THE CONTROL STRUCTURE OF IMPLY

by

Peter Bruell

August 1978 ATP-45

This paper details some of the mechanics of the theorem prover, IMPLY, being developed as part of our automatic theorem proving project. It is assumed that the reader has a listing of the program available and also a familiarity with LISP programming. The subject of concern is the control structure of IMPLY: the starting routines, the IMPLY-STOP, the answer mechanism, the trapping mechanism, and the protected variable mechanism. Each of these five components of the control structure is treated in a separate section.

SOME PRELIMINARY OBSERVATIONS

There are several preliminary observations which should be made to assist the reader in understanding this document as well as the program to which it refers. First and foremost, the reader should be aware of the syntax of formulas upon which IMPLY operates. In a word, this syntax is prefix.

For example, the formula

```
(1) (-> (& (P A) (& (P B) (Q B))) (SOME X (& (P X) (Q X))))
```

is the prefix formulation acceptable to IMPLY for the theorem

(2) $Pa \ \& \ Pb \ \& \ Qb \ \text{->} \ \exists x(Px \ \& \ Qx)$

presented in ordinary mathematical notation. Notice that the arrow is the concatenation of two symbols, the minus sign (-) and the greater than symbol (>). Also notice that the conjunction is binary (this holds for the internal representation of disjuncts too) and that universal quantification is implicit for A and B (see the description of the function CYCLE).

When (1) has been skolemized, it might look something like this internally:

```
(3) (-> (& (P (AS203)) (& (P (BS204)) (Q (BS204))))
        (& (P X) (Q X))).
```

Here the implicitly universally quantified variables A and B have become the SKOLEM constants (AS203) and (BS204) respectively (the S stands for SKOLEM), while the existentially quantified variable X in the conclusion has been skolemized as the LISP atom X. This makes life easy for the matching routines UNIFY and INSTANCE, for they know that atoms which are not numeric or members of the list ATOMICCONSTANTS (TRUE, FALSE, POSINF, NEGINF, etc) are Skolem variables.

If Skolem functions need to be introduced, as is the case for the formula

(4) $\forall x \ \exists y \ Pxy \ \text{->} \ \exists v \ \forall u \ Puv$,

then an obvious convenience suggested by (3) is to let the internal representation of the skolemized form of (4) be

```
(5) (-> (P X (YS205 X)) (P (US206 V) V)).
```

This is what IMPLY will in fact do.

While looking at the listing of the program, the reader will notice many references to atoms prefixed by a Q. For example, Q&, Q->, QEL. THe Q stands for QUOTE and the atom references the symbol obtained by deleting the Q. Thus, for example, Q-> could be replaced by (QUOTE ->). The convenience and efficiency of the single symbol are both obvious. These variables are all initialized in the function INITIALIZE.

Finally, the reader will have noticed the frequent calls to the LISP functions CADR and CADDR. This is a consequence of the binary nature of many predicates (EL, SUBSET, etc.) and connectives (&, ->, etc.). Since formulas involving these predicates and connectives are represented in prefix, their two arguments will be addressed by CADR and CADDR.

THE STARTING ROUTINES

The starting routines perform the initialization necessary prior to calling the function IMPLY to prove a theorem. This initialization includes setting some global variables to NIL (START), requesting the theorem to be proved (PROVER), and removing the quantifiers from the theorem (call to REMQ in CYCLE).

START

This function of no arguments is called to start the theorem prover.

PROVER

PROVER expects that the name of the theorem it requests is a LISP atom which has been SETQ'd to the prefix formula which represents the theorem.

CYCLE(TH,TL)

Before CYCLE calls REMQ it sets the variable REMQTH to T and immediately after the call it sets it to NIL. The first setting has the effect of forcing a universal interpretation on all free variables occurring within the formula being skolemized (by REMQ). Thus, with REMQTH set to T

```
Px -> Pa
```

and

```
Px -> Va Pa
```

both skolemize as

```
Px -> Pa0
```

where x is now a skolem variable (LISP atom) and a0 is a Skolem constant. With REMQTH set to NIL both skolemize as

```
     Px => Pa
```

where both x and a are Skolem variables. In other words, with REMQTH set to NIL, free variables are assumed to be Skolem variables. The two settings of REMQTH have two useful consequences:

1. With REMQTH set to T, universal quantification need not be specified for variables whose quantification is universal over an entire theorem.

2. With REMQTH set to NIL, universal quantification need not be specified for variables whose quantification is universal over a lemma being added to the hypothesis via the USE option.

The last line of CYCLE is an indirect call to the function IMPLY.

THE IMPLY-STOP

One of the design goals of IMPLY has always been to permit usage both as an interactive theorem prover and as an automatic theorem prover. The interactive mode is especially useful for debugging purposes. The automatic mode is, of course, the mode in which we would like to operate exclusively; however, the development of IMPLY has required extensive use of the interactive mode as well.

We shall continue to refer to the interactive mode as the interactive mode but shall henceforth call the automatic mode the QED-mode. The two modes are often both used in the proof of a theorem. The interactive mode is used to isolate difficult subgoals whose proof may require some human intervention and the QED-mode is used to dispose of routine subgoals whose proofs are easily discoverd automatically.

The IMPLY-STOP is the name we use to refer to the interactive interface of IMPLY. It is physically located in the two lines of code

```
EVLOOP
  (COND ((EQ LT (QUOTE B)) (OPTIONS)))
```

in the body of the function IMPLY. From this it is evident that on any given call to the function IMPLY we will stop at the IMPLY-STOP (i.e. enter the function OPTIONS) only if the fifth argument, LT, of IMPLY has the value (QUOTE B). LT acts as a control parameter of IMPLY. Its possible settings are tested by the COND in CNTRL.

CNTRL

The first function which IMPLY calls each time it is entered is CNTRL. The task of CNTRL is to inspect the value of LT and take the appropriate action. This may mean taking no action at all.

The only value of LT which will cause IMPLY to stop at the IMPLY-STOP is (QUOTE B). Thus, the first clause of CNTRL insures that no stops at IMPLY-STOP will be made when IMPLY is in QED-mode (QED-LT = T).

When the value of LT is a member of the list of trap lights, TRAPLTS (set in INITIALIZE), then CNTRL calls TRAP which will immediately recall IMPLY through an ERRSET with LT = (QUOTE B). This establishes a backup point. A backup point is represented by a period in the theorem label displayed on the screen. It is useful to have backup points when running in interactive mode to enable the user to resume a proof beginning at the appearance of an earlier subgoal. This is called falling back to a backup point and more will be said about this later. Of course, in QED-mode, backup points are not needed, and IMPLY is never called with a LT value which is a member of TRAPLTS, when QED-LT = T.

When the value of LT is (QUOTE CNTRL), CNTRL will immediately recall IMPLY with LT = 0. Note that on this recursive call to IMPLY, CNTRL will take no action at all.

Finally, when LT has the value 3, CNTRL will recall IMPLY with LT = (QUOTE CNTRL). If NIL is returned, failure will be reported and NIL will be returned as the value of IMPLY. If a non-NIL value is returned, this value will be reported, and INTERPRET-A will be called to decide what to do next.

TRAP(LT)

The function TRAP is the heart of the interactive backup system of IMPLY. As mentioned under the function CNTRL, TRAP is called to establish backup points in the proof tree when IMPLY is running in interactive mode. These backup points correspond to calls on the function IMPLY made through ERRSET. By "falling back to a previously established backup point" we mean trapping back to this ERRSET by generating a LISP error. An unintentional LISP error will cause UCI-LISP to automatically enter its break package. But errors may be generated intentionally also, via the LISP function ERR. Such an intentional error is generated when the user hits the escape key on the console while at the IMPLY-STOP. This causes the call

```
(ERR (SETQ GLOBALERROR 1))
```

to be executed, which generates a LISP error trapped by the ERRSET in TRAP. The effect will have been to "peel back" the LISP stack to the state it was in when this ERRSET was entered. TRAP will then print the message RESTORED and reestablish the backup point which has just been fallen back to by calling itself. Note that the QED-LT will also be set to NIL at this time. TRAP is called for the first time by the function CYCLE with LT = (QUOTE CYCLE) to establish the initial backup point. CNTRL calls TRAP whenever the argument LT of IMPLY is a memer of TRAPLTS. TRAP may be called at the IMPLY-STOP to establish a backup point by simply typing a "B".

INTERPRET-A(X)

X is a non-NIL value of IMPLY, an answer, and thus has the form (S DB) where S is a substitution and DB is a data base. If DB = NIL or the A-unit of DB is NIL then

(a) if the program is in QED-mode, this value of X will be returned as the value of IMPLY on the current subgoal.

(b) Otherwise, the parameter LT will be set to (QUOTE B) and control will be transferred to the IMPLY-STOP by the statement (GO EVLOOP).

If DB = NIL and the A-unit of DB is non-NIL then INTERPRET-A will check to see whether it is time to propose the value of a set variable which IMPLY has been trying to solve for (see [1]). It does this by searching over all the set variables being accumulated in the A-unit slot of the data base and asking if any of them has been PROPOSED yet. If the answer is yes, then INTERPRET-A calls PROPOSE to let the user know this fact.

PROPOSE(S)

S is the total accumulated value of a set variable as generated by the set building rules of [1]. (See description of PROPOSED.) Syntactically S will look, for example, like this:

```
(SET AA Z (& (P Z) (\ (= Z A)))).
```

PROPOSE will then print

```
PROPOSE AA = (E Z (& (P Z) (\ (= Z A))))
```

and set the global variable PROPOSAL to S, which will later be detected by INTERPRET-I.

PROPOSED(A)

When PROPOSED is called by INTERPRET-A, A is the name of a set variable which has received some partial value via the set building rules of [1]. PROPOSED returns T if the set variable A does not occur in any subgoals which are at a higher level on the proof tree being built by IMPLY; for, if this is the case, then no further contribution to A can be made and the accumulated partial value must be the total value. The proof tree above the current subgoal is searched by the function PCNTRLCALL which looks back on the LISP stack for all those calls to IMPLY which were made with LT = (QUOTE CNTRL) to see if the set variable A occurs in their hypothesis or conclusion.

OPTIONS

The function OPTIONS is called from the IMPLY-STOP only if the value of LT is (QUOTE B) when execution of the function IMPLY reaches the PROG label EVLOOP. OPTIONS will immediately print the message IMPLY-STOP and prompt the user for input. At this point the user may exercise any of the following options.

NAME	SYNTAX	EFFECT
Proceed	line feed	Causes IMPLY to proceed as if no stop had been made.
Backup	escape	Causes IMPLY to fall back to the most recently established backup point. If IMPLY is currently at a backup point, control will fall back to the backup point prior to the current one.
Escape	^^	Causes IMPLY to do an error exit and return to the LISP top level.
Eval	E	Causes IMPLY to enter a LISP eval-loop. This is useful for "going behind the scenes" to debug. The loop may be terminated by typing "OK", causing control to retun to the IMPLY-STOP.
Establish a backup point	B	Causes IMPLY to insert a backup point in the proof tree.
QED	QED	Causes IMPLY to enter QED-mode by setting QED-LT to T. If the proof of the current subgoal is successfully completed, the message QED will be printed. Otherwise, the message QED? will be printed. In either case, QED-LT will be set to NIL (a return to interactive mode) and control will transfer to the IMPLY-STOP.
Assume	A	Causes the empty substitution to be returned as the answer for the current subgoal.
Fail	F	Causes the current subgoal to fail by returning NIL as its answer.
Reject an answer	REJECT	Causes the answer for the current subgoal to be rejected by setting BIGX, the variable which contains the answer, to NIL.
Print the theorem	TP	Causes the theorem to be printed.
Print the hypothesis	H	Causes the hypothesis to be printed.

Print the conclusion	C	Causes the conclusion to be printed.
Print hypotheses added by forward chaining	HFC	Causes the new hypotheses of this subgoal which were generated by forward chaining to be printed.
Print the data base	DB	Causes the data base to be printed.
Print the answer	X	Causes the answer for the current subgoal to be printed. In case the answer is the empty substitution, a blank line will be printed.
Print the theorem label	TL	Causes the current theorem label to be printed.
Turn on/off FC-LT	FC	Causes FC-LT to be set to (NOT FC-LT). Initially, FC-LT = NIL. When FC-LT = T, the forward chaining routines will be called whenever the rule of promotion is used.
Claim	CLAIM	Causes the function CLAIM to be called, which will prompt the user by typing NEW GOAL:. The user is then expected to enter (in prefix) a formula which he would like to prove instead of the current subgoal. If IMPLY succeeds in establishing this new goal, the message ESTABLISHED CLAIM will be printed. IMPLY will then be called with this new goal being added as a new hypothesis. If IMPLY does not succeed the message COULD NOT ESTABLISH CLAIM will be printed.
Define conclusion	DC	Causes the definition of the main predicate of the conclusion to be instantiated.
Define a predicate	D P	Causes the definition of predicate P to be instantiated throughout the current subgoal.
Reorder	(H => C) where H is a string of numbers or the atom H and likewise for C.	Causes the hypothesis and/or conclusion of the current subgoal to be reordered according to the scheme supplied. Each number ,n, corresponds to the nth hypothesis/conclusion as numbered by the TP option.
Reduce conclusion	R C	Causes the reduce rules to be applied to the conclusion of the current subgoal.

Reduce hypothesis	R H	Causes the reduce rules to be applied to the hypothesis of the current subgoal.
Use a lemma	USE	Causes the function USE to be called, which will prompt the user by printing LEMMA:. At this point the user may either type in an atom which he has previously SETQd to a lemma or he may type in a new lemma (in prefix). The lemma thus supplied will be added to the hypothesis of the current subgoal.
Instantiate a variable	PUT	Causes the function MAN-SUBST to be called, which will prompt the user by printing FOR WHAT:. The user is then expected to type in the name of a variable which occurs in the theorem. He will then be prompted by the message PUT WHAT:, in response to which he is expected to enter the value which he would like to have the variable receive.

Many of the options request the user's approval of their effect before this effect is finalized by a recursive call to IMPLY. This approval is sought by prompting the user with the message OK???. If the user approves of the effect which has been displayed to him, he should type OK; otherwise, he should type NO.

TESTR

As described above, the proceed and backup options each require a single keystroke, the line feed and the escape key, respectively. These keystrokes are read by the function TESTR, which uses the UCI-LISP function TYI to read a single character from the input buffer. If it reads a line feed, it returns (QUOTE PROCEED); if it reads an escape, it returns (QUOTE BACK); otherwise, it calls UNTYI to "unread" the single character and returns NIL.

INTERPRET-I(READ,BIGX)

This function is called by OPTIONS in case either the proceed or the backup option is used. If backing up is required, a LISP error will be generated as explained under the description of TRAP. If proceeding is required, the message PROCEEDING will be printed and it will be decided whether proceeding means returning an answer or simply continuing, by inspecting the variable BIGX. If BIGX is non-NIL it will contain the answer to the current subgoal and will be returned as the value of this call to IMPLY. If BIGX is non-NIL and the global variable PROPOSAL is non-NIL, then PROPOSAL will contain the total accumulated value of a set variable (see PROPOSE and PROPOSED) and INTERPRET-I will substitute this accumulated value for occurrences of the corresponding set variable throughout the theorem. The substitution is done by the function SUBST-A. The resulting subgoal is the beginning of pass two for the set variable, the subgoal whose proof will establish that the generated value does indeed satisfy the theorem.

THE ANSWER MECHANISM

The term "answer mechanism" refers to the generation, application, and composition of answers within IMPLY. An answer is the value of a call to the function IMPLY and has the form

```
(S DB)
```

where S is a substitution and DB is a data base. The substitution part of an answer is a set of substituiton pairs, {t1/x1, ... ,tn/xn}, for variables occurring in the proved subgoal. The data base part of an answer is an ordered n-tuple of units. In the set variable prover the data base is a 3-tuple: the A-unit, the restriction list, and the typelist. To simplify the description of the answer mechanism we first digress and discuss the data base accessing functions.

Since DB is an ordered data base of units, it is possible to write functions which will retrieve each unit without referring to that unit by name. In the set variable prover the functions which retrieve the three units are:

```
(AU (LAMBDA (DB) (CAR DB)))

(RL (LAMBDA (DB) (CADR DB)))

(TY (LAMBDA (DB) (CADDR DB)))
```

Corresponding to these are the functions AUDB, RLDB, and TYDB, which retrieve these units from the DB part of an answer, (S DB). The function DB retrieves the DB part of an answer; thus, for example, AUDB is defined as

```
(AUDB (LAMBDA (X) (AU (DB X))))
```

where X is expected to be an answer.

We are now ready to consider the functions which belong to the answer mechanism.

Generation of answers

ANS(ANSARG)

The function ANS is called throughout the program to return generated answers in the canonical form, (S DB). Note that it is an FEXPR and thus may take a variable number of arguments. ANS expects that its first argument will be a substitution and that its remaining arguments will be pairs of the form <U e>, where U is the name of one of the units of the data base and e is an expression whose evaluation will produce the value for the unit U. The names of the units are bound to the global variable UNITS which is set in INITIALIZE. For the set variable prover, UNITS = (AU PL TY). ANS performs a MAPCAR over UNITS and calls ASSOC to pick off units which are receiving new values as part of the answer, ANSARG. Any unit which is not receiving a new value in ANSARG will be represented by a NIL in the result of the MAPCAR. This insures that the DB part of the returned answer will be ordered by UNITS. A typical call to ANS is (ANS T NIL), which will return

```
(T (NIL NIL NIL))
```

if the variable UNITS has length three. See, for example, the body of IMPLY where we find the clause

```
((EQUAL C TRUE) (ANS T NIL)).
```

Application of answers

There are three situations in which answers are not returned to higher subgoals, but rather applied to subgoals at the same level. These three situations are an and-split in the conclusion, an or-split in the hypothesis, and an application of the rule of back chaining. As explained above, an answer is a pair (S DB). Corresponding to this pair is the pair of functions APPLY-SUB and DB-A which apply an answer in the aforementioned situations.

APPLY-SUB(A,B)

A is an answer and B is a formula to which the substitution part of A is to be applied. If the substitution part of A, (SIG A), is the empty substitution (denoted by T), then APPLY-SUB simply returns B. Otherwise, APPLY-SUB* is called to apply the substitution (SIG A) to B.

APPLY-SUB*(A,B)

This function applies the substitution A to all levels of the formula B. It recursively decomposes B until it reaches the atomic level and then calls SUB2 to check whether a substitution pair is present for the given atom. Note that this makes use of our syntactic convention that Skolem variables in a formula are represented by atoms in LISP. The function REMQV is called if B contains a quantified subexpression. REMQV returns the substitution A minus the substitution pair which gives a value to the quantified variable, if there is such a pair. This is an unlikely circumstance and could be completely avoided by the use of unique variables.

DB-A(X,DB)

This function returns a data base to be used in proving the second subgoal of an and-split in C, an or-split in H, or an application of back chaining. If (DB X) is NIL then the data base returned is the result of applying the substitution part of the answer X to the old data base, DB. Otherwise, the value is a new data base (e1 e2 e3) where

e1 is
(a) If (AUDB X) is non-NIL then (AUDB X)

(b) Else the result of applying the substitution part of X to (AU DB).

e2 is the merge of the restriction lists of (DB X) and DB. This merge retains the most recent restrictions on variables if restrictions exist in both restriction lists. The most recent restrictions are those found in the restriction list of (DB X).

e3 is the typelist unit of DB.

Composition of answers

COMPOSE(X,Y)

The composition of the two answers X and Y is, of course, again an answer and hence, if non-NIL, is generated by a call to ANS within COMPOSE. The substitution part of this answer is computed as the composition of the substitution parts of X and Y with respect to the list of protected variables, PV. If this composition is inconsistent (see COMPOSE-SIG) then the value of COMPOSE will be NIL. Otherwise, in the set variable prover, the value of COMPOSE will be

```
(CLEAN (ANS Z <AU e1> <RL e2> <TY e3>))
```

where

Z = (COMPOSE-SIG (SIG X) (SIG Y))

e1 - The A-unit of the composed answer is the result of applying the substitution part of Y to the A-unit of X, if X has an A-unit.

e2 - The restriction list of the composed answer is
(a) The result of applying the substitution part of Y to those entries in (RLDB X) which are protected, if (DB Y) is NIL.

(b) The entries in (RLDB Y) which are protected, if (DB X) is NIL.

(c) The entries of (RLDB Y) which are protected, if (RLDB Y) is not NIL and neither (DB X) nor (DB Y) is NIL.

(d) The result of applying the substitution part of Y to the entries in (RLDB X) which are protected, if (DB X) and (DB Y) are non-NIL and (RLDB Y) is NIL.

e3 - Since the typelist is a hypothesis, it is not part of an answer; hence, e3 = NIL.

CLEAN(X)

This function takes an answer, X, as argument and returns a "cleaned up" answer, by removing useless units from the substitution part of X, and by removing unneeded restrictions from (RLDB X). The functions CLEANSIG and CLEANRL perform these tasks. (AUDB X) and (TYDB X) are returned untouched.

COMPOSE-SIG(X,Y)

This function returns the composition of the substitutions X and Y with respect to the list of protected variables, PV. All substitution pairs from both X and Y which are connected to PV are first collected in XX. If XX, viewed as a substitution, is not CONSISTENT, then COMPOSE-SIG returns NIL. Otherwise, X is set to (INTERSECT X XX), Y is set to (INTERSECT Y XX) and (STABILIZE (COMPOSE-S X Y)) is returned.

CONNECTION(S,L)

This function returns all substitution pairs from S, which are connected to variables belonging to the list L.

Examples

S	L	CONNECTION(S,L)
{a/x, b/y}	(x)	{a/x}
{a/x, f(x)/y, c/z}	(x)	{a/x, f(x)/y}

{a/x, f(x)/y}	(y)	{a/x, f(x)/y}

CONSISTENT(S)

If S = {t1/x1,...,tn/xn} is a substitution, then S is consistent provided that

P(t1,...,tn)

and

P(x1,...xn)

are unifiable.

Examples

{a/x, b/y}	consistent
{a/x, b/x}	inconsistent
{y/x, f(x)/y}	inconsistent

COMPOSE-S(X,Y)

If X = {t1/x1,...,tn/xn} and Y = {u1/y1,...,um/ym} then

(COMPOSE-S X Y) = {t1Y/x1,...tnY/xn, u1/y1,...,um/ym},

where tiY, $1 \leq i \leq n$, is the result of applying the substitution Y to the term ti.

STABILIZE(S)

This function repeatedly composes the substitution S with itself, until the resulting substitution does not change under self-composition.

Examples

S	STABILIZE(S)
{a/x, b/y}	{a/x, b/y}
{a/x, f(x)/y}	{a/x, f(a)/y}
{a/z, z/x, f(x)/y}	{a/z, a/x, f(a)/y}

THE TRAPPING MECHANISM

The term "trapping mechanism" refers to the mechanism IMPLY uses to backtrack and recover from substitutions which lead to unprovable subgoals. The simple scheme which we adopt may not be practical in domains where backtracking is a persistent problem. However, it has been adequate for our purposes. In fact, it has been more than adequate: some of the examples have been contrived to illustrate features which may never be needed in actual practice.

Since IMPLY inspects the hypothesis of a theorem "from left to right", it is susceptible to falling into traps. For example, when proving

(Th) Pa & Pb & Qb => Px & Qx,

IMPLY will first fall into the trap of trying the instantiation a/x. when it fails to prove

(Th 2) Pa & Pb & Qb -> Qa,

it will backtrack, exclude the instantiation a/x, and prove (Th) with the instantiation b/x.

The heart of the trapping mechanism is the variable EXCLUDE. It appears as a parameter of each of the functions AND-C, OR-H, and TRYBACKCHAINING, the three functions of IMPLY which can initiate backtracking. EXCLUDE is simply a list of substitution pairs which have led to unprovable subgoals in the proof tree. The function UNIFY will never return a unifier which has a substitution pair belonging to EXCLUDE. Thus, in the example above, AND-C added a/x to EXCLUDE, which prevented UNIFY from solving the subgoal

(Th 1) Pa & Pb & Qb -> Px

with a/x. Instead, the substitution b/x was returned, which satisfied (Th 2) as well.

We are now ready to discuss the implementation of the trapping mechanism. A few definintions along with examples which illustrate them are presented first. Next the definitions of AND-C, OR-H, and TRYBACKCHAINING are presented, followed by a description of the auxiliary functions which they call. Finally, a few examples which exercise the built in mechanism are listed.

DEF

If S = {t1/x1,...,tn/xn} is a substitution then S is <u>self-conflicting</u> if

(UNIFY (P t1...tn) (P x1...xn)) = NIL

Examples of self-conflicting substitutions

{a/x, c/y, b/x}

{f(x)/x}

{f(x)/y, g(y)/x}

{f(x)/y, g(y)/z, h(z)/x}

DEF

If $S = \{t1/x1,\ldots,tn/xn\}$ is a substitution and B is a formula then the component ti_1/xi_1 of S <u>meets</u> B if there is a subset $\{i_1,\ldots i_k\}$ of $[1..n]$ such that

x_{i_1} occurs in t_{i_2}
and
x_{i_2} occurs in t_{i_3}
.
.
.
and
$x_{i_{k-1}}$ occurs in t_{i_k}
and
x_{i_k} occurs in B.

Examples

S	B	components of S which meet B
{a/x, b/x}	P(x)	{a/x, b/x}
{a/x, b/x, c/y}	P(y)	{c/y}
{f(x)/y, g(y)/z, h(z)/x, j(x)/u}	P(x)	{f(x)/y, g(y)/z, h(z)/x}

DEF

If $S = \{t1/x1,\ldots,tn/xn\}$ is a substitution and B is a formula then S is <u>in conflict with B</u> if the components of S which meet B form a self-conflicting substitution.

Examples

S	B	S in conflict with B
{a/x, b/x}	P(x)	yes
{a/x, b/y, c/y}	P(x)	no
{f(x)/y, g(y)/x}	P(x)	yes
{f(x)/y, g(y)/z, h(z)/x}	P(x)	yes

The functions AND-C, OP-H, and TRYBACKCHAINING.

The function AND-C is called whenever the current subgoal is of the form H => A & B. The function OR-H is called whenever H contains a hypothesis of the form A B.

The function TRYBACKCHAINING is called whenever HOA encounters an arrow hypothesis. The comments refer to the numbered examples at the end of the section.

```
(AND-C (LAMBDA (EXCLUDE)

   If [H -> A]Fix-pv(A,B) returns X then

      if [H -> BX]Fix-pv(A,B) returns Y then

         if COMPOSE(X,Y) = NIL then return NIL

         else if COMPOSE(X,Y) ∩  EXCLUDE = NIL
              then return COMPOSE(X,Y)

         else (* EX1 *)
              <TEMP <- Select-intersect(EXCLUDE,X,Y)
               return AND-C(EXCLUDE  ∪ {TEMP})>

      else if X = T or B = BX then return NIL

      else if [H -> B]Fix-pv(A,B) returns Y then

              if Y = T then (* EX2 *)
                   <TEMP <- Car(Sig-occur(X,B))
                    return AND-C(EXCLUDE  ∪ {TEMP})>

              else if in-conflict-with(X ∪ Y,B) then (* EX3 *)
                      <TEMP <- Break-conflict(X ∪ Y,B)
                       return AND-C(EXCLUDE ∪ {TEMP})>

              else return NIL

      else return NIL

   else return NIL




(OR-H (LAMBDA (EXCLUDE)

   If [H & A -> C]Fix-pv(A,B) returns X then

      if [H & BX -> C]Fix-pv(A,B) returns Y then

         if COMPOSE(X,Y) = NIL then return NIL

         else if COMPOSE(X,Y) ∩ EXCLUDE = NIL
              then return COMPOSE(X,Y)

         else (* EX4 *)
              <TEMP <- Select-intersect(EXCLUDE,X,Y)
               return OR-H(EXCLUDE ∪ {TEMP})>

      else if X = T or B = BX then return NIL

      else if [H & B -> C]Fix-pv(A,B) returns Y then
```

```
            if in-conflict-with(X U Y,B) then (* EX5 *)
               <TEMP <- Break-conflict(X U Y,B)
                return OR-H(EXCLUDE  U  {TEMP})>

            else return NIL

        else return NIL

    else return NIL



(TRYBACKCHAINING (LAMBDA (EXCLUDE)

    If TRIB(B,C) returns (X ,H', C') then

       if [H => H'X]Fix-pv(H',C') returns Y then

          if COMPOSE(X,Y) = NIL then return NIL

          else if COMPOSE(X,Y) ∩ EXCLUDE = NIL
               then return COMPOSE(X,Y)

          else (* EX6 *)
               <TEMP <- Select-intersect(EXCLUDE,X,Y)
                return TRYBACKCHAINING(EXCLUDE U {TEMP})>

       else if [H => H']Fix-pv(H',C') returns Y then

          if Y = T then (* EX7 *)
                <TEMP <- Car(Sig-occur(X,H'))
                 return TRYBACKCHAINING(EXCLUDE U {TEMP})>

          else if in-conflict-with(X U Y,H') then (* EX8 *)
                  <TEMP <- Break-conflict(X U Y,H')
                   return TRYBACKCHAINING(EXCLUDE U  {TEMP})>

          else return NIL

       else return NIL

    else return NIL
```

The auxiliary functions called by AND-C, OR-H, and TRYBACKCHAINING.

BREAK-CONFLICT(S,B)

When this function is called by AND-C (OR-H, TRY-BACKCHAINING), S = {t1/x1,...,tn/xn} will be a substitution which is in conflict with B. The value returned is the first component ti/xi of S such that $\{t_{i+1}/x_{i+1}$, ...,tn/xn} is not in conflict with B.

FIX-PV(A,B)

This function returns the union of PV, the current set of protected variables, with the variables which the two formulas A and B have in common. The notation

    [H => A]Fix-pv(A,B)

denotes a call to the function IMPLY with Fix-Pv(A,B) as the value of the LAMBDA-variable PV. See the section on protected variables.

IN-CONFLICT-WITH(S,B)

This function checks to see whether the substitution S is in conflict with B. If S is in conflict with B, the value returned is the set of components of S which meet B. By definition the substitution formed by these components is self-conflicting.

SELECT-INTERSECT(EXCLUDE,X,Y)

This function returns the first component ti/xi of X such that (ti)Y/xi is a member of EXCLUDE. When it is called by AND-C (OR-H, TRYBACKCHAINING), it is known that there will be such a component.

SIG-OCCUR(S,B)

This function returns {ti/xi ε S :xi occurs in B}.

TRYB(B,C)

If B has one of the forms

(a) A => C'

(b) A => (P => C')

(c) A => $X_S \varepsilon$ S

(d) A => P=Q

and, if in the corresponding case,

(a) U <= ANDS(C',C)

(b) U <- ANDS(C', C)

(c) U <- HOA-SET($X_S \varepsilon S$,C)

(d) U <- EQ-H2(DB, H = {A -> P=Q}, C, P, Q)

is not NIL, then TRYB will return a triple (X, H', C') where

X = COMPOSE(U,V) where

V is

(i) ANS(T,NIL) if (SIG U) = T or (RL DB) = NIL

(ii) If (SIG U) = T and (RL DB) = NIL, then there is a possibility that (SIG U) will contain instantiations for variables which have restrictions in (RL DB). For example, consider

(Th) a < b < c & Qb & x(Qx -> Pcx) -> $\exists$y (a < y < b & Pyb).

(Th 1) will place the restriction ('int' v a b) on y. Hence, when back chaining is used in (Th 2) we must first verify that a < c < b in order to validly make the instantiation c/y. TRYB calls TEST-DB to perform the verification. The verification is carried out by IMPLY and the answer is placed in V. If this answer is NIL (could not verify) then the value of TRYB will be NIL.

and

H' is

(a) A

(b) P & A

(c) A

(d) A

and

C' is

(a) C'

(b) C'

(c) $X_S \varepsilon S$

(d) P=Q

EXAMPLES of theorems (and non-theorems) which exercise the built in mechanism.

1. Pfx & Pfb & Qa & Rfb => (Py & Qx) & Ry

2. Px => x<6 & (P(3) & 4<x)

3. Pa & Pb & Qb => Px & Qx

4. (Py v Qx) v Ry => Pfx v Pfb v Qa v Rfb

5. Px v Qx => (Pa v Pb) v Qb

6. Qa & (Qx => Pfx) & (Qx => Pfb) & Rfb => (Py & Ry)

7. (x < 6 & 4 < x => Px) => P(3)

8. Pba & (Pxy => Qx & Qy) => Qa

9. Pfa & Pfy & Qfb => Px & Qx

10. Rfa => (Pfx & Qx => Py & (Qa & Qb)) & Ry

11. Qa => (Px => Pa & Pb) & Qx

NOTE: x,y, and z are variables; a and b are constants; f and g are functions.

THE PROTECTED VARIABLE MECHANISM

The term "protected variable mechanism" refers to the mechanism IMPLY uses to delete substitution pairs from answers, when these pairs contribute nothing to the solution of remaining subgoals. For example, consider

(Th) Px & Qy => (Pa & Pb) & Qb.

(Th 1) will be solved with the substitution {a/x, b/x}. However, since x doe not occur in the conclusion of (Th 2), there is no need to return this substitution as part of the answer of (Th 1). This examples serves as motivation for the definition of a protected variable.

DEF

A variable x is <u>protected</u> if

(a) It occurs in both conjuncts of a conjunctive subgoal H => C1 & C2. (Recall that we think of & as a binary connective.)

or

(b) It occurs in both disjuncts of a disjunct A B in the hypothesis of a subgoal.

or

(c) It occurs in both the hypothesis and the conclusion of an arrow hypothesis.

From the definitions of AND-C, OR-H, and TRYBACKCHAINING, we see that variables are added to the list of protected variables, PV, by the function Fix-pv in accordance with this definition. Referring to the definition of COMPOSE-SIG, we see that the substitution part of an answer will contain only those substitution pairs which have some CONNECTION to PV. The following are some examples which exercise the protected variable mechanism.

1. Px -> Pa & Pb

2. (Qhz -> (Pfx -> Pfgy & Pu) & Qy) & (Rgha -> Rx)

3. Rga -> (Qhu -> (Pfx -> Pfgy & Pz) & Qy) & Rx

4. Pfac & Qb & (Qy -> R(fxy,x)) -> Rva & Pv

NOTE: u,v,x,y, and z are variables; a and b are constants; f, g, and h are functions.

APPENDIX

The task of maintaining IMPLY is made simpler by splitting up the source code into a number of separate files. These files separate the main components of IMPLY so that they may be modified or replaced to suit the implementor's needs. The following is a brief description of the component(s) on each file of the set variable prover.

INIT.LSP

This file contains the function INITIALIZE, which performs the initialization necessary to run IMPLY. It is loaded automatically by UCI-LISP.

IMPLY.LSP

This is the main file of IMPLY. It contains most of the functions which comprise the control structure of IMPLY as described in this document. It also contains the functions which perform definitional instantiation and equality substitution.

UTLTY.LSP

This file contains various utility functions which are called in all of the other files. It would be desirable to compile these functions, since they are called so frequently. They are very stable and most of them are apt to be useful in any theorem prover based on IMPLY.

MATCH.LSP

The routines which IMPLY uses to do matching are contained on this file. It also contains the routines which will force checking if unification produces a value for a variable which has an entry in the restriction list (see [2]).

PROMOT.LSP

This file contains the routines which implement the rule of promotion:

```
     H => (P => Q)
  goes to
     P & H => Q.
```

In the set variable prover the additional hypothesis, P, is inspected for ground inequalities (call to SET-TYPE) which are added to the typelist unit of the data base. The function PROMOTE2 will call the forward chaining routines if FC-LT = T.

REDUCE.LSP

This file contains all of the reduce tables and is the most domain dependent component of IMPLY. The function REDUCE* is the driver routine. It keys on the main predicate of its argument, TH, and calls the appropriate subfunction. Note that P, the second argument of REDUCE*, is a parity indicator (either T or NIL) and is used as a free variable in reduce functions (eg. EL-REDUCE) which must call REMO to perform skolemization as part of a reduction.

FC.LSP

This file contains the forward chaining routines. The driver, FC, is called by PROMOTE2 if FC-LT = T. The function PEEK implements the rule of "peek forward chaining" by calling DEFNP to expand definitions.

OPTION.LSP

The function OPTIONS is on this file.

PRNT.LSP

All of the printing functions are on this file. Note that unique print names for Skolem constants are created by UNSKO* and placed, in dotted pair form, on the global variable USED.

PLE.LSP

This file contains the functions which are used to prove inequalities. Chief among these are PROVE-LE, PROVE-LE-GROUND-CASE, RESTRICTION-LE, and MATCH-LE (see [2]). None of the functions on this file are needed when using IMPLY to prove theorems which do not involve inequalities.

SUPINF.LSP

This file contains the routines SET-TYPE, to build the type-list unit of the data base, and SUP and INF to determine the truth of inequalities by inspecting the typelist. See [2].

SETUP.LSP, XEVAL.LSP, PUBLIC.LSP, XEVALI.LSP, XEVALR.LSP

These files contain the algebraic simplifier, XEVAL, written by Don Good at the University of Texas. SETUP.LSP is an initialization file. The last four files are the source code for XEVAL, which we always run in compiled form. XEVAL is called only by SIMP in the set variable prover.

EGS.LSP

This file contains a set of bench mark examples. They should be useful to anyone trying to understand the implementation of IMPLY.

IMPLY is run on a PDP-10 KI processor at UT Austin using UCI-LISP. It requires about 120K octal allocated as follows:

```
FULL WORD SPACE = default

BIN PROG SPACE = 7400

REG PDL = 2000

SPEC PDL = 2000
```

The 7400 words of binary program space are needed to accomodate the compiled version of the algebraic simplifier, XEVAL.

References

1. W. W. Bledsoe. A Maximal Method for Set Variables in Automatic Theorem Proving. The University of Texas Mathematics Department, Memo ATP-33A, July 1977.

2. W. W. Bledsoe, Peter Bruell, and Robert Shostak. A Prover for General Inequalities. The University of Texas Mathematics Department, Memo ATP-40, June 1978.