APPLICATION OF THEOREM PROVING TO PROBLEM SOLVING*†

Cordell Green
Stanford Research Institute
Menlo Park, California

Abstract

This paper shows how an extension of the resolution proof procedure can be used to construct problem solutions. The extended proof procedure can solve problems involving state transformations. The paper explores several alternate problem representations and provides a discussion of solutions to sample problems including the "Monkey and Bananas" puzzle and the "Tower of Hanoi" puzzle. The paper exhibits solutions to these problems obtained by QA3, a computer program based on these theorem-proving methods. In addition, the paper shows how QA3 can write simple computer programs and can solve practical problems for a simple robot.

Key Words: Theorem proving, resolution, problem solving, automatic programming, program writing, robots, state transformations, question answering.

Automatic theorem proving by the resolution proof procedure[1]§ represents perhaps the most powerful known method for automatically determining the validity of a statement of first-order logic. In an earlier paper Green and Raphael[2] illustrated how an extended resolution procedure can be used as a question answerer--e.g., if the statement $(\exists x)P(x)$ can be shown to follow from a set of axioms by the resolution proof procedure, then the extended proof procedure will find or construct an x that satisfies P(x). This earlier paper (1) showed how one can axiomatize simple question-answering subjects, (2) described a question-answering program called QA2 based on this procedure, and (3) presented examples of simple question-answering dialogues with QA2. In a more recent paper[3] the author (1) presents the answer construction method in detail and proves its correctness, (2) describes the latest version of the program, QA3, and (3) introduces state-transformation methods into the constructive proof formalism. In addition to the question-answering applications illustrated in these earlier papers, QA3 has been used as an SRI robot[4] problem solver and as an automatic program writer. The purpose of this paper is twofold: (1) to explore the question of predicate calculus representation for state-transformation problems in general, and (2) to elaborate upon robot and program-writing applications of this approach and the mechanisms underlying them.

Exactly how one can use logic and theorem proving for problem solving requires careful thought on the part of the user. Judging from my experience, and that of others using QA2 and QA3, one of the first difficulties encountered is the representation of problems, especially state-transformation problems, by statements in formal logic. Interest has been shown in seeing several detailed examples that illustrate alternate methods of axiomatizing such problems--i.e., techniques for "programming" in first-order logic. This paper provides detailed examples of various methods of representation. After presenting methods in Secs. I and II, a solution to the classic "Monkey and Bananas" problem is provided in Sec. III. Next, Sec. IV compares several alternate representations for the "Tower of Hanoi puzzle. Two applications, robot problem solving and automatic programming, are discussed in Secs. V and VI, respectively.

I. An Introduction to State-Transformation Methods

The concepts of states and state transformations have of course been in existence for a long time, and the usefulness of these concepts for problem solving is well known. The purpose of this paper is not to discuss states and state transformations as such, but instead to show how these concepts can be used by an automatic resolution theorem prover. In practice, the employment of these methods has greatly extended the problem-solving capacity of QA2 and QA3. McCarthy and Hayes[5] present a relevant discussion of philosophical problems involved in attempting such formalizations.

First we will present a simple example. We begin by considering how a particular universe of discourse might be described in logic.

Facts describing the universe of discourse are expressed in the form of statements of mathematical logic. Questions or problems are stated as conjectures to be proved. If a theorem is proved, then the nature of our extended theorem prover is such that the proof is "constructive"--i.e., if the theorem asserts the existence of an object then the proof finds or constructs such an object.

At any given moment the universe under consideration may be said to be in a given state.

* This research is a part of Project Defender and was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Rome Air Development Center under Contracts AF 30(602)-4147 and F30602-69-C-0056.

† This preprint is a preliminary version and is subject to modification prior to publication.

§ References are listed at the end of this paper.

We will represent a particular state by a subscripted s--e.g., s_{17}. The letter s, with no subscript, will be a variable, ranging over states. A state is described by means of predicates. For example, if the predicate $AT(object_1, b, s_1)$ is true, then in state s_1 the object, $object_1$, is at position b. Let this predicate be axiom A1:

A1. $AT(object_1, b, s_1)$.

The question "Where is $object_1$ in state s_1?" can be expressed in logic as the theorem $(\exists x)AT(object_1, x, s_1)$. The answer found by using system QA3 to prove this theorem is "yes, x = b."

Changes in states are brought about by performing actions and sequences of actions. An action can be represented by an action function that maps states into new states (achieved by executing the action). An axiom describing the effect of an action is typically of the form

$$(\forall s)[P(s) \supset Q(f(s))]$$

where s is a state variable
- P is a predicate describing a state
- f is an action function (corresponding to some action) that maps a state into a new state (achieved by executing the action)
- Q is a predicate describing the new state.

(Entities such as P and f are termed "situational fluents" by McCarthy.)

As an example, consider an axiom describing the fact that $object_1$ can be pushed from point b to point c. The axiom is

A2. $(\forall s)[AT(object_1, b, s) \supset$
$AT(object_1, c, push(object_1, b, c, s))]$.

The function $push(object_1, b, c, s)$ corresponds to the action of pushing $object_1$ from b to c. (Assume, for example, that a robot is the executor of these actions.)

Now consider the question, "Does there exist a sequence of actions such that $object_1$ is at point c?" Equivalently, one may ask, "Does there exist a state, possibly resulting from applying action functions to an initial state s_1, such that $object_1$ is at point c?" This question, in logic, is $(\exists s)AT(object_1, c, s)$, and the answer, provided by the theorem-proving program applied to axioms A1 and A2, is "yes, $s = push(object_1, b, c, s_1)$."

Suppose a third axiom indicates that $object_1$ can be pushed from c to d:

A3. $(\forall s)[AT(object_1, c, s) \supset$
$AT(object_1, d, push(object_1, c, d, s))]$.

Together, these three axioms imply that starting in state s_1, $object_1$ can be pushed from b to c and then from c to d. This sequence of actions (a program for our robot) can be expressed by the composition of the two push functions, $push(object_1, c, d, push(object_1, b, c, s_1))$. The normal order of function evaluation, from the innermost function to the outermost, gives the correct sequence in which to perform the actions.

To find this solution to the problem of getting $object_1$ to position d, the following conjecture is posed to the theorem prover: "Does there exist a state such that $object_1$ is at position d?" or, stated in logic, $(\exists s)AT(object_1, d, s)$. The answer returned is "yes, $s = push(object_1, c, d, push(object_1, b, c, s_1))$."

The proof by resolution, given below, demonstrates how the desired answer is formed as a composition of action functions, thus describing a sequence of necessary actions. The mechanism for finding this answer is a special literal,* the "answer literal." This method of finding an answer is explained in detail in Ref. 3. For our purposes here, we will just show how it works by example. In the proof below, each answer literal is displayed beneath the clause containing it. At each step in the proof the answer literal will contain the current value of the object being constructed by the theorem prover. In this example the object being constructed is the sequence of actions s. So initially the answer literal ANSWER(s) is added to the clause representing the negation of the question. (One can interpret this clause, Clause 1, as "either $object_1$ is not at d in state s, or s is an answer.") The state variable s, inside the answer literal, is the "place holder" where the solution sequence is constructed. The construction process in this proof consists of successive instantiations of s. An instantiation of s can occur whenever a literal containing s is instantiated in the creation of a resolvent. Each instantiation* of s fills in a new action or an argument of an action function. In general, a particular inference step in the proof (either by factoring* or resolving*) need not necessarily further instantiate s. For example, the step might be an inference that verifies that some particular property holds for the current answer at that step in the proof. The final step in the proof yields Clause 7, "an answer is $push(object_1, c, d, push(object_1, b, c, s_1))$," which terminates the proof.

* We assume the reader is familiar with the vocabulary of the field of theorem proving by resolution as described in Refs. 1, 7, and 8.

Proof

1. $\sim AT(object_1, d, s)$ — Negation of theorem

 ANSWER(s)

2. $\sim AT(object_1, c, s) \lor AT(object_1, d, push(object_1, c, d, s))$ — Axiom A3

3. $\sim AT(object_1, c, s)$ — Resolve 1,2

 ANSWER(push(object$_1$,c,d,s))

4. $\sim AT(object_1, b, s) \lor AT(object_1, c, push(object_1, b, c, s))$ — Axiom A2

5. $\sim AT(object_1, b, s)$ — Resolve 3,4

 ANSWER(push(object$_1$,c,d, push(object$_1$,b,c,s)))

6. $AT(object_1, b, s_1)$ — Axiom A1

7. Contradiction — Resolve 5,6

 ANSWER(push(object$_1$,c,d, push(object$_1$,b,c,s$_1$)))

For the particular proof exhibited here, the order of generating the solution sequence during the search for the proof happens to be the same order in which the printout of the proof indicates s is instantiated. This order consists of working backward from the goal by filling in the last action, then the next-to-last action, etc. In general, the order in which the solution sequence is generated depends upon the proof strategy, since the proof strategy determines the order in which clauses are resolved or factored. The proof that this method always produces correct answers, given in Ref. 4, shows that the answers are correct regardless of the proof strategy used.

II. Refinements of the Method

The purpose of this section is to discuss variations of the formulation presented in the previous section and to show how other considerations such as time and conditional operations can be brought into the formalism. The reader who is interested in applications rather than additional material on representation may omit Secs. II, III, and IV, and read Secs. V and VI.

A. An Alternate Formulation

The first subject we shall discuss is an alternate to the previously given formulation. We shall refer to the original, presented in Sec. I, as formulation I, and this alternate as formulation II. Formulation II corresponds to a system-theoretic notion of state transformations. The state transformation function for a system gives the mapping of an action and a state into a new state. Let f represent the state transformation function, whose arguments are an action and a state and whose value is the new state obtained by applying the action to the state. Let $\{a_i\}$ be the actions, and nil be the null action. Let g be a function that maps two actions into a single composite action whose effect is the same as that of the argument actions applied sequentially. For example, axioms of the following form would partially define the state transformation function f:

B1. $(\forall s)[P(s) \supset Q(f(a_1, s))]$

B2. $(\forall s)[f(nil, s) = s]$

B3. $(\forall s, a_i, a_j)[f(a_j, f(a_i, s)) = f(g(a_i, a_j), s)]$.

The predicates P and Q represent descriptors of states. Axiom B1 describes the result of an action a_1 applied to the class of states that are equivalent in that they all have the property P(s). The resulting states are thus equivalent in that they have property Q(s). Axiom B2 indicates that the null action has no effect. The equation in B3 says that the effect of the composite action sequence $g(a_i, a_j)$ is the same as that of actions a_i and a_j applied sequentially. The question posed in this formulation can include an initial state--e.g., a question might be $(\exists x)Q(f(x, s_0))$, meaning "Does there exist a sequence of actions x that maps state s_0 into a state satisfying the predicate Q?" Observe that we are not insisting on finding a particular sequence of actions, but any sequence that leads us to a satisfactory state within the target class of states.

This representation is more complex, but has the advantage over the previous representation that both the starting state of a transformation and the sequence of actions are explicitly given as the arguments of the state-transformation function. Thus, one can quantify over, or specify in particular, either the starting state or the sequence, or both.

Next we shall show how other considerations can be brought into a state-transformation formalism. Both the original formulation (I) and the alternate (II) will be used as needed.

B. No Change of State

This kind of statement represents an implication that holds for a fixed state. An axiom typical of this class might describe the relationship between movable objects; e.g., if x is to the left of y and y is to the left of z, then x is to the left of z.

$$(\forall x, y, z, s)[LEFT(x, y, s) \land LEFT(y, z, s) \supset LEFT(x, z, s)]$$

C. Time

Time can be a function of a state, to express the timing of actions and states. For example, if the function time(s) gives the time of an instantaneous state, in the axiom

$$(\forall s)[P(s) \supset [Q(f(s)) \wedge \text{EQUAL}(\text{difference}(\text{time}(f(s)),\text{time}(s)),\tau)]],$$

where P(s) describes the initial state and Q(s) describes the final state, the state transformation takes τ seconds to complete.

D. State-Independent Truths

An example is

$$(\forall x,y,z)[\text{EQUAL}(\text{plus}(x,17),z) \supset \text{EQUAL}(\text{difference}(z,x),17)]$$

illustrating how functions and predicates are explicitly made state independent by not taking states as arguments.

E. Descriptors of Transformations

A descriptor or modifier of an action may be added in the form of a predicate that takes as an argument the state transformation that is to be described. For example,

WISHED-FOR(f(action,state),person)

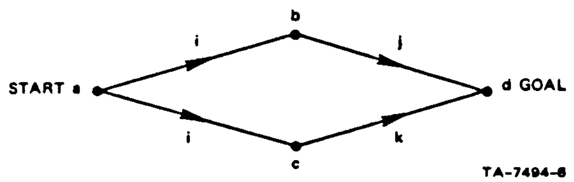
might indicate a wished-for occurrence of an action;

LOCATION(f(action,state),place)

indicates that an action occurred at a certain place.

F. Disjunctive Answers

Consider a case in which an action results in one of two possibilities. As an example, consider an automaton that is to move from a to d.



The above figure shows that action i leads to either b or c from a. The function f is single-valued but we don't know its value. The goal d can be reached from b by action j, or from c by action k. In the formalization given below it is possible to prove that the goal is reachable although a correct sequence of actions necessary to reach the goal is not generated. Instead, the answer produced is a disjunction of two sequences--$j(i(s_0))$ or $k(i(s_0))$.

We use formulation I. Axiom M1 specifies the starting state s_0 and starting position a. Axioms M2, M3, and M4 specify positions resulting from the allowed moves.

M1. $AT(a,s_0)$

M2. $(\forall s)[AT(a,s) \supset AT(b,i(s)) \vee AT(c,i(s))]$

M3. $(\forall s)[AT(b,s) \supset AT(d,j(s))]$

M4. $(\forall s)[AT(c,s) \supset AT(d,k(s))]$

To find if the goal d is reachable, we ask the following question:

Question: $(\exists s)AT(d,s)$

to which an answer is:

Answer: Yes, $s = j(i(s_0))$ or $s = k(i(s_0))$.

The proof is:

Proof

1.	$\sim AT(d,s)$	Negation of theorem
	$\underline{\text{ANSWER}}(s)$	
2.	$\sim AT(b,s) \vee AT(d,j(s))$	Axiom M3
3.	$\sim AT(b,s)$	From 1,2
	$\underline{\text{ANSWER}}(j(s))$	
4.	$\sim AT(c,s) \vee AT(d,k(s))$	Axiom M4
5.	$\sim AT(c,s)$	From 1,4
	$\underline{\text{ANSWER}}(k(s))$	
6.	$\sim AT(a,s) \vee AT(b,i(s)) \vee AT(c,i(s))$	Axiom M2
7.	$\sim AT(a,s) \vee AT(b,i(s))$	From 5,6
	$\underline{\text{ANSWER}}(k(i(s))$	
8.	$\sim AT(a,s)$	From 3,7
	$\underline{\text{ANSWER}}(j(i(s))) \vee \underline{\text{ANSWER}}(k(i(s))$	
9.	$AT(a,s_0)$	Axiom M1
10.	Contradiction	From 8,9
	$\underline{\text{ANSWER}}(j(i(s_0))) \vee \underline{\text{ANSWER}}(k(i(s_0)))$	

Observe that clause 8 has two answers, one coming from clause 3 corresponding to the action k and one from clause 7 corresponding to the action j. This shows how an "or" answer can arise.

G. Answers with Conditionals

A conditional operation such as "if p then q else r" allows a program to branch to either operation q or r depending upon the outcome of the test condition p. By allowing a conditional operation, a better solution to the above problem is made possible, namely, "beginning in state s_0 take action i; if at b take action j, otherwise take action k."

Consider the problem above that yields disjunctive answers. The information in the above problem formulation, axioms M1 through M4, plus additional information allows the creation of a program with a conditional and a test operation. The following additional information is needed, which we shall furnish in the form of axioms.

The first addition needed is a conditional operation, along with a description of what the operation does. Since our programs are in the form of functions, a conditional function is needed. One such possible function is the LISP conditional function "cond" which will be discussed in Sec. VI. However, another function, a simple "select" function is slightly easier to describe and will be used here. The function select(x,y,z,w) is defined to have the value z if x equals y and w otherwise.

M5. $(\forall x,y,z,w)[x = y \supset \text{select}(x,y,z,w) = z]$

M6. $(\forall x,y,z,w)[x \neq y \supset \text{select}(x,y,z,w) = w]$

The second addition needed is a test operation, along with a description of what it does. Since our programs are in the form of functions, a test function is needed. We shall use "atf", meaning "at-function." The function "atf" applied to a state yields the location in that state, e.g., $\text{atf}(s_0) = a$. The atf function is described by

M7. $(\forall x,s)[\text{AT}(x,s) \equiv (\text{atf}(s) = x)]$.

These axioms lead to the solution

$$s = \text{select}(\text{atf}(i(s_0)),b,j(i(s_0)),k(i(s_0))),$$

meaning "if at b after applying i to s_0, take action j otherwise action k."

Although the new axioms allow the conditional solution, just the addition of these axioms does not guarantee that disjunctive answers will not occur. To prevent the possibility of disjunctive answers, we simply tell the theorem prover not to accept any clauses having two answers that don't unify.

What may be a preferable problem formulation and solution can result from the use of the alternative state formulation, II, exemplified in axioms B1, B2, and B3 above. Recall that f(i,s) is the state transformation function that maps action i and state s into a new state, the function g(i,j) maps the action i and the action j into the sequence of the two actions--i then j. The interrelation of f and g is described by

B3. $(\forall i,j,s)[f(j,f(i,s)) = f(g(i,j),s)]$

Axioms M1 through M4 remain the same but axioms M5, M6, and M7 are replaced. The new select function is described by the two axioms:

M5′. $(\forall i,j,s,p,b)[\text{test}(p,s) = b \supset$
$f(\text{select}(p,b,i,j),s) = f(i,s)]$

M6′. $(\forall i,j,s,p,b)[\text{test}(p,s) \neq b \supset$
$f(\text{select}(p,b,i,j),s) = f(j,s)]$,

where the function test applies the test condition p (which will correspond to atf for this problem) to state s. The test condition atf is defined by

M7′. $(\forall x,s)[\text{AT}(x,s) \equiv \text{test}(\text{atf},s) = x]$.

The new solution is

$$s = f(g(i,\text{select}(\text{atf},b,j,k)),s_0).$$

Further discussion of program writing, including recursion, is given in Sec. VI.

Another method of forming conditional answers is possible. This involves inspecting an existence proof such as the one given in Sec. II-F above. First, such a proof is generated in which clauses having multiple answers are allowed. The conditional operation is constructed by observing the two literals which are resolved upon to generate the two-answer clause. For example, in the above proof clauses 3 and 7 resolve to yield 8. This step is repeated below, using the variable s′ in 3 to emphasize that s′ is different from s in 7.

Clause 3. $\sim\text{AT}(b,s')$

$\underline{\text{ANSWER}}(j(s'))$

Clause 7. $\sim\text{AT}(a,s) \lor \text{AT}(b,(i(s)))$

$\underline{\text{ANSWER}}(k(i(s)))$

Clause 8. $\sim\text{AT}(a,s)$

$\underline{\text{ANSWER}}(j(s)) \lor \underline{\text{ANSWER}}(k(i(s)))$

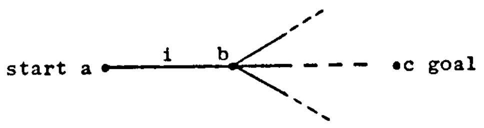
Clause 3 may be read as "if at b in state s′, the answer is to take action j when in state s′." Clause 7 may be read as "if not at b in state i(s) and if at a in state s, the answer is to take action k when in state i(s)." Observing that the resolution binds s′ to i(s) in Clause 8, one knows from Clauses 3 and 7 the test condition by which one decides which answer to choose in Clause 8, "if at a in state s the answer depends on i(s); if at b in i(s) take action j; otherwise take action k."

This discussion illustrates that the creation of a clause with two answer literals indicates that a conditional operation is needed to create a single conditional answer. This information provides a useful heuristic for the program-writing applications of QA3: When a clause having two answer literals is about to be generated, let the proof strategy call for the axioms that describe the conditional operation (such as M5 and M6). These axioms are then applied to create a single conditional answer.

Waldinger and Lee[6] have implemented a program-writing program PROW that also uses a resolution theorem prover to create constructive proofs, but by a different method than that of QA3. (The second method for creating conditionals by combining two answers is closely related to a technique used in PROW.) Information about (1) the target program operations, (2) the general relationship of the problem statement and axioms to the allowed target program operations including the test conditions, and (3) the syntax of the target language, is embedded in the PROW program. In QA3 this information is all in the axioms--such as axioms M5, M6, and M7.

H. Acquisition of Information

Another situation that arises in problem solving is one in which at the time the problem is stated and a solution is to be produced, there is insufficient information to completely specify a solution. More precisely, the solution cannot name every action and test condition in advance. As an example, consider a robot that is to move from a to c. The action i leads from a to b but no path to c is known, as illustrated below.



However, once point b is reached, more information can be acquired--for example, a guide to the area lives at b and will provide a path to point c if asked. Or perhaps once point b is reached, the robot might use its sensors to observe or discover paths to c.

To formalize this, assume that the action ask-path(b,c) will result in a proper path to c, when taken at b. For simplicity, assume that the name of the path is equal to the state resulting from asking the question. Using formulation II, one suitable set of axioms is:

N1. $AT(a,s_0) \wedge PATH(a,b,i)$

N2. $(\forall s,x,y,j)[AT(x,s) \wedge PATH(x,y,j) \supset AT(y,f(j,s))]$

N3. $(\forall s)[AT(b,s) \supset PATH(b,c,f(ask\text{-}path(b,c),s)) \wedge AT(b,f(ask\text{-}path(b,c),s))]$

where PATH(a,b,i) means that i is a path from a to b. The question $(\exists s)AT(c,s)$ results in the solution,

"yes, s = f(f(ask-path(b,c),f(i,s)),f(i,s))".

Axiom N3 illustrates an important aspect of this formalism for problem solving: If a condition (such as the robot's) is made state dependent, then we must specify how this condition changes when the state is changed. Thus in axiom N3 we must indicate that the robot's location is not changed by asking for a path. In a pure theorem-proving formalism, this means that if we want to know any condition in a given state, we must prove what that condition is. If a large number of state-dependent conditions need to be known at each state in a solution, then the theorem prover must prove what each condition is at each state in a conjectured solution. In such a case the theorem prover will take a long time to find the solution. McCarthy[5] refers to this problem as the frame problem, where the word "frame" refers to the frame of reference or the set of relevant conditions. Discussion of a method for easing this problem is presented in Sec. V.

I. Assignment Operations

An assignment operation is one that assigns a value to a variable. An example of an assignment is the statement $a \leftarrow h(a)$, meaning that the value of a is to be changed to the value of the function h(a). In our representation, we shall use an assignment function--i.e., assign(a,h(a)). Using Formulation II this function is described by the axiom

$$(\forall a,a_0,s)[VALUE(a,a_0,s) \supset VALUE(a,h(a_0),f(assign(a,h(a)),s))]$$

where the predicate $VALUE(a,a_0,s)$ means that variable a has value a_0 in state s.

III. An Example: The Monkey and The Bananas

To illustrate the methods described earlier, we present an axiomatization of McCarthy's "Monkey and Bananas" problem.

The monkey is faced with the problem of getting a bunch of bananas hanging from the ceiling just beyond his reach. To solve the problem, the monkey must push a box to an empty place under the bananas, climb on top of the box, and then reach them.

The constants are monkey, box, bananas, and under-bananas. The functions are reach, climb, and move, meaning the following:

reach(m,z,s)	The state resulting from the action of m reaching z, starting from state s
climb(m,b,s)	The state resulting from the action of m climbing b, starting from state s
move(m,b,u,s)	The state resulting from the action of m moving b to place u, starting from state s.

The predicates are:

MOVABLE(b)	b is movable
AT(m,u,s)	m is at place u in state s
ON(m,b,s)	m is on b in state s
HAS(m,z,s)	m has z in state s
CLIMBABLE(m,b,s)	m can climb b in state s
REACHABLE(m,b,s)	m can reach b in state s.

The axioms* are:

MB1. MOVABLE(box)

MB2. $AT(box,place_b,s_0)$

MB3. $(\forall x) \sim AT(x,under\text{-}bananas,s_0)$

MB4. $(\forall b,p_1,p_2,s)[[AT(b,p_1,s) \wedge MOVABLE(b) \wedge (\forall x) \sim AT(x,p_2,s)] \supset [AT(b,p_2,move(monkey,b,p_2,s)) \wedge AT(monkey,p_2,move(monkey,b,p_2,s))]]$

MB5. $(\forall s)CLIMBABLE(monkey,box,s)$

MB6. $(\forall m,p,b,s)[[AT(b,p,s) \wedge CLIMBABLE(m,b,s)] \supset [AT(b,p,climb(m,b,s)) \wedge ON(m,b,climb(m,b,s))]]$

MB7. $(\forall s)[[AT(box,under\text{-}bananas,s) \wedge ON(monkey,box,s)] \supset REACHABLE(monkey,bananas,s)]$

MB8. $(\forall m,z,s)[REACHABLE(m,z,s) \supset HAS(m,z,reach(m,z,s))]$.

The question is "Does there exist a state s (sequence of actions) in which the monkey has the bananas?"

QUESTION: $(\exists s)HAS(monkey,bananas,s)$.

The answer is yes,

$s = reach(monkey,bananas,climb(monkey, box,move(monkey,box,under\text{-}bananas,s_0)))$.

By executing this function, the monkey gets the bananas. The monkey must, of course, execute the functions in the usual order, starting with the innermost and working outward. Thus he first moves the box under the bananas, then climbs on the box, and then reaches the bananas.

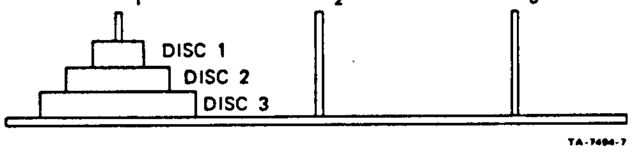
The printout of the proof is given in the appendix.

IV. Formalizations for the Tower of Hanoi Puzzle

The first applications of our question-answering programs were to "question-answering" examples. Commonly used question-answering examples have short proofs, and usually there are a few obvious formulations for a given subject area. (The major difficulty in question-answering problems usually is searching a large data base, rather than finding a long and difficult proof.) Typically any reasonable formulation works well. As one goes on to problems like the Tower of Hanoi puzzle, more effort is required to find a representation that is suitable for efficient problem solving.

This puzzle has proved to be an interesting study of representation. Several people using QA3 have set up axiom systems for the puzzle. Apparently, a "good" axiomatization--one leading to quick solutions--is not entirely obvious, since many axiomatizations did not result in solutions. In this section we will present and compare several alternate representations, including ones that lead to a solution.

There are three pegs--peg_1, peg_2, and peg_3. There are a number of discs each of whose diameter is different from that of all the other discs. Initially all discs are stacked on peg_1, in order of descending size. The three-disc version is illustrated below.



* The astute reader will notice that the axioms leave much to be desired. In keeping with the "toy problem" tradition we present an unrealistic axiomatization of this unrealistic problem. The problem's value lies in the fact that it is a reasonably interesting problem that may be familiar to the reader.

The object of the puzzle is to find a sequence of moves that will transfer all the discs from peg_1 to peg_3. The allowed moves consist of taking the top disc from any peg and placing it on another peg, but a disc can never be placed on top of a smaller disc.

In order to correctly specify the problem, any formalization must: (1) specify the positions of the discs for each state; (2) specify how actions change the position of the discs; and (3) specify the rules of the game, i.e., what is legal.

Let the predicate ON specify disc positions. In the simplest representation the predicate ON specifies the position of one disc--e.g., $ON(disc_1, peg_1, s)$ says that in state s $disc_1$ is on peg_1. This representation requires one predicate to specify the position of each disc. The relative position of each disc either must be specified by another statement, or else if two discs are on the same peg it must be implicitly understood that they are in the proper order. Perhaps the simplest extension is to allow the predicate another argument that specifies the position of the disc--i.e., $ON(disc_1, peg_1, position_2, s)$. Again, this requires many statements to specify a complete configuration.

Since various moves are constructing stacks of discs, and since stacks can be represented as lists, consider as an alternative representation a list to represent a stack of discs. Let the function $\ell(x,y)$ represent the list that has x as its first element (representing the top disc in the stack) and y as the rest of the list (representing the rest of the discs in the stack). This function ℓ corresponds to the "cons" function in LISP. Let nil be the empty list. The statement $ON(\ell(disc_1, \ell(disc_2, nil)), peg_1, s)$ asserts that the stack having top disc, $disc_1$, and second disc, $disc_2$, is on peg_1. This representation illustrates a useful technique in logic--namely, the use of functions as the construction (and selection) operators. This notion is consistent with the use of action functions as constructors of sequences.

Next, consider how to express possible changes in states. Perhaps the simplest idea is to say that a given state implies that certain moves are legal. One must then have other statements indicating the result of each move. This method is a bit lengthy. It is easier to express in one statement the fact that given some state, a new state is the result of a move. Thus one such move to a new state is described by $(\forall s)[ON(\ell(disc_1, nil), peg_1, s) \wedge ON(nil, peg_2, s) \wedge ON(\ell(disc_2, \ell(disc_3, nil)), peg_3, s) \supset ON(nil, peg_1, move(disc_1, peg_1, peg_2, s)) \wedge ON(\ell(disc_1, nil), peg_2, move(disc_1, peg_1, peg_2, s)) \wedge ON(\ell(disc_2, \ell(disc_3, nil)), peg_3, move(disc_1, peg_1, peg_2, s))]$.

With this method it is possible to enumerate all possible moves and configuration combinations. However, it is still easier to use variables to represent whole classes of states and moves. Thus $(\forall s,x,y,z,p_i,p_j,p_k,d)[ON(\ell(d,x), p_i, s) \wedge ON(y, p_j, s) \wedge ON(z, p_k, s) \supset ON(x, p_i, move(d, p_i, p_j, s)) \wedge ON(\ell(d,y), p_j, move(d, p_i, p_j, s)) \wedge ON(z, p_k, move(d, p_i, p_j, s))]$ specifies a whole class of moves. The problem here is that additional restrictions must be added so that illegal states cannot be part of a solution. In the previous formalism, one could let the axioms enumerate just the legal moves and states, thus preventing incorrect solutions.

The first method for adding restrictions is to have a predicate that restricts moves to just the legitimate states. Since the starting state is legal, one might think that only legal states can be reached. However, the resolution process (set-of-support strategy[7]) typically works backward from the goal state toward states that can reach the goal state--such states are sometimes called "forcing states." Thus illegal but forcing states can be reached by working backward from the goal state. This does not allow for incorrect solutions, since the only forcing states that can appear in the solution must be those reached from the starting state (which is a legal state). The restriction of moving only <u>to</u> new legal states thus prevents an error. But the search is unnecessarily large, since the theorem prover is considering illegal states that cannot lead to a solution. So a better solution is to eliminate these illegal forcing states by allowing moves only <u>from</u> legal states <u>to</u> legal states. This is perhaps the best specification, in a sense. Such an axiom is $(\forall s,x,y,z,p_i,p_j,p_k,d)[ON(\ell(d,x), p_i, s) \wedge ON(y, p_j, s) \wedge ON(z, p_k, s) \wedge LEGAL(\ell(d,x)) \wedge LEGAL(\ell(d,y)) \wedge DISTINCT(p_i, p_j, p_k) \supset ON(x, p_i, move(d, p_i, p_j, s)) \wedge ON(\ell(d,y), p_j, move(d, p_i, p_j, s)) \wedge ON(z, p_k, move(d, p_i, p_j, s))]$. The predicate LEGAL(x) is true if and only if the discs are listed in order of increasing size. (One can "cheat" and have a simpler axiom by omitting the predicate that requires that the state resulting from a move have a legal stack of discs. Since the set-of-support strategy forces the theorem prover to work backward starting from a legal final state, it will only consider legal states. However, one is then using an axiomatization that, by itself, is incorrect.) The additional LEGAL predicate is a typical example of how additional information in the axioms results in a quicker solution. The predicate $DISTINCT(p_i, p_j, p_k)$ means no two pegs are equal.

The clauses generated during the search that are concerned with illegal states are subsumed[8] by ~LEGAL predicates such as $(\forall s)\sim LEGAL(\ell(disc_2, (disc_1, x)))$. The stacks are formed by placing one new disc on top of a legal stack. If the new top disc is smaller than the old top disc then it is of course smaller than all the others on the stack. Thus the legal stack axioms need only to specify that the top disc is smaller than the second disc for a stack to be legal. This blocks the construction of incorrect stacks.

One complete axiomatization is as follows:

$$\begin{aligned}
\text{AX1. } & (\forall x,y,z,m,n,p_i,p_j,p_k)[\mathrm{ON}(\ell(d(m),x),p_i,s) \wedge \\
& \mathrm{ON}(y,p_j,s) \wedge \mathrm{ON}(z,p_k,s) \wedge \\
& \mathrm{DISTINCT}(p_i,p_j,p_k) \wedge \mathrm{LEGAL}(\ell(d(m),x)) \wedge \\
& \mathrm{LEGAL}(\ell(d(n),y)) \supset \\
& \mathrm{ON}(x,p_i,\mathrm{move}(d(m),p_i,p_j,s)) \wedge \\
& \mathrm{ON}(\ell(d(m),y),p_j,\mathrm{move}(d(m),p_i,p_j,s)) \wedge \\
& \mathrm{ON}(z,p_k,\mathrm{move}(d(m),p_i,p_j,s))]
\end{aligned}$$

$$\begin{aligned}
\text{AX2. } & (\forall m \quad x)[\mathrm{LEGAL}(\ell(d(m),\ell(d(n),x))) \equiv \\
& \mathrm{LESS}(m,n)] \wedge (\forall n)\mathrm{LEGAL}(\ell(d(n),\mathrm{nil})) \wedge \\
& \mathrm{LEGAL}(\mathrm{nil})
\end{aligned}$$

Instead of naming each disc, the disc number n is an argument of the function d(n) that represents the nth disc. This representation illustrates how the proof procedure can be shortened by solving frequent decidable subproblems with special available tools--namely, the LISP programming language. The theorem prover uses LISP (the "lessp" function) to evaluate the LESS(n,m) predicate--a very quick step. This mechanism has the effect of generating, wherever needed, such axioms as $\sim$LESS(3,2) or LESS(2,3) to resolve against or subsume literals in generated clauses. Similarly, LISP evaluates the DISTINCT predicate.

Note that the move axiom, AX1, breaks up into three clauses, each clause specifying the change in the task for one particular peg. The process of making one move requires nine binary resolutions, and two binary factorings of clauses.

Still other solutions are possible by using special term-matching capabilities in QA3 that extend the unification and subsumption algorithms to include list terms, set terms, and certain types of symmetries.

In another axiomatization, the complete configuration of the puzzle in a given state is specified by the predicate ON. ON(x,y,z,s) means that in state s, stack x in on peg_1, stack y is on peg_2, and stack z is on peg_3. Thus if the predicate $\mathrm{ON}(\ell(d_1,\ell(d_2,\mathrm{nil})),\mathrm{nil},\ell(d_3,\mathrm{nil}),s_k)$ holds, the stack $d_1 - d_2$ is on peg_1 and d_3 is on peg_3. The predicate LEGAL again indicates that a given stack of discs is allowed.

Two kinds of axioms are required--move axioms and legal stack axioms. One legal stack axiom is $\mathrm{LEGAL}(\ell(d_1,\ell(d_2,\mathrm{nil})))$. One move axiom is $(\forall d,x,y,z,s)[\mathrm{ON}(\ell(d,x),y,z,s) \wedge \mathrm{LEGAL}(\ell(d,x)) \wedge \mathrm{LEGAL}(\ell(d,y)) \supset \mathrm{ON}(x,\ell(d,y),z,\mathrm{move}(d,p_1,p_2,s))]$. This axiom states that disc d can be moved from peg_1 to peg_2 if the initial stack on peg_1 is legal and the resultant stack on peg_2 is legal.

In this last-mentioned formalization, using 13 axioms to specify the problem, QA3 easily solved this problem for the three-disc puzzle. During the search for a proof, 98 clauses were generated buy only 25 of the clauses were accepted. Of the 25, 12 were not in the proof. The solution entails seven moves, thus passing through eight states (counting the initial and final states). The 12 clauses not in the proof correspond to searching through 5 states that are not used in the solution. Thus the solution is found rather easily. Of course, if a sufficiently poor axiomatization is chosen--one requiring an enumeration of enough correct and incorrect disc positions--the system becomes saturated and fails to obtain a solution within time and space constraints. An important factor in the proof search is the elimination of extra clauses corresponding to alternate paths that reach a given state. In the above problem it happens that the subsumption heuristic[8] eliminates 73 of these redundant clauses. However, this particular use of subsumption is problem dependent, thus one must examine any given problem formulation to determine whether or not subsumption will eliminate alternate paths to equivalent states.

The four-disc version of the puzzle can be much more difficult than the three-disc puzzle in terms of search. At about this level of difficulty one must be somewhat more careful to obtain a low-cost solution.

Ernst[9] formalizes the notion of "difference" used by GPS and shows what properties these differences must possess for GPS to succeed on a problem. He then presents a "good" set of differences for the Tower of Hanoi problem. Utilizing this information, GPS solves the problem for four discs, considering no incorrect states in its search. Thus Ernst has chosen a set of differences that guide GPS directly to the solution.

Another method of solution is possible. First, solve the three-disc puzzle. Save the solution to the three-disc puzzle (using the answer statement[4]). Then ask for a solution to the four-disc puzzle. The solution then is: Move the top three discs from peg_1 to peg_2; move disc_4 from peg_1 to peg_3; move the three discs on peg_2 to peg_3. This method produces a much easier solution. But this can be considered as cheating, since the machine is "guided" to a solution by being told which subproblem to first solve and store away. The use of the differences by GPS similarly lets the problem solver be "guided" toward a solution.

There is another possibly more desirable solution. The four-disc puzzle can be posed as the problem, with no three-disc solution. If the solution of the three-disc puzzle occurs during the search for a solution to the four-disc puzzle, and if it is automatically recognized and saved as a lemma, then the four-disc solution should follow easily.

Finally, if an induction axiom is provided, the axioms imply a solution in the form of a recursive program that solves the puzzle for an arbitrary number of discs. Aiko Hormann[10] discusses the related solutions of the four-disc problem by the program GAKU (not an automatic theorem-proving program). The solutions by lemma finding, induction, and search guided by differences have not been run on QA3.

V. Applications to the Robot Project

A. Introduction to Robot Problem Solving

In this section we discuss how theorem-proving methods are being tested for several applications in the Stanford Research Institute Artificial Intelligence Group's Automaton (robot). We emphasize that this section describes work that is now in progress, rather than work that is completed. These methods represent explorations in problem solving, rather than final decisions about how the robot is to do problem solving. An overview of the current status of the entire SRI robot project is provided by Nilsson[4]. Coles[11] has developed an English-to-logic translator that is part of the robot.

We use theorem-proving methods for three purposes, the simplest being the use of QA3 as a central information storage and retrieval system that is accessible to various parts of the system as well as the human users. The data base of QA3 is thus one of the robot's models of its world, including itself.

A second use is as an experimental tool to test out a particular problem formulation. When a suitable formulation is found, it may then be desirable to write a faster or more efficient specific program that implements this formulation, perhaps involving little or no search. If the special program is not as general as the axiom system is, so that the special program fails in certain cases, the axioms can be retained to be used in the troublesome cases. Both solutions can be made available by storing, as the first axiom to be tried, a special axiom that describes the special solution. The predicate-evaluation mechanism can then call LISP to run the special solution. If it fails, the other axioms will then be used.

The third use is as a real-time problem solver. In the implementation we are now using, statements of logic--clauses--are the basic units of information. Statements are derived from several sources: teletype entries, axioms stored in memory, clauses or statements generated by the theorem prover, and statements evaluated by programs--subroutines in LISP, FORTRAN, or machine language. These programs can use robot sensors and sensory data to verify, disprove, or generate statements of logic.

The SRI robot is a cart on wheels, having a TV camera and a range-finder mounted on the cart. There are bumpers on the cart, but no arms or grasping agents, so the only way the robot can manipulate its environment is by simple pushing actions. Given this rather severe restriction of no grasping, the robot must be clever to effectively solve problems involving modifying its world. We present below some axioms for robot problem solving.

The first axiom describes the move routines of the robot:

R1. $(\forall s,p_1,p_2,path_{12})[AT(robot,p_1,s) \wedge PATH(p_1,p_2,path_{12},s) \supset AT(robot,p_2,move(robot,path_{12},s))]$.

This axiom says that if the robot is at p_1 and there is a path to p_2, the robot will be at p_2 after moving along the path. The predicate PATH indicates there exists a robot-path, $path_{12}$, from place p_1 to place p_2. A robot-path is a path adequate for the robot's movement. The terms p_1 and p_2 describe the position of the robot.

In general, it may be very inefficient to use the theorem prover to find the $path_{12}$ such that $PATH(p_1,p_2,path_{12})$ is true. Several existing FORTRAN subroutines, having sophisticated problem-solving capabilities of their own, may be used to determine a good path through obstacles on level ground. We will show later a case where the theorem prover may be used to find a more obscure kind of path. For the less obscure paths, the axiom R1 is merely a description of the semantics of these FORTRAN programs, so that new and meaningful programs can be generated by QA3 by using the efficient path-generating programs as subprograms. The "predicate-evaluation" mechanism is used to call the FORTRAN path-finding routines. The effect of this evaluation mechanism is the same as if the family of axioms of the form $PATH(p_1,p_2,path_{12})$ for all p_1 and p_2 such that $path_{12}$ exists, were all stored in memory and available to the theorem prover.

The second axiom is a push axiom that describes the effect of pushing an object. The robot has no arm or graspers, just a bumper. Its world consists of large objects such as boxes, wedges, cubes, etc. These objects are roughly the same size as the robot itself.

The basic predicate that specifies the position of an object is ATO, meaning at-object. The predicate

$$ATO(object_1,description_1,position_1,s_1)$$

indicates that $object_1$, having structural description "$description_1$", is in position "$position_1$", in state "s_1". At the time of this writing, a particular set of "standard" structure descriptions has not yet been selected. So far several have been used. The simplest description is a point whose position is at the estimated center of gravity of the object. This description is used for the FORTRAN "push in a straight line" routine. Since all the objects in the robot's world are polyhedrons, reasonably simple complete structural de iptions are possible. For example, one structural description consists of the set of polygons that form the surface of the polyhedron. In turn, the structure of the polygons is given by the set of vertices in its boundary. Connectivity of structures can be stated explicitly or else implied by common boundaries. The position of an object is given by a mapping of the topologically-described structure into the robot's coordinate system. Such structural descriptions may be given as axioms or supplied by the scene-analysis programs used by the robot.

A basic axiom describing the robot's manipulation of an object is

R2.

$(\forall s, obj_1, desc_1, pos_1, pos_2)[ATO(obj_1, desc_1, pos_1, s) \wedge MOVABLE(obj_1) \wedge ROTATE\text{-}TRANSLATE\text{-}ABLE(desc_1, pos_1, pos_2) \wedge OBJECT\text{-}PATH(desc_1, pos_1, pos_2, path_{12}, s) \supset ATO(obj_1, desc_1, pos_2, push(obj_1, path_{12}, s))]$

This axiom says that if object 1, described by description 1, is at position 1, and object 1 is movable, and object 1 can be theoretically rotated and translated to the new position 2, and there is an object-path from 1 to 2, then object 1 will be at position 2 as a result of pushing it along the path. The predicate $ROTATE\text{-}TRANSLATABLE(desc_1, pos_1, pos_2)$ checks the necessary condition that the object can be theoretically rotated and translated into the new position. The predicate $OBJECT\text{-}PATH(desc_1, pos_1, pos_2, path_{12})$ means that pos_2 is the estimated new position resulting from pushing along push-path, $path_{12}$.

Let us now return to the frame problem. More specifically, in a state resulting from pushing an object, how can we indicate the location of objects which were not pushed? One such axiom is

R3. $(\forall obj_1, obj_2, desc_1, pos_1, path_{12}, s)[ATO(obj_1, desc_1, pos_1, s) \wedge \sim SAME(obj_1, obj_2) \supset ATO(obj_1, desc_1, pos_1, push(obj_2, path_{12}, s))]$.

This axiom says that all objects that are not the same as the pushed object are unmoved. The predicate evaluation mechanism is used to evaluate SAME and speed up the proof. One can use this predicate evaluation mechanism, and perhaps other fast methods for handling classes of deductions (such as special representations of state-dependent information and special programs for updating this information--which is done in the robot), but another problem remains. Observe that axiom R3 assumes that only the objects directly pushed by the robot move. This is not always the case, since an object being pushed might accidentally strike another object and move it. This leads to the question of dealing with the real world and using axioms to approximate the real world.

B. Real-World Problem Solving: Feedback

Our descriptions of the real world, axiomatic or otherwise, are at best only approximations. For example, the new position of an object moved by the robot will not necessarily be accurately predicted, even if one goes to great extremes to calculate a predicted new position. The robot does not have a grasp on the object so that some slippage may occur. The floor surface is not uniform and smooth. The weight distribution of objects is not known. There is only rudimentary kinesthetic sensing feedback--namely, whether or not the bumper is still in contact with the object. Thus it appears that a large feedback loop iterating toward a solution, is necessary: Form a plan for pushing the object (possibly using the push axiom), push according to the plan, back up, take a look, see where the object is, compare the position to the desired position, start over again. The new position (to some level of accuracy) is provided by the sensors of the robot. This new position is compared to the position predicted by the axiom. If the move is not successful, the predicate (provided by sensors in the new state) that reasonably accurately gives the object's position in the new state must be used as the description of the initial state for the next attempt.

This feedback method can be extended to sequences of actions. Consider the problem: Find s_f such that $P_3(s_f)$ is true. Suppose the starting state is s_0, with property $P_0(s_0)$. Suppose the axioms are as follows:

$$P_0(s_0)$$

$$(\forall s)[P_0(s) \supset P_1(f_1(s))]$$

$$(\forall s)[P_1(s) \supset P_2(f_2(s))]$$

$$(\forall s)[P_2(s) \supset P_3(f_3(s))].$$

The sequence of actions $f_3(f_2(f_1(s_0)))$ transforms state s_0 with property $P_0(s_0)$ into state s_f having property $P_3(s_f)$.

The solution is thus $s_f = f_3(f_2(f_1(s_0)))$.

Corresponding to each "theoretical" predicate $P_i(s)$ is a corresponding "real-word" predicate $P_i'(s)$. The truth value of $P_i'(s)$ is determined by sensors and the robot's internal model of the world. It has built-in bounds on how close its measurements must be to the correct values in order to assert that it is true.* The proof implies the following description of the result after each step of execution of $f_3(f_2(f_1(s_0)))$:

* At this time, a many-valued logic having degrees of truth is not used, although this is an interesting possibility.

Actions and Successive States	Predicted Theoretical Results	Predicted Real-World Results
s_0	$P_0(s_0)$	$P'_0(s_0)$
$s_1 = f_1(s_0)$	$P_1(s_1)$	$P'_1(s_1)$
$s_2 = f_2(s_1)$	$P_2(s_2)$	$P'_2(s_2)$
$s_f = f_3(s_2)$	$P_3(s_3)$	$P'_3(s_f)$

To measure progress after, say, the ith step, one checks that $P'_i(s_i)$ is true. If not, then some other condition $P''_i(s_i)$ holds and a new problem is generated, given $P''_i(s_i)$ as the starting point. If new information is present, such as is the case when the robot hits an obstacle that is not in its model, the model is updated before a new solution is attempted. The position of this new object of course invalidates the previous plan--i.e., had the new object's position been known, the previous plan would not have been generated.

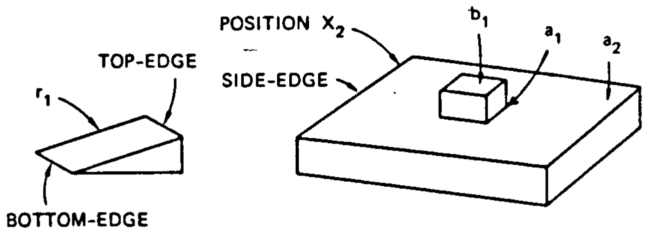
The new solution may still be able to use that part of the old solution that is not invalidated by any new information. For example, if $P''_i(s_i)$ holds, it may still be possible to reach the jth intermediate state and then continue the planned sequence of actions from the jth state. However, the object-pushing axiom is an example of an axiom that probably will incorrectly predict results and yet no further information, except for the new position, will be available. For this case, the best approach is probably to iterate toward the target state by repeated use of the push axiom to generate a new plan. Hopefully, the process converges.

For a given axiomatization feedback does not necessarily make it any easier to find a proof. However, knowing that the system uses feedback allows us to choose a simpler and less accurate axiom system. Simple axiom systems can then lead to shorter proofs.

One can envision formalizing this entire problem-solving process, including the notion of feedback, verifying whether or not a given condition is met, updating the model, recursively calling the theorem prover, etc. The author has not attempted such a formalization, although he has written a first-order formalization of the theorem prover's own problem-solving strategy. This raises the very interesting possibility of self-modification of strategy; however, in practice such problems lie well beyond the current theorem-proving capacity of the program.

C. A Simple Robot Problem

. Now let us . .sider a problem requiring the use of a ramp to roll onto a platform, as illustrated below.



TA-7494-5

The goal is to push the box b_1 from position x_1 to x_2. To get onto the platform, the robot must push the ramp r_1 to the platform, and then roll up the ramp onto the platform.

A simple problem formulation can use a special ramp-using axiom such as

R4. $(\forall x_1,x_2,s,\text{top-edge},\text{bottom-edge},\text{ramp}_1)$
$[\text{AT-RAMP}(\text{ramp}_1,\text{top-edge},x_2,\text{bottom-edge},$
$x_1,s) \wedge \text{AT-PLATFORM}(\text{side-edge},x_2,s) \supset$
$\text{AT}(\text{robot},x_2,\text{climb}(\text{ramp}_1,x_1,s))]$

with the obvious meaning. Such a solution is quick but leaves much to be desired in terms of generality.

A more general problem statement is one in which the robot has a description of its own capabilities, and a translation of this statement of its abilities into the basic terms that describe its sensory and human-given model of the world. It then learns from a fundamental level to deal with the world. Such a knowledge doesn't make for the quickest solution to a frequently-encountered problem, but certainly does lend itself to learning, greater degrees of problem-solving, and self-reliance in a new problem situation.

Closer to this extreme of greatest generality is the following axiomatization.

R5. $(\forall x_1,x_2,r)[\text{RECTANGLE}(r,x_1,x_2) \wedge$
$\text{LESSP}(\text{maxslope}(r),k_0) \wedge \text{LESSP}(r_0,\text{width}(r)) \wedge$
$\text{CLEAR}(\text{space}(r,h_0),s) \wedge \text{SOLID}(r) \supset$
$\text{PATH}(x_1,x_2,r)]$.

This axiom says that r describes a rectangle having ends x_1 and x_2. The maximum slope is less than a constant k_0, the width of r is greater than the robot's width w_0, the space above r to the robot's height h_0 is clear, and the rectangle r has a solid surface.

Two paths can be joined as follows:

R6. $(\forall x_1,x_2,x_3,r_1,r_2)[\text{PATH}(x_1,x_2,r_1) \wedge \text{PATH}(x_2,x_3,r_2) \supset \text{PATH}(x_1,x_3,\text{join}(r_1,r_2))]$.

From these two axioms (R5 and R6), the push axiom (R2), and a recognition of a solid object that can be used as a ramp, a solution can be obtained in terms of climb, push, join, etc. This more general method of solution would of course be slower than using the special ramp axiom. On the other hand, the more general method will probably be more useful if the robot will be required to construct a ramp, or recognize and push over a potential ramp that is standing on its wide end.

The danger in trying the more general methods is that one may be asking the theorem prover to rederive some significant portion of math or physics, in order to solve some simple problem.

VI. Automatic Programming

A. Introduction

The automatic writing, checking, and debugging of computer programs are problems of great interest both for their independent importance and as useful tools for intelligent machines. This section shows how a theorem prover can be used to solve certain automatic programming problems. The formalization given here will be used to precisely state and solve the problem of automatic generation of programs, including recursive programs, along with concurrent generation of proofs of the correctness of these programs. Thus any programs automatically written by this method have no errors.

We shall take LISP[12,13] as our example of a programming language. In the LISP language, a function is described by two entities: (1) its value, and (2) its side effect. Side effects can be described in terms of their effect upon the state of the program. Methods for describing state-transformation operations, as well as methods for the automatic writing of programs in a state-transformation language, were presented in Secs. I and II. For simplicity, in this section we shall discuss "pure" LISP, in which a LISP function corresponds to the standard notion of a function--i.e., it has a value but no side effect.

Thus we shall use pure LISP 1.5 without the program feature, which is essentially the lambda calculus. In this restricted system, a LISP program is merely a function. For example, the LISP function car applied to a list returns the first element of the list. Thus if the variable x has as value the list (a b c), then car(x) = a. The LISP function cdr yields the remainder of the list, thus CDR(x) = (b c), and car(cdr(x)) = b. There are several approaches one may take in formalizing LISP; the one given here is a simple mapping from LISP's lambda calculus to the predicate calculus. LISP programs are represented by functions. The syntax of pure LISP 1.5, is normal function composition, and the corresponding syntax for the formalization is also function composition. LISP "predicates" are represented in LISP--and in this formalization--as functions having either the value NIL (false) or else a value not equal to NIL (true). The semantics are given by axioms relating LISP functions to list structures, e.g., $(\forall x,y)\text{car}(\text{cons}(x,y)) = x$, where cons(x,y) is the list whose first element is x and whose remainder is y.

In our formulation of programming problems, we emphasize the distinction between the program (represented as a function in LISP), that solves a problem and a test for the validity of a solution to a problem (represented as a predicate in logic). It is often much easier to construct the predicate than it is to construct the function. Indeed, one may say that a problem is not well defined until an effective test for its solution is provided.

For example, suppose we wish to write a program that sorts a list. This problem is not fully specified until the meaning of "sort" is explained; and the method of explanation we choose is to provide a predicate R(x,y) that is true if list y is a sorted version of list x and false otherwise. (The precise method of defining this relation R will be given later.)

In general, our approach to using a theorem prover to solve programming problems in LISP requires that we give the theorem prover two sets of initial axioms:

(1) Axioms defining the functions and constructs of the subset of LISP to be used

(2) Axioms defining an input-output relation such as the relation R(x,y), which is to be true if and only if x is any input of the appropriate form for some LISP program and y is the corresponding output to be produced by such a program.

Given this relation R, and the LISP axioms, by having the theorem prover prove (or disprove) the appropriate question we can formulate the following four kinds of programming problems: checking, simulation, verifying (debugging), and program writing. These problems may be explained using the sort program as an example as follows:

(1) Checking: The form of the question is R(a,b) where a and b are two given lists. By proving R(a,b) true or false, b is checked to be either a sorted version of a or not. The desired answer is accordingly either yes or no.

(2) Simulation: The form of the question is $(\exists x)R(a,x)$, where a is a given input list. If the question $(\exists x)R(a,x)$ is answered yes, then a sorted version of x exists and a sorted version is constructed by the theorem prover. Thus the theorem prover acts as a sort program. If the answer is no, then it has proved that a sorted version of x does not exist (an impossible answer if a is a proper list).

(3) Verifying: The form of the question is $(\forall x)R(x,g(x))$, where g(x) is a program written

by the user. This mode is known as verifying, debugging, proving a program correct, or proving a program incorrect. If the answer to $(\forall x)R(x,g(x))$ is yes, then g(x) sorts every proper input list and the program is correct. If the answer is no, a counterexample list c, that the program will not sort, must be constructed by the theorem prover. This mode requires induction axioms to prove that looping or recursive programs converge.

(4) Program Writing: The form of the question is $(\forall x)(\exists y)R(x,y)$. In this synthesis mode the program is to be constructed or else proved impossible to construct. If the answer is yes, then a program, say f(x), must be constructed that will sort all proper input lists. If the answer is no, an unsortable list (impossible, in this case) must be produced. This mode also requires induction axioms. The form of the problem statement shown here is oversimplified for the sake of clarity. The exact form will be shown later.

In addition to the possibility of "yes" answer and the "no" answer, there is always the possibility of a "no proof found" answer if the search is halted by some time or space bound. The elimination of disjunctive answers, which is assumed in this section, was explained in Sec. B.

These methods are summarized in the following table. The reader may view R(x,y) as representing some general desired input-output relationship.

Programming Problem	Form of Question	Desired Answer
(1) Checking	$R(a,b)$	yes or no
(2) Simulation	$(\exists x)R(a,x)$	yes, $x = b$ or no
(3) Verifying	$(\forall x)R(x,g(x))$	yes or no, $x = c$
(4) Program Writing	$(\forall x)(\exists y)R(x,y)$	yes, $y = f(x)$ or no, $x = c$

We now present an axiomatization of LISP followed by two axiomatizations of the sort relation R (one for a special case and one more general).

B. Axiomatization of a Subset of LISP

All LISP functions and predicates will be written in small letters. The functions "equal(x,y)," "at(x)," and "null(x)" evaluate to "nil" if false and something not equal to "nil," say "T," if true. The predicates of first-order logic that are used to describe LISP are written in capital letters. These, of course, have truth values.

The version of LISP described here does not distinguish between an S-expression and a copy of that S-expression. There is some redundancy in the following formulation, in that certain functions and predicates could have been defined in terms of others; however, the redundancy allows us to state the problem more concisely. Also, some axioms could have been eliminated since thev are derivable from others, but are included for clarity. The variables x, y, and z are bound by universal quantifiers, but the quantifiers are omitted for the sake of readability wherever possible. The formulation is given below:

Predicates	Meaning
NULL(x)	x = nil
LIST(x)	x is a list
ATOM(x)	x is an atom
x = y	x is equal to y

Functions	Meaning
car(x)	The first element of the list x.
cdr(x)	The rest of the list x.
cons(x,y)	If y is a list then the value of cons(x,y) is a new list that has x as its first element and y as the rest of the list, e.g., cons(1,(2 3)) = (1 2 3). If y is an atom instead of a list, cons(x,y) has as value a "dotted pair," e.g., cons(1,2) = (1·2).
cond(x,y,z)	The conditional statement, <u>if</u> x = nil <u>then</u> y <u>else</u> z. Note that the syntax of this function is slightly different than the usual LISP syntax.
nil	The null (empty) list containing no elements.
equal(x,y)	Equality test, whose value is "nil" if x does not equal y.
atom(x)	Atom test, whose value is "nil" if x is not an atom.
null(x)	Null test, whose value is "nil" if x is not equal to nil.

Axioms

L1: $x = car(cons(x,y))$

L2: $y = cdr(cons(x,y))$

L3: $\sim ATOM(x) \supset x = cons(car(x),cdr(x))$

L4: $\sim ATOM(cons(x,y))$

L5: $ATOM(nil)$

L6: $x = nil \supset cond(x,y,z) = z$

L7: $x \neq nil \supset cond(x,y,z) = y$

L8: $x = y \equiv equal(x,y) \neq nil$

L9: $ATOM(x) \equiv atom(x) \neq nil$

L10: $NULL(x) \equiv null(x) \neq nil$

C. A Simplified Sort Problem

Before examining a more general sort problem, consider the following very simple special case.

Instead of a list-sorting program, consider a program that "sorts" a dotted pair of two distinct numbers; i.e., given an input pair the program returns as an output pair the same two numbers, but the first number of the output pair must be smaller than the second. To specify such a program, we must define the simple version of R, $R_b(x,y)$. Let us say that a dotted pair of numbers is "sorted" if the first number is less than the second. Thus $R_b(x,y)$ is true if and only if y equals x when x is sorted and y is the reverse of x when x is not sorted. Stated more precisely, we have

R1. $(\forall x,y)\{R_b(x,y) \equiv [[\text{car}(x) < \text{cdr}(x) \supset y = x] \wedge [\text{car}(x) \not< \text{cdr}(x) \supset \text{car}(y) = \text{cdr}(x) \wedge \text{cdr}(y) = \text{car}(x)]]\}.$

The correspondence of the LISP "lessp" function to the "less-than" relation is provided in the following axiom:

R2. $(\forall x,y)[\text{lessp}(x,y) \neq \text{nil} \equiv x < y].$

Using the predicate R_b we will give examples of four programming problems and their solutions:

(1) Checking:

Q: $R_b(\text{cons}(2,1),\text{cons}(1,2))$

A: yes

(2) Simulation:

Q: $(\exists x)R_b(\text{cons}(2,1),x)$

A: yes, $x = \text{cons}(1,2)$

(3) Verifying:

Q: $(\forall x)R_b(x,\text{cond}(\text{lessp}(\text{car}(x),\text{cdr}(x)),x,\text{cons}(\text{cdr}(x),\text{car}(x)))$

A: yes

Thus the program supplied by the user is correct.

(4) Program writing:

Q: $(\forall x)(\exists x)R_b(x,y)$

A: yes, $y = \text{cond}(\text{lessp}(\text{car}(x),\text{cdr}(x)),x,\text{cons}(\text{cdr}(x),\text{car}(x)))$

Translated into a more readable form, the program is:

<u>if</u> car(x) < cdr(x) <u>then</u> x <u>else</u>
cons(cdr(x),car(x)).

Given only the necessary axioms--L1, L2, L6, L7, R1, and R2--QA3 found a proof that constructed the sort program shown above. The paramodulation[14,15] rule of inference was used to handle equality.

We now turn to a more difficult problem.

D. The Sort Axioms

The definition of the predicate R is in terms of the predicates ON and SD. The meaning of these predicates is given below:

R(x,y)	A predicate stating that if x is a list of numbers with no number occurring more than once in the list, then y is a list containing the same elements as x, and y is sorted, i.e., the numbers are arranged in order of increasing size.
ON(x,y)	A predicate stating that x is an element on the list y.
SD(x)	A predicate stating that the list x is sorted.

First we define R(x,y), that y is a sorted version of x, as follows:

S1. $(\forall x,y)\{R(x,y) \equiv [(\forall z)[\text{ON}(z,x) \equiv \text{ON}(z,y)] \wedge \text{SD}(y)]$

Thus a sorted version y of list x contains the same elements as x and is sorted.

Next we define, recursively, the predicate ON(x,y):

S2. $(\forall x,y)\{\text{ON}(x,y) \equiv [\sim\text{ATOM}(y) \wedge [x = \text{car}(y) \vee \text{ON}(x,\text{cdr}(y))]]\}$

This axiom states that x is on y if and only if x is the first element of y or if x is on the rest of y.

Next we define the meaning of a sorted list:

S3. $(\forall x)\{\text{SD}(x) \equiv [\text{NULL}(x) \vee [\sim\text{ATOM}(x) \wedge \text{NULL}(\text{cdr}(x))] \vee [\sim\text{ATOM}(x) \wedge \sim\text{NULL}(\text{cdr}(x)) \wedge \text{car}(x) \leq \text{car}(\text{cdr}(x)) \wedge \text{SD}(\text{cdr}(x))]]\}.$

This axiom states that x is sorted if and only if x is empty, or x contains only one element, or the first element of x is less than the second element and the rest of x is sorted.

To simplify the problem statement we assume that the arguments of the predicates and functions range only over the proper type of objects--i.e., either numbers or lists. In effect, we are assuming that the input list will indeed be a properly formed list of numbers. (The problem statement could be modified to specify correct types by using predicates such as NUMBERP(x)--true only if x is, say, a real number).

The problem is made simpler by using a "merge" function. This function, and a predicate P describing the merge function are named and described as follows:

sort(x)	A LISP sort function (to be constructed) giving as its value a sorted version of x.
merge(x,u)	A LISP merge function merging x into the sorted list u, such that the list returned contains the elements of u, and also contains x, and this list is sorted.
P(x,u,y)	A predicate stating that y is the result of merging x into the sorted list u.

We define P(x,u,y), that y is u with x merged into it:

S4. $(\forall x,u,y)[P(x,u,y) \equiv [SD(u) \supset [SD(y) \wedge (\forall z)(ON(z,y) \equiv (ON(z,u) \vee z = x))]]]$.

Thus P(x,u,y) holds if and only if the fact that u is sorted implies that y contains x in addition to the elements of u, and y is sorted. One such merge function is merge(x,u) = cond(null(u),cons(x,u), cond(lessp(x,car(u)),cons(x,u),cons(car(u),merge(x, cdr(u))))).

The axiom required to describe the merge function is:

S5. $(\forall x,u)P(x,u,merge(x,u))$.

This completes a description of the predicates ON, SD, R, and P. Together, these specify the input-output relation for a sort function and a merge function. Before posing the problems to the theorem prover, we need to introduce axioms that describe the convergence of recursive functions.

E. Induction Axioms

In order to prove that a recursive function converges to the proper value, the theorem prover requires an induction axiom. An example of an induction principle is that if one keeps taking "cdr" of a finite list, one will reach the end of the list in a finite number of steps. This is analogous to an induction principle on the non-negative integers, i.e., let "P" be a predicate, and "h" a function. Then for finite lists,

$$[P(h(nil)) \wedge (\forall x)[\sim ATOM(x) \wedge P(h(cdr(x))) \supset P(h(x))]] \supset (\forall z)P(h(z))$$

is analogous to

$$[P(h(0)) \wedge (\forall n)[n \not< 0 \wedge P(h(n-1)) \supset P(h(n))]] \supset (\forall m)P(h(m))$$

for nonnegative integers.

There are other kinds of induction criteria besides the one given above. Unfortunately, for each recursive function that is to be shown to converge, the appropriate induction axiom must be carefully formulated by the user. The induction axiom also serves the purpose of introducing the <u>name</u> of the function to be written. We will now give the problem statement for the sort program, introducing appropriate induction information where necessary.

F. The Sort Problem

Examples illustrating the four kinds of problems are shown below.

(1) Checking:

Q: R(cons(2,cons(1,nil)),cons(1,cons(2,nil)))

A: yes

(2) Simulation:

Q: $(\exists x)$R(cons(2,cons(1,nil)),x)

A: yes, x = cons(1,cons(2,nil))

(3) Verifying: Now consider the verifying or debugging problem. Suppose we are given a proposed definition of a sort function and we want to know if it is correct. Suppose the proposed definition is

S6. $(\forall x)[sort(x) \equiv cond(null(x),nil,merge(car(x), sort(cdr(x))))]$.

Thus sort is defined in terms of car, cdr, cond, null, merge, and sort. Each of these functions except sort is already described by previously given axioms. We also need the appropriate induction axiom in terms of sort. Of course, the particular induction axiom needed depends on the definition of the particular sort function given. For this sort function the particular induction axiom needed is

S7. $[R(nil,sort(nil)) \wedge (\forall x)[\sim ATOM(x) \wedge R(cdr(x),sort(cdr(x))) \supset R(x,sort(x))]] \supset (\forall y)R(y,sort(y))$.

The following conjecture can then be posed to the theorem prover:

Q: $(\forall x)$R(x,sort(x))

A: yes

(4) Program writing: The next problem is that of synthesizing or writing a sort function. We assume, of course, that no definition such as S6 is provided. Certain information needed for this particular problem might be considered to be a part of this particular problem statement rather than a part of the data base. We shall phrase the question so that in addition to s primary purpose of asking for a solution, the question provides three more pieces of information: (a) The question assigns a name to the function that is to be constructed. A recursive function is defined in terms of itself, so to construct this definition the name of the function must be known (or else created internally). (b) The question specifies the number of arguments of the function that is to be considered.

(c) The question (rather than an induction axiom) gives the particular inductive hypothesis to be used in constructing the function.

In this form, the question and answer are

$$Q:\ (\forall x)(\exists y)\{R(nil,y) \wedge [[\sim ATOM(x) \wedge R(cdr(x),sort(cdr(x)))] \supset R(x,y)]\}$$

$$A:\ yes,\ y = cond(equal(x,nil),nil,merge(car(x),sort(cdr(x)))).$$

Thus the question names the function to be "sort" and specifies that it is a function of one argument. The question gives the inductive hypothesis--that the function sorts cdr(x)--and then asks for a function that sorts x. When the answer y is found, y is labeled to be the function sort(x).

Using this formulation QA3 was unable to write the sort program in a reasonable amount of time, although the author did find a correct proof within the resolution formalism*. The creation of the merge function can also be posed to the theorem prover by the same methods.

G. Discussion of Automatic Programming Problems

The axioms and conjectures given here illustrate the fundamental ideas of automatic programming. However, this work as well as earlier work by Simon,[16] Slagle,[17] Floyd,[18] Manna,[19] and others provides merely a small part of what needs to be done. Below we present discussion of issues that might profit from fruther investigation.

Loops. One obvious extension of this method is to create programs that have loops rather than recursion. A simple technique exists for carrying out this operation. First, one writes just recursive functions. Many recursive functions can then be converted into iteration--i.e., faster-running loops that do not use a stack. McCarthy[20] gives criteria that determine how to convert recursion to iteration. An algorithm for determining cases in which recursion can be converted to iteration, and then performing the conversion process is embedded in modern LISP compilers. This algorithm could be applied to recursive functions written by the theorem-proving program.

Separation of Aspects of Problem Solving. Let us divide information into three types: (1) Information concerning the problem description and semantics. An example of such information is given in the axiom $AT(a,s_0)$, or axiom S1 that defines a sorted list. (2) Information concerning the target programming language, such as the axiom $[x = nil \supset cond(x,y,z) = z]$. (3) Information concerning the interrelation of the problem and the target language, such as $[LESS(x,y) \equiv lessp(x,y) \neq nil]$.

* After this paper was written the problem was reformulated using a different set of axioms. In the new formulation QA3 created the sort program "sort(x) = cond(x,merge(car(x),sort(cdr(x))),nil).

These kinds of information are not, of course, mutually exclusive.

In the axiom systems presented, no distinction is made between such classes of information. Consequently, during the search for a proof the theorem prover might attempt to use axioms of type 1 for purposes where it needs information of type 2. Such attempts lead nowhere and generate useless clauses. However, as discussed in Sec. II-G, we can place in the proof strategy our knowledge of when such information is to be used, thus leading to more efficient proofs. One such method--calling for the conditional axioms at the right time, as discussed in Sec. II-G--has been implemented in QA3.

The PROW program of Waldinger and Lee[6] provides a very promising method of separating the problem of proof construction from the problem of program construction. In their system, the only axioms used are those that describe the subject--i.e., state the problem. Their proof that a solution exists does not directly construct the program. Instead, information about the target programming language, as well as information about the relationship of the target-programming language to the problem-statement language, is in another part of the PROW program--the "post-processor." The post-processor then uses this information to convert the completed proof into a program. The post-processor also converts recursion into loops and allows several target programming languages.

If our goal is to do automatic programming involving complex programs, we will probably wish to do some optimization or problem solving on the target language itself. For this reason we might want to have axioms that give the semantics of the target language, and also allow the intercommunication of information in the problem-statement language with information in the target language. Two possibilities for how to do this efficiently suggest themselves: (a) Use the methods presented here in which all information is in first-order logic. To gain efficiency, use special problem-solving strategies that minimize unnecessary interaction; (b) Use a higher-order logic system, in which the program construction is separated from the proof construction, possibly by being at another level. The program construction process might then be described in terms of the first-order existence proof.

Problem Formulation. The axiomatization given here has considerable room for improvement: Missing portions of LISP include the program features and the use of lambda to bind variables. The functions to be written must be named by the user, and the number of arguments must also be specified by the user.

Heuristics for Program-Writing Problems. Two heuristics have been considered so far. The first consists of examining the program as it is constructed (by looking inside the answer literal). Even though the syntax is guaranteed correct, the answer literal may contain various nonsense or undefined constructions (such as car(nil)). Any clause containing such constructed answers should be eliminated. Another heuristic is to actually run the partial program by a pseudo-LISP interpreter on a sample problem. The theorem prover knows the correct performance on these sample problems because they have either been solutions or else counterexamples to program-simulation questions that were stored in memory, or else they have been provided by the user. If the pseudo-LISP interpreter can produce a partial output that is incorrect, the partial program can be eliminated. If done properly, such a method might be valuable, but in our limited experience, its usefulness is not yet clear.

Higher-Level Programming Concepts. A necessary requirement for practical program writing is the development of higher-level concepts (such as the LISP "map" function) that describe the use of frequently employed constructs (functions) or partial constructs.

Induction. The various methods of proof by induction should be studied further and related to the kinds of problems in which they are useful. The automatic selection or generation of appropriate induction axioms would be most helpful.

Program Segmentation. Another interesting problem is that of automatically generating the specifications for the subfunctions to be called before writing these functions. For example, in our system, the sort problem was divided into two problems: First, specify and create a merge function, next specify a sort function and then construct this function in terms of the merge function. The segmentation into two problems and the specification of each problem was provided by the user.

VII. Discussion

The theorem prover may be considered an "interpreter" for a high-level assertional or declarative language--logic. As in the case with most high-level programming languages the user may be somewhat distant from the efficiency of "logic" programs unless he knows something about the strategies of the system.

The first applications of QA2 and QA3 were to "question answering." Typical question-answering applications are usually easy for a resolution-type theorem prover. Examples of such easy problem sets given QA3 include the questions done by Raphael's SIR,[21] Slagle's DEDUCOM,[17] and Cooper's chemistry question-answering program.[22] Usually there are a few obvious formulations for some subject area, and any reasonable formulation works well. As one goes to harder problems e the Tower of Hanoi puzzle, and program-writing problems, good and reasonably well-thought-out representations are necessary for efficient problem solving.

Some representations are better than others only because of the particular strategy used to search for a proof. It would be desirable if the theorem prover could adopt the best strategy for a given problem and representation, or even change the representation. I don't believe these goals are impossible, but at present it is not done. However, a library of strategy programs and a strategy language is slowly evolving in QA3. To change strategies in the present version the user must know about set-of-support and other program parameters such as level bound[1] and term-depth bound. To radically change the strategy, the user presently has to know the LISP language and must be able to modify certain strategy sections of the program. In practice, several individuals who have used the system have modified the search strategies to suit their needs. To add and debug a new heuristic or to modify a search strategy where reprogramming is required seems to take from a few minutes to several days, perhaps averaging one day. Ultimately it is intended that the system will be able to write simple strategy programs itself, and "understand" the semantics of its strategies.

Experience with the robot applications and the automatic programming applications emphasize the need for a very versatile logical system. A suitable higher-order logic system seems to be one of the best candidates. Several recent papers are relevant to this topic. A promising higher order system has been proposed by Robinson.[23] Banerji[24] discusses a higher order language. One crucial factor in an inference system is a suitable method for the treatment of the equality relation. Discussion of methods for the treatment of equality is provided by Wos and Robinson,[14] and Robinson and Wos,[15] and Kowalski.[25] McCarthy and Hayes[5] include a discussion of modal logics.

The theorem-proving program can be used as an experimental tool in the testing of problem formulations. In exploring difficult problems it can be useful to write a computer program to test a problem formulation and solution technique, since the machine tends to sharpen one's understanding of the problem. I believe that in some problem-solving applications the "high-level language" of logic along with a theorem-proving program can be a quick programming method for testing ideas. One reason is that a representation in the form of an axiom system can correspond quite closely to one's conceptualization of a problem. Another reason is that it is sometimes easier to reformulate an axiom system rather than to rewrite a problem-solving program, and this ease of reformulation facilitates exploration.

Resolution theorem-proving methods are shown in this paper to have the potential to serve as a general problem-solving system. A modified theorem-proving program can write simple robot problems, and solve simple puzzles. Much work remains to be done before such a system is capable of solving problems that are difficult by human standards.

Acknowledgment

I would like to acknowledge valuable discussions with Dr. Bertram Raphael and Mr. Robert Yates.

REFERENCES

1. J. A. Robinson, "The Present State of Mechanical Theorem Proving," a paper presented at the Fourth Systems Symposium, Cleveland, Ohio, November 19-20, 1968 (proceedings to be published).

2. C. Green and B. Raphael, "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. 23rd Nat'l. Conf. ACM, (Thompson Book Company, Washington, D.C., 1968).

3. C. Green, "Theorem Proving by Resolution as a Basis for Question-Answering Systems," Machine Intelligence 4, D. Michie and B. Meltzer, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

4. N. J. Nilsson, "A Mobile Automaton: An Application of Artificial Intelligence Techniques," a paper presented at the International Joint Conference on Artificial Intelligence, Washington, D.C., May 7-9, 1969 (proceedings to be published).

5. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," Machine Intelligence 4, D. Michie and B. Meltzer, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

6. R. J. Waldinger and R. C. T. Lee, "PROW: A Step Toward Automatic Program Writing," a paper presented at the International Joint Conference on Artificial Intelligence, Washington, D.C., May 7-9, 1969 (proceedings to be published).

7. L. Wos, G. A. Robinson, and D. F. Carson, "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving," J.ACM, Vol. 12, No. 4, pp. 536-541 (October 1965).

8. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," J.ACM, Vol. 12, No. 1, pp. 23-41 (January 1965).

9. George Ernst, "Sufficient Conditions for the Success of GPS," Report No. SRC-68-17, Systems Research Center, Case Western Reserve University, Celveland, Ohio (July 1968).

10. A. Hormann, "How a Computer System Can Learn," IEEE Spectrum (July 1964).

11. L. S. Coles, "Talking With a Robot in English," paper submitted at the International Joint Conference on Artificial Intelligence, Washington, D.C., May 7-9, 1969 (proceedings to be published).

12. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, LISP 1.5 Programmer's Manual (The MIT Press, Cambridge, Mass., 1962).

13. C. Weissman, LISP 1.5 Primer (Dickenson Publishing Company, Inc., Belmont, Calif., 1967).

14. Lawrence Wos and George Robinson, "Paramodulation and Set of Support," summary of paper presented at the IRIA Symposium on Automatic Demonstration at Versailles, France, December 16-21, 1968 (proceedings to be published).

15. G. Robinson and L. Wos, "Paramodulation and Theorem-Proving in First-Order Theories with Equality," Machine Intelligence 4, B. Meltzer and D. Michie, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

16. H. Simon, "Experiments with a Heuristic Compiler," J.ACM, Vol. 10, pp. 493-506 (October 1963).

17. J. R. Slagle, "Experiments with a Deductive, Question-Answering Program," Comm. ACM, Vol. 8, pp. 792-798 (December 1965).

18. R. W. Floyd, "The Verifying Compiler," Computer Science Research Review, Carnegie Mellon University (December 1967).

19. Z. Manna, "The Correctness of Programs," J. Computer and Systems Sciences, Vol. 3 (1969).

20. J. McCarthy, "Towards a Mathematical Science of Computation," Proceedings ICIP (North Holland Publishing Company, Amsterdam, 1962).

21. B. Raphael, "A Computer Program Which 'Understands'," Proc. FJCC, pp. 577-589 (1964).

22. W. S. Cooper, "Fact Retrieval and Deductive Question Answering Information Retrieval Systems," J.ACM, Vol. 11, pp. 117-137 (April 1964).

23. J. A. Robinson, "Mechanizing Higher Order Logic," Machine Intelligence 4, D. Michie and B. Meltzer, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

24. R. B. Banerji, "A Language for Pattern Recognition," Pattern Recognition, Vol. 1, No. 1, pp. 63-74 (1968).

25. R. Kowalski, "The Case for Using Equality Axioms in Automatic Demonstration," paper presented at the IRIA Symposium on Automatic Demonstration at Versailles, France, December 16-21, 1968 (proceedings to be published).

APPENDIX

The axioms for the Monkey and Bananas problem are listed below, followed by the proof. The term SK24(S,P2,P1,B) that first appears in clause 16 of the proof is a Skolem function generated by the elimination of (∀x) in the conversion of axiom MB4 to quantifier-free clause form. (One may think of it as the object that is not at place P2 in state S.)

```
LIST MONKEY
MB1  (MOVABLE BOX)
MB2  (FA(X)(NOT(AT X UNDER-BANANAS SØ)))
MB3  (AT BOX PLACEB SØ)
MB4  (FA(B P1 P2 S)(IF(AND(AT B P1 S)(MOVABLE B)(FA(X)(NOT(AT X P2 S))))(AND(AT MONKEY P2
     (MOVE(MONKEY B P2 S))(AT B P2(MOVE MONKEY B P2 S)))))
MB5  (FA(S)(CLIMBABLE MONKEY BOX S))
MB6  (FA(M P B S)(IF(AND(AT B P S)(CLIMBABLE M B S))(AND(AT B P(CLIMB M B S))(ON M B
     (CLIMB M B S)))))
MB7  (FA(S)(IF(AND(AT BOX UNDER-BANANAS S)(ON MONKEY BOX S))(REACHABLE MONKEY BANANAS S)))
MB8  (FA(M B S)(IF(REACHABLE M B S)(HAS M B(REACH M B S))))
DONE

Q    (EX(S)(HAS MONKEY BANANAS S))
A    YES, S = REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,MOVE(MONKEY,BOX,UNDER-BANANAS,SØ)))

PROOF
 1   -AT(X,UNDER-BANANAS,SØ)                                                           AXIOM
 2   AT(BOX,PLACEB,SØ)                                                                 AXIOM
 3   CLIMBABLE(MONKEY,BOX,S)                                                           AXIOM
 4   -HAS(MONKEY,BANANAS,S)                                                            NEG OF THM
         ANSWER(S)
 5   HAS(M,B,REACH(M,B,S))  -REACHABLE(M,B,S)                                          AXIOM
 6   -REACHABLE(MONKEY,BANANAS,S)                                                      FROM 4,5
         ANSWER(REACH(MONKEY,BANANAS,S))
 7   REACHABLE(MONKEY,BANANAS,S)  -AT(BOX,UNDER-BANANAS,S)  -ON(MONKEY,BOX,S)          AXIOM
 8   -AT(BOX,UNDER-BANANAS,S)  -ON(MONKEY,BOX,S)                                       FROM 6,7
         ANSWER(REACH(MONKEY,BANANAS,S))
 9   ON(M,B,CLIMB(M,B,S))  -AT(B,P,S)  -CLIMBABLE(M,B,S)                               AXIOM
10   -AT(BOX,UNDER-BANANAS,CLIMB(MONKEY,BOX,S))  -AT(BOX,P,S)  -CLIMBABLE(MONKEY,BOX,S)  FROM 8,9
         ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))
11   -AT(BOX,UNDER-BANANAS,CLIMB(MONKEY,BOX,S))  -AT(BOX,P,S)                          FROM 3,10
         ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))
12   AT(B,P,CLIMB(M,B,S))  -AT(B,P,S)  -CLIMBABLE(M,B,S)                               AXIOM
13   -AT(BOX,XX1,S)  -AT(BOX,UNDER-BANANAS,S)  -CLIMBABLE(MONKEY,BOX,S)                FROM 11,12
         ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))
14   -AT(BOX,XX1,S)  -AT(BOX,UNDER-BANANAS,S)                                          FROM 3,13
         ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))
15   -AT(BOX,UNDER-BANANAS,X)                                                          FACTOR 14
         ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))
16   AT(B,P2,MOVE(MONKEY,B,P2,S))  -MOVABLE(B)  -AT(B,P1,S)  AT(SK24(S,P2,P1,B),P2,S)  AXIOM
```

```
17   -MOVABLE(BOX)  -AT(BOX,P1,S)  AT(SK24(S,UNDER-BANANAS,P1,BOX),UNDER-BANANAS,S)     FROM 15,16
          ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,MOVE(MONKEY,BOX,UNDER-BANANAS,S))))
18   -MOVABLE(BOX)  AT(SK24(SØ,UNDER-BANANAS,PLACEB,BOX),UNDER-BANANAS,SØ)              FROM 2,17
          ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,MOVE(MONKEY,BOX,UNDER-BANANAS,SØ))))
19   -MOVABLE(BOX)                                                                      FROM 1,18
          ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,MOVE(MONKEY,BOX,UNDER-BANANAS,SØ))))
20   MOVABLE(BOX)                                                                       AXIOM
21   CONTRADICTION                                                                      FROM 19,20
          ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,MOVE(MONKEY,BOX,UNDER-BANANAS,SØ))))

11 CLAUSES LEFT
28 CLAUSES GENERATED
22 CLAUSES ENTERED
```