A Description of the Functions of the Man-Machine Topology Theorem Prover

Peter Bruell

May 1973

University of Texas, Austin, Texas

ATP-8

A Description of the Functions of the Man-Machine Topology Theorem Prover.

Peter Brvell May, 1973

Some Things to Notice

1. The third argument, HL, of HOA was at one time used as an hypothesis label in the same way that the third argument, TL, of IMPLY is used as a theorem label. However, HL now serves no purpose at all.

2. At various places calls to HOA and IMPLY are indicated with fewer arguments than are actually necessary -- only the true ingredients of the calls are listed. See the LISP code for the actual calls made in each case.

3. Many examples are written in ordinary mathematical notation (see for example EL-REDUCE) rather than the painstakingly accurate prefix notation actually required.

4. Free variables are used throughout the program. If the binding of a variable in a function FOO cannot be determined by tracing back the functions which call FOO, look in INITIALIZE -- the variable is there to be bound.

PART I -- The Main Functions

ACT N

X ≡ (CAR B) B λ-variable in HOA

Y ≡ (CAR C) C λ-variable in IMPLY

If N = 2 and Y is on the EXPLODE list of X then HOA is called using the instantiated definition of B to try to prove C. This is the PEEK feature.

Example

B ≡ Oc F_o

C ≡ Cover F

X ≡ Oc

Y ≡ Cover

Cover is among the atoms on the EXPLODE list of Oc so HOA is called using Cover $F_o \wedge F_o \subseteq \mathcal{T}$ to try to prove Cover F. This succeeds returning F_o/F as a substitution.

If N = 4,5 HOA is called using the instantiated definition of B to try to prove C.

ACTION N

B λ-variable in HOA

C λ-variable in IMPLY

If N = 2 and B and C are each "strange" then (ACT N)

If N = 4 and B is "strange" or C is not "strange" then (ACT N)

If N = 5 and B is not "strange" then (ACT N)

Otherwise NIL

Note: ACTION is called from HOA and the λ-variable N is bound to EXPL.

ADD-DEF R

ADD-DEF is called from the IMPLY-STOP to provide the program with the definition of a term for which it has no definition.

Example

Suppose we are trying to prove that every T3 space is also T2. We could add this theorem to the theorem list by calling.

ADDTH ((27 (→ (T3) (T2)))

This would make the theorem available to us as theorem number 27. At the IMPLY-STOP we would face the problem of telling the program what T3 and T2 spaces are, assuming that it does not already know. This problem is solved by using the ADD-DEF feature:

(ADD-DEF (T2) (ALL P (ALL Q (→ (~ (= P Q))

(SOME G (SOME H (^ (OPEN G) (^ (EL P G)

(^ (OPEN H) (^ (EL Q H) (= (INTERSECT G H) (EMPTY))...)

There are two things to notice here. The first is that the definition is entered in prefix notation. The second is that the definition contains quantifiers. The function ADD-DEF will make two definitions out of the one we supply. It will do this by calling REMQ twice -- once with odd parity and once with even parity. REMQ will in each case return an equivalent quantifier free form of the definition. The odd parity definition will be instantiated for the term if it appears in the hypothesis and the even parity definition will be instantiated if it appears in the conclusion. Notice that the odd parity definition of T2 will be stored under the indicator DEF2 on the property list of T2 and the even parity definition will be stored under the indicator DEF1.

ADDR-TH R

ADDR-TH is called from IMPLY-STOP to make a new entry in the REDUCE table.

Example

Suppose we are trying to prove the theorem

$$A_o \subseteq C_o \wedge B_o \subseteq C_o \rightarrow A_o \cup B_o \subseteq C_o$$

This is easily proved by IMPLY once the conclusion is defined. But there

is a way to prove this theorem by simply rewriting the conclusion. If REDUCE had the rewrite rule $P \cup Q \subseteq R \rightarrow P \subseteq R \wedge Q \subseteq R$ the above theorem would be trivially proved. This rewrite rule can be added to the REDUCE table by using the ADD-REDUCE feature at the IMPLY-STOP, viz.

(ADD-REDUCE $P \cup Q \subseteq R$ $P \subseteq R \wedge Q \subseteq R$)*

This instruction will cause the rewrite rule to be CONSed onto the global variable LU-LIST. When REDUCE is now called on the conclusion of the theorem, the function LOOK-UP will be called. LOOK-UP will match the conclusion with $P \cup Q \subseteq R$ and return $A_o \subseteq C_o \wedge B_o \subseteq C_o$. The theorem then becomes

$$A_o \subseteq C_o \wedge B_o \subseteq C_o \rightarrow A_o \subseteq C_o \wedge B_o \subseteq C_o \quad \text{QED}$$

*Note: The actual entry made at the IMPLY-STOP is

```
(ADD-REDUCE (SUBSET (UNION P Q) R)
            (∧ (SUBSET P R) (SUBSET Q R)))
```

ADD-PAIRS R

ADD-PAIRS is called from the IMPLY-STOP to simplify the proof of a theorem by using PAIRS. See PAIRS for an example.

ADD-H S D H

ADD-H stands for "add hypothesis." It is called by DPUT**. ADD-H

calls ADD-H* to make one substitution σ out of the substitutions in S and the manual substitutions on the PUTLIST and returns $D\sigma \wedge H$.

ADD-H*. P S

ADD-H* collects the list of substitutions on the PUTLIST $\equiv$ P and puts them together with the substitutions in S. Its value is not simply (APPEND P S) because the substitutions on the PUTLIST must be turned around first.

Example

P $\equiv$ ((E U ($\wedge$ (OPEN U) (SUBSET U (GS1))))). F))

S $\equiv$ ((H RS1))

Value...

((F E U ($\wedge$ (OPEN U) (SUBSET U (GS1)))) (H RS1))

ADDTH TH

ADDTH adds TH to the THLIST. It also adds the reference number (CAR TH) to the list of reference theorem numbers THN. See example in ADD-DEF.

ANDATOM L

ANDATOM removes all the $\wedge$'s from a formula in prefix form. Note that $\wedge$ is always assumed to be a binary connective in this program.

ANDIFY L

ANDIFY links together the list of formulas in L by ∧'s.

AND-ON A B

AND-ON returns (∧ A B) if neither A nor B is NIL. If one is NIL the value is the other.

ANDS H L

ANDS is called as part of the "back chaining" rule of HOA and also called from FORCH2. ANDS is a miniature HOA and IMPLY combined. H is used to prove L using unification alone. H and L are assumed to have no toplevel connectives except possibly ∧'s. ∧-splits in L are reversed (if necessary) to avoid possible trapping problems.

Examples

```
Arguments of ANDS...
(EL X M); (SUBSET (R) (Q))

Value of ANDS...
NIL

Arguments of ANDS...
(EL X M); (EL (P) (Z))

Value of ANDS...
((X P) (M Z))
```

Arguments of ANDS...

(∧ (EL (Y) (U)) (∧ (EL (P) (V)) (OPEN (V))));

(∧ (EL X A) (OPEN A))

Value of ANDS...

((X P) (A V))

Note that in this last example the substitution ((X Y) (A U)) is a trap since there is no way to prove (OPEN (U)). ANDS reverses the conjuncts in L and then produces the correct substitution.

APPEND-T X Y

The arguments of APPEND-T are substitutions. If either one is *T* the other is returned. Otherwise the composition of the two substitutions is returned.

ASSOC A L

This function is the same as the LISP function SASSOC except that it uses EQUAL where SASSOC uses EQ.

BACKUP

BACKUP asks how many levels the user wants to back up in the proof of a theorem. It expects the user to type in an integer N and initiates a sequence of N errors which cause LISP to fall back through N

errorsets. These errorsets have been entered via the function TRAP which the user has implicitly called by typing B at the IMPLY-STOP. See TRAP.

CRUNCHLIST L

This function eliminates duplications in the list L. It also removes all occurrences of the atom NIL on the toplevel of L.

CYCLE TH

CYCLE calls IMPLY to prove the quantifier free form of TH. If TH is of the form $A \leftrightarrow B$ CYCLE will first call IMPLY to prove $A \rightarrow B$ and then to prove $B \rightarrow A$. Before the actual call to IMPLY is made CYCLE will TREEP the theorem and wait (at a point called CYCLE-STOP elsewhere in this report) for the user's permission to continue. This permission is granted by typing anything except the word RUN.

If the word RUN is typed the program will bring in the functions from the file RUN and call the function RUN. RUN will ask (WHICH HISTORY?) and will expect to read either an atom U assigned to a HISTORY by SAVE-HISTORY or an integer N. If an integer N is read RUN will set the atom HISTORY to the N^{th} history read off of the local file HISTORY. If an atom U is read RUN will set the atom HISTORY to the history which is the value of U. In either case the RUN-LT will be set to *T* and IMPLY will be called.

See REMQ for a discussion of the free variable REMQTH.

DEFINEC A

This function is entered by typing DC at the IMPLY-STOP. Its effect is to define the conclusion, C, of the theorem. Along with several other functions which have A as their only λ-variable, DEFINEC calls OK to make sure the user approves of the action he has just initiated. A will be bound to either NIL or *T* according as the call is made by the user from the IMPLY-STOP or from RUNDOWN while playing back the HISTORY of the proof of a theorem. If A ≡ *T* the program is in RUN mode and OK will not be called.

DEFINITION A

DEFINITION is entered from the IMPLY-STOP by typing D. A read statement inside DEFINITION then expects an atom to be typed in. DEFINITION will instantiate the definition of this atom throughout the theorem and call IMPLY on the result.

DEFN D X P

What this function does is best illustrated by example.

D ≡ (SUBSET A (BS1))

X ≡ SUBSET

P ≡ *T*

Value...

(→ (EL (XS1) A) (EL (XS1) (BS1)))

D ≡ (SUBSET A (BS1))

X ≡ SUBSET

P ≡ NIL

Value...

(→ (EL × A) (EL × (BS1)))

The value in each case is the definition of D. The two values differ only in their skolemization and this difference is regulated by the parity P.

DEFN* D TH P

DEFN* instantiates the definition of the atom D at each occurrence of this atom in TH. The definition is instantiated with respect to parity by calling DEFN.

DIFFER-2 WL TL

DIFFER-2 takes two theorem labels as arguments. It returns *T* if the two labels differ only in their last number and the last number of WL is a 1 while the last number of TL is a 2. DIFFER-2 is called from IMPLY in the event that the free variable WATCH-LABELS contains a list of theorem labels (see DPUT**).

DPUT A

This function is called by typing DPUT at the IMPLY-STOP. BYPASS

will be set to 0 and an automatic BACKUP point will be created by a call to TRAP. TRAP will call TRAP* which will in turn call IMPLY through an ERRORSET. At this entrance to IMPLY the setting of BYPASS will be detected and a call will go out to DPUT. DPUT sets BYPASS to 1, prints a colon and calls DEFINITION. DEFINITION will expect to read an atom to be defined in the theorem. If all goes well, DEFINITION will define this atom and call IMPLY with the instantiated definition. Here again, the setting of BYPASS will be detected and a call will go out to DPUT*.

DPUT* DA

DA stands for defined atom. DPUT* will TREEP the odd parity definition of the atom and ask if a PUT is needed. If a PUT is needed, DPUT* will print a colon and expect a PUT to be typed in by the user. The format of this PUT is as follows:

(PUT var exp)

This will cause each occurrence of the variable var to be replaced by the expression exp in the definition which has been printed out. The variable BYPASS is set to 2 if a PUT is needed. This will be detected at the next entrance to IMPLY which will be made from MAN-SUBST and will cause DPUT** to be called. If no PUT is needed, control will be returned to the IMPLY-STOP.

DPUT** D R

If the atom defined by DPUT contains an arrow in its definition and a PUT has been made in DPUT*, this function will call IMPLY to prove the hypothesis of that arrow. If this succeeds, the message (ESTABLISHED HYPOTHESIS) will be printed. The new hypotheses gained by doing so will be added to the FCBOX and the theorem label will be added to WATCH-LABELS. IMPLY will then be called to prove the same subgoal it was working on when the DPUT was entered at IMPLY-STOP. However, it will now have additional hypotheses to work with as a result of the successful attempt to establish the hypothesis of the arrow. If this hypothesis cannot be established, the message (COULD NOT ESTABLISH HYPOTHESIS) will be printed and control will be returned to IMPLY-STOP.

Example

It should be clear that DPUT, DPUT*, and DPUT** work as a team. They should never be called explicitly. They should only be entered by typing DPUT at the IMPLY-STOP. In the following dialogue an "m" at the margin stands for machine output and an "h" stands for human input.

```
m   IMPLY-STOP
h   TP                          tree print
m     (REG)
      ^
        (OCLFR)
      →
        (CC G)
```

```
m   IMPLY-STOP
h   TL                          theorem label
m   (1 1)

m   IMPLY-STOP
h   DPUT
m   :
h   OCLFR
m     (OC F)
    →
        (COVER (G))
      ^
        (REF (G) F)
      ^
        (LF (G))
    (IS PUT NEEDED?)
h   YES
m   :
h   (PUT F (TTT))
m   (MANUAL SUBSTITUTION)
        (REG)
    ^
      (OC (TTT))
    →
      (COVER (G))
```

```
        ^
          (REF (G) (TTT))
        ^
            (LF (G))
      →
        (CC G)
      OK???
h     OK
m     (TRY PROVING HYPOTHESIS)
        .
        .
        .
      (ESTABLISHED HYPOTHESIS)
      IMPLY-STOP
h     TP
          (REG)
        ^
          (COVER (G))
        ^
          (REF (G) (TTT))
          ^
            (LF (G))
      →
        (CC G)
                          ***
WATCH-LABELS  ≡  ((1 1))
FCBOX  ≡  ((^ (COVER (G)) (^ (REF (G) (TTT)) (LF (G)))))
```

WATCH-LABELS is set so that the program will begin "watching labels." At each subsequent entrance to IMPLY, a check will be made to see if the theorem label is (1 2). This is done by calling DIFFER-2 on (CAR WATCH-LABELS) and TL. Before the proof of (1 2) is begun, the hypotheses in (CAR FCBOX) will be added to H. WATCH-LABELS and FCBOX will each be set to the CDR of what they were. In this case, each will be set to NIL, but in general, this will not be true. The point is, that WATCH-LABELS and FCBOX are stacks and their purpose is to insure that each subgoal is proved with the proper hypotheses.

EQL-EX A B

EQL-EX attempts to unify A and B. If σ is the most general unifier of A and B, then σ is the value of EQL-EX. If A and B cannot be unified, the value of EQL-EX is NIL.

FIXH HIST H

FIXH reinstatiates definitions that were instantiated manually in the hypothesis of a theorem at an earlier point from the IMPLY-STOP.

It also REPUTs for variables that received a value using PUT at the IMPLY-STOP. FIXH knows which definitions to instantiate and what PUTs to make by scanning the HISTORY of the proof. FIXH is called from IMPLY before the second subjoal of an $\wedge$-split is sent as the new subgoal to be proved.

GETDEF A

GETDEF returns the odd parity definition of the atom A. It is called from DPUT**.

HELP

HELP is called from the IMPLY-STOP by typing either E or EV. An EVALQUOTE or EVAL loop is entered depending on whether E or EV is typed.

HISTORY

The value of HISTORY is always NIL. If the program is in run mode, i.e. RUN-LT $\equiv$ *T*, NIL is returned immediately. Otherwise the global variable HISTORY is first set to (CONS A HISTORY) and then NIL is returned. HISTORY is called before almost every entrance to IMPLY to record a reason for this entrance. The atom HISTORY is building a linear history of the proof of a theorem. This history may be played back at a later time to repeat steps in a proof without any human intervention.

HOA B C HL

HOA is the routine IMPLY calls to use the individual hypotheses to prove the current subgoal. When HOA is called by IMPLY the free variable HOAL is set to an integer N. If HOA is called recursively N times without proving the subgoal it returns NIL. HOA always first checks to see if B and C can be unified and returns their most general unifier if they can.

The following table indicates some of HOA's actions on the basis of the form of B

B	Action Taken
$H_1 \wedge H_2$	If (HOA H_1 C) yields σ then σ Otherwise (HOA H2 C)
$D \rightarrow E$	If EXPL $\neq$ 0 and (ANDS E C) yields σ then (IMPLY H $D\sigma$) H is a λ-variable of IMPLY. Otherwise NIL. This is "back chaining."
$D \leftrightarrow E$	(HOA $(D \rightarrow E \wedge E \rightarrow D)$ C)
D = E	If D and E are the same expression then NIL
	If D occurs in C or in H - {D = E} then (IMPLY $[H - \{D = E\}]\sigma$ $C\sigma$) where σ is the substitution D/E. If this fails, the substitution E/D is attempted.
$H_1 \vee H_2$	If (HOA H_1 C) yields σ_1 and (HOA H_2 C) yields σ_2, then $\sigma_1 \circledast \sigma_2$. Otherwise NIL.
$\sim D$	(IMPLY H-{~D} D $\vee$ C)

If EXPL = 1 when HOA is entered and none of the entries in the preeceding table applies then HOA will call PAIRS if the main predicate of B matches that of C and there is an entry in the PAIRS table concerning this predicate. This set of circumstances is not as unlikely as it may seem. In practice it is a useful way of invoking lemmas to aid in the proof of a theorem. See PAIRS for examples. If EXPL = 1 and the remaining conditions cannot be satisfied, HOA will return NIL.

If EXPL = 0 and B and C cannot be matched, then HOA returns NIL. If EXPL = 2,3,4, or 5 and the table does not apply, then HOA calls (ACTION EXPL).

HOLD=

HOLD= is called from HOA when an equality substitution is made in either direction. The global variable HOLD= will have value *T* if the HOLD= option is used at the IMPLY-STOP. The purpose of this option is to allow equality substitution to act as a DETAIL call to IMPLY. Equality substitution inside of HOA always results in a call to IMPLY and the fourth argument LT of IMPLY will receive the value of a call to HOLD= as its new value. This value will be either 3 (a DETAIL call) or the current value of LT, depending on whether the HOLD= option has been used or not.

HYP R

R is a list of numbers $(n_1,...n_k)$ and HYP returns hypotheses number $n_1,...,n_k$ in a list.

HYP* L R

L is the list of hypotheses in H with toplevel ^-signs removed. R is as in HYP. HYP* calls HYP** to select the individual hypotheses in L.

HYP** L N

See HYP*.

IMPLY H C TL LT

A skeleton of IMPLY looks like this:

```
(IMPLY (λ (H C TL LT)
   (PROG (K X Y SAVE SAVE1 SUBST)
(COND
[(EQN LT 6)...]
[RUN-LT...]                                  I
[BYPASS...])
(COND
[(EQ LT "B) (SETQ LT 3) (GO DOWN)]
*  [(OR (ONEP LT) (LESSP 2  LT))...]         II
[T NIL])
```

```
DOWN
EVLOOP
(OPTIONS)
AND-SPLIT
(RETURN (COND
[          ]                                   III
[          ]

[T (GO BELOW)]    ))
BELOW
(SETQ K 0)
ABOVE
(COND
[   ]                                          IV
[   ]
[T (GO ABOVE)])      )))
```

The numbered CONDs serve the following purposes.

I. If IMPLY is entered with LT = 6 or RUN-LT ≠ NIL or BYPASS ≠ NIL this COND controls further execution.

A. LT = 6

If a theorem subgoal is of the form $H \rightarrow (A \rightarrow B)$ IMPLY will first recall itself on $A \rightarrow B$ with LT = 6. The first clause of COND I will then be satisfied. IMPLY will set EXPL = 0 and return the value of (HOA A B).

B. RUN-LT ≠ NIL

This will be the case if the program is in RUN mode. RUN mode is entered by typing RUN at the CYCLE-STOP. This clause has three subclauses:

i) The program keeps track of unproductive calls to HOA by making an entry of the form (S n) if the call to HOA was made with EXPL = n. When playing back the history of a proof this entry will be detected by the first subclause.

ii) RUNDOWN will return *T* when all steps of the HISTORY have been completed. When this happens the HISTORY list, which has been set to the CDR of what it was at each entrance to RUNDOWN, will be reset to what it was before the first entrance to IMPLY from RUN mode, i.e., will be reset to the history of the proof up to the point of the current entrance and control will be returned to the IMPLY-STOP.

iii) RUNDOWN will return NIL if the first item on the HISTORY list is a one (indicating an $\wedge$-split) or one of the following:

DC BC P ES SP $\sim$=H $\sim$=C $\vee$C $\leftrightarrow$C

This subclause will transfer control to the label AND-SPLIT in this case.

C. BYPASS ≠ NIL

The third clause of COND I will be satisfied if the DPUT option is used at the IMPLY-STOP. See DPUT.

II. The fourth argument of IMPLY is a light. The light, hereafter LT, is set to 1 when IMPLY is first called by CYCLE. With this value the clause labeled * in COND II will be satisfied. The act of satisfying this clause will be called CATCHING. CATCHING will cause a recursive call to IMPLY to be made with the same values for H,C, and TL but with (SUB1 LT).

There are three other ways of CATCHING from the IMPLY-STOP. Each of these arises from a recursive call to IMPLY caused by excercising a certain option.

OPTION	LT VALUE
DETAIL	3
(DETAIL n)	5
(CNT n)	4

The above table lists the three relevant options and the associated LT values that the recursive calls from IMPLY-STOP send.

Each time CATCHING occurs the PROG variable SAVE is set to a copy of the HISTORY list. This is because the IMPLY call that CATCHING causes may fail and we want to be able to restore the HISTORY list to what it was before such a misattempt was undertaken. This accounts for the three clauses directly below the comment card ↓ RESTORE HISTORY IF FAILED ↓

The subclause [(ZEROP HOAL)...] of * will be satisfied if HOA cannot prove the current subgoal in n recursive calls, where n is the value of HOAL. HOAL is normally set to 12, (DETAIL n) and (CNT n) will each set HOAL to n * HOAL before HOA is called.

The clause [(EQ LT "B) (SETQ LT 3) (GO DOWN)] will be satisfied if a backup point is established using either the DPUT option or the B option. Both will call TRAP which will call TRAP* where a call to IMPLY is made through an errorset with the LT set to B. The clause will also be satisfied if RUNDOWN returns *T* (see above) for in this case, IMPLY will be called recursively with LT set to B.

III. This COND is the heart of IMPLY. The following table explains some of the actions it performs.

Arguments of IMPLY	Value of IMPLY
1. H (A ∧ B) (1) 0	If (IMPLY H A (1 1) -1) yields σ_1 and (IMPLY H B (1 2) -1) yields σ_2 then $\sigma_1 \circledast \sigma_2$ Otherwise NIL

2. H (UNIV) (1) 0	*T*
3. H C (1) 0	If $H \equiv C$ then *T* If there is a substitution σ that unifies H and C then σ.
4. H NIL (1) 0	(GO BELOW)
5. H (A $\vee$ B) (1) 0	If (IMPLY $\sim A \wedge H$ B (1) 0) yields σ then σ Otherwise (IMPLY $\sim B \wedge H$ A (1) 0)
6. H (A $\rightarrow$ B) (1) 0	If (IMPLY A B (1 $\rightarrow$ C) 6) returns a substitution σ then σ Otherwise A is forward chained into H producing a set (possibly empty) of new hypotheses N and the value is (IMPLY $N \wedge A \wedge H$ B (1) 0)
7. H (A $\leftrightarrow$ B) (1) 0	(IMPLY H (A $\rightarrow$ B $\wedge$ B $\rightarrow$ A) (1) 0)
8. H (A = B) (1) 0	If A and B can be unified by σ then σ. Otherwise (GO BELOW).

The numbers at the side of the entries in the table correspond to the order of the clauses in COND III. Between the entries numbered 1 and 2 there are two additional clauses. The first of these calls REDUCE* on both H and C. The second argument to REDUCE* is a parity indicator. Thus, C is REDUCEd with even parity and H is REDUCEd with odd parity. The second clause checks to see if REDUCEing has introduced an $\wedge$ as the main connective of C. This is possible. For example, if C had been (EL X (SIGMA G)) then (SETQ C (REDUCE* C T)) would have set C to ($\wedge$ (EL B G) (EL X B)). If an $\wedge$ has been introduced control is transferred back to the label AND-SPLIT.

IV. This COND controls the calling of HOA. The PROG variable K is first set to 0. A call is made to HOA with EXPL = K for K = 0,1,2,4, or 5. If K = 3, the definition of C is instantiated and REDUCEd. This REDUCEd definition is TREEPed and becomes the new conclusion in a recursive call to IMPLY.

A call to HOA when EXPL = n will return NIL in case of honest failure or lack of time. In the latter case, HOA will set HOAL to 0. IMPLY immediately returns NIL from COND IV, if this happens. In the former case, a marker (S n) is added to the HISTORY list. This marker will prevent the identical call from being made if the history of the run is played back. This

is what is referred to under COND I as an unproductive call. The PROG variable SKIP will be set inside of the function SKIP (called from COND I) when the program is in RUN mode. It will contain a list of numbers corresponding to calls to HOA that should be skipped.

INITIALIZE

This function performs the initialization necessary to run PROVER. It is called for its effect before the overlay is created. Its last instruction is to self-destruct.

MAN-SUBST A

MAN-SUBST is called from the IMPLY-STOP by doing a PUT*. It prints (MANUAL SUBSTITUTION), TREEPs the revised theorem and calls OK. If the user approves, MAN-SUBST calls IMPLY on the new theorem. If IMPLY succeeds returning a substitution σ then the value of MAN-SUBST is the composition of the PUT and σ. If IMPLY cannot prove the new theorem, the value of MAN-SUBST will be NIL. The λ-variable A is either NIL or *T* depending on whether MAN-SUBST is called from the IMPLY-STOP or called while the program is in RUN mode.

See DPUT for the format of a PUT.

NEWTH A

NEWTH is called from IMPLY-STOP when it is desired to reorder

the hypotheses and/or conslusion of a theorem. If the user types (m → n) at the IMPLY-STOP, the program will recall IMPLY (after receiving an OK from the user) making hypothesis number m the first hypothesis in H and making conclusion number n the first conclusion in C. Typing (n → C) or (H → n) reorders only H or only C respectively. NEWTH calls PICK to perform the reordering of H and C. In practice, this function is called when it is necessary to manually avoid a trapping problem. See the REJECT option.

OCCUR X Y

OCCUR checks for an occurrence of the expression X at any level in the expression Y and returns *T* if one is found. Otherwise it returns NIL.

OK

OK is called every time the theorem is changed in any way by human intervention from the IMPLY-STOP. It prints OK??? and waits for the user to answer.

Possible answers	Action taken
OK	The program continues computing with the change in effect.
OKS	The program continues computing with the change in effect. However,

	HOAL is first set to 0. This will cause immediate failure the first time HOA is subsequently entered and control will return to IMPLY-STOP.
Anything else	The change is rejected and control returns to IMPLY-STOP.

OPTIONS

OPTIONS is called only from IMPLY. If LT =0 or 2 or LT < 0 OPTIONS transfers control to the label AND-SPLIT in IMPLY. Otherwise, options prints IMPLY-STOP and waits for the user to type in a command. Any command that is not recognized will cause IMPLY-STOP to be printed again. The commands that will be recognized are the following:

Command	Result
C	Continue with the proof.
A	Assume that the current subgoal is true, i.e. return *T*.
F	Fail the current subgoal, i.e. return NIL
EDIT	Call in the LISP editor.
B	Create a backup point by calling TRAP.

BACK	Back-up in the proof by calling BACKUP.
FREEZE	DEFSYS onto file FREEZE.
GO	See explanation below.
REJECT	See explanation below.
DETAIL	See [1] for an explanation of this command.
H	Print the HISTORY list.
SH	Save the HISTORY list.
EXPL	Print the value of EXPL.
TL	Print the theorem label, TL.
DPUT	See DPUT.
TP	Tree print (TREEP) the theorem.
TPC	TREEP the conclusion only.
TPH	TREEP the hypothesis only.
EV	Enter an EVAL loop.
E	Enter an EVALQUOTE loop.
MOVE →	Move the hypothesis of an arrow in C over to H. This can be done to avoid automatic forward chaining.
D	See DEFINITION.
DC	See DEFINEC.
R	See TRYREDUCE
HOLD=	See HOLD=

(FC $n_1 n_2 \ldots n_k$)	See FORWARDCH.
(DETAIL n)	DETAIL with n-times the usual timelimit, i.e. n * HOAL.
(CNT n)	Try proving a subgoal with n-times the usual timelimit. Unlike DETAIL and (DETAIL n) this command is not intended to split the subgoal -- failure will return to the point of departure.
(S a)	Print the internal skolen representation of a.
(PUT v e)	See MAN-SUBST.
(USEH $n_1 \ldots n_k$)	See USEH.
(USE ℓ)	See USE.
(H $\rightarrow n_1 \ldots n_k$)	
($n_1 \ldots n_k \rightarrow$ C)	See NEWTH.
($n_1 \ldots n_k \rightarrow m_1 \ldots m_j$)	
(ADD-PAIRS a c n)	See ADD-PAIRS.
(ADD-REDUCE e n)	See ADDR-TH.
(ADD-DEF a d)	See ADD-DEF.
(TP $v_1 \ldots v_n$)	
(TPC $v_1 \ldots v_n$)	See VPRINT.
(TPH $v_1 \ldots v_n$)	
(v)	See VPRINT*.

GO

If PROVER has just proved a theorem subgoal by finding an appropriate substitition σ, it will print PROVED. Typing GO will cause σ to be returned as a value to be used in further subgoals. HOAL will be set to 0 which will cause the next entrance to HOA to fail immediately.

REJECT

The user should use this command if he recognizes that PROVER has trapped itself by making an incorrect substitution. For example, when trying to prove a theorem of the form

$$P(x_o) \wedge P(y_o) \wedge Q(y_o) \rightarrow P(x) \wedge Q(x)$$

PROVER will first prove $P(x)$ by instantiating x_o for x. This is a trap, for there is no way to then prove $Q(x_o)$. Typing REJECT after $P(x)$ has been incorrectly proved, will cause the substitution x_o/x to be rejected by returning NIL. Note that REJECT and F actually do the same thing.

OR-ON A B

OR-ON returns ($\vee$ A B).

PAIRED A TL

PAIRED returns *T* if the predicate A has been "paired" on already in the proof of the current subgoal. TL is the theorem label.

This function is called only from PAIRS.

Examples

PAIRED (COVER (1 2 1 (P COVER) 1 1 2)) returns *T*.

PAIRED (REF (1 2 1 2)) returns NIL.

PAIRS L H C

PAIRS is called from HOA when EXPL = 1, the main predicate of B is the same as the main predicate of C, and there is an entry in the PAIRS table concerning this predicate. Strictly speaking, there is no PAIRS table. We speak of a predicate having an entry in the PAIRS table if the predicate has something attached under the indicator PAIRLIST on its property list. This entry may be a preset one or it may be made manually from the IMPLY-STOP by using the ADD-PAIRS option.

Examples

Suppose IMPLY is trying to prove the following subgoal of a theorem:

$$F_o \subseteq G_o \wedge \text{Cover } F_o \rightarrow \text{Cover } G_o$$

It is clear that the first hypothesis will not imply the conclusion. The second hypothesis is also usable to imply the conclusion as it stands because F_o and G_o are both constants and hence cannot be unified. Nevertheless, the feeling one has is that this second hypothesis

ought to be useful, for it has to do with a COVER. This is where PAIRS comes in. PAIRS attempts to mediate between a hypothesis and a conclusion that have the same predicate and provide sufficient conditions to prove the conclusion using this hypothesis. The preset entry for COVER in the PAIRS table is

(((COVER G) (COVER H) (REF G H)))

The form of this and all other entries is

((H C NC1) (H C NC2)...(H C NCn))

PAIRS is a recursive routine which will make a call to IMPLY for each member of this list. In our example, there is only one member on the list:

((COVER G) (COVER H) (REF G H))

This is shorthand for the lemma

$$\forall G \forall H (\text{Cover } G \wedge \text{Ref } G\ H \rightarrow \text{Cover } H)$$

Notice that we are invoking this lemma by satisfying its conclusion and one of its hypotheses. PAIRS will make the substitution F_o/G, G_o/H and propose that IMPLY try to prove

$$F_o \subseteq G_o \rightarrow \text{Ref } F_o\ G_o$$

which is easily proved once the definition of Ref is instantiated.

As another example suppose IMPLY is trying to prove

$$B_o \subseteq A_o \wedge \text{Cbl } A_o \rightarrow \text{Cbl } B_o \quad *$$

Neither of the hypotheses as they stand can be used to imply the conclusion. But again, one has the feeling that the second hypothesis should be useful. There is nothing in the PAIRS table concerning CBL but the user may create an entry by using the ADD-PAIRS command at the IMPLY-STOP. In this case, a useful entry to have in the PAIRS table would be one to invoke the lemma

$$\forall C \forall D (D \subseteq C \wedge \text{Cbl } C \rightarrow \text{Cbl } D)$$

To this end, the user may type

(ADD-PAIRS (CBL C) (CBL D) (SUBSET D C))

at the IMPLY-STOP. This will cause HOA to call PAIRS when EXPL = 1 and the goal is to show Cbl $A_o \rightarrow$ Cbl B_o. PAIRS will then call IMPLY to try to prove

$$B_o \subseteq A_o \rightarrow B_o \subseteq A_o$$

PAIRS is called only from HOA. When PAIRS calls IMPLY it appends a list of the form (P pred) to the theorem label, where "pred" is the predicate that is being "paired" on. Thus, in the first

*Cbl F means the family F is countable.

example (P COVER) would appear in the theorem label in the call PAIRS made to IMPLY and in the second example (P CBL) would appear. Before making this call to IMPLY, PAIRS calls PAIRED to check if the predicate it is about to use has already been "paired" on in the proof of the current subgoal. If this is the case, PAIRS will not allow double pairing, i.e. PAIRS will not "pair" on the same predicate again (see PAIRED).

PICK R

PICK is called from NEWTH to do the actual reordering. The value of PICK is the theorem reordered on the basis of R.

PICK* SL L

PICK* is called from PICK. It rearranges L on the basis of SL.

PROVER

PROVER is called by START only. Its only job is to ask the user to type in the number of the theorem that he wants to prove. PROVER will fetch this theorem from the theorem list and pass it along to CYCLE.

PRYNT L

This function is not called explicitly anywhere in the program.

It becomes the definition of PRINT* whenever TREEP* is called. PRYNT calls UNSKO which causes a unique (short) print name to be assigned to each skolem constant and skolem function.

PURG G B

PURG is called from HOA to purge the hypothesis B from the list of hypotheses in G.

PUTT A B C

PUTT CONSes A onto the PUTLIST and then substitutes A for B in C.

REMQ A P V U

REMQ removes quantifiers from the expression A and returns its equivalent quantifier free (skolemized) form according to the parity P. When the theorem is first put into skolem form by CYCLE, free variables are interpreted to be universally quantified over the entire theorem and thus always become skolem constants. At this time, the (LISP) free variable REMQTH will be set to *T*. Thereafter it will be set to NIL and "apparently" free variables will not be interpreted as "universally" free. An example will clarify this last remark.

Example

The process of reducing a formula sometimes introduces quantified variables whoose quantifiers must be removed by REMQ before the

reduced expression is returned to IMPLY. For example, EL-REDUCE will rewrite the formula $t \in \sigma G$ as $\exists B(B \in G \wedge t \in B)$ and then call REMQ to remove the existential quantifier.

The "apparently" free variables t and G will not be interpreted as "universally" free variables when REMQTH = NIL. Thus the value of this call to REMQ will be $(B_o \in G \wedge t \in B_o)$ or $(B \in G \wedge t \in G)$ according as the parity P is odd (= NIL) or even (= *T*).

REMQ is a recursive function. In the course of recursion, skolem variables are stacked up in the λ-variable V and skolem constants and functions are staked up in the λ-variable U. A skolem constant (function) is recognized by the convention that its second letter must be an S.

REMQV V S

REMQV is called only from SUBLISS*. V is a quantified variable and S is a substitution. REMQV returns the substitution S with the dotted pair whose first member is V removed. There may not be such a dotted pair and in this case, REMQV simply returns S.

REMOVEH A H

This function is called only from DPUT**. It removes the hypothesis whose predicate is A. Recall that if DPUT** is called (by using DPUT option at the IMPLY-STOP and passing through the functions DPUT and DPUT*) the hypothesis of an arrow hypothesis $A \rightarrow B$ is sent as a new

subgoal to IMPLY. When trying to establish A, the predicate P whose definition is $A \rightarrow B$ is first removed from H.

REMOVEH* A L

REMOVEH* is the henchman of REMOVEH. The λ-variable L is a list of hypotheses with $\wedge$-signs removed.

SAVE-HISTORY

This function is called by typing SH at the IMPLY-STOP. The HISTORY list will be printed on a local file called HISTORY. The HISTORY will also be assigned as the value of a unique atom. If it is designed to play the HISTORY back without leaving LISP, this unique atom can be entered as a reply to the question (WHICH HISTORY?) which will be printed by the program if RUN is typed at the CYCLE-STOP. See CYCLE for more details.

SKO E

SKO checks to see if the atom which is its argument is a skolem constant or skolem function. This is accomplished by the explicit typing convention explained in REMQ.

SKO-IN R

SKO-IN is called from MAN-SUBST. Its argument R will always be

of the form (PUT v exp). The value of SKO-IN is (PUT v (SKO-IN* exp)).

SKO-IN* S

SKO-IN* replaces each occurrence of a constant in S by its internal skolem representation. What this in effect means is that all PUTs (remember SKO-IN is called from MAN-SUBST) can be made in terms of the short print names the program assigns to skolem expressions. A constant in S is recognized as a list of one element whose CAR is an atom which has something on its property list under the indicator REPRESENTS.

START

START performs the initialization necessary before the proof of a theorem is begun. It calls PROVER through an errorset and prints out MADE IT upon an error free return. If an error occurs which is not caught by an errorset at a lower level, START will print BOMBED OUT.

STRANGE X

STRANGE is called only from ACTION. STRANGE returns *T* if the predicate of X has been declared strange. The predicates whose strangeness has already been declared are found in a DEFLIST in INITIALIZE.

SUB2 A Z

SUB2 is called on the atomic level by SUBLISS*. A is a substitution (a list of dotted pairs). If the atom Z is the first member of one of these dotted pairs then the value of SUB2 is the CDR of this pair. See also SUBLISS.

SUBLISS A B

A is a substitution and B is a formula to which this substitution is to be applied. A is of the form

$$((V_1 \cdot S_1) \quad (V_2 \cdot S_2) \ldots (V_n \cdot S_n))$$

where for $i = 1, \ldots, n$ V_i is a variable and S_i is a value to be substituted for V_i. By applying A to B is meant the process of replacing each occurrence of V_i in B by S_i. If the substitution A is empty (= *T*) then the value of SUBLISS is B. Otherwise SUBLISS* is called.

SUBLISS* A B

SUBLISS* applies the nonempty substitution A to B. If B is a quantified expression, a recursive call to SUBLISS* is made in which the quantified variable is removed from the substitution list. This is done to insure that a quantified variable does not accidentally receive a value intended for an actual variable somewhere else in B.

Example

Arguments of SUBLISS*...

A ≡ ((X C) (G U))*

B ≡ (∧ (SUBSET X G)

(EL (Z) (E D (∧ (OPEN D) (SOME X (∧ (EL X D) (~ (EL X G))))))))

Value of SUBLISS*...

(∧ (SUBSET (C) (U))

(EL (Z) (E D (∧ (OPEN D) (SOME X (∧ (EL X D) (~ (EL X (U)))))))))

*Notice that, in accordance with the definition given in SUBLISS, A actually is a list of dotted pairs.

A ≡ ((X · (C)) (G · (U)))

TRAP A

TRAP is called when the user wants to establish a backup point to which he can easily return should the need arise, while proving a theorem subgoal. Typing a B at the IMPLY-STOP will cause TRAP to be called with A = NIL. TRAP will then add the symbol B to the HISTORY list, save the current value of HISTORY in the IMPLY PROG-variable SAVE, and call TRAP*. If TRAP is called while in RUN mode, A will have the value *T* and TRAP will immediately call TRAP*.

TRAP*

TRAP* first prints out (BACKUP POINT) and then sets X to the value of IMPLY called through an ERRORSET. The purpose of this call is to "trap" any LISP error that is made in the scope of this ERRORSET. The utility of this call is that a LISP error can be made intentionally by typing BACK at the IMPLY-STOP. This will call the function BACKUP which will ask (HOW MANY LEVELS???) and expect to read an integer n. BACKUP will set the global variable GLOBALERROR to n and initiate a sequence of n ERRORs each of which will be "trapped", i.e. each of which will return NIL to an ERRORSET that has been entered by typing B at the IMPLY-STOP. When all n ERRORs have been made, TRAP* will set HISTORY to SAVE (which now has the value it received when the first of the n B's was entered at the IMPLY-STOP), print RESTORED, and call itself recursively. If no LISP error is made in the scope of the ERRORSET, TRAP* returns (CAR X) as the value of the call to IMPLY.

TREEP L N

TREEP tree prints the formula L using the integer N as an indentation regulator. See [1] for an example of a formula which has been TREEPed.

TREEP* L N

TREEP* first puts the EXPR for PRYNT under the indicator EXPR

of the atom PRINT* then calls TREEP L N, and finally removes the EXPR property from PRINT*. The purpose of this is to have PRINT* mean PRYNT when TREEP is called from TREEP*. Recall that PRYNT will print formulas with short print names for skolem expressions.

TRYREDUCE A

Typing an R at the IMPLY-STOP will call TRYREDUCE with A = NIL. TRYREDUCE will then expect to read either an H or a C. If a C is read, TRYREDUCE will call REDUCE* on C, TREEP the result and wait for an OK. If the OK is given, IMPLY is called on the REDUCEd version of C. The procedure is entirely analogous if an H is read. A = *T* if TRYREDUCE is called while the program is in RUN mode.

UNPUT TH

UNPUT replaces each occurrence of an expression that is the value a variable has received through a PUT by that variable. The free variable PUTL is set in VPRINT, the only function that calls UNPUT.

UNSKO L

UNSKO is called only from PRYNT. L will always be a formula having no toplevel connectives when this call is made. UNSKO will return L with short print names for skolem expressions. The short print name for a skolem expression is kept under the indicator SKO

of the corresponding skolem atom. The first time a new skolem expression is printed by PRYNT, UNSKO calls UNSKO* to assign this short print name.

UNSKO* A

A is the explosion of a skolem atom, i.e. one whose second letter is S. If (CAR A) has not been USED already in assigning print names for skolem expressions, UNSKO* will do three things

1) Put (LIST (CAR A)) under the indicator SKO of (CAAR L) -- L is the λ-variable of UNSKO

2) Put (CAR L) under the indicator REPRESENTS of (CAR A)

3) CONS ((CAR A) . 0) to the global variable USED

If (CAR A) has been USED, NN will have as its value a digit to be used in creating a new unique print name for a skolem atom whose first letter has previously been used in the assigning of short print names.

Example

TREEP* ((∧ (EL (XS1) G) (EL (XS2) H)) 1)

will cause

(EL (X) G)

∧

(EL (X1) H)

to be printed. Here PRYNT will call UNSKO twice, once on each conjunct. (XS1) will be assigned the

print name (X) and (XS2) will be assigned the print name (X1). The following calls should now have the indicated values

GET(XS1 SKO) ≡ (X)

GET(X REPRESENTS) ≡ (XS1)

GET(XS2 SKO) ≡ (X1)

GET(X1 REPRESENTS) ≡ (XS2)

EVAL(USED) ≡ ((X · 1))

USE A

USE is called from the IMPLY-STOP by typing in (USE f) where f is the number of a reference theorem prestored in the program or a fact that will be useful in the proof of a theorem subgoal. In the latter case, USE assigns a reference number to f and prints it along with a message.

USED A

USED searches the list of dotted pairs bound to the global variable USED for a pair whose first member is the atom A. If no such pair is found the value of USED is NIL. If such a pair P = (A · n) is found, TEMP1 (used freely in UNSKO*) is set to P. n is a LISP number. The LISP digit corresponding to (ADD1 n) will be the value of USED. The free variable NUMBERS is set in INITIALIZE and has as

its value

((1 $$$1$) (2 $$$2$)...(8 $$$8$))

Notice that each element of this list is a sublist whose first member is a LISP number and whose second member is the corresponding LISP digit.

USEH A

USEH is called from the IMPLY-STOP by typing (USEH n_1 n_2...n_k) where (n_1...n_k) is a list of numbers of hypotheses the user wants to retain in the proof of a theorem subgoal. USEH recalls IMPLY on the same subgoal, retaining only hypotheses $n_1, n_2, \ldots n_k$.

VPRINT R

VPRINT is called from the IMPLY-STOP by using any one of the following options:

i) (TP V_1...V_n)

ii) (TPC V_1...V_n)

iii) (TPH V_1...V_n)

In each case, (V_1...V_n) is a list of variables which have received their values through a PUT. The commands cause the display of

i) the whole theorem

ii) the conclusion only

iii) the hypothesis only

to be made in terms of these variables instead of the values they have received. VPRINT calls VPRINT* to find the values the variables have received (stored on the PUTLIST) and then TREEPs the result of UNPUTting these variables.

VPRINT* V L

VPRINT* is called by VPRINT for reasons already explained. VPRINT* is called implicitly from the IMPLY-STOP if the user types in (v) where v is a variable. If v has received a value through a PUT, this value will be printed out. Otherwise, NIL will be printed out.

PART II -- REDUCE

DOT-REDUCE X

This function performs λ-conversion. If X is of the form

(. (λ t (PR D Q(t))) y)

then the value of DOT-REDUCE is Q(y). Otherwise the value of DOT-REDUCE is X.

EL-REDUCE X

EL-REDUCE will be called by REDUCE* whenever its (REDUCE*'s) first argument is of the form (EL α β) where α and β are arbitrary formulas. The second argument of REDUCE*, P, is a parity indicator. It is used as a free variable in EL-REDUCE. The following table lists actions taken by EL-REDUCE.

	Argument of EL-REDUCE	Value of EL-REDUCE
1)	(EL (ZERO) α)	(ZERO)
2)	(EL (UNIV) α)	(ZERO)
3)	(EL α α)	(ZERO)
4)	(EL α (XXX))	(UNIV) if P = *T*
5)	(EL α (CLOSED))	(CLSD α)
6)	(EL α (TTT))	(OPEN α)
7)	(EL α (∧ A B))	(∧ (EL α A) (EL α B))
8)	(EL α (∨ A B))	(∨ (EL α A) (EL α B))

	Argument of EL-REDUCE	Value of EL-REDUCE
9)	(EL α ~A)	(~ (EL α A))
10)	(EL α (SB A))	(SUBSET α A)
11)	(EL α (SNG A))	(= α A)
12)	(EL α (E X P(X)))	P(α)
13)	(EL (α,β) (EE (X,Y) Q(X,Y)))	Q(α,β)
14)	(EL α R(X))	(EL-REDUCE (EL α (E Y S(Y)))) if R(X) $\equiv$ (E Y S(Y))
15	(EL (α,β) P(X,Y))	(EL-REDUCE (EL (α,β) (EE (X,Y) Q(X,Y)))) if P(X,Y) $\equiv$ (EE (X,Y) Q(X,Y))

If none of the entries in the table applies, the value of EL-REDUCE is its argument.

Entries (14) and (15) in the table are extremely useful. They eliminate the need for adding separate entries for terms whose definitions can be phrased in terms of sets.

Examples

Assume P = *T*

$$\sigma G = \{x \mid \exists B(B \in G \wedge x \in B)\}$$

(EL-REDUCE $t \in \sigma G$) generates a recursive call to EL-REDUCE by line (14):

(EL-REDUCE $t \in \{x \mid \exists B(B \in G \wedge x \in B)\}$)

By line (12) this reduces to $(B \in G \wedge t \in B)$. Note that a call has been made to REMQ here to eliminate the existential quantifier with respect to the parity P = *T*.

$$A \times B = \{(x,y) \mid (x \in A \wedge y \in B)\}$$

(EL-REDUCE $(p,q) \in (A \times B)$) generates a recursive call to EL-REDUCE by line (15).

$$(\text{EL-REDUCE } (p,q) \in \{(x,y) \mid (x \in A \wedge y \in B)\})$$

By line (13) this reduces to $(p \in A \wedge q \in B)$.

EQ-REDUCE $\dot{X}$

Argument of EQ-REDUCE	Value of EQ-REDUCE
(= α α)	(UNIV)
(= α (. F x))	(= α (DOT-REDUCE (. F x)))
(= (. F x) α)	(= α (DOT-REDUCE (. F x)))

If none of the entries in the table applies, the value of EQ-REDUCE is its argument.

LOOK-UP $\dot{X}$ $\dot{L}$

LOOK-UP is called by REDUCE* only. L is a list of reduction rules that have been defined by using the ADD-REDUCE option at the IMPLY-STOP. LOOK-UP attempts to match the formula X to a rule on L.

If this can be done, X is reduced according to this rule; see ADDR-TH for an example.

NOT-REDUCE X

Argument of NOT-REDUCE	Value of NOT-REDUCE
$(\sim (\text{ZERO}))$	(UNIV)
$(\sim (\wedge\ \alpha\ \beta))$	$(\vee\ \sim\alpha\ \sim\beta)$
$(\sim (\vee\ \alpha\ \beta))$	$(\wedge\ \sim\alpha\ \sim\beta)$
$(\sim\sim\alpha)$	α
$(\sim (\text{UNIV}))$	(ZERO)
$(\sim (\rightarrow\ \alpha\ \beta))$	$(\wedge\ \alpha\ \sim\beta)$
$(\sim (\longleftrightarrow\ \alpha\ \beta))$	$(\wedge\ (\vee\ \alpha\ \beta)\ (\vee\ \sim\alpha\ \sim\beta))$

If none of the entries in the table applies, the value of NOT-REDUCE is its argument.

REDUCE TH

REDUCE calls REDUCE* with even parity.

REDUCE* TH P

REDUCE* first calls LOOK-UP to check if TH can be reduced according to a rule on the look-up list (LU-LIST). These rules are the ones added by using the ADD-REDUCE option at the IMPLY-STOP. If none of these rules can be used to reduce TH, REDUCE* searches its

own table of rules.

Arguments of REDUCE*		Value of REDUCE*
(CBL α)	P	(CBL (REDUCE* α P))
(DOMAIN (λ x (PR D Q(x))))	P	D
(CLSD (CLSR α))	P	(UNIV)
(OPEN (INTERIOR α))	P	(UNIV)
(REF α α)	P	(UNIV)
(REF α (BAR α))	P	(UNIV)
($\wedge$ α β)	P	($\wedge$ (REDUCE* α P) (REDUCE* β P))
($\rightarrow$ α β)	P	($\rightarrow$ (REDUCE* α (NOT P)) (REDUCE* β P))
(EL α β)	P	Set THR = (EL-REDUCE (EL α β)) If THR = (EL α β) then THR Otherwise (REDUCE* THR P)
(SUBSET α β)	P	Set THR = (SUBSET-REDUCE (SUBSET α β)) If THR = (SUBSET α β) then THR Otherwise (REDUCE* THR P)
($\sim$ α)	P	Set THR = (NOT-REDUCE ($\sim$ α)) If THR = ($\sim$ α) then ($\sim$ (REDUCE* α (NOT P))) Otherwise (REDUCE* THR P)

If none of the entries in the table applies, the value of REDUCE* is its argument TH.

SUBSET-REDUCE X

Argument of SUBSET-REDUCE	Value of SUBSET-REDUCE
(SUBSET (BAR α) (CLOSED))	(UNIV)
(SUBSET α (CLSR α))	(UNIV)
(SUBSET α α)	(UNIV)
(SUBSET (ZERO) α)	(UNIV)
(SUBSET α (UNIV))	(UNIV)
(SUBSET ($\vee$ α β) γ)	($\wedge$ (SUBSET α γ) (SUBSET β γ))
(SUBSET (SNG α) β)	($\vee$ (EL α β) ($\sim$ (EL α (UNIV)))

If none of the entries in the table applies, the value of SUBSET-REDUCE is its argument.

PART III -- Special Functions

GENSYM GENSYMARG

The FEXPR for GENSYM can be found in INITIALIZE. GENSYM clears the LISP character buffer and then calls GENSYMH to create a short GENSYM atom using the characters in GENSYMARG as a prefix.

Example

The following calls to GENSYM return the indicated values. Note that the arguments of GENSYM are never quoted.

1) (GENSYM A) ≡ A1

2) (GENSYM A) ≡ A2

3) (GENSYM) ≡ G3

4) (GENSYM U S) ≡ US4

5) (GENSYM G E M) ≡ GE5

If GENSYM is called with no argument (3), the letter G will be used as a prefix. If GENSYM is called with more than two arguments (5) only the first two will be used in the prefix of the GENSYM atom. The arguments of GENSYM (if any) should never be numeric.

GENSYMH, SHORTEN, SHORTEN*

These functions should never be called explicitly. They should only be called by calling GENSYM.

EXPLODE A

EXPLODE returns a list of the characters in the literal atom which is its argument.

***

The last five functions are not part of the main overlay. They reside on local file RUN at run time.

***

KILLRUN L

RUN calls KILLRUN to get rid of the functions KILLRUN, READN, RUN, and RUNDOWN after the history has been played back.

READN

READN reads the Nth history off of the local file HISTORY. See CYCLE.

RUN

RUN is called by CYCLE by typing the word RUN at the CYCLE-STOP. See CYCLE.

RUNDOWN HIST

RUNDOWN is called by IMPLY when the program is in RUN mode. RUNDOWN plays back the history of a proof by duplicating the calls made to IMPLY in the course of that proof. Each entry to RUNDOWN will

cause the global variable HISTORY to be set to the CDR of what it was. When HISTORY = NIL RUNDOWN prints (WE HAVE ARRIVED) and calls IMPLY with LT = B.

SKIP HIST

SKIP is called by IMPLY when the next item on the HISTORY list is of the form (S n). SKIP will add n to the IMPLY PROG variable SKIP and call itself recursively until the next item on the HISTORY list is no longer of the form (S n). At this point, SKIP will transfer control to the label BELOW inside of IMPLY.

References

1. W.W. Bledsoe and Peter Bruell, A Man-Machine Theorem Proving System, IJCAI - 73 (to appear).