TECHNICAL REPORT NO. 262

Compiling and Executing Theorems Efficiently as Logic Programs

by

Jonathan W. Mills

September 1988

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Compiling and Executing Theorems Efficiently as Logic Programs

by

Jonathan W. Mills
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 262

Compiling and Executing Theorems
Efficiently as Logic Programs

by

Jonathan W. Mills

September, 1988

Compiling and Executing Theorems Efficiently as Logic Programs

Jonathan Wayne Mills

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101
e-mail: jwmills@iuvax.cs.indiana.edu

Abstract

A collection of axioms and the negation of a theorem expressed in first-order logic can be transformed into families of Horn clauses, and executed as a logic program. Each Horn clause family represents a procedural attribute of a non-Horn clause, such as an inference rule. The mechanism by which non-Horn clauses are compiled is described and an example theorem from lattice theory is presented. Although a logic program can derive proofs faster than an interpreted theorem proving system, the use of unification as a clause selection method is a bottleneck.

1. Introduction

Resolution-based theorem provers are typically implemented as interpreters. They obtain proofs by manipulating data structures used to represent the axioms and the negation of the theorem (collectively called the theorem in this paper). Procedures that manipulate the theorem's data structures include resolution and other inference rules, indexing (clause selection), subsumption, and integration of generated clauses. The separation of data and procedure leads to limitations in the performance of interpretive theorem provers: inference rules based on a general purpose unifier are too complex to be efficient. But if these general procedures are applied to the theorem, instances of these procedures can be obtained for each clause, i.e., the theorem can be compiled. Each instance contains clause-specific infomation which can be used to optimize the procedure. For example, in an inference rule the strength of the underlying unification algorithm can be reduced. As a result, the time needed by the compiled theorem to find a proof is decreased, and is overall much less than the time required by an interpretive theorem prover.

Using this approach, each clause in the theorem is treated as a non-procedural statement composed of procedural attributes which may be subsumption, demodulation, or one or more inference rules.

These attributes are compiled into Prolog clauses, which may be further compiled into an extended version of Warren's abstract Prolog machine. The extended Warren machine contains instructions to perform unification with an occurs check, and matching without binding variables. A proof may be obtained by combining the compiled theorem with a short control module, and executing the resulting program. The purpose of the control module is to select any of a variety of search procedures in the proof specification, rather than defaulting to the depth-first search implicit in Prolog.

The initial work with indicates that substantial performance improvements can be obtained. Using a Warren machine emulator running on a VAX 11/780, a compiled theorem proved Sam's Lemma 15 times faster than a resolution-based theorem prover (ITP) running on the same machine. Further evidence of improvement was provided when ITP was transported to a Cray XMP: the compiled theorem, still running on a VAX 11/780, obtained the same proof in half the time needed by ITP running on the Cray XMP. Recent work by McCune (1988) has reduced the disparity between compiled and interpretive theorem provers, but it will be shown that the bottleneck in compiled theorem provers is due to the use of unification and backtracking as a general-purpose indexing mechanism, which is very inefficient. Research is in progress to improve the indexing mechanism for compiled theorems.

This research was initiated at Argonne National Laboratory in 1985 during a study of Warren's abstract Prolog machine. A number of implementations and partial implementations were programmed (Mills 1986). At the same time Ross Overbeek formulated two concepts that led directly to this fusion of theorem proving and logic programming. The first concept was the extension of the Warren machine instruction set to include instructions that performed unification with an occurs check,[1] and matching without binding. The second concept was the notion of encoding a hyperresolution inference rule into these Warren machine instructions. From these two notions I generalized paradigm of compiling a non-Horn clause into multiple procedural attributes.

There are at least two other approaches that associate theorem proving and logic programming. The first, and most straightforward, is the implementation of a classical theorem prover in Prolog. This approach has been taken by Bundy (1982), but corresponds to an interpretive rather than a compiled theorem prover. The second is the extension of Prolog to include an occurs check, a complete inference system (model elimination), and modification of Prolog's unbounded depth-

1 See (Beer 1988) for a way to add the occurs check in an elegant way. See (Plaisted 1984) for a way to preprocess Prolog programs to omit occurs checking.

first search strategy. This approach is taken in the Prolog Technology Theorem Prover (Stickel 1984) and PARTHENON, a parallel theorem prover for non-Horn clauses (Clark 1988).

This approach differs from these two in the following ways:

1. Clauses corresponding to axioms and the negation of the theorem are compiled, not retained as data structures,

2. The proof specification is modified dynamically during its execution as clauses corresponding to resolvents are generated, retained, and possibly subsume other clauses,

3. Inference rules comprising a complete inference system are treated as attributes of a clause corresponding to an axiom or the negation of the theorem, and thus become part of the compiled theorem,

4. Bounded depth-first, breadth-first, A*, or other search strategies are superimposed on Prolog's unbounded depth-first search strategy.

The sequential implementation has not yet not been compared to either the PTTP or PARTHENON, but its performance is expected to be competitive even though PARTHENON is parallel.[2]

2. ITP: A RESOLUTION-BASED THEOREM PROVER

ITP/LMA is a resolution-based theorem prover [Lusk & Overbeek]. It implements a numerous and still-growing family of inference and equality reasoning rules, including a wide variety of resolution rules, demodulation, and paramodulation. Any inference rule may be selected and combined with others during the proof. ITP/LMA has been used to prove fault-tolerance of computing systems [Klaijsch], to study synthesis of organic molecules [Erlich], and to solve a variety of logic puzzles, including those of Carroll and Smullyan [Lusk & Overbeek]. ITP/LMA is an interpretive theorem prover, retaining clauses as data structures, and implementing procedures to perform inference and equality reasoning, demodulation, and subsumption. A weighting mechanism is built in to the control structure that can be used to implement an A*-like search.

---

2 Overbeek (1988) suggests that this is due to the lack of subsumption in PARTHENON.

3. Partitioning the clause space

Clauses corresponding to the axioms of a theorem are typically placed in the axiom list. The clauses corresponding to the negation of the theorem are placed in the set of support. Clauses that have participated in an inference cycle are kept in a "have-been-given" list. As the theorem prover runs, clauses will be removed from the set of support, resolved against other clauses in the axiom and have-been-given lists, and when no more resolutions are possible, placed on the have-been-given list. New clauses (i.e., resolvents) that are generated are checked against all clauses in the axiom, set of support, and have-been-given lists. If the clause is an instance of any other clause it is discarded (i.e., forward subsumed). If it is not, then all clauses are checked to determine if they are less general instances of the generated clause. Should any be found that are, they are deleted (i.e., backward subsumed). Resolvents that are not forward subsumed are placed on the set of support.

The attempt to find a proof continues until either a null clause is generated, meaning that a proof was found, or until either no more clauses remain in the set of support, or the limit on the number of clauses to be selected from the set of support has been reached.

When one of the second pair of alternatives occurs, it is the responsibility of the user of ITP/LMA to examine the output produced, and from it select a strategy for another proof attempt. The new strategy is usually implemented either by changing one of the options, adjusting the weighting mechanism in use, or selecting one or more new inference rules. Occasionally a user may develop and implement a rule not previously available on the system, but this is not common practice.

4. Compiling axioms

4.1 Context-dependent negation

Negation as failure is an adequate model for Prolog computation. This is so since Prolog is exploring a "proof tree" or world model in which failure has a direct analogy to the truth of the supposition being investigated.

However, to impose the same strategy on the theorem prover is to severely limit the design. Terms that are negated must be represented to perform unification; using negation as failure precludes manipulation of negated terms. Instead, the concept of negation will be distinguished from the

concept of a negative literal. Having done this, we can then consider the meaning of this negative literal. The quality of negativeness can be considered as oppositeness. This property can then be embodied within the predicate whose negation is required. We may represent this property by prefixing some symbol to predicate. Since we are using Prolog, a prefix of "n" is appropriate, although we might consider using "not_" instead. However, if we use this representation throughout the theorem prover, we force ourselves to use a clumsy form of resolution. Instead, since a convention has been established, we may introduce another property, context-dependent negation, for representing literals of either sign which we wish to resolve against literals of either sign. A context-dependent negative literal is one whose sign as indicated by the predicate bears the opposite meaning, e.g., the context-dependent negative literal $-p$ is represented in a program by p.

4.2 Resolution with context-dependent predicates

Resolution as defined by Robinson involved finding predicates of opposite sign, then finding a substitution for each term of each predicate so that they match (i.e., unify). This substitution is called a most general unifier. If the process of matching could be performed against the predicates of the clause, as well as against each predicate's terms, then it would be possible to conduct resolution entirely by unification. This is in fact what is done, but, since the meaning of each predicate will vary based on its usage, we must be careful to define which predicates are context-dependent, the interactions between these and other predicates, and adhere to these conventions or suffer erroneous compiled theorem behavior.

The following example will show how context-dependent negative literals are used to represent theorems. Let us name three functional areas where we may place clauses in the course of developing a proof:

1. The set of support, containing clauses from which we will make a selection for use in resolution,

2. The inference rules, containing clauses which meet our definition of an inference rule, even if they are already in the set of support, and

3. The supporting clauses, containing those clauses that either were not initially present in the set of support or the inference rules, or which were selected from the set of support after being used for resolution.

Clauses from the set of support will be serve as a "baseline" to define context dependency. Within the set of support, a positive predicate such as "p", and a negative predicate, such as "np", retain their original meaning.

Clauses in the set of support may resolve (or "clash") against clauses that are inference rules. The resulting clause or clauses are placed back into the set of support. Thus, for our example where we use hyperresolution as an inference rule, context-dependency gives us the following clause formats:

Set of support: ---------------

p q r ns nt

Inference rules: ----------------

p | q | r meaning np | nq | r

q | r | s meaning nq | nr | s

Because the supporting clauses participate only with the inference rules during resolution, they retain their original signs. Also notice that the consequent term of the hyperresolution inference rule remains unaffected. This is because we shall place this rule into the set of support if it is accepted.

4.3 ENCODING HYPERRESOLUTION

Hyperresolution is an efficient refinement of binary resolution (Chang and Lee 1973). It correseponds to the form of human reasoning most often used in expert systems, where multiple conditions must be satisfied to cause the consequent condition to be valid.

An example of a hyperresolution inference rule might be:

(p & q & r) --> s

which would be expressed in clause notation as either:

-p V -q V -r V s

or:

-p | -q | -r | s

When hyperresolution is used with non-unit clauses, it is possible to produce a consequent composed of the clause consequent, and "satellite" literals, or those literals in a clause remaining after a single literal produced a succesful resolution with one of the negative terms of the resolvent.

In the example, we shall treat hyperresolution only with unit clauses. The meaning of the consequent from a purely procedural framework is, "If we can perform resolutions against the negative literals, then the consequent should be added to the database". But, remembering that we desire inference rules to be "pure", i.e., to contain no procedural components, we must restrict the action that can be taken by the inference rule if it succeeds. We also note that it is possible for the unit clause which starts the resolution with the inference rule to match any one of the negative literals. Thus we would like some selection process to lead us quickly to that literal, if it exists.

Given these constraints, a hyperresolution inference rule takes the following general form:

```
ir( p, s ) :- sc(q), sc(r).
ir( q, s ) :- sc(p), sc(r).
ir( r, s ) :- sc(q), sc(p).
```

Context dependency gives the meaning of -p, -q and -r to p, q, and r respectively. The inference rule is denoted by the predicate "ir", thus placing it into a distinct subspace of the internal database. After resolution against the literal extracted from the set of support, the remaining literals must be resolved against. This is done by the "sc" predicates, which invoke a search of the support clause space. If the head of the clause succeeded, and the "sc" clauses succeed, then the consequent is returned to the control section.

Notice that the consequent is returned as a data structure, rather than asserted. We wish to delay asserting the consequent, since the majority of generated consequents will be discarded by subsumption. Occurs checking can be immediately limited to variables and terms of the non-consequent in the head of the inference rule. The invocations of the support clauses from within the inference rule do not need to perform occurs checking. However, the "sc" clauses themselves do need to perform occurs checks.

Even where occurs check may be necessary, they can be limited by analysis of the predicates variables and terms. For example:

p(X,Y,Z)

needs no occurs check. Each variable is distinct.

However, either of:

p(X,Y,X)

or

p(X, f(X))

will need an occurs check, but only on the second occurrence of the variable X. Since this can be determined during compilation, it is possible to keep a compiled theorem logically complete without burdening the Prolog language with a non-specific occurs check.

5. Compiling the Negation of a Theorem

Test for it in the control loop.

6. Controlling Execution of the Theorem

A significant problem in the design of a compiled theorem prover based on Prolog exists because Prolog searches for alternative solutions to a problem by backtracking. Thus, a depth-first search where intermediate solutions are discarded (if not asserted as facts) forces a procedural outlook into the design. One solution to this problem is to make inference and subsumption operations "pure", in that satisfying that portion of a clause concerned with inference or subsumption either succeeded or failed, and possibly returned an instantiated variable. No further control operations were attempted within these Prolog clauses.

Control operations, such as the search strategy employed by the compiled theorem, manipulation of the generated and retained clauses, and invocation of inference and subsumption clauses, were collected into a single module. This module was virtually independent of the rest of the compiled theorem, and constituted only a small number of the clauses initially forming the theorem.

6.1 THE CONTROL ALGORITHM

The algorithm for a resolution theorem prover similar to ITP/LMA is shown below:

INIT: Read in axiom clauses.
Integrate axioms into formula database.
Read in set of support clauses.
Integrate set of support clauses into formula database.
Develop a list of clauses that can participate in resolution.

LOOP: Select a clause from the set of support.
Generate all hyper-resolvents between it and the axioms and have been given list.
Forward subsumption (resolvent discarded).
Backward subsumption against set of support, axioms, have been givens (clauses discarded, resolvent integrated into set of support).
Move selected clause to have been given list.
If null clause generated then END1.
If no more clauses in set of support then END2.
Goto LOOP.

END1: Proof complete. Stop.

END2: No proof found. Stop.

6.2 CLAUSE EXECUTION.

The use of Prolog's depth-first search strategy was limited to the procedural handling of a clause to be clashed. Each set of Prolog clauses for inference and subsumption were considered as nodes in the proof tree. The tree was repeatedly traversed, with a final success for each traversal corresponding to retention of a clause. After each success, a "fail" operation forced backtracking, which caused another clause to be extracted by retraction from the set of support, followed by another traversal of the proof tree.

If a clause could not be generated using the inference rules, or if it was discarded by subsumption, the failure simply caused an early re-traversal.

Figure 1 shows the control structure of the compiled theorem. The components shown as shaded rectangles are the Prolog clauses corresponding to the clauses of the theorem, which are executed as parts of the Prolog program.

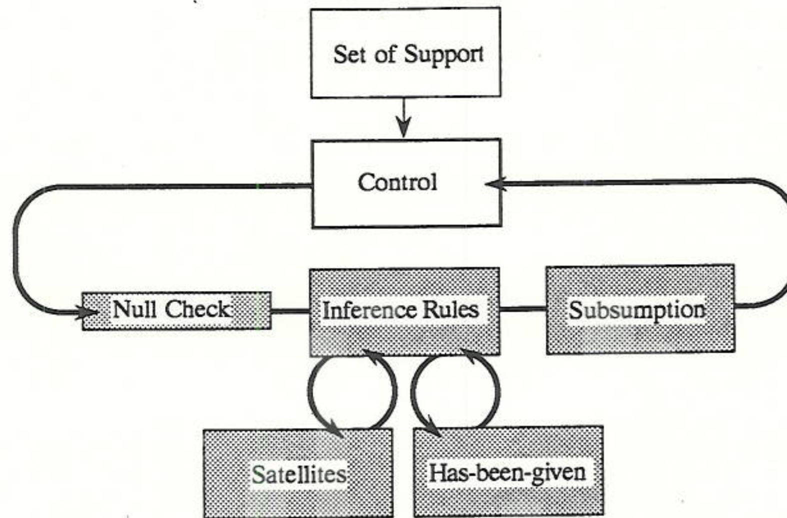

Figure 1. Structure of the compiled theorem prover

7. SAM'S LEMMA: A TEST CASE FOR THIS APPROACH.

The theorem selected, Sam's Lemma, is of additional importance in that it is the only problem in the McCharen problem set [Mc76] that does not experience a single failure due to an occurs check during unification, and that further has all generated clauses containing only ground instances of variables. These two properties allow a proof of Sam's Lemma to be developed using existing Prolog interpreters, which lack the occurs check.

```
/*

TITLE: sams lemma
Mills
9 Apr 85


REVISED: 21 Apr 85

Revision implements a further-stripped down version of subsumption, and
renames
what was earlier called "backward subsumption" (incorrectly) to the proper
name
of "forward subsumption".

History is no longer encoded into the "sc" & "sos" clauses.  It may be
```

```
recovered from the logfile if necessary.


ORIGINAL:


This program contains Sam's Lemma encoded in PROMETHEUS.

Sams Lemma may be stated as follows:

Let L be a modular lattice with 0 and 1.  Suppose that
A and B are elements of L such that (A v B) and (A ^ B)
both have not necessarily unique complements.

Then:

((A v B) v ((A ^ B) ^ B)) ^
((A v B) v ((A ^ B) ^ A))     =    (A v B)

Axioms for Sam's Lemma.


(1 v X) = 1
(X v X) = X
(0 v X) = X
(0 ^ X) = 0
(X ^ X) = X
(1 ^ X) = X

1.  MAX(cone,x,cone)
2.  MAX(x,x,x)
3.  MAX(czero,x,x)
4.  MIN(czero,x,czero)
5.  MIN(x,x,x)
6.  MIN(cone,x,x)


----------------------------------------------------------------------
The PROMETHEUS control module.

It is stripped down & takes advantage of knowledge about the problem to
omit certain more general ( & time consuming ) operations.

*/

demo :-

        prove( 'sams lemma' ).



prove( 'sams lemma' ) :-

        Start is cputime,
        repeat,
        get_clause( 1, SosID, FirstLiteral ),
        clash( SosID, FirstLiteral ),
```

```
        exhibit_proof( Start) .


save_log :-

        tell( 'logfile.sam' ),
        told.


/* fail here backtracks through all ir clauses
   then we fall through to next clash clause &
   put our "driver" clause into the hbg unless
   we found a null clause
*/

clash( SosID, FirstLiteral ) :-

        ir( FirstLiteral,
            Resolvent,
            IrID,
            History),

        check_for_null( Resolvent, SosID, IrID, History ).




check_for_null( min(b3,e2,a2), _ , _ , _ ) :-

        nl,
        write( 'min(b3,e2,a2)' ),
        nl,
        write( '[]' ),

        tell( 'logfile.sam' ),
        nl,
        write( 'min(b3,e2,a2) []' ),
        nl,
        write( '[]' ),
        tell( user ).


check_for_null( FirstLiteral, SosID, IrID, History ) :-

        /* subsumption succeeds only if the resolvent is kept
*/
        /* if we fail, then we back up & put original clause into hbg
*/

        subsume( FirstLiteral ),

        make_history( SosID, IrID, History, NewHistory ),

        retract( clause_id( Count ) ),
        NewCount is Count + 1,
```

```
        assert( clause_id( NewCount ) ),

        assert( sos( 1, FirstLiteral, Count ) ),

        assert(( fs( FirstLiteral, Count )
              )),

        nl,
        write( Count ),
        tab(1),
        write( FirstLiteral ),
        tab(1),
        write( NewHistory ),

        /* keep a logfile
*/

        tell( 'logfile.sam' ),
        nl,
        write( Count ),
        tab(1),
        write( FirstLiteral ),
        tab(1),
        write( NewHistory ),
        tell( user ),


        !,
        fail.


get_clause( Weight, SosID, FirstLiteral ) :-

        retract( sos( Weight, FirstLiteral, SosID ) ),


        assert( sc( FirstLiteral, SosID ) ),

        retract( given( Count ) ),
        NewCount is Count + 1,
        assert( given( NewCount ) ),

        nl,
        write( 'Given clause ' ),
        write( Count ),
        write( ' is clause '),
        write( SosID ),
        write( ': '),
        write( FirstLiteral ),

        /* keep a logfile
*/

        tell( 'logfile.sam' ),
        nl,
        write( 'Given clause ' ),
        write( Count ),
```

```
        write( ' is clause '),
        write( SosID ),
        write( ': '),
        write( FirstLiteral ),
        tell( user ),

        !.

get_clause( _ , _ , _ ) :-

        nl,
        write( 'Weighting excludes remaining clauses' ),
        nl,
        write( 'No proof found' ),
        nl,
        halt.




make_history( SosID, IrID, [], [ SosID, IrID ] ).

make_history( SosID, IrID, [ H | T ], [ SosID, IrID, H | T ] ).



subsume( Clause ) :-

        fs( Clause, _  ),
        !,
        fail.

subsume( _ ).


clause_id(32).


given(1).


exhibit_proof( Start) :-

        End is cputime,
        Total is End-Start,
        nl,
        write( 'Found proof in ' ), write( Total ), write( 'seconds' ),
        nl,
        write( 'Proof complete, see logfile.sam for history' ),
```

```
/* close logfile
*/

        tell( 'logfile.sam' ),
        nl,
        write( 'Proof complete' ),
        told.

/*

ORIGINAL:26 Mar 85

REVISED: 21 Apr 85

------------------------------------------------------------

INFERENCE RULE ATTRIBUTES:  INTIAL HYPERRESOLUTION SATELLITE CLAUSES

*/

sc( max(1,X,1), 1 ).
sc( max(X,X,X), 2 ).
sc( max(0,X,X), 3 ).
sc( min(0,X,0), 4 ).
sc( min(X,X,X), 5 ).
sc( min(1,X,X), 6 ).


/*

ORIGINAL:26 Mar 85

REVISED: 21 Apr 85

------------------------------------------------------------

INFERENCE RULE ATTRIBUTES:  HYPERRESOLUTION NUCLEI

*/


/*

(X ^ Y) = (Y ^ X)

7.  -MIN(x,y,z)    | MIN(y,x,z)

*/

ir( min(X,Y,Z), min(Y,X,Z), 7, [] ).



/*

(X v Y) = (Y v X)
```

```
8.  -MAX(x,y,z)    | MAX(y,x,z)

*/

ir( max(X,Y,Z), max(Y,X,Z), 8, [] ).



/*

X v (X ^ Y) = X
X ^ (X v Y) = X

9.  -MIN(x,y,z)    | MAX(x,z,x)
10. -MAX(x,y,z)    | MIN(x,z,x)

*/

ir( min(X,Y,Z), max(X,Z,X), 9, [] ).
ir( max(X,Y,Z), min(X,Z,X), 10, [] ).



/*

X ^ (Y ^ Z) = (X ^ Y) ^ Z
X v (Y v Z) = (X v Y) v Z

11. -MIN(x,y,xy)  | -MIN(y,z,yz) | -MIN(x,yz,xyz)   | MIN(xy,z,xyz)
12. -MIN(x,y,xy)  | -MIN(y,z,yz) | -MIN(xy,z,xyz)   | MIN(x,yz,xyz)
13. -MAX(x,y,xy)  | -MAX(y,z,yz) | -MAX(x,yz,xyz)   | MAX(xy,z,xyz)
14. -MAX(x,y,xy)  | -MAX(y,z,yz) | -MAX(xy,z,xyz)   | MAX(x,yz,xyz)

*/

ir( min(X,Y,XY), min(XY,Z,XYZ), 11, [ H1,H2 ] ) :-

        sc( min(Y,Z,YZ), H1),
        sc( min(X,YZ,XYZ), H2).

ir( min(Y,Z,YZ), min(XY,Z,XYZ), 11, [ H1,H2 ] ) :-

        sc( min(X,Y,XY), H1),
        sc( min(X,YZ,XYZ), H2).

ir( min(X,YZ,XYZ), min(XY,Z,XYZ), 11, [ H1,H2 ] ) :-

        sc( min(Y,Z,YZ), H1),
        sc( min(X,Y,XY), H2).



ir( min(X,Y,XY), min(X,YZ,XYZ), 12, [ H1,H2 ] ) :-

        sc( min(Y,Z,YZ), H1),
```

```
        sc( min(XY,Z,XYZ), H2).

ir( min(Y,Z,YZ), min(X,YZ,XYZ), 12, [ H1,H2 ] ) :-

        sc( min(X,Y,XY), H1),
        sc( min(XY,Z,XYZ), H2).

ir( min(XY,Z,XYZ), min(X,YZ,XYZ), 12, [ H1,H2 ] ) :-

        sc( min(Y,Z,YZ), H1),
        sc( min(X,Y,XY), H2).



ir( max(X,Y,XY), max(XY,Z,XYZ), 13, [ H1,H2 ] ) :-

        sc( max(Y,Z,YZ), H1),
        sc( max(X,YZ,XYZ), H2).

ir( max(Y,Z,YZ), max(XY,Z,XYZ), 13, [ H1,H2 ] ) :-

        sc( max(X,Y,XY), H1),
        sc( max(X,YZ,XYZ), H2).

ir( max(X,YZ,XYZ), max(XY,Z,XYZ), 13, [ H1,H2 ] ) :-

        sc( max(Y,Z,YZ), H1),
        sc( max(X,Y,XY), H2).



ir( max(X,Y,XY), max(X,YZ,XYZ), 14, [ H1,H2 ] ) :-

        sc( max(Y,Z,YZ), H1),
        sc( max(XY,Z,XYZ), H2).

ir( max(Y,Z,YZ), max(X,YZ,XYZ), 14, [ H1,H2 ] ) :-

        sc( max(X,Y,XY), H1),
        sc( max(XY,Z,XYZ), H2).

ir( max(XY,Z,XYZ), max(X,YZ,XYZ), 14, [ H1,H2 ] ) :-

        sc( max(Y,Z,YZ), H1),
        sc( max(X,Y,XY), H2).



/*

(X ^ Z) = X --> (Z ^ (X v Y)) = (X v (Y ^ Z))
```

```
(X ^ Z) = Z --> (X ^ (Y v Z)) = (Z v (X ^ Y))

15. -MIN(x,z,x)    | -MAX(x,y,x1) | -MIN(y,z,y1) | -MIN(z,x1,z1) | MAX(x,y1,z1)
16. -MIN(x,z,x)    | -MAX(x,y,x1) | -MIN(y,z,y1) | -MAX(x,y1,z1) | MIN(z,x1,z1)
17. -MIN(x,z,z)    | -MAX(y,z,y1) | -MIN(x,y,x1) | -MIN(x,y1,z1) | MAX(z,x1,z1)
18. -MIN(x,z,z)    | -MAX(y,z,y1) | -MIN(x,y,x1) | -MAX(z,x1,z1) | MIN(x,y1,z1)

*/

ir( min(X,Z,X), max(X,Y1,Z1), 15, [ H1,H2,H3 ] ) :-

        sc( max(X,Y,X1), H1),
        sc( min(Y,Z,Y1), H2),
        sc( min(Z,X1,Z1), H3).

ir( max(X,Y,X1), max(X,Y1,Z1), 15, [ H1,H2,H3 ] ) :-

        sc( min(X,Z,X), H1),
        sc( min(Y,Z,Y1), H2),
        sc( min(Z,X1,Z1), H3).

ir( min(Y,Z,Y1), max(X,Y1,Z1), 15, [ H1,H2,H3 ] ) :-

        sc( max(X,Y,X1), H1),
        sc( min(X,Z,X), H2),
        sc( min(Z,X1,Z1), H3).

ir( min(Z,X1,Z1), max(X,Y1,Z1), 15, [ H1,H2,H3 ] ) :-

        sc( max(X,Y,X1), H1),
        sc( min(Y,Z,Y1), H2),
        sc( min(X,Z,X), H3).



ir( min(X,Z,X), min(Z,X1,Z1), 16, [ H1,H2,H3 ] ) :-

        sc( max(X,Y,X1), H1),
        sc( min(Y,Z,Y1), H2),
        sc( max(X,Y1,Z1), H3).

ir( max(X,Y,X1), min(Z,X1,Z1), 16, [ H1,H2,H3 ] ) :-

        sc( min(X,Z,X), H1),
        sc( min(Y,Z,Y1), H2),
        sc( max(X,Y1,Z1), H3).

ir( min(Y,Z,Y1), min(Z,X1,Z1), 16, [ H1,H2,H3 ] ) :-

        sc( max(X,Y,X1), H1),
        sc( min(X,Z,X), H2),
        sc( max(X,Y1,Z1), H3).

ir( max(X,Y1,Z1), min(Z,X1,Z1), 16, [ H1,H2,H3 ] ) :-

        sc( max(X,Y,X1), H1),
        sc( min(Y,Z,Y1), H2),
```

```
        sc( min(X,Z,X), H3).



ir( min(X,Z,Z), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Y,X1), H2),
        sc( min(X,Y1,Z1), H3).

ir( max(Y,Z,Y1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( min(X,Z,Z), H1),
        sc( min(X,Y,X1), H2),
        sc( min(X,Y1,Z1), H3).

ir( min(X,Y,X1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Z,Z), H2),
        sc( min(X,Y1,Z1), H3).

ir( min(X,Y1,Z1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Y,X1), H2),
        sc( min(X,Z,Z), H3).



ir( min(X,Z,Z), min(X,Y1,Z1), 18, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Y,X1), H2),
        sc( max(Z,X1,Z1), H3).

ir( max(Y,Z,Y1), min(X,Y1,Z1), 18, [ H1,H2,H3 ] ) :-

        sc( min(X,Z,Z), H1),
        sc( min(X,Y,X1), H2),
        sc( max(Z,X1,Z1), H3).

ir( min(X,Y,X1), min(X,Y1,Z1), 18, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Z,Z), H2),
        sc( max(Z,X1,Z1), H3).

ir( max(Z,X1,Z1), min(X,Y1,Z1), 18, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Y,X1), H2),
        sc( min(X,Z,Z), H3).
```

```
/*

Negation of Sams Lemma

19. MIN(a,b,c)
20. MAX(d,c,cone)
21. MIN(b,d,e)
22. MIN(a,e,czero)
23. MAX(b,a2,b2)
24. MAX(a,b2,cone)
25. MAX(a,b,c2)
26. MIN(a2,c2,czero)
27. MIN(d,a,d2)
28. MAX(a2,d2,e2)
29. MIN(d,b,a3)
30. MAX(a2,a3,b3)

NO INFERENCE RULE ATTRIBUTES FOR THE ABOVE POSITIVE UNIT CLAUSES


------------------------------------------------------------
This negative unit clause has an "ir" clause, but it is not visible at the
start of the run.  It is made visible as the run progresses.

Also note that a hyperresolution rule for a unit clause will generate a null
clause automatically if there are no satellite literals.


31. -MIN(b3,e2,a2)

*/


ir( min(b3,e2,a2), 31 ).


/*

ORIGINAL:26 Mar 85

------------------------------------------------------------

SET OF SUPPORT CLAUSE ATTRIBUTES: NEGATION OF THE THEOREM

19. MIN(a,b,c)
20. MAX(d,c,cone)
21. MIN(b,d,e)
```

```
22. MIN(a,e,czero)
23. MAX(b,a2,b2)
24. MAX(a,b2,cone)
25. MAX(a,b,c2)
26. MIN(a2,c2,czero)
27. MIN(d,a,d2)
28. MAX(a2,d2,e2)
29. MIN(d,b,a3)
30. MAX(a2,a3,b3)

*/

sos( 1, min( a , b , c ), 19 ).
sos( 1, max( d , c , 1 ), 20 ).
sos( 1, min( b , d , e ), 21 ).
sos( 1, min( a , e , 0 ), 22 ).
sos( 1, max( b , a2, b2), 23 ).
sos( 1, max( a , b2, 1 ), 24 ).
sos( 1, max( a , b , c2), 25 ).
sos( 1, min( a2, c2, 0 ), 26 ).
sos( 1, min( d , a , d2), 27 ).
sos( 1, max( a2, d2, e2), 28 ).
sos( 1, min( d , b , a3), 29 ).
sos( 1, max( a2, a3, b3), 30 ).



/*

ORIGINAL:13 Apr 85

REVISED: 21 Apr 85

---------------------------------------------------------------

SUBSUMPTION ATTRIBUTES: FORWARD SUBSUMPTION USING AXIOMS


1.  MAX(cone,x,cone)
2.  MAX(x,x,x)
3.  MAX(czero,x,x)
4.  MIN(czero,x,czero)
5.  MIN(x,x,x)
6.  MIN(cone,x,x)

*/


fs( max(1,X,1), 1).
fs( max(X,X,X), 2).
fs( max(0,X,X), 3).
fs( min(0,X,0), 4).
fs( min(X,X,X), 5).
fs( min(1,X,X), 6).
```

```
/*
-------------------------------------------------------------

SUBSUMPTION ATTRIBUTES: FORWARD SUBSUMPTION USING NEGATION OF THE THEOREM

19. MIN(a,b,c)
20. MAX(d,c,cone)
21. MIN(b,d,e)
22. MIN(a,e,czero)
23. MAX(b,a2,b2)
24. MAX(a,b2,cone)
25. MAX(a,b,c2)
26. MIN(a2,c2,czero)
27. MIN(d,a,d2)
28. MAX(a2,d2,e2)
29. MIN(d,b,a3)
30. MAX(a2,a3,b3)
31. -MIN(b3,e2,a2)

*/

fs( min(a,b,c), 19).
fs( max(d,c,1), 20).
fs( min(b,d,e), 21).
fs( min(a,e,0), 22).
fs( max(b,a2,b2), 23).
fs( max(a,b2,1), 24).
fs( max(a,b,c2), 25).
fs( min(a2,c2,0), 26).
fs( min(d,a,d2), 27).
fs( max(a2,d2,e2), 28).
fs( min(d,b,a3), 29).
fs( max(a2,a3,b3), 30).
```

REFERENCES

Bundy, A. 1983. "The Computer Modelling of Mathematical Reasoning", Academic Press, London.

Gabriel, J., Lindholm, T., Lusk, E.L., and Overbeek, R.A. 1984. "A Tutorial on the Warren Abstract Machine", Argonne National Laboratory.

Overbeek, R. A., and Lusk, E. 1975. "Data Structures and Control Architecture for Implementation of Theorem-Proving Programs".

McCharen, J. D., Overbeek, R.A., and Wos, L.A. 1976. "Problems and Experiments for and with Automated Theorem Proving Programs", IEEE.

Stickel, M. E. 1984. "A Prolog Technology Theorem Prover", International Symposium on Logic Programming, IEEE.

Wos, L., Overbeek, R., Lusk, E., and Boyle, J. 1984. "Automated Reasoning", Prentice-Hall, New Jersey.