effects quite predictable.

A24. $s(x) \supset \left(s(y) \supset \left(u*v \in p(x,y) \vee \sim(u \in x) \vee \sim(v \in y)\right)\right)$

A25. $s(x) \supset \left(s(y) \supset \left(s*w \in p(x,y) \vee \sim(s = u*r) \vee \sim(w = t*v)\right.\right.$

$\left.\left. \vee \sim(u \in x) \vee \sim(v \in y) \vee \sim(r*t \in p(x,y))\right)\right)$

A26. $s(H)$

A27. $s(K)$

A28. $\sim\left(\left(p(H,K) \equiv p(K,H) \supset s(p(H,K))\right) \wedge \left(s(p(H,K)) \supset p(H,K) \equiv p(K,H)\right)\right)$

The program found a 134 step proof to Example 6 in 72 minutes during which 960 new formulas were created.[14]

To this author's knowledge, examples 5 and 6 have never appeared before in the literature on automatic theorem proving. Each of these examples is noteworthy in that the computer was confronted with a very rich set of initial axioms. Both of these examples (and especially Example 6) should prove to be quite a challenge to machine oriented automatic theorem provers.

---

[14] Same as footnote 13.

References

1. Allen, John and Luckham, David, An interactive theorem proving program. In Machine Intelligence 5, Meltzer, B. and Michie, D. (Eds.), Edinburgh University Press, Edinburgh, 1970, pp. 321-336.

2. Bledsoe, W.W., Splitting and reduction heuristics in automatic theorem proving, Artificial Intelligence, Vol. 2, No. 1, (Spring 1971), pp. 55-77.

3. Bledsoe, W.W., Boyer, Robert S., and Henneman, William H., Computer proofs of limit theorems, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1971.

4. Chang, C.L., The unit proof and the input proof in theorem proving, Journal of the Association for Computing Machinery, Vol. 17, No. 4, (October 1970), pp. 698-707.

5. Chang, C.L., Theorem proving with variable-constrained resolution, Information Sciences, Vol. 4, No. 3, (July 1972), pp. 217-231.

6. Chinthayamma, Sets of independent axioms for a ternary boolean algebra, Notices of Amer. Math. Soc., Vol. 16, No. 4, June 1969, 69T-A69, pp. 654.

7. Ernst, George W., The utility of independent subgoals in theorem proving, Information and Control, Vol. 18, No. 3, (April 1971), pp. 237-252.

8. Hewitt, Carl, Planner: A language for proving theorems in robots, Proceedings of the International Joint Conference on Artificial Intelligence, Washington, D.C., 1969, pp. 295-301.

9. Kleene, Stephen C., Introduction to Metamathematics, Van Nostrand, Princeton, New Jersey, 1952.

10. Loveland, Donald W., A simplified format for the model elimination procedure, Journal of the Association for Computing Machinery, Vol. 16, No. 3, (July 1969), pp. 349-363.

11. Nilsson, Nils J., Problem Solving Methods in Artificial Intelligence, McGraw Hill, New York City, 1971.

12. Norton, Lewis, M., Experiments with a heuristic theorem-proving program for predicate calculus with equality, Artificial Intelligence, Vol. 2, (Winter 1971), pp. 261-284.

13. Robinson, George and Wos, Lawrence, Paramodulation and theorem-proving in first order theories with equality. In Machine Intelligence 4, Meltzer, B. and Michie, D. (Eds.), Edinburgh University Press, Edinburgh, 1969, pp. 135-150.

14. Robinson, John A., Theorem-proving on the computer, Journal of the Association for Computing Machinery, Vol. 10, No. 2, (April 1963), pp. 163-174.

15. Robinson, John A., A machine-oriented logic based on the resolution principle. Journal of the Association for Computing Machinery, Vol. 12, No. 1, (January 1965), pp. 23-41.

16. Shannon, Claude E. and Weaver, Warren, The Mathematical Theory of Communication, University of Illinois Press, Urbana, Illinois, 1949.

17. Slagle, James R. and Koniver, Deena A., Finding resolution graphs and using duplicate goals in AND/OR trees. Information Sciences, Vol. 3, No. 4, (October 1971), pp. 315-342.

18. Wang, Hao, Toward mechanical mathematics. IBM Journal of Research and Development, Vol. 4, (January 1960), pp. 2-22.

19. Wos, Lawrence, Carson, Daniel F., and Robinson, George, The unit preference strategy in theorem proving. Proceedings AFIPS 1964 Fall Joint Computer Conference, Vol. 26, pt. 1, pp. 615-621 (Spartan Books, Washington, D.C).

20. Wos, Lawrence, Robinson, George A., and Carson, Daniel F. Efficiency and completeness in the set of support strategy in theorem proving. Journal of the Association for Computing Machinery, Vol. 12, No. 4, (October 1965), pp. 536-541.