A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler[1]

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

November 30, 1987

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, the model-elimination reduction rule that is added to Prolog inferences to make the inference system complete, and depth-first iterative-deepening search instead of unbounded depth-first search to make the search strategy complete. A Prolog technology theorem prover has been implemented by an extended Prolog-to-LISP compiler that supports these additional features. It is capable of proving theorems in the full first-order predicate calculus at a rate of thousands of inferences per second.

[1]This is a revised and expanded version of a paper presented at the *8th International Conference on Automated Deduction*, Oxford, England, July 1986, and is to appear in *Journal of Automated Reasoning*.

This research was supported by the Defense Advanced Research Projects Agency under Contract N00039-84-K-0078 with the Naval Electronic Systems Command and 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 Defense Advanced Research Projects Agency, the National Science Foundation, or the United States government. Approved for public release. Distribution unlimited.

1 Introduction

Despite Prolog's logic heritage and its use of theorem-proving unification and resolution operations, Prolog fails to qualify as a full general-purpose theorem-proving system. There are three main reasons for this:

- Many Prolog systems use an unsound unification algorithm
- Prolog's inference system is not complete for non-Horn clauses
- Prolog's unbounded depth-first search strategy is incomplete.

Nevertheless, Prolog is quite interesting from a theorem-proving standpoint because of its very high speed as compared with conventional theorem-proving programs. The objective of a *Prolog technology theorem prover (PTTP)* is to remedy the above deficiencies while retaining as fully as possible the high performance of well-engineered Prolog systems. PTTP also extends Prolog by providing the capability of printing the proofs it finds and by being able to limit the size of the search space somewhat.

Our current effort to secure the advantages of Prolog for general-purpose theorem proving is the construction of an extended Prolog compiler. This process yields a few thousand inferences per second (lips) for general-purpose theorem proving on the Symbolics 3600 LISP machine.

2 Implementation

Our PTTP extended Prolog-to-LISP compiler is written in COMMON LISP for the Symbolics 3600 LISP machine. It translates Prolog procedures into COMMON LISP[2] functions

[2]Two nonstandard features of Symbolics COMMON LISP are used: (1) The `stack-let` macro is employed instead of the `let` special form in some places. This permits PTTP to run with all consing done on the stack rather than in the heap, reduces execution time, and eliminates the need for garbage collection. (2) Symbolics debugger functions are used to examine the stack upon completion of a proof in order to print it. This reduces the overhead required to retain the information necessary to print the proof. When running on machines other than the 3600, the usual `let` special form is used and proof printing is impossible.

that are then compiled by the LISP compiler. Being written in COMMON LISP, PTTP can be run on other computers that support COMMON LISP, such as Sun workstations and Macintosh personal computers. However, performance on other computers may suffer, since it was developed and tuned for 3600s and not these other machines.

We shall not describe the implementation in great detail, since many of its aspects are more matters of expediency than optimal design. Nevertheless, the current implementation is sufficient to demonstrate the value of the ideas—namely, that complete theorem provers with unprecedented inference rates can be implemented.

Each Prolog procedure (a list of clauses with the same predicate in the head literal) is translated by our compiler into a single LISP function, to which is passed the procedure's arguments and a continuation. The procedure

```
p(args1).
p(args2) <- q(...).
p(args3) <- r(...), s(...).
```

is translated into something like

```
function p (args,cont);
   begin
   if unify(args,args1) then cont();
   undo-unify;
   if unify(args,args2) then q(...,cont);
   undo-unify;
   if unify(args,args3) then r(...,s(...,cont));
   undo-unify
   end
```

This approach to Prolog compilation is also described by Cohen [9].

We discuss below Prolog's deficiencies for general-purpose theorem proving and examine the manner in which they are dealt with by the current PTTP implementation.

3 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 whose successor is less than itself and that there is a person that is his own parent:

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

parent(X,mother(X)).
<- parent(Y,Y).
```

The invalid results rely upon the creation of circular bindings for `X` and `Y` during unification. Besides leading to incorrect results, circular bindings may lead to nonterminating unification. If the values of `X` and `Y` in the first example are unified later, the unification would not terminate unless a unification algorithm capable of handling infinite terms were used [10].

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.

Accordingly, we simply compile occurs checks in except when they are obviously unnecessary. During unification of the actual and formal arguments, which are initially variable-disjoint, the first binding of a variable is guaranteed not to need the occurs check; only when a second occurrence of a variable is seen does it become necessary to compile in an occurs check. In the special case where no variables are repeated in one of the terms, i.e., when the term is *linear*, no occurs check will be necessary for the entire unification operation.

There are alternative methods to assuring sound unification. Plaisted [28] has suggested an elegant method, which we currently use in another version of PTTP, of transforming clauses to isolate parts that may require unification with the occurs check. Repeated occurrences of variables are replaced by new variables so as to make the clause head linear. Matching 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 in the transformed clause body.

In the above examples, the clauses `X<(X+1)` and `parent(X,mother(X))` are replaced by the clauses `X<(X1+1) <- unify(X,X1)` and `parent(X,mother(X1)) <- unify(X,X1)`, in which the occurs check needs to be performed only during the unification of `X` and `X1`.

This transformation makes it easy to incorporate sound unification into Prolog systems that lack it. A new built-in predicate `unify` that performs sound unification must be added, but no changes to the Prolog-virtual-machine instruction set are strictly necessary.

An alternative approach is to allow creation of circular bindings by an always terminating unification algorithm and to check at some point whether the bindings are circular. This latter check can be performed immediately or be delayed, for example, until a possible proof has been completed. Delaying the check may substantially reduce the cost of unification at the risk of allowing many inferences to be drawn after a circular term is created—inferences that could have been cut off by checking immediately. We have no data on the trade-off between the cost of the occurs check and the amount of search saved.

4 Complete Inference System

Prolog's inference system is complete for Horn sets of clauses, i.e., sets of clauses such that no clause has more than a single positive literal. In developing a Prolog technology theorem prover, the inference system must be extended to make it complete for non-Horn sets of clauses as well.

However, one should consider only those means for extending Prolog's inference system that permit highly efficient Prolog implementation techniques. Some of the most important factors contributing to the high speed of well-engineered Prolog implementations are compilation and efficient representations for derived clauses and variable substitutions.

Prolog, using depth-first search and input resolution (in which one of the two clauses resolved upon must be an input clause), needs to represent only a single derived clause at a time. Variable binding during unification is accomplished by undoable destructive assignment.

Two methods for handling substitutions are usually employed in conventional resolution theorem proving. The simple method is to form resolvents fully by applying the unifying substitution to the parent clauses. This is far more expensive in both time and space than Prolog inference.

The second less frequently used method involves *structure sharing* [6], in which a resolvent is represented by the parents plus the unifying substitution. Whenever the resolvent must be examined (e.g., for printing or resolution with another clause), it is traversed, with variables being implicitly replaced by their substitution values. This is still less efficient than the method employed in Prolog; resolution requires trees of variable-binding environments, whereas Prolog requires only linear lists.

The use of input resolution also facilitates the compilation of Prolog programs. In input resolution, there is a given set of input clauses such that (ignoring run-time assertions) these clauses are always used as one of the two inputs to each resolution operation. It is thus quite natural and effective to compile this given set of input clauses. It is more difficult and expensive to use compilation in more general forms of resolution, since derived clauses can be resolved with one another and there is consequently no fixed set of clauses to compile.

All this suggests that a good approach to building a PTTP is to employ a complete inference system that is also an input procedure. Probably the simplest is [an affirmative form of] the *model elimination (ME)* procedure [16, 17, 20].[3]

The ME procedure requires only the addition of the following inference operation to Prolog to constitute a complete inference system for the first-order predicate calculus:

> If the current goal matches the complement of one of its ancestor goals, then apply the matching substitution and treat the current goal as if it were solved.

This added inference operation is the ME *reduction* operation. The normal Prolog inference operation is the ME *extension* operation. The two together comprise a complete inference system for the full first-order predicate calculus.

[3]Actually, what we are describing here is more closely related to the problem-reduction-oriented MESON procedure [20, 17], but we will use the term model elimination (ME) because it is more familiar and because the MESON procedure is derived from the ME procedure.

The reduction operation provides an extra method for solving a goal in addition to the standard Prolog methods. Just as solution of a goal by matching it against a unit clause in Prolog does not in general preclude the need for considering alternative solutions, successful application of the reduction operation cannot be used to eliminate attempts to solve the goal by other methods, including use of the reduction operation with a different ancestor goal (in both cases, alternative solutions do not have to be considered if the matching substitution is empty, i.e., does not instantiate any variables, since the goal will have been solved in a most general way).

The reduction operation is a form of reasoning by contradiction. If, in trying to prove P, we discover that P is true if Q is true (i.e., $Q \supset P$) and also that Q is true if $\sim P$ is true (i.e., $\sim P \supset Q$), then P must be true. The rationale is that P is either true or false; if we assume that P is false, then Q must be true and hence P must also be true, which is a contradiction; therefore, the hypothesis that P is false must be wrong and P must be true.

Note that although exiting a goal in Prolog means that the goal, instantiated by the current substitution, has been proved, in PTTP, when a goal is exited, all that has been proved is its instantiation disjoined with all the ancestor goals used in reduction operations in the solution of the goal. Thus, in the example of proving P from $Q \supset P$ and $\sim P \supset Q$, expressed by

```
p <- q.
q <- ~p.

<- p.
```

when goal `q` is exited, $P \vee Q$, but not Q, has been proved. The top goal `p`, when exited, has been proved; there are no ancestor goals whose negation could have been assumed in the attempt to prove the top goal.

In Prolog, when a goal is entered, a choice point is established at which the alternatives are matching the goal with the heads of all the clauses and, if the match is successful, executing the body of the clause. In PTTP, it is also necessary to consider the additional alternatives of matching the entered goal with the complement of each of its ancestor goals.

For each such successful match, we proceed in the same manner as if we had matched the goal with the head of a unit clause.

The reduction rule is implemented by maintaining and using lists of the current ancestor goals. Ancestor goals are efficiently indexed by predicate symbol and negation by keeping separate lists in global variables for each predicate symbol or negated predicate symbol. The compiled code for a procedure then includes code that maintains this list by pushing the current goal onto it before execution of the body of nonunit clauses and popping it afterwards. To perform the reduction operation, the code unifies the procedure's arguments with elements of the list of ancestor goals whose predicate is the complement of the procedure's predicate and executes the continuation for each successful match.

PTTP, unlike Prolog, requires contrapositives of the assertions to be supplied. For each clause with n literals, n Prolog-like assertions must be provided so that each literal will be the head of one of the Prolog assertions. Thus, the clause $P(a) \vee P(b)$ is encoded as the two Prolog assertions `p(a) <- ~p(b)` and `p(b) <- ~p(a)`. PTTP can generate the contrapositives automatically.

An important thing to note is that this is a complete inference system that does not require the *factoring* operation. Basing an extension of Prolog on another form of model elimination, equivalent to SL-resolution [14], would necessitate an additional factoring operation that instantiates pairs of goals to be identical.

For several reasons we regard factoring as an undesirable operation to add. Adding another inference operation requires further decision-making about how to order possible inference operations. Unlike the extension and reduction operations that operate on the current goal and on an input clause or an ancestor goal that is available on the stack, the factoring operation operates on the current goal and an unsolved pending subgoal of an ancestor goal that is not itself an ancestor goal and thus is not so readily available on the stack. Although the factoring operation is theoretically necessary for the completeness of many inference systems, it is rarely useful.

The extension of Prolog to full-first order predicate calculus introduces a feature that is absent in Prolog deduction: indefinite answers. Prolog and PTTP can compute answers to queries as well as determine their truth. When provided with the goal $P(x)$, they will attempt to find terms t such that $P(t)$ is true. In non-Horn clause theorem proving, however, there may be indefinite answers. For example, in proving $\exists x\ P(x)$ from $P(a) \vee P(b)$, there is no single term t for which it is known that $P(t)$ is true.

The example can be expressed in PTTP as

```
p(a) <- ~p(b).
p(b) <- ~p(a).

<- p(X).
```

This set of assertions and the described inference procedure are still insufficient to solve the problem. Trying to solve `p(X)` by the first rule results in the subgoal `~p(b)`, which does not match any clause head or negated ancestor goal (the ancestor `p(X)` having been instantiated to `p(a)` by the rule application). Trying to solve it by the second rule also fails. To solve problems with indefinite answers, it is necessary to add the negation of the query as another assertion (n contrapositive assertions if the query has n literals).

In this example, addition of the Prolog assertion `~p(Y)` results in the finding of two proofs (`p(X)` can be matched with `p(a)` and `~p(Y)` with `~p(b)`, and vice versa for the second proof).[4] These proofs can be taken to compute the conditional answers that if $P(b)$ is not true then $P(a)$ is and if $P(a)$ is not true then $P(b)$ is (i.e., $\neg P(b) \supset P(a)$ and $\neg P(a) \supset P(b)$, which are both equivalent to $P(a) \vee P(b)$). In other words, either $P(a)$ or $P(b)$ (or both) is true, but neither $P(a)$ nor $P(b)$ has been proved. In general, indefinite answers are disjunctions of instances of the query. One instance of the query is included for each use of the query in the deduction (i.e., its use as the initial list of goals and each use of its negation).

PTTP can thus be used to derive either definite or indefinite answers. As in Prolog, definite answers can be derived by simply solving a query. Indefinite answers can be obtained

[4]Note that no matter how many alternatives may appear in an indefinite answer (e.g., four in the case of proving $\exists x\ P(x)$ from $P(a) \vee P(b) \vee P(c) \vee P(d)$), only a single occurrence of the goal's negation `~p(Y)` needs to be added, since any single assertion can always be used arbitrarily many times with different instantiations.

by solving the query with its negation included among the axioms and examining the proof to find the query's instantiations.

Unfortunately, because the derivation of indefinite answers requires inclusion of the query's negation among the axioms, an otherwise static assertional database may have to be modified and recompiled when indefinite answers are sought.

Finally we note that PTTP can handle nonclausal assertions and goals in the same manner as Prolog.

For example, the assertions

```
p <- q, r.
p <- q, s.
```

can be collapsed into the single assertion

```
p <- q, (r; s).
```

where the `,` operator specifies conjunction and the `;` operator specifies disjunction.

This can result in a substantially diminished search space. PTTP can combine clauses in this manner automatically or can be provided nonclausal assertions that are transformed automatically into proper inputs for PTTP, i.e., into contrapositives with a single literal as the first argument of the `<-` connective.[5] Wilkins [38] developed the first nonclausal version of the model elimination procedure. Nonclausal formulas are also a vital feature of the TABLOG logic programming language [24].

If the query is nonclausal, its negation must be included among the assertions even when only definite answers are sought. For example, the proof of $P(a) \vee P(b)$ from $P(a) \vee P(b)$ requires the clauses

```
p(a) <- ~p(b).
p(b) <- ~p(a).
~p(a).
~p(b).

<- p(a); p(b).
```

[5]The current implementation of PTTP has the limitation that OR branches must be of equal length (e.g., `p <- q,(r;s)` and `p <- q,((r1,r2);(s1,s2))` are allowed, but `p <- q,(r;(s,t))` is not).

where the assertions `~p(a)` and `~p(b)` comprise the negation $\sim P(a) \wedge \sim P(b)$ of the query $P(a) \vee P(b)$. Further refinement of the inference system may make it unnecessary to include the negation of the query among the assertions.

5 Complete Search Strategy

Even if the problems of unification without the occurs check and an incomplete inference system are solved, Prolog is still unsatisfactory as a theorem prover because few theorem-proving problems can be solved using Prolog's unbounded depth-first search strategy. The first example in Section 8 illustrates the problem. Because neither negation nor function symbols are used, Prolog's unsound unification and incomplete inference system pose no difficulties; Prolog fails to solve this example solely because of its unbounded depth-first search strategy.

For theorem proving, Prolog's unbounded depth-first search strategy must be replaced by some complete search strategy, such as breadth-first search or the A* algorithm [26]. However, the arbitrary choice of a complete search strategy may result in losing much of the efficiency of Prolog implementations. In particular, adopting breadth-first search or the A* algorithm would make it necessary for Prolog to represent and retain more than one derived clause at once. Moreover, such strategies would increase memory requirements substantially.

A simple solution to this problem is to replace Prolog's unbounded depth-first search strategy with a bounded one. Backtracking when reaching the depth bound would cause the entire search space, up to a specified depth, to be searched completely.

It then becomes necessary to determine the depth bound. Too small a depth bound will result in failure to find a proof. Because the size of the search space grows exponentially as the depth bound increases, too large a depth bound may result in an enormous amount of wasted effort. An obvious solution is to run with increasing depth bounds; 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* [13] (we also called it *staged* or *consecutively bounded depth-*

first search [34, 35]). The effect is similar to breadth-first search except that results from earlier levels are recomputed rather than stored. Thus, when searching is done to depth n, level $n-1$ results are being computed for the second time, level $n-2$ results for the third time, and results at level 1 for the nth time.

Despite a long history of depth-first iterative-deepening search, notably in chess-playing programs [33], the strategy remained unanalyzed and unadvocated in theorem-proving and problem-solving applications until quite recently.

Because of the exponential growth in the size of the search space as the depth bound is increased, the number of recomputed results is not large in comparison with the size of the search space. In particular, analysis shows that 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 [35] (for $b = 1$, i.e., there is no branching, breadth-first search is $O(n)$ and depth-first iterative deepening is $O(n^2)$, where n is the depth). Korf [13] has shown that depth-first iterative deepening is asymptotically optimal among brute-force search strategies in terms of solution length, space, and time: it clearly 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.

Depth-first iterative-deepening search can also make use of heuristic information, in contrast to unbounded breadth- and depth-first search; the latter are uninformed search strategies that do not take into account heuristic estimates of the remaining distance to a solution. Informed search strategies, such as the A* algorithm [26], utilize such information to order the search space. Depth-first iterative-deepening search does not do that, but can use an estimate of the minimum number of remaining steps to a solution to perform cutoffs if the estimated number exceeds the number of levels left before the depth bound is reached. These cutoffs result in lower effective branching factors for depth-first iterative-deepening search than for breadth-first search. If these estimates uniformly exceed the number of remaining levels by more than one, then one or more levels can be skipped when the next depth bound is set. This test can also be used to determine when a finite search space has been fully explored. As with the A* algorithm, admissibility—the guarantee of finding a shortest solution path first—is preserved, provided that the heuristic estimate never exceeds the actual number of remaining steps to a solution.

The depth-first iterative-deepening search strategy is implemented by using the new metalevel predicate `search`. 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 at least `Inc` between searches. The last one, two, or three arguments of `search` can be omitted with default values of infinity, zero, and one assigned to `Max`, `Min`, and `Inc`, respectively. `Max` can be specified to put a bound on the total search effort (the number of inferences performed in trying to solve the goal). Total search effort 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.

Skipping levels by specifying `Min` or `Inc` is clearly beneficial if the shallowest solution is on a level that is searched. However, if the shallowest solution is on a skipped level, it is possible for skipping levels to result in extra effort, as some portion of the deeper levels will be searched before the shallowest solution (or some other solution) is discovered. For example, if the shortest solution has 20 subgoals, and we search with a depth bound of 100, although the level 20 solution will eventually be discovered, vast areas of the search space up to depth 100 will probably be explored first. In evaluating the trade-off between saving effort by skipping levels and sometimes wasting effort by finding a solution on a skipped level during a deeper search, it appears to be beneficial to skip every other level (by setting `Inc` to two) if the branching factor is two or less and detrimental if the branching factor is four or more [35].

The `search` predicate succeeds for each solution it discovers. Backtracking into `search` continues the search for additional solutions. When, as in theorem proving, only a single solution (proof) is needed, the `search` call can be followed by a cut operation (as in the top-level goal `<- search(p(b,a,c)), !, write(proved)`) to terminate further attempts to find a solution. Although `search` does not check whether the solution it found is the same as a previously discovered one, it will avoid succeeding a second time with solutions previously found during a level m search that are rediscovered in the course of a later level n search.

At the beginning of each search, a depth bound representing the number of allowable subgoals is established by `search`. The compiled code for each nonunit clause decrements [undoably upon backtracking] this bound by the number of literals in its body. If the resulting bound is negative, resolution with the clause fails and backtracking occurs. The code also keeps track of the minimum amount by which the depth bound is exceeded; this is used to increment the depth bound for the next search.

This process merely counts subgoals to estimate the number of steps remaining to a solution and, as is required for admissibility, never overestimates their number, since each subgoal will require at least one inference step for its solution. It is desirable to have better (but still admissable) estimators; this may be difficult to achieve, however, because subgoals can often be removed in a single step—by resolution with a unit clause or by reduction. Other estimators are discussed in Section 6.

Depth-first iterative-deepening search can be easily parallelized. The simplest way would be to assign the first n levels of search to the n available processors. As a processor completes its exhaustive search of a level, it starts working on the next available level. This method has been called *depth-first parallel deepening*. This approach has the virtues of requiring little modification of the system (e.g., multiple running copies of PTTP must each have their own versions of implementation variables that contain the depth bound, the lists of ancestor goals, etc.) and, because the searches are independent, extremely low run-time overhead. It can sometimes even yield a superlinear speedup, i.e., more than n times faster for n processors. An extreme example of this would be if a solution were found by the very first inference on level n. Searching the first n levels in parallel would result in an almost immediate discovery of a solution, while depth-first iterative deepening would require the costly, complete search of level $n-1$ before performing any level n inferences.

When the number of processors exceeds the number of levels that it is reasonable to search in parallel (e.g., if 100 processors are available, but a solution is expected to be found at level 20 or below, or when search of a single level simply takes too long) then methods that partition the search space for a single level must be employed to take advantage of all the available processors [29]. This is akin to the OR-parallel execution of standard Prolog programs.

6 Existing Refinements

The changes made in unification, the inference system, and the search strategy are all sufficient to create a Prolog technology theorem prover that is complete for the full first-order predicate calculus. It is of course possible to refine this system by adding restrictions on the current inference operations, by refining the search process, or by introducing entirely new inference operations.

The ME procedure justifies the completeness of PTTP even if some goal states are disallowed.

For example, PTTP remains complete even if we cause the the current goal to fail under any of the following circumstances:

- A goal is identical to one of its ancestor goals. (It is unnecessary to attempt to solve a goal while in the process of attempting to solve that same goal.)
- A goal with subgoals is complementary to one of its ancestor goals. (It is unnecessary to solve a goal that is complementary to an ancestor goal by any means other than the reduction operation.)
- A goal with subgoals is an instance of a unit clause. (It is unnecessary to solve a goal that is an instance of a unit clause by any means other than extension by the unit clause.)

Causing a goal that is identical to an ancestor goal to fail means that commutativity assertions, such as `p(X,Y,Z) <- p(Y,X,Z)`, do not by themselves lead to an infinite search space, since the sequence of subgoals obtained by commutativity `p(a,b,c)`, `p(b,a,c)`, `p(a,b,c)`, ...is cut off at the first repetition of `p(a,b,c)`.

Also, since there are a finite number of propositional symbols in any propositional-calculus problem and any state in which a goal is either identical or complementary (unless removed by reduction) to an ancestor goal is rejected by the above rules, there can be no infinitely long deduction sequence for propositional-calculus problems. Thus, they can be solved safely and completely without any depth bound.

Because the search space in theorem proving is generally exponential, it is always worth considering criteria for failing goals, so that the exponentially many derivative deductions can be eliminated. However, the desire to cut off deductions must be balanced against the cost of checking whether the present deduction is acceptable according to the criteria. Because depth-first iterative-deepening search requires minimal memory, there is no point in reducing the number of inferences at the expense of overall increased running time. In contrast to other theorem-proving systems, it seems that the only reasonable measure of performance for PTTP is the execution time for a proof.

After experimentation with various alternatives, the current implementation employs the following more limited forms of the restrictions:

- If the current goal (before unification with any clause in the procedure) is identical to one of its ancestor goals, then fail.
- If the current goal is exactly complementary to one of its ancestor goals, then perform the reduction operation and cut (disallow any other inferences on the current goal).
- If the current goal is an instance of a unit clause, then perform the extension operation and cut (disallow any other inferences on the current goal).

These tests can quickly check for immediate violations of the ME restrictions, but will not detect violations caused by the application of later substitutions. This is suboptimal in terms of search-space size reduction, since some states that violate the restrictions are not eliminated, but relatively sophisticated code for detecting all violations, depending on demons associated with individual variables that check for identity of goals when the variable is instantiated, has so far cost much more time than is saved by the diminished search space. Thus, these tests seem to be a reasonable compromise between the cost of checking the ME restrictions and the amount of searching eliminated.

Two possible solutions to this problem are to develop yet more efficient means for checking these conditions or to perform the checks less often. An effective means of reducing the frequency of the checks while maintaining most of their value is to restrict them to the earlier levels of the search. Given the exponential search space, cutoffs by earlier-level checks reduce the overall search-space size and running time more than do checks that are near the depth bound.

To make more effective the rule of cutting off alternatives if the current goal is an instance of a unit clause, the clauses of a procedure are automatically reordered by the PTTP compiler to put unit clauses ahead of nonunit clauses, unless precluded from doing so by the user.

We have also experimented with a generalization of the rule to nonunit clauses. In [pure] Prolog, if a goal is solved without being instantiated, alternative solutions clearly need not be examined, since they could only solve the same or a more specific goal again. Thus, it would be legitimate to place a conditional cut at the end of each clause that performs the cut operation if and only if the invoking goal that matched the head of the clause did not become further instantiated in the process of its solution (which condition should be easily checkable by examination of the trail in Prolog implementations). In PTTP, it is also necessary to verify that no ancestor of the invoking goal is further instantiated by a reduction operation during solution of the invoking goal.

In some pathological examples, addition of the cut operation results in loss of completeness when used with depth-first iterative-deepening search. Consider the example

```
p <- q(X), !.
q(s(X)) <- q(X).
q(0).
r <- s.
s.

<- search(p,r).
```

The cut operation in the first clause is legitimate because the invoking goal `p`, which lacks variables, cannot have been instantiated. However, in PTTP the depth-first iterative-deepening search process will never return a solution. When a proof with n subgoals is sought for any n, the order of the clauses for `q` will cause an n subgoal proof of `p` to be found before any shorter proof; alternatives for `p` are cut off, and no more subgoals are allowed for the solution of `r`, which requires the subgoal `s`. To solve this problem, we treat a goal with subgoals that is solved without instantiation as if its solution required zero subgoals, cut off alternatives, and proceed to solve the remaining goals.

This refinement can substantially reduce the amount of search before a proof is found. However, it infrequently increases the amount of search and has the disadvantage of not always finding a shortest proof. This refinement often leads to proofs that use many more subgoals than the current search limit specifies. For example, a 52-subgoal proof was found with a search limit of 13 subgoals for Problem 82 in Table 1.

When performing depth-first iterative-deepening search, it is sometimes possible to use better estimators of the number of remaining goals to complete a solution than to just count the number of subgoals in the body of the clause. Recognizing that completing a proof using the clause `p <- q, r` may really require more than two steps to solve `q` and `r` can result in more cutoffs and a diminished search space. In the case of Prolog-like problems consisting entirely of Horn clauses, when the reduction operation is impossible, any predicate defined entirely by nonunit clauses will always require more than a single step to solve. A goal will require in addition at least n subgoals in its solution, where n is the number of subgoals in the shortest clause in the goal predicate's definition (for Horn-clause problems, this computation could easily take account of the minimum estimated costs of the subgoals as well as simply their number).

For example, when attempting to solve the goal `p` by the clause `p <- q, r`, code using the standard estimator reduces the depth bound by two (the number of literals in the body) and proceeds if the depth bound is still not less than zero; code using the better estimator reduces the depth bound by two but proceeds only if the depth bound is still not less than $cost(\mathtt{q}) + cost(\mathtt{r})$, where $cost(x)$ is the minimum estimated cost of solving x.

In the case of non-Horn-clause problems, however, even if a predicate is defined entirely by nonunit clauses, a goal with that predicate might still be solved in a single step by a reduction operation. A solution is to include the costs of `q` or `r` in the depth-bound computation only if no ancestor goal with complementary predicate `~q` or `~r` exists. This is a quick check that `q` or `r` cannot be removed by reduction. Code using the better estimator reduces the depth bound by two and proceeds if the depth bound is still not less than

$$\left\{\begin{array}{cl} \mathit{cost}(\mathtt{q}) & \text{if no ancestor goal } \mathtt{\tilde{}q} \text{ exists} \\ 0 & \text{otherwise} \end{array}\right\} + \left\{\begin{array}{cl} \mathit{cost}(\mathtt{r}) & \text{if no ancestor goal } \mathtt{\tilde{}r} \text{ exists} \\ 0 & \text{otherwise} \end{array}\right\}$$

Note that computing *cost*(`q`) and *cost*(`r`) can be done at compile-time, but that determining whether `~q` or `~r` ancestors exist and summing the results must be done at run-time.

It is certainly possible to devise more refined estimators that check more thoroughly whether reduction is possible or that compute the cost of a goal depending on what ancestor goals there are. However, more elaborate estimators may be too expensive to use, since they may not result in enough reduction in the size of the search space to compensate for their having to be computed each time a goal is resolved with a nonunit clause. It is important to find estimators that are both effective in pruning the search space and require little run-time computation.

Another interesting refinement of the search process is to treat some subgoals that cannot lead to infinite deduction sequences as zero-cost subgoals for the purpose of iterative-deepening search. If propositional goals were treated as zero-cost subgoals, propositional-calculus problems could then be automatically solved while searching for a solution with no counted subgoals.

It seems reasonable to treat `wolf(X)` and `fox(X)` as zero-cost subgoals in the rules `animal(X) <- wolf(X)` and `animal(X) <- fox(X)`. If they are treated as ordinary subgoals in a problem, the depth bound may result in the taxonomy of animals being searched insufficiently deeply or, if it is searched deeply enough, the search may require so many levels of search that the remaining depth bound is insufficient to solve the remaining goals. The problem would then have to be solved with a greater depth bound. Whether generating the animals, as in the goal `animal(X)`, or testing that something is an animal, as in the goal `animal(fido)`, it is often better to not charge the cost of searching the taxonomy against the depth bound for the problem.

Application of the commutativity rule `p(X,Y,Z) <- p(Y,X,Z)` would be "free" if `p(Y,X,Z)` is treated as a zero-cost subgoal. This has the flavor of building in commutative unification, because both `p(a,b,c)` and `p(b,a,c)` can be derived from `p(a,b,c)` using no counted subgoals.

The use of zero-cost subgoals builds in a bias toward using rules with zero-cost subgoals (since they consume fewer search levels), increases number of inferences on any level, and effectively increases the branching factor. Proofs using zero-cost subgoals will be found with a lower depth bound with this refinement than without, often with great savings in the number of inferences performed. However, the increased branching factor can result in a greater number of inferences being performed if zero-cost subgoals appear insufficiently often in the proof.

7 Possible Future Refinements

It is valuable to investigate other search-space-pruning restrictions. General methods for "intelligent backtracking" in Prolog systems, of which the conditional-cut methods described above are a special case, would also be beneficial for PTTP. The adaptation of Prolog intelligent backtracking methods to PTTP is not entirely trivial. As it did for the conditional-cut operation, the use of bounded search causes a problem for intelligent backtracking methods. In Prolog, when solving the goals `p(X), q(Y), r(X)`, if goal `r` fails, `p` can be directly backtracked to, because the computation of `q` and the bindings for `Y` it creates are irrelevant to the failure of `r`. However, in PTTP, the computation of `q` can affect the success or failure of `r`, by using subgoals that reduce the depth bound available for solving `r`. The goal `r` might be made to succeed with alternative solutions of `q` that require fewer subgoals in their solution.

The current implementation lacks either the model-elimination *lemma* facility [16, 17] or the similar *C-reduction* operation of the graph-construction procedure [32] that can be used to shorten deductions by recognizing a goal as having been solved previously.

More exotic extensions are also worth considering. Among these are support for equality reasoning and special unification for algebraic properties such as associativity and commutativity, or for sorted logic or types.

An obvious first step toward including equality reasoning is the addition of a demodulation (equality rewriting) facility that simplifies goals to an irreducible form before any attempt to solve them. Essentially the same compilation methods as are used for Prolog clauses can be applied to demodulators, making this equality simplification process quite rapid. Though useful, demodulation is obviously insufficient for complete inference for first-order predicate calculus with equality. Worse yet, adding demodulation without fuller equality handling may make an otherwise complete inference system incomplete. Model elimination with paramodulation [17] is complete, though the branching factor may be too high for it to be effective in PTTP.

8 Examples

The first example is the problem of proving that a group is commutative if the square of every element is the identity element. This is Problem 9, for which performance results are given in Table 1; Problem 14 and Chang&Lee Problem 2 are the same except for the order of the clauses. Problem 35 adds clauses for uniqueness, totality, inverse, equality, and substitutivity.

In the example, we use the common convention that $P(x, y, z)$ denotes $x \circ y = z$, where $\circ$ is the group multiplication operation. A clause-by-clause description of the input is as follows: (1) e is a right identity, i.e., $x \circ e = x$; (2) e is a left identity; (3-4) the associativity axioms $u \circ z$ is w if and only if $x \circ v$ is w, where $x \circ y$ is u and $y \circ z$ is v;[6] (5) for all x, x^2 is e;

[6]These two 4-literal clauses for associativity can be collapsed into the single assertion `p(X,V,W) <- p(U,Z,W), ((p(X,Y,U), p(Y,Z,V)); ((p(U,Y,X), p(Y,V,Z))` after variables have been renamed and literals reordered. This produces a smaller search space.

(6) the hypothesis that $a \circ b$ is c; (7) the theorem that $b \circ a$ is c, and thus $\circ$ is commutative.

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

Compilation time for the example is specified in parts: translation from extended Prolog to LISP by the PTTP compiler (0.284 seconds) and translation from LISP to machine code by the Symbolics LISP compiler (14.436 seconds). There has been no effort to optimize compilation time. A PTTP compiler that compiled directly to Prolog-virtual-machine instructions could be expected to run as fast as the current PTTP compiler and would eliminate the time-consuming LISP compilation.

```
GROUP2-EXAMPLE
 The symbols u, v, w, x, y, and z denote variables.
  1.  p(x,e,x).
  2.  p(e,x,x).
  3.  p(x,v,w) <- p(x,y,u) , p(y,z,v) , p(u,z,w).
  4.  p(u,z,w) <- p(x,y,u) , p(y,z,v) , p(x,v,w).
  5.  p(x,x,e).
  6.  p(a,b,c).
 ----------------
  7.  query <- search(p(b,a,c)) , !.

Compilation time: 0.284 seconds (PTTP) + 14.436 seconds (LISP)

                                 Start searching with no subgoals.
          0 inferences so far.   Start searching with at most  3 subgoals.
          8 inferences so far.   Start searching with at most  6 subgoals.
        157 inferences so far.   Start searching with at most  9 subgoals.

Proof:
Goal#  Wff#  Wff Instance
-----  ----  ------------
(  0)    7   query <- p(b,a,c).
(  1)    3      p(b,a,c) <- p(b,b,e) , p(b,c,a) , p(e,c,c).
(  2)    5         p(b,b,e).
(  3)    4         p(b,c,a) <- p(a,c,b) , p(c,c,e) , p(a,e,a).
(  4)    3            p(a,c,b) <- p(a,a,e) , p(a,b,c) , p(e,b,b).
(  5)    5               p(a,a,e).
(  6)    6               p(a,b,c).
(  7)    2              p(e,b,b).
(  8)    5            p(c,c,e).
(  9)    1            p(a,e,a).
( 10)    2          p(e,c,c).
      1,136 inferences so far.   Search ended by cut.

Execution time: 1,136 inferences in 0.262 seconds (4.34 K lips)
```

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(b,a,c)`. Its only subgoal is `p(b,a,c)` whose solution starts on line (1) and uses the clause `p(b,a,c) <- p(b,b,e) , p(b,c,a) , p(e,c,c)`. Its subgoals `p(b,b,e)`, `p(b,c,a)`, and `p(e,c,c)` are then solved, starting on lines (2), (3), and (10), respectively. Indentation is used to help identify subgoal relationships.

Of particular note in this proof is the fact that PTTP did not search slavishly for a solution with 0, 1, 2, 3, ... , 9 subgoals, but instead skipped all the searches except those for solutions with 0, 3, 6, and 9 subgoals. This was accomplished not by specifying a search increment, but was done automatically, as described earlier. Upon completion of the search for a solution with no subgoals, it was recognized that any solution must extend the initial goal `p(b,a,c)` by either clause (3) or (4), since these are the only nonunit clauses. Each of these introduces 3 subgoals, so a solution with only 1 or 2 subgoals is an impossibility. The same thing occurred in the searches for solutions with 3 and 6 subgoals, as a result of which levels 4, 5, 7, and 8 were skipped.

PTTP performed 0, 8, and 157 inferences cumulatively after completing the searches for solutions with 0, 3, and 6 subgoals, respectively. Search for a solution with 9 subgoals terminated successfully after 1,136 inferences had been performed.

The second example is part of a proof that any number greater than 1 has a prime divisor. This is Problem 3 in Table 1; it is also, except for clause order, the same as Chang&Lee Problem 8.

A clause-by-clause description of the input is as follows: (1) x divides x; (2) if x divides y, and y divides z, then x divides z; (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 induction hypothesis that for all x between 1 and a there is a prime $f(x)$ that (7) divides x; (8) a is greater than 1; (9) the negation of the theorem, necessary when seeking indefinite answers; (10) the theorem that a has a prime divisor.

```
PRIM-EXAMPLE
 The symbols x, y, and z denote variables.
  1.  d(x,x).
  2.  ~d(x,y) ; ~d(y,z) ; d(x,z).
  3.  p(x) ; d(g(x),x).
  4.  p(x) ; l(1,g(x)).
  5.  p(x) ; l(g(x),x).
  6.  ~l(1,x) ; ~l(x,a) ; p(f(x)).
  7.  ~l(1,x) ; ~l(x,a) ; d(f(x),x).
  8.  l(1,a).
  9.  ~p(x) ; ~d(x,a).
 ----------------
 10.  query <- search((p(x) , d(x,a))) , !.

Compilation time: 0.875 seconds (PTTP) + 20.390 seconds (LISP)

                                 Start searching with no subgoals.
         0 inferences so far.   Start searching with at most  1 subgoal.
         3 inferences so far.   Start searching with at most  2 subgoals.
         9 inferences so far.   Start searching with at most  3 subgoals.
        27 inferences so far.   Start searching with at most  4 subgoals.
        57 inferences so far.   Start searching with at most  5 subgoals.
       110 inferences so far.   Start searching with at most  6 subgoals.
       194 inferences so far.   Start searching with at most  7 subgoals.
       355 inferences so far.   Start searching with at most  8 subgoals.
       593 inferences so far.   Start searching with at most  9 subgoals.
     1,082 inferences so far.   Start searching with at most 10 subgoals.
     1,828 inferences so far.   Start searching with at most 11 subgoals.

Proof:
Goal#  Wff#  Wff Instance
-----  ----  ------------
(  0)   10   query <- p(a) , d(a,a).
(  1)    4a     p(a) <- ~l(1,g(a)).
(  2)    6a        ~l(1,g(a)) <- l(g(a),a) , ~p(f(g(a))).
(  3)    5b           l(g(a),a) <- ~p(a).
(  4)                    ~p(a).
(  5)    9a           ~p(f(g(a))) <- d(f(g(a)),a).
(  6)    2c              d(f(g(a)),a) <- d(f(g(a)),g(a)) , d(g(a),a).
(  7)    7c                  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) <- ~p(a).
( 10)                              ~p(a).
( 11)    3b                  d(g(a),a) <- ~p(a).
( 12)                           ~p(a).
( 13)    1      d(a,a).
      3,104 inferences so far.   Search ended by cut.

Execution time: 3,104 inferences in 0.646 seconds (4.81 K lips)
```

In this proof, lines (4), (8), (10), and (12) show subgoals being solved by the reduction operation. In particular, the goals `~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 `~l(1,g(a))` in line (2).

Examination of the proof shows clauses (10) and (9), 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 required all of PTTP's extensions of Prolog: sound unification, complete search, the reduction operation, and indefinite answers.

9 Performance

It is never an easy task to find a large number of problems with suitable accessibility, variety, and difficulty. We used the Wilson and Minker study [39] as a source of problems.[7]

They took Problems 1–9 from Reboh et al. [31], problems 10–19 from Michie et al. [25], problems 20–25 from Fleisig et al. [12], problems 26–58 from Wos [41], and problems 59–86 from Lawrence and Starkey [15]. This problem set has also been used to test the Markgraf Karl Refutation Procedure connection-graph resolution theorem-proving program [7, 30].

[7]The technical-report version of their article includes a listing of the problems. Their tabulated results show problems named EX5-T1, EX5-T2, LS76, and LS86, but the listing contains only a single problem EX5, a problem LS76 with two theorems, and no problem LS86. Thus, we tried 85 problems: EX5, LS76-T1, LS76-T2, and all the other problems. Problems DBABHP, EX6-T1, EX6-T2, WOS23, and WOS26 were corrected.

We have so far solved 78 of the 85 problems; 49 were solved in less (often much less) than a second each (not counting compilation time); 70 were solved in less than a minute each. Wilson and Minker solved 62 of the problems, while 63 were solved by the Markgraf Karl Refutation Procedure. We, like the other researchers, generally had the most difficulty with some of the Fleisig examples and the Wos examples. In principle, the Fleisig examples are obtainable by a system like ours, since their system also implemented the model elimination procedure. However, to control the search space, they used completeness-limiting parameters, such as the number of times a clause could be used in a proof.

Wos, of course, solved all the problems (in 1965!) that he contributed to the Wilson and Minker study. Our failure to solve some of them is attributable to the current limits of PTTP—its need to explore an exponential search space, unmitigated by redundancy control mechanisms like demodulation and subsumption, by pure backward chaining. Fleisig et al. indicated that their implementation of model elimination rarely succeeded on problems with large sets of clauses. The Wos examples have comparatively large numbers of clauses with few predicates and many variables, so that they can be easily applied; we find some of them too difficult.

Table 1 presents the results for solved problems.[8] The depth of proof is expressed as $m + n$, where m is the number of initial goals (i.e., the length of the top clause of the derivation) and n is the number of subgoals. '**' is shown in place of the number of subgoals if the problem is solved with the refinement that allows cutting off alternatives of a goal that is solved by a nonunit clause without instantiation. These proofs often have many more subgoals than a shortest proof and the subgoal limit that was in effect at the time they were found. Time is measured on a Symbolics 3600 with an instruction fetch unit (IFU). The times shown here are just the times spent searching for proofs; compilation time and the time required for printing the proof and information about the search are not included.

[8]These results were obtained in June-July 1987 by a version of PTTP that has not been distributed. Versions distributed earlier do not have all the capabilities described in this paper, though many of these results can be duplicated with those earlier versions.

The results are for problem solutions with essentially no heuristic modification of the problem inputs or search process. Experimentation has shown that these results can often be substantially improved. The order of clauses in the input and the order of literals within the clauses were maintained, except that clauses were reordered so that unit clauses precede nonunit clauses (to facilitate pruning the search space when the current goal is an instance of a unit clause). Large performance gains have often, though not uniformly, been found when literals in the bodies of input clauses are reordered to put those with fewer free variables earlier in the list of subgoals. Literal reordering can be easily done automatically at compile-time.

Depth-first iterative deepening was performed using the default settings so that no levels of search were omitted by user specification. Low branching factors make skipping every other level of search quite effective for some of these problems. We also did not use the suggested refinements of better estimators of the number of remaining steps to a solution or zero-cost subgoals though these too can often result in much better performance.

PTTP's ability to use nonclausal formulas was also not employed. Many of the results shown here can be significantly improved upon by using nonclausal formulas. If clauses can be combined into nonclausal formulas without reordering literals or clauses, the number of inferences is guaranteed to decrease (in any search space in which the combined clauses are used).

At the end of the table, we also provide results for the problems that appear in Chang and Lee [8], pp. 298–305. All nine problems were solved in a total of a little more than one second.

10 Related Work

There are several other efforts to design and construct Prolog-like systems that perform sound and complete deduction for the full-first order predicate calculus. F-Prolog [36] is very similar in its interfaces and basic algorithms to PTTP. It includes unification with the occurs check, the reduction operation (solving a goal in the context of a set of assumptions that includes the negations of ancestor goals), and either simple bounded depth-first search or depth-first iterative-deepening search. F-Prolog also performs nonclausal inference. Its implementation in Prolog rather than as a Prolog extension makes it much slower than PTTP.

Other inference systems that are complete extensions of Prolog for non-Horn clauses have been proposed by Eder [11], Loveland [18, 19], Plaisted [28] and a group at Technische Universität München (TUM) [3].

In Eder's system, the non-Horn clause $\sim P \vee \sim Q \vee R \vee S$ is represented by the clause `r; s <- p, q`, and only the literals `r` and `s` are matched in resolution operations. Non-Horn proofs involving case analysis are generated by resolution using lemmas consisting of "excess positive literals". Unlike PTTP, Eder's system requires factoring. His system also tries to eliminate some redundancy that we do not: in his system proving $\exists x\ P(x)$ from $P(a) \vee P(b)$ results in one proof, not two.

Loveland employs the same representation for non-Horn clauses and likewise resolves only on the head literals. He intends his system to be used primarily for sets of clauses that are "near-Horn"—i.e., only a few clauses have more than one positive literal. This is a broad, useful class of problems. Loveland's system operates by solving the initial query in exactly the same way as Prolog, except that, when `r` or `s` of a non-Horn clause `r; s <- p, q` is resolved on, the remaining head literal `s` or `r` becomes a deferred head. Upon completion of a "proof" of the query, if any head literals were deferred, the proof is reattempted, this time with goals being solvable by matching them to deferred heads. When no goals or deferred heads remain, the query has been solved.

Plaisted has implemented a new theorem prover using a new method based on his simplified problem reduction format [27]. His system also does not require contrapositives. Plaisted emphasizes the unnatural backward-chaining searches that can result from the use of contrapositives. For example, solving `and(a,b)` by `and(X,Y) <- X, Y` seems natural, but use of the contrapositive `~X <- ~and(X,Y), Y` is procedurally unattractive. Much the same can be said for the functional reflexive axiom `f(X,U)=f(X,V) <- U=V` and its contrapositive `~U=V <- ~f(X,U)=f(X,V)`. His system does require the use of splitting rules that may be difficult to control (the current system being intended to improve control over the splitting rules in the original simplified problem reduction format). This inference system also has some ability to do forward chaining as well as Prolog's backward chaining. The theorem prover includes demodulation to partially implement equality. Like PTTP, Plaisted's system uses depth-first iterative-deepening search. It allows user specification of zero-cost rules by using the `:--` connective instead of the usual `:-` connective. An important operational difference between his system and PTTP is the caching of goal solutions to eliminate repeated solutions. Caching can, of course, reduce the number of inferences substantially. However, it requires different, less efficient representations of derived clauses than PTTP because results have to be retained and, in addition, it requires more storage. Caching and other features preclude the same high inference rate that is possible for PTTP; Plaisted's system currently runs at about 15 lips on a Sun 3. However, the control of redundancy that caching provides makes his system competitive with PTTP in total solution time for some problems.

TUM's PROTHEO (*Pro*log-like *theo*rem prover) is a Prolog-like extension based on the connection method [4, 1]. It uses depth-bounded search, lemmas to eliminate redundant computations, and has some bottom-up as well as top-down reasoning capabilities.

PTTP is also somewhat related to other efforts (too numerous to mention) on logic programming and, as extensions for handling equality are developed, the efforts on rewriting- and narrowing-based logic programming and functional programming. PTTP differs from this work in its emphasis on doing sound and complete deduction, as well as in deemphasis of its use as a programming language.

Another valuable method for using Prolog technology in theorem proving is being developed at Argonne National Laboratory [5]. The Argonne researchers plan to use Prolog technology (an implementation of the Warren abstract machine [37]) to reimplement their powerful interactive theorem-proving system LMA+ITP [21, 22, 23].

The work on PTTP has tried to extend Prolog to general-purpose theorem proving in such a way as to get the highest possible inference rate. We are willing to perform more inferences if their individual cost can be kept low. The Argonne approach, on the other hand, is to use Prolog technology to modify their system so as to make exactly the same inferences as before, only faster. Model elimination is perhaps the closest match to Prolog of an inference system for the full first-order predicate calculus, so that using Prolog technology to speed up the other inference operations in LMA+ITP will not yield the same high inference rate as can be attained by a Prolog technology theorem prover. Thus, there is a trade-off between the higher inference rate of PTTP and the greater flexibility of LMA+ITP that often allows much smaller search spaces.

11 Conclusion

A Prolog technology theorem prover has numerous advantages. For problems that are not too difficult, namely, if the proof is not too deep or the branching factor too large, PTTP can explore the search space rapidly and return an answer promptly. Given the use of depth-first search, memory requirements are almost neglible. It can be a very helpful reasoning utility. It is exceptionally straightforward to use, since its inference system and search strategy are essentially fixed. Prolog computations can be embedded without difficulty in the theorem-proving process because it is implemented as an extension of Prolog itself. Because of its simplicity, it is comparatively easy to implement correctly and its behavior is comprehensible.

The results in Table 1 show very competitive results that were obtained with little variation in the inference system or search strategy. The inference system and search strategy are complete without qualification. We did not, for example, eliminate costly but theoretically necessary inference rules like factoring, use extra inference operations like demodulation that can sometimes result in loss of completeness, discard any derived clauses because they were too long or their terms too complex, impose limits on the number of times a clause could be used in a proof, or utilize any other heuristics to restrict the search space. Successful proofs by other theorem-proving systems sometimes depend more on completeness-limiting restrictions than on the underlying inference system and search strategy; this is not the case for PTTP, for which completeness is limited only by the amount of time we are willing to allow in the search for progressively deeper proofs.

In solving these test problems, we did not even employ additional techniques (e.g., clause and literal reordering, nonclausal formulas, skipping levels of search, better estimators of number of remaining subgoals to a solution, or zero-cost subgoals) which we have determined often yield much better performance.

Our prototype implementation is also definitely nonoptimal. A better design for the 3600 (with microcode support for a Prolog virtual machine [37]) could perhaps yield an increase in speed by a factor of 5 or 10. Such a design would also reduce compilation time and object code size. The current PTTP-to-LISP compiler is fast enough, though there has been no effort to optimize it. However, compiling the resulting LISP code is somewhat slow and the resulting object code quite large. Compiling directly to a virtual-machine instruction set should not take much longer than compiling to LISP and the object code would be much smaller.

Despite its high performance and many possible improvements, PTTP is not a panacea for the problem of theorem proving in general. Solutions to the hardest problems will probably always require human assistance in specifying strategies and in determining where to search for a solution.

The major strength of PTTP—implementation of high-speed backward-chaining deduction—is also its major weakness. Formulas sometimes have no computationally reasonable backward-chaining procedural interpretation. For example, the associativity clause `p(U,Z,W) <- p(X,Y,U), p(Y,Z,V), p(X,V,W)`, when applied to the goal `p(b,a,c)`, will try to find instances of `X` and `Y` whose product is `b`. Reordering the literals in the body does not yield any more natural computation. Forward-chaining hyperresolution looks more attractive for problems like this. The adoption of Prolog's representation for derived clauses allows a very high inference rate, but precludes clause retention for such operations as subsumption to eliminate search-space redundancy. Deep proofs will be difficult to find.

For handling the hard problems that are currently beyond its reach, a Prolog technology theorem prover can be improved by incorporating some of the refinements suggested above. In addition, it may be speeded up by improvements in Prolog machine technology. It should always be possible to build a PTTP that runs at a respectable fraction of Prolog's speed. Projected machines with execution measured in megalips would make possible a PTTP orders of magnitude faster than the current one, which is already quite fast.

For hard problems, PTTP can be used as a reasoning component in a larger theorem-proving system. For example, a Prolog technology theorem prover could play a role like that of the Terminator [2] in the Markgraf Karl Refutation Procedure or of an implementation of the linked inference principle [40] in the Argonne theorem provers.

It is our hope that, with PTTP, we have helped establish a new standard of performance for theorem-proving programs working on problems comparable in difficulty to our examples.

Acknowledgements

The author would like to thank Mabry Tyson and Richard Waldinger for their many useful comments on earlier drafts of this paper.

Table 1

Problem	Number of Clauses	Depth of Proof	Number of Inferences	Run Time (sec)
1. BURSTALL	19	1+11	690	0.132
2. SHORTBURST	11	1+ 5	35	0.005
3. PRIM	9	2+11	3,104	0.646
4. HAS-PARTS-T1	8	1+ 9	87	0.020
5. HAS-PARTS-T2	8	1+23	14,598	4.834
6. ANCES2	7	2+17	957	0.053
7. NUM1	7	1+ 5	21	0.004
8. GROUP1	6	1+ 3	5	0.002
9. GROUP2	7	1+ 9	1,136	0.262
10. EW1	6	1+ 6	37	0.003
11. EW2	5	2+ 5	32	0.003
12. EW3	9	3+15	771	0.049
13. ROB1	3	4+ 4	15	0.001
14. ROB2	7	1+ 9	1,402	0.320
15. DM	4	1+ 3	5	0.002
16. QW	3	3+ 9	1,406	0.300
17. MQW	5	2+ 4	535	0.158
18. DBABHP	14	1+10	1,168	0.362
19. APABHP	18	1+13	1,707,214	872.509
20. EX4-T1	7	3+**	7,010	2.688
21. EX4-T2	7	2+**	13,015	5.226
22. EX5	14	1+17	414,692,795	287,077.380
24. EX6-T1	8	4+14	9,170,188	3,244.129
25. EX6-T2	8	5+14	144,144	41.747
26. WOS1	17	1+ 9	139,068	57.629
27. WOS2	16	1+ 9	20,443	6.542
28. WOS3	20	1+ 2	12	0.004
29. WOS4	23	1+12	9,061,115	4,152.200
30. WOS5	16	1+ 6	795	0.237
31. WOS6	20	1+ 8	12,707	4.236
32. WOS7	19	1+ 6	574	0.172
33. WOS8	18	1+ 5	200	0.053
34. WOS9	20	1+ 6	863	0.277
35. WOS10	20	1+ 9	78,689	29.416
36. WOS11	22	1+ 7	9,135	3.576
37. WOS12	21	1+ 3	6	0.002
38. WOS13	22	3+ 3	51	0.013
39. WOS14	21	1+ 6	118	0.027
40. WOS15	23	1+15	91,884,537	39,560.500
41. WOS16	27	1+ 6	1,165	0.404
42. WOS17	30	1+ 7	21,534	8.871
43. WOS18	25	1+ 4	46	0.011
44. WOS19	33	1+ 7	9,997	3.701
45. WOS20	33			
46. WOS21	30	1+11	2,396,485	965.747
47. WOS22	34	1+13	73,528,154	36,694.406
48. WOS23	31	1+ 4	108	0.038

Problem	Number of Clauses	Depth of Proof	Number of Inferences	Run Time (sec)
49. WOS24	29	1+ 5	224	0.073
50. WOS25	36	1+ 5	181	0.077
51. WOS26	24			
52. WOS27	30	1+ 5	157	0.045
53. WOS28	75			
54. WOS29	40	1+ 7	16,956	5.942
55. WOS30	42	1+ 3	34	0.011
56. WOS31	23			
57. WOS32	26	1+ 7	1,711	0.670
58. WOS33	26			
59. LS5	4	2+ 4	31	0.005
60. LS17	12	3+ 6	175	0.030
61. LS23	6	1+ 6	268	0.067
62. LS26	10	1+ 6	34	0.007
63. LS28	13	1+ 6	1,322	0.843
64. LS29	13	1+ 6	975	0.622
65. LS35	6	1+12	8,224	2.846
66. LS36	20	1+11	1,399,752	547.850
67. LS37	18			
68. LS41	11	1+ 2	9	0.003
69. LS55	13	1+ 3	57	0.017
70. LS65	20	1+ 8	16,674	6.160
71. LS68	15	1+ 1	2	0.001
72. LS75	16	1+ 6	3,364	1.280
73. LS76-T1	17	1+ 2	8	0.002
74. LS76-T2	18			
75. LS87	22	1+ 8	59,222	23.344
76. LS100	9	1+ 3	7	0.002
77. LS103	14	1+11	1,826	0.443
78. LS105	14	1+ 4	34	0.006
79. LS106	14	1+ 4	34	0.007
80. LS108	16	1+**	78,660	27.444
81. LS111	14	1+ 4	35	0.007
82. LS112	23	1+**	73,415	31.293
83. LS115	21	1+ 6	109	0.032
84. LS116	16	1+11	7,897	2.343
85. LS118	29	1+**	132,903	48.614
86. LS121	21	1+**	6,713	2.476
Chang&Lee 1	5	1+ 3	5	0.002
Chang&Lee 2	7	1+ 9	1,589	0.373
Chang&Lee 3	5	1+ 9	206	0.046
Chang&Lee 4	5	1+ 6	26	0.005
Chang&Lee 5	9	1+ 3	4	0.001
Chang&Lee 6	9	1+ 6	26	0.005
Chang&Lee 7	7	1+ 5	24	0.004
Chang&Lee 8	9	2+11	3,104	0.652
Chang&Lee 9	8	3+ 7	163	0.027

References

[1] Andrews, P.B. Theorem proving via general matings. *Journal of the ACM 28,* 2 (April 1981), 193–214.

[2] Antoniou, G. and H.J. Ohlbach. TERMINATOR. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* Karlsruhe, West Germany, August 1983, 916–919.

[3] Bayerl, S., F. Kurfess, R. Letz, and J. Schumann. PROTHEO/2: sequential PROLOG-like theorem prover based on the connection method, 1986.

[4] Bibel, W. *Automated Theorem Proving.* Friedr. Vieweg & Sohn, Braunschweig, West Germany, 1982.

[5] Butler, R., E. Lusk, W. McCune, and R. Overbeek. Paths to high-performance automated theorem proving. *Proceedings of the 8th Conference on Automated Deduction,* Oxford, England, July 1986, 588–597.

[6] Boyer, R.S. and J S. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer and D. Michie (eds.). *Machine Intelligence 7.* Edinburgh University Press, Edinburgh, Scotland, 1972.

[7] Bürckert, H.J., H. Wang, and R. Zheng. MKRP: a performance test by working mathematicians. Memo SEKI-83-1, Institut für Informatik I, Universität Karlsruhe, Karlsruhe, West Germany, 1983.

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

[9] Cohen, J. Describing Prolog by its interpretation and compilation. *Communications of the ACM 28,* 12 (December 1985), 1311–1324.

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

[11] Eder, G. A PROLOG-like interpreter for non-Horn clauses. D.A.I. Research Report No. 26, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, September 1976.

[12] Fleisig, S., D. Loveland, A.K. Smiley III, and D.L. Yarmush. An implementation of the model elimination proof procedure. *Journal of the ACM 21,* 1 (January 1974), 124–139.

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

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

[15] Lawrence, J.D. and J.D. Starkey. Experimental tests of resolution based theorem-proving strategies. Technical Report, Computer Science Department, Washington State University, Pullman, Washington, April 1974.

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

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

[18] Loveland, D.W. Automated theorem proving: mapping logic into AI. *Proceedings of the International Symposium on Methodologies for Intelligent Systems,* Knoxville, Tennessee, October 1986, 214–229.

[19] Loveland, D.W. Near-Horn Prolog. *Proceedings of the Fourth International Conference on Logic Programming,* Melbourne, Australia, May 1987, 456–469.

[20] Loveland, D.W. and M.E. Stickel. The hole in goal trees: some guidance from resolution theory. *IEEE Transactions on Computers C-25,* 4 (April 1976), 335–341.

[21] Lusk, E.L., W.W. McCune, and R.A. Overbeek. Logic Machine Architecture: kernel functions. *Proceedings of the 6th Conference on Automated Deduction,* New York, New York, June 1982, 70–84.

[22] Lusk, E.L., W.W. McCune, and R.A. Overbeek. Logic Machine Architecture: inference mechanisms. *Proceedings of the 6th Conference on Automated Deduction,* New York, New York, June 1982, 85–108.

[23] Lusk, E.L. and R.A. Overbeek. A portable environment for research in automated reasoning. *Proceedings of the 7th Conference on Automated Deduction,* Napa, California, May 1984, 43–52.

[24] Malachi, Y. *Nonclausal Logic Programming.* Ph.D. Dissertation, Department of Computer Science, Stanford University, March 1986.

[25] Michie, D., R. Ross, and G.J. Shannan. G-deduction. In B. Meltzer and D. Michie (eds.). *Machine Intelligence 7.* John Wiley and Sons, New York, New York, 1972, pp. 141–165.

[26] Nilsson, N.J. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, California, 1980.

[27] Plaisted, D.A. A simplified problem reduction format. *Artificial Intelligence 18,* 2 (March 1982), 227–261.

[28] Plaisted, D.A. Non-Horn clause logic programming without contrapositives. 1987.

[29] Rao, V.N., V. Kumar, and K. Ramesh. A parallel implementation of iterative-deepening A*. *Proceedings of the AAAI-87 National Conference on Artificial Intelligence,* Seattle, Washington, July 1987, 178–181.

[30] Raph, Karl Mark G. The Markgraf Karl Refutation Procedure. Memo SEKI-MK-84-01, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, West Germany, January 1984.

[31] Reboh, R., B. Raphael, R.A. Yates, R.E. Kling, and C. Velarde. Study of automatic theorem-proving programs. Technical Note 75, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California, November 1972.

[32] Shostak, R.E. Refutation graphs. *Artificial Intelligence 7,* 1 (Spring 1976), 51–64.

[33] Slate, D.J. and L.R. Atkin. CHESS 4.5—the Northwestern University University chess program. In Frey, P.W. (ed.)., *Chess Skill in Man and Machine,* Springer-Verlag, New York, New York, 1977, 82–118.

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

[35] 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.

[36] 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.

[37] Warren, D.H.D. An abstract Prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1983.

[38] Wilkins, D.E. QUEST: a non-clausal theorem proving system. M. Sc. Thesis, University of Essex, Essex, England, 1973.

[39] Wilson, G.A. and J. Minker. Resolution, refinements, and search strategies: a comparative study. *IEEE Transactions on Computers C-25,* 8 (August 1976), 782–801.

[40] Wos, L., R. Veroff, B. Smith, and W. McCune. The linked inference principle, II: the user's viewpoint. *Proceedings of the 7th International Conference on Automated Deduction,* Napa, California, May 1984, 316–332.

[41] Wos, L.T. Unpublished notes, Argonne National Laboratory, about 1965.