3

BETH-TREE METHODS IN AUTOMATIC THEOREM-PROVING

R. J. POPPLESTONE
EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

INTRODUCTION

This paper contains an account of a procedure for proving theorems mechanically. A description is first given of the language in which the theorems to be proved are stated (i.e., the first-order functional calculus). The axioms of group theory are then listed in this language. Next a process is defined for finding proofs of statements made in the language, and finally a specialised procedure for proving equalities is proposed and the results of some experimental tests are described and discussed.

The method of searching for a proof by means of a tree or 'semantic tableau' is due originally to Beth (1960). It is perhaps closer to Wang's (1960) approach than Robinson's (1965).

FUNCTIONAL CALCULUS

In order to prove theorems mechanically, a formal language with known syntactic rules is assumed to be necessary. The first-order language used in this paper has the following components:

(i) symbols of fixed meaning (delimiters): $\&$, $\vee$, $\Rightarrow$, $\neg$, $\exists$, $\forall$, $=$, meaning 'and', 'or', 'implies', 'not', 'there exists', 'for all' and 'equals' respectively;

(ii) symbols whose meaning is defined by the user of the language (these are predicates (which are properties or relations), functions, associative binary operators and variables);

(iii) brackets serving to indicate the scope of delimiters, and of predicates, functions and operators.

For example, let N be a predicate meaning 'is a natural number' and let Gr be a predicate such that $Gr(x, y)$ means $x > y$ then

$$\forall x, y, z\,((N(x) \,\&\, N(y) \,\&\, N(z) \,\&\, Gr(x, y) \,\&\, Gr(y, z)) \Rightarrow Gr(x, z))$$

is a statement saying that 'for all values of x, y and z, if x is a number, y is a number and z is a number, and $x > y$ and $y > z$ then $x > z$'.

Equality could be regarded as a predicate, since it is clearly a relation between objects, but in the proof search method to be described it is treated specially, and so becomes a symbol fixed in meaning.

The associative binary operators are entities such as $+$, $\times$, $\cap$, $\cup$. These are logically equivalent to functions of two variables, but their use makes the explicit definition and application of an axiom of associativity unnecessary. Thus $a+b+c$ is a unique representation of the prefix forms plus (plus (a, b), c) and plus (a, plus (b, c)).

By a *term* we mean a well-formed combination of functions, variables and binary operators. Thus if f is a function then $f(a+b)$ is a term, as is $a+b$ or $f(x)$.

The *atomic statements* of the language consist of predicates (or equals) with terms as arguments. Finally, the statements of the language are atomic statements combined with the logical operators &, $\vee$, $\Rightarrow$, $\neg$, $\forall$, $\exists$ (see (i) above).

Cursive letters $\mathscr{A}\,\mathscr{B}\,\mathscr{C}$ . . . will be used to denote statements. Thus $\mathscr{P}(x_1 \ldots x_n)$ is a statement containing some of the terms $x_1 \ldots x_n$.

I shall not explicitly introduce constants, as they can be regarded as functions of no variables.

Variables themselves will be used both bound by the quantifiers $\forall$ and $\exists$ and free. Thus in the statement

$$\forall\, x\, \exists\, y(x+y=z)$$

x and y are bound and z is free. The system is first order in the sense that quantification over functions or predicates is not allowed. Thus the axiom of induction, which states that *any property* of a number which is true for zero and whose truth for any number implies its truth for its successor, cannot be expressed within the system described here.

GROUP THEORY

The method I shall describe could be used to find proofs for any theory expressible in the first order functional calculus. I shall use Group Theory as an example. Most of the problems attempted on the computer have been from Group Theory. For an elementary account of Group Theory see Birkhoff & Maclane (1953). Group Theory will be considered as being anything that can be said in terms of a predicate G, meaning 'lies in the group', an associative binary operator for which I shall use an asterisk '*', a function e of no variables, being the identity of the group, and a function I of one variable, being the inverse operation, more usually written $^{-1}$. The axioms these obey are:

AG1	$\forall\, x, y(G(x) \,\&\, G(y) \Rightarrow G(x * y))$	closure with respect to *
AG2	$\forall\, x(G(x) \Rightarrow G(I(x)))$	closure with respect to inversion
AG3	$G(e)$	existence of identity
AG4	$\forall\, x(G(x) \Rightarrow x * e = x \,\&\, e * x = x)$	properties of identity
AG5	$\forall\, x(G(x) \Rightarrow x * I(x) = e \,\&\, I(x) * x = e)$	properties of inverse

Of these, AG1 and AG2, besides being closure axioms, enable the construction of the product and inverse of terms in the proof procedure to be described. There is no need for an axiom of associativity, since, as explained in 'functional calculus', this is implicit in the use of *. The closure properties of AG1 and AG2 would be useful if, say, the number axioms were to be adjoined to AG1 to AG5 in order to consider the interaction of groups and numbers. Thus, for example, we would have a function $P(x, n)$ meaning x^n and with closure axiom

$$\forall\, x, n(G(x) \,\&\, N(n) \Rightarrow G(P(x, n)))$$

where the predicate N means 'is a natural number'. Group Theory as understood by the mathematician may involve almost any branch of mathematics.

BETH TREES

Once having the apparatus to express mathematical statements in a symbolic language, the next step is to decide whether any such statement is a logical consequence of others. Beth (1962) developed a procedure, the growing of Beth trees (semantic tableaux), which is guaranteed to find a proof of any valid statement expressed in the first-order calculus. Other such exhaustive search procedures are known, for instance that described by Davis and Putnam (1960). Such procedures tend to involve a large amount of effort in proving even simple theorems.

A Beth tree is grown as follows. Each node of the tree consists of two lists of statements. One list can be thought of intuitively as a list of statements known to be true, and the other of statements one of which we would like to prove true. These lists are called K for 'known' and R for 'required to prove' and at the start contain the axioms and the theorem statement respectively.

Given a node (K, R), the node or nodes directly below it (we grow our trees downwards) are produced as follows. If all statements in K and R are atomic then the next node is the same as the current one. Otherwise a non-atomic statement in K or R is selected to be operated on. This selection is performed either by a rule for selecting statements in rotation or by a heuristic device. Next, the most global logical operator of the chosen statement is found. Thus in the statement

$$(G(x) \,\&\, G(y)) \Rightarrow G(x * y)$$

$\Rightarrow$ is the most global operator. If this operator is one of the propositional operators $\&, \vee, \Rightarrow, \neg$ then Table 1 gives the rule for forming the descendent

node or nodes. In the next section we will consider what happens when a quantified statement is to be dealt with.

A branch of a Beth tree ends in a TIP if one of the statements in K is the same as one of those in R, that is $K \cap R \neq \phi$. If all of the branches of the tree end in tips then the statement placed in R at the root of the tree is a consequence of the axioms placed in K. Conversely, if one (or more) branch has no tip, and so is infinite, then the original statement in R is not a consequence of the axioms. In some cases it may be possible to see that a branch of a Beth tree is infinite, as for example when all statements are atomic after a certain number of nodes have been developed.

As an example, consider the statement

$$(A \Rightarrow B) \,\&\, (B \Rightarrow C) \Rightarrow (A \Rightarrow C) \qquad (1)$$

where A, B and C are predicates of no variables (so that this is a statement in the propositional calculus). Fig. 1 shows a Beth tree proof that (1) is a theorem. (Since (1) is a purely logical result, there are, in this case, no axioms to place in the K-list—the axioms of logic are incorporated in the rules of the Beth tree.)

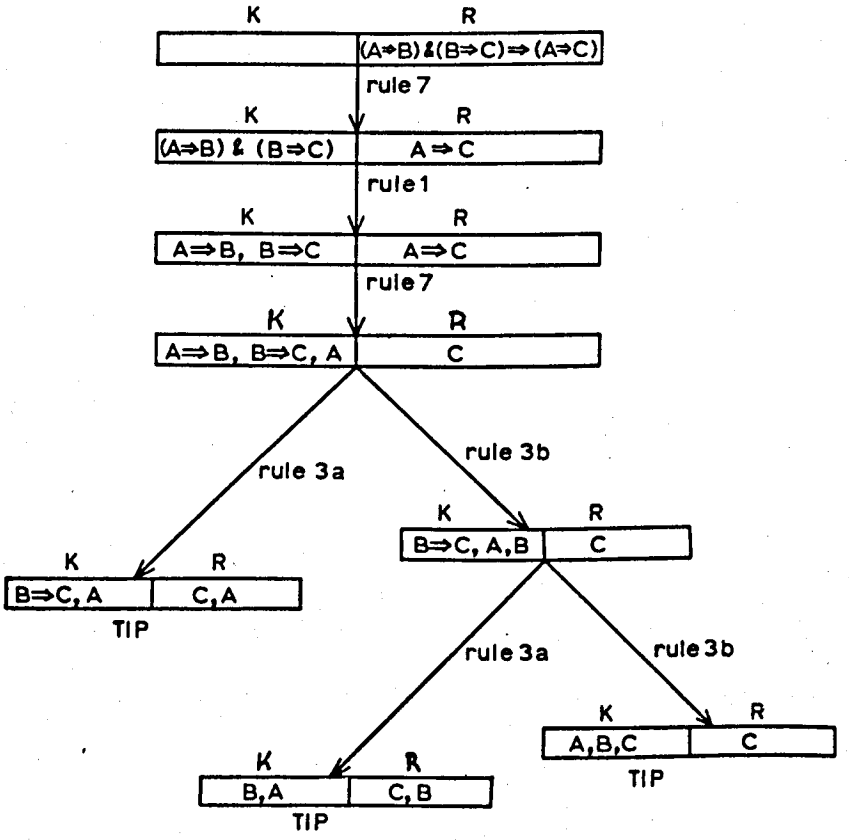
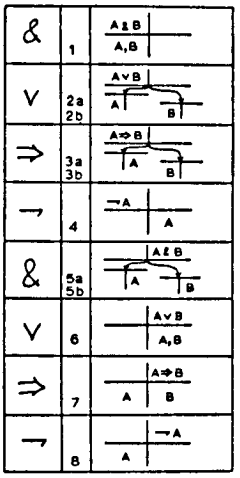


FIG. 1. A Beth-tree proof that $(A \Rightarrow B) \,\&\, (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$. The numbered rules are given in Table 1.

TABLE 1



Beth-tree rules for handling the propositional operators. The upper two quadrants of each 2 × 2 table give the state of the lists (Known and Required to prove respectively) *before* the application of the given rule, and the lower two quadrants the state *after* application of the rule.

QUANTIFIERS IN BETH TREES

The procedures for dealing with quantified expressions in Beth trees will now be discussed. Let K and R be, as before, the 'known' and 'required to prove' lists of a node of a tree.

An existentially quantified statement $\exists x\mathscr{P}(x)$ in K will be replaced by $\mathscr{P}(y)$ where y is a variable not occurring in K or R.

The handling of the case where we have a universally quantified statement $\forall x\mathscr{P}(x)$ in K is crucial to the success of any theorem proving scheme. Beth's method is to add the sentences $\mathscr{P}(t_1) \ldots \mathscr{P}(t_m)$ to K, still leaving $\forall x\mathscr{P}(x)$ in K, where $\{t_1 \ldots t_m\}$ are all terms occurring at that node. The point of leaving $\forall x\mathscr{P}(x)$ in K is that it can again be used on subsequent occasions to generate new statements and hence new terms. This method, while ensuring that a proof will be found, results in a very rapid growth in the size of K and R, and often in prolific branching of the tree.

For instance, if there were a numerical variable n among the terms at a given node of the tree, Beth's rules would have us add $G(n) \Rightarrow G(I(n))$ to K, which is true but irrelevant, and which leads to a split of the tree into two branches, thus doubling the work involved in finding a proof. This difficulty could be tackled by applying the rules only when one or other of the resulting branches will terminate immediately. A more satisfactory alternative is to introduce a new rule. The rule that I have used is that if one has an axiom of the form 'for all x, premise $\Rightarrow$ conclusion', and *if the premise is in K* when some term is substituted for x, then one may add the conclusion to K. Thus if $G(x * y) \in K$, one may apply AG2, which states $\forall x(G(x) \Rightarrow G(I(x))$, and add $G(I(x * y))$ to K.

More formally, and more generally, suppose we have a statement in K of the form:

$$\forall x_1 \ldots, x_n(\mathscr{P}_1(x_1 \ldots, x_n) \,\&\, \ldots \mathscr{P}_p(x_1 \ldots, x_n) \Rightarrow \mathscr{Q}(x_1 \ldots, x_n)) \quad (2)$$

where the $\mathscr{P}_i$ and $\mathscr{Q}$ are statements in $x_1 \ldots, x_n$. Notice that this is the form of the group axioms, apart from AG3. Suppose now that θ is a mapping from the set $\{x_1 \ldots x_n\}$ of variables of (2) bound by the quantifier into the set $\{t_1 \ldots, t_m\}$ of free (i.e., not bound by a quantifier) terms of the current node. Suppose further that for all r from $1 \ldots p$

$$\mathscr{P}_r(\theta(x_1) \ldots \theta(x_n)) \in K \quad (3)$$

That is to say, each of the $\mathscr{P}_r(x_1 \ldots, x_n)$ is a known formula if the term $\theta(x_i)$ is substituted for x_i for all i. Thus in some particular case all the premises of (2) are known, and so we may draw the conclusion. In other words, we add $\mathscr{Q}(\theta(x_1) \ldots \theta(x_n))$ to K. This is done at one node for every θ satisfying (3), and for every statement of the type (2), the additional statements being added to the K-list of the next node down the tree, so that $\{t_1 \ldots t_m\}$ remains the same throughout the operations on one node.

A simple example of the process described above is the following proof that, in a group, $e * e = e$.

K	R
AG1-AG5	$e * e = e$

There are two terms, namely e and $e * e$, in this node. Consider AG1. This states that $\forall\, x, y(G(x) \,\&\, G(y) \Rightarrow G(x * y))$. Thus there are four mappings from the set $\{x, y\}$ of bound variables of AG1 to the set $\{e, e * e\}$ of terms of the node.

θ_1	defined by	$\theta_1(x) = e$	$\theta_1(y) = e$
θ_2	defined by	$\theta_2(x) = e * e$	$\theta_2(y) = e$
θ_3	defined by	$\theta_3(x) = e$	$\theta_3(y) = e * e$
θ_4	defined by	$\theta_4(x) = e * e$	$\theta_4(y) = e * e$

Now $G(\theta_1(x))$ is $G(e)$ which is the axiom AG3 and so is in K. Similarly $G(\theta_1(y))$ is $G(e)$ and is in K. Thus both premises of AG1, when e is substituted for x and y, are known. So we may add the conclusion $G(e * e)$ to K. On the other hand, if we consider θ_2, bearing in mind that we are working with the version of K not yet altered by the addition of $G(e * e)$ allowed by θ_1, we see that $G(x)$ becomes $G(e * e)$ which is not yet in K. So θ_2 does not satisfy (2). So no addition to K results from θ_2, or indeed from θ_3 and θ_4. Operating on AG2, AG4 and AG5 likewise, we obtain with the aid of Table 1.

K	R
AG1-AG5, $G(e * e)$, $G(I(e))$, $e * e = e$	
$e * e = e$, $e * I(e) = e$, $I(e) * e = e$	$e * e = e$
TIP	
QED	

Gelernter's Geometry Machine (Gelernter 1960) uses, in effect, a process dual

to the above one. If on substituting for the bound variables in some axiom the statement '$\mathscr{P}=\mathscr{Q}$' is formed, and $\mathscr{Q}$ *is a member of* R then Gelernter's machine adds $\mathscr{P}$ to R also.

In both Gelernter's machine and my own, heuristic methods of pruning the lists K and R are used.

The chief disadvantage of the Beth-tree method, as compared with Robinson's resolution principle (Robinson 1965), is that general deductions cannot be made. Thus, in a group of exponent 2 ($\forall\, x(G(x) \Rightarrow x * x = e)$) the Beth method can deduce that for some particular x, $x = I(x)$ but not that in general $\forall\, x(G(x) \Rightarrow x = I(x))$. Thus the fact that, say, $x * y = I(x * y)$ would have to be proved the 'hard way' from the original quantified statement $\forall\, x(G(x) \Rightarrow x * x = e)$ rather than by using the general deduction $\forall\, x(G(x) \Rightarrow x = I(x))$. Just how severe a limitation this imposes remains to be discovered.

EQUALITY IN BETH TREES

In the example above, equality was treated as an ordinary predicate. Equality could be treated in this manner throughout, but to do so would require axioms for every predicate and function expressing the fact that equal terms can be substituted for each other. For instance, in Group Theory, axioms such as

$$\forall\, x, y(G(x) \,\&\, x = y \Rightarrow I(x) = I(y))$$
$$\forall\, x, y(G(x) \,\&\, x = y \Rightarrow G(y))$$

would need to be added to AG1 to AG5, as well as axioms expressing the reflexive, symmetric and transitive properties of equality. In view of this difficulty, and of the fundamental importance of equality, it is worth introducing special techniques for dealing with this relation. The essence of the method to be described is that when statements of equality appear in the R-list, a special procedure is invoked to demonstrate this equality, using known equalities. If this is successful, the node under consideration is declared to be a TIP. Thus when it is required to prove, say, that $a * I(a) * b = b$ we use the known facts that $a * I(a) = e$, and $e * b = b$, to construct a chain of equalities $a * I(a) * b = e * b = b$. This can be expressed more precisely.

With each *node* of the Beth tree associate a *graph*. The nodes of the *graph* are all terms which can be constructed from the functions, associative binary operators and variables occurring in that node of the *tree* which is associated with the graph. Two graph nodes are neighbours if one can be converted into the other by substituting for one occurrence in it of any term $\mathscr{U}$ a term $\mathscr{V}$ known to be equal to $\mathscr{U}$ (i.e., '$\mathscr{U}=\mathscr{V}$' $\in K$). Note that we are now using cursive letters for terms. If we know that $x * e = x$, then $x * x$ and $x * e * x$ are neighbours, but $x * x$ and $x * e * x * e$ are not. We say that the Beth tree has an 'equality tip' if $\mathscr{P}=\mathscr{Q}$ occurs in R and a path can be found from $\mathscr{P}$ to $\mathscr{Q}$ in the graph defined above.

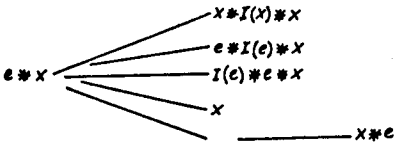
One should notice that a path from $\mathscr{P}$ to $\mathscr{Q}$ does not necessarily exist. A path between two terms may not exist either because the two terms are

not in fact equal, or because insufficient deductions have been made from the axioms to prove the terms equal. In either case it is essential to be able to break off the search after a certain point.

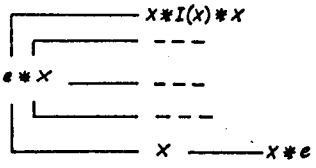
Suppose, for example, that we wish to prove that $e * x = x * e$ in a group. After some application of the group axioms, the following node of the Beth tree has been formed:

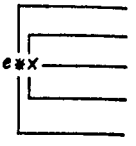
$$\begin{array}{c} K \\ G(e * x),\ x * I(x) = e,\ I(x) * x = e,\ e * x = x,\ x * e = x \\ e * I(e) = e,\ I(e) * e = e,\ e * e = e \\ R \\ e * x = x * e \end{array}$$

Exploring the graph defined above gives the following path from $e * x$ to $x * e$.



So we have an equality-tip and, the tree having no other branches, the theorem is true.





THE USE OF HEURISTICS IN PROVING EQUALITY

In this section I shall describe how the search for a path in the equality graph may be controlled. In the experiments described here, use was made of the Graph Traverser (Doran & Michie 1966, and see Doran's and Michie's papers in this volume). This is a program, written in Algol, for finding paths in graphs too large to be represented explicitly inside a computer. For my purposes it suffers from the defect of not being a list-processing program and so being incompatible with the rest of the Beth-tree program, but it has proved very valuable as a tool for demonstrating the effectiveness of heuristic methods in at least one part of theorem-proving activity. The input to this program consists of a statement of an equality that is to be proved together with a list of equalities, as produced by a Beth tree, which are to be used in the proof.

Recall that the task is to find a path from $\mathscr{P}$ to $\mathscr{Q}$, where $\mathscr{P} = \mathscr{Q}$ is a member of R, in the graph where nodes are terms, and two nodes are neighbours when they can be seen to be equal by making one application of

one known equality. Suppose the situation is as in Fig. 2. A search has been organised which has established a connexion between $\mathscr{U}$ (among other nodes generated during the search) and $\mathscr{P}$. The problem is whether to initiate the next phase of the search from $\mathscr{U}$ or whether to proceed from

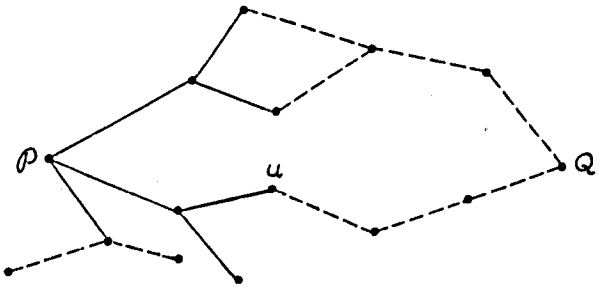


FIG. 2. A part of an equality graph. Unbroken lines denote parts of the graph already explored. Nodes represent terms, and a node can be converted into one of its neighbours by making one application of one known equality.

some other node. To make this choice we have an evaluation function F, and if $F(\mathscr{U})$ happens to be smaller than $F(\mathscr{V})$ for any other node $\mathscr{V}$ under consideration, then we 'develop' $\mathscr{U}$ in the sense of generating descendant nodes from it. A perfect evaluation function would satisfy

$$F(\mathscr{U}) \leqslant F(\mathscr{V}) \Leftrightarrow d(\mathscr{U}, \mathscr{Q}) \leqslant d(\mathscr{V}, \mathscr{Q}) \tag{4}$$

where $d(\mathscr{A}, \mathscr{B})$ is the distance between $\mathscr{A}$ and $\mathscr{B}$ in the graph.

Evaluation functions are usually combinations of 'features'. A feature f is a function which measures some property of a node. In my experiments, expressing the evaluation function as a linear combination

$$F = a_1 f_1 + a_2 f_2 + \ldots + a_n f_n \tag{5}$$

of features $f_1 \ldots f_n$, has proved adequate. An evaluation function of this type is analogous to Samuel's 'scoring polynomial' (Samuel 1959).

In the first experiments using the Graph Traverser three features were used. The first of these was L, where $L(\mathscr{U})$ is the length of $\mathscr{U}$, that is to say, the number of symbols contained therein. In fact, when calculating L, the program ignores *, but adds one, corresponding to an end-of-string marker to the length of the string as written. Thus $x * x * e * I(y)$ has a length of 8.

The second feature was E, the number of occurrences of the identity e in $\mathscr{U}$, apart from the first. This was incorporated in order to penalise nodes such as $e * e * x * e$, which are very readily formed, and which are of little use in proving any theorems other than pathological ones.

The third feature, T, was designed to measure the degree of similarity between a node $\mathscr{U}$ and the target node $\mathscr{Q}$. This was defined as the length of the longest string in $\mathscr{U}$ that was also in $\mathscr{Q}$. Thus if $\mathscr{U}$ is $I(a * b)$ and $\mathscr{Q}$ is $I(b) * I(a)$ then $\mathscr{U}$ and $\mathscr{Q}$ have the string $I(a$ in common, and, this being the longest such string, $T(\mathscr{U}) = 3$.

Three problems in Group Theory were attempted using an evaluation function:

$$F=L+5E-2T$$

The results were not impressive. The machine was allowed to generate 100 distinct nodes before giving up, and only one of the problems was solved in this time. Two measures to improve the performance of the program were adopted.

The first, and simpler, was to penalise the development of heavily bracketed structures. When asked to prove that $I(b) * I(a)=I(a * b)$, for which a solution is

$$\begin{aligned} I(b) * I(a) &= e * I(b) * I(a)=I(a * b) * a * b * I(b) * I(a) \\ &= I(a * b) * a * e * I(a)=I(a * b) * a * I(a) \\ &= I(a * b) * e=I(a * b) \end{aligned}$$

the unimproved program investigated initial possibilities such as

$$I(b) * I(a)=I(e * b) * I(a)=I(a * b * I(a * b) * b) * I(a)$$

The feature adopted to measure the degree of nesting of bracketed expressions was B, defined by

$$B(\mathscr{U})=\sum_{\substack{\alpha \in \upsilon \\ \alpha \neq (,),*}} K_\alpha$$

where K_α is the number of bracket-pairs enclosing α, and $\in$ means, rather loosely, 'is a symbol occurring in'. The usefulness of B and the weight to be attached to it remains to be determined since as yet no problem has been attempted where development inside brackets is to be preferred to development outside.

The second measure was to remedy a deficiency in the feature T. T, as defined originally, regards any matching substring between a node and the target node as being a significant likeness.

T, defined thus, can be very misleading, for instance, when required to prove that $I(a) * I(b) * b * a=a * b * I(b) * I(a)$ for which a solution might run $I(a) * I(b) * b * a=I(a) * e * a=I(a) * a=e=a * I(a)=a * e * I(a)=a * b * I(b) * I(a)$.

Instead of finding the solution, the program investigated nodes such as $I(a) * a * b * I(b)$ which has a string of 6 symbols in common with the target, but which has the term $I(a)$ at the wrong end. In the light of this it seemed sensible to modify T only to take account of those matching substrings for which the non-matching parts are (*i*) well-formed terms and (*ii*) equal to each other (extending either node if necessary by $* e$ or $e *$ to avoid having to interpret the null string; we would, for example, regard the common substring $a * b * I(b)$ of the node $I(a) * a * b * I(b)$ and the target node $a * b * I(b) * I(a)$ as being significant if $I(a)=e$). Thus we would evaluate the match between the strings $\mathscr{P}_1 2 \mathscr{R}_1$ and $\mathscr{P}_2 2 \mathscr{R}_2$ (where $\mathscr{P}$, 2 and $\mathscr{R}$ represent well-formed substrings) as significant if $\mathscr{P}_1=\mathscr{P}_2$ and $\mathscr{R}_1=\mathscr{R}_2$.

How therefore are we to decide whether these or any other two terms are

equal? It is known, since the 'word problem' in Group Theory is undecidable, that there is no certain way. We could use the Graph Traverser recursively to try and prove them equal, but this would be uncertain and time-consuming. A method that is less time-consuming (and probably less uncertain) is to calculate the values in some particular group and see if they are equal. Let us call such a group an example-group, and consider a very simple group of four elements defined by its multiplication table.

TABLE 2

*	e	a	b	c
e	e	a	b	c
a	a	e	c	b
b	b	c	e	a
c	c	b	a	e

x	$I(x)$
e	e
a	a
b	b
c	c

Looking up $I(a)$ in this table, we see that $I(a)=a\neq e$ and so $I(a) * a * b * I(b)$ has no significant likeness to $a * b * I(b) * I(a)$. This device is substantially the same as Gelernter's diagram (Gelernter 1960).

Provided that the example-group satisfies whatever additional assumptions may be in force about the group in the theorem, there is no possibility of equal terms being classified as unequal when calculated in the example. However, unequal terms might have equal values in the example-group. Thus in the group in Table 2 $a * b=b * a$, although this is not true in general.

In passing I might add that a whole battery of example-groups with different properties (commutative, exponent 2, nilpotent, etc.) could be useful in pinpointing those axioms which should be used in a proof. Thus if commutativity is given as an axiom and a matching of terms is significant in a commutation example-group, but not in a non-commutative example, then the Beth-tree routines might be told to apply the commutativity axiom (especially to the non-matching terms).

With the addition of the feature B, and with the modification of T discussed above, the three problems, together with a fourth, were re-run. This time all the problems were solved with reasonable competence. A measure of efficiency for Graph Traversing is the penetrance, defined as:

$$\frac{\text{number of developed nodes on path}}{\text{total number of nodes developed}}$$

The values of the penetrance were 100 per cent, 100 per cent, 85 per cent and 75 per cent.

Let us consider one of these results more closely (problem 2 of Table 3, condition PD omitted). The evaluation function used was

$$F=L+B+5E-3T$$

The actual printout from the run involved 77 terms and so is too lengthy to be reproduced here, but an annotated version appears below, with the missing terms replaced by comments in small print. The label $m.n$ denotes the nth node produced of the mth generation.

	Term	L	B	E	T	F
0.1	$I(b) * I(a)$	9	2	0	0	11
First descendants						
1.1	$I(b * e) * I(a)$	10	3	0	0	13
And other terms with e inside the brackets						
1.3	$I(b) * e * I(a)$	10	2	0	0	12
1.4	$e * I(b) * I(a)$	10	2	0	0	12
1.7	$I(b) * I(a) * e$	10	2	0	0	12

The next generation is produced from the first-generated of the three nodes with a score of 12, that is $I(b) * e * I(a)$. All the descendants of this node, apart from the starting node $I(b) * I(a)$, are longer than it, and so only those with a non-zero T can hope to attain a score low enough to be considered as parents of the next generation. There are however no common substrings which satisfy the criterion of significance described above.

Consequently, we have to go back to the previous generation (generation 1) for a new parent, and node 1.4 ($e * I(b) * I(a)$) is next on the list. All of its descendants save one have a zero score in T. The exception is

$$I(a * b) * a * b * I(b) * I(a)$$

for which the obvious match with $I(a * b)$ is significant. So the score for this node is

2.6	$I(a * b) * a * b * I(b) * I(a)$	16	4	0	5	5

So the evaluation function F has the value 5 for this node, and hence the node is again developed. All is now plain sailing, with a steady reduction of length at each node.

3.18	$I(a * b) * a * e * I(a)$	12	3	0	5	0
	etc.					
4.11	$I(a * b) * a * I(a)$	11	3	0	5	−1
	etc.					
5.13	$I(a * b) * e$	7	2	0	5	−6
	etc.					
6.9	$I(a * b)$	6	2	0	5	−7

While the performance of the program in these examples, as measured by the penetrance, is reasonable, the number of nodes produced during a run is unacceptably large. Thus in the last two problems of the four mentioned above, both of which had a path length of 6, the numbers of distinct nodes produced were 77 and 95 respectively. In order to try to reduce this number, a further change has been made in the program. Instead of producing all nodes which are neighbours of a node under consideration, only a few are produced. This 'partial development' is achieved by arranging the list of known equalities in an order which places first those, such as $I(a) * a = e$, which give rise to a reduction in length in a node. ($e = I(a) * a$ is entered separately from $I(a) * a = e$ in the list of known equalities.) The program then makes use of a small number of equalities, starting with the most promising, during one development of one node.

This last modification of the program has made some improvement in the performance. Over the four problems for which comparable figures are available, the total number of distinct nodes produced was reduced by this

device from 223 to 161, and the number of nodes produced (counting repetitions) was reduced from 315 to 189.

A number of problems were attempted using this version of the program. Table 3 gives the results.

TABLE 3

Results obtained with six problems under various conditions.

Condition Omitted	Result	Nodes Listed	Nodes Developed	Inspections	Penetrance %	Path Length
PROBLEM 1. $abI(b)I(a)=e$						
—	S	7	3	6	100·0	3
L	S	11	4	12	75·0	3
B	S	7	3	6	100·0	3
E	S	7	3	6	100·0	3
T	S	7	3	6	100·0	3
PD	S	30	3	39	(100·0)	3
M	S	20	7	22	42·9	3
F	S	25	11	30	63·6	3
PROBLEM 2. $I(b)I(a)=I(ab)$						
—	S	55	19	65	63·2	6
L	S	74	22	92	54·5	6
B	S	99	31	127	38·7	6
E	S	55	19	65	63·2	6
T	F	100	32	130	—	—
PD	S	88	8	122	(75·0)	6
M	F	100	40	128	—	—
F	S	61	25	79	56·0	6
PROBLEM 3. $I(a)I(b)ba=abI(b)I(a)$						
—	S	32	16	38	75·0	6
L	FL	—	—	—	—	—
B	S	32	16	38	75·0	6
E	S	32	16	38	75·0	6
T	F	100	49	134	—	—
PD	FL	—	—	—	—	—
M	F	100	27	125	—	—
F	S	29	14	34	71·4	6
PROBLEM 4. $abI(b)I(a)ba=ba$						
—	S	9	4	8	100·0	4
L	S	17	6	19	66·7	4
B	S	9	4	8	100·0	4
E	S	9	4	8	100·0	4
T	S	9	4	8	100·0	4
PD	S	46	4	63	(100·0)	4
M	S	9	4	8	100·0	4
F	S	36	17	46	58·8	4

In all the following problems the additional condition $\forall x, x^2 = e$ was assumed.

Condition Omitted	Result	Nodes Listed	Nodes Developed	Inspections	Penetrance %	Path Length
PROBLEM 5. $I(a)=a$						
—	S	31	13	33	76·9	4
L	S	33	15	35	66·7	4
B	S	57	19	65	52·6	4
E	S	31	13	33	76·9	4
T	S	51	21	58	42·9	4
PD	S	42	5	46	(80·0)	4
M	S	52	22	59	40·9	4
F	S	30	14	32	64·3	4
PROBLEM 6. $ab=ba$						
—	S	36	17	44	64·7	6
L	FF	100	45	151	—	—
B	S	36	17	44	64·7	6
E	S	36	17	44	64·7	6
T	F	100	56	196	—	—
PD	S	65	7	95	(85·7)	6
M	FF	100	32	135	—	—
F	S	37	18	43	61·1	6

SUMMARY

Condition Omitted	Total Successes	Medians			Mean Penetrance %
		Nodes Listed	Nodes Developed	Inspections	
—	6	31½	14	35½	79·9
L	5	53½	18½	63½	57·2
B	6	34	16½	41	71·8
E	6	31½	14½	35½	79·9
T	3	>75½	26½	94	49·8
PD	5	55½	6	79	(88·1)
M	3	>76	24½	92	47·4
F	6	33	15½	38½	62·5

In the 'condition omitted' column:

— denotes that the full complement of heuristic devices was in use
L denotes that the coefficient of the feature L was set to zero
B denotes that the coefficient of the feature B was set to zero
E denotes that the coefficient of the feature E was set to zero
T denotes that the coefficient of the feature T was set to zero
PD denotes that all neighbours of a node were produced at one development
M denotes that the example group was not in use

F denotes that the equality was proved in reverse, i.e., that the RHS was converted into the LHS. Note that the LHS is longer than the RHS in every problem where the lengths are different.

In the 'result' column:

S denotes that the problem was solved

F denotes that the problem was unsolved after 100 distinct nodes had been produced

FF denotes that the problem was unsolved after 100 distinct nodes had been produced but that the problem was solved after restarting one or more times from the most promising node produced so far (see Doran's paper in this volume).

FL denotes that the problem was unsolved because a term too large to hold in the array assigned to hold it was produced.

The 'nodes listed' column gives the total number of distinct nodes produced.

The 'inspections' column gives the total number of nodes produced, counting repetitions.

Penetrance values in the rows marked PD have been bracketed, since their interpretation is somewhat different under the PD condition.

The results given in Table 3 indicate that all the heuristic devices apart from the feature E play a useful part in the solution of problems. Note the great effect produced by conditions T and M acting in combination. T is ineffective without M, and M is, of course, meaningless without T.

Two further problems were attempted, namely $I(a)I(b)=I(b)I(a)$ and $I(a)I(b)ab=e$, and neither was solved. Analysis of the results indicated that the use of a more comprehensive example group or groups might have resulted in a solution of the problems.

CONCLUSION

The Beth tree procedure described in the early part of this paper has been coded and tested on a small computer (a Stantec Computing System), and proved capable of performing manipulations of the kind described between pages 33 and 38. Unfortunately, the machine was too small (*c.* 700 words of list space) to permit an extensive evaluation of the Beth-tree procedure, and it has not yet proved possible to run the program on a larger machine. The results on the exploration of the equality graph, as described in 'The use of heuristics in proving equality' (p. 38), were obtained on an Elliott 803. At present, the only heuristic device attached to the Beth tree procedure is to delete those statements in the *K*-list which do not refer to nodes developed in the preceding search of the equality-graph (assuming that an equality is to be proved), thus coupling the tree-search to the equality heuristics. I feel that, with relatively small improvements in the control of tree growth, a performance in the proving of equalities in algebraic systems which is

mathematically useful is attainable. More difficult problems, such as existence proofs, seem to be further over the horizon.

REFERENCES

Beth, E. W. (1962). *Formal methods.* Dordrecht-Holland: Reidel.

Birkhoff, G., & MacLane, S. (1953) *A survey of modern algebra* (revised ed.). New York: Macmillan.

Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *J. Ass. comput. Mach.*, **7**, 201-215.

Doran, J., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc.* (A)., **294**, 235-259.

Gelernter, H. (1959). Realisation of a geometry-theorem proving machine. *Int. Conf. Inf. Process.*, pp. 273-282. Paris: UNESCO.

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

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Jl Res. Dev.*, **3**, 211-229.

Wang, H. (1960). Towards mechanical mathematics. *IBM Jl Res. Dev.*, **4**, 2-22.

4

THE RESOLUTION PRINCIPLE IN THEOREM-PROVING

DAVID LUCKHAM
DEPARTMENT OF MATHEMATICS
UNIVERSITY OF MANCHESTER

INTRODUCTION

The evidence is already in favour of the computer becoming an aid to research in branches of mathematics which do not involve numerical computation. Programs have now been constructed for performing certain symbolic operations in mathematics, for example algebraic simplification of equations and indefinite integration, and for solving some of the problems (which are certainly not in the class of numerical calculations) now occurring in theoretical physics and other areas. One may reasonably expect that sooner or later programs of this type will be incorporated in a 'question-answering' package. As the construction of multi-programming systems progresses, and 'on-line' use of the computer is accepted as normal, it will become more and more practicable for the mathematician to rely on the computer for answers to routine non-numerical questions. It is, therefore, of interest from a practical as well as a theoretical point of view to ask if this development can be taken a stage further by mechanising the processes of mathematical proof. In this way the computer could be used as a tool for establishing true mathematical statements or checking proofs for correctness. The problem is essentially to produce practicable programs for proving theorems which will answer some reasonably complicated questions before overstepping the bounds of available time and memory space. This is potentially one of the most rewarding areas of computer applications, but as many people already know, it is also one of the most frustrating.

Although some research into the possibility of constructing heuristic programs based on the psychological knowledge of human problem-solving, and the role that mathematical intuition plays therein, has been reported, for example, in Newell, Shaw & Simon (1957), at present there can be little