01 May 1985

An Algorithm for Parallel Subsumption

Ralph M. Butler

Arlan R. Dekock
Missouri University of Science and Technology, adekock@mst.edu

AN ALGORITHM FOR PARALLEL SUBSUMPTION

Ralph M. Butler* and Arlan R. DeKock

CSc-85-1

Department of Computer Science
University of Missouri-Rolla
Rolla, Missouri 65401 (314) 341-4491

*This report is substantially the Ph.D. dissertation of the first author, completed May, 1985.

ABSTRACT

Many current automated theorem provers use a refutation procedure based on some version of the principle of resolution. These methods normally lead to the generation of large numbers of new clauses. Subsumption is a process that eliminates the superfluous clauses from the clause space, thus speeding up the proof. The research presented in this thesis is concerned with the design and implementation of a subsumption algorithm which exploits the parallelism provided by a multiprocessor. For portability, all coding is done in the programming language C. Monitors are used as the synchronization mechanism. Correct performance in both a multiprocessor and uniprocessor mode is stressed. The parallel tests are run on a Denelcor HEP located at Argonne National Laboratory.

ACKNOWLEDGEMENT

The author wishes to thank Dr. Arlan DeKock for his guidance and support as major thesis advisor throughout the writing of this thesis.

A special thanks is extended to Dr. Ross Overbeek for his continued encouragement and his extraordinary assistance on the work performed at Argonne National Laboratory.

The efforts of Dr. John Metzner, Dr. Thomas Sager, Dr. Paul Stigall, and Dr. Henry Metzner, as members of the thesis committee, are also greatly appreciated.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

I. INTRODUCTION

A. THESIS ORGANIZATION

This thesis deals with the design and implementation of a parallel subsumption algorithm. Both the algoritm itself, and its implementation on a multiprocessor, are described.

The first section of this thesis provides the necessary background and vocabulary to understand the subsequent discussions of subsumption and multiprocessing. It also presents a review of literature pertinent to the topic.

The second section presents a discussion of both the high- and low-level design of the programming procedures used. It also provides a description of the methods used in testing the completed program.

The third section describes the experimental results of several test executions of the program.

Finally, the fourth section presents an evaluation of the obtained results and suggests several ideas for continued research.

B. BACKGROUND AND VOCABULARY

1. General. Computers are used today not only to solve difficult numeric problems, but also to perform tasks that would be considered intelligent if performed by humans. Examples of such tasks include expert assistance

to professionals (expert systems), and automated theorem proving. Artificial intelligence is that portion of computer science which deals with the performance of such tasks.

Automated theorem proving is the general area of interest here. J.A. Robinson [1] suggested that:

> "'computational logic' is surely a better phrase than 'theorem proving', for the branch of artificial intelligence which deals with how to make machines do deduction efficiently".

L. Wos [2] suggested the term 'automated reasoning'. Speaking of automated theorem proving and automated reasoning, he said:

> "The difference between the two fields rests mainly with the way in which the corresponding software is used and with their scope. In automated reasoning, the emphasis is on an active collaboration between the user and the program and on many uses you would not ordinarily consider to involve 'proving theorems'. Automated theorem proving is now a part of automated reasoning."

The idea seems to be that the phrase 'theorem proving' carries too much of a mathematical connotation, that a phrase should be chosen which conveys the notion that logical reasoning extends beyond mathematics. The term theorem proving is firmly entrenched in the literature however, and will be used in this thesis with the understanding that problems other than mathematical ones may be posed in the form of a theorem to be proved.

As an example, if it is known that every man is mortal, and that Socrates is a man, then the logical conclusion that Socrates is a mortal may be cast as a theorem to be proved from the stated facts. Many problems (in unrelated fields) can be similarly formulated as theorem proving problems. The following partial list is from the excellent text by Chang and Lee [3]:

(1) question-answering systems - facts are represented as logical formulas. To answer a question from the facts, prove that a formula corresponding to the answer is derivable from the formulas representing the facts.

(2) state-transformation problem - describe the states and transition values by logical formulas. Then, transform the initial state into the desired state by proving that the formula of the desired state follows from the formula representing the state and transition rules.

(3) program-analysis problem - describe program execution by formula A and condition for termination by formula B. Then, verification that the program will terminate is equivalent to proving that B follows from A.

Within the area of automated theorem proving lies the topic of subsumption, which is the specific area of focus in this thesis. Before proceeding with an in-depth discussion of subsumption however, the more general area of theorem proving should be discussed in order to build the necessary vocabulary. Note that this discussion introduces only those terms necessary for an understanding of subsumption, and ignores many additional terms which are important to other aspects of automated theorem proving.

For the reader interested in a history of the entire area of automated theorem proving, the two-volume set

Automation of Reasoning [4, 5] contains a good history of the years 1957-1966 and 1967-1970, respectively, including reprints of the landmark papers published during those years. The recent American Mathematical Society publication Automated Theorem Proving: After 25 Years [6] briefly covers those years also, but refers to Siekmann for a detailed coverage. The AMS publication concentrates on the years since 1970. Some of the landmark papers are reprinted there also.

2. First-Order Predicate Calculus. This section reviews the basic notions of first-order logic, and establishes the terminology. For a detailed development of the subject, the reader is referred to any of several good introductory texts, e.g. [3, 7].

In a first-order logic, one is concerned with entities, relationships between entities, and properties of sets of entities. For example, it is possible to describe the entities 'Joe' and 'Ann' and to address the fact that 'Joe' is married to 'Ann'. Thus, there are functions (wife-of) on the set of entities, as well as predicates (is-married-to) describing properties and relations of entity sets. Functions define new entities in terms of previously known ones, and predicates indicate whether some set of entities has a particular property or relationship. Using the above example, possible statments are:

is-married-to(Joe, Ann) or
is-married-to(Joe, wife-of(Joe)).

In subsequent examples, the alphabet consists of:

- constants: a, b, c
- variables: x, y, z
- functions: f, g, h
- predicates: P, Q, R
- connectives: - (not), ¦ (or), & (and),
  A (for all), E (there exists)
- punctuation: (,), and comma.

Definition: A term is defined recursively as:

(1) variables or constants are terms

(2) if f is any n-place function, and t1,...,tn are terms, then f(t1,...,tn) is a term.

Definition: A formula is defined recursively as:

(1) if t1,...,tn are terms, and P is an n-place predicate (n may be zero, for propositions), then P(t1,...,tn) is an atomic formula

(2) if A and B are formulas, the so are: (-A), (A & B), and (A ¦ B)

(3) if A is a formula, then so are: Ax(A) and Ex(A).

In part 3 of the above definition, the variable is said to be universally or existentially quantified, and the formula is said to be in the scope of the quantifier. An occurrence of a variable in a formula is bound by the innermost quantifier on that variable. Typically, an automated theorem prover uses formulas with no quantifiers present. The justification for this follows.

A first-order logic formula can be transformed into a prenex normal form consisting of two portions - the left portion containing all quantifiers is called the prefix, and the right portion containing the rest of the formula is called the matrix. The existential quantifiers in the prefix can be eliminated by replacing the variables which they quantify with Skolem functions. The matrix can be transformed into conjunctive normal form. Finally, the formula can be converted to a clausal form with no quantifiers present. This form is not strictly equivalent to the original formula, but if the set of clauses is unsatisfiable, then so is the original formula.

The techniques for conversion to a clausal form can be found in any of several good texts, e.g. [3]. A simple example follows.

Example 1: Obtain a clausal form of the formula:

((Ex)P(x) ¦ ((Ay)Q(y) & -(Ax)R(x)))

Step 1. Rename the second x variable (argument to the R predicate) since it is actually different from the first occurrence (argument to the P predicate). Such a step is necessary to ensure that no variable has both free and bound occurrences, and so that there is at most one occurence of a quantifier with any particular variable.

(Ex)P(x) ¦ ((Ay)Q(y) & -(Az)R(z)))

Step 2. The -(Az)R(z) can be transformed into (Ez)-R(z) by the equivalence -(Ax)F = (Ex)-F for any formula F and variable x. Then, convert to prenex normal form where the matrix contains no quantifiers and the prefix is a sequence of quantifiers.

(Ex)(Ay)(Ez)(P(x) ¦ (Q(y) & -R(z)))

Step 3. Remove the existential quantifiers and replace the existentially quantified variables by n-place Skolem functions where n is the number of universal quantifiers preceding the existential one. For example, the x is replaced by the constant a (0-place function). Since there exists at least one such x, a particular one can be chosen, call it a in this case. The z is replaced by a new function, f(y), which is a function of the only universally quantified variable preceding it.

(Ay)(P(a) ¦ (Q(y) & -R(f(y))))

Step 4. Convert to conjunctive normal form and, since all remaining variables are universally quantifed, remove the quantifiers.

(P(a) ¦ Q(y)) & (P(a) ¦ -R(f(y)))

Step 5. The last form can be viewed as a conjunction of clauses. Normally, each individual clause of a clause set is simply listed separately, and they are understood to be conjoined. Here, they may be written as:

P(a) ¦ Q(y)
P(a) ¦ -R(f(y))

leaving off the outermost set of parentheses.

Now the predicate calculus can be extended with notions pertinent to automated theorem proving.

Definition: A literal is an atomic formula (possibly containing variables), or the negation of an atomic formula.

Definition: A clause is a (possibly empty) disjunction (|) of literals. Since the empty clause has no literal that can be satisfied, it is always false.

Definition: A clause set is a conjunction (&) of clauses.

3. Theorem Proof Procedures. Refutation procedures are generally used by automated theorem provers to reach a proof. Typically, the set of clauses used in the proof contains a set of axioms plus the negation of the theorem to be proved. The theorem prover then attempts to show that the set of clauses is inconsistent (unsatisfiable).

A classic example from group theory [2] states that "in a group, if the square of every element is the identity, the group is commutative". The axioms for this proof are:

```
P(x,y,f(x,y))  closure
P(x,e,x)       right identity (e is identity element)
P(e,x,x)       left  identity
P(x,g(x),e)    right inverse
P(g(x),x,e)    left  inverse
P(x,x,e)       square of every element is the identity
-P(x,y,u) | -P(y,z,v) | -P(u,z,w) | P(x,v,w) assoc.
-P(x,y,u) | -P(y,z,v) | -P(x,v,w) | P(u,z,w) assoc.
P(a,b,c)    denial (negation) of
-P(b,a,c)                  the theorem
```

where the 3-place predicate P(x,y,z) may be thought of as asserting that x*y=z.

Any given clause set, such as this one, may be shown to be unsatisfiable if and only if it is false under all interpretations over all domains. It is, of course impossible to examine all possibilities. Herbrand [7] developed a theorem (see the following paragraph) which permits the examination of a single, special domain called the Herbrand Universe. Simply stated, the Herbrand Universe is the set of variable-free terms that can be generated using the constant symbols from the clause set, with some special constant symbol provided if the clause set contains no constants. As an example, if the clause set consists of:

P(a)
P(f(x))

then the Herbrand universe consists of:

{a, f(a), f(f(a)), ... }.

Herbrand's Theorem states:

A set S of clauses is unsatisfiable iff there is a finite unsatisfiable set S' of ground instances of clauses of S.

A ground instance is merely one which has no variables. As an example, consider S = {P(x), -P(f(a))} which is unsatisfiable. Then some S' exists which is an unsatisfiable set of ground instances of clauses in S. One such S' is {P(f(a)), -P(f(a))}.

Several refutation procedures have been developed based on Herbrand's Theorem. The problem however is that the clause space grows exponentially as one substitutes terms from the Herbrand Universe into the elements of the clause set (generating ground clauses).

The resolution principle is a refutation procedure that avoids the need to generate these ground clauses. It checks to see if the empty clause is in the current clause set. If so, then the set is unsatisfiable. If not, it checks to see if the empty clause can be derived from the clauses in the set. Some version of the resolution principle is at the heart of many 'successful' modern-day theorem provers. For that reason, the following section will be devoted to a discussion of the resolution principle and its associated vocabulary.

4. The Resolution Principle. Application of the resolution principle depends upon the ability to locate a literal in one clause that is the complement of a literal in another clause. This task is not too difficult for clauses that contain no variables, but it can become quite difficult for clauses that do contain variables. For example, consider the two literals P(x) and -P(f(x)). By performing the substitution x -> f(a) in the first literal and x -> a in the second literal, the complementary pair of clauses P(f(a)) and -P(f(a)) is obtained, each of which is a ground instance (contains no variables) of the original.

By performing such a substitution, the two literals have been <u>unified</u>. Often, it is desirable to perform the most general <u>unification</u> possible. Above, this would mean that the substitution in P(x) would be x -> f(x), making the two clauses P(f(x)) and -P(f(x)). Now, consider the more complex example of a clause containing several literals. The substitution that unifies a pair of literals must be 'remembered' when examining subsequent literals in the clauses. For example, if the first clause above had been P(x) | Q(g(x)), then after the substitution for x was applied, the 'unified' clause would be P(f(x)) | Q(g(f(x))). Note the 'remembered' substitution is also performed for x in the literal Q(g(x)).

Unification plays an integral part in resolution-based theorem proving. It comes into play not only in the production of resolvents, but also in the performance of subsumption where two clauses must be examined to determine if the literals of one can be mapped into the literals of the other.

Substitution plays a vital role also. When testing for unification, the substitution discovered at one pair of literals must be applied to subsequent literals in the clauses (at least temporarily). If there is a consistent substitution for a pair of clauses, then a <u>resolvent</u> may be produced from them. A resolvent is produced from a pair of clauses by ignoring a complementary pair of literals in them, and copying the remaining literals into the resolvent clause. For example, given the pair of literals:

```
 P(x) | Q(x)
-Q(a) | R(b)
```

the resolvent P(a) | R(b) may be produced under the substitution {x -> a}.

Any new resolvents may be added to the clause space. If the empty clause is one of the resolvents produced, then the desired refutation has been found. New clauses produced (i.e. the resolvents) may be subsumed by old clauses, or the new clauses may subsume old clauses. In either case, a subsumed clause may be deleted from the clause space. This reduces the number of clauses that must be examined in subsequent 'passes' of the resolution procedure.

5. Subsumption.

Definition: A clause C1 subsumes a clause C2 if the variables of C1 can be instantiated in such a way that all the resulting literals of C1 appear in C2.

As an example, the clause P(x) subsumes the clause P(a) because the clause P(x) makes a more general statement (x is a variable and a is a constant). The clause P(a,b) subsumes P(a,b) | P(c,b). If the predicate P means 'is the father of', then the knowledge that a is the father of b is more useful than the mere knowledge that either a or c is the father of b.

Note that the above definition of subsumption permits a longer clause to subsume a shorter one. For example, under the substitution of {x -> a, y -> b}, the clause

P(x,y) | P(y,x)

subsumes

P(a,a).

Sometimes this particular type of subsumption is not desirable because it permits generated factors to be subsumed by their parent. However, it is a simple matter to prohibit this form of subsumption merely by counting the literals in each candidate clause. Only full subsumption is considered here, with the understanding that the above restriction may be desired in certain applications.

Automated theorem provers usually consider subsumption to be of two forms: forward and backward. In both cases, the test is to see if one clause subsumes another based on the definition. The difference lies in which clause is the subsumer and which is the subsumed. Forward subsumption checks to see if any old clauses subsumes a newly generated one. Backward subsumption checks to see if a new clause subsumes any old ones.

Subsumed clauses are deleted from the clause space to reduce the work required in subsequent steps, i.e. the fewer clauses there are to be examined, the fewer resolvents that are likely to be generated. This ability of subsumption to reduce the work required is of course the reason that it has become an integral part of many modern theorem provers. The Literature Review section discusses the use of subsumption in some of these theorem provers.

It is easy to give a reasonably straightforward subsumption algorithm based on the definition presented above. Although the purpose of this thesis is to examine a multiprocessed version of subsumption, a sequential version is examined so that the reader may gain some intuition as to the basic ideas that carry over into the multiprocessor version. The algorithm below is of course in pseudo-code. It describes the logic for performing forward subsumption; differences relevant to backward subsumption are discussed in the subsequent paragraph. The code is quite similar to that given by Overbeek and Lusk [8].

```
PROCEDURE FORWARD SUBSUMPTION (NEW_CLAUSE);
      set rc to 0;
      Point nlptr to the 1st literal in the new clause;
      while (rc == 0  and nlptr not = NIL)
            Form the set of literals (S) which are likely to
                  have this literal of the new clause as an
                  instance.
            Set j to 1;
            while (rc == 0
                    and there are more lits in the set S)
                        set the substitution to null;
                        discover if this new literal is an
                           instance of the jth literal in S
                           (adding to the subst if so);
                        if it is an instance
                           form the set of all literals in
                              the new clause except this one
                              (pointed to by nlptr);
                        see if the old clause subsumes the
                           new clause under the substitution;
                        if it does subsume the new clause
                           set rc = 1;
                        set j to j+1;
            end while;
            point nlptr to the next literal in new clause;
      end while;
```

In the above algorithm, the routine which 'sees if the old clause subsumes the new clause under the substitution', is a recursive routine which attempts to map the specified literals of one clause into the literals of another under a given substitution. It is general enough to be used by both forward and backward subsumption.

The logic for backward subsumption is very similar to that for forward subsumption, with a few minor exceptions:

- the search for literals in old clauses is for literals that are less general than the new clause's literal, i.e. literals that are likely to be an instance of the new one,

- the test for subsumption of one clause by the other is reversed, i.e. the recursive routine described must now test to see if the new clause subsumes the old one,

- the outermost while-loop is unnecessary. Leaving out this loop means that only the first literal in the new clause will be visited when attempting to find candidate literals within clauses that may subsume the new one. The reasoning here merits an example:

<u>Example 2</u>

In forward subsumption, consider the oversimplified case where the old clause set consists of only the clause :

Q(x)

and the new clause under consideration is :

P(x) | Q(x).

Of course, the old clause subsumes the new one, but it requires visiting all of the new clause's literals to discover this fact, which is what the forward subsumption algorithm does.

Now consider backward subsumption, and the same simple clauses. In this case, it is sufficient to discover that no old clause contains the predicate P in any of its literals to determine that the new clause will not subsume it.

Next, consider the procedure that tests to see if one literal is an instance of another on behalf of the subsumption routine. This routine is essentially a 1-direction unification routine. It is called 1-direction because unification is tested only in the direction for which subsumption is being performed. It examines pairs of literals, not whole clauses.

For example, consider the case in which it is necessary to determine whether the literal:

P(a)

subsumes the literal

P(x).

A general unification routine would indicate that the substitution x -> a would unify the two literals. But, examination in only one direction is required for this case. Since the substitution of a variable for a constant is illegal, no unification can be performed, and thus no subsumption takes place.

If the roles of the literals were reversed however, the substitution of a for x would be found and the desired subsumption would occur.

6. Multiprocessing Concepts. In the past, dramatic increases in computer speeds were realized due to advances in the technology for producing the electronic components. Today however, the limits of increases available through that method are being approached. Signal propagation delays, which could be ignored previously, have come to be significant. Signal propagation delays may typically be measured in terms of nanoseconds, but fast-logic delays may be measured in terms of picoseconds. In short, other avenues for gaining speed increases should be considered. Multiprocessing is one such avenue.

The term multiprocessing of course implies the use of multiple processors. Sometimes the term is used to refer to any environment consisting of more than one processor, no matter how loosely they may be coupled. Here however, the term refers to processors that are tightly coupled. They communicate and share information, through common memory, about a common problem which they are attempting to solve. The program described in this thesis coordinates work between multiple processors to determine if any subsumption occurs within a clause space.

Coordination of processes running on separate processors often requires mutual exclusion, i.e. the individual processes must be prohibited from accessing the same resource at the same time. That portion of code which accesses the shared resource is called the critical section. Research in this area originally developed out of the study of operating systems [9]. One method developed from that research makes use of special sections of code called monitors [10]. (See the Literature Review section for additonal references on these topics.)

Monitors are used extensively in the program described by this thesis. The following discussion gives a more detailed description of them.

A <u>monitor</u> is an abstract concept consisting of three parts:

(1) the shared resource itself, or a data structure representing the resource,

(2) the code to initialize the shared structures,

(3) the code which performs the critical section operations on the resource.

The operations of a monitor may be called by any program at any time. It is necessary, however, that only one program be able to enter the monitor at one time. From a program's point of view the monitor is a serially reusable resource. This does not imply that the calling programs are completely serialized; they are merely serialized through their critical sections in which they access a shared resource.

Earlier, it was mentioned that the concern in this thesis is for mutual exclusion between closely coupled processes that are attempting to solve portions of a common problem. The relevant type of machine architecture is often referred to as MIMD (Multiple Instruction stream, Multiple Data stream). In this type of machine, the separate processors may be executing separate procedures (multiple instruction streams). This differs from a SIMD architecture where a single instruction stream is executed simultaneously by several processors operating on separate sets of data. The specific hardware described is the Denelcor HEP, although the algorithm described is not specific to the HEP.

The HEP's logic functions are pipelined to gain the desired parallelism. On the HEP, a Process Execution Module (PEM) contains the pipeline. Each 100 nanoseconds a new instruction can enter the PEM, and at the same time an instruction that previously entered the PEM can exit. There are eight steps through the PEM, and thus the total time for one instruction through the PEM is 800 nanoseconds. Although a given HEP may have as many as sixteen PEMs installed, speed-ups of 8 to 12 are attainable on a single-PEM HEP.

The HEP also includes extensions to its resident languages to support parallelism by user programs. The CREATE verb allows the programmer to initiate execution of a subroutine as a separate process, i.e. the subroutine executes in parallel with the mainline. Other verbs permit the user to treat variables as asynchronous if desired.

An asynchronous AWRITE may be done to a variable only if the variable is 'empty', and an AREAD can be done only if the variable is 'full'. These asynchronous routines permit the development of many useful synchronization primitives such as a BARRIER that permits each of several processes to hang at a given location until some specified number of processes have reached the same point.

The monitors described earlier are built as macros which utilize these asynchronous routines. Lusk and Overbeek [10] have developed the monitors in such a way however, that the machine-level details are hidden. The user is provided the luxury of thinking in terms of monitors rather than in terms of low-level details of the HEP. This form of program development makes programs highly portable to multiprocessors other than the HEP, because the macros are all that must be re-written since the machine-dependent details are hidden within them. The use of these monitor macros within the program is discussed in the Procedures section of CODING AND IMPLEMENTATION.

The reader interested in a more detailed discussion of subsumption or multiprocessing is directed to the Literature Review section for references, several of which contain large bibliographies.

C. LITERATURE REVIEW

The preceding sections were intended to provide both a historical perspective of automated theorem proving in general, and a working vocabulary sufficient to understand a discussion of multiprocessed subsumption. The treatment of subsumption and multiprocessing in the literature is now examined.

1. Subsumption. The concept of subsumption was first introduced in J. A. Robinson's landmark paper "A Machine-Oriented Logic Based on the Resolution Principle" [11], where the principle of resolution was also introduced. In that paper, Robinson calls subsumption a search principle, to distinguish it from inference principles such as resolution.

Subsumption is not a rule of inference. Rather it is a process that may be used in conjunction with rules of inference to speed up the rate of convergence to a desired proof. It accomplishes this by deleting clauses that are less general than other clauses in the clause space.

Robinson describes subsumption as:

If C and D are two distinct nonempty clauses, we say that C subsumes D just in case there is a substitution % such that C% $\leq$ D (where $\leq$ is used for subset notation).

He also gives the subsumption theorem:

If S is any finite set of clauses, and D is any clause in S which is subsumed by some clause in S - {D}, then S is satisfiable if and only if S - {D} is satisfiable.

The subsumption principle is then stated:

One may delete, from a finite set S of clauses, any clause D which is subsumed by a clause in S - {D}.

And finally, Robinson gives an algorithm for deciding if one clause subsumes another.

Robinson's paper is regarded as a landmark because of its contribution of the resolution principle, thus its treatment of subsumption is often overlooked. Robinson 'invented' the resolution principle with computing machines in mind, however. Thus, he did not wish to stop with a principle that is merely correct theoretically. He knew that the principle must be applicable in real time on a computer.

Subsumption assists in making resolution faster by reducing the work that has to be done. By deleting subsumed clauses, the number of clauses that must be examined (and thus the number of resolvents that may be produced) can be greatly reduced. In other words, the resolution principle works without subsumption, but can be speeded up with its application. Further, no loss of power occurs by the application of subsumption.

This last statement can be argued to some extent. For example, sometimes a strategy is employed which governs the use of inference rules. The set of support strategy [2] is one which divides the clause space into two sets, one of which is said to 'have support'. The strategy is often useful because it helps to focus a theorem prover's attention on the problem rather than allowing it to wander aimlessly. There is the risk however that a clause with support which is needed in the proof may be subsumed. However, if the subsuming clause is given support when this happens, then the desired deduction is permitted.

Before listing any additional papers which discuss subsumption, it might prove useful to mention several textbooks that cover the topic in varying degrees of detail. They may provide additional information if the reader feels overwhelmed at this point.

First, the text by Chang and Lee [3] is an excellent reference on automated theorem proving. In an appendix they even include a small theorem prover written in Lisp. It is, of course, limited in its capabilities, but serves as a good instruction device. The chapter on the resolution principle includes a discussion of subsumption as a deletion strategy. The sample theorem prover in the appendix performs a test for subsumption by unit clauses (this is fairly common because it is moderately easy to perform).

Second, the text by Loveland [12] contains a chapter on subsumption. His definition of subsumption is somewhat stronger than that which we have been using; it requires that the subsuming clause C and the subsumed clause D have the relationship that $\forall C \rightarrow \forall D$ (clause C with all variables universally quantified implies clause D) is valid. He uses the term theta-subsumption for the form of subsumption that Robinson suggested (including the number of literals test to ensure that shorter clauses are not deleted). He states that theta-subsumption "is a more useful subsumption criterion for deletion or replacement than the stronger subsumption criterion". Thus, the term subsumption will continue to be used here as defined by Robinson.

Loveland goes on to suggest that a theorem prover may reach the point where it is not worth the effort to perform a subsumption test because such a test can be quite time-consuming. He therefore suggests its use in limited applications such as only when the subsuming clause is a unit clause. His suggestion however, is made under the assumption that the subsumption check is performed using "the resolution apparatus already available". (See Chang and Lee's text for an algorithm which demonstrates this type of apparatus.) It will be seen in some of the upcoming papers that this is not necessarily the case; refer to the description of the forward subsumption algorithm in the previous section, and note that it contains no mention of any resolution apparatus.

Third, is Nilsson's text [13]. It contains very limited detail about each of the areas discussed here, but it is fairly easy to read for those not looking for an in-depth study.

Finally, the most recent book in this area, is <u>Automated Reasoning</u> by Wos, et. al. [2]. Remember that Wos suggested the term automated reasoning for this subject. In this book, the topic of automated reasoning is treated with automated theorem proving handled as a sub-area. Each subject is treated at several levels of detail. For example, there are chapters that speak on an intuitive level, as well as chapters that treat the topics in a formal manner. Many examples are provided.

The topic of subsumption is included throughout the text as it relates to each of the other topics under discussion. Chapter four is where the best 'stand-alone' treatement of subsumption appears. The exercises are extremely helpful. They provide some of the best insights into subsumption, and answers are provided to assist the reader.

The following papers are covered in approximately chronological order of publication date, but that ordering is ignored if two or more papers should logically be grouped together. Although the citation for each paper is for the original publication of the paper, several of the landmark papers are reprinted in the two-volume set Automation of Reasoning [4, 5], which may be more readily available.

In 1964, the paper "The Unit Preference Strategy in Theorem Proving" [14] was published by Wos. This paper introduces an 'enhancement' to the basic resolution principle devised by Robinson. It suggests that unit (single literal) clauses be preferred for forming resolvents. Note that this paper actually appeared before Robinson's. It references Robinson's paper as "to be published". Thus, subsumption is not mentioned by name; rather the deletion strategies are grouped under the heading of 'subsidiary strategies'. Few details are provided; the use of deletion strategies is only briefly mentioned.

Wos, et. al. [15] also published "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving" in 1965. The set of support strategy was mentioned in their previous paper [14], but is treated in detail in this one, including a proof of a theorem giving sufficient conditions for its logical completeness. In the Examples section of the paper, details of various program executions (employing the set of support strategy) are given. Here again, subsumption is not treated in any detail. It is simply mentioned that the given statistics show a difference in the number of clauses generated and the number retained, due to the use of deletion strategies.

Kowalski has published several papers which discuss subsumption. In three of these [16, 17, 18], the discussion of subsumption centers around the fact that "certain inference-related rules can be defined only in the context of search strategies. Deletion of subsumed clauses is an important example." Kowalski's Ph.D. thesis [17] gives an example (repeated in Loveland's text [12] pp.207-208) using the set-of-support strategy where no refutation is obtainable using backward subsumption, but is possible with no backward subsumption. He states that the faulty situation is entirely a problem of the search plan and theta-subsumption combination, and that one way to maintain completeness with such strategies is to remove only certain subsumed clauses.

Kowalski's paper "Linear Resolution with Selection Function" [19], discusses subsumption in the context of yet another version of the resolution principle, SL-resolution, i.e. linear resolution with selection function.

Sibert [20], in his paper "A Machine-Oriented Logic Incorporating the Equality Relation", develops the theoretical basis for the design of theorem-proving programs with the equality relation built-in. He states that this is not enough "for an efficient procedure", however. Thus, he goes on to treat subsumption at some length as a technique for increasing the efficiency of refutation procedures.

Green [21], in his paper "Theorem-Proving by Resolution as a Basis for Question-Answering Systems", shows how "a question-answering system can be constructed using first-order logic as its language and a resoution-type theorem-prover as its deductive mechanism". The paper contains a description of the program (QA3) which includes a subsumption component.

Loveland [22], in his paper "A Linear Format for Resolution", shows that resolution remains complete when the refutations permitted are restricted by three special conditions on any two clauses and their resolvent.

Allen and Luckham [23] describe "An Interactive Theorem-Proving Program" in their paper. The program has a subsumption component which is described.

Plotkin [24], in his paper "A Note on Inductive Generalization", does not discuss the topic of subsumption directly. Instead, he is interested in a discussion of the generalization of literals. He uses subsumption as a method for defining a "more general literal", i.e. literal L1 is more general than literal L2 if L1 subsumes L2.

J.A. Robinson's paper [25] "Automatic Deduction with Hyper-Resolution" does not address the topic of subsumption. It is worthy of note here however, because several of the following papers are concerned with hyper-resolution, and this paper is the best starting point for the interested reader.

"An Implementation of Hyper-Resolution" by Ross Overbeek [26] is an excellent reference for a description of data structures and some of the algorithms employed in one of the most successful theorem-proving programs to date. The subsumption program described in later sections of this thesis was developed using many of the ideas presented in [26], e.g. FPA lists, only one copy of a literal in the data structures, etc.

Winker [27], in his paper "An Evaluation of Qualified Hyper-Resolution" describes extensions to the hyper-resolution program to support 'qualifiers', which provide certain advantages in problems "involving functions which are not defined for some values of their arguments". His paper references Overbeek's. He states that the use of qualifiers is compatible with deletion of subsumed clauses.

McCharen, Overbeek, and Wos [28], in their paper "Problems and Experiments for and with Automated Theorem-Proving Programs" describe the performance of their program on several problems from the trivial to the very difficult (on which the program failed). They include statistics about the number of unifications attempted and successful, and the number of clauses generated and retained (not subsumed).

Wos [29] in his paper "Automated Reasoning: Real Uses and Potential Uses", mentions some of the capabilities of their program (including subsumption), while describing some its successes in answering open questions and speculating on future applications.

In their three articles "Data Structures and Control Architecture for Implementation of Theorem-Proving Programs" [8], "Logic Machine Architecture: Kernel Functions" [30], and "Logic Machine Architecture: Inference Mechanisms" [31], Lusk and Overbeek discuss in great detail their implementation of a new theorem proving system designed to aid researchers in the field. In the first article, they even include a brief discussion of some multiprocessing concepts which they hope the new system will eventually be able to exploit.

Many of the ideas for data structures and control structures which they describe have been incorporated into the subsumption routines described in this thesis. And, of course, the major thrust of this thesis is to develop a version of subsumption which exploits some of the multiprocessing power available today (through Overbeek, et.al. at Argonne National Labs).

Indeed, Lusk and Overbeek have written a paper [32] entitled "Research Topics: Multiprocessing Algorithms for Computational Logic" in which they suggest research topics for anyone interested in the area. Two of the suggested topics are multiprocessor versions of subsumption and demodulation.

It should be noted here that the theorem proving system developed by Lusk and Overbeek has been placed in the public domain. Therefore, in addition to their articles describing its implementation, they have also published manuals describing its use. "Logic Machine Architecture Inference Mechanisms - Layer 2 User Reference Manual" [33] describes the interface to the layer two of their system. It contains the necessary information to write LMA-based systems which reside at layer 3; such systems might include "theorem provers, reasoning components for expert systems, or customized deduction components". "The Automated Reasoning System ITP" [34] describes the use of a powerful automated theorem prover which has been developed from the LMA tools and which is provided as part of the package. One of the tools in the package, of course, is the subsumption component developed using the data and control structures described in the papers above.

2. Multiprocessing Concepts. Probably the best place to start in the literature is with the March 1973 issue of the ACM Computing Surveys. In that issue, J.L. Baer published the article, "A Survey of Some Theoretical Aspects of Multiprocessing" [35]. Baer's article contains an excellent bibliography of the relevant multiprocessing literature at that time. In the article, Baer examines language features which help exploit parallelism (including additional instructions for multiprocessing architectures), problems such as mutual exclusion, and more theoretical aspects such as models for parallel computation (e.g. parallel flowcharts). An appendix attempts to classify the contemporary multiprocessors.

The article "Concurrent Programming Concepts" [36] by Per Brinch Hansen appeared in the December 1973 issue of the ACM Computing Surveys. The paper discusses programming language features such as critical regions and monitors.

In March 1977, an entire special issue of the ACM Computing Surveys [37] was devoted to Parallel Processors and Processing. The articles in that issue are "Associative Processor Architecture - A Survey" [38], "A Survey of Parallel Machine Organization and Programming"

[39], "Pipeline Architecture" [40], and "Multiprocessor Organization - A Survey" [41]. The latter two articles relate most closely to this thesis because they discuss hardware topics relevant to the HEP. Each article in the issue contains a good bibliography for further reading.

In the 1978 Proceedings of the International Conference on Parallel Processing [42], the paper "A Pipelined, Shared Resource Computer" [43] describes a version of the HEP computer that has four PEMs. Of course the other papers in that proceedings cover topics of interest in parallel processing, but none of them are as closely related to this thesis.

A 1981 <u>Tutorial on Parallel Processing</u> was published by the IEEE Computer Society [44]. This publication contains reprints of some of the papers mentioned previously, e.g. Enslow's multiprocessor organization survey [41]. Smith's paper on the HEP [43] is reprinted under the section on dataflow architectures, but the reader is informed that the HEP is not a dataflow machine; that it is related to dataflow because of its synchronization mechanism.

Another paper of interest in the tutorial is "Some Computer Organizations and Their Effectiveness" [45]. This paper is a reprint of a classic paper that introduced the taxonomy of computers into SISD, MIMD, etc. It, of course, describes the shared resource multiprocessor model on which the HEP is based.

Other tutorial reprints relevant to this thesis are "Communicating Sequential Processes" [46] which discusses the fact that "component processors must be able to communicate and to synchronize with each other", and "The Programming Language Concurrent Pascal" [47] which describes the use of monitors in a systems programming language.

The text Introduction to Computer Architecture [48] contains a good survey and description of the various types of multiprocessors available in the early 1980s.

The HEP Hardware Reference Manual [49] is an introduction to the HEP computer and "is intended for audiences with a general or moderately technical interest". It includes an overview of the HEP system and architecture, the CPU, the data switch, and the data memory.

Finally, the two papers "Use of Monitors in Fortran: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors" [10] and "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors" [50] provide an excellent discussion of the monitors used by the subsumption program described later in this thesis. Even though one of the titles mentions Fortran explicitly, the same monitors have been provided for use by C programs as well.

II. METHODS AND PROCEDURES

A. PLAN OF ATTACK

1. General. This section should provide a high-level overview of the program which has been developed. The section CODING AND IMPLEMENTATION provides the low-level implementation details for those interested.

For portability, the program is written in the programming language C. The forward and backward subsumption routines have borrowed heavily from the work of Overbeek [26]. The idea, of course, is to take advantage of the best efforts in current uniprocessor versions of subsumption and to expand those efforts to exploit parallelism.

Two major levels of parallelism are integrated into the program. They will be referred to as "coarse-grained" and "medium-grained" parallelism.

The "grain" of the parallelism refers to the size of the problems being performed in parallel. For example, the addition of two integers is a very small problem and would probably be too small to justify the overhead necessary to spawn a new task. On the other hand, the problems of backward and forward subsumption are much larger (especially for large clause spaces). This is the coarse-grained level.

Within forward subsumption, a newly generated clause may have to be compared against every old clause to see if any of them subsumes it. It may have to be compared against every old clause, but this is not very likely. Overbeek [26] has developed methods for selecting candidate clauses that are most likely to subsume the new one. This program takes advantage of those methods of selecting candidates. Assuming that there is a list of several candidate subsuming clauses, it may be the case that only the last candidate clause subsumes the new one. In a sequential program this fact is discovered only after checking all previous candidates in the list. In the parallel program, candidates are examined simultaneously. This is the medium-grained level.

2. Process Creation. It is important to note here that creation of a process may be quite an 'expensive' operation. Creation is not prohibitively expensive on the HEP, but it may be on other multiprocessors. Thus, in the interest of generality, the program described here attempts to reduce that overhead by spawning parallel processes only once, allowing them to stay quiescent until released by some other process. The net effect is additional memory usage instead of additional CPU time.

Early in the mainline code, the program creates forward subsumption as a parallel process. Forward subsumption is suspended until it is released later by the mainline (with a problem to solve). Also, early in the mainline, several smaller forward sub-processes are started. These are the routines that aid forward subsumption in examining candidate, subsuming clauses in parallel. These processes are also suspended until they are released with a problem to work on. They are released by forward subsumption when it has determined the set of candidates.

This situation poses an interesting question: What if there are more candidate clauses than there are processes? This does not become a problem because the program is designed such that it will always run in uniprocessor mode. When the processes are told that there is a problem to work on, they each ask for a subproblem, i.e. a candidate clause to examine. If a process completes examination of one candidate (with no subsumption occurring) then that process asks for a new candidate to examine. If no subsumption occurs, the processes will each eventually be told that the problem is over by exhaustion. If subsumption does occur within a process, the process can signal an early end to the problem so that the others do not continue to execute needlessly. The pseudo-code description of the algorithm in the next section will make these points much clearer.

3. <u>The Algorithm</u>. Pseudo-code of selected portions of the algorithm are given in Figures 1, 2, and 3. Figure 4 illustrates the various parallel paths which may be followed through the code. Finally, a verbal description is provided which connects the ideas presented in the pseudo-code and in Figure 4. In each case, the

```
mainline procedure:

read num_mac_processes;   /* run fwd & bwd in parallel ? */
read num_fwd_processes;   /* how many med-level fwds */
read num_bwd_processes;   /* how many med-level bwds */
if (num_mac_processes == 2)
    CREATE (forwardsubsum);
i = 1;
while (i < num_fwd_processes)
      CREATE (fwd (slave));
      i = i + 1;
end while;
i = 1;
while (i < num_bwd_processes)
      CREATE (bwd (slave));
      i = i +1;
end while;
get 1st new clause;
while (more_new_clauses)
     fwd_occurred = 'no';
     bwd_occurred = 'no';
     if (num_mac_processes == 1)
          call forwardsubsum;
     if (fwd_occurred == 'no'  or  num_mac_processes == 2)
          hang (synch point 1); /* activate forward */
          call backwardsubsum;
          hang (synch point 2); /*tell forward prob over*/
     if (bwd_occurred == 'yes' or fwd_occurred == 'no')
          call integrateclause;
     else
          num_fwd_subsumed = num_fwd_subsumed + 1;
     get next new clause;
end while;
pgm_done = 'yes';
hang; /*allow forward to terminate */
notify the fwd (slaves) of program termination;
notify the bwd (slaves) of program termination;
stop;

end program;
```

Figure 1. Mainline Pseudo-code

```
forwardsubsum procedure:

forever
     hang (synch point 1);  /* wait for a prob to work on */
     if (pgm_done == 'yes')
          break out of the forever loop;
     rc = 0;
     new_lit = 1st literal in the new clause;
     while (rc == 0   and   new_lit  not=  NIL)
           form the set S of literals from the data
                 structures that clash with new_lit;
           clashlit = 1st literal in S;
           while (rc == 0   and   clashlit not=  NIL)
              see if new_lit will unify with (is an
                   instance of) clashlit,
                   forming a substitution if so;
              if (the two lits unify)
                 start fwd (slaves) on the new problem;
                 rc = fwd (master);
              clashlit = next lit in S;
           end while;
           new_lit = next literal in new clause;
      end while;
      hang (synch point 2); /* wait for bwd to end */
      if (num_mac_processes == 1)
          break out of the forever loop;
end forever;
return (rc);

end program;
```

Figure 2. Forwardsubsum Pseudo-code

```
fwd procedure:

rc = 0;
forever;
      ASKFOR a new problem (pt clashcls to the candidate);
      if (program terminating  OR
         (this problem is solved and I am the master))
           break out of the forever loop;
      if (this problem is solved)
           continue;  /* back to top of forever loop */
      form L the set of remaining lits in the new clause;
      call subsum (L, current substitution);
      if (subsump occurs)
           signal this problem over;
           LOCK
           fwd_occurred = 'yes';
           UNLOCK
end forever;
if (fwd_occurred  ==  'yes')
   rc = 1;  /* master indicates that subsump occurred */
return (rc);

end program;
```

Figure 3. Fwd Pseudo-code

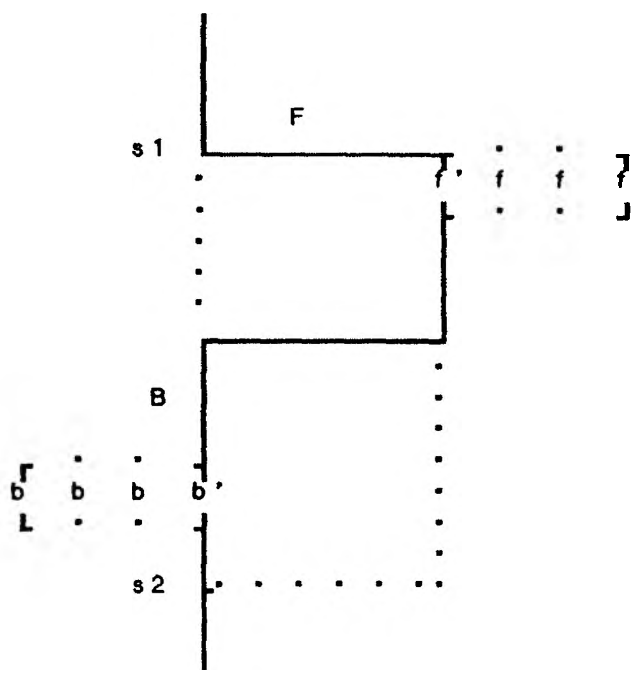

Figure 4. Parallel Paths

parallelism is stressed, ignoring low-level details of clause construction and manipulation.

For the sake of simplicity, the pseudo-code does not include the backward subsumption routines. They have been omitted because they are so similar to the forward subsumption procedures (Figures 2 and 3).

a. Mainline Description. The mainline procedure (Figure 1) is fairly straightforward. It first reads in two values that indicate how many separate processes will be spawned to run in parallel. The first variable, num_mac_processes takes on a value of either 1 or 2, indicating whether forward and backward subsumption should be run sequentially or should be run as two parallel processes. The next variables, num_fwd_processes and num_bwd_processes, can take on any integer values (up to the maximum number of processes supported by the hardware). They indicate the number of "medium-grained" processes that should be employed by forward and backward subsumption respectively, to check candidate clauses for subsumption.

The first while loop creates all but one of the medium-grained processes (named fwd) for forward subsumption. The pseudo-code for these processes (Figure 3) shows that they are suspended in an ASKFOR monitor, to be activated later by forward subsumption. The fwds are activated and the last of them is invoked by the forward subsumption routine itself when a set of candidate clauses have selected for examination. The bwd routines play a similar role in backward subsumption.

The next line of code in the mainline gets the first new clause. In a real theorem prover, this new clause would be generated as part of the refutation; here, however, new clauses are simply read in from a file.

In the following descriptions it is convenient to think in terms of 'fork' and 'join' operations. For example, if forward and backward subsumption are to be run in parallel, it is natural to think of forward subsumption as being forked as a separate process while the mainline invokes backward subsumption. When the two processes finish checking for subsumption, it is natural to think of them as joining together again in the mainline.

Inside the mainline's large while loop a new clause is examined for subsumption, then another clause is retrieved. Before the processing of each new clause, indicators are set stating that no subsumption has occurred. These indicators are changed in fwd and bwd respectively, if they discover that subsumption does indeed occur. Next, if forward and backward subsumption are to be executed sequentially, then forward subsumption is invoked. If they are to be performed in parallel, the assumption is that forward subsumption was previously spawned as a parallel process and is suspended waiting for a problem to work on.

Figure 4 demonstrates the alternative of executing forward and backward subsumption sequentially or in parallel. In Figure 4, the 'F' represents the forward subsumption routine and the 'B' represents the corresponding backward subsumption routine. Following the flow of control from top to bottom, the solid lines represent a sequential execution. The dotted lines represent alternative paths that were utilized for the parallel tests. For example, at synchronization point one (labeled s1 in Figure 4) forward subsumption may either be entered via a call, or activated as a parallel process running concurrently with backward subsumption. A major point to remember here, is that the forward subsumption routine is created as a separate process much earlier in the program, and then is immediately suspended. At the fork, forward subsumption is simply activated.

At synchronization point two (labeled s2 in Figure 4), the forward and backward subsumption routines come together as a single process. This synchronization point represents the join operation. The join operation must occur prior to the 'if' statement that checks to see if any backward subsumption occurred. The 'if' statement checks for backward subsumption first because the potential 'pay off' is larger for backward than for forward. This is possible because, in forward subsumption at most one clause may be subsumed, but in backward subsumption several old clauses may be subsumed.

When the mainline's large while loop is exited, the forward routine is started one last time. This time, it is notified that the program is ending, and thus it may terminate itself. Finally, the medium-grained processes (fwds and bwds) are notified to terminate.

b. Forwardsubsum Description. If forward subsumption (Figure 2) is running as a separate process, it stays in a loop until the program is terminated. Otherwise, it returns to the mainline after each call. Assuming that the forward subsumption routine is running as a parallel process, it is suspended at the top of the loop waiting to be activated by the mainline immediately prior to starting the backward subsumption routine.

When the forward subsumption process is activated, it begins by examining a literal of the new clause. It is important at this point to recognize the fact that a given literal may appear in several old clauses. If so, there is only one copy of the literal in the data structures. That copy points to each containing clause. Maintaining a single copy saves space and helps to speed the search process.

A set of literals which may contain the current literal of the new clause as an instance is formed from the data structures. The first literal from that set is checked against the current literal of the new clause. If the new one is an instance, then the clauses containing the old literal become a set of candidate clauses that may subsume the new one. If all literals are used from the set with no subsumption occurring, then the next literal from the new clause is chosen and the process starts over again.

The forward subsumption process invokes one of the medium-grained processes (fwds in Figure 3) to check candidate clauses to see if any subsumes the new one. When this invocation occurs, if there are several fwds waiting for a problem, they all start executing. Each one asks for a problem to work on. The problems are, of course, candidate clauses to check against the new clause.

Figure 4 shows the available parallelism within the forward subsumption routine. When the forward subsumption routine wishes to check the candidate clauses for subsumption, it invokes a master copy of fwd (designated by f' in the Figure 4) to perform the tests. If there are parallel copies of fwd (designated by f in Figure 4) suspended, waiting for a problem to work on, they are all activated and run concurrently with the master copy.

c. Fwd Description. Each copy of fwd (Figure 3) is capable of examining all candidate clauses by itself. Each fwd consists of a loop in which it enters the ASKFOR monitor and requests the next candidate to be examined. It then tests to see if the candidate subsumes the new clause, signaling an end to the problem if so, looping to get the next candidate if not.

Each fwd contains code to determine if the mainline is ending, so that it may end also. There is special code executed by the master fwd (f' in the above description) that permits it to return to the forwardsubsum routine rather than to continue looping. The master fwd must be able to return in order to report the results, i.e. whether or not any subsumption occurred.

The ability of each fwd to examine all candidate clauses in the set is what provides the capability to test the program on a sequential machine. On a sequential machine, only one copy of fwd is used.

B. CODING AND IMPLEMENTATION

1. Detailed Program Description.

a. General. This section provides a detailed description of the data items and logic used in the subsumption program. It includes a description of the driver program which reads in clauses and constructs their internal representation for the subsumption program to process. Line number references are to the program listing in Appendix A.

The clauses are represented internally in structures declared to be of types: 'clauses' and 'items'. The declaration of these structure types appear in lines 20 - 33. The clause headers are stored in structures of type clauses and the literals are stored in structures of type items. Figure 5 gives a conceptual view of an internal representation of a clause. In the figure, items are below the clause header. Items below a predicate may be either on the same level, e.g. an argument to the predicate, or subordinate to other items, e.g. a1 is subordinate to (is an argument of) f1.

p1(x1,f1(a1),y1) : q1(x1,y1)

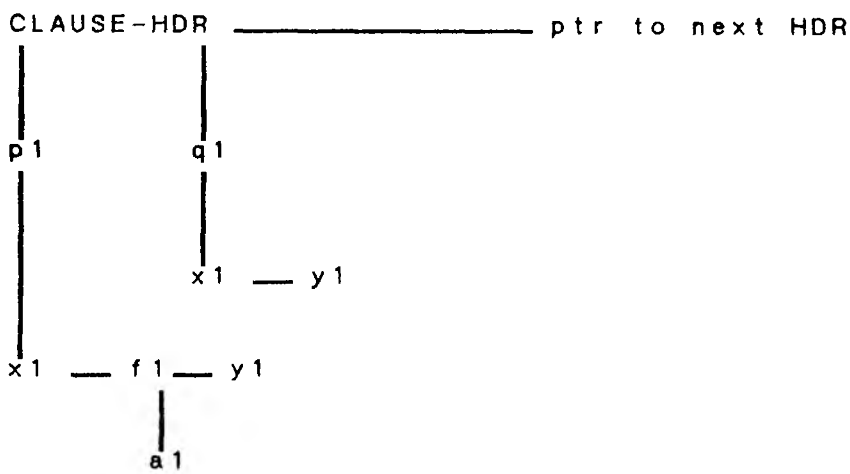

Figure 5. Clause Internal Representation

The structure type definitions contain references to pointers. Any time the word pointer is used here, it actually refers to an array subscript. This method was used partially because subscripts tend to simplify the debugging chore. Also, in C, there is a strong relationship between pointers and arrays, strong enough that they are usually treated simultaneously in texts.

Clauses which have been integrated as part of the clause space are stored in the structures oldclause and olditem. Clauses which are 'new' (newly generated by a theorem prover) are stored in newclause and newitem.

b. Compile-time Variables.

NIL - assigned the value -9. Any negative number would work. This variable was defined because some implementations of C assign the value 0 to NULL, and a negative value is definitely required since an array subscript of 0 is valid in C.

STDERR - assigned the value 2. This is the unit to which error messages are written from the procedure 'error'.

MAXOLDITEMS - the dimension of olditem.

MAXNEWITEMS - the dimension of newitem.

TOKENSIZE - the length of a variable, constant, or predicate. For the present, all are forced to a single letter and a digit, therefore this value is 2.

SUBSIZE - the dimension of the substitution array.

LITSIZE - the dimension of several 'temporary' arrays into which literals are sometimes copied. For example, subsum copies a domain-set literal into a temporary location before passing it to unify because unify may alter the literal when it performs a substitution. Note that in such a case, the LITSIZE must be large enough to hold the original literal plus any added as part of a substitution.

MAXLITPERCLS - in a clause header, the number of pointers to literals contained in that clause.

MAXCLAUSES - the dimension of oldclause, i.e. the maximum number of clauses that may appear in the data structures.

MAXLITS - the dimension of the litlist, i.e. the maximum number of literals that may appear in the data structures.

MAXLITTOCLS - the number of pointers in each litlist entry to a clause containing that literal. Note that these pointers are in the litlist and not the items containing the predicates.

MAXFPATOLIT - in an fpa entry (terminology employed by Overbeek in [26]), this is the number of pointers to litlist entries for literals containing such an fpa.

FPASPERHASHV - the number of fpa entries defined for each possible 'hash-to' location in the fpa list. For example, if the FPAMODVAL (next variable) has a value of 5, then there are 5 possible places to hash to in the fpa list. Since collisions may occur, we need several slots at each 'hash-to' location, say 10. If there are 5 hash-to locations and 10 slots at each then the fpa list has 50 entries. Note that the dimension on the fpa list is FPASPERHASHV * FPAMODVAL.

FPAMODVAL - the number of 'hash-to' locations in the fpa list (see previous variable).

c. <u>Structure Type Definitions</u>.

items - each entry contains a type (p-predicate, v-variable, etc.), a predicate sign, an id (the predicate, variable, etc.), a pointer to this predicate's entry in the literal list (for predicates), and a left and a right pointer. The left pointer points to items 'owned' by this item, for example in f1(x1,x2) the function f1 owns the variables x1 and x2. The right pointer is used to point to the next item at its same level. In this example, x1 would point to x2.

clauses - Each entry contains a set of pointers to predicates of the clause in items. Each also contains a pointer to the next clause header in the array of clause structures. Finally, each entry contains a delete indicator to tell whether that clause has been deleted by a program or not. It is initially set to '-' but is changed to 'd' when the clause is deleted. At first, this approach does not seem nearly as clean as simply adjusting the pointer in the previous clause header to point to the subsequent one. The chosen approach seems to work better here however, because there may be pointers to the deleted clause at other locations in the data structures. To seek these out and delete or adjust them also would be a fairly large task. The major drawback to the present approach is that routines examining the clause headers must be prepared to skip 'deleted' ones.

litlists - this list contains one entry for each literal in the integrated data structures. The predptr points to the associated predicate. The clsptrs point to each clause which contain this literal.

fpalists - contains fpa entries. An fpa entry consists of a predicate and its sign, an argument to that predicate, the number of that argument within the predicate, and a set of pointers to literals that contain the fpa. This list is used to quickly find literals which are good candidates to unify with a given literal.

substitution - a substitution entry consists of a variable (to be substituted for) and a pointer to the term in items to substitute for that variable.

d. External Variables. Lines 53 - 76 of the program listing define the external variables several of which are of types defined above. Note that there are several macro invocations, e.g. ADEC(fs), which define external variables to be used by the monitor macros. Documentation for these macro definitions can be found in Lusk and Overbeek's [10].

e. Macro Definitions. Recall from the high-level description of the program that the ASKFOR monitor defines how subproblems are to be determined for solution by the fwds and bwds. The ASKFOR is actually very general-purpose. The FWDGETPROB (and BWDGETPROB) defined here actually form part of ASKFOR monitors. For example, the FWDGETPROB indicates that another subproblem is available if the fsub variable is greater than -1. In that case, the next problem to solve is indicated by the next subscript value of fsub. Note that the limitation is imposed that clsptr[fsub] not be NIL. This is because the next problem to be solved is the next clause pointed to by the litlist entry of the clashable literal. The litlist points to all clauses containing that literal. The end of the list is marked with the NIL value.

The FWDRESET and FWDPROBST macros are used to reset the fsub to -1 and to indicate that a new problem is available for solving, respectively.

f. The Procedures.

main - The mainline routine contains the initialization code. It then consists of two large loops, one to read in the oldclauses (existing before a subsumption check), and one to read in newclauses. The loop that reads the newclauses may read in a new clause each time, or it may just use the same new clause an indicated number of times. The option of using the same new clause over and over permits timings to be taken for several clauses by typing in only one. The rest of the logic of the mainline is as described in the high-level description.

integrateclause - This routine copies the new clause into the integrated data structures (oldclause and olditem) one literal at a time. Each time, before it copies a literal, it verifies that the literal does not already exist. If it does already exist, then the current clause is merely added to the litlist entry for that literal, and the new copy of the literal is removed.

After all literals for the clause are copied, entries are added to the fpalist for each literal in the clause. A new 'end' is then marked in the oldclause structure.

buildliteral - This routine constructs an entire literal. It does so by calling itself recursively an item at a time. It calls getoken to return the next token (predicate, variable, sign, etc.) from the input stream. '?' marks end of input for oldclauses and for newclauses. ';' and ')' are skipped on input; they are merely remembered as the previous token for purposes of parsing the literals. For each predicate, function, variable, and constant this procedure calls builditem to construct an item to place in the data structures; then it makes the recursive call to build the next item in the literal. ';' marks the end of a literal.

builditem - This short routine constructs the next item. It will construct it in any items type-of-object at the specified location.

litexistchk - This routine checks in the specified items to see if some newly added literal already exists there. The assumption is that the calling routine will remove the new copy if it is already there.

litcompare - This procedure examines two literals in the specified items to see if they are identical. rc = 1 indicates that they are.

addtolitlist - This procedure adds a new entry to the literal list if that literal does not already have an entry there. If there is one in the literal list already, then it merely augments the literal list entry with a pointer to the new clause containing that literal.

buildfpalist - This procedure constructs each fpalist entry from the specified items beginning at the item pointed to by start_item. It calls addtofpalist to enter each new entry into the list. Note that a special case arises if a literal has no arguments (proposition, e.g. p1()). In this case, a single fpa entry (for argument number zero) is constructed with blank argument. This is necessary because the subsumption routines gain access to the old literals through the fpalist entries.

addtofpalist - This procedure adds an fpa entry constructed by buildfpalist to the fpalist. It calls hashfpa to determine the point in the fpalist to which this particular entry hashes. If that entry is already in the list, it merely adds a litlistptr value to it pointing to the litlist entry for the new literal containing this fpa. If that entry is not already in the list, the routine adds the entry and gives it an initial litlistptr to the literal. The new entry is added at a location in the list pointed to by new_fpa[hashval], and then new_fpa[hashval] is incremented by 1. There is one entry in new_fpa for each possible location in the fpalist to which the hash may occur. This new_fpa entry contains a pointer into the fpalist to the next open position for entries hashing to that location. If an entry hashes to a full location, then a sequential search is performed looking for an open slot.

hashfpa - This routine hashes the predicate and argument to a slot in the fpalist based solely on the predicate and argument number within that literal. The argument itself is not used in the hash because all arguments at that position in a given predicate should hash to the same location so that when searching for literals that may unify, both variables and constants (or functions) will be found at the same spot. A variable at a given argument position might possibly unify with a constant or function depending on the direction of subsumption; e.g. p1(x1) will subsume p1(a1).

prtclses - This routine prints the specified clauses from the indicated items beginning at the particular clause indicated by cc (current clause). It skips 'deleted' clauses. It will print either the number of clauses indicated by howmany or print until it reaches the end of the clauses, whichever comes first. It calls prtlit to print the individual literals. Note that the two routines together rebuild the clauses for printing, i.e. they must put back in the '!', '(', ')', and ';' symbols that were stripped out when the literals were stored in their internal format.

prtlit - This routine is called (by prtclses) to print an individual literal from an entire clause. The two routines work in concert as described above.

getoken - This routine acquires the next token from the the input stream of clauses. A token may be a predicate, constant, variable, or function. Also included are '!', ';', '(', and ')'. '?' is a special token used to delimit each of the sets of clauses, i.e. oldclauses and newclauses. Note that predicate signs (+ or -) are also retrieved from the input but are only used to set a flag, they are not returned as tokens.

This routine calls getnextchar to retrieve the next character from the input in its attempt to construct a token. Note that following the construction of an item such as p1 it gets one additional character from the input to determine if that next character is '(' which would indicate that the token currently in hand is a predicate or function. Additional context is used to determine which. The additional character retrieved is placed back in the input stream by ungetc to be retrieved later as part of the next token.

getnextchar - This short procedure retrieves the next character from the input stream of clauses, skipping the following characters:

blank, \n (linefeed),
\r (carriage return),
\t (tab), and
';'.

forsubsum - Note the external data definition immediately prior to this procedure. This procedure performs the forward subsumption check, i.e. it checks to see if the current 'new' clause is subsumed by an old clause.

The outermost loop is a 'forever' loop that is exited if the variable pgmdone is assigned the value 'y' or if the routine is called in uniprocessor mode (fwd_bwd_parallel='n').

The nested while loop executes until either all literals in the new clause have been examined or until subsumption of the new clause is discovered (rc = 1). Within this loop is a call to getclashlits. This call retrieves a list of all literals in the data structures that may clash with the current literal of the new clause. A nested for loop examines each literal in the list to see if it unifies with the current literal of the new clause.

If any literal unifies with the current new one, a FWDPROBST is executed and a call to fwd is performed. If no literal unifies with the new one, the next literal in the new clause is examined.

Note that BARRIERs are at the top and bottom of the forsubsum routine to keep it in synchronization with the mainline which may execute backsubsum in parallel.

fwd - This routine examines the clauses that may subsume the new clause. Each clause is pointed to by a literal in litlist that has been determined to be unifiable with a literal in the new clause. The clauses that the litlist entry point to constitute the subproblems, and thus pointers to them are retrieved via calls to the ASKFOR monitor. Note that one literal of the old clause is already known to unify with a literal in the new clause, so that literal is skipped and not rechecked.

Recall that there may be multiple copies of this routine running in parallel, therefore it executes as a 'forever' loop. A copy may exit only if it is the 'master' copy, or if the entire program is terminating.

backsubsum and bwd - These procedures looks for old clauses which are subsumed by the current new clause. They differ from forsubsum and fwd in the following ways:

(1) only the first literal in the new clause is clashed with old clauses in the data structures.

(2) the clauses selected for clash are of

course selected as possible instances of the new clause.

(3) when subsumption occurs rc = 1 is set, but control is not returned until all old clauses are examined to see if they are subsumed; also, any old clauses subsumed are 'deleted' by setting their delete indicators to 'd' (see clause description above).

subsum - This routine checks to see if a clause in the domain set subsumes a clause in the range set (Overbeek terminology). Most of the logic in this routine is outlined well in Lusk and Overbeek's [8]. They refer to the equivalent routine as 'subtest'.

It accomplishes its purpose by examining each literal in the range set and seeing if it will unify with some literal in the domain set under the current substitution. This current substitution may have been supplied by a calling routine such as forsubsum, or it may be passed down by subsum itself in recursive calls.

As an example, if we wish to see if p1(x1) ¦ q1(x1) subsumes q1(a1) ¦ p1(a1) we must first try to unify p1(x1) with q1(a1). This obviously fails at comparison of the predicates. Next, we try p1(x1) against p1(a1). This succeeds with the substitution x1/a1. Next, when we attempt to unify q1(x1) with q1(a1), we must perform the substitution x1/a1 before attempting the unification.

Note that in the procedure, variables in the 'subsuming' literals are renamed (see renamevars) before attempting the unification. This is done of course, because the same variable name may appear in both clauses, but it is actually a different variable when in separate clauses.

unify - This procedure performs the unify function, although it is only a 1-direction unification in the sense that it does not attempt to produce a most general unifier for the two literals. For example, full unification would produce the general unifier x/a, y/b for the two clauses p(x,b) and p(a,y). Here, however it is necessary to discover if one clause is an instance of another for purposes of subsumption - above, neither is. For p(x,b) and p(a,b) the substitution x/a will permit p(x,b) to subsume p(a,b), therefore the unification is one direction, i.e. the direction in which subsumption is to be performed.

This routine calls itself recursively attempting to unify the individual items. Note that the substitution in force at any given point is passed down to the next level.

skiplit - This short routine copies a clause header to a temporary location skipping the literal specified by lit_to_skip.

getclashlits - Note that this routine has an external structure definition above it that is used for definition of temporary data items.

This procedure examines the arguments of a literal and uses the fpalist to find literals that may unify with it in the specified direction, i.e. forward or backward. It builds an fpa for each argument and calls fpamatchk to look for matching entries in the fpalist. Note that an fpa for a proposition, e.g. p1(), is a special case in that the argument number is 0. Each literal discovered by fpamatchk is added to a temporary clash list (see addtotempclash below).

After all matching fpas have been discovered and pointers to their associated literals (actually their litlist entries) have been placed in the temporary clash list, the entries in tempclash are examined. Each entry has associated with it a reference count. If the reference count matches the number of arguments that were in the literal, i.e. this literal unifies with the new literal in every possible argument, then the literal is added to the litstoclash list of literals to clash with the current new literal. Note that the argument number of 0 for propositions is treated special here because the argument number will not match the reference count of 1.

fpamatchk - This routine looks for fpalist entries that match a given argument of some specified predicate. First, it hashes the predicate and argument number using the same routine (hashfpa) that is used when when fpalist entries are created. Once reaching the hash-to position, it searches forward looking for matching fpa entries.

Note that fpa entries do not have to match exactly. For example, when performing forward subsumption, the fpa entry for p1(x1, will match with p1(a1, from the new clause because the literal containing the x1 may subsume the one containing the a1.

isavariable - This procedure examines the first character of any specified item to determine if it is a variable. Variables begin with one of the letters s-z. Predicates, constants, and functions begin with one of the letters a-r or A-Z, as in ITP [34].

addtotempclash - This procedure adds a litlist pointer to the temporary clash list for getclashlits. If the 'new' entry is already there, it merely increments the reference count (see getclashlits), otherwise it adds the entry and initializes the reference count to 1.

copyterm - This routine will copy any items type-of-object beginning at from[f1] to a location beginning at to[t1]. It considers each item to have a right and left side as depicted in Figure 5, where for example, the function f1 has the left side a1 and the right side y1. The left side is subordinate to the item, and the right side is on its same level.

If copy_type = 'l' (left) copyterm will copy only the first item and its left side; 'r' it copies only the item and its right; 'b' the item and both of its sides; any other value - it copies only the one item. Litlistentrys are copied for predicates even though the litlist entries do not point to the copies, only to the originals.

The procedure is recursive for the left and right parts if they are to be copied. Note that if an outside routine calls this one to print an item and its left, the item is first copied, then a recursive call is made to copy the left item and <u>both</u> of its sides.

renamevars - This procedure renames the variables in a literal. It is typically called before the unification process is performed. The variables need to be renamed in a 'subsuming' clause because the same variable names may appear in two clauses, but of course represent different variables.

The variables are given names that cannot be supplied as names by the user. Recall that variables must begin with one of the letters 's' - 'z'. These are merely changed to '1' - '8', respectively in the internal representation. Note that the variable names are actually altered, so a calling would normally copy the literal to some temporary location, before calling this routine.

substitute - This procedure performs the substitution for variables in a literal which is being unified with another. Only variables which have renamed values (see renamevars) will be replaced. Consider the case where 61 (representing x1) is to be replaced by g1(b1,x1). The substitutuion entry would contain:

(1) the variable name - 61 here

(2) a pointer to the substitution - the first

item in the function g1(b1,x1).

When the substitution is performed the first item and its entire left side is substituted for the indicated variable. Here, the 61 would be replaced by g1(b1,x1).

error - This routine is a general routine that can be called by any procedure that detects an array overflow. It accepts arguments containing the name of the procedure detecting the error, the name of the variable that exceeded the array size, and the name of the variable containing the maximum array size.

The procedure prints out a message giving this information, and then halts program execution by issuing an exit (1) instruction.

2. Testing. The testing of the subsumption program was essentially done in three stages.

In stage 1, a version of the program was tested which contained no parallelism, in order to verify that the program would correctly perform the subsumption process. Most testing at this stage was done on an IBM personal computer. Eventually, the program was uploaded and tested on a VAX 11/780 at UMR.

At this point, it should be mentioned that a version of the monitor macros was available for use on a VAX. This version of the macros, for the most part, generates no code; it just allows compilation of the program with no changes to the source. Therefore, it was possible to execute the 'multiprocessor' version of the program on the VAX merely supplying the necessary parameters to tell the program that it should spawn no parallel processes.

In stage 2, the program was augmented with the monitor macros. Recall that this version of the program is coded in such a manner that it runs successfully in uniprocessor mode. Thus, it was possible to perform initial testing of the multiprocessor version on the VAX at UMR.

Stage 3 of the testing involved executing the program on the HEP multiprocessor at Argonne National Labs. At first, this testing was done in uniprocessor mode just to verify that the program would still work. Then, testing was done with various numbers of parallel processes.

At this point, some problems were encountered which would occasionally lead to ABEND situations. Debugging was usually done by placing print statements at selected points in the routines in question. Of course, locks had to be set when such printing was done because the routines were running in parallel and potentially could interfere with each other.

When each problem was found, a correction was applied and the testing retried. Usually, for large changes, the program was retested on the VAX at UMR first to verify that the logic remained sound in uniprocessor mode.

Typically, an ABEND would arise in the form of a memory protection error. Referencing beyond the end of an array could cause one of these errors, but those errors encountered in this testing were all caused by two (or more) parallel routines attempting to change the same external variable at the same time.

It is, of course, best to minimize the number of external variables in a program, but in this environment, it is almost impossible to do away with them completely since the processes communicate through common memory locations and often must examine the same location. Care must be exercised when an external item may change however, and locks set if necessary.

Innumerable problems arose during stage 3 of the testing phase. Several of them were due to communication over phone lines between UMR and Argonne.

Most problems however, were due to the fact that the HEP had a new version of its UNIX operating system installed, which tended to be quite unstable. It was not unusual to get numerous intermittent memory protection errors for no apparent reason. The task of determining which errors actually resulted from program errors was usually accomplished by simply rerunning the program several times. Program-induced memory protect errors would usually recur. This version of the operating system also tended to lose files frequently, which caused considerable additional logon time just to restore files. Also, some of the C routines that perform asynchronous operations were not initially available.

This discussion is not intended to downgrade the HEP. Instead, it should be stated that the HEP does present a very powerful environment for research. This thesis just happened to begin when things were in a state of change at Argonne. The situation has progressively improved, and hopefully, will continue to do so.

The results of the testing phase, and suggestions for further tests, may be found in the RESULTS and CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH sections, respectively.

III. RESULTS

A. GENERAL

This section presents the results of the stage 3 testing (see TESTING section) done on the HEP computer. These results are intended to demonstrate that when certain forms of parallelism are available in a given problem, that the program can indeed exploit that parallelism; in fact, that it can speed up certain portions of the program by as much as one order of magnitude on a single-PEM HEP. The results also demonstrate the fact that when these forms of parallelism are not available, performance may be downgraded somewhat.

Because these two conflicting cases may arise, subsequent sections (CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH) discuss program options that permit the choice of which, if any forms of parallelism to use, and when to use them. They also discuss other forms of parallelism that may be desirable to build into the program. In this section however, the experimental results obtained thus far are simply presented, with minimal comments or suggestions.

Associated with each of the following sets of test results, is a figure that shows the approximate run-time of each test in milliseconds. These figures are not intended to provide exact values, but rather to demonstrate the relative times of one run to another. Each of these figures has an axis called "Type and Number of Processes". The labeling on these axes should be described before continuing.

On each axis, if the label "fbp" is printed, it means that forward and backward subsumption are run in parallel. If this label is not present, then it is assumed that they are run sequentially, i.e. forward, then backward. All other labelings contain two numeric values, e.g. "4,8", which means that 4 copies of fwd are running in parallel with 8 copies of bwd. Note that both forms of label may be present for a single execution. If a given run shows "1,1" and does not show fbp, then it is essentially a uniprocessor execution.

B. TEST 1

Test 1's data has been set up such that several of the old clauses are candidates to subsume the new clause, but only the last one actually does subsume it. Therefore, in sequential mode, several old clauses must be examined to discover that the last one subsumes. In a parallel mode however, several old clauses may be examined at one time.

For this test, the timings were taken immediately before the FWDPROBST instruction (line 814) and immediately after the rc = fwd (master) instruction (line 815). The timings were taken at these locations because, in this test, the concern is not with the overall run-time of the subsumption algorithm, but rather only with speeding up that portion of the algorithm which is responsible for examining candidate clauses.

Note from Figure 6 that five executions are represented with increasing numbers of fwds devoted to the problem. They range from a uniprocessor execution to an execution with 16 fwds. The approximate timings recorded are: 140, 71, 39, 20, 17, and 13 ms.

Note that the range on the timings is from approximately 140 ms to 13 ms, or about one order of magnitude speed-up. This is the most dramatic speed-up obtained with the program thus far, although the same is obtainable with multiple bwds on a similar problem.

C. TEST 2

Test 2's data has been set up such that no subsumption actually occurs in either direction, but several old clauses are candidates to subsume the new clause in forward subsumption and several old clauses are candidates to be subsumed by the new clause in backward subsumption. Since no subsumption occurs, both forward and backward subsumption must be run to determine this fact.

For this test, the timings were taken immediately before the if (fwd_bwd_parallel == 'n) instruction (line 296) and immediately after the backsubsum () instruction (line 301). The timings were taken at these locations in order to calculate only the time spent in forward and backward subsumption, without including any unnecessary

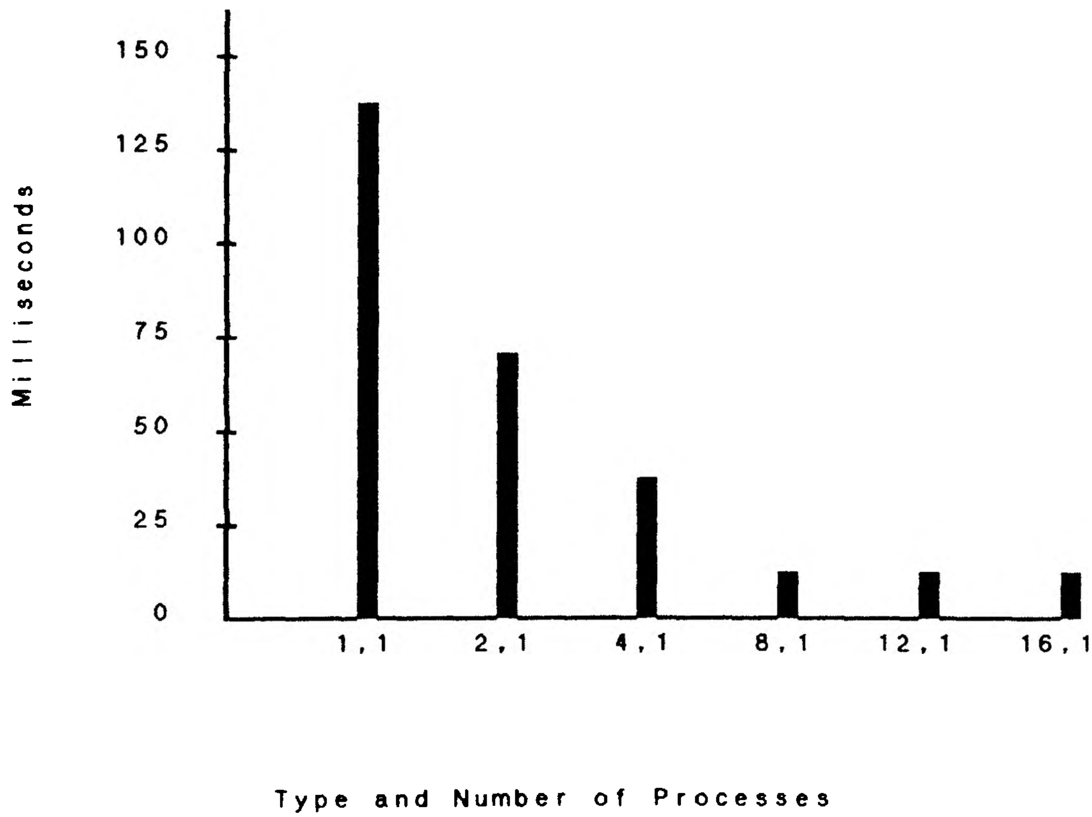

Figure 6. Test 1 execution times

overhead of time looping in the mainline, etc.

Note from Figure 7, that five executions are represented: a uniprocessor run, a run with forward and backward in parallel, and then three other runs with forward and backward in parallel and various numbers of fwds and bwds devoted to examination of candidate clauses. The approximate timings recorded are: 298, 250, 175, 139, and 120 ms.

Note that more overhead is present in these timings than in Test 1, because we are examining the entire time through the forward and backward subsumption routines, not just the time to examine a set of candidate literals. It should be apparent however, that a reasonable speed-up (about 2.5 times, here) is attainable for the overall subsumption algorithm for this type of problem.

D. TEST 3

Test 3's data has been set up such that several of the old clauses forward subsume the new clause, but also such that the new clause backward subsumes several old clauses. This test is designed to demonstrate the aspect of the program that gives backward subsumption precedence when both forward and backward are run in parallel. Recall that if they are running in parallel and both forward and backward occur, then the backward is used because with forward only the new clause can be subsumed, but with backward, several old clauses may be subsumed.

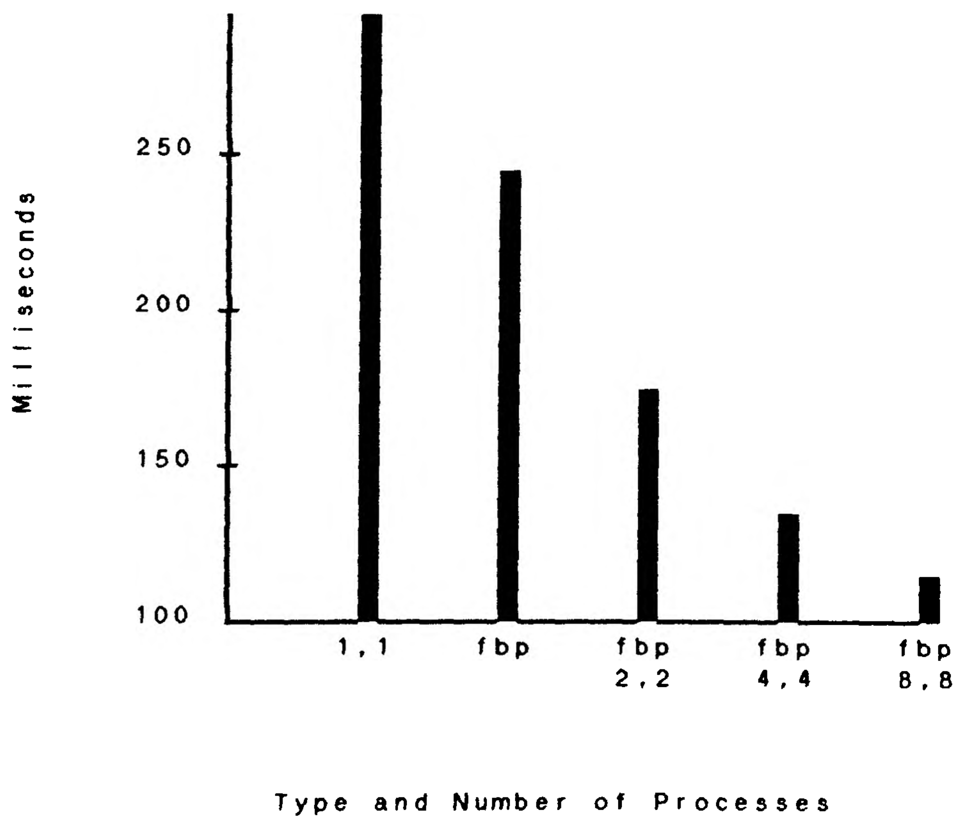

Figure 7. Test 2 execution times

For this test, the timings were taken at the same locations as in Test 2, i.e. immediately before the invocation of forsubsum and immediately after the invocation of backsubsum (lines 296 and 301). As in Test 2, the concern is only with the time spent in both forward and backward subsumption, without including any additional overhead.

Note from Figure 8 that four executions are represented: a uniprocessor run, a run with forward and backward in parallel, and then two other runs with forward and backward in parallel and various numbers of fwds and bwds devoted to examination of candidate clauses. The third run gives 8 copies each of fwd and bwd. The fourth run tries 14 bwds, because we know that in this run there are more clauses backward subsumed. It does not show much speed-up over the "8,8" run however, due to the fact that even though not much forward subsumption occurs, there are several old clauses that are candidates to forward subsume.

The approximate timings are: 50, 178, 70, and 61 ms. Figure 8 indicates therefore, that the uniprocessor run of this test takes less time than any of the other runs. The important statistic for this test however, is not merely the run-time. Rather, it is the number of clauses subsumed. Note that Figure 9 indicates that the uniprocessor version of the program only found one clause to be subsumed (forward). In the other runs, 28 clauses were subsumed (backward). Also, when fwds and bwds were

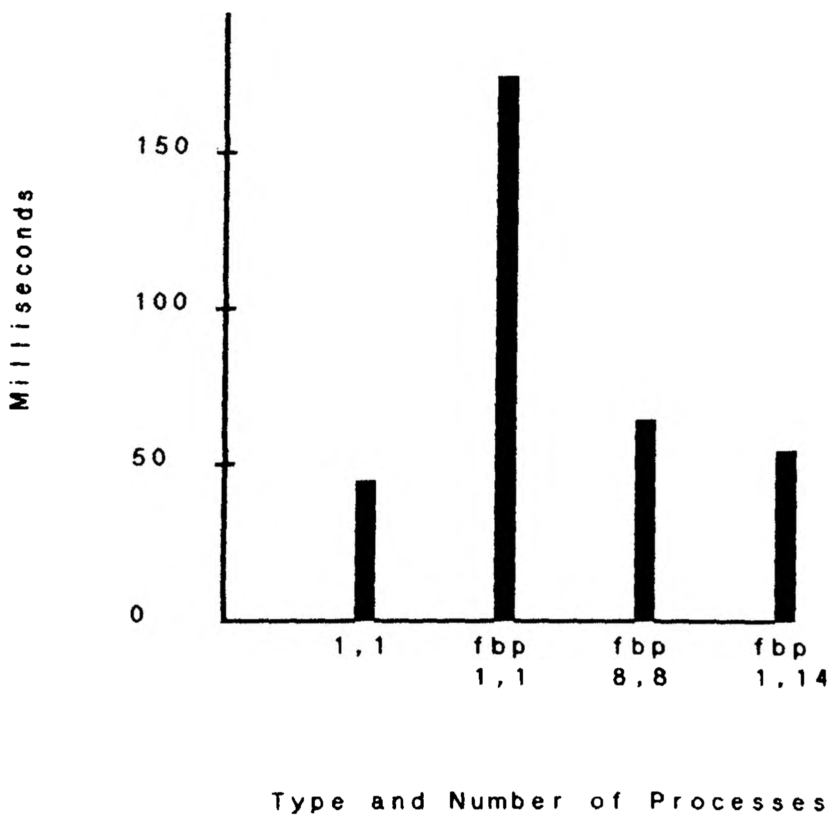

Figure 8. Test 3 execution times

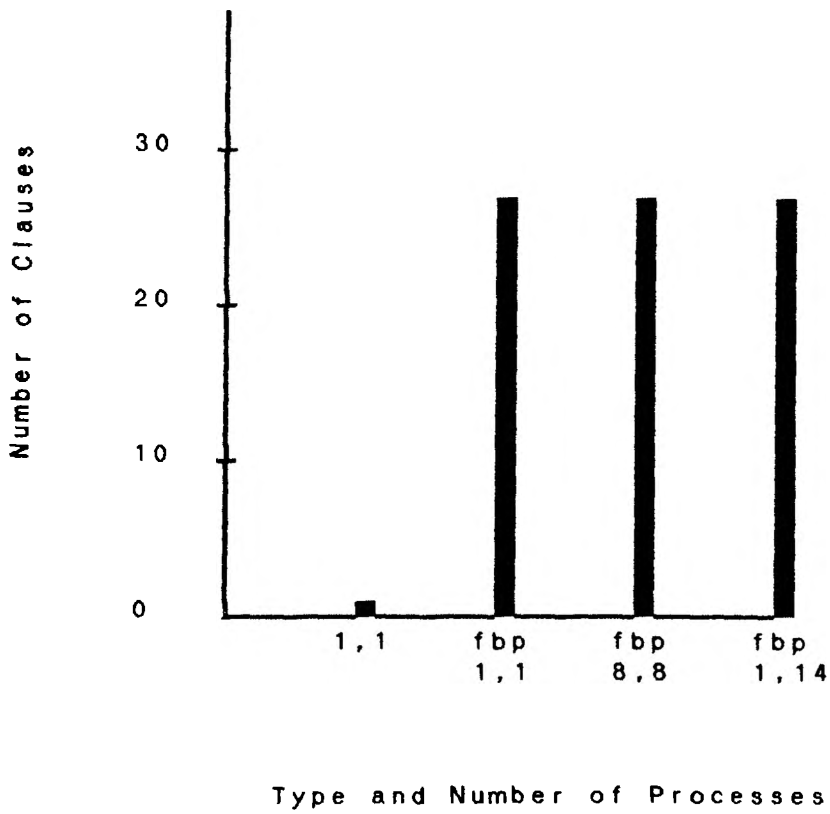

Figure 9. Test 3 number of clauses subsumed

added to assist the process, the run-time came back down close to the time for the uniprocessor execution.

E. TESTS 4 and 5

Tests 4 and 5 are not 'invented' data. Rather, they are problems that have been published elsewhere, e.g. in [28, 2]. For these tests, the problems were first run through the automated theorem proving system ITP [34], saving generated clauses in a file. Recall that the program described in this thesis has a mainline routine which reads in new clauses and then invokes the subsumption routines. The mainline is playing the role of an automated theorem prover which generates the new clauses and then invokes the subsumption routines.

Test 4 is the group theory problem G5 described in "Problems and Experiments for and with Automated Theorem-Proving Programs" [28] by McCharen, Overbeek, and Wos. All axioms and the denial of the theorem are stated in [28] along with statistics from an execution of their theorem prover documenting the number of clauses generated, number kept, etc.

For this test, the timings were taken immediately before and after the large loop in the mainline which reads in new clauses and invokes the subsumption routines (lines 256 and 311). This time is the entire time to solve the problem for all generated clauses.

In all executions of this test, 30 clauses were subsumed. In the cases where forward and backward were run sequentially, 30 clauses were forward subsumed. However, in those cases where forward and backward were run in parallel, 10 clauses were forward subsumed, and 20 were backward subsumed.

Note from Figure 10 that five executions are represented: a uniprocessor run, a run with forward and backward in parallel, and runs with various numbers of fwds and bwds.

It is interesting to note that for this problem, the parallelism actually slowed the runs down. This fact is partially due to the additional overhead encountered in telling parallel routines that there is no work for them to do. For example, in Test 1 it was demonstrated that if there are several candidate clauses to subsume a new clause, then multiple fwds speed the process up. Here however, it can be seen that in cases where there is only a small number of candidate clauses, multiple parallel processes may actually slow the subsumption down because time must be spent informing the parallel processes that there is no work for them to do.

In the cases where forward and backward were run in parallel, synchroniztion overhead was encountered, and yet there were no additional clauses subsumed to compensate as there were in Test 3.

Test 4 could be used to support an argument against the use of any parallelism at all in a subsumption program.

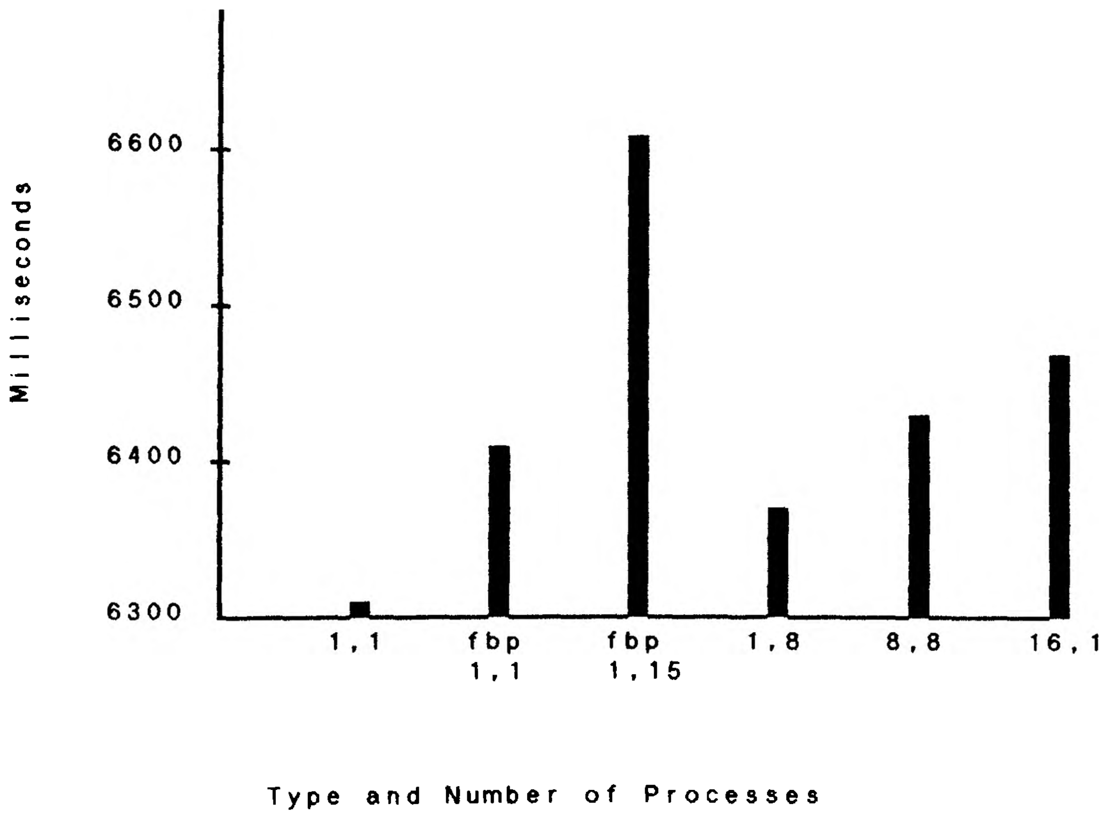

Figure 10. Test 4 execution times

Test 5 would not support such an argument however, as may be seen in the subsequent paragraphs.

Test 5 is the 'Missionaries and Cannibals' problem presented in <u>Automated Reasoning Introduction and Applications</u> [2] by Wos, Overbeek, Lusk, and Boyle. All axioms are stated in [2] including four clauses created just for subsumption purposes; they enable an automated theorem prover to subsume generated clauses which represent trips that result in distress to the missionaries.

For this test, the timings are the same as for Test 4. In all executions of this test, 8 clauses are forward subsumed. No backward subsumption occurs.

Note from Figure 11 that seven executions are represented: a uniprocessor run, a run with forward and backward in parallel, and several runs with various numbers of fwds and bwds.

The run with forward and backward in parallel and no fwds or bwds did the best in terms of run-time. This is because there were several new clauses generated but for which no subsumption at all occurred. For those clauses, the sequential execution had to run forward followed by backward just to discover that no subsumption occurred. In the case where forward and backward were done in parallel however, the fact that no backward subsumption occurred was discovered at approximately the same time as the fact that no forward subsumption occurred.

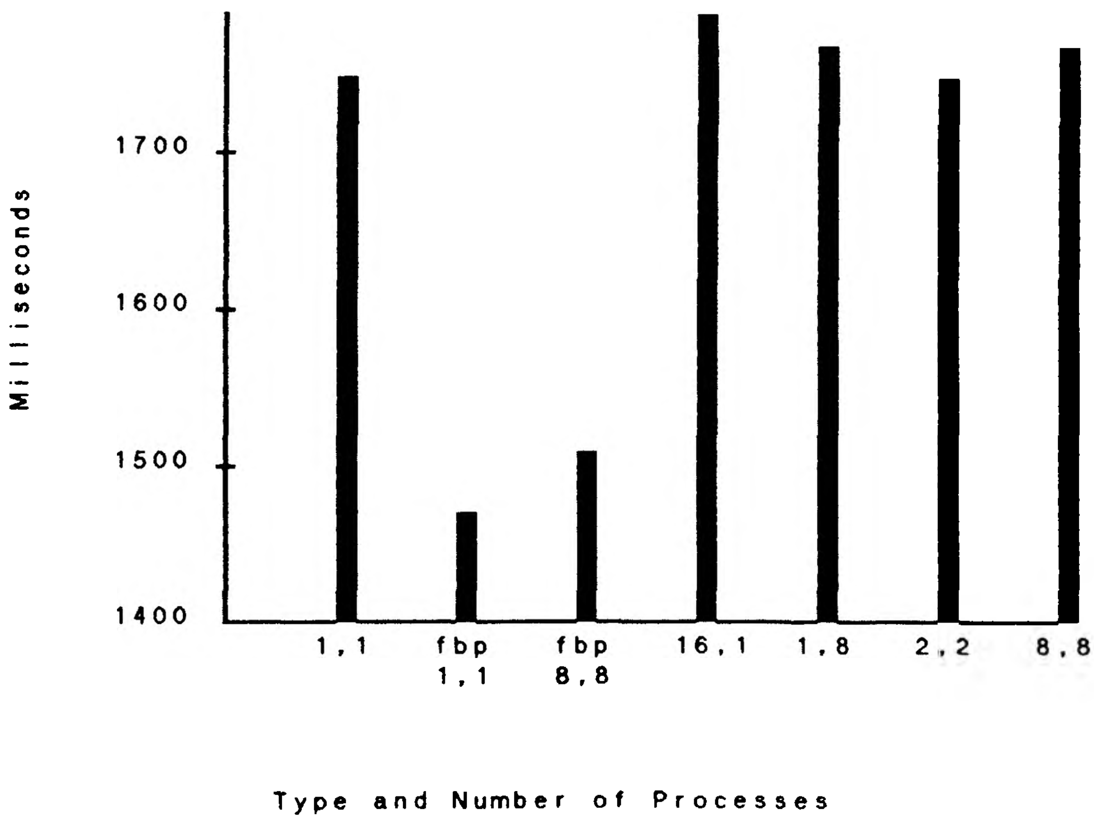

Figure 11. Test 5 execution times

This test, like Test 4, did not benefit from the use of multiple fwds and bwds because there were not several 'candidate' clauses associated with each new clause.

IV. CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

Tests 1, 2, and 5 support the fact that substantial speed-ups are realizable through the use of parallelism in a subsumption program. Tests 3 and 4 demonstrate that more research is needed to learn when the use of that parallelism is warranted for a given problem.

For problems where very little, if any, subsumption is expected to occur it would seem natural to have forward and backward subsumption running in parallel because those cases would require both routines to be invoked during examination of most of the new clauses.

For problems where the new clauses tend to clash with several old ones, it would be best to use several fwds and bwds. Sometimes this case would be relatively easy to spot by simply looking at the original set of clauses. If they have widely varying predicates in the individual clauses, then such parallelism would probably not be warranted. On the other hand, if several clauses have the same predicates and arguments in some of their literals, then such parallelism might prove useful.

It would seem that the future research to determine when to use a particular form of parallelism should follow three steps.

First, several runs should be made of different problems from widely varying classes. This should help to build some intuition as to when each form of parallelism would prove most useful. Second, an attempt should be made to move from the intuitive level to a point where the gained understanding can be described to others. Finally, the descriptive level should be incorporated into the program, i.e. the program should be able to decide when to employ a particular form of parallelism.

Note that these suggestions of how to approach the use of parallelism in subsumption do not differ very much from the way in which other theorem proving parameters are approached. Powerful theorem provers such as ITP have a wide range of variables which may be altered by the user. For example, they often include options of whether subsumption is to be used at all or not, whether short clauses can subsume long ones, how to weight clauses to determine which to choose next from the set of support, etc. The wrong choice on some of these options can often drastically affect the time to a proof, and in some cases can even prohibit a proof from being found.

In addition to the further research described above, more research is also needed to determine additional areas within subsumption where parallelism may be exploited. Several areas seem quite promising.

The first promising area to exploit new parallelism is in the selection of candidate clauses. At present, the program can check multiple candidates for subsumption after the candidates have been chosen, but the process of choosing the candidates is still sequential. Recall that Overbeek's [26] method of selecting candidates (only clauses that contain a literal which clashes with a literal in the new clause) is used in the program. The addition of parallelism would either have to be tailored to work with that method, or an alternative method could be developed.

Two ideas come to mind to develop parallelism in the selection of candidates.

First, multiple literals in the new clause could be examined to determine if there are any literals in the data structures which clash with them. This would involve changes to procedures forsubsum and backsubsum.

Second, for each literal in a new clause, multiple old literals could be examined in parallel to see if they clash with the new one. This would involve changes to procedure getclashlits. The new parallelism would probably prove useful in clause spaces where several clauses have literals with the same predicates.

Another promising area to exploit new parallelism is in the subsum procedure which is invoked by both forward and backward subsumption. This routine examines a pair of clauses to see if the first subsumes the second. For each literal in the 'subsuming' clause, it might prove useful to check it against every literal in the 'subsumed' clause simultaneously. Of course, this would only be useful for large clauses, i.e. clauses with several literals.

Beyond the new parallelism ideas mentioned above, it might prove interesting to experiment with making larger changes to the algorithm. For example, forward and backward subsumption might be combined into a single routine called subsumption. Then, when the procedure getclashlits is invoked, it would return not just a list of literals that are unifiable in a single direction, but a list of all literals unifiable in either direction. Then, as suggested above, multiple clauses containing the clashable literals could be tried at once. The subsum procedure would need to be changed to determine not only if clause X subsumes clause Y, but also to determine if clause Y subsumes clause X (running forward and backward in parallel could then be performed at this level). Also, subsum could be coded so that it would only try one of the two directions if backward subsumption had previously occurred, very similar to the way things are done in the current algorithm.

Another research idea is to make _all_ problems that can be done in parallel part of a pool, and to make the spawned processes intelligent enough to accept any type of problem to work on. This is perhaps the most elegant approach for a final version of the algorithm, but when still in the experimental stage it tends to add unnecessary complexity in controlling the number of processes that are devoted to a particular problem type.

Of course, areas of automated theorem proving other than subsumption are wide open for research when combined with multiprocessing. Overbeek and Lusk [32] suggest demodulation as one possibility.

BIBLIOGRAPHY

[1] Robinson, J. A., "Computational logic: the unification computation", pp. 63-72 in Machine Intelligence 6, ed. by B. Meltzer and D. Michie, Halsted Press (1971).

[2] Wos, L., Overbeek, R., Lusk, E., and Boyle, J, Automated Reasoning Introduction and Applications, Prentice-Hall (1984).

[3] Chang, Chin-Liang, and Lee, Richard Char-Tung, Symbolic Logic and Mechanical Theorem Proving, Academic Press (1973).

[4] Siekmann, Jorg, and Wrightson, Graham, editors, Automation of Reasoning 1 Classical Papers on Computational Logic 1957-1966, Springer-Verlag (1983).

[5] Siekmann, Jorg, and Wrightson, Graham, editors, Automation of Reasoning 2 Classical Papers on Computational Logic 1967-1970, Springer-Verlag (1983).

[6] Bledsoe, W., and Loveland, D., editors, Automated Theorem Proving: After 25 Years, American Mathematical Society (1984).

[7] Lewis, H., and Papadimitriou, C., Elements of the Theory of Computation, Prentice-Hall (1981).

[8] Overbeek, R., and Lusk, E., "Data Structures and Control Architecture for Implementation of Theorem-Proving Programs", pp. 232-249 in _Proceedings of the Fifth Conference on Automated Deduction, Springer-Verlag Lecture Notes on Computer Science_, Vol. 87, ed. Robert Kowalski and Wolfgang Bibel (1980).

[9] Hoare, C. A. R., "Monitors: an operating system structuring concept", _Communications of the ACM_, pp. 549-557 (October 1974).

[10] Lusk, E. and Overbeek, R., "Use of Monitors in Fortran: A Tutorial on the Barrier, Self-scheduling Do-loop, and Askfor Monitors", Technical Report ANL-84-51, Argonne National Laboratory (July 1984).

[11] Robinson, J.A., "A machine-oriented logic based on the resolution principle," _Journal of the ACM_ 12, pp.23-41 (1965).

[12] Loveland, D., _Automated Theorem Proving: a Logical Basis_. North-Holland (1978).

[13] Nilsson, N., _Problem-solving Methods in Artificial Intelligence_. McGraw-Hill (1971).

[14] Wos, L., Carson, D., and Robinson, G., "The unit preference strategy in theorem proving", pp.615-621 in _Proceedings of the Fall Joint Computer Conference_, Thompson Book Company (1964).

[15] Wos, L., Carson, D., and Robinson, G., "Efficiency and completeness of the set-of-support strategy in theorem proving", _Journal of the ACM_ 12, pp. 536-541 (1965).

[16] Kowalski, R., and Hayes, P., "Semantic trees in automatic theorem-proving", pp.87-102 in Machine Intelligence 4, ed. by B. Meltzer and D. Michie, Edinburgh University Press (1969).

[17] Kowalski, R., "Studies in the completeness and efficiency of theorem-proving by resolution", Ph.D. thesis, University of Edinburgh (1970).

[18] Kowalski, R., "Search strategies for theorem-proving", pp.181-202 in Machine Intelligence 5, ed. by B. Meltzer and D. Michie, Edinburgh Press (1970).

[19] Kowalski, R. and Kuehner, D., "Linear Resolution with Selection Function", pp. 542-577 in Automation of Reasoning 2, ed. by Jorg Siekmann and Graham Wrightson, Springer-Verlag (1983).

[20] Sibert, E., "A machine-oriented logic incorporating the equality relation", pp. 103-134 in Machine Intelligence 4, ed. by B. Meltzer and D. Michie, Edinburgh University Press (1969).

[21] Green, C., "Theorem-proving by resolution as a basis for question-answering systems", pp. 183-208 in Machine Intelligence 4, ed. by B. Meltzer and D. Michie, Edinburgh University Press (1969).

[22] Loveland, D., "A Linear Format for Resolution", pp. 399-416 in Automation of Reasoning 2, ed. by Jorg Siekmann and G. Wrightson, Springer-Verlag (1983).

[23] Allen, J., and Luckham, D., "An interactive theorem-proving program", pp. 321-336 in _Machine Intelligence_ 5, ed. by B. Meltzer and D. Michie, Edinburgh University Press (1970).

[24] Plotkin, G., "A note on inductive generalization", pp. 153-164 in _Machine Intelligence_ 5, ed. by B. Meltzer and D. Michie, Edinburgh University Press (1970).

[25] Robinson, J. A., "Automatic Deduction with Hyper-resolution", _International Journal of Computer Mathematics_. vol. 1 (1965).

[26] Overbeek, R., "An Implementation of Hyper-resolution", _Computers and Mathematics with Applications_. vol. 1, pp. 201-214, Pergamon Press (1975).

[27] Winker, S., "An Evaluation of an Implementation of Qualified Hyperresolution", _IEEE Trnasactions on Computers_ C-25 (8), pp. 835-843 (1976).

[28] McCharen, J., Overbeek, R., and Wos, L., "Problems and Experiments for and with Automated Theorem-Proving Programs", _IEEE Transactions on Computers_ C-25 (8), pp. 773-782 (1976).

[29] Wos, L., "Automated Reasoning: Real Uses and Potential Uses", pp. 867-876 in _Proceedings of the Eighth International Joint Conference on Artificial Intelligence_, vol. 2 (1983).

[30] Lusk, E., McCune, W., and Overbeek, R., "Logic Machine Architecture: Kernel Functions", pp.70-84 in _Proceedings of the Sixth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science_, vol. 138, ed. by D. Loveland, Springer-Verlag (1982).

[31] Lusk, E., McCune, W., and Overbeek, R., "Logic Machine Architecture: Inference Mechanisms", pp. 85-108 in _Proceedings of the Sixth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science_, vol. 138, ed. by D. Loveland, Springer-Verlag (1982).

[32] Lusk, E., and Overbeek, R., "Research Topics: Multiprocessing Algorithms for Computational Logic", Technical Report, Argonne National Laboratory, being prepared for publication (1984).

[33] Lusk, E., and Overbeek, R., "Logic Machine Architecture Inference Mechanisms - Layer 2 User Reference Manual", Technical Report ANL-82-84, Argonne National Laboratory (April 1984).

[34] Lusk, E., and Overbeek, R., "The Automated Reasoning System ITP", Technical Report ANL-84-27, Argonne National Laboratory (April 1984).

[35] Baer, J., "A Survey of Some Theoretical Aspects of Multiprocessing", pp. 31-80 in _ACM Computing Surveys_, Vol. 5, No. 1 (March 1973).

[36] Brinch-Hansen, P., "Concurrent Programming Concepts", pp. 223-245 in ACM Computing Survey, Vol. 5, No.4 (December 1973).

[37] ACM Computing Surveys. Vol. 9, No. 1 (March 1977).

[38] Yau, S., and Fung, H., "Associative Processor Architecture-A Survey", pp. 3-28 in ACM Computing Surveys, Vol. 9, No. 1 (March 1977).

[39] Kuck, D., "A Survey of Parallel Machine Organization and Programming", pp. 29-60 in ACM Computing Surveys, Vol. 9, No. 1 (March 1977).

[40] Ramamoorthy, C. and Li, H., "Pipeline Architecture", pp. 61-102 in ACM Computing Surveys, Vol. 9, No. 1 (March 1977).

[41] Enslow, P., "Multiprocessor Organization-A Survey", pp. 103-129 in ACM Computing Surveys, Vol. 9, No. 1 (March 1977).

[42] Proceedings of the 1978 International Conference on Parallel Processing, ed. J. Lipovski, IEEE Computer Society (August 1978).

[43] Smith, B., "A Pipelined, Shared Resource MIMD Computer", pp. 6-8 in Proceedings of the 1978 International Conference on Parallel Processing, ed. J. Lipovski, IEEE Computer Society (August 1978).

[44] Tutorial on Parallel Processing, ed. R. Kuhn and D. Padua, IEEE Computer Society (1981).

[45] Flynn, M., "Some Computer Organizations and Their Effectiveness", pp. 11-24 in _Tutorial on Parallel Processing_. ed. R. Kuhn and D. Padua, IEEE Computer Society (1981); reprinted from _IEEE Transactions on Computers_. pp. 948-960 (sept. 1972).

[46] Hoare, C. A. R., "Communicating Sequential Processes", pp. 323-334 in _Tutorial on Parallel Processing_. ed. R. Kuhn and D. Padua, IEEE Computer Society (1981); reprinted from _Communications of the ACM_. pp. 666-677 (August 1978).

[47] Brinch-Hansen, P., "The Programming Language Concurrent Pascal", pp. 313-322 in _Tutorial on Parallel Processing_. ed. R. Kuhn and D. Padua, IEEE Computer Society (1981); reprinted from _IEEE Transactions on Software Engineering_, p. 199-207 (June 1975).

[48] _Introduction to Computer Architecture_, ed. Harold S. Stone, Science Research Associates (1980).

[49] _HEP Hardware Reference Manual_, Denelcor, Inc. (1982).

[50] Lusk, E. and Overbeek, R., "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors", Technical Report ANL-83-97, Argonne National Laboratory (December 1983).

VITA

Ralph Milton Butler was born on July 17, 1954 in Columbia, Tennessee. He received his primary and secondary education in Columbia, Tennessee. He received his Bachelor of Science degree from Tennessee Technological University in May 1976, where he was initiated into two honor societies: Kappa Mu Epsilon and Phi Kappa Phi. He worked for over three years as an Analyst - Data Systems for South Central Bell Telephone Company before beginning his graduate work.

He has been enrolled in the Graduate School of the University of Missouri - Rolla since August 1979. He completed his Master's degree in Computer Science in 1981. He has held a Teaching Assistantship and a Chancellor's Fellowship for the entire time of his graduate work. He is a student member of the Association for Computing Machinery. He is also a member of the American Association for Artificial Intelligence. He has been initiated into the Upsilon Pi Epsilon honor society while at Rolla.

APPENDIX A

PROGRAM LISTING

```
#include <stdio.h>
#include <ctype.h>
#define DEBUG y

#define NIL -9
#define STDERR 2
#define MAXOLDITEMS 750
#define MAXNEWITEMS 100
#define TOKENSIZE 2
#define SUBSIZE 100
#define LITSIZE 300
#define MAXLITPERCLS 20
#define MAXCLAUSES 50
#define MAXLITS 100
#define MAXLITTOCLS 50
#define MAXFPATOLIT 50
#define FPASPERHASHV 50
#define FPAMODVAL 15

struct items {
                char type;
                char pred_sign;
                char id[TOKENSIZE];
                int  litlistentry; /*for a pred, pts to litlistentry*/
                int  left;
                int  right;
             };

struct clauses {
                char delind;
                int litptr[MAXLITPERCLS];  /* ptr into items */
                int nextcls;
               };

struct litlists {
                int predptr;   /* ptr into items */
```

```
                    int clsptr[MAXLITTOCLS];
                  };

struct fpalists {
                    char pred[TOKENSIZE+2];
                    char arg[TOKENSIZE+1];
                    int argnum;
                     int litlistptr[MAXFPATOLIT];  /*ptr into litlist*/
                  };

struct substitution {
                        char var[TOKENSIZE];
                        int termptr; /*ptr to subst term in items*/
                      };


int new_lit, old_prev_pred_in, new_prev_pred_in, debug, n1, o1,
    nxt_newitem[1], nxt_olditem[1], new_fpa[FPAMODVAL+1],
    nummacprocs, fwd_bwd_parallel, numfwdsub, numbwdsub,
    numfwdprocs, numbwdprocs, fsub, bsub, clk1, clk2, tot_time;
char fwd_occurred, bwd_occurred, pgmdone;
char token[TOKENSIZE],
     token_type,
     prev_clause,
     curr_clause,
     prev_token,
     pred_sign;
struct items newitem [MAXNEWITEMS];
struct items olditem [MAXOLDITEMS];
struct clauses oldclause [MAXCLAUSES];
struct clauses newclause [1];
struct litlists litlist [MAXLITS];
struct fpalists fpalist [FPASPERHASHV * FPAMODVAL];

/******  monitor declarations  ***********/
ADEC(fs)
```

```
ADEC(bs)
BARDEC(f1)
BARDEC(f2)
LOCKDEC(3)



/******* macro definitions *********/
define(FWDGETPROB,
        ‘if (fsub > -1)
            {
              if (litlist[fclashlit].clsptr[fsub] != NIL)
                {
                  $1 = fsub;
                  fsub++;
                  $2 = 0;
                }
            }'
       )

define(FWDRESET,
        ‘fsub = -1;'
      )

define(FWDPROBST,
        ‘MENTER(fs,0)
         fsub = 0;
         CONTINUE(fs,0,0)
         MEXIT(fs,0)'
      )

define(BWDGETPROB,
        ‘if (bsub > -1)
            {
              if (litlist[bclashlit].clsptr[bsub] != NIL)
                {
```

```
                        $1 = bsub;
                        bsub++;
                        $2 = 0;
                      }
                 }'
            )

define(BWDRESET,
        'bsub = -1;'
       )

define(BWDPROBST,
        'MENTER(bs,0)
         bsub = 0;
         CONTINUE(bs,0,0)
         MEXIT(bs,0)'
       )

main()
{

/****** declare parallel processes ********/
NEWPROC(fwdslv)
NEWPROC(bwdslv)
NEWPROC(forwardproc)

int nxtlit, prtind, rc, i, lit_start, master,
    newclsctr, numnewclses, reuse_newcls_ind;

/****** initialize monitors and associated variables ******/
AINIT(fs)
AINIT(bs)
BARINIT(f1)
BARINIT(f2)
LOCKINIT(3)
fsub = -1;
```

```
bsub = -1;
master = 0;
pgmdone = 'n';
numfwdsub = 0;
numbwdsub = 0;

nxt_olditem[0] = -1;
new_lit = 0;
for (i=0; i < FPAMODVAL+1; i++) {
    new_fpa[i] = i * FPASPERHASHV;
}
token_type = ';';
oldclause[0].litptr[0]  =  NIL; /*1st cls currently has 0 lits*/
oldclause[0].nextcls  =  NIL;
oldclause[0].delind = '-';
prev_clause = -1;
curr_clause = 0;
old_prev_pred_in = -1;
i = 0;

printf("\n\nSubsumption beginning\n\n");

scanf ("%d", &numnewclses);
scanf ("%d", &numfwdprocs);
scanf ("%d", &numbwdprocs);
printf("numnewclses = %d   numfwdprocs = %d   numbwdprocs = %d \n",
        numnewclses,numfwdprocs,numbwdprocs);
debug  = getchar ();  /* skip linefeed in the input stream */
debug  = getchar ();
prtind = getchar ();
reuse_newcls_ind = getchar ();
fwd_bwd_parallel = getchar ();
printf("debug = %c    prtind = %c \n", debug, prtind);
printf("reuse_newcls_ind = %c   fwd_bwd_parallel = %c \n",
        reuse_newcls_ind, fwd_bwd_parallel);
```

```
    for (;;) {
        lit_start = nxt_olditem[0];  /*save place new lit starts*/
        nxtlit = buildliteral (oldclause, olditem, nxt_olditem);
        if (nxt_olditem[0] > MAXOLDITEMS)
           error ("main", "nxt_olditem[0]", "MAXOLDITEMS");
        if (nxtlit == '?')
           break;
        oldclause[curr_clause].litptr[i] = nxtlit;
        i++;
        if (i > MAXLITPERCLS)
           error ("main", "i", "MAXLITPERCLS");
        if (nxtlit == NIL)  /* end of a clause */
           {
              i = 0;
              if (prev_clause != -1)
                 oldclause[prev_clause].nextcls  =  curr_clause;
              prev_clause  =  curr_clause;
              curr_clause++;
              if (curr_clause > MAXCLAUSES)
                 error ("main", "curr_clause", "MAXCLAUSES");
              oldclause[curr_clause].litptr[0] = NIL;
              oldclause[curr_clause].nextcls = NIL;
              oldclause[curr_clause].delind = '-';
           }
        else
           {
              rc = litexistchk (olditem, nxtlit);   /*lit already in olditem? */
              if (rc == NIL)  /* lit did not previously exist */
                 {
                    if (old_prev_pred_in  !=  -1)
                       olditem[old_prev_pred_in].right = nxtlit;
                    old_prev_pred_in = nxtlit;
                    rc = addtolitlist (nxtlit, curr_clause, 'n' /*new*/);
                    olditem[nxtlit].litlistentry = rc;  /*build pred to litlist ptr*/
                 }
              else  /* lit was already in the data structures  */
```

```
                    {
                      nxt_olditem[0] = lit_start;  /* remove this new copy from item */
                      oldclause[curr_clause].litptr[i-1] = rc;  /*pt cls to oldlit*/
                      rc = addtolitlist (rc, curr_clause, 'o' /*old*/);
                    }
                }
        }

        if (prtind == 'y')
            {
              printf("\n\nthe old clauses are \n");
              prtclses(oldclause,olditem,0,MAXCLAUSES);  /* print old clauses  */
            }


        buildfpalist (olditem, 0);


        /******** create the slave processes  *********/
        if (fwd_bwd_parallel == 'y')
            {
              nummacprocs = 2;
              printf("creating forwardproc\n");
              CREATE(forwardproc)
            }
        else
              nummacprocs = 1;

        for (i=1; i < numfwdprocs; i++)  {
             CREATE(fwdslv)
        }

        for (i=1; i < numbwdprocs; i++)  {
             CREATE(bwdslv)
        }
```

```
    /* process the new clauses */

    CLOCK(clk1);
    for (newclsctr = 0; newclsctr < numnewclses; newclsctr++)  {

        if (reuse_newcls_ind == 'n'  ||  newclsctr == 0)
           {
             nxt_newitem[0] = -1;
             token_type = ';';
             newclause[0].litptr[0] = NIL;
             newclause[0].nextcls = NIL;
             newclause[0].delind = '-';
             new_prev_pred_in = -1;
             i = 0;
             for (;;)  {
                 nxtlit = buildliteral (newclause, newitem, nxt_newitem);
                 if (nxt_newitem[0] > MAXNEWITEMS)
                     error ("main", "nxt_newitem[0]", "MAXNEWITEMS");
                 if (nxtlit == '?')
                    break; /* out of inner for loop */
                 newclause[0].litptr[i] = nxtlit;
                 if (nxtlit == NIL && token_type == ';')
                    break;
                 i++;
                 newclause[0].litptr[i] = NIL;
                 if (nxtlit != NIL  &&  new_prev_pred_in != -1)
                    newitem[new_prev_pred_in].right = nxtlit;
                 new_prev_pred_in = nxtlit;
             } /* end for */
             if (nxtlit == '?')
                break;  /* from outer for loop  */
             if (prtind == 'y')
                {
                  printf("\n\nNext new clause is: \n");
                  prtclses(newclause,newitem,0,1);  /* print new clause  */
```

```
                }
            } /* end if */

        n1 = 0;
        b1 = 0;
        fwd_occurred = 'n';
        bwd_occurred = 'n';
        if (fwd_bwd_parallel == 'n')
            forsubsum ();
        if (fwd_occurred == 'n' && nummacprocs == 2)
            {
              BARRIER(f1,nummacprocs)    /* start forward if it is held */
              backsubsum ();
              BARRIER(f2,nummacprocs)    /* wait for forward to finish */
            }
        if (bwd_occurred == 'y' && fwd_occurred == 'n')
            integrateclause ();
        else
            numfwdsub++;

    } /*end new clause loop */

    [illegible]
    tot_time = [illegible]

    pgmdone = 'y';
    BARRIER(f1,nummacprocs)       /* tell forward the pgm is done */

    [illegible]
    printf("\n The total time was %d \n", tot_time);
    printf("\n The number of clauses forward subsumed was %d \n", numfwdsub);
    printf("\n The number of clauses backward subsumed was %d \n", numbwdsub);
    [illegible]

    ****** terminate clauses ******
    [illegible]
```

```
PROGEND(bs)

return (0);
} /* end main */


/* Pforwardproc */

forwardproc ()
{

forsubsum ();

return (0);

} /* end forwardproc */


/* Pfwdslv */

fwdslv ()
{
int slave = 1;

fwd (slave);

return (0);

} /* end fwdslv */


/* Pbwdslv */

bwdslv ()
{
int slave = 1;
```

```
bwd (slave);

return (0);

} /* end bwdslv */



/* Pintegrateclause */
integrateclause ()
{

int i, nlit, olit, rc, lit_start, cls_start;

i = 0;
cls_start = nxt_olditem[0] + 1;
for (nlit = 0; nlit != NIL; nlit = newitem[nlit].right)  {
     lit_start = nxt_olditem[0] + 1;  /*save place where new lit will start */
     olit = copyterm (newitem, nlit, olditem, lit_start, 'l', MAXOLDITEMS);
     nxt_olditem[0] = olit;
     oldclause[curr_clause].litptr[i]  =  lit_start;
     i++;
     if (i > MAXLITPERCLS)
        error ("integrateclause", "i", "MAXLITPERCLS");
     oldclause[curr_clause].litptr[i]  =  NIL;
     rc = litexistchk (olditem, lit_start);   /*lit already in olditem? */
     if (rc == NIL)  /* lit did not previously exist */
        {
          if (old_prev_pred_in  !=  -1)
             olditem[old_prev_pred_in].right = lit_start;
          old_prev_pred_in = lit_start;
          rc = addtolitlist (lit_start, curr_clause, 'n' /*new*/);
          olditem[lit_start].litlistentry = rc;  /*build pred to litlist ptr*/
        }
     else  /* lit was already in the data structures  */
```

```
            {
              nxt_olditem[0] = lit_start;  /* remove this new copy from item  */
              oldclause[curr_clause].litptr[i-1] = rc;  /*pt cls to oldlit*/
              rc = addtolitlist (rc, curr_clause, 'o' /*old*/);
            }
      }
      buildfpalist (olditem, cls_start);
      if (prev_clause != -1)
         oldclause[prev_clause].nextcls = curr_clause;
      prev_clause = curr_clause;
      curr_clause++;
      oldclause[curr_clause].litptr[0] = NIL;
      oldclause[curr_clause].nextcls = NIL;
      oldclause[curr_clause].delind = '-';

      return (0);
      } /* end integrateclause */



      /* Pbuildliteral  */
      buildliteral(clause, item, nxtitem)
      struct clauses clause[];
      struct items item[];
      int nxtitem[];

      {
      int rc, curr_item;

      rc = getoken();
      if (rc == '?')
         return ('?');
      if (token_type == '-' || token_type == '(')
         {
           rc = getoken();
         }
```

```
   switch (token_type) {
        case ';' :
        case ')' :
          rc = NIL;  /* null */
          break;
        case 'p' :
        case 'f' :
          builditem(item, nxtitem);  /* build a predicate or function  */
          curr_item = nxtitem[0];
          item[curr_item].left = buildliteral(clause, item, nxtitem);
          if (item[curr_item].type == 'p')
             item[curr_item].right = NIL;
          else
             item[curr_item].right = buildliteral(clause, item, nxtitem);
          rc = curr_item;
          break;
        case 'c' :
        case 'v' :
          builditem(item, nxtitem);  /* build a constant or variable */
          curr_item = nxtitem[0];
          item[curr_item].right = buildliteral(clause, item, nxtitem);
          rc = curr_item;
          break;
        default :
          printf("\n invalid token_type returned *1* ");
          rc = EOF;
          break;
   } /* end switch */
   return (rc);
   } /* end buildliteral */

   /* Pbuilditem */
   builditem(item, nxtitem)    /*  build tree with root = next term in input */
   struct items item[];
   int nxtitem[];
```

```
{
nxtitem[0]++;
item[nxtitem[0]].type = token_type;
item[nxtitem[0]].pred_sign = pred_sign;
item[nxtitem[0]].id [0] = token[0];
item[nxtitem[0]].id [1] = token[1];
item[nxtitem[0]].litlistentry = NIL; /*null */
item[nxtitem[0]].left = NIL;   /*null */
item[nxtitem[0]].right = NIL;  /*null */
} /* end builditem */


/*Plitexistchk */
litexistchk (item, newlit)
struct items item[];
int newlit;

{
int oldlit, rc;

if (newlit == 0)  /*very first lit created */
   return (NIL);
oldlit = 0;
while (oldlit  !=  NIL)  {
   rc = litcompare (item, newlit, oldlit);
   if (rc == 1)  /*they are equal */
      break;
   oldlit = item[oldlit].right;
}
return (oldlit);

} /* end litexistchk */


/* Plitcompare */
litcompare (item, newlit, oldlit)
```

```
struct items item[];
int newlit, oldlit;

{
int rc, nright, oright, nleft, oleft;

rc = 1;
nright = item[newlit].right;
oright = item[oldlit].right;
nleft = item[newlit].left;
oleft = item[oldlit].left;
if (item[newlit].id[0] != item[oldlit].id[0]  ||
    item[newlit].id[1] != item[oldlit].id[1]  ||
    item[newlit].pred_sign != item[oldlit].pred_sign)
      return (0);  /* not equal */
if (nleft  !=  NIL  &&  oleft  !=  NIL)
   rc = litcompare (item, nleft, oleft);
if (rc == 1  &&  nright != NIL  &&  oright != NIL  &&  item[newlit].type != 'p')
    rc = litcompare (item, nright, oright);
return (rc);

} /*end litcompare */


/* Paddtolitlist */
addtolitlist (litnum, clsnum, old_new)
int litnum, clsnum;
char old_new;

{
int i, j;

if (old_new == 'n')  /* new literal */
   {
     litlist[new_lit].predptr = litnum;
     litlist[new_lit].clsptr[0] = clsnum;
```

```
          litlist[new_lit].clsptr[1] = NIL;
          new_lit++;
          if (new_lit > MAXLITS)
             error ("addtolitlist", "new_lit", "MAXLITS");
          return (new_lit - 1);
        }
     /*old literal appears in new clause */
     for (i=0; litlist[i].predptr != litnum; i++)  {;}  /* null stmt */
     for (j=0; litlist[i].clsptr[j] != NIL; j++)  {;}  /* null stmt */
     if (j >= MAXLITTOCLS)
        error ("addtolitlist", "j", "MAXLITTOCLS");
     litlist[i].clsptr[j] = clsnum;
     litlist[i].clsptr[j+1] = NIL;
     return(i);

     } /*end addtolitlist */




     /* Pbuildfpalist */
     buildfpalist (item, start_item)
     int start_item;
     struct items item[];

     {
     int i, argptr, argcnt;
     struct fpalists tempfpa[1];

     for (i=start_item; i != NIL; i = item[i].right)  {
          tempfpa[0].pred[0] = item[i].pred_sign;
          tempfpa[0].pred[1] = item[i].id[0];
          tempfpa[0].pred[2] = item[i].id[1];
          tempfpa[0].pred[3] = '\0';
          tempfpa[0].arg[0] = ' ';
          tempfpa[0].arg[1] = ' ';
```

```
            tempfpa[0].arg[2] = '\0';
            tempfpa[0].argnum = 0;
            argptr = item[i].left;  /* pt to 1st arg for this lit */
            do {
                    if (argptr != NIL)  /* if there is another arg to handle  */
                      {
                         tempfpa[0].argnum++;
                         tempfpa[0].arg[0] = item[argptr].id[0];
                         tempfpa[0].arg[1] = item[argptr].id[1];
                         argptr = item[argptr].right;
                      }
                    addtofpalist (item[i].litlistentry, tempfpa);
               } while (argptr != NIL);
        }
        return (0);

        } /*end buildfpalist */



        /* Paddtofpalist */
        addtofpalist (newlitptr, tempfpa)

        int newlitptr;
        struct fpalists tempfpa[];

        {
        int i, j, hashval;

        hashval = hashfpa (tempfpa[0].pred, tempfpa[0].argnum);
        for (i=hashval; i < new_fpa[hashval]; i++)  {
            if ((strcmp(fpalist[i].pred, tempfpa[0].pred) == 0)  &&
                 (strcmp(fpalist[i].arg, tempfpa[0].arg) == 0)  &&
               fpalist[i].argnum == tempfpa[0].argnum)
                 {
                    for (j=0; fpalist[i].litlistptr[j] != NIL; j++)  {;} /*null stmt*/
```

```
                if (j >= MAXFPATOLIT)
                   error ("addtofpalist", "j", "MAXFPATOLIT");
                fpalist[i].litlistptr[j] = newlitptr;
                fpalist[i].litlistptr[j+1] = NIL;
                return (0);
             }
     }
     i=new_fpa[hashval];
     strcpy(fpalist[i].pred, tempfpa[0].pred);
     strcpy(fpalist[i].arg, tempfpa[0].arg);
     fpalist[i].argnum = tempfpa[0].argnum;
     fpalist[i].litlistptr[0] = newlitptr;
     fpalist[i].litlistptr[1] = NIL;
     new_fpa[hashval]++;
     if (new_fpa[hashval] >= new_fpa[hashval+1])
        error ("addtofpalist", "new_fpa[hashval]", "new_fpa[hashval+1]");

     return (0);

     } /* end addtofpalist */



     /* Phashfpa */
     hashfpa (pred, argnum)
     int argnum;
     char pred[];

     {
     return ((pred[0] + pred[1] + pred[2] + argnum) % FPAMODVAL);

     }  /* end hashfpa */


     /* Pprtclses */
     prtclses (clses, item, cc, howmany)
```

```
int cc, /* current clause */
    howmany;
struct clauses clses[];
struct items item[];

{
int cl;  /* current lit  */

printf("\n");

/* the following for-stmt skips leading 'deleted' clause headers */
for ( ; clses[cc].delind == 'd'; cc = clses[cc].nextcls)
     {;}  /*null stmt*/
while (cc  !=  NIL  &&  howmany > 0)  {
    cl = 0;
    while (clses[cc].litptr[cl]  !=  NIL)  {
        if (cl  !=  0)
           printf ("! ");
        prtlit (item, clses[cc].litptr[cl]);
        cl++;
    }
    if (cl != 0)
        printf ("; \n");
    cc = clses[cc].nextcls;
    /* the following for-stmt skips 'deleted' clause headers */
    for ( ; clses[cc].delind == 'd'; cc = clses[cc].nextcls)
         {;}  /*null stmt*/
    howmany--;
}

} /* end prtclses */


/* Pprtlit */
prtlit(lit, cr)   /* print selected literals */
int cr; /* current root */
```

```
struct items lit[];

{
int cr_left,
    cr_right;
char cr_type;

cr_type = lit[cr].type;
cr_left = lit[cr].left;
cr_right = lit[cr].right;

if (lit[cr].pred_sign  ==  '-')
   putchar('-');
printf("%1c%1c",lit[cr].id[0],lit[cr].id[1]);
if (cr_type == 'p'  ||  cr_type == 'f')
   printf("( ");
else
   putchar(' ');
if (cr_left != NIL)  /* not null */
   prtlit (lit, cr_left);  /* visit left subtree */
if (cr_type == 'p'  ||  cr_type == 'f')
   printf(") ");
if (cr_right != NIL  &&  cr_type != 'p')
   prtlit(lit, cr_right);    /* visit right subtree */
return (0);
} /* end prtlit */



/* Pgetoken  */
getoken()
{
int c;

pred_sign = '+';
prev_token = token_type;
```

```
        c = getnextchar();
        if (c == '?')
           return ('?');
        if (c == '-' && (prev_token == '|' || prev_token == ';' ))
           {
             pred_sign = '-';
             c = getnextchar();
           }
        if (c == '|'  ||  c == ';'  ||  c == '('  ||  c == ')')
           {
             token_type = c;
             token[0] = c;
             token[1] = ' ';
             pred_sign = '+';
             return (0);
           }
        if (!isalpha(c))  /* not alpha*/
           return (1);
        token[0] = c;
        c = getnextchar();
        if (!isdigit(c))  /* not digit*/
           return(2);
        token[1] = c;
        c = getnextchar();  /* peek at the next char */
        ungetc (c,stdin);         /* and then put it back  */
        if (c == '(')
           {
             if (prev_token == '|'  ||  prev_token == ';')
                token_type = 'p';
             else
                token_type = 'f';
           }
        else
           {
             if (token[0] >= 's'  &&  token[0]  <=  'z')
                token_type = 'v';
```

```
        else
            token_type = 'c';
        }
    return(0);

    } /* end getoken */


    /* Pgetnextchar */
    getnextchar ()
    {
    int c;
    while ((c = getc (stdin))  ==  ' '  ||
                        c  ==  '\n' ||
                        c  ==  '\r' ||
                        c  ==  '\t' ||
                        c  ==  ',')
          ;  /* null stmt  */
    return (c);
    } /* end getnextchar */



    /***************  EXTERNAL DEFNS   *********************/

    int fclashlit;   /* externs for fwd routines  */
    struct substitution fsubst[SUBSIZE];

    /*****************************************************/


    /* Pforsubsum */    /*does an old cls subsume a new one ?  */
    forsubsum ()

    {
    int i, j, rc, urc, master;
```

```
int nlptr, nxtopndlit, litstoclash[MAXLITS];
struct items dlit[LITSIZE];

for (;;)  {
   BARRIER(f1,nummacprocs)  /* let bwd and fwd start together */
   if (pgmdone == 'y')
      break;
   i = 0;
   rc = 0;
   master = 0;
   nlptr = newclause[n1].litptr[i];  /* pt to 1st lit in newitem */
   while (rc == 0  &&  nlptr != NIL)  {
      getclashlits ('f',newitem,nlptr,litstoclash);
      for (j=0; rc == 0  &&  (fclashlit = litstoclash[j]) != NIL; j++)  {
          fsubst[0].termptr = NIL;
          nxtopndlit = copyterm (olditem,litlist[fclashlit].predptr,
                                       dlit,0,'l',LITSIZE);
          renamevars (dlit,0);
          urc = unify(dlit,0,newitem,nlptr,fsubst,nxtopndlit);
          if (urc == 1)  /* the 2 lits unify */
             {
                FWDPROBST
                rc = fwd (master);
             } /*end if*/
      } /*end for */
      i++;
      nlptr = newclause[n1].litptr[i];
   } /* end while */
   BARRIER(f2,nummacprocs)   /* forward returns here in uniproc mode */
   if (fwd_bwd_parallel == 'n')
      break;
} /* end forever */

return (rc);

} /* end forsubsum */
```

```
/* Pfwd */
fwd (who)

int who;
{

int clashcls, arc, rc, k;
struct clauses tempcls[1];

rc = 0;
for (;;)  {
     ASKFOR(fs,arc,numfwdprocs,FWDGETPROB(k,arc),FWDRESET)
     if (arc == -1  ||  (arc != 0  &&  who == 0))
        break;
     if (arc != 0)
        continue;
     clashcls = litlist[fclashlit].clsptr[k];
     if (oldclause[clashcls].delind == 'd')
        continue;
     skiplit (oldclause, clashcls, litlist[fclashlit].predptr, tempcls);
     if (tempcls[0].litptr[0] == NIL)
        rc = 1;
     else
        rc = subsum(tempcls,0,olditem,0,newclause,n1,newitem,fsubst);
     if (rc == 1)  /* old subsumes new */
        {
           PROBEND(fs,2)   /* tell fwds that this problem is solved */
           LOCK(1)
           fwd_occurred = 'y';
           UNLOCK(1)
        }
} /* end forever */

if (fwd_occurred == 'y')
```

```
            rc = 1;

        return (rc);
        } /* end fwd */




        /*********   EXTERNAL  DEFNS   ********************************/

        int bclashlit;     /* externs for bwd routines */
        struct substitution bsubst[SUBSIZE];

        /************************************************************/


        /* Pbacksubsum */     /*does a new cls subsume an old one ?  */
        backsubsum ()

        {
        int j, k, rc, urc, master;
        int nlptr, clashcls, nxtopndlit, litstoclash[MAXLITS];
        struct items dlit[LITSIZE];

        rc = 0;
        master = 0;
        nlptr = newclause[n1].litptr[0];  /* pt to 1st lit in newitem */
        getclashlits ('b',newitem,nlptr,litstoclash);
        for (j=0; (bclashlit = litstoclash[j]) != NIL; j++)  {
             bsubst[0].termptr = NIL;
             nxtopndlit = copyterm (newitem,nlptr,dlit,0,'l',LITSIZE);
             renamevars (dlit,0);
             urc = unify(dlit,0,olditem,litlist[bclashlit].predptr,
                            bsubst,nxtopndlit);
             if (urc == 1)  /* the 2 lits unify */
                {
```

```
                BWDPROBST
                rc = bwd (master);
             } /*end if*/
      } /*end for */

      return (rc);

      } /* end backsubsum */


      /* Pbwd */
      bwd (who)

      int who;
      {

      int clashcls, arc, rc, k;

      rc = 0;
      for (;;)  {
          ASKFOR(bs,arc,numbwdprocs,BWDGETPROB(k,arc),BWDRESET)
          if (arc == -1   ||   (arc != 0  &&  who == 0))
             break;
          if (arc != 0)
             continue;
          clashcls = litlist[bclashlit].clsptr[k];
          if (oldclause[clashcls].delind == 'd')
             continue;
          if (newclause[n1].litptr[1] == NIL)
             rc = 1;
          else
             rc = subsum(newclause,n1,newitem,1,oldclause,clashcls,olditem,bsubst);
          if (rc == 1)  /*new cls subsumes an old one */
             {
                PROBEND(fs,2)   /*tell fwd subsump chks to stop*/
                LOCK(1)
```

```
                numbwdsub++;
                bwd_occurred = 'y';
                oldclause[clashcls].delind = 'd';
                UNLOCK(1)
             } /*end if*/
    }  /*end forever */

    if (bwd_occurred == 'y')
       rc = 1;

    return (rc);

    } /* end bwd */



    /* Psubsum */
    subsum (ds, dsc, dsitem, d1, rs, rsc, rsitem, rcvd_subst)

    int dsc, rsc, d1;
    struct clauses ds[], rs[];
    struct items dsitem[], rsitem[];
    struct substitution rcvd_subst[];

    {
    int i, rc, dsptr, d2, r1, rsptr, nxtopndlit;
    struct items dlit[LITSIZE];
    struct substitution subst[SUBSIZE];

    /* Initially, rcvd_subst is empty from main, dsc points to
       the 'subsuming' clause, and rsc points to the 'subsumed'
       clause.  We want to see if ds subsumes rs.
    */

    r1 = 0;
```

```
    dsptr = ds[dsc].litptr[d1];
    rsptr = rs[rsc].litptr[r1];
    rc = 0;  /* not subsumed  */

    while (rc == 0  &&  rsptr != NIL /*null*/)
      {
        i = 0;
        do {
                subst[i].var[0] = rcvd_subst[i].var[0];
                subst[i].var[1] = rcvd_subst[i].var[1];
                subst[i].termptr   = rcvd_subst[i].termptr;
           }  while (subst [i++].termptr  != NIL); /*null*/
        nxtopndlit = copyterm (dsitem, dsptr, dlit, 0, 'l', LITSIZE);
        if (nxtopndlit == NIL)
           return (0);
        renamevars (dlit, 0);
        rc = unify (dlit,0,rsitem,rsptr,subst,nxtopndlit);  /*chgs dlit and subst */
        if (rc == 1)  /* they unify */
           {
             d2 = d1 + 1;
             if (ds[dsc].litptr[d2]  ==  NIL)  /*null*/
                rc = 1;
             else
                {
                  rc = subsum (ds,dsc,dsitem,d2,rs,rsc,rsitem,subst);
                }
           }
        r1++;
        rsptr = rs[rsc].litptr[r1];
      }
    return (rc);

    } /* end subsum */


    /* Punify  */
```

```
unify (ulit, u1, rlit, r1, subst, nxtopnulit)

int u1, r1, nxtopnulit;
struct items ulit[], rlit[];
struct substitution subst[];

{
int i, rc, uleft, rleft, uright, rright;
char copy_type;

nxtopnulit = substitute (ulit, u1, nxtopnulit, rlit, subst);
uleft  = ulit[u1].left;
uright = ulit[u1].right;
rleft  = rlit[r1].left;
rright = rlit[r1].right;
if (ulit[u1].pred_sign == rlit[r1].pred_sign  &&
     ulit[u1].id[0] == rlit[r1].id[0]  &&
     ulit[u1].id[1] == rlit[r1].id[1])
       {
         if (uleft == NIL  &&  uright == NIL)  /* both null */
            {
              if (rleft == NIL  &&  (rright == NIL  ||  rlit[r1].type == 'p'))
                 return (1);  /* they unify */
            }
         if (uleft != NIL  &&  rleft != NIL) /*null*/
            {
              rc = unify (ulit, uleft, rlit, rleft, subst, nxtopnulit);
              if (rc == 0)
                 return (0);
            }
         else
              if ((uleft == NIL  &&  rleft != NIL)   ||    /* one null and  */
                   (uleft != NIL  &&  rleft == NIL))        /* not the other */
                 {
                   return (0);
                 }
```

```
            }
      else
            {
              if (ulit[u1].type != 'v'  ||  !isdigit(ulit[u1].id[0])) /*nonsubst var*/
                 return(0);
              for (i = 0; subst[i].termptr != NIL; i++)  {;}    /* null stmt */
              subst[i].var[0] = ulit[u1].id[0];
              subst[i].var[1] = ulit[u1].id[1];
              subst[i].termptr = r1;  /* ptr into rlit (in items) */
              subst[++i].termptr = NIL; /* null */
              if (i > SUBSIZE)
                 error ("unify", "i", "SUBSIZE");
              if (nxtopnulit == NIL)
                 return(0);
              rc = 1;
            }
      if (uright != NIL  &&  rright != NIL)
          {
            rc = unify (ulit, uright, rlit, rright, subst, nxtopnulit);
          }
      return (rc);

      } /* end unify  */



      /* Pskiplit */
      skiplit (ocls, oc1, lit_to_skip, tempcls)
      struct clauses ocls[], tempcls[];
      int oc1, lit_to_skip;

      {
      int i, j, olitptr;

      j =0;
      for (i=0; (olitptr = ocls[oc1].litptr[i]) != NIL; i++)  {
```

```
        if (olitptr != lit_to_skip)
            tempcls[0].litptr[j++] = olitptr;
}

tempcls[0].litptr[j] = NIL;
return (0);
} /* end skiplit */



/****************** EXTERNAL STRUCT DEFINITION ***************/
struct tclash {
                    int llptr;
                    int refcnt;
                };
/***************************************************************/


/* Pgetclashlits */
getclashlits (fwd_bwd, nitem, nlptr, litstoclash)

char fwd_bwd;
int nlptr, litstoclash[];
struct items nitem[];

{
int i, j, k, argptr, next_tempclash[1],
     fpamatch[(FPAMODVAL+3)*FPASPERHASHV];
struct tclash tempclash[MAXLITS];
struct fpalists tempfpa[1];

next_tempclash[0] = 0;
tempfpa[0].pred[0] = nitem[nlptr].pred_sign;
tempfpa[0].pred[1] = nitem[nlptr].id[0];
tempfpa[0].pred[2] = nitem[nlptr].id[1];
tempfpa[0].pred[3] = '\0';
```

```
    tempfpa[0].arg[0] = ' ';
    tempfpa[0].arg[1] = ' ';
    tempfpa[0].arg[2] = '\0';
    tempfpa[0].argnum = 0;
    argptr = nitem[nlptr].left;  /* pt to 1st arg for this lit */
    do {
          if (argptr != NIL)
             {
                tempfpa[0].argnum++;
                tempfpa[0].arg[0] = nitem[argptr].id[0];
                tempfpa[0].arg[1] = nitem[argptr].id[1];
                argptr = nitem[argptr].right;
             }
          fpamatchk (fwd_bwd, fpamatch, tempfpa);
          for (i=0; fpamatch[i] != NIL; i++)  {

              for (j=0; (k=fpalist[fpamatch[i]].litlistptr[j]) != NIL; j++)  {
                  addtotempclash (k, next_tempclash, tempclash);
              }
          }
    } while (argptr != NIL);

    i=0;
    for (j=0; j < next_tempclash[0]; j++)  {
         if ((tempclash[j].refcnt == tempfpa[0].argnum)  ||
             (tempclash[j].refcnt == 1  &&  tempfpa[0].argnum == 0)) /*proposition*/
            {
              litstoclash[i] = tempclash[j].llptr;
              i++;
              if (i > MAXLITS)
                 error ("getclashlits", "i", "MAXLITS");
            }
    }

    litstoclash[i] = NIL;
    return (0);
```

```
} /*end getclashlits */


/* Pfpamatchk */
fpamatchk (fwd_bwd, fpamatch, tempfpa)

char fwd_bwd;
int fpamatch[];
struct fpalists tempfpa[];

{
int i, j, hashval;

fpamatch[0] = NIL;

j=0;
hashval = hashfpa (tempfpa[0].pred, tempfpa[0].argnum);
for (i=hashval; i < new_fpa[hashval]; i++)  {
     if (tempfpa[0].argnum == fpalist[i].argnum  &&
          strcmp(tempfpa[0].pred,fpalist[i].pred) == 0)
        {
            if (strcmp(tempfpa[0].arg,fpalist[i].arg) == 0)
               {
                   fpamatch[j] = i;
                   j++;
                   fpamatch[j] = NIL;
               }
            else
               {
                   if ((fwd_bwd == 'f'  &&  isavariable(fpalist[i].arg[0]))  ||
```

```
                              (fwd_bwd == 'b'  &&  isavariable(tempfpa[0].arg[0])))
                             {
                                fpamatch[j] = i;
                                j++;
                                fpamatch[j] = NIL;
                             }
                     }
             }
     }

     return (0);

     } /* end fpamatchk */

     /* Pisavariable */
     isavariable (c)

     char c;

     {
     if (c >= 's'  &&  c <= 'z')
        return (1);
     return (0);

     } /* end isavariable */


     /* Paddtotempclash */
     addtotempclash (litlistptr, next_tempclash, tempclash)

     int litlistptr, next_tempclash[];
     struct tclash tempclash[];

     {
     int i;
```

```
for (i=0; i < next_tempclash[0] && tempclash[i].llptr != litlistptr; i++)
{;}  /* null stmt */

if (i == next_tempclash[0])  /*litlistptr was not there */
   {
     tempclash[next_tempclash[0]].llptr = litlistptr;
     tempclash[next_tempclash[0]].refcnt = 1;
     next_tempclash[0]++;
     if (next_tempclash[0] > MAXLITS)
        error ("addtotempclash", "next_tempclash", "MAXLITS");
   }
else
   {
      tempclash[i].refcnt++;
   }
return (0);

}  /* end addtotempclash */



/* Pcopyterm */
copyterm (from, fr1, to, t1, copy_type, last_avail_item)

/* This routine will copy any item type-of-object beginning at from[fr1]
   to a location beginning at to[t1].  If copy_type = 'l' (left) it will
   copy only the item and its left side; r - only the item and its right,
   b - the item and both sides, any other value - copies only the item.
   *** Note that litlistentrys are copied for predicates even though the
   litlist entries do not point to these copies, only to the originals. ***
*/

int fr1, t1, last_avail_item;
char copy_type;
struct items from[], to[];
```

```
{
int t2;

if (t1 >= last_avail_item)
   error ("copyterm", "t1", "last_avail_item");
to[t1].type = from[fr1].type;
to[t1].pred_sign = from[fr1].pred_sign;
to[t1].id[0] = from[fr1].id[0];
to[t1].id[1] = from[fr1].id[1];
to[t1].litlistentry = from[fr1].litlistentry;
t2 = t1 + 1;
if (from[fr1].left != NIL /*null*/   &&
     (copy_type == 'l'  ||  copy_type == 'b'))
      {
        to[t1].left = t2;
        t2 = copyterm (from,from[fr1].left,to,t2,'b',last_avail_item);
        if (t2 == NIL)
           return (NIL);
      }
else
      to[t1].left = NIL;  /*null*/
if (from[fr1].right != NIL /*null*/   &&
     (copy_type == 'r'  ||  copy_type == 'b'))
      {
        to[t1].right = t2;
        t2 = copyterm (from,from[fr1].right,to,t2,'b',last_avail_item);
        if (t2 == NIL)
           return (NIL);
      }
else
      to[t1].right = NIL; /*null*/
return (t2);

} /* end copyterm */
```

```

/* Prenamevars */
renamevars (dlit, d1)

struct items dlit[];
int d1;

{
int dleft, dright;

dleft  = dlit[d1].left;
dright = dlit[d1].right;
if (dlit[d1].type == 'v')
   dlit[d1].id[0]  -=  66;  /* change 's'-'z'  to  '1'-'8' */
if (dleft != NIL) /*null*/
   renamevars (dlit, dleft);
if (dright != NIL) /*null*/
   renamevars (dlit, dright);
return (0);

} /*end renamevars */



/* Psubstitute */
substitute (to, t1, nxtopntolit, from, subst)

int t1, nxtopntolit;
struct items to[], from[];
struct substitution subst[];

/* Substitute for variables in a literal which are to be replaced in
   a literal which is being unified with another.  Only literals which
   still have their renamed values (by renamevars) will be substituted for.
*/

```

```
{
int i, fr1;

for (i = 0; subst[i].termptr != NIL; i++)
     {
       if (subst[i].var[0] == to[t1].id[0]  &&  subst[i].var[1] == to[t1].id[1])
          break;
     }
 if (subst[i].termptr  !=  NIL)
    {
      fr1 = subst[i].termptr;
      to[t1].type = from[fr1].type;
      to[t1].id[0] = from[fr1].id[0];
      to[t1].id[1] = from[fr1].id[1];
      if (from[fr1].left != NIL)
         {
           to[t1].left = nxtopntolit;
           nxtopntolit = copyterm (from, from[fr1].left, to, nxtopntolit, 'b',
                                        LITSIZE);
         }
    }
 return (nxtopntolit);

 } /* end substitute */




 /* Perror */
 error (procname, indexname, maxvalname)
 char procname[], indexname[], maxvalname[];

 {

write (STDERR, "\n\n**** early exit - table overflow in procedure \n  ", 50);
write (STDERR, procname, strlen(procname));
```

```
write (STDERR, "\n", 1);
write (STDERR, indexname, strlen(indexname));
write (STDERR, "\n", 1);
write (STDERR, maxvalname, strlen(maxvalname));
exit (1);

} /* end error */
```