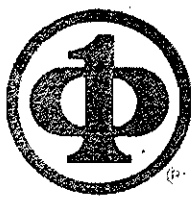
A PROLOG TECHNOLOGY THEOREM PROVER

Mark E. Stickel

TECHNICAL NOTE 336

Reprinted from IEEE 1984 INTERNATIONAL SYMPOSIUM
ON LOGIC PROGRAMMING



IEEE COMPUTER SOCIETY
1109 Spring Street, Suite 300
Silver Spring, MD 20910

IEEE
COMPUTER
SOCIETY
PRESS

A PROLOG TECHNOLOGY THEOREM PROVER

Mark E. Stickel

TECHNICAL NOTE 336

Reprinted from IEEE 1984 INTERNATIONAL SYMPOSIUM
ON LOGIC PROGRAMMING

IEEE COMPUTER SOCIETY
1109 Spring Street, Suite 300
Silver Spring, MD 20910

A PROLOG TECHNOLOGY THEOREM PROVER

Mark E. Stickel

Artificial Intelligence Center
SRI International
Menlo Park, California 94025

Abstract

An extension of Prolog, based on the model elimination theorem-proving procedure, would permit production of a logically complete Prolog technology theorem prover capable of performing inference operations at a rate approaching that of Prolog itself.

1. Introduction

Prolog is a powerful and versatile programming language based on theorem-proving unification and resolution operations.

The best Prolog implementations perform inferences at a rate that is often at least two orders of magnitude faster than theorem provers. Some of this disparity in speed can be accounted for by the fact that theorem provers often perform more complex inferences than Prolog (such as keeping results in fully simplified form and checking for subsumption).

However, one important reason for the higher speed of Prolog, compared with theorem provers, is the implementation. Given the present efficiency advantage of Prolog over theorem provers, and the fact that enormously more powerful Prolog machines are being contemplated (up to 10^9 logical inferences per second (lips) as opposed to the current best $10^4 - 10^5$ lips), it is worthwhile to examine the possibilities of adapting Prolog technology to theorem proving.

Prolog technology could be applied to theorem proving in a number of ways. To date, the most frequently used method of applying Prolog technology to theorem-proving problems is to substantially recode the problem in Prolog. Although performance may be high, this approach has significant limitations resulting from such implementation features of Prolog as unification without the "occurs check" and unbounded depth-first search. Also, the recoding process itself is time-consuming and error prone.

Prolog technology could be used in theorem proving by writing a theorem prover in Prolog, but this offers uncertain advantages in comparison with writing a theorem prover in any other language, such as LISP. Writing a theorem prover in Prolog would certainly result in a theorem prover whose inference operations are performed at a markedly lower rate than Prolog's own, since several Prolog inference operations would have to be performed for each theorem-proving inference operation.

Prolog, as it now exists, almost meets the requirements for a complete theorem prover. Thus, we propose implementation of a slight extension of Prolog that permits full theorem proving directly. Direct modification of a Prolog interpreter, rather than coding a theorem prover in Prolog, preserves the speed of the Prolog interpreter by making extended Prolog operations be theorem-proving operations.

We are taking a fairly conservative approach to the extension of Prolog implementations for theorem proving. Simple additions to the Prolog interpreter should suffice to make the complete theorem prover—thus making the theorem prover easy to implement and similar to Prolog in its use. We retain such features of Prolog as the ordering of alternative inferences by statically ordering assertions in the database, the ordering of subgoals by statically ordering literals in assertions, and the cut operation. These features should be useful for programming a theorem prover just as they are for logic programming. Depth-first search, though bounded, will continue to be employed both for its comprehensibility and low storage requirements. Prolog also provides a convention for procedural attachment (built-in predicates) that should be useful in theorem proving as well.

We have two things in mind in presenting this design for a *Prolog technology theorem prover (PTTP)*. The first is that it employs highly efficient *Prolog technology* in its implementation. The second is that it is a *technology theorem prover* in the same way that TECH was a *technology chess player*[4]. It is a "brute force" theorem prover

This research was supported by the Defense Advanced Research Projects Agency under Contract N00039-80-C-0575 with the Naval Electronic Systems Command. The views and conclusions contained in this document are those of the author and should not be interpreted as representative of the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States government. APPROVED FOR PUBLIC RELEASE. DISTRIBUTION UNLIMITED.

CH2007-3/84/0000/0212 $01.00 © 1984 IEEE

that relies less on detailed analysis than on high-speed execution of small logical steps. The capability of a PTTP would increase substantially as Prolog machine technology progresses.

We are currently experimenting with the concept of a PTTP that uses an extended Prolog interpreter (without all the Prolog built-in predicates) written in LISP with the same unification and substitution code employed in our other theorem-proving research. This allows experimentation with extended unification algorithms, but means that we do not yet have the efficiency of a true PTTP because the Prolog-style substitution representation is not being used.

2. A Minimal Prolog Technology Theorem Prover

Although Prolog uses unification and resolution for its matching and inference processes, it cannot be regarded as a full-fledged theorem prover. The deficiencies[1] lie in three areas:

- Unification without the occurs check
- Incomplete inference system
- Unbounded depth-first search strategy.

We will examine each of these problems in more detail and offer minimal solutions to them. The result will be the design of a minimal PTTP.

2.1 Unification

Prolog matching differs from the theorem-proving unification operation in only one respect: the absence in the former of the occurs check. In the theorem-proving unification operation, a variable is permitted to be instantiated to a term only if the variable does not occur in the term. This restriction eliminates the creation of infinite terms. The logical importance of this restriction is evident from the fact that without the occurs check it is possible to "prove" that $\forall x \exists y . P(x, y)$ implies $\exists y \forall x . P(x, y)$.[2] To prove this invalid result in Prolog, we match the skolemized form $P(sk2(y), y)$ of the goal $\exists y \forall x . P(x, y)$ and the skolemized form $P(x, sk1(x))$ of the assertion $\forall x \exists y . P(x, y)$. This match is successful without the occurs check.

It is clear that adding a straightforward occurs check to Prolog matching would impose unacceptable performance penalties on the operation of many logic programs. The lesser deduction depth and term complexity in typical theorem-proving applications would probably make it acceptable to add the occurs check. Furthermore, there are some easily verified circumstances in which the occurs check is unnecessary. When matching a goal with a clause head, it is unnecessary to perform the occurs check for the first variable binding; it is also unnecessary if the goal or the clause head has no variable occurring more than once. Use of the occurs check could be controlled by a run-time or compile-time switch.

An alternative approach to using the occurs check in each matching operation is a provision for checking at the completion of a proof to verify that no infinite term was created in the course of that proof.[3] Note that this approach requires that all bindings created during the course of a proof be available for checking upon its completion. This may not be the case for some Prolog implementations.

Either of these approaches should be easy to incorporate in an implementation of Prolog. There is a trade-off involved in the choice of approach. The first adds overhead to each unification but immediately blocks inferences using infinite terms. The second has little or no overhead for each unification but may permit many inferences to be drawn after an infinite term is created; these inferences could have been cut off by immediately using the occurs check.

2.2 Inference System

As is well known, the inference system used in Prolog is complete only for Horn sets of clauses, i.e., sets of clauses in which there is no more than one positive literal in each clause. We present a method of extending the Prolog inference system to a complete inference system that retains most of the character and efficiency of Prolog deduction.

In developing a PTTP, we should consider only those means for extending Prolog's inference system that permit highly efficient Prolog implementation techniques to be used. We observe that one of the most important reasons for the high speed of well-engineered Prolog implementations is the efficiency of their representation for variable substitutions. This representation is made possible both by the depth-first search strategy and by Prolog's use of a form of input resolution as its inference procedure.

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

The second method is the *structure-sharing* approach [1], 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 method consumes far less space than the simple method of fully forming the

[1] While these are deficiencies from the standpoint of theorem proving, they are often assets in logic programming because they increase efficiency or comprehensibility of Prolog programs.

[2] I am indebted to Bob Moore for this observation.

[3] I first heard this suggestion from David Warren.

resolvents, but is still not very efficient in time, compared with Prolog. The reason for this relative inefficiency is clear.

In general resolution, a variable of an input clause may have more than one value per use of the clause in a deduction because the clause is implicitly reused whenever a descendant clause is used more than once. For example, if we resolve $P(x)$ and $\neg P(y) \vee Q(y)$, setting y to x, we obtain $Q(x)$. This resolvent can now be used twice to derive the empty clause from $\neg Q(a) \vee \neg Q(b)$. But this means that two instances of $P(x)$, $P(a)$ and $P(b)$, have been implicitly used in the proof, even though $P(x)$ was used explicitly only once. The substitution representation must accommodate these multiple variable values, whereas in Prolog the variable x can be implemented as a stack location containing [a pointer to] its single current value. The problem of multiple variable values does not occur in input resolution because derived clauses can only be used once. If it is assumed that each input clause is treated as a new clause with distinct variables as it is used, each variable will have only a single value in a single deduction.

This suggests that a good approach to building a PTTP is to employ a complete inference system that is an input procedure. Probably the simplest is the *model elimination* procedure [7, 8]. (Actually, what we are proposing here is more closely related to the problem-reduction-oriented MESON procedure [8, 9], but we will use the term model elimination (ME) because it is more familiar and the MESON procedure is derived from the ME procedure.)

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.

An important thing to note is that this is a complete inference system that does not require the theorem-proving *factoring* operation. Basing an extension of Prolog on another form of model elimination, equivalent to SL-resolution [6], would require an additional factoring operation that would instantiate pairs of goals to be identical. Eder's Prolog-like interpreter for non-Horn clauses [2] also requires factoring. (However, we have not yet addressed Eder's concern regarding the type of search space redundancy that results in two proofs, not just one, of $\exists x.P(x)$ from $P(a) \vee P(b)$.)

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 operation that operates on the current goal and an input clause, and the reduction operation that operates on the current goal and an ancestor goal that is available on the stack, the factoring operation must operate on the current goal and an unsolved subgoal of an ancestor goal that is not itself an ancestor goal, i.e., a goal that is not currently being solved and thus is not on the stack, except in the list of remaining unopened subgoals of its parent goal. The factoring operation, though necessary for completeness of many inference systems, has a tendency to instantiate goals excessively, thereby eliminating any possibility of solution.

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 $\neg P$ is true (i.e., $\neg P \supset Q$), 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.

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 executing the body of the clause if the match is successful. In this extension of Prolog, we must also consider the additional alternatives of matching the entered goal with 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 (a clause with an empty body).

In Prolog, when a goal is exited, the goal, instantiated by the current substitution, has been proved. In this extension of Prolog, when a goal is exited, all that has been proved is the instantiation of the goal disjoined with all the ancestor goals used in reduction operations in the process of "proving" the goal. Thus, in the example of proving P from $Q \supset P$ and $\neg P \supset Q$, expressed in Prolog by

```
p :- q.
q :- ¬p.
?- p.
```

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

One of the implementation requirements imposed by the addition of the reduction operation is that ancestor goals must be accessible. This precludes some optimizations such as a tail-recursive-call optimization that reuses the top stack frame when the next step is determinate and thus erases the current goal so that it cannot be used in a reduction operation.

There are two additional prerequisites for using this inference system. First, contrapositives of the assertions must be furnished. For each assertion with n literals, n Prolog assertions must be provided so that each literal is

the head of one of the Prolog assertions. The order of the literals in the clause body can be freely specified by the user, as for ordinary Prolog assertions.

The second additional prerequisite relates to a feature of theorem proving that is absent in Prolog deduction: indefinite answers. Prolog, when provided with the goal $P(x)$, will attempt to generate all terms t such that $P(t)$ is definitely known to be true. In non-Horn clause theorem proving, there may be indefinite answers.

For example, consider proving $\exists x.P(x)$ from $P(a) \vee P(b)$. In our extension to Prolog, this can be expressed 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 because there is no term t for which it is definitely known that $P(t)$ is true. To solve problems with indefinite answers, it is necessary to add the negation of the query as another assertion (n assertions if the query has n literals).

In this example, addition of the Prolog assertion ¬p(Y) results in the finding of two proofs (one in which p(X) is matched with p(a) and ¬p(Y) is matched with ¬p(b), one in which p(X) matched with p(b) and ¬p(Y) is matched with ¬p(a)). The answer to the query is thus $P(a) \vee P(b)$, i.e., 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 (the use of the query as the initial list of goals and each use of the negation of the query).

2.3 Search Strategy

Even if the problems of unification without the occurs check and an incomplete inference system are solved, or are irrelevant for a particular problem, Prolog is still unsatisfactory as a theorem prover because of its unbounded depth-first search strategy.

Consider the problem of proving that, in a monoid, if $x \times x$ is the identity element for every x, then $\times$ is commutative. This is often formulated in terms of the ternary predicate P, where $P(x, y, z)$ means $x \times y = z$ (this is quite consistent with Prolog relational programming style). The problem can then be expressed in Prolog by the following assertions and goal:

```
p(X,e,X).                                          % x × e = x
p(e,X,X).                                          % e × x = x
p(X,X,e).                                          % x × x = e
p(a,b,c).                                          % a × b = c
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).          % assoc. 1
p(X,V,W) :- p(X,Y,U), p(Y,Z,V), p(U,Z,W).          % assoc. 2
?- p(b,a,c).                                       % b × a = c
```

For this problem, Prolog's lack of the occurs check in unification and incomplete inference system do not matter, because no nonconstant function symbols appear and the set of clauses is a Horn set. However, Prolog will still fail to solve the problem because its unbounded depth-first search strategy will cause infinite recursion using the first associativity rule.

The minimal solution to the problem is to use bounded rather than unbounded depth-first search. Backtracking when reaching the depth bound will cause the entire search space, up to a specified depth, to be searched completely.

Because the search space size grows exponentially as the depth bound increases, assigning too large a depth bound for a particular problem may result in an enormous amount of wasted effort, and the amount of effort expended before discovering a proof will be highly dependent on the specified depth bound. The obvious solution to this problem is to run a PTTP with increasing depth bounds—first one tries to find a proof with depth 1, then 2, etc. We will call this the *staged depth-first search strategy*. Because of the exponential growth of the size of the search space as the depth bound increases, the cost of searching all of levels $1, \ldots, n$ before first finding a proof at level $n+1$ will probably not be unacceptably high relative to the cost of just searching at level $n+1$.[4]

Rather then make all inferences up to level n, we should make only those that have some chance of resulting in a proof by level n. Because each as yet unsolved goal will require at least one inference step to solve it, we should not perform any inference step that would result in there being more unsolved goals than there are levels remaining before the depth bound is reached.

This approach has some other consequences for logic programming. The use of depth-bounded search changes the meaning of failure from "not provable" to "not provable within depth bound", thus requiring rejection or modification of the treatment of failure as negation. The use of depth-bounded search with increasing depth bound also will cause side effects to be repeated, because deduction steps occurring in the level n search will be repeated in the level $n+1$ search.

[4] Assuming that the search space has a uniform branching factor b, $S(b,n) = b^n + b^{n-1} + \cdots + b^2 + b$ is the number of inferences made in exhaustively searching through level n and $SS(b,n) = b^n + 2b^{n-1} + \cdots + (n-1)b^2 + nb$ is the cumulative number of inferences made in exhaustively searching through level $1, 2, \ldots, n$. Then $S(b, n+1) = b^{n+1} + S(b,n) = b^{n+1} + SS(b,n) - SS(b,n-1)$ and $S(b,n+1) - SS(b,n) = b^{n+1} - SS(b,n-1) = b^n(b - \frac{1}{b} - \frac{2}{b^2} - \cdots - \frac{n-1}{b^{n-1}})$ implying that the cost of exhaustively searching through level $n+1$ usually greatly exceeds the accumulated costs of exhaustively searching through all of the previous levels.

3. Refinements

3.1 Goal Acceptability

The ME procedure justifies the completeness of our extension of Prolog even if some goal states are disallowed. Let us call a goal currently being worked on (either it or one of its subgoals is the current goal) an *open goal*. An *unopened goal* is a goal not yet open in the current deduction. A *closed goal* is a goal that has been exited in the current deduction.

Our extension of Prolog remains complete even if we allow the current goal to be failed under any of the following circumstances:

- Two unopened goals from the same clause are complementary
- A goal is identical to an ancestor goal
- A goal extended upon is complementary to an ancestor goal.

The first rule is justified because, in that situation, a tautologous instance of the clause is being used. Completeness is preserved if tautologous input clauses are not used. The second rule requires a more detailed justification, but in essence states that it is unnecessary to attempt to solve a goal while in the process of attempting to solve that same goal. The third rule merely affirms that it is unnecessary to attempt to solve a goal that is complementary to an ancestor goal by any means other than the reduction operation.

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 applying the check to determine whether the present deduction is acceptable according to the criteria.

The ME procedure applicability tests enumerated above can be expensive to apply. Because each inference operation is potentially capable of instantiating any goal, one of the conditions for unacceptability may become true for a pair of goals after any inference operation. Thus, after each inference operation we would have to check each pair of unopened goals from the same clause for complementarity, and each goal and its ancestor goals for identity and complementarity. The latter is $O(n^2)$, where n is the number of ancestor goals.

There are two solutions to the high cost of these applicability tests. The first is to develop an implementation that can perform these tests cheaply. One method would be to keep track of which pairs of goals could conceivably be instantiated to identity or complementarity and check only those pairs. However, this would make it more difficult to adapt present Prolog implementations to be a PTTP.

The second solution is to restrict the applicability tests. First, we would eliminate the test for complementarity of unopened goals from the same clause. Besides saving the effort of performing the test, this eliminates the requirement for accessing unopened goals. The checking of a goal and its ancestor goals for identity and complementarity can be restricted to the case where the goal is the current goal; this is done after instantiation by the substitution for the contemplated inference operation. This single check is still quite successful in cutting off search at less cost (linear in the number of ancestor goals) than the fuller check.

Another possible effort-saving restriction on the applicability tests would be to perform them less frequently than after every inference operation.

The previous theorem prover that most closely resembles a PTTP (in operation but not in implementation or speed) is an implementation of the ME procedure by Fleisig et al [3]. They concluded that ME was a competitive procedure; neither the ME theorem prover nor a unit preference and set-of-support resolution theorem prover they also developed strongly dominated the other for their examples. Their ME theorem prover uses full acceptability checking and a bounded depth-first search strategy. Unlike our staged depth-first search strategy, a single depth bound is given by the user, making performance very sensitive to the depth bound.

The Fleisig theorem prover also provides for cutoffs by allowing restrictions to be placed on the depth of function nesting, the number of open goals (or number of ancestor goals) in a deduction, the number of uses of particular clauses in a deduction, and the number of uses of clauses of specified length in a deduction. Such cutoffs can also be employed in a PTTP. Although they may ultimately be necessary to reduce the size of the exponential search space for difficult problems, we are somewhat wary of such cutoffs because they are sensitive parameters whose values are difficult to assign. To be useful, the cutoffs must be assigned small values, but not so small as to preclude all proofs. When there are many such parameters, there may be little guidance on which parameter values to alter to admit more inferences when no proof is found with one set of parameter values.

3.2 Extended Unification

It is sometimes quite useful to extend the unification algorithm. For example, building associativity and/or commutativity into the unification algorithm can result in significantly improved performance. Extended unification can also be used for helping to produce systems that reason effectively with equality, taxonomies, ordering, etc. [11]. Kornfeld's work on building uses of equality into Prolog

to support object-oriented programming is a further example [5]. Unlike his work however, full support for extended unification must accommodate the possible presence of multiple unifiers. This means additional alternatives at each choice point—alternative unifiers as well as alternative inferences. The clearest implementation of this would require that all alternative unifiers for an inference be tried before the next alternative inference is tried. It would also be useful to have an additional cut operation that cuts off alternative unifiers but not alternative inferences.

3.3 Operation Ordering

We retain the operation ordering of Prolog (solving subgoals from left to right; using clauses in order from the database) for familiarity, comprehensibility, and programmability. However, the addition of the reduction operation means that the reduction operation must be fitted in somewhere among the other operations. It must be decided whether reduction operations should be performed before, after, or interleaved with extension operations (e.g., after all extensions by unit clauses). This can be specified *a priori* or, perhaps, for each predicate P by including a clause "p :- reduce." in the procedure for P at the point where we wish reduction operations to be attempted (which could also make reduction optional). The order of reduction operations among themselves must also be decided—for example, whether to reduce by the shallowest or deepest ancestor goals first.

It is also worth pointing out that it is unnecessary to consider alternative inference operations if a reduction operation is possible where the two goals are already complementary (the empty substitution unifies their atoms) or if an extension operation by a unit clause instantiates only the clause and not the goal. Discarding alternative inference operations in these situations can save substantial effort.

The fullest possible benefit of this would be obtained if all reduction and unit extension operations were checked first to determine whether the current goal can be solved immediately without further instantiation before performing any inference operation that either further instantiates the current goal or adds subgoals. This suggests a two-pass procedure for attempting to solve a goal: checking ancestor goals for exact complementarity and for subsuming unit clauses; then, if that fails, performing the normal inference operations.

3.4 Additional Inference Operations

It is possible to consider adding more inference operations to a PTTP beyond the extension and reduction operations it minimally requires. We have already considered and rejected the idea of including a factoring operation.

A more useful operation to add may be the graph construction procedure *C-reduction* operation [10]. If the current goal matches a closed goal in the current deduction, the substitution can be applied and the current goal considered as solved, provided that all the ancestor goals used in reduction operations to solve the closed goal are also ancestors of the current goal. The C-reduction operation is similar to the factoring operation, but is superior to it because, since it matches the current goal with a closed goal rather than an unopened goal, (1) unopened goals need not be examined and (2) it is less likely to cause over-instantiation because the closed goal has been solved whereas in the case of factoring, neither goal has been solved and instantiating them to be identical may make them unsolvable.

4. Conclusion

We have presented the design of a minimal Prolog technology theorem prover and numerous possible refinements. Further experimentation will determine how worthwhile this concept is. Numerous questions, of course, remain. We have based our design on the ME procedure because that appears to be the procedure best suited to the use of present Prolog-style implementation. Is this the most effective procedure for us? How useful is it when viewed as a logic programming language? Will the lack of subsumption, equality reasoning, or other features impose too great a limit on its effectiveness? If necessary, how can we add such features while retaining the speed advantage?

In any case, production of a PTTP would result in a theorem prover capable of performing inferences at a far greater speed than before and offering prospects of even greater speed as Prolog machine technology progresses. It is surely a concept that is worth exploring.

Acknowledgments

The author would like to thank Fernando Pereira, Mabry Tyson, and David Warren for their helpful comments on an earlier draft of this paper.

References

[1] 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.

[2] 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.

[3] Fleisig, S., D. Loveland, A.K. Smiley III, and D.L. Yarmush. An implementation of the model elimination

proof procedure. *J. ACM 21*, 1 (January 1974), 124–139.

[4] Gillogly, J.J. The technology chess program. *Artificial Intelligence 3*, 3 (Fall 1972), 145–163.

[5] Kornfeld, W.A. Equality for Prolog. *Proc. Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, August 1983.

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

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

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

[9] Loveland, D.W. and M.E. Stickel. The hole in goal trees: some guidance from resolution theory. *Proc. Third International Joint Conference on Artificial Intelligence*, Stanford, California, August 1973, 153–161. Reproduced in *IEEE Transactions on Computers C-25*, (April 1976), 335–341.

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

[11] Stickel, M.E. Theory resolution: building in nonequational theories. *Proc. AAAI-83 National Conference on Artificial Intelligence*, Washington, D.C., August 1983.