# A UNIT RESOLUTION THEOREM PROVING SYSTEM

by

PETER NEAVE LEQUESNE

B.Sc., University of British Columbia, 1963

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the Department

of

Computer Science

We accept this thesis as conforming to the

required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April, 1972

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date April 22nd, 1972

## Abstract

A unit resolution theorem proving system is developed and compared with the previous work of C.L. Chang. This thesis includes a description of a particular approach to unit resolution and a description of the resulting program and its effectiveness.

Table of Contents

# List of Tables

## Acknowledgement

Most of the ideas developed in this thesis are the suggestions of Professor Raymond Reiter. I am deeply grateful for the patient assistance he has given me in completing this work.

Thanks are also extended to the Department of Computer Science for a most satisfying two years of study and to Mrs. Olwen Sutton for an excellent typing job.

# I Concepts and Terminology

## 1. Introduction

Application of the full resolution proof procedure normally results in a very large amount of matching and clause generation. Unit resolution is a means of reducing the size of such problems. This work is an implementation of a particular form of unit resolution, paralleling closely that of C.L. Chang [2]. This Chapter outlines the theoretical basis of the program produced and describes the notation and terminology used. A full description of resolution is not attempted however unit resolution will be described as a special case of the more general technique.

## 2. Preliminary Formulation of the Axioms and Proposed Theorem.

For any resolution procedure, the axioms and proposed theorem of a given system must be expressed in a particular format for expressions of first order mathematical logic called conjunctive normal form. Procedures for converting to conjunctive normal form are given in [3] and [8]. A conjunctive normal form (cnf) has the following structure.

Let $C_1,\ldots,C_k$ be logical phrases called clauses. Then a c.n.f. has the following form:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

That is, a c.n.f. is a conjunction of clauses. A clause is a disjunction of literals. Say $L_1,\ldots,L_M$ are literals then a clause has the following structure for $M > 0$: $L_1 \vee \ldots \vee L_M$. A literal is a logical phrase of the following general form, where + or - indicates a sign, P a predicate symbol, and $t_1,\ldots,t_k$ are expressions

called terms: $\pm P\ t_1,\ldots,t_k$ for some $k \geq 0$. A term may be a free variable, a constant or a functional value where a functional value is a functional symbol followed by a number of terms in brackets, e.g., $f(t_1,\ldots,t_\ell)$ for some $\ell \geq 0$. Consider the following examples of

a) a free variable, x

b) a constant, 3

c) a functional value, g(x,3)

d) a literal, + T x g(x,3) x

e) a clause, + P x g(x,3) x V + Q x, g(x,5), 5

f) a c.n.f.,

+ P x g(x,3) x V + Q x g(x,5) 5 ∧ + T x g(x,3) x

For purposes of programming in the LISP 1.5 language the following notation is used for the date types described above

1) A free variable is designated by the letter x followed by any number, e.g. x20, x15.

2) A constant is any other letter or an integer, e.g. A,B,C,1,15,10.

3) A functional value has the following list form $(F\ t_1\ t_2 \ldots t_k)$ for example (F 1 x3 10) or (G x1 x2).

4) A literal is again a list but always begins with a + or - sign followed by a predicate symbol. For example: (+ P x1 x2 x3) or (- Q (F x1) 2)

5) A clause is simply a list of its literals. Further information on input of the c.n.f. to the program is given in Chapter 2.

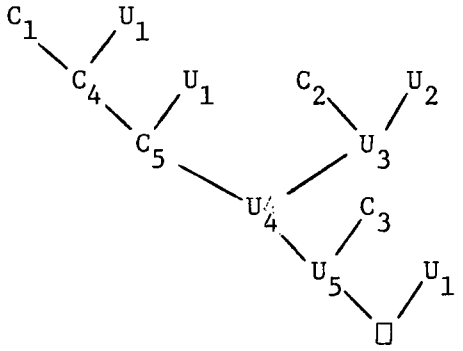
Fundamental to this thesis is the concept of a unit clause. A unit clause is a clause of one literal. For instance (( - P x1 x2)) where one set of brackets denotes a clause and the other a single literal, is a unit clause. In practise, for example, in the context

of the program written, the outer set of brackets is dropped.

For a unit resolution theorem prover to proceed on a proposed theorem and associated set of axioms at least one unit clause must be present in the associated conjunctive normal form.

3. The Unit Proof

A resolution proof is normally sketched as an inverted binary tree whose bottom-most node is the □ symbol. A unit proof is defined as a resolution proof such that for each node generated downwards in the proof tree, at least one parent node is a unit clause. For example if we denote unit clauses as $U_i$, regular non-unit clauses as $C_j$ and have the following c.n.f.; $C_1 \wedge U_1 \wedge U_2 \wedge C_2 \wedge C_3$; then <u>if</u> a proof for this c.n.f. is found as follows,




we then have a proof by <u>unit</u> resolution.

Normal resolution is a process combining clauses to form new clauses until two clauses combine to nil or □. In the above hypothetical diagram, three new unit clauses ($U_3$, $U_4$, $U_5$) and two new non-unit clauses ($C_4$, $C_5$) are generated. In each case the new clause has at least one parent which is a unit clause. Since we are describing what is really a combining process an advantage of unit resolution is apparent. For any given resolution the length of

the newly generated clause (i.e., the number of literals which it contains) will be one less than that of the non-unit parent. An approach which attempts to make such unit resolutions first in search of a proof is termed the unit preference strategy [1]. The theorem proving system developed in this work does not attempt non-unit resolution and hence will not succeed on problems which do not have a unit proof. That is, it will not prove proposed theorems whose c.n.f. has no unit clauses or whose proof is dependent on the resolution of two non-unit clauses.

4. Unification and Substitution for a Unit Proof.

Essentially, unification is the process of finding a common instance of two sets of terms. For the purposes of resolution these two sets of terms appear in separate clauses and are associated with the same predicate symbols of differing signs. This common instance of two sets of terms is demonstrated by the generation of a substitution to be made to the two clauses in question, the result of which is the presence of contradictory literals within the clauses in question, which are then combined to form a resolvent (minus the contradictory literals). Further to the concept of a unifying substitution, is the idea of a most general unifier. This notion which is developed in (10) basically means that there exists a substitution for two unifiable literals such that the most general common instance of the two is the result.

In normal resolution systems it is necessary to apply a unifying substitution to both clauses being resolved as the two will be combined (minus the literals "resolved away") to form a new clause.

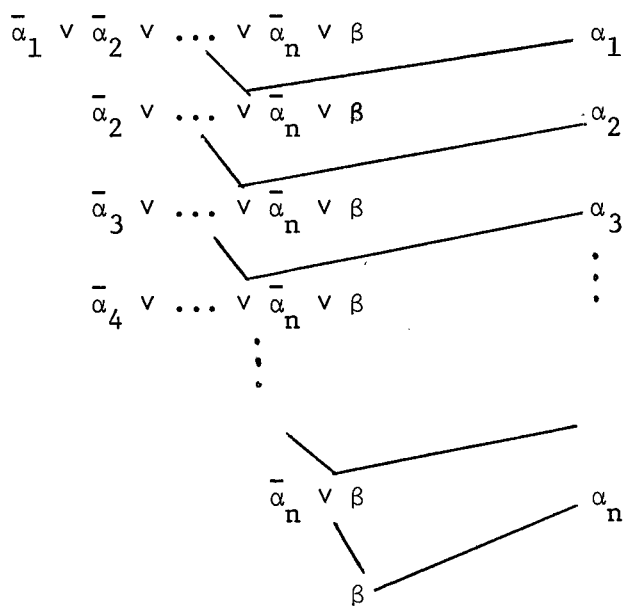
Here unit resolution has a special advantage. Because one of the parent clauses in unit resolution is a unit clause, a given substitution need only be applied to one clause (minus the resolved literal). Since these substitutions are quite expensive this is a considerable advantage.

5. Subsumption

Closely related to the concept of unification is the subsumed clause. A clause M is said to be subsumed by a clause N if there exists a substitution $\theta$ such that $N\theta \subseteq M$. Basically, this means whenever M is true, N is true and hence there is little value in terms of the resolution process in keeping the subsumed clause. A check for subsumed clauses in included in this theorem proving system.

6. The "hyper" Unit Resolution Strategy

Consider a sequence of resolutions of the following type where $\alpha_1, \ldots, \alpha_n$ are unit clauses and $\bar{\alpha}_1 \vee \bar{\alpha}_2 \vee \ldots \vee \bar{\alpha}_n \vee \beta$ is a non-unit clause (axiom):



In short where all the consequent literals may be matched with known unit clauses it is possible that a new unit clause can be generated by in this case n resolutions. A necessary condition for this to occur is that the literals must have differing signs and matching predicate symbols with their corresponding units in $\{\alpha_i\}$. In additional each literal/unit pair must unify at the appropriate stage of the overall resolution sequence. The "hyper" unit resolution strategy is simply to generate only unit resolvents in the above fashion. The n resolutions outlined above are treated as occurring simultaneously for purposes of retaining new resolvents. A resolvent is generated only if it is a unit clause, that is, all literals appearing in a given axiom except the "target" literal already exist as axioms or previously generated unit resolvents. This target literal, i.e., that literal of the axiom which becomes the unit resolvent, may be any of the literals appearing in the axiom.

The use of this apparent restriction on clauses generated by unit resolution greatly reduces the number of clauses generated for further use in search of a contradiction. For instance (not including multiple instances of the same target literal which differ in their terms) for three axioms of length four there are twelve possible unit resolvents while there are forty-eight possible resolvents including non-unit clauses. Appendix 1 contains a full example demonstrating this advantage.

The following theorem demonstrates that the use of "hyper" resolution is in fact not a further restriction on the effectiveness of unit resolution. "A set of clauses can be proved insatisfiable

by unit resolution if and only if it can be proved unsatisfiable by "hyper" unit resolution". Clearly, from the definition of unit resolution, any proof by regular unit resolution can be converted to a proof by "hyper" unit resolution and hence the proof is trivial.

There is significant combinatorial aspect of "hyper" resolution to be described. Consider the generation of unit resolvents from a clause $C = L_1 \vee \ldots \vee L_r$ of r literals. There are r possible literals which may become unit resolvents. These possible "target" literals each may become a unit clause after r-1 other literals are resolved away using the, say k, unit clauses available. In order to insure that all possible resolvents are generated at a particular level, it is necessary to match each sequence of r-1 literals corresponding to a particular target literal with all possible lists of length r-1 of existing units. Since a given unit may be used more than once there are $\binom{k+r-2}{r-1}$ such combinations of units. Each of these combinations may be ordered in (r-1)! ways hence for a clause c of r possible target literals there are $\binom{k+r-2}{r-1}(r-1)!\cdot r$ combinations of clause literals and unit clauses to be checked for possible results. Thus it is clear that while the number of retained units is significantly reduced with a "hyper" resolution approach there is no reduction apparent in the amount of matching which must take place in order to generate a particular unit clause.

## 7. Embedded Equality Axioms.

The treatment of problems of proof generation with axiom sets expressing equality has had considerable attention in a number of papers on mechanical theorem proving [7, 9]. The problem is a familiar one - only more so. Consider the following axiom (reflexive):

$E\ (x1\ x2) \supset E\ (x2\ x1)$ or in clausal form $\bar{E}\ (x1\ x2) \vee E\ (x2\ x1)$. Now any unit clause with E as its predicate symbol will unify with one or the other of the literals in this clause. In consequence an abnormally large number of new unit clauses will be generated, most of which are quite useless to the production of a proof. Since the nature of axioms expressing equality are relatively mechanical in their effect, treating them as any other axiom unnecessarily marrs the efficiency of a theorem proving program. Such axioms may be embedded within the code of the program and thus produce much more efficiently as required. In light of this the axioms of reflexitivity and transitivity of equality, i.e., $E\ (x1\ x2) \supset E\ (x2\ x1)$ and $E\ (x1\ x2) \wedge E\ (x2\ x3) \supset E\ (x1\ x3)$ have been coded directly into this program and may be used by setting a switch variable on input.

## II A Description of the Program and its Use

The "hyper" unit resolution strategy described in Chapter I has been implemented as a theorem proving program written in LISP 1.5 [5,11]. Appendix 4 outlines how this program may be accessed on the U.B.C./MTS system. This chapter comprises a description of the input format required, an outline of the flow of processing performed by the program and finally a description of program output and its interpretation.

### 1. Input to the Program

A representation of the conjunctive normal form of a given proposed theorem and a few controlling parameters are passed to the predicate UPRØØF in order to initiate the generation of a proof. This call has the following general form:

UPRØØF (CLAUSES UNITSET DEPTH FD ES)

CLAUSES is a structured list of the non-unit clauses of the c.n.f. and UNITSET is a list of the unit clauses. DEPTH is an integer indicating the allowable number of times the CLAUSES may be used before quitting the search for a proof. FD, also an integer, indicates the maximum function depth in retained unit clauses. ES is a switch variable; its value is T if embedded equality relations are to be used and NIL otherwise.

The latter three parameters have a self explanatory format as indicated in the example which follows. CLAUSES and UNITSET, however, require further description.

CLAUSES has a certain amount of structure in its format in order to aid the search process of the program. As indicated earlier in Chapter I, a clause is represented as a list of its

literals which in turn are lists of signs, predicate symbols and variables etc. An example is ((+ P x1 x2)(+ Q x1 x2)(+ R x2 x3)). Given this format for a clause, CLAUSES is imply a set of clauses grouped by length. In other words, CLAUSES is a list of lists of clauses, each sublist containing clauses of the same length. The following is a set of four clauses so arranged:

((((+ P x1 x2)(+ Q x2 x3()- R x1 x2)))

(((+ P x2 x3)(- Q x3 x2)(- P x3 x2))

((- R x1 x4)(+ Q x1 x2)(+ P x1 x2)))

(((+ P x1 x2)(- R x2 x1))))

UNITSET is simply a list of unit clauses. As earlier mentioned in Chapter I, the additional set of brackets which would normally denote a clause as opposed to a literal have been dropped hence UNITSET appears as shown in the following example:

((+ P x1 x2)(+ Q x1 x3)(- R x2 x3))

All variables which appear in CLAUSES and UNITSET are x-standardized according to the following condition. The same variable name may appear in two or more non-unit clauses but may not appear in any unit clause. In addition those variable names occurring in a unit clause must be distinct from all other variable names appearing in unit clauses (as well as those appearing in CLAUSES). This is done to avoid ambiguity in unification and substitution. In addition all variable names must include an integer larger than 20 to avoid conflict with internal variables in the equality processing. A complete example of a call to UPRØØF follows:

```
UPRØØF ((((+ P x100)(+ D (G x100) x100))
         ((+ P x100)(+ L 1 (G x100)))
         ((+ P x100)(+ L (G x100) x100))
         ((- P x100)(- D x100 A)))
        (((- L 1 x100)(- L x100 A)(+ P (F x100)))
         ((- L 1 x100)(- L x100 A)(+ D (F x100) x100))
          (- D x100 x200)(- D x200 x300)(+ D x100 x300))))
         ((+ D x400 x400)(+ L 1 A)) 5 3 NIL)
```

This is a formulation of example eight from Chang [2].

## 2. Program Operation

This program generates and tests unit clauses in a breadth first fashion using the general procedure outlined in the following steps.

1. Select a set of non-unit clauses of the same length (SLCSET)
2. Generate a list of unit clauses from the unitset (UNITS)
3. If level of generation is greater than one test above units for presence of at least one unit from previous level.
4. Match UNITS against all clauses in SLCSET and generate any new resolvents possible.
5. Test newly generated resolvents for function depth and subsumption by other units.
6. Test remaining units generated for contradiction with other units already existing, if found return T.
7. Is a new string of units possible if so go to step 2.
8. Is a new set of clauses of the same length available? If yes, go to step 1.
9. Test all units generated at this level for contradiction and subsumption among each other if contradiction return T, if no units remain return 1.
10. Has maximum depth of clause use been reached if so return 0.
11. Update unitset with newly generated units and go to step 1.

In short, this program matches every clause of length $\ell$-1 at a given level. Using clause rotation each literal in a clause is treated as a possible unit resolvent. A simple combinatorial algorithm [4] is used to generate lists of units from the UNITSET of a given level. Initial checks are made on such lists and a given clause (or rotation of a clause) to insure appropriate sign and predicate symbols in each literal pair from left to right. Given the success of the above test, unification and substitution proceed on the literal pairings from the left until the given list of units is exhausted or unification fails. If a unit resolvent is generated it is tested for function depth, subsumption by existing units and for possible contradiction by resolution against one of the existing units. If it does not contradict or fail the other test it is saved for use during the next level of resolution.

There are three major sections or LISP functions in this theorem proving system. They are UPRØØF, ALØNZØI, and SIGMA.

UPRØØF is the overall driver of the system. It controls passage through the clauses, updates the set of units between levels of clause use, and communicates with the user. UPRØØF calls on ALØNZØI to perform the matching of clauses with lists of units and to do most of the screening of generated unit clauses.

ALØNZØI is the most active of the functions of the system. It generates lists of units of the appropriate length for the clause or clauses it has been passed, checks for sign and predicate symbol matching, controls and calls the unification and substitution functions as required, rotates clauses as needed, and performs subsumption, function depth and contradictory unit tests as needed. If a

contradiction is found or all possible units have been generated for a given list of clauses of the same length, ALØNZØI returns to UPRØØF.

SIGMA is the controling LISP function for a collection of routines which perform unification (if possible) for a given pair of lists of terms of a literal pair. SIGMA produces a most general unifier if unification is possible or else returns the LISP atom NIL.

In order to obtain a run which utilizes implicit coding of the equality axioms, i.e.,

$$E(x, y) \wedge E(y, z) \supset E(x, z)$$

$$E(x, y) \supset E(y, x)$$

the value of the input variable ES must be T. In order to have these axioms be effective at the appropriate point in use of the clauses there are a few dynamic switch variables set in UPRØØF and passed to ALØNZØI. The function EQGEN is called by ALØNZØI to generate new units resulting from said axioms, when such generation is signaled by UPRØØF.

Appendices 2 and 3 contain relatively detailed flow charts for UPRØØF and ALØNZØI and Appendix 5, The LISP Code.

3. Program Output

A machine generated proof must be recoverable. This program achieves this condition for its proofs by printing out the history of a unit just after it has been generated. This history's format is as follows:

< parent clause
or a rotation of
that clause >

< the list of units
used to resolve
against the clause
above >

< the unit resolvent >

an example might be;

```
CLAUSE USED
  ((- P x1 x2)(- P x2 x3)(+ P x1 x3))
UNITS USED
  ((+ P A B )(+ P B C))
RESULTANT UNIT CLAUSE
   (+ P A C)
```

Units generated by the use of implicit equality axioms are simply listed as such and normally are simply traced by checking with previous existing units. Upon the discovery of two contradictory unit clauses, UPRØØF returns the value T. The two units which contradict are printed under the heading.

PRØØF FØUND FØR THIS THEØREM

At this point the user may reconstruct the proof tree for the theorem in question by tracing the printed histories of the units in question. Since variables are restandardized relatively often within the proof generation process, the user must generally look back for a different instance (in respect of variable names) than the current unit being traced.

Each time a new set of clauses is passed to ALØNZØI, the level of clause use, clauses passed, and new units from the previous label are printed.

If the program fails to find a proof it will be indicated by a return value of 0 or 1 or an obvious time over-run. A return value

of 0 indicates that a proof was not found at the maximum depth specified by the input parameter DEPTH. A return value of 1 signifies that at the last indicated level of clause use no new units were generated (or more specifically no new units were generated that passed both the subsumption and function depth tests). In the latter circumstance, if a greater function depth does not change matters then no unit proof exists for the proposed theorem.

The output format described above is motivated by two considerations; generation of a trace of a proof is costly and involves a considerable amount of programming and, secondly, to effectively study the possible implementation of heuristics in a theorem prover it is helpful to print as much as possible concerning the directions taken.

A sample run is given of example 1 from Chang in Appendix 6.

## III Evaluation of Experimental Results

### 1. Comparison with the work of Chang [2]

Apart from his use of the set of support technique, the work of Chang is quite similar to that presented here. The following table displays the results of the two programmes when applied to the examples given by Chang.

| EXAMPLE # | UNITS GENERATED | CLAUSE DEPTH | UNITS RETAINED | TIME (sec) |
|---|---|---|---|---|
| 1 | 16 | 1 | 0 | 1.550 |
| | 2 | 1 | 0 | .524 |
| 2 | 109 | 3 | 3 | 7.510 |
| | 72 | 2 | 4 | 28.222 |
| 3 | 44 | 3 | 3 | 3.417 |
| | 14 | 2 | 3 | 3.821 |
| 4 | 32 | 2 | 2 | 2.650 |
| | 14 | 1 | 0 | 2.051 |
| 5 | 21 | 1 | 0 | 0.667 |
| | 1 | 1 | 0 | 0.646 |
| 6 | 32 | 2 | 1 | 2.300 |
| | 74 | 1 | 0 | 15.267 |
| 7 | 58 | 2 | 5 | 3.566 |
| | 6 | 1 | 0 | 5.602 |
| 8 | 65 | ? | 11 | 5.184 |
| | 17 | 4 | 4 | 4.843 |
| 9 | 34 | ? | 10 | 4.050 |
| | 15 | 4 | 9 | 6.640 |
| 10(a) | 136 | 4 | 17 | 24.550 |
| | 154 | 2 | 16 | 31.457 |
| 10(b) | 59 | 3 | 6 | 9.367 |
| | 169 | 2 | 16 | 37.228 |
| 10(c) | 53 | 3 | 4 | 6.916 |
| | 141 | 2 | 16 | 28.151 |
| 10(d) | 48 | 3 | 5 | 7.317 |
| | 167 | 2 | 16 | 37.055 |

TABLE I: A Comparative Table of Chang's Examples

For each entry in the table the value obtained by Chang is followed by that obtained with this system. The column headings have the following, more elaborate explanations:

UNITS GENERATED - the total number of unit clauses produced before a proof was found.

CLAUSE DEPTH - the number of times the set of non-unit clauses was used in order to find a proof.

UNITS RETAINED - the number of generated unit clauses which were retained for use in further levels before a proof was found.

Comparison of the two sets of results reveals the following general differences:

(1) With the exception of examples 6 and 10 this system produces significantly fewer unit clauses than Chang's.

(2) The clause depth necessary for proof for this system is most often lower than that for Chang's.

(3) Relative to the number of clauses generated, the number of clauses retained by both programmes is about the same.

(4) The running times required by Chang's programme are (not withstanding differences in hardware and software) significantly lower than those required by this programme.

2. Comments on the Differences in the Results

The conclusions stated here are by necessity somewhat speculative both for reasons of inadequate information about the internal operation of Chang's system and, perhaps more importantly, lack of an adequate or conclusive technique for measuring the effectiveness of a theorem proving programme [6].

Points (1) and (2) are probable evidence of an inherent disadvantage of the set of support strategy. Simply put, it quite often results in deeper proofs than are necessary. In many circumstances an "inferential" proof exists at a shallower depth than one provided by set of support. The set of support programme must pass through, in general, more levels of clause use (and generate more clauses as a result) in order to find a contradiction. This fault in the set of support approach is probably not particularly significant for theorems which do not admit to a relatively simple inferential proof.

It should be noted at this point that although not explicitly stated, Chang's program is using the same basic strategy as the "hyper" unit resolution developed here. In addition, his routine includes a mechanism to avoid the generation of duplicate clauses.

Point (3) is approximately as expected since the routines have the same clause rejection mechanisms, i.e., subsumption and function depth.

Point (4), judging by points (1)-(3), relates to the amount of effort expended on non-productive matching in the larger search tree a program without set of support techniques must face. Here also the effects of not having a means of avoiding duplicate clause generation are shown as well as a probable difference in programming skill.

In summary, our programme has a definite advantage over Chang's in respect to depth of search and units generated where a proof of a shallow inferential nature exists. It also seems likely that with or without set of support, either program could not prove significantly deeper theorems without additional techniques.

## IV Possible Directions for Further Work

### 1. The nature of inferential unit resolution.

While incomplete as an inference system, unit resolution appears to admit many proofs which are inferential in nature. That is, a consequent of the axioms conflicts with the negation of the theorem. It is perhaps possible that a program could be developed to choose such proof circumstances and, using a form of backtracking from the negation of the proposed theorem, construct possible proof trees for development by regular unit resolution. In short this sort of idea points to the development of a "measure of complexity" for resolution proofs.

### 2. Heuristic Tree Search

Insofar as unit resolution is not complete, i.e. there are possible theorems it cannot theoretically prove, it can therefore be thought of as a heuristic approach. As Chang states "If a theorem, i.e., a set of clauses, does not have an input or unit proof, we may convert it into a theorem which has an input or unit proof by putting into it appropriate lemmas"[2]. This would certainly seem to be the direction to follow if it were clear that unit resolution, as now implemented can find proofs for all such lemmas. Results so far seem to indicate this is not the case and that continued work on the development of stronger unit resolution procedures is very much needed. Most refinements to resolution have centered on the logical power of the technique in order to reduce the overall size of the search tree. Like unit resolution in this system, they have generally been applied to that tree so defined

in a breadth first manner, normally plowing through masses of units in quite rote fashion. Work with this programme has emphasised this blindness of flow in two respects. First, once a clause is retained it is not evaluated for significance in further proof development. Secondly, axioms are selected and used rotely without any type of dynamic choice. When these types of concepts are entertained, much of the theory and philosophy of tree search becomes relevant to theorem proving. It is simply a matter of saying logical completeness or algorithmic process is not worth much if the program is not able to reach interesting levels.

A couple of facts drawn from runs with this theorem prover are worthy of consideration in developing a genuinely heuristic theorem proving system. The first of these is the significance of ground instances of literals or unit clauses. Most proofs include at least one of these in their tree structure and hence appear to be of more than average value when generated. The other fact is the difference in time required for proof using varying orders of clause selection. For this program using breadth first search, the times may vary as much as the total processing time for the deepest level required. Obviously even a slight amount of selectively or dynamic clause ordering would reduce the mean time for such proofs.

## 3. Concluding Remarks

The original intent of this work was twofold. First to provide a form of theorem proving system for further work and secondly to try the "hyper" unit resolution technique described in Chapter I.

It seems clear now that the "further work" which motivated this project will involve some major changes to the theorem prover

itself as its potential for "interesting" results is limited. The "hyper" unit resolution strategy seems to boil down to a method of description of internal workings of the program rather than an advancement as a natural approach. If the problems it entails with respect to the internal ordering and construction of lists can be reduced it seems as good as any other unit resolution approach. Finally insofar as it is relatively uncluttered in concept, unit resolution should perhaps be given more time and effort than it appears to have received. If the problems of resolution theorem proving are to be solved it seems wise to work on the simpler case first.

## Bibliography


1. Carson, D., Robinson, G., and Wos, L., The unit preference strategy in theorem proving, Proceedings, 1964 Fall Joint Computer Conference, 615-621, Spartan Press, 1964.

2. Chang, C.L., The unit proof and the input proof in Theorem Proving. J.ACM 17, 4 (October 1970), 698-707.

3. Davis, M., Eliminating the irrelevant from mechanical proofs, Proc. Symp. App. Math XV, 1963, pp. 15-30.

4. Lehmer, D.H., The machine tools of Combinatorics, in Applied Combinatorial Mathematics, E.F. Beckinbach, ed., Wiley, New York, 1964, pp. 17-18.

5. McCarthy, J., et. al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., 1962.

6. Meltzer, B., Prolegomena to a theory of efficiency of proof procedures, Artificial Intelligence and Heuristic Programming, N.V. Findler, B. Meltzer, eds., American Elsevier, N.Y., 1971.

7. Morris, J.B., E resolution; extension of resolution to include the equality relation., Proceedings Int. Joint Conference on Artificial Intelligence, Wash. D.C., May 7-9, 1969.

8. Nilsson, N.J., Problem Solving Methods in Artificial Intelligence, McGraw-Hill, 1971.

9. Robinson, G., and Wos, L., Paramodulation and theorem proving in first order theories with equality, in Machine Intelligence IV, Meltzer and Michie, eds., Edinburgh University Press, 1969.

10. Robinson, J.A., A machine oriented logic based on the resolution principle, J.ACM, 12, 1 (Jan. 1965), 23-41.

11. Weissman, C., LISP 1.5 Primer, Dickenson Publishing Company, 1967.

Appendix 1

An example of "hyper" unit resolution

We wish to prove a=d from the following axioms:

a=b, b=c, c=d

$x=y \wedge y=z \supset x=z$

In the notation of this theorem proving system we are thus working with the unit clauses (+ E A B), (+ E B C)(+ E C D) and the non unit clause ((- E x1 x2)(- E x2 x3)(+ E x1 x3))

If we apply regular unit resolution the following clauses are generated:

((- E B x3)(+ E A x3))

((- E C x3)(+ E B x3))

((- E D x3)(+ E C x3))

((- E x1 A)(+ E x1 B)) 1st level resolvents

((- E x1 B)(+ E x1 C))

((- E x1 C)(+ E x1 D))

(+ E A C)

(+ E B D) 2nd level resolvents

((- E C x3)(+ E A x3))

((- E B x3)(+ E D x3))

((- E x1 A)(+ E x1 C)) 3rd level resolvents

((- E x1 B)(+ E x1 D))

(+ E A D)

In short, twelve clauses were generated and retained before the clause desired appeared.

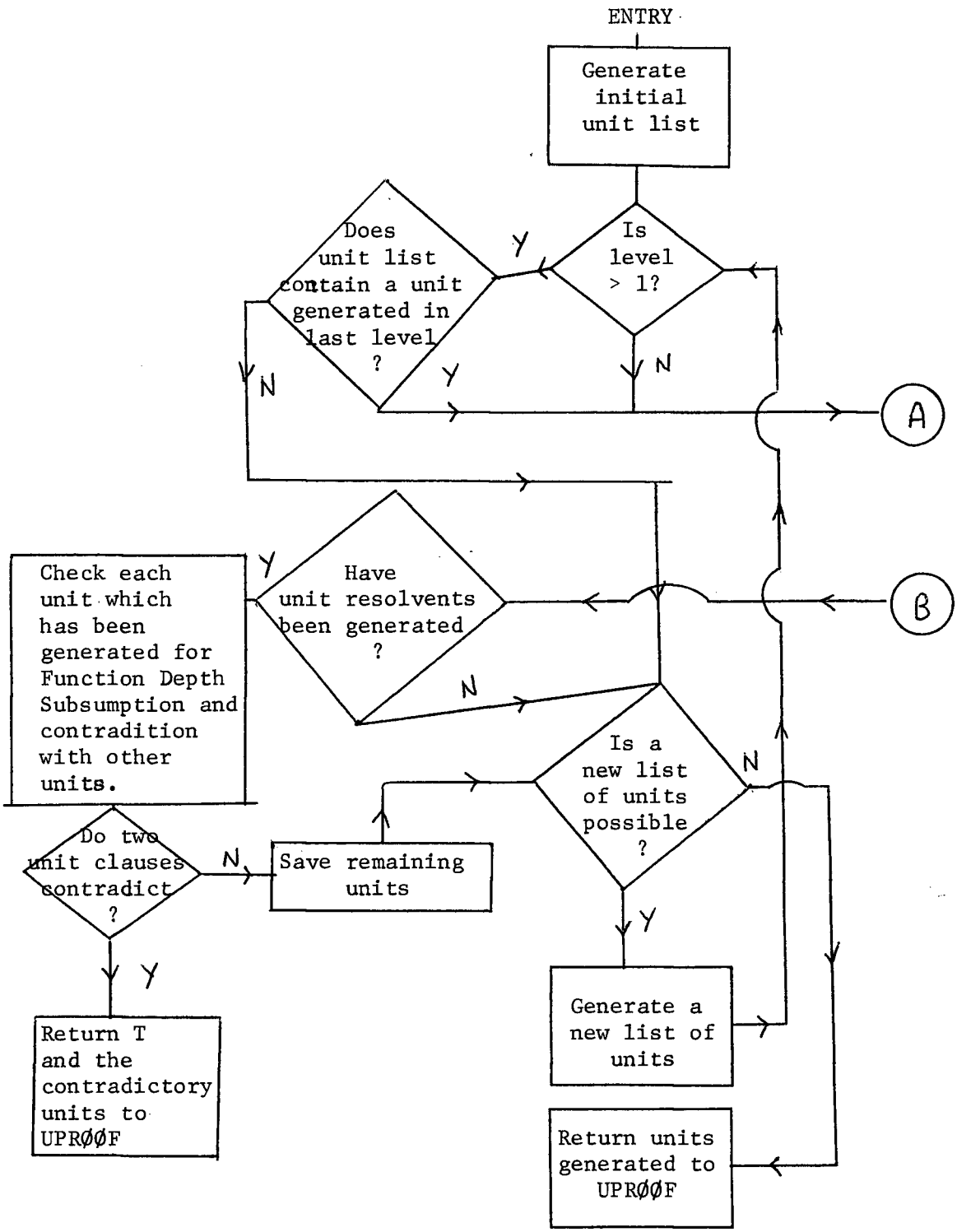
Using "hyper" unit resolution, we take our three initial positive units in pairs and try to match them with the two negative literals of the non-unit clause. The only two pairs which work are ((+ E A B)(+ E B C)) and ((+ E B C)(+ E C D)) which in turn produce the pair of resolvents (+ E A C) and (+ E B D).

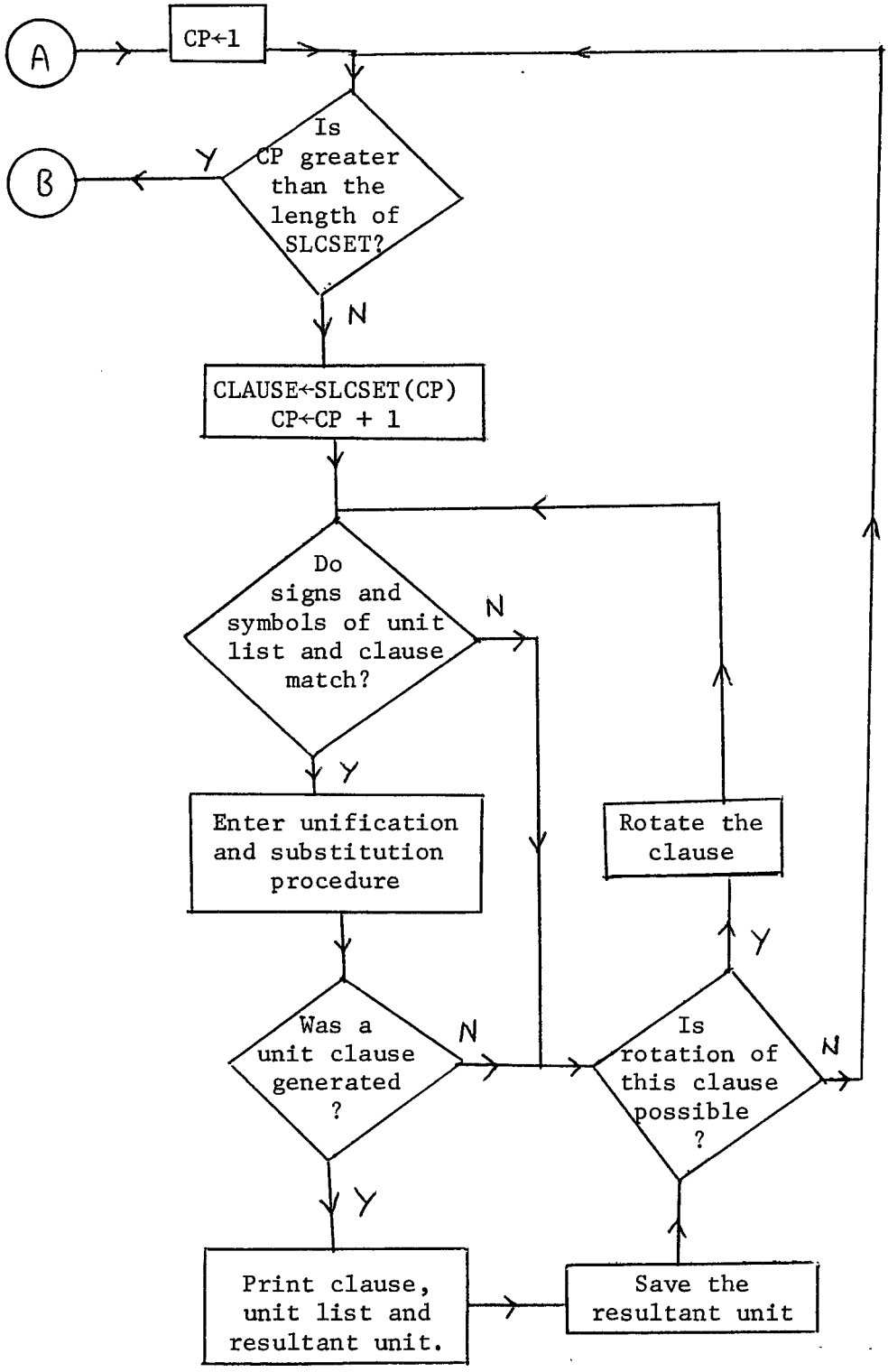
Hence for the next level of clause use we have five unit clauses. From these five units only two pairs ((+ E A B)(+ E B D)) and ((+ E A C)(+ E C D)) will produce a unit resolvent from our non-unit clause. (Repeated matchings from earlier levels are not used.) In both cases the unit result is (+ E A D) which is the desired clause in this proof.

The important difference in these two methods is clearly illustrated; the regular unit proof must retain 12 clauses before the desired result is found whereas the "hyper" unit resolution technique needs to hold only 2 intermediate results.

Appendix 2

Flow of ALØNZØI



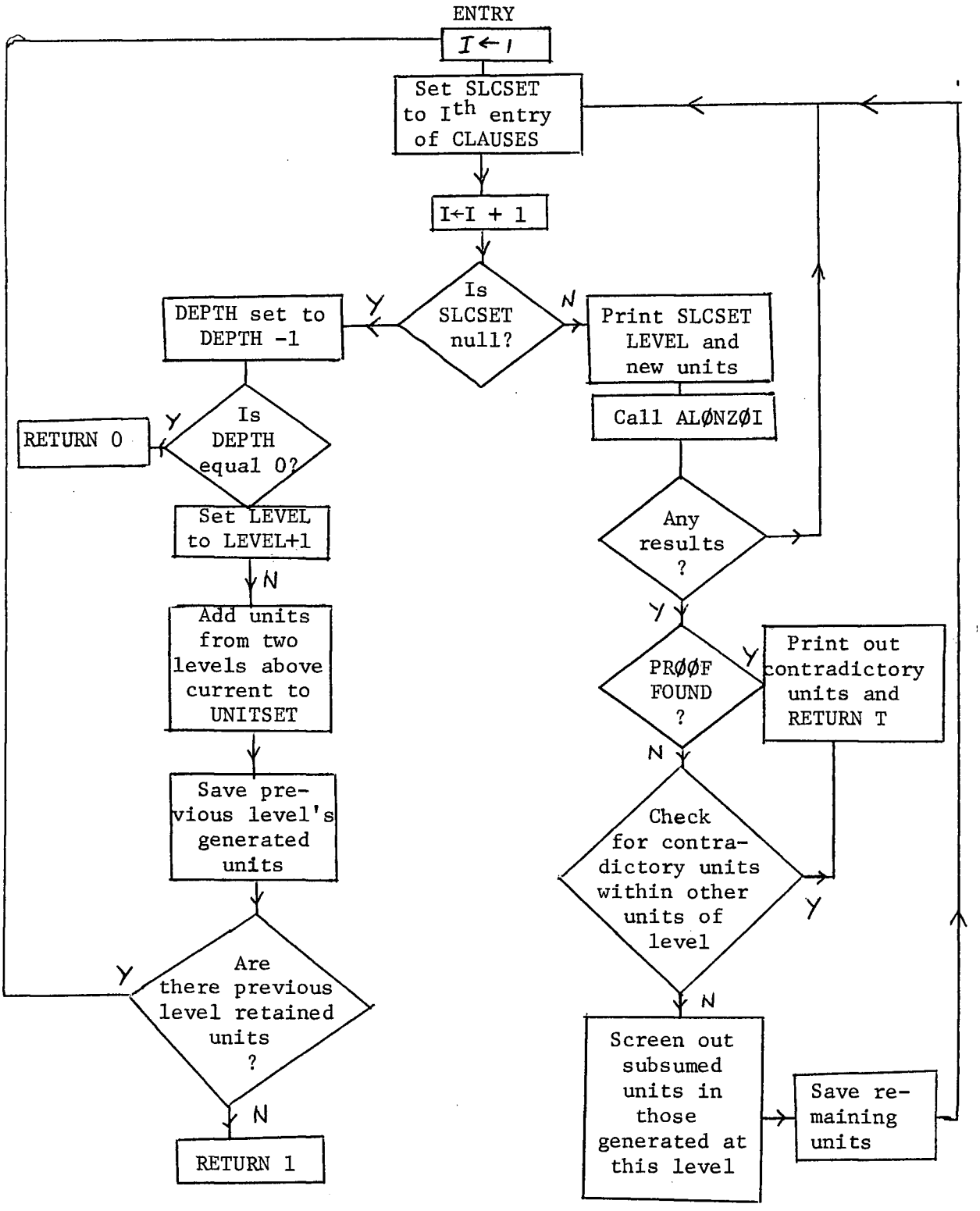



CP - position in set of clauses of the same length.

SLCSET - set of clauses of the same length.

CLAUSE - a particular clause from SLCSET.

Flow of UPRØØF



## Appendix 4

## Use of the Programme on the U.B.C. M.T.S. System

Because of the somewhat awkward method of compilation used by LISP 1.5 the following job set up must be used to run the program.

```
1)  $SIG <ID> T= , etc.
     PW
2)  $R LISP:LISP SCARDS=*SOURCE*
3)  ØPEN(THRM:UTP SYSFILE INPUT)
4)  RESTØRE(THRM:UTP)
5)  CLØSE(THRM:UTP)
6)  UPRØØF (<args as given in Chapter II>)
7)  MTS()
8)  $SIG
```

1) signon card — allow enough time and pages (program is somewhat verbose to say the least).

2) run card — note that the program does not use the regular LISP 1.5 system (for reasons of data cell space required).

3) ØPEN — opens the data cell file containing the compiled theorem prover.

4) RESTØRE — reads the compiled program into the LISP system.

5) CLØSE — closes the source file - must be included.

6) UPRØØF — statement to call the program exactly as outlined in Chapter II. Can be several cards but don't use past column 72.

7) MTS() — this call returns control to MTS. Note that there is a nil argument.

8) signoff

The file THRM:UTP is marked public and takes approximately 3 seconds to restore.

Finally, check input to UPRØØF carefully before attempting a long run. LISP is not the cheapest way to fly.

SØURCE CØDE

1. The functions ALØNZØI and UPRØØF and their supporting functions.

```
>DEFINE(((NEWUCHK (LAMBDA(CTRL NIS)
>(COND((NULL CTRL) NIL)((MEMBER (CAR CTRL) NIS) T)
>(T (NEWUCHK (CDR CTRL) NIS)) ))))))))))))
>DEFINE(((SUBS (LAMBDA (S STRING)
>(PROG()
>S2 (COND((NULL S) (RETURN STRING)))
>(SETQ STRING (ATOMSUBST (CAAR S) (CADAR S" STRING))
>(SETQ S (CDR S))
>(GO S2))))))))))))
>DEFINE(((FETCH (LAMBDA (J LIST)
>(COND((NULL LIST)NIL)((EQUAL J 1 )(CAR LIST))
>(T (FETCH (SUB1 J)(CDR LIST))))))))))))))
>DEFINE(((REPLACE(LAMBDA(J B LIST)
>(PROG(TEMP1)
>A1(COND((EQUAL J 1)(GO END)))
>(SETQ TEMP1 (APPEND1 TEMP1 (CAR LIST)))
>(SETQ LIST (CDR LIST))
>(SETQ J (SUB1 J))  (GO A1)
>END (RETURN (NCONC TEMP1 (RPLACA LIST B))))))))))
>DEFINE (((CONTRA (LAMBDA (UNIT ULIST)
>(PROG (X)
>L1 (COND((NULL ULIST)(RETURN NIL)))
>(SETQ X (CAR ULIST))
>(COND((AND(NULL(EQ(CAR UNIT)(CAR X)))(EQ (CADR UNIT)(CADR X)))
>(COND((SIGMA(CDDR UNIT)(CDDR X))(RETURN X))(T NIL))))
>(SETQ ULIST (CDR ULIST))          (GO L1))))))))))))))))
>DEFINE(((ALONZO1(LAMBDA(SLCSET UNITSET NEWU FD ES EQS FS)
>(PROG (FIRST COUNTER L NLIST J X RESULTS Y COUNT ANS CTEMP UTEMP S
>CLAUSE CP CLAUSE1 ULIST1 UT1 ELE NC V1 V2 CTRL NIS K)
>(COND((NULL EQS)(GO ABB)))
>(SDTQ ANS (NCONC ANS (EQGEN NEWU UNITSET)))
>(COND (FS (SETQ ANS (NCONC ANS (EQGEN UNITSET NIL)))))
>(PRINT (QUOTE $$'UNITS GENERATED BY IMPLICIT EQUALITY RELATIONS'))
>(PRINT ANS)
>ABC (SETQ NC (LENGTH SLCSET))
>(SETQ L (SUB1 (LENGTH (CAR SLCSET))))
>LAB1 (SETQ FIRST (FETCH 1 UNITSET))
>(SETQ COUNTER L)
>L1 (SETQ NLIST (APPEND1 NLIST (STAND FIRSS)))
>(SETQ CTRL (APPEND1 CTRL (QUOTE 1)))
>(SETQ COUNTER (SUB1 COUNTER))
>(COND((GREATERP COUNTER 0)(GO L1)))
>(COND((NULL NEWU) (GO Z2)))
>(SETQ COUNTER (ADD1 (LENGTH UNITSET)))
>(SETQ UNITSET (APPEND UNITSET NEWU))
>Z1 (SETQ NIS (APPEND1 NIS COUNTER))
#
```

```
>(SETQ COUNTER (ADD1 COUNTER))
>(COND((GREATERP COUNTER (LENGTH UNITSET))(GO Z2)) (T (GO Z1)))
>Z2 (SETQ K (LENGTH  UNITSET))
>L2 (COND((NULL NEWU) (GO PAST)))
>(COND((NULL (NEWUCHK CTRL NIS))(GO ON)))
>PAST (SETQ CP 1)
>CF (COND ((GREATERP CP NC)(GO OUT)))
>(SETQ CLAUSE (FETCH CP SLCSET))
>(SETQ CP (ADD1 CP))
>(SETQ COUNT (ADD1 L))
>L00(SETQ ULIST1 NLIST)
>(SETQ CLAUSE1 CLAUSE)
>L100(COND((EQ (CAAR ULIST1)(CAAR CLAUSE1))(GO L44)))
>(COND((NOT(EQ (CADAR ULIST1)(CADAR CLAUSE1)))(GO L44)))
>(SETQ ULIST1 (CDR ULIST1))
>(SETQ CLAUSE1 (CDR CLAUSE1))
>(COND((NULL ULIST1)(GO L22)))
>(GO L100)
>L44 (SETQ COUNT (SUB1 COUNT))
>(SETQ CLAUSE (APPEND (CDR CLAUSE)(LIST (CAR CLAUSE))))
>(COND((EQUAL COUNT 0)(GO CF)))
>(GO L00)
>L22 (SETQ CTEMP CLAUSE)(SETQ UTEMP NLISS)
>L77(SETQ S (SIGMA (CDDAR CTEMP)(CDDAR UTEMP)))
>(COND (S (GO L33))(T (GO L44)))
>L33 (COND ((EQUAL S T)(GO L55)))
>(SETQ CTEMP (SUBS S CTEMP))
>L55(COND((EQUAL (CDR UTEMP) NIL)(GO L66)))
>(SETQ CTEMP (CDR CTEMP))(SETQ UTEMP (CDR UTEMP))
>(GO L77)
>L66 (PRINT (QUOTE $$'CLAUSE USED')) (PRINT CLAUSE)
>(PRINT (QUOTE $$'UNITS USED')) (PRINT NLIST)
>(PRINT(QUOTE $$'RESULTANT UNIT CLAUSE')) (PRINT (CADR CTEMP))
>(SETQ ANS (APPEND1 ANS (STAND (CADR CTEMP))))(GO L44)
>OUT(COND((NULL ANS)(GO ON)))
>(SETQ V2 (APPEND UNITSET RESULTS))
>(SETQ V1 (CAR ANS))
>(COND((FDCHK (CDDR V1) FD)(GO BY)))
>(COND((NULL ES)(GO L500)))
>(COND((NOT (EQ (CADR V1)(QUOTE E)))(GO L500)))
>(COND((EQ (CAR V1)(QUOTE -))(GO E1)))
>(COND((SIGMA (LIST(CADDR V1))(CDDDR V1))(GO BY)))
>(GO L500)
>E1 (COND((SIGMA(LIST(CADDR V1))(CDDDR V1))
>(RETURN (APPEND1 (LIST T (LIST (QUOTE +)(QUOTE E)(QUOTE X)
>(QUOTE X)))V1))))
>L500(COND((NOT(EQ(CADR V1)(CADAR V2)))(GO L501))
>((EQ(CAR V1)(CAAR V2))(GO L502)))
>(COND((SIGMA (CDDR V1)(CDDAR V2))
>(RETURN (APPEND1 (LIST T (CAR V2)) V1)))  )
>(GO L501)
>L502 (SETQ Y (SIGMA (CDDAR V2)(CDDR V1)))
>(COND(Y (GO L601))(T (GO L501)))
>L601 (COND((EQUAL Y T) (GO BZ)))
#
```

```
>L601 (COND((EQUAL Y T) (GO BY)))
>L506 (COND((NOT(ATVAROF (CADAR Y)(CDDAR U2)))(GO L501)))
>(COND((NULL (CDR Y)) (GO BY)))
>(SETQ Y (CDR Y))(GO L506)
>L501 (SETQ V2 (CDR V2))
>(COND(V2 (GO L500)))
>X1(SETQ RESULTS (APPEND1 RESULTS (CAR ANS)))
>BY (SETQ ANS (CDR ANS)) (GO OUT)
>ON (SETQ J 1)
>L4 (SETQ ELE (ADD1 (FETCH J CTRL)))
>(COND((GREATERP ELE K)(GO L3)))
>(SETQ X (STAND (FETCH ELE UNITSET)))
>(SDTQ NLIST (REPLACE J X NLIST))
>(SETQ CTRL (REPLACE J ELE CTRL))
>(COND((EQUAL J 1) (GO L2)))
>(SETQ COUNTER (SUB1 J))
>L5 (SETQ NLIST (REPLACE COUNTER (STAND FIRST) NLIST))
>(SETQ CTRL (REPLACE COUNTER (QUOTE 1) CTRL))
>(SETQ COUNTER (SUB1 COUNTER))
>(COND((GREATERP COUNTER 0)(GO L5)))
>(GO L2)
>L3 (SETQ J (ADD1 J))
>(COND ((LESSP J L)(GO L4))
>((EQUAL J L )(GO L4)))
>(RETURN RESULTS))))))))))))))))
>DEFINE(((UPROOF(LAMBDA(CLAUSES UNITSET DEPTH FD ES)
>(PROG(I SLCSET A NEW NU V A1 LEVEL EQS FS"
>(SETQ LEVEL 1)
>(COND(ES(SETQ EQS T)))
>(SETQ FS T)
>SETCC (SETQ I 1)
>GETC (SETQ SLCSET (FETCH I CLAUSES))
>(SETQ I (ADD1 I))
>(COND(SLCSET (GO ON)))
>(SETQ DEPTH (SUB1 DEPTH))
>(COND((EQUAL DEPTH 0)(RETURN 0)))
>(SETQ LEVEL (ADD1 LEVEL))
>(COND((NULL ES)(GO AA)))
>(SETQ EQS T)
>AA (SETQ UNITSET (NCONC UNITSET NU))
>SL1(SETQ I (LENGTH NEW))
>(COND((LESSP I 2)(GO SL2)))
>SL4(COND((SUBSUMED (CAR NEW)(CDR NEW))(GO SL3)))
>(SETQ NEW (APPEND (CDR NEW)(LIST (CAR NEW))))
>(SETQ I (SUB1 I))
>(COND((EQUAL I 0)(GO SL2)))
>(GO SL4)
>SL3 (SETQ NEW (CDR NEW))(GO SL1)
>SL2 (SETQ NU NEW)
>(COND((NULL NU)(RETURN 1)))
>(SETQ NEW NIL)(GO SETCC)
>ON(PRINT (QUOTE $$'LEVEL OF CLAUSE USE'))
>(PRINT LEVEL)
>(PRINT (QUOTE CLAUSES))
>(PRINT SLCSET)
>(PRINT(QUOTE $$'NEW UNITS FROM PREVIOUS LEVEL'))
>(PRINT NU)
>(SETQ A (ALONZO1 SLCSET UNITSET NU FD ES EQS FS))
>(SETQ EQS NIL)(SETQ FS NIL)
>(COND(A (GO TEST))(T(GO GETC)))
```

```
>(COND(A (GO TEST))(T(GO GETC)))
>TEST(COND((NULL(EQUAL (CAR A) T))(GO B1)))
>(PRINT (QUOTE $$'PROOF FOUND FOR THIS THEORM'))
>(PRINT (CADR A))
>(PRINT (QUOTE $$'CONTRADICTS'))
>(PRINT (CADDR A))(RETURN T)
>B1 (SETQ A1 A)
>B2 (SETQ V (CONTRA (CAR A) NEW))
>(COND (V (GO END1)))
>(SETQ A (CDR A))
>(COND (A (GO B2)))
>(SETQ NEW (NCONC NEW A1))
>(GO GETC)
>END1 (PRINT T) (PRINT V) (PRINT (CAR A))
>(RETURN T)))))))))))))))
>DEFINE(((TA (LAMBDA (ARG)
>(COND((NULL ARG) 0)((ATOM ARG) 0)
>(T (MAX(ADD1 (TA (CAR ARG)))(TA (CDR ARG)))))))))))))))
>DEFL E(((FDCHK(LAMBDA (ARG FD)
>(PROG (A)
>L1(SETQ A (CAR ARG))
>(COND((GREATERP (TA A ) FD)(RETURN T)))
>(SETQ ARG (CDR ARG))
>(COND((NULL ARG)(RETURN NIL))(T (GO L1)))))))))))))))
>DEFINE(((SUBSUMED(LAMBDA(U UL)
>(PROG (SIGN SYMB S VARLIST1 VARLIST2)
>(SETQ SIGN (CAR U))(SETQ SYMB (CADR U))
>(SETQ VARLIST1 (CDDR U))
>CHECK (COND((AND(EQ SIGN (CAAR UL))(EQ SYMB (CADAR UL)))
>(SETQ VARLIST2 (CDDAR UL)))
>(T (GO NEXTUNIT)))
>(SETQ S (SIGMA VARLIST2 VARLIST1))
>(COND((NULL S)(GO NEXTUNIT))
>    ((EQUAL S T)(RETURN (CAR UL))))
>TEST (COND((NOT (ATVAROF (CADAR S) VARLISS2))
>          (GO NEXTUNIT)))
>(SETQ S (CDR S))
>(COND( S (GO TEST))(T (RETURN (CAR UL))))
>NEXTUNIT (SETQ UL (CDR UL))
>(COND(UL (GO CHECK))(T (RETURN NIL)))))))))))))))
#
```

2. The Unification Routines

```
>DEFINE (((VARSET (LAMBDA (C)
>(COND
>((ATOM C)
>(COND
>((EQ (CAR (EXPLODE C)) (QUOTE X)) (LIST C))
>(T NIL)))
>(T (ATUNION (VARSET (CAR C)) (VARSET (CDR C)))))))))
>DEFINE(((ATUNION (LAMBDA (A B)
>(COND
>((NULL A) B)
>((ATMEMBDR (CAR A) B) (ATUNION (CDR A) B))
>(T (CONS (CAR A) (UNION (CDR A) B)))))))
>DEFINE(((ATMEMBDR (LAMBDA (A B)
>(COND
>((NULL B) NIL)
>((EQ A (CAR B)) T)
>(T (ATMEMBER A (CDR B")))))))
>DEFINE(((STAND (LAMBDA (C)
>(PROG (A)
>(SETQ A (VARSET C))
>TAG (COND
>((NULL A) (RETURN C)))
>(SETQ C (ATOMSUBST (GENSYM1 (QUOTE X)) (CAR A) C))
>(SETQ A (CDR A))
>(GO TAG))))))
>DEFINE(((SIGMA (LAMBDA (C D)
>(PROG (L A B)
>(SETQ L NIL)
>L1 (SETQ A (CAR C))
>(SETQ B (CAR D))
>(COND
>((ATOM A) (GO L4))
>((ATOM B) (GO L5))
>((NOT (EQ (CAR A) (CAR B))) (RETURN NIL)))
>(SETQ C (NCONC (CDR A) (CDR C)))
>(SETQ D (NCONC (CDR B) (CDR D)))
>(GO L1)
>L4 (COND
>((EQ (CAR (EXPLODE A)) (QUOTE X)) (GO L6))
>((NOT (ATOM B)) (RETURN NIL))
>((EQ (CAR (EXPLODE B)) (QUOTE X)) (GO L3))
>((NOT (EQ A B)) (RETURN NIL)))
>L7 (SETQ C (CDR C))
>(SETQ D (CDR D))
>(GO L2)
>L5 (COND
>((NOT (EQ (CAR (EXPLODE B)) (QUOTE X))) (RETURN NIL))
>((ATVAROF B A) (RETURN NIL)))
>(GO L3)
>L6 (COND ((EQ A B" (GO L7))
#
```

```
>((ATVAROF A B" (RETURN NIL)))
>(SETQ L (APPEND1 L (LIST B A)))
>(SETQ C (ATOMSUBST B A (CDR C)))
>(SETQ D (ATOMSUBST B A (CDR D)))
>L2 (COND
>((NOT (NULL C)) (GO L1))
>((NULL L) (RETURN T)))
>(RETURN L)
>L3 (SETQ L (APPEND1 L (LIST A B)))
>(SDTQ C (ATOMSUBST A B (CDR C)))
>(SETQ D (ATOMSUBSS A B (CDR D)))
>(GO L2))))))
>DEFINE(((UNION (LAMBDA (X Y)
>(COND ((NULL X) Y) ((MEMBER (CAR X) Y) (UNION (CDR X) Y))
>(T (CONS (CAR X) (UNION (CDR X) Y))))))))
>DEFINE(((EFFACE (LAMBDA (A X)
>(COND
>((NULL X) NIL)
>((EQUAL A (CAR X)) (CDR X))
>(T (CONS (CAR X) (EFFACE A (CDR X)))))))))
>DEFINE(((SUBST (LAMBDA (X Y Z)
>(COND
>((EQUAL Y Z) X)
>((ATOM Z) Z)
>(T (CONS (SUBST X Y (CAR Z))(SUBST X Y (CDR Z)))))))))
>DEFINE (((STRINGSUB (LAMBDA (ST V Z)
>(COND
>((ATOM Z) Z)
>((EQ V (CAR Z)) (APPEND ST (STRINGSUB ST V (CDR Z))))
>(T (CONS (STRINGSUB ST V (CAR Z))(STRINGSUB ST V (CDR Z)))))))))
>DEFINE(((ATOMSUBST (LAMBDA (X A Z)
>(COND
>((EQ A Z) X)
>((ATOM Z) Z)
>(T (CONS (ATOMSUBST X A (CAR Z))
>(ATOMSUBST X A (CDR Z)))))))))
>DEFINE(((ATVAROF (LAMBDA (A Y)
>(COND
>((ATOM Y) (COND
>((EQ A Y) T)
>(T NIL)))
>(T (OR (ATVAROF A (CAR Y)) (ATVAROF A (CDR Y)))))))))
#
```

3. The function EQGEN and supporting routines.

```
>DEFINE(((EQGEN(LAMBDA(NU OU)
>(PROG (NEW OLD P1 P2 P3 NEU OEU RL FLAG1)
>(SETQ NEU (STRAIN NU))(SETQ OEU (SSRAIN OU))
>(COND((NULL NEU)(RETURN NIL))
>     ((NULL OEU)(GO NONLY)))
>(SETQ NEW NEU)(SETQ OLD OEU)
>SU (SETQ P1 (CAR NEW))(SETQ P2 (CAR OLD))
>FSP (COND((AND(EQ (CAR P1) PLUSS)(EQ (CAR P2) PLUSS))
>          (GO PP))
>         ((AND(EQ(CAR P1) DASH)(EQ (CAR P2) PLUSS))
>          (GO NP))
>         ((AND(EQ(CAR P1) PLUSS)(EQ (CAR P2) DASH))
>          (GO PN))
>         (T (GO CU)))
>PP (SETQ P3 (TRANS 1 P1 P2))  (GO SA)
>NP (SETQ P3 (TRANS 2 P1 P2))  (GO SA)
>PN (SETQ P3 (TRANS 3 P1 P2))  (GO SA)
>SA (COND(P3(SETQ RL (APPEND1 RL (STAND P3)))))
>(COND(FLAG1(GO CU)))
>(SETQ FLAG1 T) (SETQ P3 P2)(SETQ P2 P1)(SETQ P1 P3)(GO FSP)
>CU (SETQ FLAG1 NIL)(SETQ OLD (CDR OLD))
>  (COND((NULL OLD)(GO CNU)) (T (GO SU)))
>CNU (SETQ NEW (CDR NEW))
>   (COND((NULL NEW)(GO NONLY)))
>   (SETQ OLD OEU)(GO SU)
>NONLY (SETQ RL (NCONC RL (REFLEX NEU)))
>     (SETQ RL (NCONC RL (TRANSN NEU)))
>(RETURN RL))))))))))))))))
>DEFINE(((TRANS(LAMBDA (N P1 P2)
>(PROG (T1 T2 T3 T4 S R)
>(SETQ T1 (CADDR P1))(SETQ T2 (CAR (CDDDR P1)))
>(SETQ T3 (CADDR P2))(SETQ T4 (CAR (CDDDR P2)))
>(COND((EQUAL N 1)(GO PP))
>     ((EQUAL N 2)(GO NP))
>       (T (GO PN)))
>PP(SETQ S (SIGMA (LIST T2 (QUOTE X3))
>                 (LIST T3 T4)))
>(COND((NULL S)(RETURN NIL))
>     ((EQUAL S T)(RETURN
>              (LIST PLUSS (QUOTE E) T1 T4)))
>     (T (RETURN (APPEND (LIST (QUOTE +)(QUOTE E))
>                        (SUBS S (LIST T1 (QUOTE X3)))))))
>PN (SETQ S (SIGMA (LIST (QUOTE X1) T2)(LIST T3 T4)))
>(COND((NULL S)(RETURN NIL))
>     ((EQUAL S T)(RETURN (LIST PLUSS (QUOTE E) T3 T1)))
>     (T (RETURN (APPEND (LISS DASH (QUOTE E))
>                        (SUBS S (LIST (QUOTE X1) T1))))))
>NP (SETQ S (SIGMA (LIST T1 (QUOTE X2))
>                 (LIST T3 T4)))
>(COND((NULL S)(RETURN NIL))
#
```

```
>     ((EQUAL S T)(RETURN (LIST DASH (QUOTE E) T4 T2)))
>     (T (RETURN (APPEND (LIST DASH (QUOTE E))
>                        (SUBS S (LIST (QUOTE X2) T2))))))))))))))))))))
>DEFINE(((REFLEX (LAMBDA (NU)
>(PROG (L I A U)
>(SETQ L (LENGTH NU))(SETQ I 1)
>L1 (SETQ U (CAR NU))
>(SETQ A (APPEND1 A (STAND (LIST (CAR U)(QUOTE E)
>(CAR (CDDDR U))(CADDR U)))))
>(SETQ I (ADD1 I))
>(COND((GREATERP I L)(RETURN A)))
>(SETQ NU (CDR NU))(GO L1)))))))))))
>DEFINE(((TRANSN (LAMBDA (NU)
>(PROG (A PCDR P1 P2 P3 I R FLAG)
>L1 (SETQ P1 (CAR NU))(SETQ PCDR (CDR NU))
>(COND((NULL PCDR)(RETURN A)))
>L2 (SETQ P2 (CAR PCDR))
>L0 (COND((AND (EQ (CAR P1) PLUSS)(EQ (CAR P2) PLUSS))
>(SETQ I 1))
>     ((AND(EQ (CAR P1) DASH)(EQ (CAR P2) PLUSS))
>(SETQ I 2))
>      ((AND(EQ (CAR P1) PLUSS"(EQ (CAR P2) DASH ))
>(SETQ I 3))
>(T (GO L3)))
>(SETQ R (TRANS I P1 P2))
>(COND(R (SETQ A (APPEND1 A (STAND R)))))
>L3 (COND(FLAG (GO L4)))
>(SETQ P3 P2)(SETQ P2 P1)(SETQ P1 P3)(SETQ FLAG T)(GO L0)
>L4(SETQ FLAG NIL)
>(SETQ P1 P2)
>(SETQ PCDR (CDR PCDR))
>(COND(PCDR (GO L2)))
>(SETQ NU (CDR NU))  (GO L1))))))))))))))))
>DEFINE(((STRAIN (LAMBDA (NU)
>(PROG (UL X)
>A (COND((NULL NU)(RETURN UL)))
>(SETQ X (CAR NU))
>(COND((EQ (CADR X)(QUOTE E))(SETQ UL (APPEND1 UL X))  ))
>(SETQ NU (CDR NU))
>(GO A))))))))))))))))
#
```

Appendix 6

A run of example 1 from Chang

```
$R LISP:LISP SCARDS=*SOURCE*
EXECUTION BEGINS

  ARGUMENTS FOR EVALQUOTE ...
    OPEN
    (UTP SYSFILE INPUT)

  TIME       10MS,  VALUE IS ...
    UTP

  ARGUMENTS FOR EVALQUOTE ...
    RESTORE
    (UTP)
  COLLECTED      18140    CELLS AND STACK HAS          5994    UNITS LEFT.

  TIME     3259MS,  VALUE IS ...
    NIL

  ARGUMENTS FOR EVALQUOTE ...
    UPROOF
    (((((- P X1 X2 X4) (- P X2 X3 X5) (- P X1 X5 X6)(+ P X4 X3 X6)))) ((+ P (G X7 X8) X7 X8)
    (+ P X9 (H X9 X10) X10) (-P (K X11) X11 (K X11))) 1 2 NIL)
    LEVEL OF CLAUSE USE
    1
    CLAUSES
    (((- P X1 X2 X4) (- P X2 X3 X5) (- P X1 X5 X6) (+ P X4 X3 X6)))
    NEW UNITS FROM PREVIOUS LEVEL
    NIL
    CLAUSE USED
    ((+ P X4 X3 X6) (- P X1 X2 X4) (- P X2 X3 X5) (- P X1 X5 X6))
```

```
  UNITS USED
  ((- P (K X0000009) X0000009 (K X0000009)) (+ P (G X0000003 X0000004) X0000003 X0000004)
  (+ P (G X0000005 X0000006) X0000005 X0000006))
  RESULTANT UNIT CLAUSE
  (- P (G (G X0000005 X0000006) (K X0000005)) X0000006 (K X0000005))
  CLAUSE USED
  ((- P X1 X2 X4) (- P X2 X3 X5) (- P X1 X5 X6) (+ P X4 X3 X6))
  UNITS
  ((+ P (G X0000014 X0000015) X0000014 X0000015) (+ P X0000012 (H X0000012 X0000013) X0000013)
  (+ P (G X0000005 X0000006) X0000005 X0000006))
  RESULTANT UNIT CLAUSE
  ( + P X0000006 (H X0000005 X0000005) X0000006)
  PROOF FOUND FOR THIS THEOREM
  (- P (K X11) X11 (K X11))
  CONTRADICTS
  (+ P X0000017 (H X0000016 X0000016) X0000017)

TIME     524MS,  VALUE IS ...
  T
```