Theoretical Computer Science 104 (1992) 109-128
Elsevier

A Prolog technology theorem prover: a new exposition and implementation in Prolog*

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

Abstract

Stickel, M.E., A Prolog technology theorem prover: a new exposition and implementation in Prolog, Theoretical Computer Science 104 (1992) 109-128.

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 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. Introduction

A Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus [39]. Its name connotes two

Correspondence to: Mark E. Stickel, Artificial Intelligence Center, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA.

* This is a revised version of a paper presented at the International Symposium on Design and Implementation of Symbolic Computation Systems, Capri, Italy, April 1990.

This research was supported by the National Science Foundation under Grants CCR-8611116 and CCR-8922330. 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.

things: PTTP employs *Prolog technology* in its implementation. It is also a *technology theorem prover* in the same way that TECH was a *technology chess program* [10], i.e., it is a "brute force" theorem prover that relies less on detailed analysis than on high-speed execution of small logical steps and whose capabilities will increase as Prolog machine technology progresses. We present here a new exposition and implementation of PTTP that uses Prolog to explain and implement PTTP.

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 [37];
- a Lisp-based compiler [39];
- F-Prolog, a Prolog-based interpreter [45];
- Expert Thinker, a commercial version of F-Prolog [32];
- Parthenon [3] and METEOR [2], parallel implementations based on the Warren abstract machine and SRI model for OR-parallel execution of Prolog, for shared-memory and nonuniform-access memory machines;
- SETHEO [18] and PARTHEO [33], sequential and parallel Warren abstract machine implementations inspired by the connection method with input-formula preprocessing and additional inference and search strategy options.

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.

Besides being useful for deduction, the model elimination procedure and PTTP can be extended for use in abductive reasoning in diagnosis [30, 6, 7], design synthesis [8], and natural-language interpretation [11, 41, 42]. Adding the capability to "skip" and thereby assume literals instead of prove them permits this extension from deductive to abductive reasoning [24, 34, 13]. The model elimination procedure and PTTP can then play a fundamental role in the computation of default logics [4, 29], circumscription [31, 14], and truth-maintenance systems [12].

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, although still not equal to that of the Lisp-based compiler implementation.

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. PTTP has been run essentially unaltered on the Aurora OR-parallel implementation of Prolog [21]. Even prior to the measurement and tuning we plan to do to optimize its performance on Aurora, PTTP has demonstrated good speedup on large enough problems.

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.

First-order predicate calculus formulas are first translated to Prolog clauses and their contrapositives. The exact input format allowed is a conjunction of assertions that are in negation normal form (possibly nested conjunctions and disjunctions of literals) and a conclusion that is a conjunction of literals. The assertions are implicitly universally 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 other connectives and to do the skolemization. We will not describe the translation process, since it is not specific to PTTP. The resulting Prolog clauses are then transformed to new ones that incorporate sound unification, bounded search, and model elimination inference.

The recipe uses the following three compile-time transformations.

- A 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 does sound unification with the occurs check.

- A 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 bodies of nonunit clauses.
- A 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 bodies of nonunit clauses; additional clauses are added to perform the model elimination reduction and pruning 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 compile-time transformation enables collection of the information required to print the proof after it is found. We will not describe this transformation, since it is not part of PTTP's inference or search procedure, but it does contribute substantially to PTTP's usefulness.

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 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. Group theory problems are sometimes presented as here using the literal `p(X,Y,Z)`, which denotes $x \circ y = z$, where $\circ$ is the group multiplication operation. In the example below, the literal `p(X,Y,f(X,Y))` states that every X and Y have a product `f(X,Y)`).

```
X<(X+1).      p(X,Y,f(X,Y)).
:-Y<Y.        :-p(a,Z,Z).
```

The invalid results rely upon the creation of circular bindings for variables during unification.

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.

Although it is easy to write a Prolog predicate `unify` that performs sound unification with the occurs check [26, 36], 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 [26, 27]. 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):-          p(X,Y,f(X1,Y1)):-
 unify(X,X1).        unify(X,X1),
                     unify(Y,Y1).
```

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

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 if we disregard the incompleteness of Prolog's inference system for theorem proving, Prolog is still 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 0, then depth 1, and so on, until a proof is found. This is called *depth-first iterative-deepening search* [15]. 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 $b/(b-1)$ times as many operations as breadth-first search, where b is the branching factor [43] (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 [15] 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 [16], 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 [39] but may require a test of whether a potentially complementary ancestor exists, which is costly in this implementation (see Sections 4 and 6.3).

A "driver" predicate search can be written easily 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. 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.

In this approach to bounded search, the number of inference steps in the proof (i.e., the length of the proof) is bounded. Other measures on proof size could be bounded instead. Notable alternative measures are the maximum length of the list of ancestor goals (see Section 4) or the maximum number of instances of each clause that are used in the proof, both of which have been used in other systems [9, 18].

None of these measures is uniformly superior to the others. A strong reason for our preferring to bound the number of inference steps in the proof (besides its often performing well) is the relatively slow, smooth growth in the size of the search space as the bound is increased. Increasing the maximum allowed length of the list of ancestor goals, or uniformly increasing the maximum number of instances of each clause that can be used in the proof, can result in an explosive increase in the size of the search space from one bound to the next.

When there is large, unpredictable variability in the results of searches that use alternative cost measures, it can be beneficial to try to prove a theorem by using multiple cost measures in parallel (e.g., by running *n* copies of the theorem prover) and stop as soon as one of the proof attempts succeeds. A major reason why the SETHEO system [18] sometimes significantly outperforms PTTP is its (simulated) parallel execution of these three search strategies (SETHEO's overall computation time is computed as three times the proof time for the fastest strategy).

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 $\neg P \vee Q$

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

are insufficient to reduce `:- q` to the empty list of literals. We represent the complement of the literal p by not_p. Rather than use a negation operator, we use pairs of predicate names p and not_p, q and not_q, and so on.

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 [19, 20] can be viewed as a 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. The *SL resolution* procedure [17] 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 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
```

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).
```

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.

For propositional goals, the ME reduction operation can be optimized:

```
p(Ancestors) :- identical_member(not_p,Ancestors), !.
```

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

In addition, an extra clause at the beginning of each procedure that eliminates some cases of looping has been found to be cost-effective. 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.
```

This pruning operation eliminates the possibility of infinite branches in the case of propositional problems and some more general problems, which can then be solved, if desired, with unbounded instead of bounded search.

A problem with unrestricted use of the reduction operation is some unnecessary redundancy in the search space. For example, in addition to the earlier proof we gave, Q can also be proved from $P \vee Q$ and $\neg P \vee Q$ by

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

Note that when solving not_p by reduction by the ancestor goal p, q is an intervening ancestor goal in the first proof and not_q is an intervening ancestor goal in the second. Whenever a reduction operation applies to a goal and an ancestor goal with intervening ancestor goals $Q_1, \ldots, Q_n$, there is an alternative deduction with intervening ancestor goals $\neg Q_n, \ldots, \neg Q_1$ (i.e., the complements of the goals, in reverse order). By assigning labels from $\{-1, 0, +1\}$ in a particular way to literals in the contrapositives, and allowing reduction only if the labels of the goal to be reduced and the intervening ancestor goals sum to greater than zero, the *foothold format* will eliminate one of the two alternative deductions [35].

A *positive refinement* of model elimination also reduces the number of allowable reduction operations [28]. In this refinement, only goals that are positive literals are eligible for solution by the reduction operation, and thus only negative-literal ancestor goals need be recorded. The need to record fewer ancestors can make inference faster and also make caching intermediate results more feasible, since a cache "hit" could occur for goals with the same negative ancestors instead of for goals with all the same ancestors.

We have tried these refinements, but they are not part of "standard" PTTP. They have some disadvantages as well as their obvious advantages, and the tradeoffs have not yet been fully evaluated. The foothold format adds some complexity and run-time cost. Both refinements sometimes eliminate the shortest model elimination proof and make it necessary to search deeper to find a proof (there may still be a net benefit [25]). The positive refinement makes it more frequently necessary to add the negation of the query as an additional assertion; PTTP's model elimination procedure requires this only when seeking indefinite answers.

Indefinite answers are a feature of full first-order predicate calculus theorem proving that is absent in Prolog. 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 theorem proving with non-Horn clauses, 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) :- not_p(b).
p(b) :- not_p(a).
:- p(X).
```

These assertions and the described inference procedure are still insufficient to solve the problem. 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 not_p(Y) results in the discovery of two proofs (p(X) can be matched with p(a) and not_p(Y) with not_p(b), and vice versa for the second proof). 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 not_p(Y) need be added, since any assertion can always be used arbitrarily many times with different instantiations. To extract an indefinite answer from a proof, 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 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 partly recompiled when indefinite answers are sought.

If PTTP were to use the positive refinement of model elimination instead of the standard model elimination procedure, the negation of the query would have to be included even in the case of definite answers. For example, in proofs of $\neg P$ from $\neg P \vee Q$ and $\neg P \vee \neg Q$, the subgoal P cannot be solved by reduction in the positive refinement, so it must be solved by use of the query's negation P.

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

For example, the clauses

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

can be replaced by the single clause

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

This can result in a substantially diminished search space. Wilkins [46] developed the first nonclausal version of the model elimination procedure and nonclausal formulas are also a vital feature of the TABLOG logic programming language [23].

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) :- not_p(b).
p(b) :- not_p(a).
not_p(a).
not_p(b).
:- p(a); p(b).
```

where the extra clauses not_p(a) and not_p(b) comprise the negation of the query. Further refinement of the inference system may make it unnecessary to include the negation of the query among the assertions.

Another presentation of the model elimination procedure and its implementation in the manner of Prolog can be found in Maier and Warren [22].

5. Example

Following is a sample model elimination proof found by PTTP. This is Example 8 from Chang and Lee [5, pp. 298-305], for which statistics are presented in Table 1.

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 search 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.

```
PTTP input formulas:
  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...
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...
Proof:
Goal#  Wff#  Wff Instance
-----  ----  ------------
  [0]   10   query :- [1], [13].
  [1]    4      p(a) :- [2].
  [2]    8         not_l(1,g(a)) :- [3], [5].
  [3]    5          l(g(a),a) :- [4].
  [4]  red               not_p(a).
  [5]    6            not_p(f(g(a))) :- [6].
  [6]    7               d(f(g(a)),a) :- [7] , [11].
  [7]    9                  d(f(g(a)),g(a)) :- [8] , [9].
  [8]  red                     l(1,g(a)).
  [9]    5                     l(g(a),a) :- [10].
 [10]  red                        not_p(a).
 [11]    3                   d(g(a),a) :- [12].
 [12]  red                      not_p(a).
 [13]    2       d(a,a).
3830 inferences.
```

A 13-step proof is discovered during the cost 11 search; each step eliminates a single goal by resolution of reduction. The query has 2 goals and 11 are introduced by resolution operations, totalling 13. 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.

6. Evaluation

The cost of PTTP compared to Prolog in terms of size of the input can be determined as follows.

- 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.
- 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 to compute the information needed to print the proof after it is found.

Table 1 gives results for the examples that appear in Chang and Lee [5, pp. 298–305], for both the Lisp implementation [39] 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.

6.1. 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.

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

Table 1
PTTP Performance on Chang and Lee examples

Example	Number of clauses	Depth of proof	Lisp implementation		Prolog implementation	
			Number of inferences	Run time (sec)	Number of inferences	Run time (sec)
1	5	4	5	0.002	5	0.005
2	7	10	1589	0.373	1938	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	3104	0.652	3830	2.522
9	8	10	163	0.027	191	0.135
Total			5147	1.115	6320	3.422

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. 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.

6.2. *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.

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 optionally counting the number of inferences so that the total can be printed 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.

6.3. *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 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. So, 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. The ME reduction operation will then always check for membership of a goal in an empty list of positive-literal ancestors when the problem is Prolog-like, i.e., Horn clauses with negative query.

Further indexing of ancestor goals would be beneficial. 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. Separate lists for each signed predicate 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. 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 by adding extra arguments to predicates (e.g., one for each global variable).

7. 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.

PTTP's high inference rate is achieved at the cost of not allowing more flexible search strategies or elimination of redundancy in the search space by subsumption. Although PTTP is one of the fastest theorem provers in existence when evaluated by its inference rate and performance on easy problems, and it has been used to solve reasoning problems in planning and natural-language-understanding systems effectively, its high inference rate can be overwhelmed by its exponential search space and it is unsuitable for many difficult theorems for which conventional theorem provers have demonstrated some success.

Besides being used as a stand-alone theorem prover, PTTP can play a useful subordinate role in the proof of difficult theorems if the theorem can be decomposed into manageable chunks [44], by performing fast refutation checks on newly derived clauses [1], or by executing the theory resolution [38] or linked inference principle [47] procedures. We are currently investigating the latter approach by developing an extension of PTTP that, instead of proving a query outright, finds single literal assumptions that would suffice to complete a proof. This "Unit-Resulting PTTP" can then perform by fast, compiled inference operations essentially the computation of linked unit-resulting resolution and can be used in a larger deduction system in the same manner.

A technical report contains full source code and sample output for PTTP in Prolog [40].

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 in the technical report as well.

References

[1] G. Antoniou and H.J. Ohlbach, Terminator, in: *Proc. 8th Internat. Joint Conf. on Artificial Intelligence*, Karlsruhe, Germany, August 1983, 916-919.

[2] O. Astrachan, METEOR: model elimination theorem proving for efficient OR-parallelism, Masters Thesis, Department of Computer Science, Duke University, 1989.

[3] S. Bose, E.M. Clarke, D.E. Long and S. Michaylov, Parthenon: a parallel theorem prover for non-Horn clauses, in: *Proc. 4th IEEE Symp. on Logic in Computer Science*, Asilomar, CA, June 1989.

[4] M.A. Casanova, R. Guerreiro and A. Silva, Logic programming with general clauses and defaults based on model elimination, in: *Proc. 11th Internat. Joint Conf. on Artificial Intelligence*, Detroit, MI, August 1989, 395-400.

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

[6] P.T. Cox and T. Pietrzykowski, Causes for events: their computation and applications, in: *Proc. 8th Conf. on Automated Deduction*, Oxford, July 1986, 608-621.

[7] P.T. Cox and T. Pietrzykowski, General diagnosis by abductive inference, in: *Proc. 1987 Symp. on Logic Programming*, San Francisco, CA, August 1987, 183-189.

[8] J.J. Finger, Exploiting constraints in design synthesis, Ph.D. dissertation, Department of Computer Science, Stanford University, Stanford, CA, February 1987.

[9] S. Fleisig, D. Loveland, A. Smiley and D. Yarmush, An implementation of the model elimination proof procedure, *J. ACM* **21** (1974) 124-139.

[10] J.J. Gillogly, The technology chess program, *Artificial Intelligence* **3** (1972) 145-163.

[11] J.R. Hobbs, M. Stickel, D. Appelt and P. Martin, Interpretation as abduction. Technical Note 499, Artificial Intelligence Center, SRI International, Menlo Park, CA, December 1990.

[12] K. Inoue, Procedural interpretation for an extended ATMS, Technical Report TR-547, Institute for New Generation Computer Technology, Tokyo, March 1990.

[13] K. Inoue, Consequence-finding based on ordered linear resolution, in: *Proc. 12th Internat. Joint Conf. on Artificial Intelligence*, Sydney, Australia, August 1991, to appear.

[14] K. Inoue and N. Helft, On theorem provers for circumscription, in: *Proc. 8th Biennial Conf. of the Canadian Society for Computational Studies of Intelligence*, Ottawa, Canada, May 1990, 212-219.

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

[16] R.E. Korf, Iterative-deepening A*: an optimal admissable tree search, in: *Proc. 8th Internat. Joint Conf. on Artificial Intelligence*, Los Angeles, CA, August 1985, 1034-1036.

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

[18] R. Letz, J. Schumann, S. Bayerl and W. Bibel, SETHEO: a high-performance theorem prover, *J. Automated Reasoning*, to appear.

[19] D.W. Loveland, A simplified format for the model elimination procedure, *J. ACM* **16** (1969) 349-363.

[20] D.W. Loveland, *Automated Theorem Proving: A Logical Basis* (North-Holland, Amsterdam, 1978).

[21] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepelewski and B. Hausman, The Aurora or-parallel Prolog system, *New Generation Comput.* **7** (1990) 243-271.

[22] D. Maier and D.S. Warren, *Computing with Logic* (Benjamin/Cummings, Menlo Park, CA, 1988).

[23] Y. Malachi, Nonclausal logic programming, Ph.D. Dissertation, Department of Computer Science, Stanford University, March 1986.

[24] E. Minicozzi and R. Reiter, A note on linear resolution strategies in consequence finding, *Artificial Intelligence* **3** (1972) 175-180.

[25] X. Nie, Model elimination and its positive refinement, *AAR Newsletter* **15** (May 1990) 2-3.

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

[27] D.A. Plaisted, Non-Horn clause logic programming without contrapositives, *J. Automated Reasoning* **4** (1988) 287-325.

[28] D.A. Plaisted, A sequent-style model elimination strategy and a positive refinement, *J. Automated Reasoning* **6** (1990) 389-402.

[29] D. Poole, A logical framework for default reasoning, *Artificial Intelligence* **36** (1988) 27-47.

[30] H.E. Pople, Jr., On the mechanization of abductive logic, in: *Proc. 3rd Internat. Joint Conf. on Artificial Intelligence*, Stanford, CA, August 1973, 147-152.

[31] T.C. Przymusininski, On algorithm to compute circumscription, *Artificial Intelligence* **38** (1989) 49-73.

[32] R.W. Satz, Expert Thinker software package. Transpower Corporation, Parkerford, PA, 1988.

[33] J. Schumann and R. Letz, PARTHEO: a high-performance parallel theorem prover, in: *Proc. 10th Internat. Conf. on Automated Deduction*, Kaiserslautern, Germany, July 1990, 40-56.

[34] P. Siegel, Représentation et utilisation de la connaissance en calcul propositionnel, Thèse d'Etat, Université de Aix-Marseille II, 1987.

[35] B. Spencer, Avoiding duplicate explanations, in: *Proc. North-American Conf. on Logic Programming*, Austin, TX, October 1990.

[36] L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, Cambridge, MA, 1986).

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

[38] M.E. Stickel, Automated deduction by theory resolution, *J. Automated Reasoning* **1** (1985) 333-355.

[39] M.E. Stickel, A Prolog technology theorem prover: implementation by an extended Prolog compiler, *J. Automated Reasoning* **4** (1988) 353-380.

[40] M.E. Stickel, A Prolog technology theorem prover: a new exposition and implementation in Prolog, Technical Note 464, Artificial Intelligence Center, SRI International, Menlo Park, CA, June 1989.

[41] M.E. Stickel, A Prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation, *Ann. Math. Artificial Intelligence*, to appear.

[42] M.E. Stickel, Rationale and methods for abductive reasoning in natural-language interpretation, in: *Proc. IBM Symp. on Natural Language and Logic*, Hamburg, Germany, May 1989.

[43] M.E. Stickel and W.M. Tyson, An analysis of consecutively bounded depth-first search with applications in automated deduction, in: *Proc. 9th Internat. Joint Conf. on Artificial Intelligence*, Los Angeles, CA, August 1985, 1073-1075.

[44] M. Tarver, An examination of the Prolog technology theorem prover, in: *Proc. 10th Internat. Conf. on Automated Deduction*, Kaiserslautern, Germany, July 1990, 322-335.

[45] Z.D. Umrigar and V. Pitchumani, An experiment in programming with full first-order logic, in: *Proc. 1985 Symp. on Logic Programming*, Boston, MA, July 1985, 40-47.

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

[47] L. Wos, R. Veroff, B. Smith and W. McCune, The linked inference principle, II: the user's viewpoint, in: *Proc. 7th Internat. Conf. on Automated Deduction*, Napa, CA, May 1984, 316-332.