The UT Interactive Prover

by

W. W. Bledsoe and Mabry Tyson

May, 1975 ATP 17

* The work reported here was supported by NSF Grant #DCR74-12886.

The UT Interactive Prover

W.W. Bledsoe and Mabry Tyson

ABSTRACT

The interactive theorem prover developed by Bledsoe's group at The University of Texas is described. Algorithms are given for its principle routines IMPLY and HOA, and its set of interactive commands are tabulated.

The prover itself (without interaction) is a natural deduction system which uses the concepts of: subgoaling, reductions (rewrite rules), procedures, controlled definition instantiation, controlled forward chaining, conditional rewriting and conditional procedures, algebraic simplification, and induction.

It, or variations of it, have been used to prove theorems in set theory and topology, theorems arising from program verification, and limit theorems of calculus and analysis.

Table of Contents

The UT Interactive Prover

W.W. Bledsoe and Mabry Tyson

1. Introduction

The prover we describe in this paper is a natural deduction type system that proves theorems in first order logic, and some extensions of that by subgoaling, splitting, matching, and rewriting, simplification, and other such procedures. It has been partially described in [1-6] but there remains some uncertainty as to exactly what it does. We will attempt to explain it in a precise manner, but the ultimate explanation is in the LISP program itself, which is available upon request.

There is no attempt here to review all the literature on automatic theorem proving. Suffice it to say that our work is based to a great extent on that of others. The reader is referred to Chang and Lee [7], and Loveland [8] for information and references on resolution type systems, and to the work of Allen and LucKham [9], Guard, et al [10], and Huet [11], on interactive provers. Our prover is in the spirit of Newell, Simon, and Shaw [12], Gelerntner [13], and has much in common with the work of Gentzen [14], Nevins [15-17], Reiter [18], Ernst [19], Bibel [20], Hewitt [21], McDermott and Sussman [22], Wang [23], Maslov [48], and Rulifson, et al [24]. See also Nilsson's Review [26].

In using the interactive prover, the theorem (and subsequent subgoals) are shown on the user terminal's screen in a natural, easy to read form, and the user is provided with several interactive commands (see Section 7) for

communicating with the prover. The prover is based upon natural deduction (or is a Gentzen type system [14-17,25,20,49]), as opposed to a "less natural" system such as resolution. When the human user desired to interact directly with the prover, the dialogue is expressed in terms that are (hopefully) natural and convenient for him. The intent is that the computer will act as a support to the user in the proof of a theorem; although, the machine-only system is a powerful prover in its own right.

The interactive policy of the prover is based on the premise that if the prover can construct a proof it will do so fairly quickly. For each theorem or subgoal, a time limit is set; if a proof has not been constructed in that time, the prover stops and waits for interactive direction. The user then has available a number of commands for displaying the theorem and the details of what the prover has done so far. Using these commands the user isolates the difficulty and then can allocate more time, direct the prover into a new line of reasoning, supply additional information (hypotheses, lemmas, definitions) about the whole thing, or simply assume that the current subgoal is true and go on to another part of the proof. Often proofs of subgoals will fail initially because not enough information has been provided. (Failure may well, of course, be due to attempting to prove a non-theorem). A very useful feature of the prover is that these additional hypotheses need not be stated initially, but rather can be supplied at the point in the proof when it is realized that they are necessary. This prevents the objectionable activity of the user having to prove the theorem himself before he asks the prover to do so, in order to determine what additional hypotheses and definitions will be needed.

This system was developed by Bledsoe's group at The University of Texas. While it is a general theorem prover, earlier versions were mainly exercised on theorems in set theory [2], limit theorems [3,45] and topology [1], and a current version is working on theorems arising from program verification [6]. It has been extended [5,27] to handle these program verification theorems; Larry Fagan and Peter Bruell at Information Sciences Institute, USC, have helped considerably in this extension.

2. IMPLY and HOA

The central routines of PROVER are IMPLY and HOA which are described below. They attempt to establish the validity of an expression of the form

$$(H \longrightarrow C) \ ,$$

(H and C are arguments of IMPLY), by applying a set of (sound) rules (see Tables I and II). These routines are recursive, they call each other and themselves, but the initial call is to IMPLY.

These two algorithms, and their supporting subroutines, form a natural deduction type system. It is like a Gentzen system [14,25], but is more "human like" in that no attempt is made to force the formula being proved into a canonical form. In particular the implication symbol, $\longrightarrow$, is retained, and we believe that the proof proceeds in a manner that would be natural to a mathematician.

IMPLY has five arguments (TYPELIST,H,C,TL,LT) but we will deal with only two of them, H and C at this time. TL and LT are discussed later but TYPELIST is not discussed in this paper. See [27]. HOA has three arguments (B,C,HL) and we will deal with only two of them, B and C, at this time.

When we make a call IMPLY(H,C), the algorithm IMPLY tries to establish the validity of the formula $(H \longrightarrow C)$ by applying a set of (sound) rules. Similarly a call to HOA(B,C) causes the algorithm HOA to try to establish the validity of $(B \longrightarrow C)$.

Actually, neither algorithm is complete[1], but they call upon each other to perform various tasks. IMPLY performs AND-SPLITS, as when the conclusion is a conjunction (Rule 4) or the hypothesis is a disjunction (Rule 3); and HOA handles OR-SPLITS, as when the conclusion is a disjunction (Rule 4) or the hypothesis is a conjunction (Rule 6) or an implication (Rule 7, Back-Chaining). Additionally IMPLY handles various manipulations of the conclusion C, while HOA handles those for the hypothesis B.

A theorem being proved is first sent to IMPLY which calls HOA and itself as needed. Before a formula E is initially sent to IMPLY, it is first converted to quantifier free form (but without converting it to prenex form) by skolemization (see Appendix 1). This (usually) produces skolem variables in E which are replaced by terms during the proof. A substitution θ is derived which consists of these replacements.

If H and C are formulas, then IMPLY either returns NIL or a substitution θ, such that

$$(H\theta \longrightarrow C\theta)^{2}$$

---

[1] Even the combination of both of them working together is not complete, in that there are valid formulas which PROVER cannot prove. See Appendix 2.

[2] Sometimes when multiple substitutions are necessary the implication $(H\theta \longrightarrow C\theta)$ is not valid, even though $(H \longrightarrow C)$ is. See Appendix 3.

is valid (usually a theorem in propositional logic). θ is usually the most general such substitution. If no substitution is needed them IMPLY returns "T". It will return "NIL" if $(H \longrightarrow C)$ is not valid or if it cannot find a proof in the prescribed time limit.

Similarly HOA and many of the supporting routines such as UNIFY return substitutions θ.

The routines IMPLY and HOA are described in algorithmic form in Tables I and II. These tables give only the basic rules of IMPLY and HOA. Some additional details are mentioned in footnotes and in the later descriptions.

A formula E is initially sent to IMPLY by a call IMPLY(NIL,E).

Table I

ALGORITHM

IMPLY(H, C)

	IF	ACTION	RETURN
1.	C ≡ "T" or H ≡ "FALSE"		"T"
2.	TYPELIST*		
3.	H ≡ (A ∨ B)[3]		IMPLY(NIL, (A ⟶ C) ∧ (B ⟶ C))
4.	(AND-SPLIT) C ≡ (A ∧ B)	Put θ: = IMPLY(H,A)	
4.1	θ ≡ NIL		NIL
4.2	θ ≠ NIL	Put λ: = IMPLY(H,Bθ)[4]	
4.3	λ ≡ NIL		NIL
4.4	λ ≠ NIL		θ ∘ λ[5]
5.	(REDUCE)	Put H: = REDUCE(H) Put C: = REDUCE(C)	
5.1	C ≡ "T" or H ≡ "FALSE"	Go to 1	
5.2	H ≡ (A ∨ B)	Go to 3	
5.3	C ≡ (A ∧ B)	Go to 4	
5.4	ELSE	Go to 6	

*See [27].

[3]By the expression "H ≡ (A∨B)" we mean that H has the form "A∨B". Rules 4 and 3 are called "AND-SPLIT's". See [2] and [19].

[4]If θ has two entries, a/x, b/x with a ≠ b, then two λ's, λ_1 and λ_2 are computed, one for each case, and $\lambda_1 \circ \lambda_2$ is returned for λ.

[5]This is just (APPEND θλ). If θ has an entry a/x and λ has an entry b/x where a ≠ b, then leave both values in θ ∘ λ. For example, if θ = (a/x b/y), λ = (c/x d/z) then θ ∘ λ = (a/x b/y c/x d/z).

IMPLY(H,C) Cont'd

	IF	ACTION	RETURN
6.	$C \equiv (A \vee B)$		HOA(H,C)
7.	(PROMOTE) $C \equiv (A \longrightarrow B)$		IMPLY$(H \wedge A, B)$[6]
7.1	Forward Chaining		
7.2	PEEK forward chaining		
8.	$C \equiv (A \longleftrightarrow B)$		IMPLY(H, $(A \longrightarrow B) \wedge (B \longrightarrow A)$
9.	$C \equiv (A = B)$	Put θ: = UNIFY(A,B)	
9.1	$\theta \not\equiv$ NIL		θ
9.2	$\theta \equiv$ NIL	Go To 10	
10.	$C \equiv (\sim A)$		IMPLY$(H \wedge A$, NIL)
11.	INEQUALITY*		
12.	(call HOA)	Put θ: = HOA(H,C)	
12.1	$\theta \not\equiv$ NIL		θ
12.2	(PEEK) $\theta \equiv$ NIL	Put PEEK[7] light "ON" Put θ: = HOA(H,C)	
12.3	$\theta \not\equiv$ NIL		θ
12.4	$\theta \equiv$ NIL	Go To 13	

[6] Actually we call IMPLY(OR-OUT$(H \wedge A)$, AND-OUT(B)). See p. 17.

[7] See p.30. The PEEK Light is turned off at the entry to IMPLY.

IMPLY(H,C) Cont'd

	IF	ACTION	RETURN
13.	(Define C)	Put C': = DEFINE(C)	
13.1	C' ≡ NIL	Go To 14	
13.2	C' ≠ NIL		IMPLY(H,C')
14.	(See Section 2 of [27])		
15.	ELSE		NIL

Table II

ALGORITHM

HOA(B,C)

	IF	ACTION	RETURN
1.	Time limit Exceeded		NIL
2.	(MATCH)	Put θ: = UNIFY(B,C)	
2.1	$\theta \neq$ NIL		θ
2.2	PEEK (See Section 4)		HOA(B,C)
3.	PAIRS (See Section 4)		
4.	(OR-SPLIT) $C \equiv (A \vee D)$	Put C': = AND-OUT(C)	
4.1	$C' \neq C$		IMPLY(H,C')
4.2	$C' \equiv C$	Put θ: = HOA($B \wedge \sim D, A$)[8]	
4.3	$\theta \neq$ NIL		θ
4.4	$\theta \equiv$ NIL		HOA($B \wedge \sim A, D$)[8]
5.1	$C \equiv (A \longrightarrow D)$		IMPLY(B,C)
5.2	$C \equiv (A \wedge D)$		IMPLY(B,C)
6.	$B \equiv (A \wedge D)$	Put θ: = HOA(A,C)	
6.1	$\theta \neq$ NIL		θ
6.2	$\theta \equiv$ NIL		HOA(D,C)

[8] In Step 4.2, the "$\sim$" in $(\sim D)$ is pushed to the inside; e.g., $\sim(\sim P)$ goes to P, and $\sim(P \longrightarrow Q)$ goes to $P \wedge \sim Q$. If D contains no "$\sim$" or "$\longrightarrow$" then $(\sim D)$ is omitted and the call is made HOA(B,A). Similarly in Step 4.4.

HOA(B,C) Cont'd

	IF	ACTION	RETURN
7.	(Back-chaining) $B \equiv (A \longrightarrow D)$	Put θ: = ANDS(D,C)*	
7.1	$\theta \equiv$ NIL	Go To 7E	
7.2	$\theta \not\equiv$ NIL	Put λ: = IMPLY(H, Aθ)[4]	
7.3	$\lambda \equiv$ NIL	Go To 8	
7.4	$\lambda \not\equiv$ NIL		$\theta \circ \lambda$
7E.	$B \equiv (A \longrightarrow a = b)$	Put θ: = HOA(a = b, C)	
7E.1	$\theta \equiv$ NIL		NIL
7E.2	$\theta \not\equiv$ NIL	Put λ: $\equiv$ IMPLY(H, Aθ)[4]	
7E.3	$\lambda \equiv$ NIL	Go To 8	
7E.4	$\lambda \not\equiv$ NIL		$\theta \circ \lambda$
8.	$B \equiv (A \longleftrightarrow D)$		HOA($(A \longrightarrow D) \wedge (D \longrightarrow A)$, C)
9.	$B \equiv (a = b)$	Put Z: = MINUS-ON(a,b)	
9.1	$Z \equiv 0$		NIL
9.2	Z is a number		T
9.3	Z is not a number	Put a': = CHOOSE(a,b), b': = OTHER(a,b) (see p. 20)	
		Put H': = H(a'/b'), C': = C(a'/b')	IMPLY(H', C')
10.	$B \equiv (A \vee D)$		IMPLY(B,C)
11.	$B \equiv \sim A$		IMPLY(H, $A \vee C$)[8]
12.	ELSE		NIL

*ANDS is explained on p.15.

[8]Actually we use AND-PURGE(H,~A) instead of H, which removes ~A from H.

When proving a theorem of the form

$$(H \longrightarrow A \wedge B)$$

IMPLY uses Rule 4 to split it into the two subgoals

$$(H \longrightarrow A)$$

and

$$(H \longrightarrow B)$$

which it tries to prove separately. It is (of course) necessary that the substitution θ derived for $(H \longrightarrow A)$ be applied to B (but not to H) in proving the second subgoal, $(H \longrightarrow B\theta)$.[9]

The fourth argument, TL, of IMPLY is a "theorem label" (or more appropriately, a "subgoal label"), which is a sequence of 1's and 2's that indicate the progress that has been made in proving the theorem. For example, a theorem

$$(H \longrightarrow C_1 \wedge C_2)$$

would have theorem label (1) and its two principal subgoals

$$(H \longrightarrow C_1) \quad \text{and} \quad (H \longrightarrow C_2)$$

would have theorem labels (1 1) and (1 2). Such theorem labels are exhibited in the left margin for the examples given in this paper. In addition to 1's and 2's we also utilize other letters such as H, P, and =, to indicate other actions of the prover.

---

[9] The reader can see the necessity of this rule by considering the three examples $(P(a) \wedge Q(a) \longrightarrow P(x) \wedge Q(x))$, $(P(a) \wedge Q(b) \longrightarrow P(x) \wedge Q(x))$, and $(P(x) \longrightarrow P(a) \wedge P(b))$, where x is a skolem variable, and a and b are constants.

Some Examples

Ex. 1. $(A \longrightarrow A)$

A call is made to

$$\text{IMPLY}(\text{NIL},\ A \longrightarrow A)$$

which in turn uses Rule 7 to call

$$\text{IMPLY}(A, A)$$

which uses Rule 11 to call

$$\text{HOA}(A, A)$$

which returns "T" by HOA Rule 2.

In order to shorten the presentation of this example and those that follow, we will use the notation

(TL) $(D \Rightarrow C)$

in place of IMPLY(D,C) and HOA(D,C).

Thus Ex. 1 becomes

(1) $(\text{NIL} \Rightarrow (A \longrightarrow A))$

(1) $(A \Rightarrow A)$ I 7
Returns "T" I 11, H 2

The theorem label, which is (1) in this case, will be exhibited in the left margin, and some rule numbers from Tables I and II will be given in the right margin, with the prefix I for Table I and the Prefix H for Table II.

Ex. 2. $\forall a(\ \forall x\ \ P(x) \longrightarrow P(a))$.

(1) $(NIL \Rightarrow (P(x) \longrightarrow P(a_o)))$ Skolemized

(1) $(P(x) \Rightarrow P(a_o))$ I 7

UNIFY$(P(x), P(a_o))$ returns a_o/x H 2

Henceforth we will drop "NIL $\Rightarrow$" and write "A" instead of "NIL $\Rightarrow$ A". Thus Ex. 2 becomes

(1) $(P(x) \longrightarrow P(a_o))$

(1) $(P(x) \Rightarrow P(a_o))$ I 7

Returns a_o/x H 2

ANDS.

In the following example we use the algorithm ANDS. It is a mini version of IMPLY which handles only theorems of the form

$$(H_1 \wedge H_2 \wedge \ldots \wedge H_n \longrightarrow C)$$

where $(H_i\theta = C\theta)$ for some θ. (In which case θ is returned).

Ex. 3. $\forall a(P(a) \wedge \forall x(P(x) \longrightarrow Q(x)) \longrightarrow Q(a))$.

(1) $(P(a_o) \wedge (P(x) \longrightarrow Q(x)) \longrightarrow Q(a_o))$

(1)	$(P(a_o) \wedge (P(x) \longrightarrow Q(x)) \Rightarrow Q(a_o))$	I 7
	$(P(a_o) \Rightarrow Q(a_o))$	H 6
	Returns NIL	
	$((P(x) \longrightarrow Q(x)) \Rightarrow Q(a_o))$	H 6.1
	ANDS$(Q(x), Q(a_o))$	H 7
	Returns a_o/x	Back-chaining
(1 H)	$(P(a_o) \wedge (P(x) \longrightarrow Q(x)) \Rightarrow P(a_o))$	H 7.2
	Returns "T"	H 6, H 2
	Returns a_o/x for (1)	H 7.2.2.2

Ex. 3'. $(A \vee B \longrightarrow A \vee B)$

(1)	$(A \vee B \Rightarrow A \vee B)$	I 7
(1 1)	$(A \Rightarrow A \vee B)$	I 3, 4, 7
	$(A \Rightarrow A)$	H 4.2, Footnote 5
	"T"	H 2
(1 2)	$(B \Rightarrow A \vee B)$	I 4.2
	"T"	H 4.2, H 2

Ex. 3". $(A \longrightarrow B \vee C)$ (Not a theorem)

In this example if we applied HOA Step 4.2 without the footnote we would obtain an indefinite repetition as follows:

(1)	$(A \Rightarrow B \vee C)$		I 7
	$(A \not\Rightarrow \sim C \Rightarrow B)$		H 4.1
	$(A \Rightarrow B)$	NIL	H 6
	$(\sim C \Rightarrow B)$		H 6.2
	$(A \Rightarrow B \vee C)$		H 11
	Repeat		

But by preventing the addition of $\sim C$ to the hypothesis, unless it is fundamentally changed, we eliminate this problem.

(1)	$(A \Rightarrow B \vee C)$		I 7
	$(A \Rightarrow B)$	NIL	H 6
	$(A \Rightarrow C)$	NIL	H 6.2
	NIL is returned for (1).		

AND-OUT is an algorithm which puts expressions in conjunctive form (but does not convert implications).

For example

AND-OUT$(A \vee (B \wedge C))$ returns $((A \vee B) \wedge (A \vee C))$,

AND-OUT$(A \vee (D \longrightarrow B \wedge C))$ returns $(A \vee (D \longrightarrow B \wedge C))$.

Similarly OR-OUT puts expressions in disjunctive form.

Ex. 3'''. $B \longrightarrow A \barwedge (\sim A \vee B)$

This example shows the utility of "AND-OUT" in Rule H4. For without it we would get

(1) $(B \Rightarrow A \barwedge (\sim A \vee B))$ I 7

If we don't use AND-OUT of H4

(1 1) $(B \Rightarrow A)$ Returns NIL

(1 2) $(B \Rightarrow \sim A \barwedge B)$ Returns NIL

Returns NIL for (1)

Since we do use AND-OUT in H4, we get

(1) $(B \Rightarrow A \vee (\sim A \wedge B))$ I 7

(1) $(B \Rightarrow (A \vee \sim A) \wedge (A \vee B))$ H 4

(1) $(B \Rightarrow A \vee B)$ I 4
REDUCE Rules 15, 17

(1 1) $(B \Rightarrow A)$ Returns NIL

(1 2) $(B \Rightarrow B)$ H 4.1

Returns "T" for (1 2) and (1) as desired H 2, H 4. 4

Ex. 3''''. $(A \wedge (\sim A \vee B) \longrightarrow B)$

Similarly OR-OUT is required in I 7. Because without it we would get

(1)	$(A \wedge (\sim A \vee B) \Rightarrow B)$		I 7
(1 1)	$(A \Rightarrow B)$	Returns NIL	H 6
(1 2)	$(\sim A \vee B \Rightarrow B)$		
(1 2 1)	$(\sim A \Rightarrow B)$	Returns NIL	I 3

Returns NIL for (1 2) and (1)

But since we use OR-OUT in I 7 we get

(1)	$(A \wedge (\sim A \vee B) \longrightarrow B)$	Original
(1)	$(\text{OR-OUT}(A \wedge (\sim A \vee B)) \Rightarrow B)$	I 7
	$((A \wedge \sim A) \vee (A \wedge B) \Rightarrow B)$	
(1 1)	$(A \wedge \sim A) \Rightarrow B)$	I 4
	$(\text{FALSE} \Rightarrow B)$	I 5
	"T"	I 1
(1 2)	$(A \wedge B \Rightarrow B)$	I 4.2
	"T"	H 6.2, H 2
	Returns "T" for (1) as desired	I 4.4

Substituting Equals

HOA Rule 9 gives the prover an ability to substitute equals. When an equality unit (a = b) is in the hypothesis, the program uses the algorithm CHOOSE(a,b) to select either a or b, and replaces it by the other in H and C. CHOOSE selects neither if neither a or b occurs in H or C. It selects a if b is a number, and vice versa. It will not choose a if b occurs in a, and vice versa. In the interactive mode the user can enter this decision process (see Section 7).

3. Definitions and Reduction

Definitions.

Rule 12 of IMPLY calls DEFINE(C) which expands definitions from a stored list. Table III gives some such definitions.

When the defining form introduces quantifiers (e.g., Rule 2 of Table III) it is necessary to eliminate these quantifiers by skolemization. We have done this by pre-skolemizing the formula in the table, but it is necessary to store two such skolemizations because the correct one will depend on whether the formula occupies a positive[10] or negative position in the theorem being proved. For example, $(A \subseteq B)$ is replaced by $(x_o \in A \rightarrow x_o \in B)$ in

$$(H \longrightarrow A \subseteq B)$$

whereas it would be replaced by $(x \in A \rightarrow x \in B)$ in

$$(A \subseteq B \longrightarrow C) .$$

[10] See [23, 3] and Appendix 1.

Table III

SOME DEFINITIONS

	Formula Being Defined	Defining Form
1.	$(A = B)$[11]	$(A \subseteq B \wedge B \subseteq A)$
2.	$(A \subseteq B)$	$\forall x(x \in A \rightarrow x \in B)$ Skolem form[12] $(x_o \in A \rightarrow x_o \in B)$ in "Conclusion" $(x \in A \rightarrow x \in B)$ in "Hypothesis"
3.	$(A \cup B)$	$\{x: x \in A \vee x \in B\}$
4.	$(A \cap B)$	$\{x: x \in A \wedge x \in B\}$
5.	$\bigcup_{t \in S} A(t)$	$\{x: \exists t(t \in S \wedge x \in A(t))$[12]
6.	$\bigcap_{t \in S} A(t)$	$\{x: \forall t(t \in S \rightarrow t \in A(t))$[12]
7.	subsets(A)	$\{x: x \subseteq A\}$
7'.	sb(A)	subsets(A)
8.	range f	$\{y: \exists x(y = f(x))\}$
9.	Oc F	(Open F $\wedge$ Cover F)

[11] A different symbol is used for set equality to distinguish it from the arithmetic equality. Here in Entry 1 we mean set equality.

[12] When the defining form introduces quantifiers, two versions of its skolemization are needed. See page 21.

REDUCE

Rule 5 of IMPLY calls REDUCE(H) and REDUCE(C). If E is a formula then a call to REDUCE(E) causes the algorithm REDUCE to apply a set of rewrite rules to convert parts of the formula E. See [2,29-36]. Table IV gives some examples of rewrite rules in use.

REDUCE helps convert expressions into forms which are more easily proved by IMPLY. Also the rewrite table is a convenient place to store facts that can be conveniently used by the machine as they are needed. For example, REDUCE returns "T"(TRUE), when applied to the formulas (Closed(Clsr A)), (Open $\emptyset$), (Open(interior A)), ($\emptyset \subseteq$ A).

Table IV

REDUCE Rewrite Rules

	INPUT	OUTPUT
1.	$(t \in A \cap B)$	$(t \in A \wedge t \in B)$
2.	$(t \in A \cup B)$	$(t \in A \vee t \in B)$
3.	$(t \in \{x: P(x)\})$	$P(t)$
4.	$(t \in A)$ If A has Definition $\{x: P(x)\}$	$P(t)$
5.	$t \in$ subsets(A)	$t \subseteq A$
6.	$t \subseteq A \cap B$	$(t \subseteq A \wedge t \subseteq B)$
7.	$(A \cap A)$	A
8.	$(A \cup A)$	A
9.	$(A \cap \emptyset)$	$\emptyset$
10.	$(A \cup \emptyset)$	A
11.	$(\emptyset \subseteq A)$	"T"
12.	$A \in \{B\}$	$A = B$
13.	(range λ x f(x))	$\{y: \exists\ x(y = f(x)\}$
14.	(Choice A $\in$ A)	$A \neq \emptyset$
15.	$(A \vee \sim A)$	"T"
16.	$(A \wedge \sim A)$	"FALSE"
17.	("T" $\wedge$ A)	A
18.	$(A \wedge$ "T")	A

Table IV (Con't)

	INPUT	OUTPUT
19.	$(A \vee$ "T")	"T"
20.	("T" $\vee$ A)	"T"
21.	$(G \subseteq\subseteq G)$[13]	"T"
22.	$(G \subseteq\subseteq \overline{G})$[13]	"T"
23.	$(A \subseteq A)$	"T"
24.	$(A \subseteq \overline{A})$	"T"
25.	$A \wedge$ FALSE	FALSE
26.	FALSE $\wedge$ A	FALSE
27.	$A \vee$ FALSE	A
28.	FALSE $\vee$ A	A
	etc.	

[13] It need not concern the reader here but $\overline{G}$ is the set of closures of members of G. That is if $\overline{A}$ is the closure of the set A, then $\overline{G} = \{\overline{A}: A \in G\}$. And $(H \subseteq\subseteq J)$ means that H is a refinement of J, that is, each member of H is a subset of a member of J.

Ex. 4. $\forall A \ \forall B \ (A \subseteq A \cup B)$

(1)	$(A_o \subseteq A_o \cup B_o)$	
(1)	$(x_o \in A_o \Longrightarrow x_o \in (A_o \cup B_o))$	I 12
(1)	$(x_o \in A_o \Longrightarrow x_o \in A_o \vee x_o \in B_o)$	I 5 REDUCE Rule 2
(1)	$(x_o \in A_o \Rightarrow x_o \in A_o \vee x_o \in B_o)$	I 7
(1 1)	$(x_o \in A_o \Rightarrow x_o \in A_o)$	H 4.1
(1 1)	"T"	H 2

Return "T" for (1).

Notice how closely this parallels the usual mathematician's proof, i.e.,

$A \subseteq A \cup B$

$(x \in A \longrightarrow x \in (A \cup B))$

$(x \in A \longrightarrow x \in A \vee x \in B)$

TRUE.

Ex. 5. $\forall A \; \forall B$ (subsets $(A \cap B)$ = subsets $(A) \cap$ subsets (B))

(1) $\text{subsets}(A_o \cap B_o) = \text{subsets}(A_o) \cap \text{subsets}(B_o)$

We will here contract "subsets" to "sb" and drop the subscripts.

(1) $sb(A \cap B) = sb(A) \cap sb(B)$

(1) $[sb(A \cap B) \subseteq sb(A) \cap sb(B)] \wedge [sb(A) \cap sb(B) \subseteq sb(A \cap B)]$ I 12
Definition 1

(1 1) $[sb(A \cap B) \subseteq sb(A) \cap sb(B)]$ I 4
This is an AND-SPLIT

(1 1) $[t_o \in sb(A \cap B) \Longrightarrow t_o \in (sb(A) \cap sb(B))]$ I 12
Definition 2

(1 1) $[t_o \subseteq A \cap B \Longrightarrow t_o \in sb(A) \wedge t_o \in sb(B)]$ I 5
REDUCE Rules 5, 1

(1 1) $[t_o \subseteq A \wedge t_o \subseteq B \Rightarrow t_o \subseteq A \wedge t_o \subseteq B]$ I 5, I 7
REDUCE Rules 6, 5

Return "T" for (1 1) I 4, H 6, H 2

(1 2) $[sb(A) \cap sb(B) \subseteq sb(A \cap B)]$
Return "T" for (1 2) (Similarly)
Return "T" for (1) .

It should be noted that the use of Definitions and REDUCE on this example has eliminated the need for additional hypotheses (or axioms). The required hypotheses must be given by the user but they are given once and for all in REDUCE and definition tables and never used except when needed in the proof. An ordinary resolution proof or Gentzen type proof which did not use such mechanisms would require four additional axioms and a lengthy proof.

1. $(\alpha = \beta \longleftrightarrow \forall t(t \in \alpha \leftrightarrow t \in \beta))$
2. $(t \in A \cap B \leftrightarrow t \in A \wedge t \in B)$
3. $(t \in \text{subsets } A \leftrightarrow t \subseteq A)$
4. $(t \subseteq A \cap B \leftrightarrow t \subseteq A \wedge t \subseteq B)$.

Rule 4 of Table IV is a conditional rule. When attempting to convert a formula of the form $t \in A$, the algorithm REDUCE first checks to see if A has a definition of the form $\{x: P(x)\}$, in which case it (in effect) instantiates that definition and applies Rule 3. For example the expression

$$x_o \in \bigcup_{t \in Q} A(t)$$

is reduced by Rule 4 of Table IV and Rule 5 of Table III, to

$$\exists t(t \in Q \wedge x_o \in A(t))$$

(or actually to the skolemized form $(t \in Q \wedge x_o \in A(t))$).

Ex. 6. . $(A \in G \longrightarrow A \subseteq \bigcup_{B \in G} B)$

(1)	$(A_o \in G \Rightarrow A_o \subseteq \bigcup_{B \in G} B)$	I 7
(1)	$(A_o \in G \Rightarrow (t_o \in A_o \longrightarrow t_o \in \bigcup_{B \in G} B))$	I 12 Definition 2
(1)	$(A_o \in G \Rightarrow (t_o \in A_o \longrightarrow B \in G \wedge t_o \in B)$	I 5 REDUCE Rule 4, Definition 5
(1)	$(A_o \in G \wedge t_o \in A_o \Rightarrow B \in G \wedge t_o \in B)$	I 7
(1 1)	$(A_o \in G \wedge t_o \in A_o \Rightarrow B \in G)$	I 4
	Returns A_o/B for (1 1)	H 6.1, H 2
(1 2)	$(A_o \in G \wedge t_o \in A_o \longrightarrow t_o \in A_o)$	I 4.2
	Returns "T" for (1 2)	H 6.2, H 2
	Returns A_o/B for (1)	I 4.4

4. PEEKing and Forward Chaining

PEEK.

We saw on page 21 that when all else fails, we expand the definition of the conclusion C. Such is not the case for the hypothesis H. However, when proving $(B \longrightarrow C)$, the algorithm HOA sometimes "peeks" at the definition of B to see if it has the potential of helping with the proof of C, and if so it then (temporarily) expands that definition. This is done after a regular call to HOA has failed and the "peek light" has been turned on.

To facilitate this, the program has a PEEK property list for each of the main predicates. Table V gives some of its entries. This enables the program to quickly check whether an expansion of the definition of B would have a chance of helping with the proof.

Table V
PEEK Property Lists

1. (Oc [Open Cover])

2. (Reg [Subset Open Clsr])

etc.

Ex. 7. $(Reg \wedge Oc\ F \longrightarrow \exists G(Cover\ G))$

(1) $(Reg \wedge Oc\ F_o \Rightarrow Cover\ G)$ I 7

HOA is called at Step 12 of IMPLY and fails; then the PEEK light is turn ON.

(1) $(Reg \wedge Oc\ F_o \Rightarrow Cover\ G)$ I 11.2

(1 1) $(Reg \Rightarrow Cover\ G)$ NIL H 6

(1 2) $(Oc\ F_o \Rightarrow Cover\ G)$ H 6.2

$((Open\ F_o \wedge Cover\ F_o) \Rightarrow Cover\ G)$ H 2.2 (PEEK) Table V, Entry 1.

F_o/G is returned for (1 2) and (1).

Notice that it did not expand the definition of Reg in (1 1), i.e.,

(1 1) $(Reg \Rightarrow Cover\ G)$

because in Rule 2 of Table V, "Reg" did not have "Cover" on its PEEK property list.

After such a use of PEEK, the expanded definition is not retained the original form Oc F_o is retained for any further proofs that may be required. This permits the proofs to proceed at a high level where possible, and resorting to expanded definitions only when necessary. It also facilitates human understanding when operated in a man-machine mode.

Forward Chaining.

In IMPLY Rule 7, when a new hypothesis is added to H we try to "forward chain" with it. Forward chaining is another name for modus ponens: If $P'\theta = P\theta$, then a hypothesis

$$P' \wedge (P \longrightarrow Q)$$

is converted into

$$P' \wedge (P \longrightarrow Q) \wedge Q\theta \quad .$$

Ex. 8. $\forall a(P(a) \wedge \forall x(P(x) \longrightarrow Q(x)) \longrightarrow Q(a))$

(1) $(NIL \Rightarrow (P(a_o) \wedge (P(x) \longrightarrow Q(x)) \longrightarrow Q(a_o)))$

$(P(a_o) \wedge (P(x) \longrightarrow Q(x)) \wedge Q(a_o) \Rightarrow Q(a_o))$ I 7, 7.1 forward chaining

Returns "T".

It should be noted that this is Example 3 which was proved earlier using Rule H 7 (Back-Chaining). Forward chaining is an option which is available

to the user. In some instances he may want to control its use. For example, forward chain with $P(x_o)$ only when $P(x_o)$ is a ground formula, or forward chain with an atom $P(x)$ only when P is a member of a predescribed list. Limited forward chaining has been used in a powerful way by Bundy [37], Ballantyne and Bennett [38,39], Nevins [17], Reiter [18], Siklossy et al [36], and others.

PEEK forward chaining.

If $P'\theta = P\theta$, A has the definition $(P \longrightarrow Q)$ then a hypothesis

$$P' \wedge A$$

is converted into

$$P' \wedge A \wedge Q\theta$$

Ex. 9. $(A \subseteq B \wedge B \subseteq C \longrightarrow A \subseteq C)$

(1) $(A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C)$ I 7

We have dropped the subscripts of A_o, B_o and C_o in this example.

$(A \subseteq B \wedge B \subseteq C \Rightarrow (t_o \in A \longrightarrow t_o \in C))$ I 12

Definition 2

$(A \subseteq B \wedge B \subseteq C \wedge t_o \in A \Rightarrow t_o \in C)$ I 7

$(A \subseteq B \wedge B \subseteq C \wedge t_o \in A \wedge t_o \in B \wedge t_o \in C \Rightarrow t_o \in C)$ I 7.2

PEEK forward chaining

Returns "T" .

In the above, $(t_o \in A)$ was PEEK forward chained into $(A \subseteq B)$ by expanding the definition of $(A \subseteq B)$ to

$$(t \in A \longrightarrow t \in B)$$

and matching $(t \in A)$ to $(t_o \in A)$ with t_o/t, getting $(t_o \in B)$ as a result. Then $(t_o \in B)$ was PEEK forward chained into $(B \subseteq C)$ getting $(t_o \in C)$. The program has a checking mechanism to prevent an infinite continuation in adverse cases.

Ex. 9. $(A \subseteq B \wedge \bar{B} \subseteq C \wedge \forall D \forall E (D \subseteq E \longrightarrow \bar{D} \subseteq \bar{E}) \longrightarrow \bar{A} \subseteq C)$

(1) $(A_o \subseteq B_o \wedge \bar{B}_o \subseteq C_o \wedge \overbrace{(D \subseteq E \longrightarrow \bar{D} \subseteq \bar{E})}^{\alpha} \longrightarrow \bar{A}_o \subseteq C_o)$

When Rule I 7 is applied it forward chains $(A_o \subseteq B_o)$ into α to get $(\bar{A}_o \subseteq \bar{B}_o)$. A control is used to prevent repeated use of α to get, $\bar{\bar{A}}_o \subseteq \bar{\bar{B}}_o$, etc.

(1) $(A_o \subseteq B_o \wedge \bar{B}_o \subseteq C_o \wedge \alpha \wedge \bar{A}_o \subseteq \bar{B}_o \Rightarrow \bar{A}_o \subseteq C_o)$ I 7

(" $\Rightarrow (t_o \in \bar{A}_o \longrightarrow t_o \in C_o))$ I 12, Definition 2

$(A_o \subseteq B_o \wedge \bar{B}_o \subseteq C_o \wedge \alpha \wedge \bar{A}_o \subseteq \bar{B}_o \wedge t_o \in \bar{A}_o \wedge t_o \in \bar{B}_o \wedge t_o \in C_o$

$\longrightarrow t_o \in C_o)$

In the above application of Rule I 7, $(t_o \in \bar{A}_o)$ was forward chained into $(\bar{A}_o \subseteq \bar{B}_o)$ to obtain $(t_o \in \bar{B}_o)$, which is turn was forward chained into $(\bar{B}_o \subseteq C_o)$ to obtain $(t_o \in C_o)$

(" $\wedge t_o \in C_o \longrightarrow t_o \in C_o)$ "T"

Ex. 9A. $(Oc\ F \wedge \forall F\ \exists G(Oc\ F \longrightarrow Cover\ G \wedge \overline{G} \subseteq\subseteq F) \longrightarrow \exists H(H \subseteq\subseteq F))$[13]

(1) $(Oc\ F_o \wedge (Oc\ F \longrightarrow Cover\ G(F) \wedge \overline{G(F)} \subseteq\subseteq F) \longrightarrow H \subseteq\subseteq F_o)$

$(Oc\ F_o \wedge (Oc\ F \longrightarrow Cover\ G(F) \wedge \overline{G(F)} \subseteq\subseteq F) \wedge Cover\ G(F_o) \wedge \overline{G(F_o)} \subseteq\subseteq F_o \Rightarrow H \subseteq\subseteq F_o)$ I 7

Forward chaining

Returns $\overline{G(F_o)}/H$.

Ex. 9B. $(Oc\ F \wedge Reg \longrightarrow \exists H(H \subseteq\subseteq F))$

(1) $(Oc\ F_o \wedge Reg \wedge Cover\ G(F_o) \wedge \overline{G(F_o)} \subseteq\subseteq F_o \Rightarrow H \subseteq\subseteq F_o)$ I 7

Here $Oc\ F_o$ has been PEEK Forward Chained into Reg which has the definition

$$\forall F\ \exists G(Oc\ F \longrightarrow Cover\ G \wedge \overline{G} \subseteq\subseteq F)$$

which has skolem form (in this case)

$$(Oc\ F \longrightarrow Cover\ G(F) \wedge \overline{G(F)} \subseteq\subseteq F).$$

As in the previous example $\overline{G(F_o)}/H$ is returned.

5. Conditional Rewriting and Conditional Procedures

Conditional Rewrite Rules.

In Section 3 we described the REDUCE feature which causes various formulas (or subformulas) to be rewritten. For example, the expression

$$t \in A \cap B$$

is rewritten as

$$(t \in A \wedge t \in B) .$$

Sometimes we wish such a conversion to be made only if a certain condition is satisfied. Such rules, are called "conditional rewrite rules", and are added to the REDUCE table in the form

$$(* \; P \; A \; B) .$$

The program upon detecting the *, checks the validity of P before rewriting B for A (with proper instantiation). If P is not true then A is not rewritten. The * is placed there to distinguish conditional rules from ordinary REDUCE rules. For example, the entry

$$(* \; A \neq \text{NULL} \; \text{NODES}(A) \; \text{NODES}(\text{LEFT}(A)) + \text{NODES}(\text{RIGHT}(A)))$$

means that NODES(A)† can be "reduced" to NODES(LEFT(A)) + NODES(RIGHT(A)) if A ≠ NULL. The rewrite rule is not valid if A = NULL because LEFT(NULL) and RIGHT(NULL) are not defined, thus the rewrite rule is applicable only

†NODES(T) is one plus the number of nodes in a binary tree T. NODES(NULL) = 1
LEFT(T) is the left-hand son of T.

only if $A \neq NULL$ is known. Notice also that the result of the rewrite rule contains forms to which the rewrite rule could be applied. This would result in an infinite expansion normally but the condition on the rewrite rule precludes this. Generally this rule would be used once and then it would not be known if $LEFT(A) \neq NULL$ or if $RIGHT(A) \neq NULL$ so the rule would not be applied again.

Rewrite rules are expected to be applied quickly or not at all. Their power lies in the quickness with which they can be applied. Accordingly we avoid long drawn-out procedures for checking the validity of P. For example we do not call IMPLY itself to check P. Rather we have a "mini" version of IMPLY, for this purpose, which includes ANDS (See p. 15), which we call QK-IMPLY.

A similar remark can be made for conditional procedures described below.

Conditional Procedures.

Some procedures are conditional in that they are initiated only when certain conditions are satisfied. Examples of these are PAIRS described below, INDUCTION described on page 58 below and in [2], and the limit heuristic described in [3]. See also [40,29].

PAIRS.

Sometimes in HOA the expressions C and B will not unify even though the main predicates of C and B are the same. For example,

$$(G_o \subseteq\subseteq F_o \Rightarrow H_o \subseteq\subseteq J_o)^{13}.$$

In this case, at Step 3 of HOA, the algorithm consults the PAIRS property list of "$\subseteq \subseteq$" for advice. That property list may (or may not) list one or more subgoals that can be proved to establish the given goal. Table VI gives some such entries.

Table VI

PAIR Property Lists

1. (Cover (Cover G $\longrightarrow$ Cover F)[(G $\subseteq\subseteq$ F) ( )$\cdots$])

2. ($\subseteq\subseteq$[13] (G $\subseteq\subseteq$ F $\longrightarrow$ H $\subseteq\subseteq$ J)
   [(H $\subseteq\subseteq$ G $\wedge$ F $\subseteq\subseteq$ J)( )$\cdots$])

3. (Lf[14] (Lf G $\longrightarrow$ Lf F)[(F = $\overline{\text{G}}$)])

4. (countable (countable A $\longrightarrow$ countable B)
   [$\exists$ f(f is a function $\wedge$ domain f $\subseteq$ A $\wedge$ B $\subseteq$ range f)
   (B $\subseteq$ A)$\cdots$]

etc.

[14] Lf G means that G is locally finite. That is, at any point x, there is an open set A which intersects only a finite number of members of G.

Ex. 10. $(G \subseteq\subseteq F \longrightarrow G \subseteq\subseteq \overline{F})$

(1) $(G_o \subseteq\subseteq F_o \Rightarrow G_o \subseteq\subseteq \overline{F}_o)$ I 7

$(G_o \subseteq\subseteq G_o) \wedge (F_o \subseteq\subseteq \overline{F}_o)$ H 2.3

PAIRS Entry 2

(1 1) $(G_o \subseteq\subseteq G_o)$

"T" I 5

Reduce Rule 21

(1 2) $(F_o \subseteq\subseteq \overline{F}_o)$

"T" I 5

Reduce Rule 22

Notice that the PAIRS Rule H 3 has converted the goal (1) into a subgoal that is easily proved by the REDUCE rules 21 and 22.

REDUCE and PAIRS act a lot alike in that they change one goal into another, the difference being that REDUCE acts on a "single entry" (i.e., a given formula is rewritten as another), while PAIRS acts on a double entry. However, that double entry requires that the two input formulas be partially matched (their main predicates are identical).

Such a pairs concept can be extended to include pairs of predicates that are <u>not</u> identical, but that has not been done for the present algorithms.

In general we favor procedure which are triggered by easy to check conditions.

Ex. 11. Th. (g is a function) $\wedge$ countable (domain g)

$\wedge$ A $\subseteq$ range g $\longrightarrow$ countable A

(1) (g_o is a function) $\wedge$ countable (domain g_o)

$\wedge$ $A_o \subseteq$ range $g_o \Rightarrow$ countable A_o I 7

countable (domain g_o) $\Rightarrow$ countable A_o H 6.2

(1 P) (g_o is a function) $\wedge$ $A_o \subseteq$ range g $\Rightarrow$ ((f is a function)

$\wedge$ (domain f $\subseteq$ domain g_o) $\wedge$ ($A_o \subseteq$ range f)) PAIRS Entry 4

(1 P 1) (g_o is a function) $\wedge$ $A_o \subseteq$ range $g_o \Rightarrow$ (f is a function)

g_o/f

(1 P 2) (g_o is a function) $\wedge$ $A_o \subseteq$ range g_o

$\Rightarrow$ (domain $g_o \subseteq$ domain g_o) $\wedge$ ($A_o \subseteq$ range g_o)

(1 P2 1) (") $\Rightarrow$ (domain $g_o \subseteq$ domain g_o)

"T" by REDUCE Rule 23

(1 P2 2) (g_o is a function) $\wedge$ $A_o \subseteq$ range $g_o \Longrightarrow A_o \subseteq$ range g_o.

"T"

So g_o/f is returned for (1 P) and for (1).

6. Complete Sets of Reductions

The use of rewrite rules as in our REDUCE procedure is a very powerful device. It is extremely more efficient than ordinary substitution of equals as is used in Paramodulation or in HOA Rules 9 and 7E, because the latter allows substitution both ways. Thus it is highly desirable to get as many entries as possible in the REDUCE table and to remove the corresponding equality units from the hypotheses.

The questions that naturally arise are: How far can you go with rewrite rules? Can such a system be made complete in some sense? How do we choose the entries for the REDUCE table? Can we generate all needed REDUCE table entries from a few key ones?

Very general, although incomplete, answers to these questions are given by a beautiful paper of Lankford [30] which is based on pioneering work of Knuth and Bendix [31] and some earlier work of Slagle [32].

The reader is referred to [30] for details but the general idea is that some theories, such as group theory, allow a "complete set of reductions." For example, there exists a set of entries for a REDUCE table which handles all equality substitutions for the equational axioms of group theory. A very powerful algorithm is given which often generates a complete set of reductions from the axioms of a given equational theory. One problem with the concept of the rewrite rule currently in vogue is that it does not allow commutative axioms to be included in a REDUCE table since, for example, the rewrite rule $x \cdot y \longrightarrow y \cdot x$ when applied to $a \cdot b$ produces the infinite sequence of rewrites $a \cdot b$, $b \cdot a$, $a \cdot b$, $b \cdot a, \ldots$. However, Lankford [30] has shown how commutative theories, such as

commutative, groups, rings, Boolean algebras, and modules over rings, which allow no complete sets of reductions, can nevertheless be treated efficiently and in a complete way with most of the equality units in a REDUCE table. Earlier, Bledsoe, et al [3] used such a decision procedure for ring theory as the basis of a heuristic approximation of an unavailable decision procedure for field theory with encouraging results.

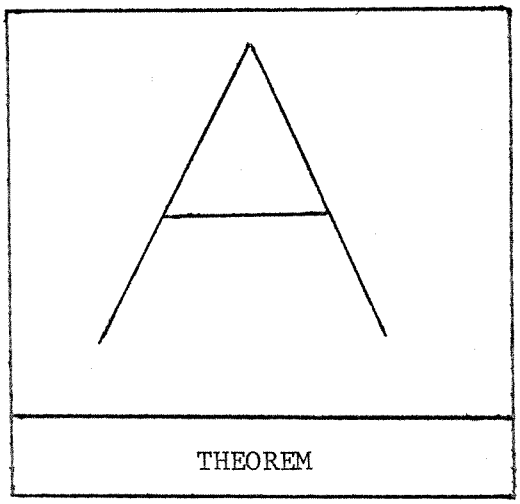
Table IV shows only a few of the REDUCE rules used by our prover, and many others can be easily added (see for example, ADD-REDUCE in Section 6). The largeness of the table does not impede the speed of its use because hash code techniques can be employed.

As pointed out earlier, the REDUCE table is a convenient place to store facts that may be needed at some point in a proof but which will never be accessed until actually needed. If these same facts were made part of the hypothesis they would greatly clutter up and slow down the operation of the prover.

7. Interactive System

Large Data base problem.

One of the irksome things about most automatic theorem proving systems, is that the human user has to prove the theorem before he asks the computer to do so.



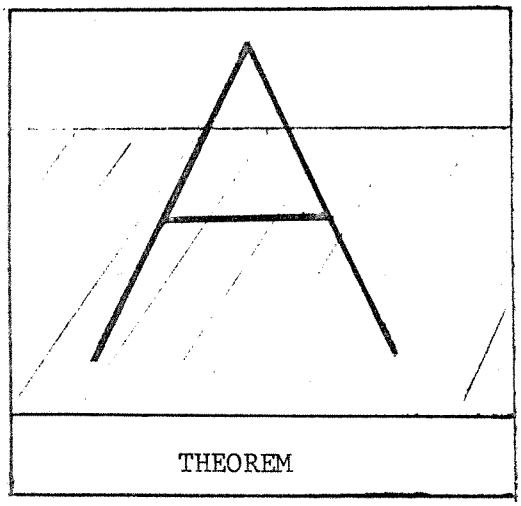
In this figure we depict a theorem to be proved, along with the "Axioms" or reference theorems needed for the proof. If we don't list enough reference theorems then the automatic prover cannot succeed; on the other hand, if we list too many, the prover again cannot succeed because it will be overwhelmed with too much data. So we cannot just list "all known theorems" as hypothesis and expect success, because most provers will then hang up on computing many useless inferences (lemmas) which have nothing to do with the objective at hand.[15]

---

[15] A few recent programs have attacked the large data base problem [18, 41, 42] with some success.

Thus in order to determine exactly the correct reference theorems needed, the human user is forced to prove the theorem first.

We have partially eliminated this problem by storing information in the form of definitions, rewrite (REDUCE) tables, and procedures, which are used only as needed and in no way clutter up the system (see Section 3).



Enter as needed

Rewrite Rules
(REDUCE)
Definitions
Procedures
Conditional Rewrite
Rules and
Procedures
(PAIRS, INDUCTION)

The remainder of the difficulty is eliminated by having the human user insert references theorems only as they are needed, during the actual proof. Of course he will have to know when to do this, and what to insert.

Hard Theorems.

Equally irksome is the fact that present programs cannot prove very hard theorems. So they don't get involved with very interesting mathematics, and don't come to grips with some of the problems that the computer will have to face if we are to have acceptable computer mathematics.

Man-Machine.

For these reasons, and others, we have decided to include the theorem prover described in Sections 1-6 above, as part of a man-machine interactive, prover.

The System.

The system consists of one or more interactive computer terminals connected to a large digital computer. At present we are using the CDC 6600 and PDP-10 computers at The University of Texas, and the UT time sharing system. A version of the program is also running on the DEC 10 computer at the Information Sciences Institute, USC, Los Angeles.

The system was developed at UT, MIT, and ISI, by the authors, and Bob Boyer, Robert Anderson, Peter Bruell, Mike Ballantyne, Bill Bennett and Larry Fagan.

This system has much in common with earlier programs of Guard, et al [10], Allen and Luckham [9] (especially with recent additions [33]), and Huet [11], but is quite different, (e.g., in its use of DETAIL, PUT, etc., defined below and does not use Resolution).

User Requirements.

We believe that such a system must be built for the convenience of the user and not the programmer. For otherwise, the system will not be used. As long as the pain in using the system exceeds the help obtained, the potential user will stay away.

In order to interact effectively, the user must be able to

(1) Read and easily comprehend the scope

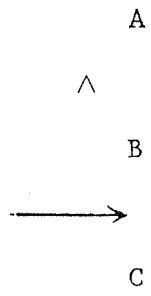
(2) Follow the proof

(3) Help the computer only when needed

(4) Use convenient commands

The UT interactive Prover.

In our system the formulas which are being proved appear on the terminal screen in an infix notation. For example, the formula whose internal representation is $(\longrightarrow (\wedge\ A\ B)\ P)$ would appear on the screen as



Larry Fagan has recently developed a package for the DEC 10 at ISI to allow a formula to remain stationary on the scope while other material is

scrolled up in a normal fashion. He and Mabry Tyson have adopted it to also operate on the CDC 6600 at UT. Having the theorem (or current subgoal) remain stationary on the scope during the proof is a great help to the user in understanding and following the proof.

The user has at his disposal a set of options which give (interactive) commands to the program. Some of these cause information to be displayed on the terminal screen, while others affect the course of the proof.

Table VII gives a listing of some of the interactive commands being used. A few of these are further explained below. In the following, the work "theorem" is used to represent the current subgoal being proved.

Most of the human input (i.e., the use of the interactive commands) takes place at "IMPLY STOP", a point in the routine IMPLY, near its beginning. The program is a slave to the user, working on tasks assigned it by the user. It halts at IMPLY STOP and reports after such a calculation. It may report

"PROVED"

or

"FAILED"

or other things described below.

Table VII

Some Interactive Commands

Name of command	User types	The machine's response
PRETTY-PRINT	TP	It prints the theorem on the scope in an easily readable form (see example below).
	TP F	If PUT F = ( ) has been used earlier, it prints the theorem on the scope with each occurrence of ( ) replaced by the symbol F.
	TP F G ...	Similarly for F, G, etc.
	TPC F	Similarly for conclusion only.
	TPH F	Similarly for hypothesis only.
	TL	It prints the theorem label.
	TY	It pretty-prints TYPELIST.
	TPR	It pretty-prints the REDUCE table.
ADD-DEFN	ADD-DEFN A ( )	( ) is added to the definition table as the definition of the expression A.
ADD-REDUCE	ADD-REDUCE ( )	( ) is (permanently) added to the REDUCE table.
ADD-PAIRS	ADD-PAIRS ( )	( ) is (permanently) added to the PAIRS table.
DEFN	D A	It replaces all occurrences of A by its (stored) definition.
	DC	It defines the conclusion of the the current goal.
USE	USE N	It fetches theorem number N from memory and adds it to the hypothesis of the current theorem.
	USE ( )	It adds ( ) to the hypothesis.

Name of command	User types	The machine's response
LEMMA	LEMMA ( )	It first proves ( ) and then calls USE ( ).
SUBGOAL	SUBGOAL A	It calls (Lemma (H⟶A)) where H is the current hypothesis.
PROCEED	CONTINUE	It proceeds with the proof with no changes by the user.
	GO	Exit current subgoal with "PROVED" or "FAILED" as was determined by the program.
TIMELIMIT	CNT N	It increases the timelimit on the current subgoal by a factor N.
ASSUME	A	It assumes the current subgoal to be proved and proceeds.
FAIL	F	It fails the current subgoal (i.e., returns NIL).
BACKUP	BACK	It backs up to the previous preset back-up point.
	B	Create a backup point.
REORDER	(N ⟶ M)	It reorders the goal, placing hypothesis number N first and conclusion number M first.
	(N1 N2 ... ⟶ C)	It reorders the goal placing hypothesis number N1 N2 ... first in that order.
	(H⟶M1 M2 ...)	Similarly for conclusions M1 M2 ...
	(N1 N2 ... ⟶ M1 M2 ...)	Similarly for both.

Name of command	User types	The machine's response
DELETE	DELETE N M ...	It deletes hypotheses number N, M, ...
PUT	PUT X ( )	The machine replaces each occurrence of x in theorem being proved, by ( ).

Time Limit.

In all cases the program worker under a time limit determined by the user. If it does not prove the current subgoal within that time limit it will halt and report

"FAILED TIMELIMIT"

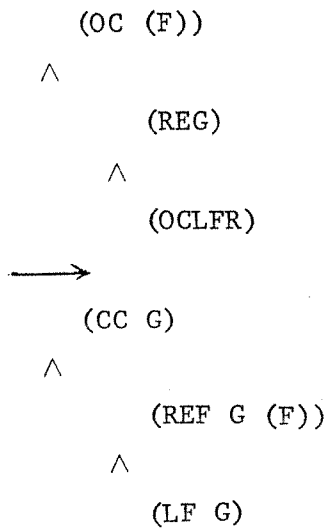
The time limit can be increased (or decreased) by use of the command (CNT N) (See Table VII).

Pretty-Print.

The command TP causes the machine to print the current theorem (subgoal) in a parsed, easy to read form. For example, if the theorem is

(⟶ (∧ (OC (FSDI))(∧ (REG)(OCLFR)))(∧ (CC G)(∧ (REF G(FSDI))(LF G))))

the command TP will cause to be printed on the scope:



Note that the skolem constant (FSDI) has been printed as (F), though its complete form is retained by the program.

Now if the command

PUT G {C: Closed C}

is used, the conclusion is altered accordingly. The command TPC if issued now will cause

$$\begin{array}{l} (CC\{C\text{: Closed } C\}) \\ \wedge \\ \quad (REF\{C\text{: Closed } C\}(F)) \\ \wedge \\ \quad (LF\{C\text{: Closed } C\}) \end{array}$$

to be printed, whereas TPC G causes

$$\begin{array}{l} (CC\ G) \\ \wedge \\ \quad (REF\ G\ (F)) \\ \wedge \\ \quad (LF\ G) \end{array}$$

to be printed.

ADD-DEFN, ADD-REDUCE, and ADD-PAIRS allows the user to easily add entries in Tables III, IV and VI.

The command "D A" causes the program to expand the definition of A through the theorem. For example, if the current subgoal is

$$(Oc\ F \longrightarrow Cover\ F)$$

and the command "(D Oc)" is issued by the user, then the subgoal is changed to

$$(Open\ F \wedge Cover\ F \longrightarrow Cover\ F)$$

"(DH A)" and "(DC A)" would cause such changes only in the hypothesis or the conclusion respectively.

(USE A) simply allows the user to add the additional hypothesis A, whereas (LEMMA A) requires the prover to prove A first and then add it as a hypothesis, whereas (SUBGOAL A) calls (LEMMA (H$\rightarrow$A)) where H is the current hypothesis.

The commands A (for "ASSUME") and F (for "FAIL") are useful for terminating a long proof or for maneuvering the proof to parts of the theorem that the user is interested in.

The command (n m $\longrightarrow$ i j) causes the hypotheses and conclusions to be reordered with hypothesis number n first, and number m second, and with conclusion number i first and number j second, etc. The command (DELETE n m...) causes hypotheses numbers n, m,... to be deleted. For example if the current goal is

$$
\begin{array}{ll}
 & (x_o \in A) \\
\wedge & (x_o \in B) \\
\wedge & (t \in A \longrightarrow t \in C) \\
\wedge & \text{Open } A \\
\longrightarrow & \\
 & (x_o \in C) \\
\wedge & \text{Open } B
\end{array}
$$

and the command (4 1 3 $\longrightarrow$ 2) is issued the goal is changed to

$$\begin{array}{ll} & \text{Open } A \\ \wedge & (x_o \in A) \\ \wedge & (t \in A \longrightarrow t \in C) \\ \wedge & (x_o \in B) \\ \longrightarrow & \\ & \text{Open } B \\ \wedge & (x_o \in C) \ , \end{array}$$

and if the command (DELETE 2 4 1) is now issued the goal is changed to

$$\begin{array}{ll} & (t \in A \longrightarrow t \in C) \\ \longrightarrow & \\ & \text{Open } B \\ \wedge & (x_o \in C) \ . \end{array}$$

PUT is one of the most important commands. It allows us to instantiate a skolem variable with a desired formula. For example, if the prover is trying to prove the theorem

$$\forall\, x(x \in A \longrightarrow \exists B(\text{Open } B \wedge B \subseteq A \wedge x \in B)) \longrightarrow \exists F(F \subseteq \text{OPEN} \wedge A = \cup F)$$

it will obtain the goal

(1) $$\overbrace{((x \in A_o \longrightarrow \text{Open } B_o \wedge B_o \subseteq A_o \wedge x \in B)}^{\alpha} \longrightarrow F \subseteq \text{OPEN} \wedge A_o = \cup F)$$

At this point the machine may be unable to determine the required family F of open sets whose union is A_o. If the user decides to help, he can easily do so by using the command "PUT" to give a value to F. For example the command (PUT F (OPEN $\cap$ Subsets A_o)) will cause (1) to be changed to

(1) $(\alpha \Rightarrow (\text{OPEN} \cap \text{Subsets } A_o) \subseteq \text{OPEN} \wedge \cup (\text{OPEN} \cap \text{Subsets } A_o) = A_o)$,

which is easily proved, as follows.

(1 1) $(\alpha \Rightarrow (\text{OPEN} \cap \text{Subsets } A_o) \subseteq \text{OPEN})$ I 4

$(\alpha \Rightarrow (B_o \in (\text{OPEN} \cap \text{Subsets } A_o) \longrightarrow B_o \in \text{OPEN}))$ I 13

$(\alpha \wedge \text{Open } B_o \wedge B_o \subseteq A_o \Longrightarrow \text{Open } B_o)$, TRUE I 7, 5

(1 2) $(\alpha \Rightarrow \cup (\text{OPEN} \cap \text{Subsets } A_o) = A_o)$ I 4.2

(1 2 1) $(\alpha \Rightarrow \cup (\text{OPEN} \cap \text{Subsets } A_o) \subseteq A_o)$ I 13, I 4

$(\alpha \Rightarrow x_o \in \cup (\quad) \longrightarrow x_o \in A_o)$ I 13

$(\alpha \Rightarrow (B_o \in (\text{OPEN} \cap \text{Subsets } A_o) \wedge x_o \in B_o \longrightarrow x_o \in A_o))$ I 5

$(\alpha \wedge \text{Open } B_o \wedge B_o \subseteq A_o \wedge x_o \in B_o \wedge x_o \in A_o \Longrightarrow x_o \in A_o)$
TRUE

Forward Chaining was used in the previous step.

(1 2 2) $(\alpha \Rightarrow A_o \subseteq \cup (\text{OPEN} \cap \text{Subset } A_o)$ I 4.2

$(\alpha \wedge x_o \in A_o \Rightarrow \text{Open } B \wedge B \subseteq A_o \wedge x_o \in B)$ I 13, 7, 5

$(x_o \in A_o)$ is forward chained into α to get

$(\text{Open } B_o \wedge B_o \subseteq A_o \wedge x_o \in B_o)$. Thus (1 2 2) becomes

(1 2 2) $(\alpha \wedge x_o \in A_o \wedge \text{Open } B_o \wedge B_o \subseteq A_o \wedge x_o \in B_o \Rightarrow \text{Open } B \wedge B \subseteq A_o \wedge x_o \in B)$,

which holds for B_o/B. Thus B_o/B is returned for (1 2 2), (1 2), and (1).

Other examples using PUT and the other interactive commands are given in [1].

The DETAIL command is fully explained in Section 2.1 of [1] and is used in several examples there. It is used to let the machine tell the user which part of the proof it is having trouble with. If the prover fails on a goal of the type

(1 2) $(H \Rightarrow A \ \wedge \ B)$

the command "DETAIL" will (in essence) ask for information on the proof of each of the subgoals $(H \longrightarrow A)$ and $(H \longrightarrow B)$. Thus the machine might respond

(1 2 1) $(H \longrightarrow A)$

PROVED..

Then after a user commands "PROCEED", it might respond

(1 2 2) $(H \longrightarrow B)$

FAILED..

In this way the user is told which of the two subgoals the prover is having trouble with, and can direct his help accordingly. A further command "DETAIL" would act similarly on subgoals of B (if there are any).

INDUCTION K commands the program to try to prove the current subgoal using mathematical (finite) induction on K. For example if the current goal is

$P(K)$

it will rewrite it as

$$P(0) \wedge (P(K) \longrightarrow P(K+1)) ,$$

also universally quantifying any free variables in P(K). For example,

$$\sum_{i=0}^{N} i = N(N+1)/2$$

is converted by the command INDUCTION N, to

$$\left(\sum_{i=0}^{0} i = 0(0+1)/2\right)$$

$$\wedge$$

$$\left(\sum_{i=0}^{N} i = N(N+1)/2 \longrightarrow \sum_{i=0}^{N+1} i = (N+1)((N+1)+1)/2\right)$$

which can now be proved automatically by the use of a simplification routine and REDUCE rewrite rules which convert

$$\sum_{i=0}^{0} f(i)$$

to 0, and convert

$$\sum_{i=0}^{K+1} f(i)$$

to

$$\sum_{i=0}^{K} f(i) + f(K+1) .$$

In many examples the subgoal itself is not a sufficient induction hypothesis. Thus it is necessary to "prove more" in order to get the desired result. To facilitate this, the command (INDUCTION K Q) can be used whereby the user supplies the induction hypothesis Q.

Many other researchers have used induction in their automatic theorem proving programs [40,2,43,44,29]. Boyer and Moore [29] have employed an interesting concept called "generalization" which converts the current subgoal into a more general theorem, but one which then can be proved by induction.

Optional REDUCE.

For some large theorems, for example like those encountered in program verification (see Part I), it is not desirable to call REDUCE each time IMPLY is executed, because this can be very time consuming. Accordingly the program can be operated in a mode that causes it to stop before executing Rule 7 of IMPLY, and print "DO YOU WANT TO REDUCE? TYPE: Y or N". A "N" will cause it to proceed without reducing.

User Equals Substitution.

In Rule 9 of HOA the program applys a "substitution of equals". Given a hypothesis (a = b) it selects either a or b and replaces it by the other throughout H and C. An optional mode of operation is provided that allows the human user to override this process. In this mode the program chooses either a or b and proposes that to the user for him to accept as proposed, reverse, or reject altogether. The program stops and prints

"a Replaced by b?"

The user then says one of:

"Yes"	(means do the substitution)
"R"	(means replace b by a)
"No"	(means do not do any substitution. Proceed)
"Next"	If $b \equiv c+d$ (or $a = e+f$) the program will find the next possible substitution and print "c Replaced by d-a?"

and the whole process repeats.

Most of these interactive commands are retractable. If a command has changed the theorem in any way, the machine displays the changed version and then asks "OK???". The program will then make the change permanent only if the user types "OK".

All the user commands such as PUT, USE, etc., may be called initially without arguments. When this happens the program asks the user for the required arguments. For example the following is a sample dialogue where "h" stands for user input and "c" for machine quiries.

```
c       IMPLY STOP
h       ADD-REDUCE
c       PATTERN:
h       a + 0
c       REWRITE AS:
h       a
c       CONDITIONAL (YES OR NO)?
h       NO
```

Some More Details.

In Section 1 we said that IMPLY was called with five arguments, (TYPELIST, H, C, TL, LT), but only the principal ones, H, C, and the theorem label TL, were discussed in Sections 1-6. TYPELIST is discussed in [27] and LT is a "light" which helps control the man-machine interaction. This is discussed below.

The routine IMPLY has three major sections -- CNTRL (control), OPTIONS, and the features described in Sections 2-6. CNTRL is executed at the entry to IMPLY and is the only section that uses the light (LT). The purpose of LT is to differentiate between calls to DETAIL (LT = 3), CNT (LT = 5), which are described above, and (LT = B). A DETAIL call stops at OPTIONS before returning from the top level sub-calls to IMPLY. A CNT call (which gives a larger time-limit) does not stop at OPTIONS on any sub-call. A "B" call stops at OPTIONS before any proof is attempted. If the theorem is $(H \longrightarrow C_1 \wedge C_2)$, there are top-level sub-calls to IMPLY for $(H \longrightarrow C_1)$ and $(H \longrightarrow C_2)$. DETAIL stops at OPTIONS after $(H \longrightarrow C_1)$ is attempted and then after $(H \longrightarrow C_2)$ is attempted. CNT does not stop until after the attempt to prove $(H \longrightarrow C_1 \wedge C_2)$ is completed. A "B" call stops before $(H \longrightarrow C_1)$ is attempted. CNTRL does various things according to the value of LT. If LT = 3, CNTRL resets LT to 3 and goes directly down to OPTIONS for human intervention. If LT = 1 or LT > 2, CNTRL recalls IMPLY with LT one less than its present value. The result of this call ("PROVED", "FAILED", or "PROVED CONDITIONALLY" -- see Section 3 of Part I) is printed and execution continues at OPTIONS. In most cases control passes down to OPTIONS directly.

At OPTIONS human intervention is bypassed if LT < 1 or LT = 2 and control is passed on to the IMPLY Rules (See Table I). Otherwise "IMPLY-STOP" is printed on the screen and the program waits for the user to enter commands. At present there are about 35 different commands available, including those listed in Table VII. As mentioned earlier, some commands cause information to be printed, some cause special proving methods to be tried (perhaps with a larger "time" limit or with more human intervention). Some allow the user to give prover more information, or to arrange the theorem.

8. Some Applications

This prover has been used to prove theorems in the following areas:

(1) Set Theory [2]

(2) Limit Theorems of Calculus [3]

(3) Topology [1, 38, 39]

(4) Limit Theorem of Analysis [45]

(5) Program Verification [6, 27].

In [45] the methods of non-standard analysis are used whereby the theorem in question is converted automatically to a theorem in non-standard analysis, and then proved in the new setting which seems to be more conductive to automatic proofs. The typing concepts (see Section 1 of [27]) and Reductions (see Section 3) play a major role in handling infinitesimals, and other typed quantities.

Appendix 1

Skolemization -- Elimination of Quantifiers

First we give examples. The formula

$$\exists x\ P(x)$$

is skolemized as

$$P(x)$$

where x is a "skolem variable" which can be replaced by any term during the proof. Similar,

$$Q \longrightarrow \exists x\ P(x)\ ,$$

and

$$\exists x (Q \longrightarrow P(x))\ ,$$

are skolemized as

$$Q \longrightarrow P(x)\ ,$$

and

$$(\forall x\ P(x) \longrightarrow C)$$

is skolemized as

$$(P(x) \longrightarrow C)\ ,$$

where x is a skolem variable.

On the other hand, the formulas

$$\forall x\, P(x)\ ,$$

$$Q \longrightarrow \forall x\, P(x)\ ,$$

$$\forall x (Q \longrightarrow P(x))\ ,$$

$$(\exists x\, P(x) \longrightarrow C)\ ,$$

are skolemized as

$$P(x_o)\ ,$$

$$(Q \longrightarrow P(x_o))$$

and

$$(P(x_o) \longrightarrow C)\ ,$$

where x_o is a skolem constant (cannot be replaced).

Finally

$$(\forall x \exists y\, P(x,y) \longrightarrow \exists u \forall v\, Q(u,v))$$

is skolemized as

$$(P(x, g(x)) \longrightarrow Q(u, h(u)))$$

where g and h are skolem functions.

Notice that we do <u>not</u> place the formula being skolemized in prenex form, but skolemize it in place, leave each logical symbol except $\forall$ and $\exists$, in its original position.

We now give the general rules.

Given a formula E, we recursively define[2] as "positive" or "negative" the subformulas of E, as follows:

1. E is positive
2. If $(A \wedge B)$ is positive (negative) then so are A and B
3. If $(A \vee B)$ " " " " " " " " "
4. If $\sim A$ is positive (negative) then A is negative (positive)
5. If $(A \longrightarrow B)$ is positive (negative) then
   A is negative (positive), and
   B is a positive (negative)
6. If $(\forall x\ A)$ is positive (negative) then
   A is positive (negative), and
   $\forall$ is a positive (negative) quantifier
7. If $(\exists x\ A)$ is positive (negative) then
   A is positive (negative), and
   $\exists$ is a negative (positive) quantifier.

For example if E is the formula

$$([H \longrightarrow (C \longrightarrow \sim D)] \longrightarrow [\sim A \vee (B \longrightarrow F)])$$

then E, $[\sim A \vee (B \longrightarrow F)]$, $\sim A$, $(B \longrightarrow F)$, F, H, C and D are positive, while $[H \longrightarrow (C \longrightarrow \sim D)]$, $(C \longrightarrow \sim D)$, $\sim D$, A, and B are negative.

Given a formula E with no free variables, we eliminate the quantifiers of E by deleting each quantifier and each variable immediately after it, and replacing each variable v bound by a positive quantifier with the skolem

---

[2] See Wang [23].

expression $g(x_1,\ldots,x_n)$ where g is a new function symbol (a "skolem function" symbol) and $x_1,\ldots,x_n$, consists of those variables of E which are bound by negative quantifiers whose scope includes v. The result is called the "skolem form of E".

For example if E is the formula

$$\exists x \forall y \; P(x,y)$$

then $\exists$ is a negative quantifier, $\forall$ is a positive quantifier, and

$$P(x,g(x))$$

is the skolem form of E, whereas the formulas

$$\forall x(P(x) \rightarrow \exists y \; Q(x,y)) ,$$

and

$$\exists x(\forall y \exists z \; P(x,y,z) \rightarrow \forall w \; Q(x,w))$$

have the skolem forms

$$(P(x_o) \rightarrow Q(x_o,y)) ,$$

and

$$(P(x,y,g(x,y)) \rightarrow Q(x,h(x)))$$

respectively, and the formula

$$\forall x(\forall y[\exists z \; H(x,y,z) \rightarrow \exists u(C(x,u) \rightarrow \sim \forall v \; D(u,v))]$$
$$\rightarrow \forall w[\sim A(x,w) \vee \exists s(\exists t \; B(s,t) \rightarrow \forall r \; F(x,r,s,t))])$$

has skolem form

$$([H(x_o, yz) \longrightarrow (C(x_o, g(y)) \longrightarrow \sim D(g(y), h(y)))]$$
$$\longrightarrow [\sim A(x_o, w_o) \vee (B(s, j(s)) \longrightarrow F(x_o, k(s), s, j(s))]) \ .$$

It should be noted (by those familiar with Resolution proofs) that the formula E is <u>not</u> first negated before the skolem form is derived. This difference reverses the roles of $\forall$ and $\exists$ in the skolemization process.

Appendix 2

Incompleteness of the Prover

As mentioned earlier the prover is incomplete, in that there are theorems that it cannot prove. Of course, it has the usual incompleteness, that humans and other systems possess, of not being able to prove really hard theorems, like Fermat's last theorem (if it is a theorem). But our system is incomplete in three other ways which we refer to under the headings of "using ANDS in backchaining", "trapping", and "multiple copies". We will discuss these and changes we could make to eliminate this incompleteness, and why it is not desirable to do so. See also [46].

Using ANDS in Backchaining.

In Rule 7 of HOA if the hypothesis B has the form $(A \longrightarrow D)$ then we put

(i) $$\Theta := \text{ANDS}(D, C)$$

and then try to prove $(H \longrightarrow A\Theta)$. A more complete procedure would use

(ii) $$\Theta := \text{IMPLY}(D \wedge H,\ C)$$

bringing to bear the whole strength of IMPLY instead of the weaker routine ANDS. The following example, illustrates this inadequacy

Ex. A1. $$(A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \longrightarrow C).$$

Of course forward chaining would quickly prove this, but let us assume,

for the sake of the point we are making, that forward chaining is inoperative. Then we obtain:

(1)	$(A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \Rightarrow C)$		I 7
	$(A \Rightarrow C)$	NIL	H 6
	$(B \Rightarrow C)$	NIL	H 6
	$((A \longrightarrow (B \longrightarrow C)) \Rightarrow C)$		H 6
	ANDS $(B \longrightarrow C, C)$	Returns NIL	H 7

So NIL is returned for (1).

If in Rule 7 of HOA, we had used (ii) instead of (i), (Calling it Rule 7'), then the proof would proceed to a successful conclusion as follows:

(1)	$(A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \Rightarrow C)$		I 7
	$(A \Rightarrow C)$	NIL	H 6
	$(B \Rightarrow C)$	NIL	H 6
	$(A \longrightarrow (B \longrightarrow C)) \Rightarrow C$		H 6
(1 C)	$((B \longrightarrow C) \wedge A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \Rightarrow C)$		H 7'
(1 C C)	$(C \wedge (B \longrightarrow C) \wedge A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \Rightarrow C)$		H 7'
		"T"	H 6, 112
(1 C H)	$((B \longrightarrow C) \wedge A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \Rightarrow B)$		H 7.2
		"T"	H 6, 2
(1 H)	$(A \wedge B \wedge (A \longrightarrow (B \longrightarrow C)) \longrightarrow A)$		H 7.2
		"T"	H 6, 2

However we do not use Rule 7' because it causes the procedure to spend too much time trying to prove the subgoal IMPLY(D $\wedge$ H $\longrightarrow$ C) in cases where that is impossible, instead of proceeding with other lines of attack. Using ANDS instead of IMPLY prevents this futile attempt at backchaining, allowing it to happen only in cases when D essentially matches C.

Trapping.

In Rule 4 Table I when a conclusion of the form

$$A \wedge B$$

is being proved, we first prove A, getting a substitution θ, and then prove B θ. Sometimes, as in the following example, the value of θ returned by IMPLY for the proof of A, will not work for B. We call this "trapping".

Ex. A2. $(P(a) \wedge P(b) \wedge Q(b) \longrightarrow \exists x(P(x) \wedge Q(x)))$

(1)	$(P(a) \wedge P(b) \wedge Q(b) \Rightarrow P(x) \wedge Q(x))$	I 7
(1 1)	$(P(a) \wedge P(b) \wedge Q(b) \Rightarrow P(x))$	I 4
	Returns a|x for (1 1)	H 6, H 2
(1 2)	$(P(a) \wedge P(b) \wedge Q(b) \Rightarrow Q(a))$	I 4.2
	Returns NIL for (1 2)	
	Returns NIL for (1)	I 4.3

We could have prevented trapping in this example by trying Q(x) first and then P(x). So in general, when proving (P(x) $\wedge$ Q(x)) we might

want to first try $(P(x) \wedge Q(x))$ and if that fails then try $(Q(x) \wedge P(x))$. This could be effected by changing Rules 4.3 and 4.4 of Table I to read

4.3'	$\lambda \equiv$ NIL	Put θ': = IMPLY(H,B)	
4.3.1	$\theta' \equiv$ NIL		NIL
4.3.2	$\theta' \equiv$ NIL	Put λ': = IMPLY(H,Aθ')	
4.3.2.1	$\lambda' \equiv$ NIL		NIL
4.3.2.2	$\lambda' \not\equiv$ NIL		$\theta' \circ \lambda'$
4.4'	$\lambda \not\equiv$ NIL		$\theta \circ \lambda$

A similar permutation would have to been provided for in the hypothesis in order to handle an example like

Ex. A3. $(P(a) \wedge Q(b) \wedge P(c) \wedge Q(c) \rightarrow \exists x(P(x) \wedge Q(x)))$

because there either $P(x)$ or $Q(x)$ first would produce trapping. So it would be necessary to permute the hypothesis to $(Q(b) \wedge P(c) \wedge P(a) \wedge Q(c))$ or one of the other successful configurations. This could be effected by further changing 4.3.2.1 to

4.3.2.1'	$\lambda' \equiv$ NIL	
4.3.2.1.1	H $\not\equiv$ (D $\wedge$ E)	NIL
4.3.2.1.2	H $\equiv$ (D $\wedge$ E)	IMPLY(E $\wedge$ D, A $\wedge$ B).

(In this case we would need to set and test a light to prevent Step 4.3.2.1.2 from being repeated indefinitely).

Unfortunately such changes as these greatly increase the running time (sometimes by orders of magnitude) for all theorems including those that need no such permutations. So our present version has neither change. Most of the examples encountered so far don't require it. Also in our interactive system (see Section 7) such a failure is shown to the human operator who can permute the conclusions and/or the hypotheses with one simple command and thereby achieve the desired result.

Nevins [17] has prevented trapping on an AND-SPLIT

$$(A(x) \wedge B(x))$$

by obtaining the set S_A of all values of x satisfying $A(x)$, and the set S_B of all values of x satisfying $B(x)$, and then intersecting S_A and S_B for the solution of $(A(x) \wedge B(x))$. See also [47]. In using this method one must be careful to provide a special mechanism for cases where S_A or S_B is infinite.

Multiple Copies.

In the following example the hypothesis $\forall x\, P(x)$ is used twice in proving the theorem.

Ex. A4. $(\forall x\ P(x) \longrightarrow P(a) \wedge P(b))$

(1)	$(P(x) \Rightarrow P(a) \wedge P(b))$		I 7
(1 1)	$(P(x) \Rightarrow P(a))$	a/x	I 4, H 2
(1 2)	$(P(x) \Rightarrow P(b))$	b/x	I 4.2, H 2

Returns (b/x, a/x) for (1).

There is no problem here, and also we have no difficulty with the following equivalent version of this example.

Ex. A5. $\exists x(P(x) \longrightarrow P(a) \wedge P(b))$

(1)	$(P(x) \longrightarrow P(a) \wedge P(b))$		I 7

etc. as in the previous example.

However, in the following equivalent version we reach an impass.

Ex. A6. $\exists x[(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b))]$

(1)	$(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b))$		
(1 1)	$(P(x) \Rightarrow P(a))$	a/x	I 4, I 7, H 2
(1 2)	$(P(a) \Rightarrow P(b))$	NIL	I 4.4, I 7

NIL is returned for (1).

If the program was able to convert this example to its equivalent form

$$\exists x(P(x) \longrightarrow P(a) \wedge P(b))$$

then there would be no difficulty. But then a similar theorem such as

Ex. A7. $\forall z(Q(z) \longrightarrow P(z)) \longrightarrow \exists x[(P(x) \longrightarrow P(a)) \wedge (Q(x) \longrightarrow P(b))]$

would be very difficult to convert without eliminating the implication symbol "$\longrightarrow$".

There are two possible ways to cope with this type of difficulty. First we could eliminate all of the implication symbols (using $(\sim A \vee B)$ for $(A \longrightarrow B)$) from the given theorem, and work from there. But this would change the basic nature of our system, and we do not wish to do it for reasons which we give later.

Secondly, we could require that the program make "multiple copies" of existentially quantified disjuncts in the conclusion. For example

$$(H \longrightarrow \exists x \ P(x))$$

would be copied as

$$(H \longrightarrow P(x) \vee P(x')) \ .$$

(Similarly, universally quantified conjunctions in the hypothesis

$$(\forall x \ P(x) \longrightarrow C)$$

should be copied as

$$(P(x) \wedge P(x') \longrightarrow C) \ ,$$

but this is already done by Rule I 4.2). (More generally, we would copy

existentially quantified expressions in any "positive" position (see Appendix 1) of the theorem, and any universally quantified expression in a "negative" position.

Thus in Ex. A6, after copying, we get

(1) $[(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b))] \vee [(P(x') \longrightarrow P(a)) \wedge (P(x') \quad P(b))]$

$\sim [(P(x') \longrightarrow P(a)) \wedge (P(x') \longrightarrow P(b))] \Rightarrow [(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b))]$

H 4.2

$(P(x') \wedge \sim P(a)) \vee (P(x') \wedge \sim P(b)) \Rightarrow [(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b))]$

(1 1) $(P(x') \wedge \sim P(a) \Rightarrow [(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b))]$ I 3

(1 1 1) $(P(x') \wedge \sim P(a) \Rightarrow (P(x) \longrightarrow P(a)))$ I 4

$(P(x) \wedge P(a') \wedge \sim P(a) \Rightarrow P(a))$ a/x I 7, H 2

(1 1 2) $(P(x') \wedge \sim P(a) \Rightarrow (P(a) \longrightarrow P(b)))$ b/x' I 7, H 2

(1 2) $(P(b) \wedge \sim P(b) \Rightarrow [(P(a) \longrightarrow P(a)) \wedge (P(a) \longrightarrow P(b))]$

(1 2 1) $(P(b) \wedge \sim P(b) \Rightarrow (P(a) \longrightarrow P(a)))$ "T" I 4, 7, H 2

(1 2 2) $(P(b) \wedge \sim P(b) \Rightarrow (P(a) \longrightarrow P(b)))$ "T" I 7, H 2

So (a/x, b/x') is returned for (1).

A similar attack will succeed for Ex. A7. Of course, we may sometime need more copies than two, and in fact, a procedure would need to be established whereby copies are continually generated, at intervals, until the theorem is proved. Unfortunately this leads to an infinite regression on most non-theorems.

To show what copying does to a non-theorem we try

Ex. A8. $Q(a) \longrightarrow \exists x[(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b)) \wedge Q(x)]$

NOT A THEOREM.

This example also needs copying

(1) $(Q(a) \longrightarrow [(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b)) \wedge Q(x)]$

$\vee [(P(x') \longrightarrow P(a)) \wedge (P(x') \longrightarrow P(b)) \wedge Q(x')])$

$Q(a) \wedge \sim [(P(x') \longrightarrow \cdots] \Rightarrow [(P(x) \longrightarrow P(a)) \cdots]$

$[(Q(a) \wedge P(x') \wedge \sim P(a)) \vee (Q(a) \wedge P(x') \wedge \sim P(b)) \vee (Q(a) \wedge \sim Q(x'))$

$\Rightarrow \overbrace{[(P(x) \longrightarrow P(a)) \wedge (P(x) \longrightarrow P(b)) \wedge Q(x)]}^{C}$

(1 1) $(Q(a) \wedge P(x') \wedge \sim P(a)) \Rightarrow$ C

(1 1 1) $(Q(a) \wedge P(x') \wedge \sim P(a) \Rightarrow (P(x) \longrightarrow P(a))$ a/x

(1 1 2) $(Q(a) \wedge P(x') \wedge \sim P(a) \Rightarrow (P(a) \longrightarrow P(b)) \wedge Q(a))$

(1 1 2 1) $(Q(a) \wedge P(x') \wedge \sim P(a) \Rightarrow (P(a) \longrightarrow P(b)))$ b/x'

(1 1 2 2) $(Q(a) \wedge P(b) \wedge \sim P(a) \Rightarrow Q(a))$ "T"

Returns a/x, b/x' for (1 1)

(1 2) $[(Q(a) \wedge P(x') \wedge \sim P(b)) \wedge (Q(a) \wedge \sim Q(x')](b/x') \Rightarrow C(a/x)$

(1 2 1) $(Q(a) \wedge P(b) \wedge \sim P(b) \Rightarrow C(a/x))$

(1 2 1 1) $(Q(a) \wedge P(b) \wedge \sim P(b) \Rightarrow (P(a) \longrightarrow P(a)))$ "T"

(1 2 1 2) $(Q(a) \wedge P(b) \wedge \sim P(b) \Rightarrow (P(a) \longrightarrow P(b)) \wedge Q(a))$

(1 2 1 2 1) $(Q(a) \wedge P(b) \wedge \sim P(b) \Rightarrow (P(a) \longrightarrow P(b)))$ "T"

(1 2 1 2 2) $(Q(a) \wedge P(b) \wedge \sim P(b) \Rightarrow Q(a))$ "T"

(1 2 2) $(Q(a) \wedge \sim Q(b) \Rightarrow C(a/x))$

(1 2 2 1) $(Q(a) \wedge \sim Q(b) \Rightarrow (P(a) \longrightarrow P(a))$ "T"

(1 2 2 2) $(Q(a) \wedge \sim Q(b) \Rightarrow (P(a) \longrightarrow P(b)) \wedge Q(a))$

(1 2 2 2 1) $(Q(a) \wedge \sim Q(b) \Rightarrow (P(a) \longrightarrow P(b)))$

$(Q(a) \wedge P(a) \Rightarrow P(b) \vee Q(b))$ NIL

So NIL is returned for (1 2). It should also return NIL for (1), since it is false, but it would copy again instead and never return.

Clearly an additional hypothesis Q(b) would make (1) valid in which case "T" would have been returned for (1 2 2 2 1), and (a/x, b/x') returned for (1).

Comment.

We object to the inclusion of "copying" rules and rules to permute hypotheses and conclusions to prevent "trapping" for several reasons.

These rules are not needed on a very large percentage of theorems, and yet they greatly increase the computing time in nearly all cases, sometimes by a large order of magnitude. If we ever expect to prove really difficult theorems in mathematics we must not strangle the mechanism that does it by making sure that it handles every case. Rather we believe (in the spirit of information theory) that it should be allowed to fail on a few cases so that it can succeed on a number of others, especially the hard ones.

The difficulties we point out here are faced by all other proving systems. Resolution systems pay the price by continuing all proofs of the theorem (allowed by the particular restriction on resolution). Gentzen type systems pay it through copying and search. They both remove the implication symbol and work with the result which we feel is very unnatural.

The implication symbol "$\longrightarrow$" or its equivalent plays a crucial role in mathematics. Much of Mathematics consists of stating and proving theorems of the form

$$(H_1 \wedge H_2 \wedge \ldots \wedge H_n \longrightarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m) \quad .$$

Human proof technique, developed over a period of a few thousand years, center around using one or more of the H's to imply one of the C's. We have wanted to keep this same spirit in our prover so that we can easily use some of the powerful heuristics developed by mathematicians, so we can best interact with the prover on a man-machine basis.

Appendix 3

Some Proofs of Soundness

In Section 2 we stated that a call to IMPLY(H,C) would return NIL or a substitution θ such that

(1) $$(H\theta \rightarrow C\theta)$$

is valid. Indeed this is the case if θ has no conflicting entries (entries of the form a/x and b/x, $a \neq b$, or g(y)/x and f(x)/y). We wish now to describe a slight change that could be made to IMPLY which would allow it to handle such conflicting entries, and which would permit us to prove a soundness result of the form (1). In doing so we will generalize the notion of a substitution, θ, allowing symbolic disjunctions

$$\theta = (\theta_1 \vee \theta_2)$$

to be returned from IMPLY. In this new setting the formula

(2) $$(H \rightarrow C)\theta$$

will be shown to be valid.

Before considering these generalized substitutions let us firm up our position for ordinary substitutions. Conflicts (of the form mentioned above) are <u>not</u> introduced when two formulas are unified. Indeed the classical unification algorithm prevents that. But conflicts may be introduced at AND-SPLIT's like Rule I4, where a goal of the form

(3) $$(H \Rightarrow A \wedge B)$$

is being proved. If θ is returned for $(H \Rightarrow A)$, and λ is returned for

$(H \Rightarrow B\theta)$, then Rule I4.4 returns $\sigma = \theta \circ \lambda$ for (3). (It would be easy to prevent any such conflict by requiring the subgoal $(H\theta \rightarrow B\theta)$ instead of $(H \rightarrow B\theta)$, but this would greatly weaken the prover, preventing the proof of such simple theorems as Example A4, Appendix 2.) If a conflict is introduced into σ it in no way implies the invalidity of formula (3). (We confirm this in Theorem 1 below.) It is just that such a σ with a conflict cannot be substituted (in the usual way) into other formulas. We overcome this difficulty by generalized substitutions.

Theorem 1. If θ and λ are ordinary substitutions, each without conflict, then

$$(4) \qquad (H \rightarrow A)\theta \wedge (H \rightarrow B\theta)\lambda \rightarrow (H \rightarrow A \wedge B) .$$

Proof. What are we trying to prove here? We must show that there are ordinary substitutions $\sigma_1, \sigma_2, \ldots, \sigma_n$ for which

$$(5) \qquad (H \rightarrow A \wedge B)\sigma_1 \vee \ldots \vee (H \rightarrow A \wedge B)\sigma_n$$

is a valid consequence of the hypothesis

$$(6) \qquad (H\theta \rightarrow A\theta) \wedge (H\lambda \rightarrow B\theta\lambda).$$

We will show (5) for $\sigma_1 = \theta$, $\sigma_2 = \lambda$.

The proof is by contradiction. Suppose that

$$(7) \qquad (H \rightarrow A \wedge B)\theta \vee (H \rightarrow A \wedge B)\lambda$$

is false. Then we have

$$(8) \qquad H\theta \wedge \sim(A\theta \wedge B\theta) \wedge H\lambda \wedge \sim(A\lambda \wedge B\theta) ,$$

from which we infer by (6), that

$$(A\theta \wedge B\theta\lambda) \wedge \sim(A\theta \wedge B\theta)$$
$$= \sim[A\theta \wedge B\theta \longrightarrow A\theta \wedge B\theta\lambda]$$
$$= \sim(\text{TRUE}),$$

a contradiction. Thus our assumption (8) is false and (7) is True. Q.E.D.

It should be noted that in our new notation below, the substitution $(\theta \vee \lambda)$ is returned for (3).

When we do have a conflict, according to Footnote 4, Section 2, we must consider two cases separately and combine the results from these into a resultant substitution.

We now propose instead of this another way of handling conflicting entries. The soundness of this new method is easier to prove, and its soundness implies the soundness of our present system.

Generalized Substitutions.

Definition. θ is a generalized substitution if

(i) θ is an ordinary substitution, or

(ii) θ has the form

$$(\theta_1 \vee \theta_2) \quad \text{or} \quad (\theta_1 \wedge \theta_2)$$

where θ_1 and θ_2 are generalized substitutions.

Some examples are,

$$\theta_1, \quad \theta_1 \vee \theta_2, \quad ((\theta_1 \vee \theta_2) \wedge \theta_3),$$

where the θ_i are ordinary substitutions.

Definition. If θ is a generalized substitution, then we define θ' by

(i) $\theta' = \theta$ if θ is an ordinary substitution

(ii) $(\theta_1 \vee \theta_2)' = (\theta_1' \wedge \theta_2')$,

(iii) $(\theta_1 \wedge \theta_2)' = (\theta_1' \vee \theta_2')$.

(This definition is for this Appendix only).

Definition. A generalized substitution is said to be a pure disjunction (conjunction) if it contains no $\wedge$ symbols ($\vee$ symbols).

Notice that this definition allows ordinary substitutions to be called pure disjunctions (and pure conjunctions).

Definition. If A is a formula and θ is a generalized substitution, then

$$A\theta$$

is the formula gotten by applying θ from left to right, i.e.,

(i) $A\theta$ is the usual result if θ is an ordinary substitution,

(ii) $A(\theta_1 \vee \theta_2) = A\theta_1 \vee A\theta_2$

(iii) $A(\theta_1 \wedge \theta_2) = A\theta_1 \wedge A\theta_2$.

Definition. For ordinary substitutions θ and λ, the iteration $\theta\lambda$ is defined to be the symbolic disjunction

$$\theta \vee \lambda$$

if θ and λ have conflicting entries (i.e., there are entries a/x in θ and b/x in λ, with $a \neq b$, or $g(y)/x$ and $f(x)/y$), and $\theta \circ \lambda$ otherwise.

Whenever we have an iterated generalized substitution such as

$\theta\lambda$

we can convert it into a generalized substitution by applying λ to θ. For example, if θ_i and λ_i are ordinary substitutions then

$$\begin{aligned}&(\theta_1 \wedge \theta_2)(\lambda_1 \vee \lambda_2)\\ &= (\theta_1\lambda_1 \wedge \theta_2\lambda_1) \vee (\theta_1\lambda_2 \wedge \theta_2\lambda_2)\\ &= (\theta_1\lambda_1 \vee \theta_1\lambda_2) \wedge (\theta_1\lambda_1 \vee \theta_2\lambda_2)\\ &\quad (\theta_2\lambda_1 \vee \theta_1\lambda_2) \wedge (\theta_2\lambda_1 \vee \theta_2\lambda_2)\ ,\end{aligned}$$

where each of the terms $\theta_i\lambda_j$ is converted into $\theta_i \circ \lambda_j$ or $\theta_i \vee \lambda_j$.

Properties of generalized substitutions.

Lemma 1. If Φ and λ are generalized substitutions, λ is a pure disjunction, and A and B are formulas then λ' is a pure conjunction and

.1 $(\Phi')' = \Phi$

.2 $\sim(A\Phi) = \sim B\Phi'$

.3 $(A \vee B)\lambda = A\lambda \vee B\lambda$

.4 $(A \wedge B)\lambda' = A\lambda' \wedge B\lambda'$

.5 $(A \rightarrow B)\lambda = (A\lambda' \rightarrow B\lambda)$

Proof. .1 and .2 follow directly from the definition of Φ' and the properties of $\sim$. .3 and .4 follow from the associativity of $\vee$ and of $\wedge$. Then .5 follows from .3, .2, and .1, as follows

$$\begin{aligned}(A \rightarrow B)\lambda &= (\sim A \vee B)\lambda\\ &= (\sim A\lambda \vee B\lambda)\\ &= (\sim(A\lambda') \vee B\lambda)\\ &= (A\lambda' \rightarrow B\lambda)\ .\end{aligned}$$

Generalized substitutions in IMPLY and HOA.

The change we propose in the program applies only at AND-SPLITS (Rules 3 and 4 of IMPLY, Rules 7 and 7E of HOA). Two changes are required in proving an AND-SPLIT of the form

$$(H \Rightarrow A \wedge B) .$$

(1) We must state how the substitution θ which is returned for the first subgoal

$$(H \Rightarrow A)$$

is applied to B, before calling IMPLY again on the second subgoal.

(2) And we must state how we combine θ with the substitution λ returned from the second subgoal.

Table A-I gives these changes for IMPLY Rule 4. A similar change is needed for IMPLY Rule 3, and HOA Rules 7 and 7E.

Table A-I

	IF	ACTION	RETURN
4.	$C \equiv A \wedge B$	Put θ: = IMPLY(H,A)	
4.1	$\theta \equiv$ NIL		NIL
4.2	$\theta \not\equiv$ NIL	Put λ: = IMPLY(H,Bθ') where $\theta' = \theta$, if θ is an ordinary substitution, $(\theta_1 \vee \theta_2)' = \theta_1' \wedge \theta_2'$, and $(\theta_1 \wedge \theta_2)' = (\theta_1' \vee \theta_2')$.	
4.3	$\lambda \equiv$ NIL		NIL
4.4	$\lambda \not\equiv$ NIL		COMBINE(θ,λ)

COMBINE(θ,λ)

IF	ACTION	RETURN
$\lambda \equiv (\lambda_1 \vee \lambda_2)$		(COMBINE(θ,λ_1), COMBINE(θ,λ_2))
$\theta \equiv (\theta_1 \vee \theta_2)$		(COMBINE(θ_1,λ), COMBINE(θ_2,λ))
θ and λ have a conflict		$(\theta \vee \lambda)$
ELSE		$\theta \circ \lambda$

Notice that if θ and λ are pure disjunctions (which may be ordinary substitutions) then so also is COMBINE(θ,λ).

It should be noted that if θ and λ are ordinary substitutions which do not have conflicting entries then $\theta \circ \lambda$ is returned as before. If they do have conflicting entries, then $(\theta \vee \lambda)$, the symbolic disjunction of the two, is returned. If θ (or λ) is already such a disjunction

$$\theta = (\theta_1 \vee \theta_2) ,$$

then λ is combined to θ_1 and θ_2 separately. If λ has no conflict with θ_1 and θ_2 then the result is

$$(\theta_1 \circ \lambda \vee \theta_2 \circ \lambda) .$$

If λ has a conflict with θ_1 the result could be

$$((\theta_1 \vee \lambda) \vee \theta_2 \circ \lambda) ,$$

etc. In this way a pure disjunction is always returned.

The reader should note that by Rule A-I 4.2, when proving

$$(H \Rightarrow A \wedge B)$$

if $(\theta_1 \vee \theta_2)$ is returned for

$$(H \Rightarrow A)$$

then the second subgoal is

$$(H \Rightarrow B(\theta_1 \wedge \theta_2)) .$$

This requires <u>both</u> $B\theta_1$ and $B\theta_2$ to be proved, and is consistent with our Footnote 4, in Section 2, which states that if two conflicting values are returned from the first subgoal, then two cases must be handled.

We could have omitted $\theta \circ \lambda$ altogether even when θ and λ have no conflicts, and always use $\theta \vee \lambda$, but this would be less efficient for most proofs.

Soundness.

We are now in a position to prove the soundness of our extended system. We will do so only for Rule I4. The proof for the other rules is similar.

If we are proving

$$(H \Rightarrow A \wedge B) ,$$

and θ is returned for

$$(H \Rightarrow A)$$

and λ is returned for

$$(H \Rightarrow B\theta')$$

then it will return COMBINE(θ,λ) for

$$(H \Rightarrow A \wedge B) .$$

We show that this is a valid way of proceeding by proving Theorem 4 below.

Lemma 2. If λ is a pure disjunction then

$$(D \nrightarrow D\lambda) .$$

Proof. The proof is by induction on the structure of λ.

Case 1. If λ is a ordinary substitution then $D\lambda$ is an instance of D and the result is immediate.

Case 2. (Induction step). If $\lambda = \lambda_1 \vee \lambda_2$, and we assume the induction hypothesis

$$(D \rightarrow D\lambda_1)$$

and

$$(D \rightarrow D\lambda_2)$$

then we have

$$D \rightarrow D\lambda_1 \wedge D\lambda_2 = D(\lambda_1 \vee \lambda_2) = D\lambda \ ,$$

as required.

Lemma 3. If θ is a pure disjunction, then

$$(D\theta \wedge (D \rightarrow E)\theta' \rightarrow E\theta) \ .$$

Proof. The proof is by induction on the structure of θ.

Case 1. If θ is an ordinary substitution then $\theta' = \theta$, and the result follows by modus ponens.

Case 2. (Induction step). If $\theta = \theta_1 \vee \theta_2$, and we assume the induction hypothesis,

$$D\theta_1 \wedge (D \rightarrow E)\theta_1' \rightarrow E\theta_1 \ ,$$

and

$$D\theta_2 \wedge (D \rightarrow E)\theta_2' \rightarrow E\theta_2 \ ,$$

then we have

$$\begin{aligned} & D\theta \wedge (D \rightarrow E)\theta' \\ = \ & D(\theta_1 \vee \theta_2) \wedge (D \rightarrow E)(\theta_1' \wedge \theta_2') \\ = \ & (D\theta_1 \vee D\theta_2) \wedge (D \rightarrow E)\theta_1' \wedge (D \rightarrow E)\theta_2' \\ \longrightarrow \ & E\theta_1 \vee E\theta_2 \\ = \ & E\theta \end{aligned}$$

as required. Q.E.D.

Comment. Lemma 2 is not surprising, but Lemma 3 is a curious form of modus ponens, which seems false at first glance. One should not make the mistake of putting $(D \to E)\theta'$ equal to $(D\theta \to E\theta')$. But even if he did the result would still not follow by modus ponens. Lemma 3 is crucial in the proof of Theorem 2 below.

Theorem 2. If θ and λ are pure disjunctions, then

(1) $(H \to A)\theta$

(2) $\wedge\ (H \to B\theta')\lambda$

(3) $\longrightarrow (H \to A \wedge B)(\theta \wedge \lambda)$.

Proof. The proof is by contradiction. Suppose that the hypotheses (1) and (2) hold and the conclusion (3) is false.

Thus using Lemma 1.5 we have

(4) $(H\theta' \to A\theta)$

and

(5) $(H\lambda' \to B\theta'\lambda)$,

and

(6) $\sim[(H \to A \wedge B)(\theta \wedge \lambda)]$

which, using Lemmas 1.5 and 1.2, is equivalent to

$\sim[H(\theta \vee \lambda)' \longrightarrow (A \wedge B)(\theta \vee \lambda)]$

$= \sim[H(\theta' \wedge \lambda') \longrightarrow (A \wedge B)(\theta \vee \lambda)]$

$= H\theta' \wedge H\lambda' \wedge \sim(A \wedge B)(\theta \vee \lambda)'$

$= H\theta' \wedge H\lambda' \wedge (\sim A \vee \sim B)\theta' \wedge (\sim A \vee \sim B)\lambda'$

(7)$= H\theta' \wedge H\lambda' \wedge (A \to \sim B)\theta' \wedge (A \to \sim B)\lambda'$.

It follows from (7) and (4) and (5) (using Modus ponens) that

(8) $A\theta$,

(9) $B\theta'\lambda$,

and from (8), (7) and Lemma 3 that

(10) $\sim B\theta = \sim[B\theta']$

But (10) is in contradiction of (9) by Lemma 2, and hence our assumption (6) is false and the theorem is true. Q.E.D.

Theorem 2 shows that we can dispense altogether wish $\theta \circ \lambda$ using always instead $\theta \vee \lambda$. However, as mentioned earlier, this would be much less efficient when θ and λ have no conflict.

A weaker version of Theorem 2 would have sufficed in the proof of Theorem 4.

Theorem 3. If θ and λ are ordinary substitutions with no mutual conflicts and H is a formula, then

$$(H\lambda \rightarrow H\theta\lambda) .$$

Proof. We will sketch the proof only for the case where H has only two variables x and y. We write $H = G(x,y)$.

Since θ and λ have no conflict within themselves then they can take only the following possible forms:

θ: a/x. b/y, $g(y)/x$, $f(x)/y$,

$a/x\ b/y$, $a/x\ f(x)/y$, $g(y)/x\ b/y$,

λ: c/x, d/y, $j(y)/x$, $h(x)/y$,

$c/x\ d/y$, $c/x\ h(x)/y$, $j(y)/x\ d/y$,

where a,b,c,d are constants.

Since θ and λ have no mutual conflicts, many combinations such as:

$$\theta = a/x \qquad \lambda = c/x \quad ,$$

$$\theta = g(y)/x \qquad \lambda = f(x)/y \ ,$$

are conflicting and need not be considered. Others such as

$$\theta = a/x \qquad \lambda = h(x)/y \ ,$$

$$\theta = g(y)/x \qquad \lambda = d/y \quad ,$$

are not conflicting, and for these cases the desired conclusion holds. For example if $\theta = a/x$, $\lambda = h(x)/y$, then

$$H\theta\lambda = G(a,h(x)), \quad H\lambda = G(x,h(x)) \ ,$$

and

$$(H\lambda \rightarrow H\theta\lambda)$$

becomes

$$G(x,h(x)) \rightarrow G(a,h(x))$$

which is valid (put a for x).

Theorem 4. If θ and λ are pure disjunctions, then

$$(H \longrightarrow A)\theta$$
$$\wedge\ (H \longrightarrow B\theta')\lambda$$
$$\longrightarrow (H \longrightarrow A \wedge B)\ \mathrm{COMBINE}(\theta,\lambda)$$

Proof. We will abbreviate $\mathrm{COMBINE}(\theta,\lambda)$ as $C(\theta,\lambda)$ in this proof. The proof is by induction of the structures of λ and θ.

Case 1. λ is an ordinary substitution.

Case 1.1. θ is an ordinary substitution.

Case 1.1.1. λ and θ have no mutual conflict.

In this case $C(\theta,\lambda) = \theta \circ \lambda$, and the desired result follows from Theorem 3.

Case 1.1.2. λ and θ have a conflict.

In this case $C(\theta,\lambda) = \theta \vee \lambda$, and the desired conclusion follows as a special case of Theorem 2.

Case 1.2. $\theta = \theta_1 \vee \theta_2$.

In this case $C(\theta,\lambda) = C(\theta_1,\lambda) \vee C(\theta_2,\lambda)$, and the induction hypotheses are

(1) $$(H \rightarrow A)\theta_1 \wedge (H \rightarrow B\theta_1')\lambda \rightarrow (H \rightarrow A \wedge B)C(\theta_1,\lambda) ,$$

(2) $$(H \rightarrow A)\theta_2 \wedge (H \rightarrow B\theta_2')\lambda \rightarrow (H \rightarrow A \wedge B)C(\theta_2,\lambda) .$$

Thus

$$\begin{aligned} & (H \rightarrow A)\theta \wedge (H \rightarrow B\theta')\lambda \\ &= [(H \rightarrow A)\theta_1 \vee (H \rightarrow A)\theta_2] \wedge (H \rightarrow B\theta_1' \wedge B\theta_2')\lambda \\ &= [(H \rightarrow A)\theta_1 \vee (H \rightarrow A)\theta_2] \wedge (H \rightarrow B\theta_1')\lambda \wedge (H \rightarrow B\theta_2')\lambda \end{aligned}$$

since λ is an ordinary substitution

$$\longrightarrow (H \rightarrow A \wedge B)C(\theta_1,\lambda) \vee (H \rightarrow A \wedge B)C(\theta_2,\lambda)$$

by (1) and (2)

$$\begin{aligned} &= (H \rightarrow A \wedge B)[C(\theta_1,\lambda) \vee C(\theta_2,\lambda)] \\ &= (H \rightarrow A \wedge B)C(\theta,\lambda) . \end{aligned}$$

Case 2. $\lambda = \lambda_1 \vee \lambda_2$.

In this case, $C(\theta,\lambda) = C(\theta,\lambda_1) \vee C(\theta,\lambda_2)$ and the induction hypotheses are

(3) $$(H \rightarrow A)\theta \wedge (H \rightarrow B\theta')\lambda_1 \longrightarrow (H \rightarrow A \wedge B)C(\theta,\lambda_1) ,$$

(4) $$(H \rightarrow A)\theta \wedge (H \rightarrow B\theta')\lambda_2 \longrightarrow (h \rightarrow A \wedge B)C(\theta,\lambda_2) .$$

Thus

$$(H \rightarrow A)\theta \wedge (H \rightarrow B\theta')\lambda$$
$$= (H \rightarrow A)\theta \wedge [(H \rightarrow B\theta')\lambda_1 \vee (H \rightarrow B\theta')\lambda_2]$$
$$\longrightarrow (H \rightarrow A \wedge B)C(\theta,\lambda_1) \vee (H \rightarrow A \wedge B)C(\theta,\lambda_2)$$

by (3) and (4)

$$= (H \rightarrow A \wedge B)[C(\theta,\lambda_1) \vee C(\theta,\lambda_2)]$$
$$= (H \rightarrow A \wedge B)C(\theta,\lambda) . \quad \text{Q.E.D.}$$

Example.

$$(Q(a) \wedge Q(b) \longrightarrow \exists x[(P(x) \longrightarrow P(a) \wedge P(b)) \wedge Q(x)])$$

(1) $$(Q(a) \wedge Q(b) \Rightarrow (P(x) \longrightarrow P(a) \wedge P(b)) \wedge Q(x))$$

(1 1) $$(Q(a) \wedge Q(b) \Rightarrow (P(x) \longrightarrow P(a) \wedge P(b)))$$

(1 1 1) $$(Q(a) \wedge Q(b) \wedge P(x) \Rightarrow P(a))$$ a/x

(1 1 2) $$(Q(a) \wedge Q(b) \wedge P(x) \Rightarrow P(b))$$ b/x

Returns $(a/x \vee b/x)$ for (1 1).

(1 2) $$(Q(a) \wedge Q(b) \Rightarrow Q(x)(a/x \vee b/x)')$$

$$(Q(a) \wedge Q(b) \Rightarrow Q(a) \wedge Q(b))$$ TRUE

Returns $(a/x \vee b/x)$ for (1).

References

1. W.W. Bledsoe and P. Bruell. A man-machine theorem-proving system. In Adv. Papers 3rd Int. Joint Conf. Artif. Intell., 1973, pp. 55-65; also Artif. Intell., vol. 5, no. 1, pp. 51-72, Spring 1974.

2. W.W. Bledsoe. Splitting and reduction heuristics in automatic theorem proving. Artificial Intelligence 2(1971), 55-77.

3. W.W. Bledsoe, R.S. Boyer, and W.H. Henneman. Computer proofs of limit theorems. Artif. Intell., vol. 3, no. 1, pp. 27-60, Spring 1972.

4. P. Bruell. A description of the functions of the man-machine topology theorem prover. The Univ. of Texas at Austin. Math Dept. Memo ATP8, 1973.

5. W.W. Bledsoe. The sup-inf method in Presburger arithmetic. Dept. Math., Univ. Texas, Austin, Memo ATP 18. Dec. 1974. Essentially the same as: A new method for proving certain Presburger formulas. Fourth IJCAI, Tblisi, USSR, Sept. 3-8, 1975.

6. D.I. Good, R.L. London, and W.W. Bledsoe. An interactive verification system. Proceedings of the 1975 International Conf. on Reliable Software, Los Angeles, April 1975, pp. 482-492, and IEEE Trans. on Software Engineering 1(1975), pp. 59-67.

7. C. Chang and R.C. Lee. Symbolic logic and mechanical theorem proving. Academic Press, 1973.

8. Donald Loveland. Forthcoming book on Mechanical theorem proving in first order logic. (Duke University)

9. J. Allen and D. Luckham. An interactive theorem-proving program. Machine Intelligence 5(1970), 321-336.

10. J.R. Guard, F.C. Oglesby, J.H. Bennett and L.G. Settle. Semi-automated mathematics. J. ACM 16(1969), 49-62.

11. G.P. Huet. Experiments with an interactive prover for logic with equality. Rept. 1106, Jennings Computing Center, Case Western Reserve University.

12. A. Newell, J.C. Shaw and H.A. Simon. Empirical explorations of the logic theory machine: a case study in heuristics. RAND Corp. Memo P-951, Feb. 28, 1957. Proc. Western Joint Computer Conf. 1956, 218-239.

    A. Newell, J.C. Shaw and H.A. Simon. Report on a general problem-solving program. RAND Corp. Memo P-1584, Dec. 30, 1958.

13. H. Gelerntner. Realization of a geometry theorem-proving machine. Proc. Int'1. Conf. Information Processing, 1959, Paris UNESCO House, 273-282.

14. G. Gentzen. Untersuchungen uber das logische Schließen I. Mathemat. Zeitschrift 39, 176-210, (1935).

15. Arthur J. Nevins. A human oriented logic for automatic theorem proving. MIT-AI-Lab Memo 268, Oct. 1972. JACM 21(1974), 606-621.

16. Arthus J. Nevins. A relaxation approach to splitting in an automatic theorem prover. MIT-AI-Lab. Memo 302, Jan. 1974. To appear in the AI Jour.

17. Arthur J. Nevins. Plane geometry theorem proving using forward chaining. MIT-AI-Lab Memo 303, Jan. 1974.

18. Raymond Reiter. A semantically guided deductive system for automatic theorem proving. Proc. Third Int'1. Joint Conf. on Art. Intel., 1973, 41-46.

19. George W. Ernst, The utility of independent subgoals in theorem proving. Information and Control, April 1971. A definition-driven theorem prover. Int'1. Joint Conf. on Artificial Intelligence, Stanford, Calif., August 1973, pp. 51-55.

20. W. Bibel and J. Schreiber. Proof search in a Gentzen-like system of first-order logic. Bericht Nr. 7412, Technische Universitat, 1974.

21. Carl Hewitt. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Ph.D. Thesis (June 1971). AI-TR-258 MIT-AI-Lab. April 1972.

22. D.V. McDermott and Gerald J. Sussman. The CONNIVER reference manual. AI Memo 259. MIT-AI-Lab. (May 1972) (Revised July 1973).

23. Hao Wang. Toward mechanical mathematics, IBM J. Res. Dev. 4, 2-22.

24. J.R. Rulifson, J.A. Derksen and R.J. Waldinger. "QA4: a procedural calculus for intuitive reasoning. Standford Res. Inst. Artif. Intell. Center, Standford, Calif., Tech. Note 13, Nov. 1972.

25. D. Prawitz. An imporved proof procedure. Theoria 25, 102-139 (1960).

26. Nils Nilsson. Artificial Intelligence / Review of automatic theorem proving. IF1P, Stockholm, Sweden, 1974.

27. W.W. Bledsoe and Mabry Tyson. Typing and proof by cases in program verification. Univ. of Texas Math. Dept. Memo ATP 15, May 1975.

29. R.S. Boyer and J.S. Moore. Proving theorems about Lisp functions. J. Ass. Comput. Mach., vol. 22, pp. 129-144, Jan. 1975.

30. Dallas S. Lankford. Complete sets of reductions for computational logic. Univ. of Texas Math. Dept. Memo ATP-21, Jan. 1975.

31. D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. Computational Problems in Abstract Algebra. J. Leech, Ed., Pergamon Press, 1970, 263-297.

32. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. J. ACM, 21(1974), 622-642.

33. N. Suzuki. Verifying programs by algebraic and logical reduction. Proc. Int'l. Conf. on Reliable Software, 1975, 473-481.

34. Mabry Tyson. An algebraic simplifier. Univ. of Texas Math. Dept. Memo ATP-26 (to appear).

35. A.C. Hearn. Reduce 2: A system and language for algebraic manipulation. In Proc. Ass. Comput. Mach., 2nd Symp. Symbolic and Algebraic Manipulation, 1971, pp. 128-133; also Reduce 2 User's Manual, 2nd ed., Univ. Utah, Salt Lake City, UCP-19, 1974.

36. L. Siklossy, A. Rich and V. Marinov. Breadth-first search: some surprising results. A.I. Jour. 4(1973), 1-28.

37. A. Bundy. Doing arithmetic with diagrams. In Adv. Papers 3rd Int. Joint Conf. Artif. Intell., 1973, pp. 130-138.

38. Michael Ballantyne and William Bennett. Graphing methods for topological proofs. Univ. of Texas at Austin, Math. Dept. Memo ATP-7, 1973.

39. Michael Ballantyne, Computer generation of counterexamples in topology. Univ. of Texas at Austin Math. Dept. Memo ATP-24, 1975.

40. J.L. Darlington. Automatic theorem proving with equality substitution and mathematical induction. Machine Intelligence, 3(1968), 113-127.

41. Jack Minker, D.H. Fishman and J.R. McSkimin. The Q^* algorithm -- a search strategy for a deductive question-answering system. A.I. Jour. 4(1973), 225-243.

42. D.H. Fishman. Experiments with a resolution-based deductive question-answering system and a proposed clause representation for parallel search. Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Maryland, (1973).

43. R.J. Waldinger and K.N. Levitt. Reasoning about programs. Artif. Intel. 5(1974). 235-316.

44. J.C. King. A program verifier. Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1969.

45. Michael Ballantyne. Automatic proofs of limit theorems in analysis. Univ. of Texas at Austin Math. Dept. Memo ATP-23, 1975.

46. Donald W. Loveland and M.E. Stickel. A hole in goal trees: some guidance from resolution theory. Proc. third Int'l. Joint Conf. on Art. Intel., Stanford, 1973, 153-161.

47. Raymond Reiter. A paradigm for automated formal inference. To be presented at the IEEE theorem proving workshop, Argonne Nat'l. Lab., Ill., June 3-5, 1975.

48. S. Ju Maslov. Proof-search startegies for methods of the resolution type. Machine Intelligence 6(1971), 77-90.

49. Bernard Luya. Un systeme complet de deduction maturelle. Thesis, University of Paris VII, Jan. 1975.