22 Jan 74

ATP

PROGRAM CORRECTNESS

The following is an explanation of routines that have been added to our interactive theorem prover ([1], [2]) for efficient handling of inequalities and proofs by cases which arise from proving correctness of computer programs.

1. TYPES

Certain variables such as i, j, k, l, m, n, are typed. These are the ones that serve as indices in computer programs, and for which inequalities are stated. The human user is asked to give a list of the variables to be typed at the beginning of the proof and they are placed in the list TYPE-ATOMS. Actually these will be skolem constants of the form

$$(j\_\_),\ (k\_),\ (\ell\_\_\_),\ \ldots$$

but the first letter of each of these (i.e., the "car") is stored in TYPE-ATOMS. At some point in the proof the program might ask the question

$$(\text{TYPED}\ (j\_\_))$$

in which case the answer is true if j is an element of TYPE-ATOMS.

The typed variables (or equally correct, we could say the typed skolem constants) will assume only non-negative integer values: 0, 1, 2, 3,... . The program will maintain an interval

(1) $$a \leq j \leq b$$

for each of them, which indicates that, at this stage in the proof, the variable j is restricted between the values a and b. a and b are either integers or they

may be expressions given in terms of other such typed variables, e.g.,

(2) $$N - 1 \leq j \leq k + 5.$$

This interval information is maintained in a list called TYPE-LIST, which represents the "state of the world" for these variables at that time. The inequality (1) (or (2)) is represented in TYPE-LIST by the expression

$$(j \text{ int } a \quad b)$$

which means simply that $a \leq j \leq b$. In this case, j is said to have been "typed" and has type [a,b]. This is not unlike the typing used in [2].

After the human has designated the members of TYPE-ATOMS, the program uses the subroutine ORIG-TYPE to give them their initial types

$$(j \text{ int } 0 \text{ inf})$$

where inf stands for ∞. Thus at the beginning we assume that

$$0 \leq j < \infty$$

for each j in TYPE-ATOMS.

The type (j int a b) of j may be changed as the proof proceeds. For example, if the theorem has a typothesis of the form

$$j \leq 3$$

it is used to convert the original type (j int 0 inf) to (j int 0 3). This is accomplished by the subroutine SET-TYPE which is called at the beginning of a proof and at other times when new material is added to the hypothesis of the theorem

being proved.

For example if we are proving the theorem

$$(3) \qquad (\alpha \wedge \beta \wedge 1 \leq j \wedge j \leq N \wedge j \leq 1 \longrightarrow (1 \leq K \wedge K \leq N - 1 \longrightarrow r))$$

the original types are ((j int 0 inf) (K int 0 inf) (N inf 0 inf)), but then SET-TYPE is called on (3) which uses

$$1 \leq j \wedge j \leq N \wedge j \leq 1$$

to change the types to ((j int 1 1) (K int 0 inf) (N int j inf)), and to convert (3) to

$$(4) \qquad (j = 1 \wedge \alpha \wedge \beta \longrightarrow (1 \leq K \wedge K \leq N - 1 \longrightarrow r)).$$

Note that the program has detected that $j = 1$ from its type (j int 1 1), and at a later point in the proof, the value 1 will be substituted for j throughout the theorem.

The formula (4) is then converted (by IMPLY Rule 5 (See Section 4 of [2]) to

$$(5) \qquad (j = 1 \wedge \alpha \wedge \beta \wedge 1 \leq K \wedge K \leq N - 1 \longrightarrow r)$$

at which time SET-TYPE is again called which uses $(1 \leq K \wedge K \leq N - 1)$ to change the types to ((j int 1 1)(K int 1 N - 1)(N int K + 1 inf), and converts (5) to

$$(6) \qquad (j = 1 \wedge \alpha \wedge \beta \longrightarrow r).$$

The typing information given in TYPE-LIST is derived from parts of the theorem's hypothesis and hence itself serves as <u>additional hypotheses</u>. Thus when

a contradiction such as (int j 2 1) occurs in TYPE-LIST, this represents a null hypothesis and thus terminates successfully the proof of the theorem. Also when an entry of the form

(j int 2 2)

occurs, the expression $j = 2$ is transferred back to the hypothesis of the theorem and is later used to replace j by 2 throughout the theorem. The subroutine REDUCE-TT is used to detect such contradictions and equalities in TYPE-LIST, and to generally update TYPE-LIST, accounting for the fact that the entries have effects on each other.

Since TYPE-LIST is dynamic, changing as the proof proceeds, it is carried as an additional argument of IMPLY.

Entries of the form

(j int a b)

which occur in TYPE-LIST are complicated by the fact that a and b may themselves be expressed in terms of other typed variables such as K, N, M, etc. Thus in any one case it may not be immediately obvious whether $b < a$ or $a = b$, and we have provided routines to establish <u>absolute</u> lower and upper bounds $\underline{a}$ and $\overline{b}$ for j,

$$\underline{a} \leq a \leq j \leq b \leq \overline{b}.$$

$\underline{a}$ and $\overline{b}$ (which are numbers) are obtained by calling the subroutines (GRE a) and (LES b) respectively.

For example, if the present types are

$$((j \text{ int } 1 \text{ } (\min 1\ N - 1)) \quad (N \overset{0}{\text{int}}\ 1\ 2))$$

then (GRE 1) = 1, and

$$(\text{LES } (\min 1\ N - 1)) = (\min 1\ (\text{LES } N - 1))$$

$$= (\min 1\ ((\text{LES } N) - 1))$$

$$= (\min 1\ ((\text{LES } 1) - 1))$$

$$= (\min 1\ (1 - 1))$$

$$= (\min 1\ 0) = 0$$

Hence

$$1 \leq 1 \leq j \leq (\min 1\ (N - 1)) \leq 0$$

$$\underline{a} \quad a \qquad\qquad b \qquad\qquad \overline{b}$$

Since $1 \not\leq 0$, this would successfully terminate the proof.

In all such manipulations a simplification routine SIMP is used to force algebraic expressions into canonical form. Such simplification routines are invaluable in proofs in analysis since they avert the need for adding to the hypothesis the field exams for real numbers.

2. CASES

Many of the theorems (verification conditions) arising in program correctness require proof by cases. For example, Example _____ below requires the cases $K = 1$ and $K \neq 1$. Our program handles this in the following way.

When proving a theorem $(H \rightarrow C)$ a call is made to (IMPLY H C)*. If C has

*Actually IMPLY also has the argument TYPE-LIST but we have suppressed that in this write-up.

the form of an inequality

$$(A \leq B)$$

then an appeal is made by IMPLY to the routine SOLVE$\leq$, which tries to determine whether $(A \leq B)$ is true by consulting TYPE-LIST. If it cannot completely verify $(A \leq B)$ it might still be able to <u>partially</u> verify it; in which case the program returns a message indicating the part yet unverified which might be proved in some other way.

For example, if TYPE-LIST is equal to

((j int 1 2)(K int 1 inf))

i.e.,

(7) $$(j \leq 1 \ \wedge \ 1 \leq K)$$

and we are trying to prove

(8) $$(H \rightarrow 2 \leq K)$$

then SOLVE$\leq$ will return the expression $(K = 1)$ which represent the CASE not handled. That is, for the case $K \geq 2$ (8) follows, and (7) shows that the case $K = 0$ is impossible, so the case $K = 1$ is the only one left. In this way $(K = 1)$ can be added to the hypothesis in alternate proof routes. These proceedures which have been developed to handle CASES are based on the following theorems which are implemented by the rules shown.

<u>Theorem 1</u>. $$(P \rightarrow Q) \wedge (\sim P \rightarrow R) \rightarrow Q \vee R$$

RULE 1. When attempting (IMPLY H (Q $\vee$ R))
If (IMPLY H Q) returns (P, σ_1)*
and (IMPLY H (P $\rightarrow \tilde{Q}\ \sigma_1$)) returns σ_2
then return $\sigma_1 \circ \sigma_2$ for (IMPLY H (Q $\vee$ R))

Example. If (IMPLY H Q) returns (K = 1),
and (IMPLY H (K = 1 $\rightarrow$ R)) returns "TRUE"
then (IMPLY H (Q $\vee$ R)) returns "TRUE".

Theorem 2. $(P \wedge A \rightarrow C) \wedge (\sim P \wedge B \rightarrow C) \rightarrow (A \wedge B \rightarrow C)$

RULE 2. When attempting (IMPLY (A $\wedge$ B) C)**

(a) If (IMPLY A C) returns (P, σ_1)

(b) and (IMPLY B (P $\rightarrow C\sigma_1$)) returns σ_2

(c) then (IMPLY (A $\wedge$ B) C) returns $\sigma_1 \circ \sigma_2$.

Example. Suppose TYPE-LIST = ((j int 1 2)(k int 1 3)) and we are trying to evaluate

(9) (IMPLY [(2 $\leq$ K $\rightarrow$ C) $\wedge$ (K $\leq$ 1 $\rightarrow$ C)] C).

RULE 2a: (IMPLY (2 $\leq$ K $\rightarrow$ C) C) returns (K = 1) (see below).

RULE 2b: (IMPLY (K $\leq$ 1 $\rightarrow$ C)(K = 1 $\rightarrow$ C)) returns "TRUE" (see below).

RULE 2c: Gives "TRUE" as the result for 9.

To see how (IMPLY (2 $\leq$ K $\rightarrow$ C) C) returns (K = 1), the program first backchains to get

*Here σ_1 is a substitution of the form usually returned by IMPLY and P is an expression of the form under discussion here. For example, P might be (K = 1).

**Actually this rule is implemented in HOA a subroutine of IMPLY.

(IMPLY NIL $2 \leq K$)

which returns $K = 1$ as explained above, since the typing of K assures that $1 \leq K \leq 3$. Also (IMPLY $(K \leq 1 \rightarrow C)(K = 1 \rightarrow C)$) is converted to (IMPLY $(K = 1 \wedge (K \leq 1 \rightarrow C))$ C) and then to (IMPLY $(1 \leq 1 \rightarrow C)$ C) and to (IMPLY C C) which is "TRUE".

These rules, when installed as indicated in IMPLY, cause the machine to automatically do cases. Also there is an interactive command (CASES K i <) which allows the user to force proof by cases. This can be used to override or augment the program's automatic CASES proceedure.

3. DETAILS

These remarks will refer to certain routines already in PROVER and others that we are now adding. The reader is assumed to be generally familiar with PROVER.

3.1. CYCLE

CYCLE is a routine which calls the main routine IMPLY. Before the theorem is placed in skolem form, preparing it to be sent to IMPLY, it should be SPLIT if possible, thus sending only those theorems that cannot be further SPLIT.

3.2. TYPE-ATOMS

After the theorem is skolemized and before it is sent to IMPLY, the human user is asked to designate the members of TYPE-ATOMS. He does this by considering expressions of the form ($\leq$ a b), ($<$ a b), ($\geq$ a b), ($>$ a b), which occur in the theorem, and choosing from them the first entry (i.e., the "car") of each skolemized expression. Thus if j has been skolemized as (js x y) and

$$(\leq (js\ x\ y)\ N)$$

occurs in the theorem then js is placed in TYPE-ATOMS. At this point the program also calls (ORIG-TYPE TYPE-ATOMS) and (SET-TYPE H) as explained in Section 1, and puts C-LIST to NIL.

3.3. CASES

IMPLY (and HOA) ordinarily returns a substitution σ (see [1] and [2]) which represents a set of instantiations for variables in the theorem (or subgoal) being proved. The entries of σ are of the form $(x \cdot t)$, where t is a term to be substituted for the variable x.

In the augmented systems expressions of the form

$$(TL\ j\ int\ a\ b) \quad or \quad (TL \leq a\ b)$$

(where TL is the current theorem label) are placed in a list, called C-LIST, by the routine SOLVE$\leq$ (see above).

Expressions of the form $(TL \leq a\ b)$ which occur in C-LIST cause no harm, and have no effect unless acted upon by the human user at IMPLY-STOP. The user can treat these as information which he may or may not use in deciding what to do next (cases he might try, etc.).

On the other hand expressions of the form (TL j int a b) which occur in C-LIST must all be removed before IMPLY reports back to CYCLE. These represent cases that have not yet been handled by the program, and these cases must be dealt with before the proof is complete. Thus at each OR-BRANCH in IMPLY and HOA, we look for the occurance in C-LIST for expressions of the form (TL' j int a b), where the current theorem label TL is an initial part of TL', and if found try to clear

them out. For example, if the current theorem label is (121) and we are trying to prove the subgoal

$$(A \wedge B \rightarrow C),$$

(which is equivalent to $((A \rightarrow C) \vee (B \rightarrow C))$), (IMPLY (A $\wedge$ B) C) calls (HOA (A $\wedge$ B) C) which would try (HOA A C) and (HOA B C). If (HOA A C) succeeds there is no need for a call to (HOA B C) unless the call to (HOA A C) has placed on C-LIST an entry of the form

(1) ((12122) j int a b).

In that case, a new call is made with (j int a b) as an added hypothesis, i.e.,

(2) (IMPLY ((j int a b) $\wedge$ B) C).

If (2) succeeds then the entry (1) is eliminated from C-LIST and the proof proceeds, if not the entry (1) is left in C-LIST and the proof proceeds. ~~(HOA (A $\wedge$ B) C), which calls (HOA A C), (HOA A C) returns~~

~~(3) c = ((x.t)(j int a b)),~~

~~then a call is made to~~

~~(4) (IMPLY ((j int a b) $\wedge$ B) C),~~

~~(i.e., we try again to prove C using the second hypothesis B and the new hypothesis (j int a b)). If (4) succeeds then (j int a b) is eliminated from (3), if not it is retained and passed back as the answer to (1).~~ This is explained by Rules 1 and 2 of Section 2 above, and is more precisely detailed in the LISP corrections given for

IMPLY and HOA. It is also depicted in the flowchart in Figure 1. Figure 1 also applies to goals of the form

$$(A \wedge B \rightarrow C)$$

since this is equivalent to

$$(A \rightarrow C) \vee (B \rightarrow C).$$

SEE ATTACHED CHART

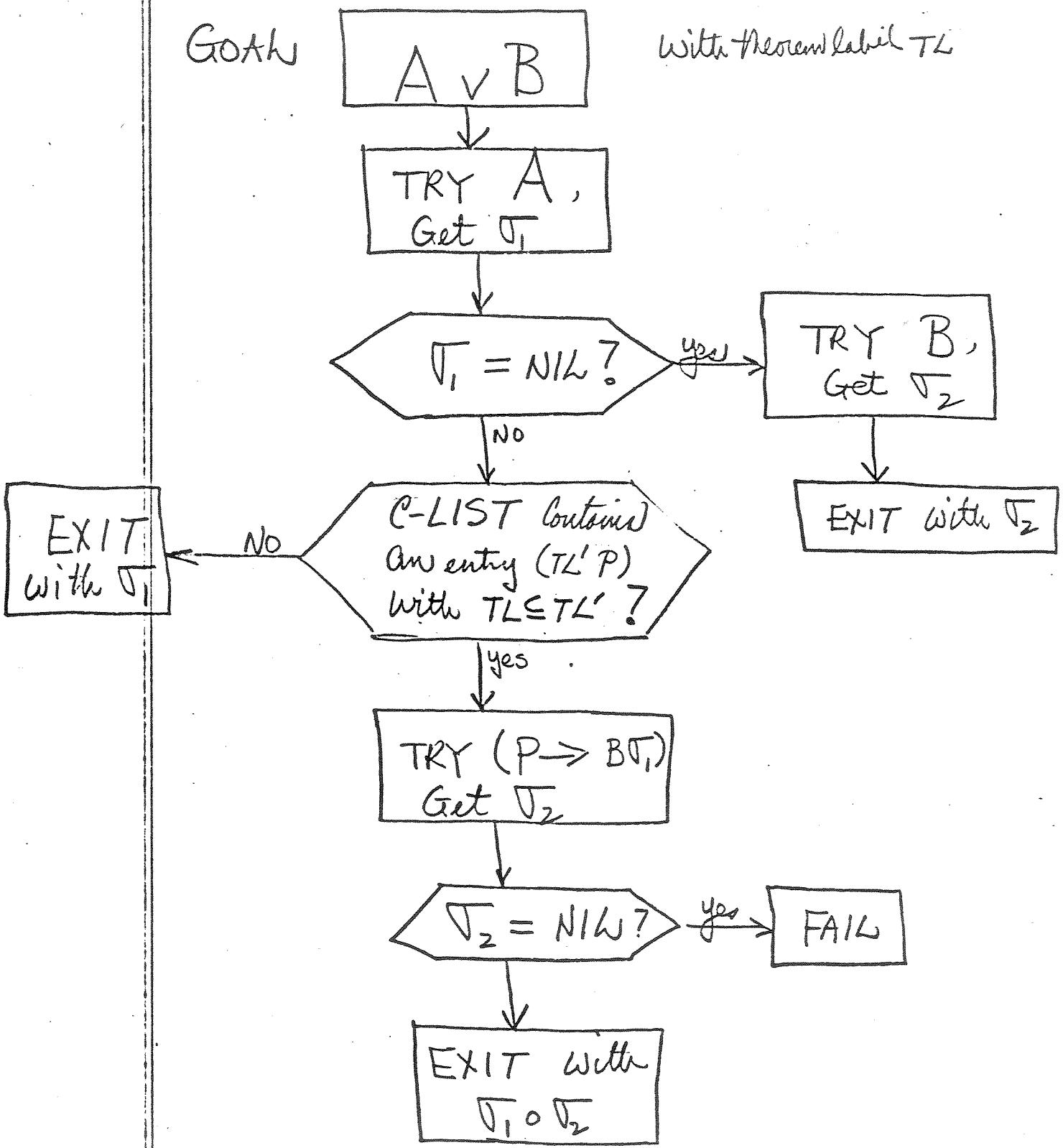

Figure 1
Flow Chart for CASES

P represents a case not handled in the proof of A.

3.4. SIMPLIFICATION

The simplification routine SIMP is designed to place algebraic expressions in canonical form, and in doing so like terms with different signs are cancelled (e.g., (7 + B + A + (-B)) goes to A + 7). SIMP now only handles + and -. Earlier simplification routines in [2] handled $\cdot$ and $\div$ as well, and the present one may be extended when needed.

A list, SORT-LIST, is kept which contains all expressions that have already been simplified. Thus a check is first made to see if the formula being simplified is already on SORT-LIST. If not, it is simplified and then placed there.

The basic function of SIMP is a sorting routine. SORT tries to place expressions in canonical form. It depends on a list L+ which tells which atoms preceeds others in its ordering. For example if L+ = (A B C) then the expression ((A+C) + B) is simplified to (A+(B+C)), whereas if L+ = (C A B) then it is simplified to (C + (A+B)). Numbers are always placed last, and ordinary additions and substractions are made.

SORTT is a routine that counts the number of calls to SORT, forcing it to stop after 12 attempts (usually two or three will suffice). SORTL shecks to see if SORT-LIST already has the expression and if not calls SORTT.

The basic rules used by SORT are shown in Table 1.

4. CASES (by hand)

The following is an interactive command

(CASES K i >)

where K is an atom which has been "typed" (i.e., is a member of TYPE-ATOMS), i

is an integer (like 0, 1, or 2), and $>$ might also be replaced by $<$.

This causes the computer to call the two subgoals

$$(K = i \wedge H \rightarrow C)$$

and

$$(i+1 \leq K \wedge H \rightarrow C)$$

instead of the current goal $(H \rightarrow C)$. That is the machine calls

```
(IMPLY NIL (∧ (→ [∧ (= Ki) H] C)
              (→ [∧ (≤ i+1 K) H] C))  )
```

Before the user gives the CASES command he needs to see those entries of C-LIST which are related to the current theorem label. He does this by the interactive command

CL

which causes the machine to print out all entries (TL' j int a b) of C-LIST where the current theorem label TL is a subset of TL'. The machine finds and prints these by calling

(PRINT (car (FIND-CS C-LIST TL))).

References

1. W.W. Bledsoe and Peter Bruell, "A Man-Machine theorem proving system", IJCAI-73.

2. W.W. Bledsoe, Robert S. Boyer, and William H. Henneman, "Computer Proofs of Limit Theorems", Artificial Intelligence 3(1972), 27-60.

NOTES

Change	GRE	to	INF	
	LES	to	SUP	
	INF	to	INFINITY	
	REDUCE-TT	to	UPDATE	
	(j int a b)	to	(j a b)	
	TYPE-LIST	to	TY	(maybe not)

Later