A Prolog Technology Theorem Prover: A New Exposition and Implementation in Prolog[1]

Mark E. Stickel
Artificial Intelligence Center
SRI International
Menlo Park, California 94025

Technical Note 464
June 1989

Abstract

A Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus. It differs from Prolog in its use of unification with the occurs check for soundness, depth-first iterative-deepening search instead of unbounded depth-first search to make the search strategy complete, and the model elimination reduction rule that is added to Prolog inferences to make the inference system complete. This paper describes a new Prolog-based implementation of PTTP. It uses three compile-time transformations to translate formulas into Prolog clauses that directly execute, with the support of a few run-time predicates, the model elimination procedure with depth-first iterative-deepening search and unification with the occurs check. Its high performance exceeds that of Prolog-based PTTP interpreters, and it is more concise and readable than the earlier Lisp-based compiler, which makes it superior for expository purposes. Examples of inputs and outputs of the compile-time transformations provide an easy and quite precise way to explain how PTTP works. This Prolog-based version makes it easier to incorporate PTTP theorem-proving ideas into Prolog programs. Some suggestions are made on extensions to Prolog that could be used to improve PTTP's performance.

[1]This research was supported by the National Science Foundation under Grant CCR-8611116. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation or the United States government.

1 Introduction

A Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus. We present here a new exposition and implementation of PTTP that uses Prolog to explain and implement PTTP. There are few new ideas here—the principles, implementation, and performance of PTTP have been described in detail before [17]—but the new implementation still has several advantages.

PTTP is characterized by the use of sound unification with the occurs check where necessary, the complete model elimination inference procedure rather than just Prolog inference, and the depth-first iterative-deepening search procedure rather than unbounded depth-first search. These particular inference and search methods are used instead of other complete methods because they can be implemented using basically the same implementation ideas, including compilation, that enable Prolog's very high inference rate. Other inference systems and search methods may explore radically different and smaller search spaces than PTTP, but PTTP's design enables it to come closer to matching Prolog's inference rate.

Several PTTP-like systems have been implemented:

- A Lisp-based interpreter [16].
- A Lisp-based compiler [17].
- F-Prolog, a Prolog-based interpreter [19].
- Expert Thinker, a commercial version of F-Prolog [14].
- Parthenon [2] and METEOR [1], parallel implementations based on the Warren abstract machine and SRI model for OR-parallel execution of Prolog.

Several other deduction systems developed in recent years also use features associated with PTTP, such as compiled inference operations for the full first-order predicate calculus, especially for linear strategies, and the use of depth-first iterative-deepening search in deduction.

We present here a new implementation of Prolog using a Prolog-based compiler. First-order predicate calculus formulas are translated by the PTTP compiler, written in Prolog, to Prolog clauses that are compiled by the Prolog compiler and will then directly execute the PTTP inference and search procedure.

The new implementation has several advantages. First, its performance is high (superior to that of PTTP interpreters), although still not equal to that of the Lisp-based compiler implementation. (Missing features of Prolog such as global variables and greater access to information on the stack contribute to this inefficiency. We make some suggestions in Section 5 on extensions that would permit performance improvement. Of course, the highest possible PTTP performance, which requires the largest investment in implementation work, entails developing Warren abstract machine extensions specific to the needs of PTTP, as is being done in the parallel implementations cited above.)

Second, the Prolog-based PTTP should generally produce much shorter object code than our Lisp-based compiler and compilation speed should also be improved. The Prolog clauses produced by the PTTP compiler typically will be compiled by the Prolog compiler to a concise abstract-machine target language. Our Lisp-based PTTP compiled its input to Lisp code that was then compiled to machine code rather than a Prolog abstract-machine language, so object code could be quite large and compilation time long.

The code for the Prolog-based version is also shorter and more perspicuous than that for the Lisp-based version. Modifiability is enhanced. Elements of PTTP, like logical variables and backtracking, that are basic features of Prolog had to be explicitly handled in the Lisp version of the PTTP compiler. In effect, we had to write a PTTP-to-Prolog compiler *and* a Prolog-to-Lisp compiler for the Lisp version; for this Prolog-based version, only the former is necessary.

The Prolog-based version is also more readily usable by those who would like to incorporate PTTP reasoning for some tasks into larger logic programs written in Prolog. Since the output of this PTTP-to-Prolog compiler is pure Prolog code, it is easy to achieve parallel execution of PTTP inference by simply executing the code on any parallel implementation of standard, sequential Prolog.

Finally, we feel that this version of "PTTP in Prolog" has pedagogical value. This description, and the code for the PTTP-to-Prolog compiler, explain clearly and precisely the principles of a Prolog technology theorem prover. Example inputs and outputs of the transformations used by PTTP clearly describe PTTP's operation.

We illustrate by example PTTP's recipe for transforming first-order predicate calculus formulas to Prolog clauses that, when executed, perform the complete model elimination theorem-proving procedure on the formulas. The fundamental problems with Prolog for theorem proving—unsound unification without the occurs check, incomplete unbounded depth-first search, and an inference system that is complete only for Horn clauses—are all overcome in this approach.

First, first-order predicate calculus formulas[2] are translated to Prolog clauses and their contrapositives. (We will not describe this process, since it is not specific to PTTP, but see Appendix A, pp. 31ff., for the code.)

The recipe then specifies application of

- A compile-time transformation for sound unification that linearizes clause heads and moves unification operations that require the occurs check into the body of the clause where they are performed by a new predicate that performs sound unification with the occurs check.

- A compile-time transformation for complete depth-bounded search that adds extra arguments for the input and output depth bounds to each predicate and adds depth-bound test and decrement operations to the clause bodies.

- A compile-time transformation for complete model elimination inference that adds an extra argument for the list of ancestor goals to each predicate and adds ancestor-list update operations to the clause bodies; additional clauses are added to perform the

[2]The exact input format allowed is a conjunction of assertions that are in negation normal form (nested conjunctions and disjunctions of literals) and a conclusion that is a conjunction of literals. The assertions are implicitly univerally quantified and the conclusion is implicitly existentially quantified; it is assumed that all quantifiers have been removed previously by skolemization. It would be easy to extend the input format to connectives other than AND and OR and to do the skolemization.

model elimination pruning and reduction operations.

The recipe also requires run-time support in the form of

- The `unify` predicate that unifies its arguments soundly with the occurs check.
- The `search` predicate that controls iterative-deepening search's sequence of bounded depth-first searches.
- The `identical_member` and `unifiable_member` predicates that determine if a literal is identical to or unifiable with members of the ancestor list.

An additional, optional compile-time transformation with run-time support permits an abbreviated form of the proof to be printed after it is found. (We will not describe this transformation, since it is not part of PTTP's inference or search procedure, but see Appendix A, pp. 30ff., for the code.)

2 Sound Unification

The first obstacle to general-purpose theorem proving that must be overcome is Prolog's use of unification without the occurs check. For reasons of efficiency, many implementations of Prolog do not check whether a variable is being bound to a term that contains that same variable. This can result in unsound or even nonterminating unification. The following Prolog programs "prove" that there is a number that is less than itself and that in a group $a \circ z = z$ for some z.[3]

```
X<(X+1).
:- Y<Y.

p(X,Y,f(X,Y)).
:- p(a,Z,Z).
```

[3] The literal `p(X,Y,Z)` denotes $x \circ y = z$, where $\circ$ is the group multiplication operation. The literal `p(X,Y,f(X,Y))` states that every `X` and `Y` have a product `f(X,Y)`.

The invalid results rely upon the creation of circular bindings for variables during unification. If the values are unified later, unification may not terminate unless a unification algorithm for infinite terms is used [4, 5].

Although applying the occurs check in logic programming can be quite costly, it is less likely to be too expensive in theorem proving, since the huge terms sometimes generated in logic programming are less likely to appear in theorem proving.

It was not always apparent that the problem of unification without the occurs check could be remedied without changing Prolog's underlying architecture (e.g., altering or extending the Prolog-machine instruction set).

Although it is easy to write a Prolog predicate `unify` that performs sound unification with the occurs check [12, 15] (see Appendix A, pp. 23ff.), the trick is to invoke this unification algorithm instead of Prolog's whenever necessary during the unification of a goal and the head of a clause.

It has often been noted that one case in which the occurs check is certain to be unnecessary is in the unification of a pair of terms with no variables in common (as is the case of Prolog goals and clause heads) provided at least one of the terms has no repeated variables (terms without repeated variables are called *linear*).

Based on the existence of a Prolog predicate `unify` that performs sound unification with the occurs check and the observation that the occurs check is unnecessary if the clause head is linear, there is an elegant method of transforming clauses to isolate parts that may require unification with the occurs check [12, 13]. Repeated occurrences of variables are replaced by new variables to make the clause head linear. Unifying the clause head with a goal can then proceed without the occurs check and will not create any circular bindings. The new variables in the transformed clause head are then unified with the original variables by sound unification with the occurs check in the transformed clause body.

In the examples above, the clauses

```
X<(X+1).
p(X,Y,f(X,Y)).
```

are replaced by the clauses

```
X<(X1+1) :-
    unify(X,X1).
p(X,Y,f(X1,Y1)) :-
    unify(X,X1),
    unify(Y,Y1).
```

in which the occurs check needs to be performed only during the calls to `unify` in the body.

The code for this transformation is shown in Appendix A, p. 22.

This transformation makes it easy to incorporate sound unification into Prolog systems that lack it. A new predicate `unify` that performs sound unification must be added, but no changes to the Prolog-machine instruction set are necessary. The predicate `unify` can be written in Prolog, although writing it in a lower-level language may yield a large improvement in performance.

For those Prolog systems that support unification of infinite terms, it is sufficient to add to the body of a clause acyclicity tests for repeated variables in the head of the clause.

3 Complete Search Strategy

Even for Horn clauses, Prolog is unsatisfactory as a theorem prover because many theorem-proving problems cannot be solved using Prolog's unbounded depth-first search strategy.

A simple solution to this problem is to replace Prolog's unbounded depth-first search strategy with bounded depth-first search. Backtracking when reaching the depth bound would cause the entire search space, up to a specified depth, to be searched completely. A complete search strategy could perform a sequence of bounded depth-first searches: first one tries to find a proof with depth 1, then depth 2, and so on, until a proof is found. This is called *depth-first iterative-deepening search* [6]. The effect is similar to breadth-first search except that results from earlier levels are recomputed rather than stored. The lower storage requirements and greater efficiency of the stack-based representation for derived clauses used in depth-first search compensate for the recomputation cost.

Because the size of the search space grows exponentially as the depth bound is increased, the number of recomputed results is not excessive. In particular, depth-first iterative-deepening search performs only about $\frac{b}{b-1}$ times as many operations as breadth-first search, where b is the branching factor [18] (for $b = 1$, when there is no branching, breadth-first search is $O(n)$ and depth-first iterative-deepening search is $O(n^2)$, where n is the depth). Korf [6] has shown that depth-first iterative-deepening search is asymptotically optimal among brute-force search strategies in terms of solution length, space, and time: it always finds a shortest solution; the amount of space required is proportional to the depth; and, although the amount of time required is exponential, this is the case for all brute-force search strategies; in general, it is still only a constant factor more expensive than breadth-first search.

Consider the following fragment of a set of axioms of group theory:

```
p(e,X,X).                                    % left identity
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).   % associativity clause (1 of 2)
```

Use of these clauses can be controlled during depth-first iterative-deepening search by adding extra arguments for the depth bound before and after the literal is proved. The depth bound is reduced by one at each inference step and the computation is allowed to proceed only if the depth bound remains nonnegative. The transformed clauses are:

```
p(e,X,X,DepthIn,DepthOut) :-
    DepthIn >= 1, DepthOut is DepthIn - 1.
p(U,Z,W,DepthIn,DepthOut) :-
    DepthIn >= 1, Depth1 is DepthIn - 1,
    p(X,Y,U,Depth1,Depth2),
    p(Y,Z,V,Depth2,Depth3),
    p(X,V,W,Depth3,DepthOut).
```

Counting inferences at the time they are performed as above is comparatively inefficient. The depth bound is often reached with many goals still pending; the search should have been stopped earlier. Reducing the depth bound when subgoals are added to the set of pending goals by an inference operation instead of when they are removed results in much better performance through earlier cutoffs and lower overhead. In this method, the transformed clauses are:

```
p(e,X,X,Depth,Depth).
p(U,Z,W,DepthIn,DepthOut) :-
```

```
        DepthIn >= 3, Depth1 is DepthIn - 3,
        p(X,Y,U,Depth1,Depth2),
        p(Y,Z,V,Depth2,Depth3),
        p(X,V,W,Depth3,DepthOut).
```

Technically, this employs the iterative-deepening A* algorithm [7], not simply depth-first iterative-deepening search, because the depth bound is reduced by the albeit trivial admissable estimator that estimates n inference steps will be required to prove the n subgoals in the body of a clause. Better, but still admissable, estimators are possible [17] but may require a test of whether a potentially complementary ancestor exists, which is costly in this implementation (see Sections 4 and 5.4).

The code for this transformation is shown in Appendix A, pp. 24ff.

A "driver" predicate `search` can be written easily (see Appendix A, pp. 26ff.) to try to prove its goal argument with progressively greater depth bounds within specified limits. The execution of `search(Goal,Max,Min,Inc)` attempts to solve `Goal` by a sequence of bounded depth-first searches that allow at least `Min` and at most `Max` subgoals, incrementing by `Inc` between searches. The last one, two, or three arguments of `search` can be omitted with default values of infinity, zero, and one. `Max` can be specified to bound the total search effort. It can also be reduced by specifying `Min` when it is known that no solution can be found with fewer than `Min` subgoals. When the branching factor is small and there are few new inferences for each additional level of search, total search effort may be reduced by skipping some levels by specifying an `Inc` value greater than one.

The `search` predicate succeeds for each solution it discovers. Backtracking into `search` continues the search for additional solutions. When only a single solution (proof) is needed, the `search` call can be followed by a cut operation to terminate further attempts to find a solution.

4 Complete Inference System

Prolog's inference system is often described in terms of the reduction of the initial list of literals in the query to the empty list by a sequence of Prolog inference steps. Each step matches the leftmost literal in the list with the head of a clause, eliminates the leftmost literal, and adds the body of the clause to the beginning of the list. If the list of literals is `:- q1,...,qn` then the lists

```
:- q2,...,qn
:- p1,...,pm,q2,...,qn
```

can be derived by resolution with the clauses `q1` and `q1 :- p1,...,pm`.

Prolog's incompleteness for non-Horn clauses can be demonstrated by its failure to prove Q from $P \vee Q$ and $\neg P \vee Q$. All the contrapositive clauses of $P \vee Q$ and $\neg P \vee Q$[4]

```
q :- not_p.
p :- not_q.
q :- p.
not_p :- not_q.
```

are insufficient to reduce `:- q` to the empty list of literals.

Prolog employs the *input* restriction of resolution; derived clauses are allowed to be resolved only with input clauses. Although input resolution is complete for Horn clauses, it is incomplete in general. However, the *linear* restriction of resolution, in which derived clauses can be resolved with their own ancestor clauses or with input clauses, is complete in general.

The *model elimination (ME)* procedure [9, 10] can be viewed as very convenient and efficient way to implement linear resolution. It is a complete inference system for non-Horn as well as Horn sets of clauses.[5] The model elimination procedure does not eliminate the leftmost literal in the resulting list of literals as Prolog does, but instead retains it as a *framed literal*:

```
:- [q1],q2,...,qn
:- p1,...,pm,[q1],q2,...,qn
```

[4]The complement of literal `p` is `not_p`. Rather than use a negation operator, we use pairs of predicate names `p` and `not_p`, `q` and `not_q`, etc.

[5]The *SL resolution* procedure [8] is similar; the principal difference is its need for an additional factoring operation. Prolog's inference system is often referred to as *SLD resolution* (SL resolution for definite, i.e., Horn, clauses).

The literal `q1` is framed (and shown as `[q1]` to signify its framed status); the literals `p1,...,pm` are unframed; the literals `q2,...,qn` are framed or unframed as they were in `:- q1,...,qn`. Leftmost framed literals are removed immediately.

The *ME reduction* inference rule uses framed literals to eliminate complementary literals:

```
:- q2,...,qn
```

can be derived from `:- q1,...,[qi],...,qn` if `q1` is complementary to some framed literal `qi`.

This inference rule makes it possible to prove Q from $P \vee Q$ and $\neg P \vee Q$:

```
:- q                    % initial goal
:- p,[q]                % resolve with q :- p
:- not_q,[p],[q]        % resolve with p :- not_q
:- [p],[q]              % use ME reduction rule
:-                      % delete leftmost framed literals
```

The ME reduction rule employs reasoning by contradiction. If, as in the above proof, in trying to prove Q, we discover that Q is true if P is true and also that P is true if $\neg Q$ is true, then Q must be true. The rationale is that Q is either true or false; if we assume that Q is false, then P must be true, and hence Q must also be true, which is a contradiction; therefore, the hypothesis that Q is false must be wrong and Q must be true.

The list of framed literals to the right of a literal is just the list of that goal's ancestors. The list of ancestor literals can be passed in an extra argument position; the current goal can be added to the front of the list and the new list passed to subgoals in nonunit clause bodies.

The clauses

```
p(e,X,X).
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
```

can be transformed to

```
p(e,X,X,Ancestors).
p(U,Z,W,Ancestors) :-
    NewAncestors = [p(U,Z,W) | Ancestors],
    p(X,Y,U,NewAncestors),
    p(Y,Z,V,NewAncestors),
    p(X,V,W,NewAncestors).
```

The code for this transformation is shown in Appendix A, pp. 28ff.

An extra clause that performs the ME reduction operation is included in each transformed procedure:

```
p(X,Y,Z,Ancestors) :-
    unifiable_member(not_p(X,Y,Z),Ancestors).
```

This clause succeeds each time the literal `p(X,Y,Z)` can be made complementary to an ancestor literal. The `unifiable_member` predicate is a membership-testing predicate that uses sound unification with the occurs check.

In addition, an extra clause at the beginning of each procedure that eliminates some cases of looping has been found to be a cost-effective addition. The model elimination procedure remains complete with this search-space pruning by identical ancestor operation.

```
p(X,Y,Z,Ancestors) :-
    identical_member(p(X,Y,Z),Ancestors),
    !, fail.
```

The `identical_member` predicate tests whether a literal is identical (by using the `==` predicate) to a literal in the list.

Another presentation of the model elimination procedure and its implementation in the manner of Prolog can be found in Maier and Warren [11].

5 Evaluation

The cost of PTTP compared to Prolog in terms of size of the input can be determined by

- A Prolog clause is required for each literal (all contrapositives are required).
- Two clauses are added to each procedure: one for the model elimination reduction operation and one for the identical-ancestor pruning operation.
- An extra `unify` literal is added to the body of a clause for each repeated occurrence of a variable in the head of the clause.

Example	Number of Clauses	Depth of Proof	Lisp Implementation Number of Inferences	Lisp Implementation Run Time (sec)	Prolog Implementation Number of Inferences	Prolog Implementation Run Time (sec)
1	5	4	5	0.002	5	0.005
2	7	10	1,589	0.373	1,938	0.637
3	5	10	206	0.046	264	0.095
4	5	7	26	0.005	32	0.010
5	9	4	4	0.001	4	0.002
6	9	7	26	0.005	32	0.010
7	7	6	24	0.004	24	0.006
8	9	13	3,104	0.652	3,830	2.522
9	8	10	163	0.027	191	0.135
Total			5,147	1.115	6,320	3.422

Table 1: PTTP Performance on Chang and Lee Examples

- Three extra literals are added to the body of each nonunit clause: one to test the depth bound, one to decrement it, and one to save the head on the list of ancestor goals.
- Two extra arguments are added to each literal for the input and output depth bounds.
- One (or more—our implementation uses two) extra argument is added to each literal for the list of ancestor goals.
- Additional arguments and literals may optionally be added (in our implementation, two extra arguments for each literal and one extra literal in the body of each clause) to compute the information needed to print the proof after it is found.

Appendix B contains an example of the input and output of the PTTP-to-Prolog compiler; Appendix C gives a proof of the example problem.

Table 1 gives results for the examples that appear in Chang and Lee [3], pp. 298–305, for both the Lisp implementation [17] and this Prolog implementation of PTTP running on a Symbolics 3600 with IFU. The Prolog implementation performs one thousand to three thousand model elimination inferences per second. This is a high inference rate for a theorem prover, although it is low for Prolog. The Lisp implementation of PTTP is somewhat more efficient.

We examine here some sources of inefficiency in this Prolog implementation of PTTP. Because many of these are inherent limitations of Prolog, this discussion can be taken as identifying some problems with Prolog that inhibit the development of the highest possible performance PTTP in Prolog and arguing for particular extensions to Prolog. Similar extensions exist in some Prolog implementations. In particular, there have been many proposed schemes for destructive assignment operations on data structures or global variables, though none has become standard or widely available.

5.1 Merging Clauses

Merging clauses that have the same heads and initial goals in the body would improve efficiency. For example, the following two clauses from procedure `p` shown in Appendix B

```
p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-          % clause from wff 3
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [p(X) | PosAnc],                % save head goal as ancestor
    not_d(g(X),X,NewPosAnc,NegAnc,D1,DepthOut). % solve the subgoal
p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-          % clause from wff 4
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [p(X) | PosAnc],                % save head goal as ancestor
    not_l(1,g(X),NewPosAnc,NegAnc,D1,DepthOut). % solve the subgoal
```

can be merged into the single clause

```
p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-          % clause from wffs 3 and 4
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [p(X) | PosAnc],                % save head goal as ancestor
    (not_d(g(X),X,NewPosAnc,NegAnc,D1,DepthOut);% solve subgoal from wff 3
     not_l(1,g(X),NewPosAnc,NegAnc,D1,DepthOut)).% solve subgoal from wff 4
```

More and stronger merges are possible if reordering clauses and literals is allowed.

5.2 Inefficiency of Sound Unification

The sound unification procedure with the occurs check is written in Prolog. For Prolog implementations that allow predicates programmed in lower-level languages, it should be possible to substantially speed up the unification done by `unify` calls introduced by the sound-unification transformation and `unifiable_member` calls introduced by the complete-search transformation. Ideally, Prolog systems should provide an efficient `unify` predicate (or an efficient acyclicity test if they support unification of infinite terms).

The principal reason for the Lisp implementation of PTTP performing fewer inferences than the Prolog implementation is that the Lisp implementation performs a cut operation if the head of a unit clause subsumes rather than merely unifies with the goal. For example, no alternatives need be tried, and a cut operation can be performed if the goal `p(e,a,a)` is solved by the unit clause `p(e,X,X)`, since the goal has been solved without instantiation. But if the goal `p(e,Y,a)` is solved with this clause, alternatives that do not match `Y` and `a` must still be considered. A cut operation can likewise be performed in the ME reduction operation if a goal is identical to the complement of an ancestor goal, not merely unifiable with it.

Determining whether to cut is done at very little cost in the Lisp implementation of PTTP by checking whether the unification operation added any entries to the trail.[6] It would be desirable if this could be done equally cheaply in Prolog. Unification with the clause head would be constrained so that the substitution would instantiate only the head if possible, and the user would be able to determine if subsumption occurred. This eliminates the need to perform both unification and subsumption tests.

5.3 Inefficiency of Complete Search

We see the possibility of only relatively small improvements of the basic method of incorporating iterative-deepening search. The extra operations appear to be quite efficient. The test and decrement operations surely need to be performed regardless of implementation alternatives (although perhaps they could be performed more efficiently with unboxed numbers if iterative-deepening search were built in). A possible saving is the elimination of the extra arguments by storing the depth bound in a global variable. However, even this saving would probably be modest, given the efficiency of passing a small number of extra

[6] This makes some assumptions about which variables are trailed.

arguments in Prolog.

However, there is an occasionally useful optimization of the iterative-deepening search strategy that is expensive to implement in Prolog. Suppose that, in an exhaustive depth-bounded search, every time a goal fails due to the depth-bound test, the number of subgoals in the clause exceeds the depth bound by more than one. Then incrementing the depth bound by only one for the next search will surely lead to failure again. To ensure the possibility of finding a new proof in the next search, the depth bound should be increased by the minimum amount by which the number of subgoals exceeds the depth bound. Adding the extra in-line code or procedure for this in Prolog would probably be ineffective. The only way of saving this minimum in Prolog is with database assertions, which makes accessing and especially updating the minimum quite expensive. The extra time required would be noticeable; only rarely would search levels be skipped in compensation.

Another example of inefficiency is the extremely high cost of adding inference counting so that the number of inferences can be reported at the end of each bounded depth-first search and when a proof is found. Because inferences on success and failure branches must both be counted, the count can be saved only with database assertions. Assignable global variables would be much more efficient for keeping track of the inference count and the minimum amount by which the number of subgoals exceeds the depth bound.

5.4 Inefficiency of Complete Inference

The retention and access of ancestor goals in lists is quite inefficient. This inefficiency is difficult to remedy in Prolog.

There are two major problems. The first is that in the transformed clause

```
p(U,Z,W,Ancestors) :-
    NewAncestors = [p(U,Z,W) | Ancestors],
    p(X,Y,U,NewAncestors),
    p(Y,Z,V,NewAncestors),
    p(X,V,W,NewAncestors).
```

the goal that matches `p(U,Z,W)` is reconstructed and added to the front of `Ancestors` to form `NewAncestors`. This is quite wasteful since the goal (or rather its arguments) is already stored in its choice point on the stack. Making the ancestor goal directly available to the user as a term could eliminate the need for reconstructing it to add it to the ancestor list.

The second problem is the retention of the goals in an unindexed linear list. Even indexing on just the sign and predicate symbol, as in the Lisp implementation of PTTP, appreciably reduces the number of attempted matches in the model elimination reduction and pruning operations.

Although looking up a goal in a linear list is expensive, using a more complex data structure may be even more costly because clause heads are added to the ancestor list frequently (whenever solving the body of nonunit clauses) and their addition must be temporary (the head of a clause must be in the ancestor list only for the duration of the solution of the body).

A separate linear list could be used for each signed predicate, but this could result in a very large number (twice the number of predicates in the problem) of extra arguments to each predicate.[7] Separate lists are used in the Lisp implementation of PTTP, but instead of being passed as extra arguments, they are maintained in global variables that can be dynamically rebound.

Adding global variables that can be dynamically rebound like the `special` variables of Lisp would likewise provide an efficient mechanism for Prolog to access this information without the cost of passing the information through extra argument positions. Global variables, if they can be dynamically rebound, can be very useful even without destructive assignment operations. They could be a "conservative extension" of Prolog that promotes efficiency without adding side-effects that would damage or conceal the logical, nonprocedural interpretation of logic programs.[8]

With an imagined Lisp-inspired syntax for such an operation, the clause

[7]Actually, our implementation uses two extra arguments for ancestors—one for positive-literal ancestors and one for negative-literal ancestors—instead of the single list described here.

[8]Anything that can be done with nonassignable, dynamically rebindable global variables can be done in standard Prolog with some loss of efficiency, convenience, and clarity of programs by adding extra arguments to predicates (e.g., one for each global variable).

```
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
```

could be transformed to

```
p(U,Z,W) :-
    let([P_Ancestors = [p(U,Z,W) | P_Ancestors]],
        (p(X,Y,U),
         p(Y,Z,V),
         p(X,V,W))).
```

`P_Ancestors` is a global variable initialized to the empty list. It is locally rebound in the body of the transformed clause to include `p(U,Z,W)` when the body literals are solved. The following code could then be used for the ME pruning and reduction operations.

```
p(X,Y,Z) :-
    identical_member(p(X,Y,Z),P_Ancestors),
    !, fail.
p(X,Y,Z) :-
    unifiable_member(not_p(X,Y,Z),Not_P_Ancestors),
```

5.5 Proof Printing

Finally, a defect of this implementation of PTTP is the lack of ability to print in full the proof that it finds. Prolog, though it can be viewed as doing Horn clause theorem proving, provides no proof printing capability (sometimes it might helpful if it did), but a real theorem prover should. An optional compile-time transformation with run-time support permits the printing of *some* information about the proof, namely the list of indices of the input formulas used at each step in the proof. This is enough to make manual verification of the proof feasible though laborious. Information about which literal of the formula was resolved on and the variable bindings is unavailable. See Appendix C for sample output of the Lisp and Prolog versions of PTTP. The full proof is displayed by the Lisp version and a partial description of it is printed by the Prolog version.

Printing the full proof as the Lisp version does would require further development of the compile-time transformation and more run time to keep track of the additional information needed. Run-time overhead can be minimized by keeping track of the minimum amount of information required to fully describe the proof and computing the print representation of the proof from the minimal description of the proof and the uncompiled input formulas.

Greater access to Prolog internals could permit the proof to be extracted at the conclusion with little impact on run time. That is how it is done in the Lisp implementation: information needed to print the proof is found by using Symbolics Lisp debugger functions to examine the stack at the completion of the proof.

6 Conclusion

We have described and demonstrated by example the extension of Prolog to full first-order predicate calculus theorem proving, with sound unification, a complete search strategy, and a complete inference system, by means of three simple compiler transformations. The result is an implementation of a Prolog technology theorem prover (PTTP) in which transformed Prolog clauses perform PTTP-style theorem proving at a rate of thousands of inferences per second. We have also suggested some extensions to Prolog that would enable higher performance.

Writing the transformations in Prolog and transforming first-order predicate calculus formulas to Prolog clauses minimizes the effort necessary to implement a PTTP, makes PTTP-style theorem proving readily available in Prolog, and makes it easy to explain how PTTP theorem proving works.

Acknowledgements

I would like to thank Fernando Pereira and Mabry Tyson for their useful comments on the text of this paper and to thank Fernando for giving me feedback on the Prolog code as well.

References

[1] Astrachan, O. METEOR: model elimination theorem prover for efficient OR-parallelism. Unpublished, 1989.

[2] Bose, S., E.M. Clarke, D.E. Long, and S. Michaylov. Parthenon: a parallel theorem prover for non-Horn clauses. *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science,* Asilomar, California, June 1989.

[3] Chang, C.L. and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, New York, 1973.

[4] Colmerauer, A. Prolog and infinite trees. In Clark, K.L. and S.A. Tarnlund (Eds.). *Logic Programming.* Academic Press, New York, New York, 1982.

[5] Jaffar, J. Efficient unification over infinite terms. *New Generation Computing 2,* 3 (1984), 207–219.

[6] Korf, R.E. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence 27,* 1 (September 1985), 97–109.

[7] Korf, R.E. Iterative-deepening A*: an optimal admissable tree search. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* Los Angeles, California, August 1985, 1034-1036.

[8] Kowalski, R. and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence 2* (1971), 227–260.

[9] Loveland, D.W. A simplified format for the model elimination procedure. *Journal of the ACM 16,* 3 (July 1969), 349–363.

[10] Loveland, D.W. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, the Netherlands, 1978.

[11] Maier, D. and D.S. Warren. *Computing with Logic.* Benjamin/Cummings Publishing Co., Menlo Park, California, 1988.

[12] O'Keefe, R.A. Programming meta-logical operations in Prolog. DAI Working Paper No. 142, Department of Artificial Intelligence, University of Edinburgh, June 1983.

[13] Plaisted, D.A. Non-Horn clause logic programming without contrapositives. *Journal of Automated Reasoning 4,* 3 (September 1988), 287–325.

[14] Satz, R.W. *Expert Thinker* software package. Transpower Corporation, Parkerford, Pennsylvania, 1988.

[15] Sterling, L. and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[16] Stickel, M.E. A Prolog technology theorem prover. *New Generation Computing 2,* 4 (1984), 371–383.

[17] Stickel, M.E. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning 4,* 4 (December 1988), 353–380.

[18] Stickel, M.E. and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* Los Angeles, California, August 1985, 1073–1075.

[19] Umrigar, Z.D. and V. Pitchumani. An experiment in programming with full first-order logic. *Proceedings of the 1985 Symposium on Logic Programming,* Boston, Massachusetts, July 1985, 40–47.

A PTTP Compiler

Following is the Prolog code for the PTTP-to-Prolog compiler. Sample input and output of the compiler are shown in Appendix B.

```
%%% Sound unification.
%%%
%%% `add_sound_unification' transforms a clause so that its
%%% head has no repeated variables.  Unifying a goal with
%%% the clause head can then be done soundly without the occurs
%%% check.  The rest of the unification can then be done in
%%% the body of the transformed clause, using the sound `unify'
%%% predicate.
%%%
%%% For example,
%%%    p(X,Y,f(X,Y)) :- true.
%%% is transformed into
%%%    p(X,Y,f(X1,Y1)) :- unify(X,X1), unify(Y,Y1).

add_sound_unification((Head :- Body),(Head1 :- Body1)) :-
        linearize(Head,Head1,[],_,true,Matches),
        conjoin(Matches,Body,Body1).

linearize(TermIn,TermOut,VarsIn,VarsOut,MatchesIn,MatchesOut) :-
        nonvar(TermIn) ->
                functor(TermIn,F,N),
                myfunctor(TermOut,F,N),
                linearize_args(TermIn,TermOut,VarsIn,VarsOut,MatchesIn,MatchesOut,1,N);
        identical_member(TermIn,VarsIn) ->
                VarsOut = VarsIn,
                conjoin(MatchesIn,unify(TermIn,TermOut),MatchesOut);
        %true ->
                TermOut = TermIn,
                VarsOut = [TermIn|VarsIn],
                MatchesOut = MatchesIn.

linearize_args(TermIn,TermOut,VarsIn,VarsOut,MatchesIn,MatchesOut,I,N) :-
        I > N ->
                VarsOut = VarsIn,
                MatchesOut = MatchesIn;
        %true ->
                arg(I,TermIn,ArgI),
                linearize(ArgI,NewArgI,VarsIn,Vars1,MatchesIn,Matches1),
                arg(I,TermOut,NewArgI),
                I1 is I + 1,
                linearize_args(TermIn,TermOut,Vars1,VarsOut,Matches1,MatchesOut,I1,N).
```

```
%%% Sound unification algorithm with occurs check that is called
%%% by code resulting from the 'add_sound_unification' transformation.
%%% This should be coded in a lower-level language for efficiency.

unify(X,Y) :-
        var(X) ->
                (var(Y) ->
                        X = Y;
                %true ->
                        functor(Y,_,N),
                        (N = 0 ->
                                true;
                        N = 1 ->
                                arg(1,Y,Y1), not_occurs_in(X,Y1);
                        %true ->
                                not_occurs_in_args(X,Y,N)),
                        X = Y);
        var(Y) ->
                functor(X,_,N),
                (N = 0 ->
                        true;
                N = 1 ->
                        arg(1,X,X1), not_occurs_in(Y,X1);
                %true ->
                        not_occurs_in_args(Y,X,N)),
                X = Y;
        %true ->
                functor(X,F,N),
                functor(Y,F,N),
                (N = 0 ->
                        true;
                N = 1 ->
                        arg(1,X,X1), arg(1,Y,Y1), unify(X1,Y1);
                %true ->
                        unify_args(X,Y,N)).

unify_args(X,Y,N) :-
        N = 2 ->
                arg(2,X,X2), arg(2,Y,Y2), unify(X2,Y2),
                arg(1,X,X1), arg(1,Y,Y1), unify(X1,Y1);
        %true ->
                arg(N,X,Xn), arg(N,Y,Yn), unify(Xn,Yn),
                N1 is N - 1, unify_args(X,Y,N1).

not_occurs_in(Var,Term) :-
        Var == Term ->
                fail;
        var(Term) ->
                true;
        %true ->
                functor(Term,_,N),
```

```
                (N = 0 ->
                        true;
                N = 1 ->
                        arg(1,Term,Arg1), not_occurs_in(Var,Arg1);
                %true ->
                        not_occurs_in_args(Var,Term,N)).

not_occurs_in_args(Var,Term,N) :-
        N = 2 ->
                arg(2,Term,Arg2), not_occurs_in(Var,Arg2),
                arg(1,Term,Arg1), not_occurs_in(Var,Arg1);
        %true ->
                arg(N,Term,ArgN), not_occurs_in(Var,ArgN),
                N1 is N - 1, not_occurs_in_args(Var,Term,N1).


%%% Depth-first iterative-deepening search.
%%%
%%% `add_complete_search' adds arguments DepthIn and DepthOut
%%% to each PTTP literal to control bounded depth-first
%%% search.  When a literal is called,
%%% DepthIn is the current depth bound.  When
%%% the literal exits, DepthOut is the new number
%%% of levels remaining after the solution of
%%% the literal (DepthIn - DepthOut is the number
%%% of levels used in the solution of the goal.)
%%%
%%% For clauses with empty bodies or bodies
%%% composed only of builtin functions,
%%% DepthIn = DepthOut.
%%%
%%% For other clauses, the depth bound is
%%% compared to the cost of the body.  If the
%%% depth bound is exceeded, the clause fails.
%%% Otherwise the depth bound is reduced by
%%% the cost of the body.
%%%
%%% p :- q , r.
%%% is transformed into
%%% p(DepthIn,DepthOut) :-
%%%     DepthIn >= 2, Depth1 is DepthIn - 2,
%%%     q(Depth1,Depth2),
%%%     r(Depth2,DepthOut).
%%%
%%% p :- q ; r.
%%% is transformed into
%%% p(DepthIn,DepthOut) :-
%%%     DepthIn >= 1, Depth1 is DepthIn - 1,
%%%     (q(Depth1,DepthOut) ; r(Depth1,DepthOut)).

add_complete_search((Head :- Body),(Head1 :- Body1)) :-
```

```
        Head =.. L,
        append(L,[DepthIn,DepthOut],L1),
        Head1 =.. L1,
        (functor(Head,query,_) ->
                add_complete_search_args(Body,DepthIn,DepthOut,Body1);
        nonzero_search_cost(Body,Cost) ->
                add_complete_search_args(Body,Depth1,DepthOut,Body2),
                conjoin((DepthIn >= Cost , Depth1 is DepthIn - Cost),Body2,Body1);
        %true ->
                add_complete_search_args(Body,DepthIn,DepthOut,Body1)).

add_complete_search_args(Body,DepthIn,DepthOut,Body1) :-
        Body = (A , B) ->
                add_complete_search_args(A,DepthIn,Depth1,A1),
                add_complete_search_args(B,Depth1,DepthOut,B1),
                conjoin(A1,B1,Body1);
        Body = (A ; B) ->
                search_cost(A,CostA),
                search_cost(B,CostB),
                (CostA < CostB ->
                        add_complete_search_args(A,DepthIn,DepthOut,A1),
                        add_complete_search_args(B,Depth1,DepthOut,B2),
                        Cost is CostB - CostA,
                        conjoin((DepthIn >= Cost , Depth1 is DepthIn - Cost),B2,B1);
                CostA > CostB ->
                        add_complete_search_args(A,Depth1,DepthOut,A2),
                        add_complete_search_args(B,DepthIn,DepthOut,B1),
                        Cost is CostA - CostB,
                        conjoin((DepthIn >= Cost , Depth1 is DepthIn - Cost),A2,A1);
                %true ->
                        add_complete_search_args(A,DepthIn,DepthOut,A1),
                        add_complete_search_args(B,DepthIn,DepthOut,B1)),
                disjoin(A1,B1,Body1);
        Body = search(Goal,Max,Min,Inc) ->
                PrevInc is Min + 1,
                add_complete_search_args(Goal,DepthIn1,DepthOut1,Goal1),
                DepthIn = DepthOut,
                Body1 = search(Goal1,Max,Min,Inc,PrevInc,DepthIn1,DepthOut1);
        Body = search(Goal,Max,Min) ->
                add_complete_search_args(search(Goal,Max,Min,1),DepthIn,DepthOut,Body1);
        Body = search(Goal,Max) ->
                add_complete_search_args(search(Goal,Max,0),DepthIn,DepthOut,Body1);
        Body = search(Goal) ->
                add_complete_search_args(search(Goal,1000000),DepthIn,DepthOut,Body1);
        functor(Body,search_cost,_) ->
                DepthIn = DepthOut,
                Body1 = true;
        builtin(Body) ->
                DepthIn = DepthOut,
                Body1 = Body;
        %true ->
```

```
                Body =.. L,
                append(L,[DepthIn,DepthOut],L1),
                Body1 =.. L1.

nonzero_search_cost(Body,Cost) :-
        search_cost(Body,Cost),
        Cost > 0.

%%% Search cost is computed by counting literals in the body.
%%% It can be given explicitly instead by including a number, as in
%%%   p :- search_cost(3).     (ordinarily, cost would be 0)
%%%   p :- search_cost(1),q,r. (ordinarily, cost would be 2)
%%%   p :- search_cost(0),s.   (ordinarily, cost would be 1)
%%%
%%% Propositional goals are not counted into the search cost so
%%% that fully propositional problems can be solved without
%%% deepening when iterative-deepening search is used.

search_cost(Body,N) :-
        Body = search_cost(M) ->
                N = M;
        Body = (A , B) ->
                (A = search_cost(M) ->  % if first conjunct is search_cost(M),
                        N = M;          % search cost of the entire conjunction is M
                %true ->
                        search_cost(A,N1),
                        search_cost(B,N2),
                        N is N1 + N2);
        Body = (A ; B) ->
                search_cost(A,N1),
                search_cost(B,N2),
                min(N1,N2,N);
        builtin(Body) ->
                N = 0;
        functor(Body,_,2) ->  % zero-cost 2-ary (0-ary plus ancestor lists) predicates
                N = 0;        % heuristic for propositional problems
        %true ->
                N = 1.


%%% Depth-first iterative-deepening search can be
%%% specified for a goal by wrapping it in a call
%%% on the search predicate:
%%%    search(Goal,Max,Min,Inc)
%%% Max is the maximum depth to search (defaults to a big number),
%%% Min is the minimum depth to search (defaults to 0),
%%% Inc is the amount to increment the bound each time (defaults to 1).
%%%
%%% Depth-first iterative deepening search can be
%%% specified inside the PTTP formula by compiling
%%%   query :- search(p(b,a,c),Max,Min,Inc)
```

```
%%% and executing
%%%   query.
%%% or directly by the user by compiling
%%%   query :- p(b,a,c))
%%% and executing
%%%   search(query,Max,Min,Inc).
%%%
%%% The search(Goal,Max,Min,Inc) predicate adds
%%% DepthIn and DepthOut arguments to its goal argument.

search(Goal,Max,Min,Inc) :-
        PrevInc is Min + 1,
        add_complete_search_args(Goal,DepthIn,DepthOut,Goal1),
        (compile_proof_printing ->
                add_proof_recording_args(Goal1,_Proof,_ProofEnd,Goal2);
        %true ->
                Goal2 = Goal1),
        !,
        search(Goal2,Max,Min,Inc,PrevInc,DepthIn,DepthOut).

search(Goal,Max,Min) :-
        search(Goal,Max,Min,1).

search(Goal,Max) :-
        search(Goal,Max,0).

search(Goal) :-
        search(Goal,1000000).

%%% Actual search driver predicate.
%%% Note that depth-bounded execution of Goal is enabled by
%%% the fact that the DepthIn and DepthOut arguments of
%%% search are also the DepthIn and DepthOut arguments of Goal.

search(_Goal,Max,Min,_Inc,_PrevInc,_DepthIn,_DepthOut) :-
        Min > Max,
        !,
        fail.
search(Goal,_Max,Min,_Inc,PrevInc,DepthIn,DepthOut) :-
        trace_search_progress_pred(P1),
        L1 =.. [P1,Min],
        call(L1),
        DepthIn = Min,
        call(Goal),
        DepthOut < PrevInc.   % fail if this solution was found in previous search
search(Goal,Max,Min,Inc,_PrevInc,DepthIn,DepthOut) :-
        Min1 is Min + Inc,
        search(Goal,Max,Min1,Inc,Inc,DepthIn,DepthOut).
```

```
%%% Complete inference.
%%%
%%% Model elimination reduction operation and
%%% identical ancestor goal pruning.
%%%
%%% Two arguments are added to each literal, one
%%% for all the positive ancestors, one for all
%%% the negative ancestors.
%%%
%%% Unifiable membership is checked in the list
%%% of opposite polarity to the goal
%%% for performing the reduction operation.
%%%
%%% Identity membership is checked in the list
%%% of same polarity as the goal
%%% for performing the ancestor goal pruning operation.
%%% This is not necessary for soundness or completeness,
%%% but is often effective at substantially reducing the
%%% number of inferences.
%%%
%%% The current head goal is added to the front
%%% of the appropriate ancestor list during the
%%% call on subgoals in bodies of nonunit clauses.

add_ancestor((Head :- Body),(Head1 :- Body1)) :-
        functor(Head,query,_) ->
                Head1 = Head,
                add_ancestor_args(Body,[[],[]],Body1);
        %true ->
                Head =.. L,
                append(L,[PosAncestors,NegAncestors],L1),
                Head1 =.. L1,
                add_ancestor_args(Body,[NewPosAncestors,NewNegAncestors],Body2),
                (Body == Body2 ->
                        Body1 = Body2;
                negative_literal(Head) ->
                        NewPosAncestors = PosAncestors,
                        conjoin((NewNegAncestors = [Head|NegAncestors]),Body2,Body1);
                %true ->
                        NewNegAncestors = NegAncestors,
                        conjoin((NewPosAncestors = [Head|PosAncestors]),Body2,Body1)).

add_ancestor_args(Body,AncestorLists,Body1) :-
        Body = (A , B) ->
                add_ancestor_args(A,AncestorLists,A1),
                add_ancestor_args(B,AncestorLists,B1),
                conjoin(A1,B1,Body1);
        Body = (A ; B) ->
                add_ancestor_args(A,AncestorLists,A1),
                add_ancestor_args(B,AncestorLists,B1),
                disjoin(A1,B1,Body1);
```

```
        Body =.. [search,Goal|L] ->
                add_ancestor_args(Goal,AncestorLists,Goal1),
                Body1 =.. [search,Goal1|L];
        builtin(Body) ->
                Body1 = Body;
        %true ->
                Body =.. L,
                append(L,AncestorLists,L1),
                Body1 =.. L1.

ancestor_tests(P,N,Result) :-
        P == query ->
                Result = true;
        %true ->
                negated_functor(P,NotP),
                N2 is N - 2,            % N - 2 due to two ancestor-list arguments
                functor(Head1,P,N2),
                Head1 =.. [P|Args1],
                Head2 =.. [NotP|Args1],
                append(Args1,[PosAncestors,NegAncestors],Args),
                Head =.. [P|Args],
                (negative_functor(P) ->
                        C1Ancestors = NegAncestors, C2Ancestors = PosAncestors;
                %true ->
                        C1Ancestors = PosAncestors, C2Ancestors = NegAncestors),
                C1 = (Head :- identical_member(Head1,C1Ancestors), !, fail),
                count_inferences_pred(IncNcalls),
                (N2 = 0 ->              % special case for propositional calculus
                        conjoin((identical_member(Head2,C2Ancestors) , !),IncNcalls,V);
                %true ->
                        conjoin(unifiable_member(Head2,C2Ancestors),IncNcalls,V)),
                (compile_proof_printing ->
                        conjoin(V,infer_by(red),V1);
                %true ->
                        V1 = V),
                C2 = (Head :- V1),
                conjoin(C1,C2,Result).

procedures_with_ancestor_tests([[P,N]|Preds],Clauses,Procs) :-
        procedure(P,N,Clauses,Proc1),
        ancestor_tests(P,N,Tests),
        conjoin(Tests,Proc1,Proc),
        procedures_with_ancestor_tests(Preds,Clauses,Procs2),
        conjoin(Proc,Procs2,Procs).
procedures_with_ancestor_tests([],_Clauses,true).

identical_member(X,[Y|_])  :-           % run-time predicate for
        X == Y,                         % finding identical ancestor
        !.
identical_member(X,[_|L]) :-
        identical_member(X,L).
```

```
unifiable_member(X,[Y|_]) :-           % run-time predicate for
        unify(X,Y).                    % finding complementary ancestor
unifiable_member(X,[_|L]) :-
        unifiable_member(X,L).


%%% Proof Printing.
%%%
%%% Add extra arguments to each goal so that information
%%% on what inferences were made in the proof can be printed
%%% at the end.

add_proof_recording((Head :- Body),(Head1 :- Body1)) :-
        Head =.. L,
        append(L,[Proof,ProofEnd],L1),
        Head1 =.. L1,
        add_proof_recording_args(Body,Proof,ProofEnd,Body2),
        (functor(Head,query,_) ->
                conjoin(Body2,write_proved(Proof,ProofEnd),Body1);
        %true ->
                Body1 = Body2).

add_proof_recording_args(Body,Proof,ProofEnd,Body1) :-
        Body = (A , B) ->
                add_proof_recording_args(A,Proof,Proof1,A1),
                add_proof_recording_args(B,Proof1,ProofEnd,B1),
                conjoin(A1,B1,Body1);
        Body = (A ; B) ->
                add_proof_recording_args(A,Proof,ProofEnd,A1),
                add_proof_recording_args(B,Proof,ProofEnd,B1),
                disjoin(A1,B1,Body1);
        Body =.. [search,Goal|L] ->
                add_proof_recording_args(Goal,Proof,ProofEnd,Goal1),
                Body1 =.. [search,Goal1|L];
        Body = infer_by(X) ->
                Body1 = (Proof = [X|ProofEnd]);
        Body = fail ->
                Body1 = Body;
        builtin(Body) ->
                Proof = ProofEnd,
                Body1 = Body;
        %true ->
                Body =.. L,
                append(L,[Proof,ProofEnd],L1),
                Body1 =.. L1.

write_proved(Proof,ProofEnd) :-
        write('proved by'),
        write_proof(Proof,ProofEnd).
```

```
write_proof(Proof,ProofEnd) :-
        Proof == ProofEnd,
        !.
write_proof([X|Y],ProofEnd) :-
        write(' '),
        write(X),
        write_proof(Y,ProofEnd).


%%% Negation normal form to Prolog clause translation.
%%% Include a literal in the body of each clause to
%%% indicate the number of the formula the clause came from.

clauses((A , B),L,WffNum) :-
        !,
        clauses(A,L1,WffNum),
        WffNum2 is WffNum + 1,
        clauses(B,L2,WffNum2),
        conjoin(L1,L2,L).
clauses(A,L,WffNum) :-
        head_literals(A,Lits),
        clauses(A,Lits,L,WffNum).

clauses(A,[Lit|Lits],L,WffNum) :-
        body_for_head_literal(Lit,A,Body1),
        (compile_proof_printing ->
                conjoin(infer_by(WffNum),Body1,Body);
        %true ->
                Body = Body1),
        clauses(A,Lits,L1,WffNum),
        conjoin((Lit :- Body),L1,L).
clauses(_,[],true,_).

head_literals(Wff,L) :-
        Wff = (A :- B) ->                % contrapositives are not formed for A :- ... inputs
                head_literals(A,L);
        Wff = (A , B) ->
                head_literals(A,L1),
                head_literals(B,L2),
                union(L1,L2,L);
        Wff = (A ; B) ->
                head_literals(A,L1),
                head_literals(B,L2),
                union(L1,L2,L);
        %true ->
                L = [Wff].

body_for_head_literal(Head,Wff,Body) :-
        Wff = (A :- B) ->
                body_for_head_literal(Head,A,A1),
                conjoin(A1,B,Body);
```

```
        Wff = (A , B) ->
                body_for_head_literal(Head,A,A1),
                body_for_head_literal(Head,B,B1),
                disjoin(A1,B1,Body);
        Wff = (A ; B) ->
                body_for_head_literal(Head,A,A1),
                body_for_head_literal(Head,B,B1),
                conjoin(A1,B1,Body);
        Wff == Head ->
                Body = true;
        negated_literal(Wff,Head) ->
                Body = false;
        %true ->
                negated_literal(Wff,Body).

%%% predicates returns a list of the predicates appearing in a formula.

predicates(Wff,L) :-
        Wff = (A :- B) ->
                predicates(A,L1),
                predicates(B,L2),
                union(L2,L1,L);
        Wff = (A , B) ->
                predicates(A,L1),
                predicates(B,L2),
                union(L2,L1,L);
        Wff = (A ; B) ->
                predicates(A,L1),
                predicates(B,L2),
                union(L2,L1,L);
        functor(Wff,search,_) ->        % list predicates in first argument of search
                arg(1,Wff,X),
                predicates(X,L);
        builtin(Wff) ->
                L = [];
        %true ->
                functor(Wff,F,N),
                L = [[F,N]].

%%% procedure returns a conjunction of the clauses
%%% with head predicate P/N.

procedure(P,N,Clauses,Proc) :-
        Clauses = (A , B) ->
                procedure(P,N,A,ProcA),
                procedure(P,N,B,ProcB),
                conjoin(ProcA,ProcB,Proc);
        (Clauses = (A :- B) , functor(A,P,N)) ->
                Proc = Clauses;
        %true ->
                Proc = true.
```

```
%%% pttp is the PTTP compiler top-level predicate.

pttp(X) :-
        time(pttp1(X),'Compilation').

pttp1(X) :-
        nl,
        write('PTTP input formulas:'),
        apply_to_conjuncts(X,write_clause,_),
        nl,
        clauses(X,X0,1),
        apply_to_conjuncts(X0,add_count_inferences,X1),
        apply_to_conjuncts(X1,add_ancestor,X2),
        predicates(X2,Preds0),
        reverse(Preds0,Preds),
        procedures_with_ancestor_tests(Preds,X2,X3),
        apply_to_conjuncts(X3,add_sound_unification,X4),
        apply_to_conjuncts(X4,add_complete_search,X5),
        (compile_proof_printing ->
                apply_to_conjuncts(X5,add_proof_recording,Y);
        %true ->
                Y = X5),
        nl,
        write('PTTP output formulas:'),
        apply_to_conjuncts(Y,write_clause,_),
        nl,
        nl,

%       File = 'temp.prolog',                    % Quintus Prolog on Sun
        File = 'darwin:>stickel>pttp>temp.prolog',% change file name for other systems

        tell(File),
        apply_to_conjuncts(Y,write_clause,_),
        told,
        compile(File),
        nl,
        !.

query(M) :-                             % call query with depth bound M
        compile_proof_printing ->
                query(M,_N,_Proof,_ProofEnd);
        %true ->
                query(M,_N).

query :-                                % unbounded search of query
        query(1000000).

%%% Utility functions.

%%% Sometimes the `functor' predicate doesn't work as expected and
%%% a more comprehensive predicate is needed.  The `myfunctor'
```

```
%%% predicate overcomes the problem of functor(X,13,0) causing
%%% an error in Symbolics Prolog.  You may need to use it if
%%% 'functor' in your Prolog system fails to construct or decompose
%%% terms that are numbers or constants.

myfunctor(Term,F,N) :-
        nonvar(F),
        atomic(F),
        N == 0,
        !,
        Term = F.
myfunctor(Term,F,N) :-
        nonvar(Term),
        atomic(Term),
        !,
        F = Term,
        N = 0.
myfunctor(Term,F,N) :-
        functor(Term,F,N).

nop(_).

append([X|L1],L2,[X|L3]) :-
        append(L1,L2,L3).
append([],L,L).

reverse([X|L0],L) :-
        reverse(L0,L1),
        append(L1,[X],L).
reverse([],[]).

union([X|L1],L2,L3) :-
        identical_member(X,L2),
        !,
        union(L1,L2,L3).
union([X|L1],L2,[X|L3]) :-
        union(L1,L2,L3).
union([],L,L).

intersection([X|L1],L2,[X|L3]) :-
        identical_member(X,L2),
        !,
        intersection(L1,L2,L3).
intersection([_X|L1],L2,L3) :-
        intersection(L1,L2,L3).
intersection([],_L,[]).

min(X,Y,Min) :-
        X =< Y ->
                Min = X;
        %true ->
```

```
                Min = Y.

conjoin(A,B,C) :-
        A == true ->
                C = B;
        B == true ->
                C = A;
        A == false ->
                C = false;
        B == false ->
                C = false;
        %true ->
                C = (A , B).

disjoin(A,B,C) :-
        A == true ->
                C = true;
        B == true ->
                C = true;
        A == false ->
                C = B;
        B == false ->
                C = A;
        %true ->
                C = (A ; B).

negated_functor(F,NotF) :-
        name(F,L),
        name(not_,L1),
        (append(L1,L2,L) ->
                true;
        %true ->
                append(L1,L,L2)),
        name(NotF,L2).

negated_literal(Lit,NotLit) :-
        Lit =.. [F1|L1],
        negated_functor(F1,F2),
        (var(NotLit) ->
                NotLit =.. [F2|L1];
        %true ->
                NotLit =.. [F2|L2],
                L1 == L2).

negative_functor(F) :-
        name(F,L),
        name(not_,L1),
        append(L1,_,L).

negative_literal(Lit) :-
        functor(Lit,F,_),
```

```
        negative_functor(F).

apply_to_conjuncts(Wff,P,Wff1) :-
        Wff = (A , B) ->
                apply_to_conjuncts(A,P,A1),
                apply_to_conjuncts(B,P,B1),
                conjoin(A1,B1,Wff1);
        %true ->
                T1 =.. [P,Wff,Wff1],
                call(T1).

write_clause(A) :-
        nl,
        write(A),
        write(.).

write_clause(A,_) :-                    % 2-ary predicate can be used as
        write_clause(A).                % argument of apply_to_conjuncts

%%% Inference counting is turned on by count_inferences,
%%% off by dont_count_inferences.
%%%
%%% Inferences are counted by retracting the current count
%%% and asserting the incremented count, so inference counting
%%% is very slow.

count_inferences :-                     % enables compilation of inference counting
        retract(count_inferences_pred(_)),% this slows down the code substantially
        fail.
count_inferences :-
        assert(count_inferences_pred(inc_ncalls)).

dont_count_inferences :-                % disables compilation of inference counting
        retract(count_inferences_pred(_)),% this is the default for acceptable performance
        fail.
dont_count_inferences :-
        assert(count_inferences_pred(true)).

:- dont_count_inferences.               % default is to not count inferences

%%% Transformation to add inference counting to a clause.

add_count_inferences((Head :- Body),(Head :- Body1)) :-
        functor(Head,query,_) ->
                Body1 = Body;
        %true ->
                count_inferences_pred(P),
                conjoin(P,Body,Body1).

clear_ncalls :-
        retract(ncalls(_)),
```

```
        fail.
clear_ncalls :-
        assert(ncalls(0)).

inc_ncalls :-
        retract(ncalls(N)),
        N1 is N + 1,
        assert(ncalls(N1)),
        !.

%%% Search tracing is turned on by trace_search,
%%% off by dont_trace_search.

trace_search :-                         % enables search progress reports
        retract(trace_search_progress_pred(_)),
        fail.
trace_search :-
        assert(trace_search_progress_pred(write_search_progress)).

dont_trace_search :-                    % disables search progress reports
        retract(trace_search_progress_pred(_)),
        fail.
dont_trace_search :-
        assert(trace_search_progress_pred(nop)).

:- trace_search.                        % default is to trace searching

write_search_progress(Level) :-
        ncalls(N),
        (N > 0 -> write(N) , write(' inferences so far.') ; true),
        nl,
        write('Begin cost '),
        write(Level),
        write(' search...  ').

%%% Proof printing is turned on by print_proof,
%%% off by dont_print_proof.
%%%
%%% print_proof or dont_print_proof should be
%%% executed before the problem is compiled.

print_proof :-                          % enable proof printing
        retract(compile_proof_printing),
        fail.
print_proof :-
        assert(compile_proof_printing).

dont_print_proof :-                     % disable proof printing
        retract(compile_proof_printing),
        fail.
dont_print_proof.
```

```
:- print_proof.                          % default is to print proof

%%% A query can be timed by time(query).

time(X) :-
        time(X,'Execution').

time(X,Type) :-
        clear_ncalls,

%       statistics(runtime,[T1,_]),     % Quintus Prolog on Sun
        T1 is œget-internal-run-timeœ,  % Common Lisp time function

        call(X),

%       statistics(runtime,[T2,_]),     % Quintus Prolog on Sun
%       Secs is (T2 - T1) / 1000.0,     % Quintus measures runtime in milliseconds
        T2 is œget-internal-run-timeœ,  % Common Lisp time function
        Secs is (T2 - T1) / 977.0,      % internal-time-units-per-second on Darwin

        nl,
        write(Type),
        write(' time: '),
        ncalls(N),
        (N > 0 -> write(N) , write(' inferences in ') ; true),
        write(Secs),
        write(' seconds, including printing'),
        nl.

%%% List of builtin predicates that can appear in clause bodies.
%%% No extra arguments are added for ancestor goals or depth-first
%%% iterative-deepening search.  Also, if a clause body is
%%% composed entirely of builtin goals, the head is not saved
%%% as an ancestor for use in reduction or pruning.
%%% This list can be added to as required.

builtin(T) :-
        functor(T,F,N),
        builtin(F,N).

builtin(!,0).
builtin(true,0).
builtin(fail,0).
builtin(succeed,0).
builtin(trace,0).
builtin(atom,1).
builtin(integer,1).
builtin(number,1).
builtin(atomic,1).
builtin(constant,1).
```

```
builtin(functor,3).
builtin(arg,3).
builtin(var,1).
builtin(nonvar,1).
builtin(call,1).
builtin(=,2).
builtin(\=,2).
builtin(==,2).
builtin(\==,2).
builtin(>,2).
builtin(<,2).
builtin(>=,2).
builtin(=<,2).
builtin(is,2).
builtin(display,1).
builtin(write,1).
builtin(nl,0).
builtin(infer_by,_).
builtin(write_proved,_).
builtin(search,_).
builtin(search_cost,_).
builtin(unify,_).
builtin(identical_member,_).
builtin(unifiable_member,_).
builtin(inc_ncalls,0).
```

```
%%% Theorem proving examples from
%%%   Chang, C.L. and R.C.T. Lee.
%%%   Symbolic Logic and Mechanical Theorem Proving.
%%%   Academic Press, New York, 1973, pp. 298-305.

%%% Note that the search driver predicate
%%% can be invoked by search(query) as in
%%% chang_lee_example8 or can be explicitly
%%% included in the query as in chang_lee_example2.

chang_lee_example2 :-
        nl,
        write(chang_lee_example2),
        pttp((
                p(e,X,X),
                p(X,e,X),
                p(X,X,e),
                p(a,b,c),
                (p(U,Z,W) :- p(X,Y,U) , p(Y,Z,V) , p(X,V,W)),
                (p(X,V,W) :- p(X,Y,U) , p(Y,Z,V) , p(U,Z,W)),
                (query :- search(p(b,a,c)))
        )),
        time(query).

chang_lee_example8 :-
        nl,
        write(chang_lee_example8),
        pttp((
                l(1,a),
                d(X,X),
                (p(X) ; d(g(X),X)),
                (p(X) ; l(1,g(X))),
                (p(X) ; l(g(X),X)),
                (not_p(X) ; not_d(X,a)),
                (not_d(X,Y) ; not_d(Y,Z) ; d(X,Z)),
                (not_l(1,X) ; not_l(X,a) ; p(f(X))),
                (not_l(1,X) ; not_l(X,a) ; d(f(X),X)),
                (query :- (p(X) , d(X,a)))
        )),
        time(search(query)).
```

B PTTP Compiler Sample Input and Output

The following is an example of the input and output of the PTTP-to-Prolog compiler operating on a problem that requires the PTTP transformations for sound unification, complete search, and complete inference. The transformation to add information to enable proof printing was not used for this output. This is Example 8 from Chang and Lee [3], pp. 298-305, for which statistics are presented in Table 1. See Appendix C for a description of the problem and its proof. Note that wff 6 duplicates the query; it is included to allow for the discovery of indefinite answers [17].

PTTP input formulas:

```
l(1,a).                                         % wff 1
d(X,X).                                         % wff 2
p(X) ; d(g(X),X).                               % wff 3
p(X) ; l(1,g(X)).                               % wff 4
p(X) ; l(g(X),X).                               % wff 5
not_p(X) ; not_d(X,a).                          % wff 6
not_d(X,Y) ; not_d(Y,Z) ; d(X,Z).               % wff 7
not_l(1,X) ; not_l(X,a) ; p(f(X)).              % wff 8
not_l(1,X) ; not_l(X,a) ; d(f(X),X).            % wff 9
query :- p(X) , d(X,a).                         % wff 10
```

PTTP output formulas:
Procedure l clauses:

```
l(X,Y,PosAnc,NegAnc,Depth,Depth) :-             % pruning by ancestor
    identical_member(l(X,Y),PosAnc),            % operation for ‘l’
    !, fail.                                    % aborts repeated goals
l(X,Y,PosAnc,NegAnc,Depth,Depth) :-             % ME reduction operation
    unifiable_member(not_l(X,Y),NegAnc).        % for ‘l’
l(1,a,PosAnc,NegAnc,Depth,Depth).               % clause from wff 1
l(1,g(X),PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 4
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [l(1,g(X)) | PosAnc],           % save head goal as ancestor
    not_p(X,NewPosAnc,NegAnc,D1,DepthOut).      % solve the subgoal
l(g(X),X1,PosAnc,NegAnc,DepthIn,DepthOut) :-    % clause from wff 5
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    unify(X,X1),                                % unify with occurs check
    NewPosAnc = [l(g(X),X) | PosAnc],           % save head goal as ancestor
    not_p(X,NewPosAnc,NegAnc,D1,DepthOut).      % solve the subgoal
```

Procedure `d` clauses:

```
d(X,Y,PosAnc,NegAnc,Depth,Depth) :-             % pruning by ancestor
    identical_member(d(X,Y),PosAnc),            % operation for ‘d’
    !, fail.                                    % aborts repeated goals
d(X,Y,PosAnc,NegAnc,Depth,Depth) :-             % ME reduction operation
    unifiable_member(not_d(X,Y),NegAnc).        % for ‘d’
d(X,X1,PosAnc,NegAnc,Depth,Depth) :-            % clause from wff 2
    unify(X,X1).                                % unify with occurs check
d(g(X),X1,PosAnc,NegAnc,DepthIn,DepthOut) :-    % clause from wff 3
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    unify(X,X1),                                % unify with occurs check
    NewPosAnc = [d(g(X),X) | PosAnc],           % save head goal as ancestor
    not_p(X,NewPosAnc,NegAnc,D1,DepthOut).      % solve the subgoal
d(X,Z,PosAnc,NegAnc,DepthIn,DepthOut) :-        % clause from wff 7
    DepthIn >= 2, D1 is DepthIn - 2,            % test and decr. depth bound
    NewPosAnc = [d(X,Z) | PosAnc],              % save head goal as ancestor
    d(X,Y,NewPosAnc,NegAnc,D1,D2),              % solve first subgoal
    d(Y,Z,NewPosAnc,NegAnc,D2,DepthOut).        % solve second subgoal
d(f(X),X1,PosAnc,NegAnc,DepthIn,DepthOut) :-    % clause from wff 9
    DepthIn >= 2, D1 is DepthIn - 2,            % test and decr. depth bound
    unify(X,X1),                                % unify with occurs check
    NewPosAnc = [d(f(X),X) | PosAnc],           % save head goal as ancestor
    l(1,X,NewPosAnc,NegAnc,D1,D2),              % solve first subgoal
    l(X,a,NewPosAnc,NegAnc,D2,DepthOut).        % solve second subgoal
```

Procedure `p` clauses:

```
p(X,PosAnc,NegAnc,Depth,Depth) :-               % pruning by ancestor
    identical_member(p(X),PosAnc),              % operation for ‘p’
    !, fail.                                    % aborts repeated goals
p(X,PosAnc,NegAnc,Depth,Depth) :-               % ME reduction operation
    unifiable_member(not_p(X),NegAnc).          % for ‘p’
p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-          % clause from wff 3
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [p(X) | PosAnc],                % save head goal as ancestor
    not_d(g(X),X,NewPosAnc,NegAnc,D1,DepthOut). % solve the subgoal
p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-          % clause from wff 4
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [p(X) | PosAnc],                % save head goal as ancestor
    not_l(1,g(X),NewPosAnc,NegAnc,D1,DepthOut). % solve the subgoal
p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-          % clause from wff 5
    DepthIn >= 1, D1 is DepthIn - 1,            % test and decr. depth bound
    NewPosAnc = [p(X) | PosAnc],                % save head goal as ancestor
    not_l(g(X),X,NewPosAnc,NegAnc,D1,DepthOut). % solve the subgoal
p(f(X),PosAnc,NegAnc,DepthIn,DepthOut) :-       % clause from wff 8
    DepthIn >= 2, D1 is DepthIn - 2,            % test and decr. depth bound
    NewPosAnc = [p(f(X)) | PosAnc],             % save head goal as ancestor
    l(1,X,NewPosAnc,NegAnc,D1,D2),              % solve first subgoal
    l(X,a,NewPosAnc,NegAnc,D2,DepthOut).        % solve second subgoal
```

Procedure `not_d` clauses:

```
not_d(X,Y,PosAnc,NegAnc,Depth,Depth) :-          % pruning by ancestor
    identical_member(not_d(X,Y),NegAnc),         % operation for ‘not_d’
    !, fail.                                     % aborts repeated goals
not_d(X,Y,PosAnc,NegAnc,Depth,Depth) :-          % ME reduction operation
    unifiable_member(d(X,Y),PosAnc).             % for ‘not_d’
not_d(X,a,PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 6
    DepthIn >= 1, D1 is DepthIn - 1,             % test and decr. depth bound
    NewNegAnc = [not_d(X,a) | NegAnc],           % save head goal as ancestor
    p(X,PosAnc,NewNegAnc,D1,DepthOut).           % solve the subgoal
not_d(X,Y,PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 7
    DepthIn >= 2, D1 is DepthIn - 2,             % test and decr. depth bound
    NewNegAnc = [not_d(X,Y) | NegAnc],           % save head goal as ancestor
    d(Y,Z,PosAnc,NewNegAnc,D1,D2),               % solve first subgoal
    not_d(X,Z,PosAnc,NewNegAnc,D2,DepthOut).     % solve second subgoal
not_d(Y,Z,PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 7
    DepthIn >= 2, D1 is DepthIn - 2,             % test and decr. depth bound
    NewNegAnc = [not_d(Y,Z) | NegAnc],           % save head goal as ancestor
    d(X,Y,PosAnc,NewNegAnc,D1,D2),               % solve first subgoal
    not_d(X,Z,PosAnc,NewNegAnc,D2,DepthOut).     % solve second subgoal
```

Procedure **not_p** clauses:

```
not_p(X,PosAnc,NegAnc,Depth,Depth) :-            % pruning by ancestor
    identical_member(not_p(X),NegAnc),           % operation for ‘not_p’
    !, fail.                                     % aborts repeated goals
not_p(X,PosAnc,NegAnc,Depth,Depth) :-            % ME reduction operation
    unifiable_member(p(X),PosAnc).               % for ‘not_p’
not_p(X,PosAnc,NegAnc,DepthIn,DepthOut) :-       % clause from wff 6
    DepthIn >= 1, D1 is DepthIn - 1,             % test and decr. depth bound
    NewNegAnc = [not_p(X) | NegAnc],             % save head goal as ancestor
    d(X,a,PosAnc,NewNegAnc,D1,DepthOut).         % solve the subgoal
```

Procedure **not_l** clauses:

```
not_l(X,Y,PosAnc,NegAnc,Depth,Depth) :-          % pruning by ancestor
    identical_member(not_l(X,Y),NegAnc),         % operation for ‘not_l’
    !, fail.                                     % aborts repeated goals
not_l(X,Y,PosAnc,NegAnc,Depth,Depth) :-          % ME reduction operation
    unifiable_member(l(X,Y),PosAnc).             % for ‘not_l’
not_l(1,X,PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 8
    DepthIn >= 2, D1 is DepthIn - 2,             % test and decr. depth bound
    NewNegAnc = [not_l(1,X) | NegAnc],           % save head goal as ancestor
    l(X,a,PosAnc,NewNegAnc,D1,D2),               % solve first subgoal
    not_p(f(X),PosAnc,NewNegAnc,D2,DepthOut).    % solve second subgoal
not_l(X,a,PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 8
    DepthIn >= 2, D1 is DepthIn - 2,             % test and decr. depth bound
    NewNegAnc = [not_l(X,a) | NegAnc],           % save head goal as ancestor
    l(1,X,PosAnc,NewNegAnc,D1,D2),               % solve first subgoal
    not_p(f(X),PosAnc,NewNegAnc,D2,DepthOut).    % solve second subgoal
not_l(1,X,PosAnc,NegAnc,DepthIn,DepthOut) :-     % clause from wff 9
    DepthIn >= 2, D1 is DepthIn - 2,             % test and decr. depth bound
```

```
    NewNegAnc = [not_l(1,X) | NegAnc],          % save head goal as ancestor
    l(X,a,PosAnc,NewNegAnc,D1,D2),              % solve first subgoal
    not_d(f(X),X,PosAnc,NewNegAnc,D2,DepthOut). % solve second subgoal
not_l(X,a,PosAnc,NegAnc,DepthIn,DepthOut) :-    % clause from wff 9
    DepthIn >= 2, D1 is DepthIn - 2,            % test and decr. depth bound
    NewNegAnc = [not_l(X,a) | NegAnc],          % save head goal as ancestor
    l(1,X,PosAnc,NewNegAnc,D1,D2),              % solve first subgoal
    not_d(f(X),X,PosAnc,NewNegAnc,D2,DepthOut). % solve second subgoal
```

Procedure **query** clause:

```
query(DepthIn,DepthOut) :-                      % clause from query wff 10
    p(X,[],[],DepthIn,D1),                      % solve first subgoal
    d(X,a,[],[],D1,DepthOut).                   % solve second subgoal
```

C PTTP Sample Proof

Following is a sample model elimination proof found by PTTP. Because the Prolog version of PTTP is incapable of fully printing proofs, this proof was produced by the Lisp version of PTTP. This is Example 8 from Chang and Lee [3], pp. 298-305, for which statistics are presented in Table 1. The PTTP compiler output for this example is shown in Appendix B.

The special literal `query` is used to specify the initial goal in the proof attempt. The literal `search((p(X) , d(X,a)))` attempts to solve the goals `p(X)` and `d(X,a)` by using depth-first iterative-deepening search; the conjoined cut operation `!` discontinues the seach after the first solution is found.

A clause-by-clause description of the input is as follows: (1) a is greater than 1; (2) x divides x; (3) if x is not prime, then it has a divisor $g(x)$ that is (4) greater than 1 and (5) less than x; (6) the negation of the theorem, necessary when seeking indefinite answers; (7) if x divides y, and y divides z, then x divides z; (8) the induction hypothesis that for all x between 1 and a there is a prime $f(x)$ that (9) divides x; (10) the theorem that a has a prime divisor.

```
  1.  l(1,a).
  2.  d(X,X).
  3.  p(X) ; d(g(X),X).
  4.  p(X) ; l(1,g(X)).
  5.  p(X) ; l(g(X),X).
  6.  not_p(X) ; not_d(X,a).
  7.  not_d(X,Y) ; not_d(Y,Z) ; d(X,Z).
  8.  not_l(1,X) ; not_l(X,a) ; p(f(X)).
  9.  not_l(1,X) ; not_l(X,a) ; d(f(X),X).
 ----------------
 10.  query :- search((p(X) , d(X,a))) , !.
```

```
                                                     Begin cost   0 search.
End cost   0 search.          0 inferences so far.  Begin cost   1 search.
End cost   1 search.          3 inferences so far.  Begin cost   2 search.
End cost   2 search.          9 inferences so far.  Begin cost   3 search.
End cost   3 search.         27 inferences so far.  Begin cost   4 search.
End cost   4 search.         57 inferences so far.  Begin cost   5 search.
End cost   5 search.        118 inferences so far.  Begin cost   6 search.
End cost   6 search.        212 inferences so far.  Begin cost   7 search.
End cost   7 search.        405 inferences so far.  Begin cost   8 search.
End cost   8 search.        700 inferences so far.  Begin cost   9 search.
End cost   9 search.      1,317 inferences so far.  Begin cost  10 search.
```

```
End cost  10 search.      2,291 inferences so far.  Begin cost  11 search.

Proof:
Goal#  Wff#  Wff Instance
-----  ----  ------------
(  0)   10   query :- p(a) , d(a,a).
(  1)    4a     p(a) :- not_l(1,g(a)).
(  2)    8a        not_l(1,g(a)) :- l(g(a),a) , not_p(f(g(a))).
(  3)    5b           l(g(a),a) :- not_p(a).
(  4)                    not_p(a).
(  5)    6a           not_p(f(g(a))) :- d(f(g(a)),a).
(  6)    7c              d(f(g(a)),a) :- d(f(g(a)),g(a)) , d(g(a),a).
(  7)    9c                 d(f(g(a)),g(a)) :- l(1,g(a)) , l(g(a),a).
(  8)                          l(1,g(a)).
(  9)    5b                    l(g(a),a) :- not_p(a).
( 10)                             not_p(a).
( 11)    3b                 d(g(a),a) :- not_p(a).
( 12)                          not_p(a).
( 13)    2      d(a,a).
End cost  11 search.      3,830 inferences so far.  Search ended by cut.
```

The proof is printed as a list of the final instantiations of the clauses that are used in each proof step. The initial clause is `query :- p(a) , d(a,a)`. Its subgoals are `p(a)` and `d(a,a)` whose solutions start on lines (1) and (13). Indentation is used to help identify subgoal relationships.

In this proof, lines (4), (8), (10), and (12) show subgoals being solved by the reduction operation. In particular, the goals `not_p(a)` of lines (4), (10), and (12) match the complement of their ancestor goal `p(a)` in line (1), while the goal `l(1,g(a))` of line (8) matches the complement of its ancestor goal `not_l(1,g(a))` in line (2).

Examination of the proof shows clauses 10 and 6, the theorem and its negation, each appearing once in the proof. The instantiations used reveal the answer to be that either (a) a is prime and a divides a or (b) $f(g(a))$, a prime divisor of a divisor of a, divides a.

This problem requires all of PTTP's extensions of Prolog: sound unification, complete search, the reduction operation, and indefinite answers.

The output of the Prolog version of PTTP for this example with inference counting, search tracing, and proof printing enabled follows.[9] The full proof is not printed, but

[9]The execution time shown here, unlike in Table 1, is poor because of the cost of printing and inference counting, especially the latter, which requires `assert` and `retract` operations.

the clause number or `red` (denoting the reduction operation) is printed for each inference operation. Information about which literal of the clause was resolved on and the variable bindings is unavailable.

```
Begin cost 0 search...
Begin cost 1 search...  3 inferences so far.
Begin cost 2 search...  9 inferences so far.
Begin cost 3 search...  27 inferences so far.
Begin cost 4 search...  57 inferences so far.
Begin cost 5 search...  118 inferences so far.
Begin cost 6 search...  212 inferences so far.
Begin cost 7 search...  405 inferences so far.
Begin cost 8 search...  700 inferences so far.
Begin cost 9 search...  1317 inferences so far.
Begin cost 10 search...  2291 inferences so far.
Begin cost 11 search...  proved by 10 4 8 5 red 6 7 9 red 5 red 3 red 2
Execution time: 3830 inferences in 15.784033 seconds, including printing
```