AD-755 719

THE ROLE OF THEOREM PROVING IN ARTIFICIAL INTELLIGENCE

Hartmut G. M. Huber

Naval Weapons Laboratory
Dahlgren, Virginia

November 1972

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified

1. ORIGINATING ACTIVITY *(Corporate author)* Naval Weapons Laboratory Dahlgren, Virginia 22448	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

THE ROLE OF THEOREM PROVING IN ARTIFICIAL INTELLIGENCE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*

Hartmut G. M. Huber

6. REPORT DATE November 1972	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S) TR-2864	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	
d.		

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

~~This~~ paper describes and evaluates theorem proving and its role in artificial intelligence in non-technical terms. It discusses the general principles underlying automatic theorem proving on the computer and considers the different strategies and techniques that are used for improving performance. It is shown by examples that theorem proving plays a central role in artificial intelligence. The application of theorem proving to automatic program writing is treated in detail. A candid evaluation of the situation will reveal that further research in specific directions is desirable and that certain other areas do not appear to be promising in the near future.

NWL Technical Report No. 2864
November 1972

THE ROLE OF THEOREM PROVING IN ARTIFICIAL INTELLIGENCE

by

Hartmut G. M. Huber

Warfare Analysis Department

1D

FOREWORD

Artificial intelligence in many of its facets has become an esoteric field that is hard to be evaluated and properly appreciated by an individual who does not work in the field. Experience indicates that research in this field carries perhaps a greater risk of failure than other research projects. In this situation, the decisions to allocate funds for particular projects become very difficult. The purpose of this report is to alleviate this problem by providing an introductory survey of some of the ideas and techniques that have evolved in the vast field of artificial intelligence in recent years, and by indicating some of the research areas that appear to be promising.

The work reported here was funded under the Independent Exploratory Development program element 62713N. This paper is the final report on the research project in the field of general decision making and theorem proving on the computer conducted at the Naval Weapons Laboratory between FY 1969 and FY 1972. The project was terminated in June 1972.

This report was reviewed by Mr. Alfred H. Morris, Jr. and Mr. Hermon Thombs of the Programming Systems Branch.

Released by:

[illegible]

RALPH A. NIEMANN
Head, Warfare Analysis Department

ABSTRACT

This paper describes and evaluates theorem proving and its role in artificial intelligence in non-technical terms. It discusses the general principles underlying automatic theorem proving on the computer and considers the different strategies and techniques that are used for improving performance. It is shown by examples that theorem proving plays a central role in artificial intelligence. The application of theorem proving to automatic program writing is treated in detail. A candid evaluation of the situation will reveal that further research in specific directions is desirable and that certain other areas do not appear to be promising in the near future.

CONTENTS

I. INTRODUCTION

Artificial intelligence in many of its facets has become an esoteric field that is hard to be evaluated and properly appreciated by an individual who does not work in the field. Experience indicates that research in this field carries perhaps a greater risk of failure than other research projects. In this situation, the decisions to allocate funds for particular projects become very difficult. The purpose of this report is to alleviate this problem by providing an introductory survey of some of the ideas and techniques that have evolved in the vast field of artificial intelligence in recent years, and by indicating some of the research areas that appear to be promising.

The objective is to build an intelligent machine capable of sensing and understanding its environment, of reasoning about it, of making sensible decisions under uncertainty, and of controlling its environment. For example, Stanford Research Institute has built a robot equipped with wheels, arms, and a television camera as an eye, connected to a computer as its brain [29]. However, most researchers carry out theoretical studies and write computer programs that perform more specific tasks such as playing chess [21], proving mathematical theorems [33,13], making conversation and solving simple problems, and constructing programs from general non-algorithmic specifications of the program [40]. These computer programs normally require search in a large network and require complex decisions based on incomplete information at the point where the decision must be made. Among the approaches for this decision making process, we single out one and call it the axiomatic approach. Here the original search and decision problem is reformulated as a problem to prove a theorem from a set of hypotheses, and then theorem proving methods and techniques for the solution of tnis problem are applied. Other approaches may include assigning weights to the nodes in the network and making the decisions by searching through the network and evaluating these weights or possibly modifying them. In many cases a wide variety of tricks and heuristics not based on a well developed theory is employed.

It is the purpose of this paper to describe and evaluate the axiomatic approach in non-technical terms. We shall discuss the general principles underlying automatic theorem proving on the computer and consider the different strategies and techniques that are used for improving performance. We also show by examples that theorem proving plays a central role in the field of artificial intelligence. A candid evaluation of the situation will reveal that further research in specific directions is well worth the money and that certain other areas do not appear to be promising in the near future.

II. AUTOMATIC LOGICAL REASONING

Logical Reasoning involves analyzing given sentences (also called statements, formulae, well formed formulae) and inferring new logically valid sentences from them. Therefore, a language is needed in which sentences can be formulated, and rules of inference are needed to produce new logically valid sentences from old ones. A language together with some rules of inference constitutes a system of logic. A ***first order predicate calculus*** is such a system of logic comprising a first order language L and some rules or inference.

The elements of a first order language L are constants, variables, function symbols, predicate symbols, several propositional connectives such as $\sim$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), and the quantifiers ***for all*** and ***there exists***. A ***predicate*** is just a function with $\{$true, false$\}$ as its range of values. "First order" means that functions do not take functions as arguments and that the domain of the variables always contains data, not functions. Thus, in a first-order language one cannot express directly the situation that a certain statement involving a function symbol f is true for all functions over the same domain as f. However, a first-order language is still very rich and much of mathematics can be expressed quite conveniently in it. Certain elements of the language, namely the propositional connectives and the quantifiers, have a fixed universal interpretation (meaning). However, constants, function and predicate symbols may be given different meanings depending on the circumstances For example, consider the statement

$$(x)(y)(Ez)(x = f(y,z))$$

which asserts that "for all x and y there exists a z such that $x = f(y,z)$". We may interpret this statement as follows: The variables x,y,z range over the set of integers, the symbol = denotes the equality relation, $f(y,z)$ is interpreted as the sum of y and z. Then the above statement is true under this interpretation of the data x, y, z and the operators = and f, and we may say that this interpretation is a ***model*** for the statement. If, however, we modify this interpretation by letting $f(y,z)$ denote the product of y and z, then the above statement is false since for $x = 1$ and $y = 0$ there is no z such that $1 = 0 \cdot z$. This modified interpretation is not a model for the above statement, call it A, but is a model for its denial $\sim$A.

The second part of a first-order predicate calculus is a set of rules of inference that generate logically valid statements from given ones. This set should be powerful enough to generate all the logical consequences from a given set S of sentences, or at least all of the important ones. In automatic theorem proving, we normally deal with inconsistent (contradictory) sets S. Here the rules of inference should be powerful enough to generate the explicit contradiction ***false***, in which case the rules of inference are called ***complete***. An example of a rule of inference which, by itself, is not complete, is the so-called ***modus ponens*** which produces from the two statements A and $A \Rightarrow B$ the statement B. In the sequel, we shall normally consider only one rule of inference, namely resolution.

In a first order predicate calculus, a certain set Σ of basic statements is always true. The statements in Σ are called logical axioms. If we further require that an additional set Γ of statements be true, then the first-order predicate calculus becomes a ***first order theory***. In this case Γ is called the set of (proper) axioms or hypotheses and the logical consequences of $\Gamma \cup \Sigma$ are called theorems. This is the general picture. For further details, the reader is referred to the excellent book by Mendelson [28].

We can now formulate the basic problem in theorem proving as follows: If $A_1, \cdots, A_n$ and C are statements, then prove that C follows from $A_1, \cdots, A_n$; i.e., prove that $A_1 \wedge \cdots \wedge A_n \Rightarrow C$ is a logically valid statement being true for all interpretations. By a proof of C from $A_1, \cdots, A_n$ we mean a sequence of statements $(B_1, \cdots, B_m)$ such that B_m is C, and each B_i is either a given statement A_j or is the result of a rule of inference applied to some of the previous statements $B_1, \cdots, B_{i-1}$.

This problem might appear to be very difficult, and yet, it is well known that it can be solved by a simple mechanical algorithm if it can be solved at all, that is, if C does indeed follow from $A_1, \cdots, A_n$. The algorithm is based on the fact that all proofs in a predicate calculus can be systematically enumerated. Therefore, if there is a proof of C from $A_1, \cdots, A_n$ then a brute force search through this enumeration will turn up a proof and the problem is solved. However, if no proof exists then the algorithm searches forever. In general, it is not possible to decide by an algorithm whether a statement is logically valid or not. A procedure that solves the problem stated above is called a ***proof procedure***.

Even though there exists in principle a proof procedure, we must consider the problem of actually finding a proof of C from $A_1, \cdots, A_n$ as unsolved. Searching through the set P of all proofs, where a uniform strategy is employed that provides no intelligent guidance to a small subset of P that contains a proof, is simply out of the question. The set P is too gigantic. In practice, it is as impossible as to search through the finite set of all chess games. Therefore, the existence of such an algorithm just states that the problem is computable, that it makes sense to try to construct a computationally feasible procedure. For a noncomputable problem, such an undertaking would be futile.

Basic research in theorem proving during the past ten years has concentrated on developing general techniques, as well as special strategies for certain special theories, that help to construct computationally feasible proof procedures. These procedures use the proof by contradiction method. If A is the statement to be proved, then the denial $\sim A$ of A is adjoined to the set Γ of hypotheses and this combined set of statements must now be inconsistent; i.e., must lead to a contradiction. It is easy to show that $\Gamma \cup \{\sim A\}$ is inconsistent if and only if A is a logical consequence of Γ. This means that our everyday mathematical reasoning using the proof by contradiction method is sound, of course.

It turns out that the statements one is dealing with in theorem proving can all be assumed to be in a very simple form. This is the case because any finite set Σ of statements can be transformed into a finite set S of statements in ***quantifier free normal form*** such that S is inconsistent if and only if Σ is inconsistent. Such a transformation is described by M. Davis [7] and is easy to implement on a computer. The transformed set S is a finite set of elements called ***clauses***, each clause is a finite disjunction of items called ***literals***, and each literal is either an atomic formula or a negated atomic formula. An ***atomic formula*** is, as may be expected, a well formed formula that does not contain any propositional connectives or quantifiers. We also write a clause $\ell_1 \vee \cdots \vee \ell_k$ just as a list of literals $(\ell_1, \cdots, \ell_k)$. A clause $C = (\ell_1, \cdots, \ell_k)$ represents the statement $\overline{C} = (x_1) \cdots (x_n)(\ell_1 \vee \cdots \vee \ell_k)$ where $x_1, \cdots, x_n$ are all the variables occuring in the literals of C and each (x_i) means "for all x_i". Thus all variables in a clause are implied to be universally quantified. A set of clauses $S = \{C_1, \cdots . C_m\}$ represents the statement $\overline{C}_1 \wedge \cdots \wedge \overline{C}_m$. The empty clause denoted by $\square$, plays a special role. This clause can never be made true by any interpretation and represents, therefore, a contradiction.

We see that the general theorem proving problem is now reduced to the problem of showing that a finite set of clauses is inconsistent. An algorithm that solves this inconsistency problem is called a ***refutation procedure***.

III. REFUTATION PROCEDURES FOR SETS OF CLAUSES

If a statement is true in general, that is, for all values of the variables occurring in it, then it is true for specific cases. Therefore, replacing variables in a clause C by variables or constant expressions will produce a new clause C′ which is a logical consequence of C. We say C′ is an ***instance*** of C, and if C′ does not contain any variables then C′ is called a ***ground instance*** of C.

Early refutation procedures [7,8] were based on the fundamental fact known as the ***Herbrand Theorem***, that a set S of clauses is inconsistent if and only if a finite set of ground instances of S is inconsistent. Thus we can produce from the set S progressively larger sets S_n of ground instances of clauses in S and check if S_n is inconsistent. This will give a refutation procedure if the sets S_n $(n = 1,2,\cdots)$ are generated in a certain systematic manner. The inconsistency of a set of ground clauses can be checked efficiently using the Davis-Putnam procedure [8,13].

The difficulty with this approach was the enormous growth of the sets S_n. To alleviate the situation, a method had to be found that avoided the ground instantiation. Such a method was proposed by J. A. Robinson in 1965 [32].

Robinson succeeded in defining a rule of inference called ***resolution*** or ***resolution operator*** which produces one or more new clauses called ***resolvents*** from two given parent clauses and has the important property that it is ***sound*** and ***complete***. This means that a resolvent is a logical consequence of the parent clauses and if the original set of clauses is inconsistent, then this rule of inference will eventually turn up the explicit contradiction represented by the empty clause $\square$.

Resolution can be considered as a generalization of the modus ponens. The following description may give a rough idea of how it works. Two ground clauses have a resolvent if one clause contains a literal which occurs negated in the other clause. Thus, if $C = (\ell,\ell_1,\cdots,\ell_k)$ and $C' = (\sim\ell,\ell'_1,\cdots,\ell'_m)$ then the union of the literals $\ell_1,\cdots,\ell_k$ in C and $\ell'_1,\cdots,\ell'_m$ in C′ is a resolvent of C and C′. The literals ℓ and $\sim\ell$ are said to be ***complementary***. Thus, resolution on the ground level is elimination of complementary literals. If the parent clauses C and C′ contain variables then first a substitution is made that produces complementary literals and then these complementary literals are eliminated just as in the ground case. The heart of the matter is to find these substitutions and it was probably Robinson's greatest achievement to define an algorithm called ***most general unification algorithm***, that computes the proper substitutions if there are any. This algorithm is of crucial importance in present day theorem provers and plays the fundamental role of basic arithmetic in scientific programs. An efficient implementation of it has recently been proposed by Robinson [35].

It is important that resolution is defined in such a way that the set of all resolvents of a finite set of clauses is finite. Therefore, one can generate all of the resolvents from a given set S, adjoin the new clauses to S, and compute again all resolvents. In this manner, layer after layer of resolvents is generated. This process terminates when □ is generated. This must occur eventually if the set S of clauses is inconsistent since resolution is complete. The history of generating □ then represents a proof. A sequence $(C_1,\cdots,C_n)$ such that $C_n = □$ and each C_i is either in S or is a resolvent of some previous clauses $C_1,\cdots,C_{i-1}$ is called a ***refutation of S***. Recalling the concept of a proof of a formula, we see that a refutation is just a proof of □ from S.

This new approach, revolutionary as it was, still suffered from the fact that, before a refutation was found, too many resolvents had to be computed, the majority of which were irrelevant ballast. The question was: How can one make the resolution operator more selective so that ***a priori*** the resolvents of many clauses, are recognized to be irrelevant for the proof under consideration and are therefore never computed? This idea of defining a restrictive (or refined) resolution operator which is still complete underlies most papers that appeared after the introduction of resolution. It is indeed possible to put a variety of conditions on the clauses and to compute resolvents only if these conditions are satisfied. We give a brief outline of the most important cases.

An interpretation will assign to every statement a truth value. Therefore, an interpretation will partition any set of clauses into two disjoint sets, one set containing true clauses and the other containing false clauses. It can be proved that the only resolvents that need be computed are those which can be derived from clauses, one of which is false under an arbitrary but fixed interpretation. Of course, this fact gives rise to a great variety of refined resolution strategies depending on the interpretation being used. They are all called ***semantic strategies*** since interpretations are involved. There are many interesting special cases. For example, if there are no false clauses then obviously the set S is consistent since the interpretation is a model of it and we are done. In this case, no proof of □ can be found. Semantic strategies were discovered by Slagle [37] and independently in a slightly weaker form by Luckham [24]. A weak but very important version of a semantic strategy, called the ***set of support strategy***, was proposed very early by Wos, Carson and G. A. Robinson [41].

A second type of condition that can be imposed on the resolution operator has to do with a geometric property of refutations. One can prove that if S is inconsistent then there is always a ***linear refutation***. A linear refutation is a sequence $(C_1,\cdots,C_{n-1},C_n)$ such that C_n is □ and such that each C_i is in S or is a resolvent of C_{i-1} and a clause which is in S or is some previous C_j, $j < i$.

Certain other conditions can be combined with linearity resulting in finer strategies. Unfortunately the semantic condition cannot be combined with linearity without losing completeness. ***Linear strategies*** have been discussed by Luckham [25], Anderson and Bledsoe [2], Loveland [23], Kowalski and Kuehner [19], Reiter [31], Huber and Morris [14,15], and Chang and Slagle [5].

A third kind of condition can be obtained from partially ordering the literals occurring in S (***A-Ordering***) or totally ordering the clauses in S (***C-Ordering***). Strategies using A-Ordering or C-Ordering have been introduced by Slagle [37] and Reiter [31]. They are also discussed in Kowalski and Hayes [18] and Huber and Morris [13,15].

Frequently the situation occurs that a clause C' in S is a consequence of a clause C in S. As may be expected, the weaker clause C' can normally be discarded. Strategies that allow this are called ***subsumptive*** [13,14].

It appears that the attempts to define restricted resolution operators that give rise to a drastically improved strategy have hit the end of the road. We have now much finer strategies than, say, seven years ago, and we can now solve simple theorem proving problems that could not be solved seven years ago. And yet, it is quite discouraging to see that more complex problems are still far out of the range of present day theorem provers using those strategies. There are two avenues of research which have been tried and which are promising to improve the situation. One is to use larger inference steps and the other is to use special inference rules for handling certain special basic symbols like = or $\in$.

The inference rules considered so far draw a conclusion from two statements (clauses). One would expect that one could draw better conclusions from more than two statements. Therefore proofs based on inference rules that use more than two statements should be shorter and easier to compute. Such inference rules have been proposed by Robinson [33] (***clash resolution***) and by Slagle [37] (***maximal clash resolution***). It turns out that maximal clash resolution is a great tool for theoretical work, but for practical computation this extended inference rule does not appear to help much. It is certainly true that these inference rules do not have to be applied as frequently as ordinary pairwise resolution, but, unfortunately, one application is so tremendously much more involved.

There is, however, another way of obtaining better inference rules. Instead of looking at more clauses, the machine can be made to remember and put to use the history of previous inference steps. This is a totally new idea. The resolution operators considered so far are completely local in their scope. When the operator is applied, then the resolvent is computed and kept if it is new, but the rest of the information that was available when the resolvent was inferred is 'iscarded and the next inference step is unaware of what the previous steps did. One certainly would not expect much of a person who is so totally unaware of past events, even if he is quite reliable and fast in his work. It was this fascinating idea to save the relevant history of previous resolution steps and to put it to use when the next inference is made which was systematically developed in the paper Contracted Resolution by the author and A. Morris [15]. The technical approach of Contracted Resolution is based on Loveland [22]. Some of the results of Contracted Resolution have been discovered independently and published earlier by Kowalski [19] but the authors of [15] were not aware of this. A theorem proving system based on Contracted Resolution was written is LISP for the CDC 6700 computer by Huber and Morris. The performance of this system compared favorably with the other theorem provers.

The second approach to improve the efficiency of theorem provers is to handle the basic mathematical relations like =, $\in$, $\subset$ by special inference rules. The general idea is to absorb most of the properties of these relations into an inference rule that is used in addition to resolution and to drop the axioms that define these properties explicitly. For example, Wos and G. Robinson proposed and proved the completeness of such an inference rule for the equality relation, called ***paramodulation*** [42], which is essentially a substitution rule. In this case, most of the axioms describing the properties of = need not be included in the set of hypotheses from which the proposed theorem is supposed to follow. Only certain "reflexivity clauses" must be included, namely $x = x$ and $f(x_1,\cdots,x_n) = f(x_1,\cdots,x_n)$ for each function symbol f occurring in the set S of clauses, where n, of course, is the number of arguments of f, and x and $x_1,\cdots,x_n$ are variables. It is still an open problem if paramodulation together with resolution is complete if all the reflexivity clauses are dropped except $x = x$.

Primary paramodulation is a much more restrictive inference rule, where substitution is normally made only into the arguments of the predicate symbol but not into a subexpression embedded deep inside of an atom. Completeness of paramodulation together with resolution was proved by Wos and G. Robinson [42], the completeness of primary paramodulation together with resolution independently by Kowalski [17], Chang [4], and by Huber and Morris [14]. This approach of handling the equality symbol by a special inference rule was quite successful and one would like to do the same thing for other basic symbols like $\in$ and $\subset$. Slagle attempts in [38] to formalize this approach and to set up a technology for defining special inference rules for basic mathematical symbols. However, it is not clear yet how effective this is for practical computation.

In summing up this section, we must say that even though much progress has been made in the field of theorem proving in recent years, it is still by far not enough to handle complex problems. It must be realized that in the applications, which will be discussed in the next section, the theorem prover is the weak point. With a powerful theorem prover some of these applications could become quite spectacular.

IV. APPLICATIONS

Resolution based theorem proving plays a central role in the field of artificial intelligence. For example, it can be used directly for proving a proposed theorem in, say, elementary group theory In this case a theorem prover would be asked the question: "Why does statement A follow from the set Γ of axioms?" The answer will be a proof of A from Γ, actually a refutation of $\Gamma \cup \{\sim A\}$. More generally, one could ask: "Does statement A follow from Γ?" The possible answers would be: "*Yes*, here is a proof of A from Γ: ••••", or "*No*, here is a proof of ~A from Γ: ••••", or "I cannot find the answer within the given time limit". This last alternative cannot be eliminated, no matter how much computation time is allowed, since the predicate calculus is undecidable, (Church [6]).

However, a system having theorem proving capacity can be used for solving many other problems. The general approach [10,11] is this: Facts about the problem environment are represented as statements in a first-order language. Questions are formulated as conjectures in this language to be proved. An extended theorem prover [10] will construct a proof of the conjecture and as a by-product of this it will generate an answer statement to the question associated with the conjecture. Using this approach C. Green [11] was able to apply theorem proving techniques to general question answering, general problem solving (the tower of Hanoi puzzle, the monkey and the bananas problem), robot problem solving of the Stanford Research Institute robot, and automatic program writing.

The problem of constructing an answer to the original question can also be separated from the problem of finding a proof of the associated conjecture [26]. Using this approach, the answer statement is derived from the refutation of the denial of the conjecture and all hypotheses by a general extraction process.

All applications mentioned above are still in an experimental stage. To illustrate the ideas more clearly, we shall now describe in some detail the problem of automatic program writing which is possibly one of the most general and attractive applications.

When a programmer writes a program he represents functions as algorithms in some programming language. How difficult this task is depends to a great degree on how the functions are defined originally.

Functions may be defined in many different ways which can be classified as algorithmic and non-algorithmic definitions. An algorithmic definition specifies in a step-by-step or recursive manner how for any given argument the function value is to be computed, whereas in a non-algorithmic definition merely the relationship between the argument and the function value is given. For example, consider the greatest common divisor of two integers:

(1) gcd: gcd(x,y) is the largest positive integer that divides both x and y if $x \neq 0$ or $y \neq 0$.

(2) gcd: $(x)(y)\{x \neq 0 \vee y \neq 0 \Rightarrow (Ez)[z/x \wedge z/y \wedge (u)[u/x \wedge u/y \Rightarrow u \leq z]]\}$

(3) gcd:
```
(lambda(x y)(prog(z)
A       (cond((zerop y)(return(abs x))))
        (setq z(divide x y))
        (setq x y)
        (setq y(cadr z))
        (go A)))
```

(1) and (2) characterize the function gcd in a non-algorithmic manner, the only difference being that (1) does it in English and (2) does it in a first order language. However (3) expresses gcd as an algorithm, essentially the Euclidean Algorithm, in the programming language LISP. In general, a formalism in which algorithms are expressed is called a programming language. The first-order language used in (2) is ***not*** a programming language.

The task of transforming an algorithm represented in one language to a corresponding algorithm represented in another language is performed by a translator (compiler, assembler). Translator technology has been highly developed over the past twenty years and we shall not consider this problem here. It is, however, a very different and much more difficult problem to construct an algorithmic representation of a function *f* from its non-algorithmic definition and this is what we mean by automatic program writing (or synthesis). A program that can solve this problem must have available a large body of knowledge concerning the mathematical environment of the function *f*. It must know about the meaning of the mathematical symbols occurring in the definition of *f*, about the target programming language, and it must be able to reason about it and to assemble the functional units of the target programming languag properly to an algorithm that represents *f*.

The theoretical foundations for automatic program writing go back to S. Kleene [16], who did the original work relating recursive function theory to intuitionist logic. However, R. Waldinger [39] was the first one who applied Kleene's method for automatic program writing. His approach is as follows: An input relation $Dom(x_1,\cdots,x_n)$ defines the domain of the function f as the collection of all n-tuples $(x_1,\cdots,x_n)$ for which $Dom(x_1,\cdots,x_n)$ is true. Also, a statement of the form

$$w: (x_1)\cdots(x_n)[Dom(x_1,\cdots,x_n) \Rightarrow (Ey)R(x_1,\cdots,x_n,y)]$$

asserts that for each $(x_1,\cdots,x_n)$ in the domain of the function there exists an element y (that is the function value) such that the relation $R(x_1,\cdots,x_n,y)$ holds. The knowledge concerning the properties of the mathematical symbols occurring in $Dom(x_1,\cdots,x_n)$ and $R(x_1,\cdots,x_n,y)$, which is relevant for the problem at hand, is represented as a set Γ of statements (hypotheses). Now a refutation of $\Gamma \cup \{\sim w\}$ is generated which is required to be constructive in a very precise sense. Roughly speaking this means that in the course of generating the refutation under certain conditions only such symbols that are defined in the target programming language must be used. When the refutation is generated, certain substitutions are made in each resolution step which, when properly composed, will define an instance of the variable y. This instance is the function value for $x_1,\cdots,x_n$. Thus, when the refutation is found, then a post processor will collect the relevant substitutions and assemble them properly as assignment statements. Since the substitutions depend on the argument data $(x_1,\cdots,x_n)$, therefore the generated program will, in general, also contain conditional GO TO statements. The main feature of this post processor is that it does not involve any search or decision making. It just assembles certain prefabricated program pieces together in a way that is directly controlled by the refutation. It works very much like a syntax directed compiler and might therefore be called a ***refutation directed compiler***.

This initial approach to automatic program writing is quite limited. Only programs consisting of a sequence of assignment statements, conditional GO TO statements and RETURN statements can be produced. It is clear that this is not sufficient machinery for useful applications.

It turns out that recursion and loops are closely connected to the principle of mathematical induction. The induction axiom

$$\begin{aligned} &P(0) \\ &\wedge (z)(P(z) \Rightarrow P(z+1)) \\ &\Rightarrow (x)P(x) \end{aligned}$$

says that if a property P is true for 0 and P carries over from n to n + 1 then P is true for all numbers 0,1,2,···. We let (x)P(x) stand for (x)(Ey)R(x,y), which means that for every x there is a certain y (namely the function value $f(x)$ of x) such that R(x,y) is true. The objective is now to prove (x)P(x) and to produce as a by-product the function f.

A more detailed analysis reveals that with P(0) [= (Ey)R(0,y)] a constant c is associated, namely that value c of y for which R(0,c) is true, that with (z)(P(z) ⇒ P(z + 1)) a function g of two arguments is associated, and that f is defined in terms of c and g as follows:

$$\begin{aligned} f(0) &= c \\ f(x+1) &= g(x,f(x)). \end{aligned}$$

The fact that f is expressed in terms of c and g directly corresponds to the fact that (x)P(x) is a logical consequence of P(0), (z)(P(z) ⇒ P(z + 1)), and the induction axiom.

Now the automatic program writer can be equipped with the capacity to produce programs containing recursion and loops as follows: First P(0) is proved and c is generated as a by-product. Then (z)(P(z) ⇒ P(z + 1)) is proved and g is generated. The program writer has a program scheme available that represents the above function definition of f in terms of c and g. This program scheme may be recursive or in loop form. Thus, the problem of finding f is in effect broken up into two subproblems, namely to find c and to find g. Then the induction principle is used to compose c and g properly to the function f which we wanted to construct originally.

Slightly different approaches to automatic program writing have been proposed by C. Green [11] and by Lee, Chang and Waldinger [20]. A very readable paper on this subject has been written by Z. Manna and Waldinger [27]. The same techniques can also be used for program debugging and program simulation [11]. It is conceivable that automatic program writing can be developed to a technology similar to the translator technology. However, it seems to the author that the theorem proving approach above is insufficient and that new ideas must come in. For example, in the field of proving the correctness of programs various techniques have been developed, most of which are based on a simple minded but ingenious idea of R. Floyd [9]. Possibly a combination of these and other ideas will give a breakthrough.

V. CONCLUDING REMARKS

The axiomatic approach in Artificial Intelligence looks at every problem as a theorem proving problem. This unified approach is theoretically simple, but practically weak The main reason for this may be that there is only one basic and very general inference machinery available, namely resolution in the predicate calculus (with or without equality). The goal to handle everything by one basic method seems to be too ambitious. A system which would have a variety of different, normally incomplete, inference mechanisms available, using one or another depending on the problem area would probably be eminently more powerful. The problem is that special inference systems for particular areas have not been developed yet except in a very rudimentary form such as special handling of the equality symbol [36,14] of the set theoretical symbols $\subset$ and $\in$ [38], and of $<$ in analysis [3]. Also, it is not clear at all how to organize such a complex system so that the computer can choose the appropriate inference system in which to operate.

In research projects, in particular in those involving large programs, the generality and extendability of certain techniques and methods that are being used are in general more important than the immediate results. For example, suppose a program knows about a very restricted world of simple geometric bodies, like cubes, cylinders, and pyramids, and can understand commands in English to move an object and to put it on top of another body, etc... If the techniques and methods used by the program to solve its problems depend to a great degree on this special environment, then this program is not of much value since nobody is interested in a system that can understand sentences concerning only this restricted world of cubes and can respond intelligently. What we are interested in are the principles and techniques used in the program and how far they would carry in a much more complex environment, given for example by a collection of ships of different types, their positions and functions, a distribution of sonar bouys and the task to hunt down an enemy submarine hidden somewhere among the ships.

It is the potential of the principles and methods for wider applications which is most important. From this viewpoint it appears that within the theorem proving approach to artificial intelligence certain research directions still have a tremendous potential and therefore research in these areas should be carried on. However, it is becoming more and more apparent that the effort to obtain complete strategies by refining existing strategies - that is, by imposing an extra condition on the parent clauses that must be satisfied before an inference is computed from them – are of diminishing return. Also, completeness is a theoretical concept similar to computability. The implementation of any complete strategy on a computer necessarily becomes incomplete because computer time and storage space is ***bounded***, not only ***finite***. Therefore, in practice, if a program solves say 90% of all problems of a certain representative set of benchmark problems in a reasonable time, then this fact is more important than the completeness of the underlying strategy.

The following directions of research seem to be promising:

1. In automatic program writing the theorem prover operates in an environment determined by a programming language L that is the language in which the program is to be written. L can be considered as being characterized by its interpreter which actually executes the programs. The interpreter can be regarded as a very powerful machine which is not aware of what it is doing. On the other hand, the theorem prover is aware of what it is doing, it can reason, but it is weak in its computational power. Thus, we have the interpreter which is powerful but stupid, and the theorem prover which is intelligent but weak. It would indeed be very desirable to equip the theorem prover with some of the power of the interpreter. For example, the interpreter immediately says that 3 = 1 is false while the theorem prover must painfully construct an explicit refutation of it. Thus, the general idea is to incorporate computing power of the interpreter into the theorem prover. This is in the spirit of Bledsoe when he writes [3]: "A source of power to a mathematician is his ability to leave to calculation those things that can be calculated and thereby freeing his mind for the harder task of finding inferences."

2. The axiomatic approach considers theorem proving as ***basic*** and reduces, at least partially, other problems to theorem proving problems. In this way, all techniques developed in this one field carry over automatically to other areas. Since theorem proving has been studied extensively in recent years and various techniques are available, this approach seems natural. Yet, theorem proving techniques do not directly come to grips with the basic issue of decision making in a situation where the available information does not suffice to determine uniquely the action to be taken. Theorem proving programs investigate every situation that can be reached from the present one using a certain (complete) strategy. Thus every path which is possible under the strategy being used is followed until the first successful one is located which represents a solution, that is, a proof of the theorem. Therefore making a correct decision in a certain situation may be equivalent to solving the whole problem starting from the present situation.

   As one of the applications of theorem proving we described automatic program writing. Here certain primitive function elements are assembled into a program in such a way that it satisfies the initial specifications. However, if one can develop another more direct method of composing the program from its elementary constituents then one may regard this techniqu of automatic program synthesis as ***basic*** and apply it to theorem proving as follows: Consider a rule of inference as a primitive function f_p, taking s me statements as arguments and yielding a statement as its value. f_p serve as a building block for constructing composed functions. Then a proof is an expression represented by a composed function applied to its argumen s. The problem of finding a proof can now be reformulated as follows: Find a list $(a_1,\cdots,a_n)$ of statements each of which being in the given set S, and a function f of n arguments, f being expressed in terms of the primitive functions f_p that represent rules of inference, such that $f(a_1,\cdots,a_n)$ is the statement that is to be proved. If we use the p:oof by contradiction method then $f(a_1,\cdots,a_n)$ will be the explicit contraction □. Thus, if one can develop direct techniques for solving the problem of constructing functions that satisfy certain conditions, then it appears natural to consider this problem as ***basic***. Progress in the solution of this problem would automatically carry over to theorem proving and other fields of artificial intelligence.

3. Finally, the implementation of a system capable of proving, reason ng, and making decisions on a computer is a major research project in itself. One problem is to study and optimize certain functions representing, so to speak, the basic arithmetic of the system (i.e., substitution, most general unification, factoring, subsumption). The major problem, however, is the overall organization of such a system, the incorporation of powerful heuristics, and the implementation of a device for handling any heuristics, that are suggested by a user, in a systematic manner. By a heuristic, we mean a computational rule which in many cases helps to compute the desired result but in other cases it does not. Typically, a heuristic utilizes a simplifying assumption which is true in many cases but not in general. Here it must be emphasized that the contracted resolution strategy [15] simplifies the organization problem tremendously, since it is an input strategy, which means, that at least one parent clause of a resolvent is always a clause in the originally given set S. This fact appears to be the major advantage of this strategy which, on the other hand, has the disadvantage that each single inference step is quite complex.

   It seems almost impossible to anticipate all of the conceivable strategies, heuristics, and special assumptions that a user might want to employ, and to provide appropriate routines for them. The most natural thing to do is to provide the basic theorem proving machinery in the form of a programming language and to let the user program his problems. Such a language should be gradually developed and implemented on the computer at the same time. Initially, it should contain the following components: one or more basic inference operators, the basic building blocks for defining search strategies and heuristics, and a flexible linkage system that allows for dynamic binding of variables to data, function definitions, and assumptions. The structure, feasibility, and implementation of a theorem proving language appears to be a worthwhile future project. A study of application problems will naturally go along with it.

Sometimes the argument can be heard that the computer hardware has not yet developed far enough to allow for the building of an intelligent machine. The author disagrees with this viewpoint. It is true that for most artificial intelligence problems much more memory is needed. However, even if we had all the memory in the world, we still would not know how to organize theorem proving programs so that the computer could draw intelligent conclusions in a complex environment of assumptions within a reasonable time. Our knowledge about the theoretical principles upon which a system capable of automatic reasoning must be built is still scanty. Thus, it appears that the time when we will have intelligent machines around us doing much of the work now being done by humans is still far away.

REFERENCES

1. Anderson, R., *Completeness Results for E-Resolution*. **Proc. AFIPS Spring Joint Computer Conference**, 36, 1970, pp. 653-656.

2. __________ and W. W. Bledsoe, *A Linear Format for Resolution With Merging and a New Technique for Establishing Completeness*. **J. ACM 17** (1970), pp. 525-584.

3. Bledsoe, W. W., R. S. Boyer and W. H. Henneman, *Computer Proofs of Limit Theorems*. **Artificial Intelligence 3**, (1972), pp. 27-60.

4. Chang, C. L., *Renamable Paramodulation for Automatic Theorem Proving With Equality*. **Artificial Intelligence 1**, (1970), pp. 247-256.

5. __________, and J. R. Slagle, *Completeness of Linear Refutation for Theories With Equality*. **J. ACM 18**, (1971), pp. 126-136.

6. Church, A., *A Note on the Entscheidungs Problem*. **Journal of Symbolic Logic**, Vol 1, pp. 40-41.

7. Davis, M., *Eliminating the Irrelevant from Mechanical Proofs*. **Proc. Sympos. Appl. Math.**, Vol. 18, Amer. Math. Soc., Providence, R.I., 1963, pp. 15-30.

8. __________, and H. A. Putnam, *A Computing Procedure for Quantification Theory*. **J. Acm 7** (1960), pp. 201-215.

9. Floyd, R. W., *Assigning Meanings to Programs*. **Proc. Symp. Appl. Math.**, Vol 19, Amer. Math. Soc., Providence, R.I., 1967, pp. 19-32.

10. Green, C., *Theorem Proving by Resolution as a Basis for Question Answering Systems*. **Machine Intelligence 4** (eds Meltzer, B. and Michie, D.), American Elsevier, New York, N.Y., 1969, pp. 183-205.

11. __________, *Applications of Theorem Proving to Problem Solving*. **Proc. of the Intern. Joint Conf. on Artificial Intelligence**, Washington, D. C., 1969, pp. 219-239.

12. __________, and B. Raphael, *The Use of Theorem Proving Techniques in Question Answering Systems*. **Proc. 23rd National Conf. ACM**, Brandon Systems Press, Princeton, N. J., 1968, pp. 169-181.

13. Huber, H. G., and A. H. Morris, Jr., *"Resolution in First Order Theories,"* NWL Technical Report No. TR-2405, 1970, Naval Weapons Laboratory, Dahlgren, Virginia.

14. __________, __________, *"Primary Paramodulation,"* NWL Technical Report No. TR-2552, 1971, Naval Weapons Laboratory, Dahlgren, Virginia.

15. __________, __________, *"Contracted Resolution,"* NWL Technical Report No. TR-2655, 1972, Naval Weapons Laboratory, Dahlgren, Virginia.

16. Kleene, S. C., *On the Interpretation of Intuitionistic Number Theory*. **Journal of Symbolic Logic,** Vol 10, pp. 109-112.

17. Kowalski, R., *The Case for Using Equality Axioms in Automatic Demonstration.* **Proc. IRIA Symposium on Automatic Demonstration,** Versailles, France, 1968, Springer Verlag, 1970, pp. 112-127.

18. __________, and P. J. Hayes, *Semantic Trees in Automatic Theorem Proving.* **Machine Intelligence 3** (ed. Michie, D.), American Elsevier, New York. N. Y., 1968, pp.95-112.

19. __________, and D. Kuehner, *Linear Resolution With Selection Function*, **Artificial Intelligence 2** (1971), pp. 227-260.

20. Lee, R. C. T., C. L. Chang, and R. J. Waldinger, *An Improved Program Synthesizing Algorithm and Its Correctness*, Memo of the Heuristics Laboratory, National Institutes of Health, Bethesda, Md. 20014.

21. Levy, D. N. L., *Computer Chess – A Case Study on the CDC 6600*, **Machine Intelligence 6** (eds. Meltzer, B. and Michie, D.), American Elsevier, New York, N. Y., 1971, pp. 151-163.

22. Loveland, D. W., *A Simplified Format for the Model Elimination Theorem Proving Procedure*. **J. ACM 16** (1969), pp. 349-363.

23. Loveland, D. W., *A Unifying View of Some Linear Herbrand Procedures*, **J. ACM 19** (1972), pp. 366-384.

24. Luckham, D. C., *Some Tree-Paring Strategies for Theorem Proving*. **Machine Intelligence 3**, (ed. Michie, D.), American Elsevier, New York, N. Y., 1968, pp. 95-112.

25. _________ *Refinement Theorems in Resolution Theory*, **Symposium on Automatic Demonstration**, Versailles, France, 1968, Springer Verlag 1970, pp. 147-162

26. _________, and N. J. Nilsson, *Extracting Information From Resolution Trees*, **Artificial Intelligence 2** (1971), pp. 27-54.

27. Manna, Z., and R. J. Waldinger, *Towards Automatic Program Synthesis*, Report No. CS174 Computer Science Department, Stanford University.

28. Mendelson, E., **Introduction to Mathematical Logic**, Van Nostrand, Princeton, N. Y., 1964.

29. Nilsson, N. J., *A Mobil Automaton, An Application of Artificial Intelligence Techniques*, **Proc. of the Intern. Joint Conf. on Artificial Intelligence**, Washington, D. C., 1969, pp. 509-520.

30. _________, **Problem-Solving Methods in Artificial Intelligence**, McGraw-Hill Book Company, New York, 1971.

31. Reiter, R., *Two Results on Ordering for Resolution With Merging and Linear Format*. **J. ACM 18** (1971), pp. 630-646.

32. Robinson, J. A., *A Machine-Oriented Logic Based on the Resolution Principle*, **J. ACM 12** (1965), pp. 23-41.

33. _________, *A Review of Automatic Theorem Proving*. **Annual Symposia in Appl. Math. XIX**, Amer. Math. Soc., Providence, R. I., 1967, pp. 1-18

34. _________, *The Generalized Resolution Principle*. **Machine Intelligence 3** (ed. Michie, D.), American Elsevier, New York, N Y., 1968, pp. 77-93.

35. _________, *Computational Logic. The Unification Computation*. **Machine Intelligence 6** (eds. Meltzer, B., and Michie, D.), American Elsevier, New York, N. Y., 1971, pp. 63-72.

36. Robinson, G. and L. Wos, *Paramodulation and Theorem Proving in First Order Theories With Equality*. **Machine Intelligence 4** (eds. Meltzer, B. and Michie, D.), American Elsevier, New York, N. Y., 1969, pp. 135-150.

37. Slagle, J. R., *Automatic Theorem Proving With Renamable and Semantic Resolution*. **J. ACM 14** (1967), pp. 687-697.

38. __________, *Automatic Theorem Proving With Built-In Theories Including Equality, Partial Ordering, and Sets*. **J. ACM 19** (1972), pp. 120-135.

39. Waldinger, R. J., *Constructing Programs Automatically Using Theorem Proving*. Computer Science Department, Carnegie-Mellon University, Pittsburg, Pennsylvania, 1969, Ph.D. Thesis.

40. __________, and R. C. T. Lee, *PROW: A Step Toward Automatic Program Writing*. **Proc. of the Joint Conf. on Artificial Intelligence**, Washington, D. C., 1969, pp. 241-252.

41. Wos, L. T., D. F. Carson, and G. A. Robinson, *Efficiency and Completeness of the Set of Support Strategy in Theorem Proving*. **J. ACM 12** (1965), pp. 536-541.

42. __________, and G. A. Robinson, *Paramodulation and Set of Support*. **Proc. IRIA Symposium on Automatic Demonstration**, Versailles, France, 1968, Springer-Verlag, 1970, pp. 276-310.

43. **Procedings of the International Joint Conference on Artificial Intelligence**, Washington, D. C., 1969 (eds. Walker, D. E., and Norton, L. M.).

44. **Proceedings of an ACM Conference on Proving Assertions About Programs**. New Mexico State University, Las Cruces, New Mexico, 1972, ACM, New York, N. Y.