

February 1972

THE ROLE OF FORMAL THEOREM PROVING IN
ARTIFICIAL INTELLIGENCE

by

Bertram Raphael

First of two lectures to be presented at JITA (Japanese
Industrial Technology Association) International Symposium
on Information Processing Systems, Tokyo, March 6-17, 1972.

Artificial Intelligence Center

Technical Note 63

SRI Projects 8776 and 1530

This lecture is based upon research supported at SRI by
the National Science Foundation under Grant GJ-1060, and
by the Advanced Research Projects Agency of the Department
of Defense, monitored by U. S. Army Research Office-Durham
under Contract DAHC04 72 C 0008.

ABSTRACT

This tutorial paper begins by dividing Artificial Intelligence into three major subfields: formal problem solving, machines that understand, and machines that interact with the physical world. Each subfield contains a need for a component of formal logic. After summarizing the basic concepts of Mathematical Logic, the paper defines both the Propositional Calculus and the First-Order Predicate Calculus and describes some proof procedures for each. Finally, techniques are outlined for tailoring the Resolution proof method to particular kinds of problems of Artificial Intelligence.

This lecture will discuss how to use a digital computer for proving mathematical theorems. We are interested in this problem for several reasons: First, because automating the process of proving mathematical theorems is an interesting and challenging intellectual problem. Second, it could have important consequences for mathematics because there are many unproved theorems and interesting mathematical questions that might be solved by computer systems if we can discover how to couple the large memories and high speed of computers with the strategies, intuition, and guidance of human mathematicians. Finally, and perhaps most important from the viewpoint of this symposium, an automatic mathematical theorem-proving system can be an important component of a variety of larger automatic systems in the general field of Artificial Intelligence. Before going into theorem-proving methods per se we shall first present a sketch of the major problem areas in Artificial Intelligence and how logical inference systems can be fit into them.

A. Overview of Artificial Intelligence

Artificial Intelligence is the label that has been attached to a research area of computer science concerned principally with how to increase the problem-solving and perceptual capabilities of computers; that is, how to give computers more of the intellectual abilities generally associated with human intelligence. We frequently hear the cliché that computers are nothing but large, fast, adding machines and that they can do nothing that they have not been specifically told how to do. Although this may be true in a certain very limited sense, it would be more productive to take a contrary but equally valid viewpoint; namely, that computers are large, fast, symbol-manipulation machines and that they are capable of solving any sufficiently definable problem (because, after all, they are universal calculating machines in the Turing sense). The

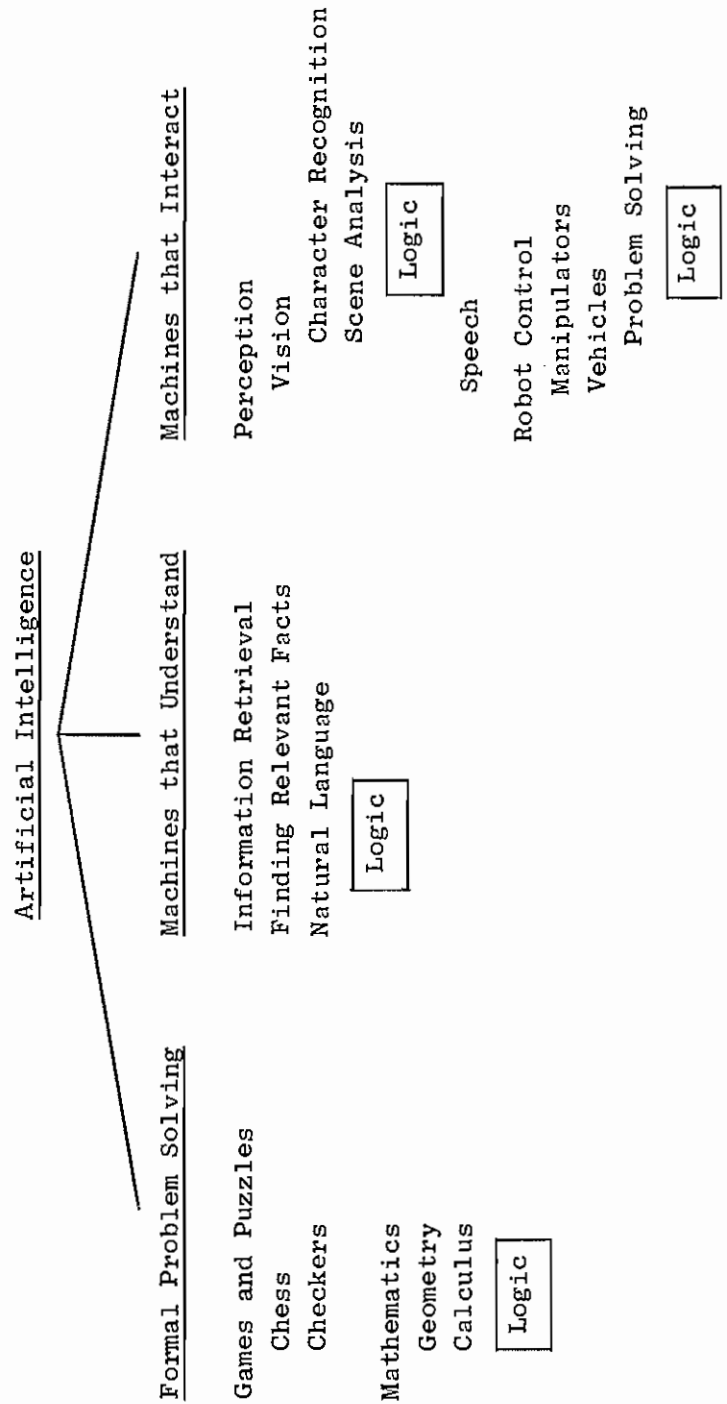
error in the myth that computers are only designed to do arithmetic can easily be realized by looking at the instruction repertoire of any large computer or the actual set of machine-language instructions used in any large program. A very small percentage of these specific directions, or wired-in commands to the machine, actually have anything to do with the arithmetic values of numbers; that is, a very small percentage of these instructions are of the form add, subtract, multiply, etc. The rest, by far the preponderance of instructions used on any computer, are instructions for moving or comparing pieces of data. These pieces of data happen to be binary bits in most instances. However, with appropriate intermediate levels of software the user of the computer can imagine the machine to be any type of calculating machine he wishes.

Most scientific computing views computers as number calculators and therefore programming languages such as FORTRAN and ALGOL give the user the impression that he is working with a large, fast, adding machine. Business-oriented languages, for example COBOL, augment the arithmetic capabilities seen by the user with a variety of input-output and file-manipulation capabilities. For the types of problems of interest to Artificial Intelligence it is useful to view the computer as a symbol- or list-processing machine. Therefore, most of the computer programs in this field have been written in LISP or some other list-processing language. In fact, the design of the first list-processing languages was motivated by applications of this kind.

The major problem domains that have been tackled under the general heading of Artificial Intelligence are sketched in Figure 1. First we have the area of formal problem solving. This is one of the earliest set of problems tackled in our field. The motivation for choosing such problems is fairly clear. If one wishes to demonstrate the ability of a digital

FIGURE 1

Artificial Intelligence and Logic



computer to solve interesting, intellectual problems then first, one should pick problems that are clearly defined (that is, whose rules can be implemented in computer programs in some fairly direct way); and secondly, problems whose solutions would be clearly recognizable (that is, it should not require any obscure argument to convince the human critic that the computer has indeed solved the problem). Therefore, the first domains chosen as examples of Artificial Intelligence research consisted of formal problem-solving situations, for example games, puzzles, and certain areas of mathematics.

Now, one interesting feature of work in Artificial Intelligence is that we are frequently surprised at how poor our intuitions may be concerning the difficulties of new tasks. Some tasks, such as programming a computer to play an expert or master game of chess, appear natural at first for computer implementation. After all, the machine need merely examine a large search tree and it would undoubtedly be able to find the best move. In fact it has turned out that the problem of chess has been extremely difficult for a computer because the size of the search tree examinable by a computer in a reasonable amount of time is still insignificant compared to the entire tree of the game, and therefore we must build into computer chess programs a considerable amount of knowledge of the game, in fact an amount comparable apparently to that used by human chess players. On the other hand, certain pattern-recognition problems, for example the problem of recognizing faces, had been viewed by some as virtually insoluble by computers since the criteria of recognizing faces seem so ill defined that it could be claimed to be an inherently human capability. On the other hand, recent systems such as those discussed by Dr. Harmon show that the computer has considerable promise for solving even this ill-defined problem.

Thus, progress in solving formal problems by computer seems to lie in utilizing a mixture of techniques that take advantage of the unique computer capabilities for high-speed calculation and large memory capacity, and imitation or simulation of the heuristic approaches used by humans for solving similar problems. Let us look at progress in various areas of doing mathematics by computer. In the late 1950s a group at IBM headed by Dr. Gelernter^{1*} programmed a computer to do plane geometry at the high school level. The principal interesting feature of this program was that it achieved most of its success by making use of a technique that has been taught to high school students for centuries but that no one had considered trying to build into a machine before; namely, the program in a sense sketched for itself a diagram in general position that satisfied the conditions of its current problem and then used the diagram as a test for hypotheses that the program generated in the course of its attempted proof. In 1962 Dr. Slagle at MIT completed a computer program for solving calculus integration problems.² This program was able to score close to 90% on an MIT freshman calculus final examination. Once again its success in symbolic mathematics was attributable principally to its use of the same kinds of heuristic principles that human students are taught in solving similar problems.

Formal mathematical logic has been a domain for experimentation in heuristic programming for many years. The implementation of a truth-table decision procedure for propositional logic is a common exercise in beginning programming courses. (The nature of this and other procedures will be discussed further below.) In the early 1960s Newell, Shaw, and Simon used the domain of propositional logic as a basis for experimenting with computer simulations of human problem-solving techniques.³ Shortly thereafter,

* References are listed at the end of this paper.

Professor Wang at Harvard presented a computer-oriented algorithm for efficiently proving theorems in propositional logic.⁴

The more interesting domain of predicate calculus has been much more difficult for automated proof. We shall discuss some approaches below. Here let me simply summarize the current status. There now exist machine proof procedures (based on Robinson's resolution principle⁵) that can prove theorems in mathematics at the graduate level and that have already been useful when guided by human mathematicians in solving open questions in mathematics.⁶ Since such work is of great interest and potential value to mathematics, these methods are currently actively being studied and improved.

Let us now look at another area of Artificial Intelligence--namely, the development of machines that "understand," or as McCarthy put it in a pioneering paper,⁷ "machines with common sense." How can we organize computer systems that would have broad knowledge of the world and be able to solve simple everyday problems such as, for example, planning routes to travel from one point to another, or deciding on appropriate clothing to wear for particular occasions? We might call this class of problems "informal problem solving." Formal problem solving is characterized by problems that require deep solutions based upon a small amount of initial information; for example, finding the right move in a chess game given the rules of the game and the current position, or proving a theorem in group theory given the axioms of group theory and a set of logical deduction rules. Informal problem solving, on the other hand, is characterized by problems that require very shallow reasoning to produce conclusions from a large amount of initial data; for example, it is easy to deduce that I should wear my raincoat from the premises that it is now raining and I am about to go outside. However, these are merely two of almost uncountable

many facts that are in my mind at any one time and if a computer is to draw similar conclusions it will have to have some kind of mechanism for extracting the relevant premises from its broad store of knowledge.

Conventional information-retrieval systems are one limiting example of informal problem solvers. These systems have little or no deductive ability; they merely retrieve from memory specific facts on the basis of specific indexed clues. If we wish a machine to truly understand some subject domain we have to give the machine not only the ability to retrieve specific facts stored in memory but also the ability to deduce from those facts whatever consequences may be needed to respond to a particular question. Perhaps we will be able to combine a general, logical, theorem-proving system with a conventional information-retrieval system to produce a more powerful understanding (or "question-answering") machine. We shall discuss below one approach to using a theorem prover.

Another important aspect of the informal understanding machine is the language with which it communicates with the human users. Logical inference systems naturally require a formal logical terminology. Informal communication, on the other hand, normally takes place in natural language such as ordinary English. Therefore, an obvious step would be to attempt to translate ordinary natural language into a formal language that could be used with a deductive system. One major research issue deals with the nature of the formal language required to represent all the concepts normally expressible in natural language. First-order predicate calculus is a step in this direction but is clearly not adequate for many of the concepts that we would like to handle. I expect that Professor McCarthy's lectures will contain proposals for more adequate formal languages. However, first-order predicate calculus is the most powerful logical system for which we now have reasonably adequate proof procedures. We will discuss

below how it could be used in our first attempt at constructing intelligent question-answering systems.

A third major set of problem domains for Artificial Intelligence concerns machines that interact with the physical world. These systems can be divided into two parts: those that perceive the world and those that affect it. Most of the work on perception has concentrated upon the visual sense and, until recently, has emphasized very specific well-defined task domains. Character recognition of both printed and hand-drawn characters is the principal example. In the acoustic domain considerable work has been done on machine recognition of individual words.

More recently research interest in perception has shifted to larger contexts. Visual-perception research is now concentrating on "scene analysis"; that is, the recognition of objects in pictures of three-dimensional scenes. (I'll have more to say about these projects in a later lecture.) Similarly, work has recently been started on speech understanding systems in which the recognition of sounds will make use of knowledge of syntax and semantics of language, as well as sound patterns of individual words, in analyzing complete utterances. It is likely that logical inference methods will be useful in integrating the diverse components of these new perceptual systems; however, this idea has not yet been implemented in any existing systems.

Finally, we would like to build systems in which a computer affects the world; for example, by operating a manipulator or driving a vehicle. Such systems will undoubtedly require a combination of problem-solving, logical-inference, and perceptual capabilities in order to enable the controlling computer to decide what actions it should take with the various switches and motors under its control. (The design of one such complete robot system will be the subject of a later lecture.)

One goal of many Artificial Intelligence researchers has been, in Professor Newell's words, "a search for generality."⁸ We would like to find ways of transferring techniques from one problem domain to another, or even better, of building single programs or systems that will be applicable to a variety of problem situations. We observe here that logical inference in first-order predicate calculus is a candidate for such a general system. It is a useful and perhaps necessary component of certain formal problem-solving systems, informal problem-solving systems, and probably perceptual and robot systems. At Stanford Research Institute we are already using one such subroutine package, the QA3 theorem prover,⁹ for studies in mathematical logic, question answering, and robot problem solving. Before explaining exactly how this can be done, I must explain some of the principles of logical theorem proving. The next sections will discuss the terminology and structure of theorem-proving systems. Then we shall be able to return to the issue of how such systems can be embedded in larger Artificial Intelligence systems.

B. Basic Concepts of Mathematical Logic

There are generally two viewpoints associated with every logical system: the syntactic and the semantic. Syntax deals with manipulation of strings of symbols according to certain formal rules. The symbols are not considered to have any external significance. This approach is frequently most convenient for forming rigorous proofs. The semantic viewpoint is mainly concerned with interpretation of the symbols and significance of the results. Although the design of a logical system may be motivated by its semantics, the operation of the system generally consists of basically syntactic manipulations. Occasionally a broadly used logical syntax, for example that of the predicate calculus to be discussed below, may have a wide variety of possible semantic interpretations. In this case we may

arbitrarily choose a "standard interpretation," usually based upon numbers and sets. However, this standard interpretation is usually isomorphic to most of the possible interpretations of the syntactic formulas.

Let us first look at the syntactic approach. Every logical system is concerned with some well-defined class of symbol strings (frequently defined recursively, for example by a context-free grammar) called the well-formed formulas (wffs) of the system. A subset of the wffs is then identified and designated to be the theorems of the system. The theorems are frequently defined to be those wffs that can be generated by a set of string-manipulation rewriting rules called the rules of inference or rules of deduction for the logic. Theoremhood may be determined with respect to a set of wffs given as a priori theorems or axioms (the Hilbert approach), or may be determined in an absolute sense, i.e. with no axioms (the Gentzen approach). These two approaches are essentially equivalent. For example, C is a theorem in the Hilbert sense with axioms A and B, if and only if a single wff corresponding to the assertion "the conjunction of A and B implies C" is a theorem in the Gentzen sense.

The semantics of a logical system refers to the association of certain objects and interpretations with the syntactic symbols of the logic. The semantics consist of two parts: a model and a valuation.

The model (sometimes called the assignment) consists of both the identification of certain objects or mathematical entities, e.g. sets and relations, as the semantic domain, and the associations between those objects and particular symbols used in the wffs of the logic. The valuation is an algorithm for mapping every wff and an associated model into either truth or falsity (T or F) but not both.

The valuation procedure is usually defined once and for all as part of the logical system. The model, on the other hand, may be selected at will.

A model is said to satisfy a wff if the valuation is T for that wff and model.

A wff is said to be satisfiable if there exists any model that satisfies it.

A wff is said to be valid if every possible model satisfies it.

The usefulness of a logical system frequently depends upon compatibility between the semantics and the syntax of the system. In particular, we generally intend the theorems to coincide with the set of valid formulas. This coincidence is not always achievable. A logical system is said to be sound (and its rules of inference are consistent) if every theorem is valid. A logical system (or equivalently, its set of rules of inference) is said to be complete if every valid wff is a theorem.

Observe that if a system is not sound then theorems can be proven that are occasionally false, and therefore the system would not be a useful basis for analyzing the semantic domain. Therefore we generally do not use systems that are unsound. In an incomplete system, on the other hand, we can only prove wffs that are true (although not all of them), and therefore the system may still be useful.

A decision procedure for a logical system is an algorithm that is guaranteed to determine in a finite number of steps whether or not any given wff is a theorem. A system is said to be decidable if there exists a decision procedure for it; otherwise it is undecidable.

A proof procedure for a logical system is an algorithm that can prove in a finite number of steps that certain wffs are theorems. However, for nontheorems (and perhaps for certain theorems), the proof procedure may never terminate.

NOTE: It is possible for a system to be both sound and complete and to possess a proof procedure but to be undecidable. It need merely be possible to construct a nontheorem that the inference rules cannot determine to be a nontheorem in a bounded number of steps. Thus an observer who watches the proof procedure operate never knows whether the procedure is working on a theorem and will terminate in the next few steps or whether it has a nontheorem and will keep running forever.

Logicians frequently concern themselves with searching for decision procedures for various classes of wffs. In Artificial Intelligence, on the other hand, we are concerned with finding efficient proof procedures for proving particular theorems whether or not they belong to decidable classes.

C. The Propositional Calculus

Propositional logic is perhaps the simplest useful system of logic. Table 1 summarizes its symbols. There are two kinds of symbols used: propositional variables and connectives. The wffs are defined recursively as follows:

- (1) Every propositional variable is a wff (and is called an "atomic formula").
- (2) If A and B are wffs then so are $A \wedge B$, $A \vee B$, $A \supset B$, $A \Leftrightarrow B$, and $\sim A$.

The semantics of the propositional calculus may be defined as follows:

A model is any set, \mathcal{S} , of propositional variables.

A valuation, τ , is defined as follows:

If A is a propositional variable, then

$$\tau(A) = T \text{ if } A \text{ is in } \mathcal{S}$$

$$\tau(A) = F \text{ if } A \text{ is not in } \mathcal{S}.$$

TABLE 1

Symbols of Propositional Calculus

p, q, r, \dots		propositional variables
\wedge	and	} propositional connectives
\vee	or	
\supset	implies	
\Leftrightarrow	equivalence	
\sim	not	

TABLE 2

Truth Table for Propositional Connectives

A	B	$A \wedge B$	$A \vee B$	$A \supset B$	$A \Leftrightarrow B$	$\sim A$
T	T	T	T	T	T	F
T	F	F	T	F	F	F
F	T	F	T	T	F	T
F	F	F	F	T	T	T

Otherwise, $\tau(A)$ is established recursively from the component of the wff A as given by the truth table for the connectives (see Table 2).

Before proceeding to discuss proof procedures for the propositional calculus let us consider two points that are troublesome to many people when they are first introduced to logic: First, the definition of " \supset " which we call "implies." Some philosophers do not like to use the label "implies" for the symbol having the truth table definition of \supset given in Table 2. By the definition in Table 2, $A \supset B$ is true whenever A is false or whenever B is true, independent of any possible causal relation between them. On the other hand, to many people the word "implies" requires a causal relation. If this discrepancy is disturbing to you then think of \supset as having some other name. The fact is that a symbol with the truth table definition of \supset in Table 2 is useful in logic, as we shall see below, regardless of what it may be called. Calling it "implies" has been a convenient if slightly misleading mnemonic.

The other more serious issue is the question of why we are concerned with validity, that is with formulas that are true for all possible models. One's first impression might be that there are very few such formulas, for example formulas like $p \vee \sim p$ and that they form an uninteresting subset of all the possible wffs. The answer is based on the fact that usually we are interested in logical consequence; that is, in whether a given formula follows from another. Suppose we wish to know whether a formula B is true whenever some other formula A is true. If this were so we would say that B is a logical consequence of A . This is a very common and useful question to pose to a logical system. We could use a Hilbert approach and pose the question to our syntactic theorem prover in the form "Assume statement A is a theorem. Now prove that statement B is a theorem." On the other hand,

this can be a confusing approach, especially from the semantic viewpoint, because in fact statement A is not valid and therefore it is not clear how to "assume it is a theorem." However, we can now use the Gentzen approach and pose the question "Prove that the formula $A \supset B$ is a theorem." Let us consider this formula semantically. If it is a theorem then every model whose valuation makes A true must also make B true. Also, any model for which A is false also makes the statement $A \supset B$ true, independent of the truth value of B. Therefore B is a logical consequence of A if and only if $A \supset B$ is true for all possible models; and the truth table definition of \supset is precisely the one we require in order to ignore the inappropriate models and focus our attention on just the ones that affect our intended interpretation of the logical consequence relation. Therefore any logical deduction argument can be converted into a single statement of the logic which is valid if and only if that argument is valid.

D. Propositional Logic Proof Procedures

1. Truth Tables

The propositional calculus is decidable. In fact, one decision procedure begins by simply enumerating all possible models. Since a model consists of a subset of the (finite number of) propositional variables that appear in a wff, there are only a finite number of models for every wff. We need merely enumerate the models, say in a table such as the one in Table 3, and calculate the valuation for the wff for each model. If all valuations are T the wff is valid and we could define it to be a theorem. In this case the syntactic theorem-proving procedure is based upon the semantics so we are guaranteed that the system is sound and complete as well as decidable. Unfortunately this truth-table procedure is frequently rather inefficient. If there are n propositional variables in a wff then

TABLE 3

Proof by Truth-Table

Theorem: $[p \supset q \wedge [q \supset r \wedge \sim s \supset \sim r]] \supset [p \supset s]$

Notation: let $A = p \supset q$

$B = q \supset r$

$C = \sim s \supset \sim r$

$D = p \supset s$

$E = B \wedge C$

$F = A \wedge E$

$G = F \supset D$, the entire theorem.

Table:

p	q	r	s	$\sim r$	$\sim s$	A	B	C	D	E	F	G
T	T	T	T	F	F	T	T	T	T	T	T	T
T	T	T	F	F	T	T	T	F	F	F	F	T
T	T	F	T	T	F	T	F	T	T	F	F	T
T	T	F	F	T	T	T	F	T	F	F	F	T
T	F	T	T	F	F	F	T	T	T	T	F	T
T	F	T	F	F	T	F	T	F	F	F	F	T
T	F	F	T	T	F	F	T	T	T	T	F	T
T	F	F	F	T	T	F	T	T	F	T	F	T
F	T	T	T	F	F	T	T	T	T	T	T	T
F	T	T	F	F	T	T	T	F	T	F	F	T
F	T	F	T	T	F	T	F	T	T	F	F	T
F	T	F	F	T	T	T	F	T	T	F	F	T
F	F	T	T	F	F	T	T	T	T	T	T	T
F	F	T	F	F	T	T	T	F	T	F	F	T
F	F	F	T	T	F	T	T	T	T	T	T	T
F	F	F	F	T	T	T	T	T	T	T	T	T

Theorem is proved because column G is all Ts.

the truth table must have 2^n lines, which might strain the capacity of the calculator.

2. Axiomatic Procedure

One alternative to the truth-table approach to propositional calculus is a Hilbert-type system such as the one used in Russell and Whitehead's Principia Mathematica.¹⁰ This system consists of a set of about one dozen axioms--particular wffs that essentially define the properties and interrelations of the connectives--and two rules of inference--"substitution" and "modus ponens." It can be shown that every theorem of propositional calculus can be derived from the given set of axioms by an appropriate sequence of applications of the rules of inference. This system was the basis for Newell, Shaw, and Simon's Logic Theory machine,³ a computer program that simulated the behavior of naive students in attempting to solve problems in the propositional calculus. Unfortunately such a system provides no guidance as to which axioms to select next or how to apply the rules of inference to them. Perhaps the most obvious mechanical procedure is the "British museum algorithm" in which all rules are systematically applied to all axioms generating all possible theorems. Of course, such an approach is clearly not a practical way to produce by computer a proof of a particular desired theorem.

3. Wang's Algorithm

Wang's algorithm⁴ is probably as efficient a procedure for mechanically proving theorems in propositional calculus as any that has ever been proposed. It consists of systematically replacing the wff currently being worked on with one or two other wffs that contain fewer connectives. When no connectives remain a simple test made on the final formulas determines whether the initial wff of the process was a theorem

or not. If there are n connectives in the initial wff then Wang's algorithm requires somewhere between n and 2^n lines dependent on the particular structure of the formula.

4. Propositional Resolution

We now present one additional decision procedure for propositional calculus. This procedure is probably comparable to Wang's algorithm with respect to efficiency. However, it is also the basis for the most useful predicate-calculus proof procedure that we will be presenting later on.

These procedures are based upon the following fact: If a wff is a theorem (and the procedure is sound) then its negation is not satisfiable; because, by the standard definition of valuation for the negation symbol, any model that satisfies the negation could not satisfy the wff and therefore the original wff could not be valid. Certain wffs, e.g. $p \wedge \sim p$, are patently not satisfiable. Moreover, if from some wff A we can deduce a wff B that is patently not satisfiable then clearly A is also not satisfiable (because " B deducible from A " means that every model that satisfies A also satisfies B and therefore if no model satisfies B then no model can satisfy A).

The proof procedure is as follows: First place a negation symbol in front of the theorem to be proved. Now place this new wff in a certain standard form called conjunctive normal form. This is a form in which negation symbols only apply to individual propositional variables and no conjunction ("and") symbol lies within the scope of any disjunction ("or") symbol. For every wff of propositional logic there exists an equivalent wff in conjunctive normal form. This equivalent wff can be constructed by systematically making substitutions using the equivalence rules given in Table 4a. Table 4b contains an example of the translation process.

TABLE 4a

Translation to Conjunctive Normal Form

Any propositional calculus wff may be converted to conjunctive normal form by recursively applying the following rules:

(1) Eliminate \Leftrightarrow and \supset .

Replace $A \Leftrightarrow B$ by $A \supset B \wedge B \supset A$

Replace $A \supset B$ by $\sim A \vee B$

(2) Reduce the scope of \sim .

Replace $\sim [A \wedge B]$ by $\sim A \vee \sim B$

Replace $\sim [A \vee B]$ by $\sim A \wedge \sim B$

Replace $\sim \sim A$ by A

(3) Eliminate \wedge from lower levels.

Replace $A \vee [B \wedge C]$ by $[A \vee B] \wedge [A \vee C]$

TABLE 4b

Example

Negated theorem:

$$\sim [[p \supset q \wedge [q \supset r \wedge \sim s \supset \sim r]] \supset [p \supset s]]$$

Eliminate \supset :

$$\sim [\sim [[\sim p \vee q] \wedge [[\sim q \vee r] \wedge [\sim \sim s \vee \sim r]]] \vee [\sim p \vee s]]$$

Reduce the scope of \sim :

$$[[\sim p \vee q] \wedge [[\sim q \vee r] \wedge [s \vee \sim r]]] \wedge \sim [\sim p \vee s]$$

$$[\sim p \vee q] \wedge [\sim q \vee r] \wedge [s \vee \sim r] \wedge p \wedge \sim s$$

(Conjunctive form--no lower level " \wedge " appears in this example.)

Set of clauses:

$$\{ \{ \sim p, q \}, \{ \sim q, r \}, \{ s, \sim r \}, \{ p \}, \{ \sim s \} \}$$

We shall call each atomic formula or its negation a literal and each disjunction of literals a clause. The entire formula in conjunctive normal form (the negation of the theorem to be proved) then has the form of a conjunction of clauses. We shall view this conjunction as a set of clauses and each clause as a set of literals. The set of clauses is satisfiable if and only if for every clause in the set there exists a literal in the clause whose valuation is T. Therefore, a set of clauses containing an empty clause, i.e. a clause without any literals, is not satisfiable whereas the empty set of clauses is satisfiable. Propositional resolution will proceed by adding or deleting clauses from this initial set until the resulting set is patently satisfiable or unsatisfiable.

We modify the set of clauses as follows: Any clause containing both a literal and its negation may be immediately eliminated since it is clearly tautological. If any literal occurs in the wff but its negation does not occur we eliminate all clauses containing that literal (the literal which is called "pure" in the wff, may be placed in the model thereby satisfying all those clauses). Any clause consisting of a single literal may be eliminated, all other clauses containing that literal are also eliminated, and all occurrences of the negation of that literal in other clauses are removed. Finally, if a literal L occurs in one or more clauses and its negation $\sim L$ occurs in one or more other clauses we may choose any subset of the clauses containing L and any subset of the clauses containing $\sim L$ and form a new clause consisting of all the literals in those two subsets except for L and $\sim L$. (This new clause is a consequence of the selected sets of clauses.)

It can be shown that if the above rules are applied systematically in all possible ways to the initial set of clauses, then in a finite number of steps we shall have a set of clauses containing the empty clause,*

*The empty clause is usually denoted by \square .

thereby proving that the original wff is a theorem, or the empty set of clauses, thereby proving that the original wff is not a theorem. Table 5 is a proof by propositional resolution.

E. First-Order Predicate Calculus

The propositional calculus is a fairly uninteresting branch of logic. The most elementary statement you can make is a proposition and there is no mechanism for looking inside of that statement. For example, the statement "The book is on the table" can be represented in propositional calculus only as a single proposition that is either true or false. There is no way to relate that statement to any other assertions about books, tables, or positions. The predicate calculus, on the other hand, allows us to discuss in the formalism individual objects and relations between objects.

Table 6 summarizes the syntax and semantics of first-order predicate calculus. In addition to the propositional connectives we now have individual variables, function symbols, predicate symbols, and quantifiers. For the semantics a model consists first of the selection of the domain, \mathcal{D} , and an assignment of individual variables to elements of the domain. Next each function symbol is assigned a mapping from ordered n-tuples of \mathcal{D} into \mathcal{D} (thus each function of an appropriate number of arguments corresponds, in each model, to a particular element of the domain). Next each predicate symbol is assigned a subset of n-tuples from the domain that it maps into truth.

We can begin to build up the wffs of predicate calculus. We first must define a term to be any appropriate composition of function symbols and individual variables. An atomic formula is then a predicate symbol applied to an appropriate number of terms for arguments. (Note that a function of no arguments is a constant and a predicate of no arguments may be considered to be a propositional variable.)

TABLE 5

A Proof by Propositional Resolution

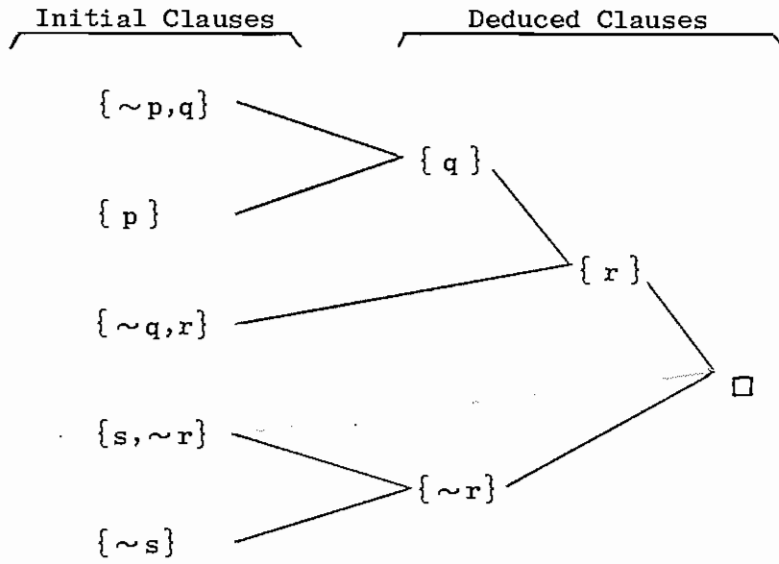


TABLE 6
First-Order Predicate Calculus

<u>Symbols</u>	<u>Definition</u>	<u>Semantics</u>
x, y, z, \dots	individual variables	elements of a domain \mathcal{D}
f, g, h, \dots	function symbols	mappings: $\mathcal{D}^n \rightarrow \mathcal{D}$, where n = number of arguments
P, Q, R, \dots	predicate symbols	subsets of \mathcal{D}^n , where n = number of arguments
\forall, \exists	quantifier symbols	[see text]
$\wedge, \vee, \supset, \Leftrightarrow, \sim$	propositional connectives	

Propositional connectives applied to atomic formula form more complex wffs in exactly the same way as they did for the propositional logic. This completes the definition of the syntax of predicate calculus except for quantifiers.

Let A_x^y be the notation for "the formula obtained by replacing every occurrence of y by x in formula A ," where y is usually an individual variable and x is a term.*

If A is a wff then $(\forall x)A$ is also a wff and its meaning, its corresponding semantic interpretation, is as follows: "Give this formula the same truth value you would get if you could compute $A_{x_1}^x \wedge A_{x_2}^x \wedge \dots \wedge A_{x_n}^x \wedge \dots$ where x_i ranges over every element of the semantic domain." $(\forall x)A$ is true iff (if and only if) the formulas you get by substituting every possible individual x_i for x are all true; otherwise $(\forall x)A$ is false. Similarly if A is a wff then $(\exists x)A$ is a wff and its corresponding interpretation is "give this formula the same truth value you would get if you could compute $A_{x_1}^x \vee A_{x_2}^x \vee \dots$ " That is, $(\exists x)A$ is true iff at least one of the instances of A for some individual x_i in the domain is true.

Finally, we define a sentence of first-order predicate calculus as a wff in which every variable is quantified. That is, if A is a wff and x occurs in A then x must occur within the scope of some quantifier $(\forall x)$ or $(\exists x)$. There must be no variable occurring inside of any sentence that is not within the scope of a quantifier. It turns out that essentially all the expressive power of first-order predicate calculus is contained in its sentences. Interpretation of nonsentences is occasionally somewhat awkward, so we shall restrict ourselves to considering the proofs of theorems in first-order predicate calculus that have the form of sentences.

* This discussion is somewhat oversimplified because we are not going into the definitions of bound and free occurrences.

F. Predicate-Calculus Proof Procedures

When we discussed the propositional calculus we observed that one method of proving theorems consisted of going to the semantics and enumerating all possible models to determine the validity of a particular wff. Although this procedure was inefficient it was always possible. In the predicate calculus, on the other hand, the domain selected in the model may be infinite and quantified statements range over all elements of the domain, so in general not only can we not enumerate all possible models but we frequently cannot even test the truth of a formula with respect to a single model in a finite amount of time. Therefore we need some other approach to testing theoremhood.

One approach frequently used by mathematicians who work in first-order predicate calculus is called "natural inference." This is a method of proving theorems in predicate calculus analogous to the axiomatic approach of Whitehead and Russell for the propositional calculus. It involves starting with zero or more initial assumptions or axioms and a fairly complex set of rules of inference, in the form of rewriting rules, that allows one to add or delete quantifiers and make various substitutions in formulas. Since this type of procedure gives no guidance to the user as to which previous statement to select next and which rule of inference to apply to that statement, it does not lead to any natural way of implementing a mechanical theorem-proving process. Although it is possible that appropriate heuristics could be superimposed upon a natural inference scheme so that practical mechanical theorem-proving programs could be based upon such an approach no one has seriously pursued this idea.

The most promising systems for automatic theorem proving in predicate calculus are based upon some ideas developed by Herbrand in about 1930. These ideas were not used by mathematicians in practical theorem-proving

work because of the complexity of the bookkeeping involved in implementing them. Now, with computers, a combination of the Herbrand approach and human-inspired heuristic-search strategies promises to give us practical and useful automatic theorem-proving systems.

One way of using the Herbrand approach resembles propositional resolution discussed above. In order to prove that a formula is a theorem we consider the negation of the formula and try to construct a satisfying model for it--or rather try to prove that it is not possible to construct a satisfying model. If such a proof can be completed then the original formula will have been shown to be a theorem. To begin such a proof, as in the propositional case, we first transform the formula into an equivalent formula in conjunctive normal form. However, the presence of quantifiers interferes with this transformation process. At this point a theorem by Skölem comes to our rescue. It turns out that for any formula containing existential quantifiers there exists another formula that does not contain any existential quantifiers and that is satisfiable iff the original formula is satisfiable. The existential quantifiers are eliminated by introducing new function symbols. For example, the formula $(\forall x)(\exists y)P(x,y)$ is a theorem iff there is a functional relationship between x and y ; that is, the above formula asserts that for every x there is some corresponding y , or equivalently, y is a function of x . Therefore, the above formula is a theorem iff the formula $(\forall x)P(x,f(x))$ is a theorem (where f is a brand new function symbol that represents the relationship between x and y).

If we reduce the scope of negation symbols according to the rules in Table 4 plus the following additional rules that allow negations to be moved past quantifiers (these are clearly generalizations of the rules in Table 4):

Replace $\sim (\forall x)A$ by $(\exists x) \sim A$

Replace $\sim (\exists x)A$ by $(\forall x) \sim A$

and we then delete every existential quantifier and instead replace the existentially quantified variables by brand new function symbols whose arguments are the appropriate universally quantified variables, then we can move all the universal quantifiers out to the front of the expression and proceed to transform the remaining formula into conjunctive form as in the propositional case. Moreover, since the quantifiers are now all in front, all universal, and quantify all remaining variable symbols in the formula, their explicit occurrence is redundant and they may be dropped. The resulting formula is the negation of the original theorem to be proved in conjunctive quantifier-free normal form, and this formula is the negation of a theorem iff the original formula was the negation of a theorem.

A remaining problem is the fact that this formula contains universally quantified variable symbols. How are we to prove that it is not satisfiable? We now have Herbrand's theorem: A formula in quantifier-free conjunctive form is not satisfiable iff some finite conjunction of its ground instances is not satisfiable. (A ground instance is a formula obtained by substituting for each variable some term that does not contain any variables.)

Early theorem-proving programs based upon the Herbrand approach proceeded as follows: First prepare an enumeration of the (generally infinite) set of ground terms that could be formed from the symbols occurring in the original wff. Second, begin using these terms to systematically generate all ground instances of the clauses in the quantifier-free conjunctive negation of the theorem. Finally, stop from time to time to test whether the set of ground clauses generated thus far can be proved to be not satisfiable using, for example, propositional resolution.

Before proceeding to a discussion of the much more efficient methods now in use let us illustrate the procedures we are discussing with an

example. I will choose an example that might come from a question-answering or problem-solving situation rather than from deep mathematics. (Since this problem will require a rather shallow proof and I will provide to the system only the small set of relevant axioms, the task should be much easier than the kinds of problems that a mathematician would like an automatic system to do in, for example, number theory or group theory.)

Suppose I wish to give the system the following two general pieces of information:

If x visits y then x is where y is, and

If u is at place v then there is a phone number at which u can be reached.

I will represent this in the logic by

(1) $(\forall x)(\forall y)(\forall z)[\text{Visits}(x,y) \wedge \text{At}(y,z) \supset \text{At}(x,z)]$, and

(2) $(\forall v)(\exists w)(\forall u)[\text{At}(u,v) \supset \text{Numb}(u,w)]$.

That is, "for all x, y, and z, if x visits y and y is at place z, then x will also be at place z" and "for every place v there is a phone number w such that for every person u, if person u is at place v, then the number at which u can be reached is w." Such a system might be used, for example, by a doctor who wished an answering service to know how he could be reached. We shall see below how the appropriate phone number can be calculated.

Let us now add a couple of specific facts. For example,

(3) $\text{Visits}(\text{Bert}, \text{John})$

(4) $\text{At}(\text{John}, \text{Tokyo-Prince})$.

We consider (1), (2), (3), and (4) all to be axioms in the system. Now suppose someone wishes to know what phone number Bert can be reached at. In predicate calculus one can easily ask yes-or-no questions by framing them in the form of desired proofs, i.e., one can ask "Is a given statement a theorem?"

We shall see a little later how additional information can also be extracted from the system. However, at this point let us simply pose the yes-or-no question, "Is it true that there exists a telephone number at which Bert can be reached?" This would appear in first-order logic as a request to prove the statement

$$(5) \quad (\exists n) [\text{Numb}(\text{Bert}, n)].$$

In order to use the Herbrand approach I must form a single statement of predicate calculus of the form $\{\text{axioms}\} \supset \text{theorem}$, and then negate that statement and place the resulting statement in clause form. It is easy to show that $\sim [A \wedge B \wedge \dots C \supset D]$ is equivalent to $A \wedge B \wedge \dots C \wedge \sim D$. Therefore, the set of clauses I must give to the Herbrand procedure consists of the clauses obtained by transforming each axiom into clauses and adding the clauses obtained from the negation of the theorem to be proved. The first of the above axioms transforms into the wff

$$(1) \quad \sim \text{Visits}(x, y) \vee \sim \text{At}(y, z) \vee \text{At}(x, z).$$

In translating the second axiom we notice that the existentially quantified variable w "depends upon" (because it follows) the universally quantified variable v . Therefore, we can replace w by $f(v)$ and eliminate the existential quantifier. The resulting clause is

$$(2) \quad \sim \text{At}(u, v) \vee \text{Numb}(u, f(v))$$

(3) and (4) are already clauses. Finally, the negation of the theorem becomes

$$(5) \quad \sim \text{Numb}(\text{Bert}, n).$$

The ground terms that can be generated from symbols appearing in these five clauses are all the terms gotten from the constants Bert, John, and Tokyo-Prince, and the function symbol of one variable f . Thus each variable occurring anywhere in the clauses may be replaced by any of the following constants: Bert, John, Tokyo-Prince, $f(\text{Bert})$, $f(\text{John})$, ..., $f(f(f(\text{Tokyo-Prince})))$..., and so on. A blind syntactic procedure of substituting these

ground terms into these clauses will generate clauses whose interpretations might be "either the Tokyo-Prince is not visiting the Tokyo-Prince or the Tokyo-Prince is not at the Tokyo-Prince or the Tokyo-Prince is at the Tokyo-Prince" and "Bert is not at $f(f(\text{John}))$ or the number of John is at the Tokyo-Prince," and so on. Eventually we will get to a useful clause such as the one meaning "either Bert is not visiting John or John is not at the Tokyo-Prince or Bert is at the Tokyo-Prince" but it may take an extremely long time before that point is reached.

In 1965 Robinson published a paper⁵ describing the Resolution principle, an inference rule that basically shows how it is possible to avoid generating most of these ground instances. Instead one need only generate a limited number of much more meaningful terms by using a matching procedure called unification. This means that we decide which terms to select from the many possible ones by picking just those terms that allow us to make some progress toward eventually deducing an empty clause and thereby proving unsatisfiability.

Resolution is similar to propositional resolution except, obviously, that it does not have to begin with propositions. In fact, one picks any two clauses such that a literal in one contains the same predicate symbol and opposite sign as some literal in the other. If the corresponding arguments of those literals can be made identical by some substitution, then by making that substitution in both clauses the two clauses become candidates for a propositional resolution step. An algorithm exists for finding the "most general unifier" of any set of literals. By using most general unifiers--which do not necessarily produce ground substitutions--each unification step is an abbreviation for a possibly infinite number of specific ground instantiations. Thus the resolution procedure can proceed toward a proof in a much more efficient well-directed manner than any procedure based upon a systematic enumeration of ground instances.

Returning to the example (see Table 7^{*}) we can take any two lines, say (5) and (1), and see immediately whether there is any chance of producing a significant deduction from those lines. Since the clause in line (5) contains the single literal which has a negated occurrence of the predicate Numb we look in line (1) for any occurrence of a positive predicate Numb. Since there is none we have no reason to pursue consideration of this pair of lines. Now look at (2) and (5). (5) has a \sim Numb and 2 has a Numb. Therefore it appears worthwhile to look for a substitution that will make the arguments of the Numb in (2) and the \sim Numb in (5) identical. One such substitution is Bert for u and f(v) for n. If I make this substitution throughout lines (2) and (5) and then form a new clause according to the propositional resolution rule, that new clause contains only a single literal in line (6). If I now try to resolve that new clause with something I have only two candidates, that is, only two clauses that contain a positive occurrence of the predicate symbol At: clauses (1) and (4). However (4) can be quickly rejected because there is no way to make the first argument of 4, John, identical to the first argument of (6), Bert, (since John and Bert are both constants). So if (6) is to be useful at all it must be resolved with (1). Therefore in clause (1) let x be Bert and z be v. This will produce clause (7). Now (7) and (3) immediately produce (8), by letting y be John. And finally, clause (8) and clause (4), with the substitution $z = \text{Tokyo-Prince}$, are immediately contradictory, that is they may be resolved to produce the empty clause and complete the contradiction.

Thus unification and the resolution rule provide a powerful way of testing a set of clauses for unsatisfiability. However, the resolution principle alone is not sufficient for constructing efficient machine proofs. Making all possible resolution inferences from a set of clauses is clearly much less time consuming and inefficient than making all possible ground

* In Table 7, the circled literals should be ignored until the reader considers the answer-construction process, to be discussed in Section G2.

TABLE 7

Proof by Resolution

(1)	$\sim \text{Visits}(x,y) \sim \text{At}(y,z) \text{ At}(x,z)$	}	Axioms
(2)	$\sim \text{At}(u,v) \text{ Numb}(u,f(v))$		
(3)	$\text{Visits}(\text{Bert},\text{John})$		
(4)	$\text{At}(\text{John},\text{Tokyo-Prince})$		
(5)	$\sim \text{Numb}(\text{Bert},n) \text{ Numb}(\text{Bert},n)$		Negated Theorem
(6)	$\sim \text{At}(\text{Bert},v) \text{ Numb}(\text{Bert},f(v))$		From (2) and (5)
(7)	$\sim \text{Visits}(\text{Bert},y) \sim \text{At}(y,v) \text{ Numb}(\text{Bert},f(v))$		From (1) and (6)
(8)	$\sim \text{At}(\text{John},v) \text{ Numb}(\text{Bert},f(v))$		From (3) and (7)
(9)	$\square \text{ Numb}(\text{Bert},f(\text{Tokyo-Prince}))$		From (4) and (8)

substitutions into a set of clauses. However, it is still not sufficiently restrictive to produce proofs to interesting theorems in reasonable times. We must superimpose upon the general principle of resolution some set of guiding heuristics that determine which clauses to select for trying resolution on next and which literals within those clauses to focus attention on first. Considerable work has gone into selecting such principles and heuristics in the last few years. A large variety of principles have been published, generally asserting that one may restrict the use of resolution to extremely narrow subsets of the available clauses and still produce proofs in finite time if any proof is possible. Unfortunately, most of these strategies for using resolution do not have anything to say about the average overall efficiency of the proof. That is, the proof found when using such strategies may require considerably fewer alternatives to be explored at each point in the search tree but may also may require a much deeper search before finding the first complete proof and therefore may not contribute to the overall efficiency of the procedure.

Certain of these strategies, for example the "unit-preference strategy," which says essentially "since we are trying to deduce the empty clause always pick as one of your resolvents the shortest clause available and try to resolve away some literals," seem to be extremely important. The usefulness of other proposed strategies is nowhere nearly as clear. Experimental work is just now being started in several places to gather statistics and compare the effectiveness of various strategies upon various types of theorem-proving problems.

Since this work is being pursued by the mathematicians who wish to develop effective theorem provers for complex problems, we should be able to take advantage of their results and apply the latest theorem-proving strategies, as they are developed, to our needs for shallow proofs in much broader problem domains such as question answering and problem solving.

G. Applications of Resolution Theorem Proving

Let us turn our attention now to ways of using predicate-calculus proof methods in broader systems for Artificial Intelligence.

1. Levels of Memory

The mathematical theorem-proving systems are being tailored to produce efficient proofs in situations where a small set of relevant axioms is supplied to the theorem prover at the beginning of its proving process. The selection of such a set of axioms is a heuristic process that may require completely different techniques than those built into the mathematical theorem prover. Therefore it seems wise to have two or more levels of memory in our problem-solving system. The primary memory can be limited to a small number of axioms that some program has decided is relevant to the problem at hand. A secondary, larger memory can consist of all knowledge that the computer has of the appropriate problem domain. We then can modify the theorem prover itself so that it would in a sense "know" about the availability of this large background memory but not have direct access to all the facts stored there. Instead the theorem prover can converse with another program--one responsible for identifying relevant facts--and ask for additional axioms only when the theorem prover decides that it does not have enough information to solve a particular problem. The QA3 system⁹ at SRI essentially uses such a two-level memory system; however, the criteria for transferring facts from secondary to primary memory are presently extremely crude and much more work needs to be done on this aspect of the system.

2. Answer Construction

As we mentioned above a theorem prover is designed only to answer the yes-or-no question--Is this statement a theorem? In question answering, on the other hand, we generally wish not only to know whether an answer

exists but also to exhibit the answer. Luckily it turns out that the Herbrand-type of theorem-proving procedure generally constructs considerably more data about the answer than simply the fact that it exists. However, in mathematically oriented theorem-proving programs this data is never brought out explicitly and made available to the user.

Luckham and Nilsson¹¹ describe an ingenious procedure for generating from a resolution-type proof the most complete description of the answer that can be obtained. A simplified form of this procedure involves going back to the initial unsatisfiable set of clauses and adding one or more literals that convert the set from being unsatisfiable into being tautological. Then, by following the steps of the initial proof, the additional literals serve as bookkeeping devices to extract the answer information from the proof.

In Table 7 the circled literals are part of this answer-construction procedure. We see that the answer to the question, "At what telephone number can Bert be reached?" is $\text{Numb}(\text{Bert}, f(\text{Tokyo-Prince}))$, that is, Bert can be reached by the number gotten by calculating the f function of Tokyo-Prince. We then must recall that f is a function relating places to their phone numbers, so that "calculating the f function" means looking up the Tokyo-Prince in the telephone director.

3. Evaluable Predicates

If we were doing pure mathematics we would be satisfied with the basic idea in logic of carrying all significant information in the axioms themselves. If we were studying number theory, for example, we would be perfectly content to write the axioms for an elementary theory of numbers and see what we could prove from them. On the other hand, in applying theorem proving to more practical problems we occasionally wish

to do some calculation, for example some arithmetic, that is only incidental to the main problem. For example, in the example of Table 7, suppose we wish also to add the requirement that Bert is visiting John only after 1700 O'clock. Then we might change axiom 3 from $\text{Visits}(\text{Bert}, \text{John})$ to:

(3') $\text{Time}(x) \wedge \text{Greater}(x, 1700) \supset \text{Visits}(\text{Bert}, \text{John}),$

and we would add the current time as a new axiom

(0) $\text{Time}(1930).$

Ideally, these changes should have no significant effect on our proof. We should be able to deduce very quickly from Axioms (0) and (3') that since it is after 1700 and Bert visits John after 1700 then Bert is visiting John. Unfortunately, if we resolve (0) with the clause form of (3') we get:
 $\sim \text{Greater}(1930, 1700) \vee \text{Visits}(\text{Bert}, \text{John}).$ We have no convenient formal way of handling the Greater relation, i.e. of noticing that $1930 > 1700$. It would be ridiculous to add all the axioms of ordering on the integers and in fact all the definitions of integers represented in a logical syntax in order to do this trivial computation. Therefore, we require that resolution proof procedure be augmented by specific calculational procedures whenever such an approach is more convenient. In this case we need merely flag the Greater relation so that the proof procedure would notice, any time a literal containing the Greater predicate is modified, that this is one of the special class of predicates that can sometimes be evaluated. In particular, as soon as all the arguments of Greater are numeric, an arithmetic comparison subroutine can be called and the literal can be replaced by truth or falsity. This device of calculating the values of certain predicates when such calculations are convenient and leaving them in the formalism for the logic system to deal with when they cannot be easily calculated has proved to be a valuable concept in the construction of useful resolution theorem-proving systems.

We can extend the concept of evaluable predicate to that of evaluable function. In the above example there is no reason why the function f could not be flagged as evaluable. As soon as its argument was instantiated, i.e., known to be Tokyo-Prince, an information-retrieval subroutine could be called that would look up the telephone number of the Tokyo-Prince in an appropriate data file and produce the specific answer to the question.

4. Generating Plans

Cordell Green proposed the intriguing idea of using a theorem prover to generate plans of action for a robot (or programs for a computer). The concept was based on that of answer construction; namely, if one wished a plan of action that would achieve a certain result one could simply pose to the theorem prover a problem of the form

Prove that there exists a plan of action that achieves
the desired result.

If the initial state of the world and the effects of the possible action operators are appropriately axiomatized then a standard resolution theorem prover can indeed prove that there exists the desired state, and the answer-generation procedure applied to that proof would construct the appropriate sequence of operators. Unfortunately this approach has some severe limitations in efficiency and it has proved to be more elegant than practical. Examples of the problem and a current approach for getting around it will be discussed in the next lecture.

REFERENCES

1. H. Gelernter, J. Hansen, and D. Loveland, "Empirical Explorations of the Geometry Theorem Proving Machine," in Computers and Thought, pp. 153-167, E. Feigenbaum and J. Feldman, Eds. (McGraw-Hill Book Company, New York, NY, 1963).
2. J. Slagle, "A Computer for Solving Problems in Freshman Calculus (SAINT)," Doctoral Dissertation, Massachusetts Institute of Technology, Cambridge, Mass., 1961. Also printed as Lincoln Laboratory Report 5G-0001, MIT, Cambridge, Mass., May 10, 1961.
3. A. Newell, J. Shaw, and H. Simon, "Empirical Explorations of the Logic Theory Machine," in Computers and Thought, pp. 109-133, E. Feigenbaum and J. Feldman, Eds. (McGraw-Hill Book Company, New York, NY, 1963).
4. H. Wang, "Towards Mechanical Mathematics," IBM Journal of Research and Development, Vol. 4, pp. 2-22 (1960).
5. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," Journal of the ACM, Vol. 12, No. 1, pp. 23-41 (January 1965).
6. J. Guard et al., "Semi-Automated Mathematics," Journal of the ACM, Vol. 16, No. 1, pp. 49-62 (January 1969).
7. J. McCarthy, "Programs with Common Sense," in Mechanization of Thought Processes, Vol. I, pp. 77-84, Proceedings Symposium, National Physical Laboratory, London, November 24-27, 1958. Reprinted in Semantic Information Processing, pp. 403-410, M. Minsky, Ed. (The MIT Press, Cambridge, Mass., 1968).
8. A. Newell and G. Ernst, "The Search for Generality," Proceedings IFIP Congress '65, Vol. 1, pp. 17-24 (Spartan Books, Washington, DC, 1965).
9. T. D. Garvey and R. E. Kling, "User's Guide to QA3.5 Question-Answering System," Technical Note 15, Artificial Intelligence Group, Stanford Research Institute, Menlo Park, CA (December 1969).
10. A. N. Whitehead and B. Russell, Principia Mathematica (Cambridge University Press, Cambridge, England, 1910).

11. D. Luckham and N. J. Nilsson, "Extracting Information from Resolution Proof Trees," Artificial Intelligence, Vol. 2, No. 1, pp. 27-54 (North-Holland Publishing Company, Amsterdam, The Netherlands, 1971).
12. C. Green, "Application of Theorem Proving to Problem Solving," Proceedings IJCAI (The Mitre Corporation, Bedford, Mass., 1969).