Gracefully adding negation and disjunction to Prolog

David L Poole
Randy Goebel

Research Report CS-85-51
December 1985
Revised March 1986

Gracefully adding negation and disjunction to Prolog

David L Poole
Randy Goebel

Logic Programming and Artificial Intelligence Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract

We show how one can add negation and disjunction to Prolog, with the property that there is no overhead in run time if we do not use the negation, and we only pay for the negation when we actually use it. The extension is based on Loveland's MESON proof procedure, which requires that a negative ancestor search and availability of contrapostive forms of formulae be added to Prolog. We identify a property of literals that can be statically determined, in order to avoid using the full generality of the full clausal proof procedure when not required.

1. Introduction

There are two reasons for wanting to add negation to Prolog. The first is a desire to express incomplete knowledge of some domain. For example we may know that someone's professor is either David or Robin, but we don't know which. The second is that we may want our system to have predictive power [Poole86]. That is, we desire a system that maintains a consistent refutable theory which should have predictive power. This requires an ability to demonstrate inconsistency, or to derive a contradiction.

In Poole et al. [Poole86] we exploit a general method of adding negation and disjunction to Prolog. This is the known technique of using the contrapositives of all formulae and searching back up the relevant "and" branches of the resolution proof tree for the complement of each newly generated subgoal [Umrigar85, Loveland78].

There is a folklore that Horn clauses can be made efficient, but if we add negation, then we get to a full resolution system [Robinson65] which are inherently inefficient. When constructing a system for diagnosing faults (based on DART [Genesereth85]) in electrical circuits, we found that much of the knowledge could be specified as Prolog assertions, and didn't require the ancestor search or all contrapositive forms. It then occured to us that there were classes of formulae which could be detected before search time, which could be expressed as Prolog assertions, and so gain the corresponding efficiency. Here we attempt to characterise such classes. It is intended that one would be able to add clauses to a collection of assertions and let the system automatically detect where Horn-clause-like (although the class is larger then just Horn clauses) implementation strategies can be used. Note that we need not restrict ourselve to clausal systems, as one of us has previously shown how to transform clausal theorem-provers into non-clausal theorem-provers [Poole84].

The goal here is to determine when the general procedure is unnecessary, so that the overhead of using the contrapositive formulae and searching back up the proof tree is, when possible, avoided. We do this by defining a syntactic classification of atomic symbols in a general clause which allows us to statically determine subgoals for which the use of contrapositives or the ancestor search is unnecessary. This class is larger than that of definite clauses. Of course a corollary is that if a subsystem is definite, then it can be implemented as Prolog. We thus pay for the extra cost for using true negation only when necessary and only in the cases where it will make a difference.

Notice that adding negation to Prolog is equivalent to adding disjunction, as $a \vee b$ is equivalent to $a \leftarrow \neg b$.

2. Syntax

Here we present a syntax that will incorporate normal clause form and Prolog's definite clauses as special cases. We first define an atomic symbol to be as in Prolog. A **literal** is an atom or the negation of an atom. The negation of a is written $\neg a$. A **clause** has the form

$$L_1 \vee \cdots \vee L_k \leftarrow L_{k+1} \wedge \cdots \wedge L_n ;$$

where L_i is a literal. If $k=1$ and all of the literals are atoms then this is a definite clause, as in Prolog. If $k=n$ then we have the normal definition of a clause [Chang73, Robinson79].

We assume the commutativity and associativity of conjunction and disjunction. Semantically, the order of the literals in the left and right parts of a clause is irrelevant. Note that in a particular proof procedure the order may affect the efficiency or the completeness, and so it may be appropriate to modify the search strategy for the particular problem.

As in Prolog we have the notion of a query. The query

$$?L_1 \wedge \cdots \wedge L_n ;$$

is defined to mean, add to the current set of clauses C the clause

$$? \leftarrow L_1 \wedge \cdots \wedge L_n ;$$

where "*?*" is a new atom, then ask the question, "is *?* a logical consequence of the given clauses."

3. Semantics

The interpretation of the set of clauses C is the normal model-theoretic semantics, viz.

P Q	*true* *true*	*false* *true*	*true* *false*	*false* *false*
$\neg P$	*false*	*true*	*false*	*true*
$P \wedge Q$	*true*	*false*	*false*	*false*
$P \vee Q$	*true*	*true*	*true*	*false*
$P \leftarrow Q$	*true*	*false*	*true*	*true*

An answer *yes* means that *?* is a logical consequence of the clause set C. That is, *?* is true in every model of C.

In summary, a clause is false in some interpretation if and only if all of the $L_1, \cdots, L_k$ are false, and $L_{k+1}, \cdots, L_n$ are all true in that interpretation. The following lemma trivially follows from the semantics:

Lemma 1. We can swap a literal from one side of the "$\leftarrow$" to the other if we negate it.

For any clause c, a **contrapositive** of c is the clause with exactly one literal on the left hand side of the "$\leftarrow$," that results from applying lemma 1 to c. Note that if there are n literals in a clause, there are n contrapositive forms of that clause.

The **normal form** of a clause c is an equivalent clause without any negation symbols (i.e., any literal with a not sign, is moved to the other side of the arrow). Note that any clause has a unique normal form (up to associativity and commutativity of conjunction and disjunction). For example the clause $a \vee b \leftarrow c \wedge \neg d$ has the contrapositive forms

$$a \leftarrow \neg b \wedge c \wedge \neg d$$
$$b \leftarrow \neg a \wedge c \wedge \neg d$$
$$\neg c \leftarrow \neg a \wedge \neg b \wedge \neg d$$
$$d \leftarrow \neg a \wedge \neg b \wedge c$$

and the normal form $a \vee b \vee d \leftarrow c$.

4. The proof procedure

The Prolog proof procedure is augmented to have accessible (1) the contrapositive of each clause in the clause set C, and (2) to search up the relevant proof tree branch for the negation of the current subgoal. The second modification corresponds to "reductio ad absurdum" or "proof by contradiction." (See Umrigar and Pitchumani [Umrigar85] for an example implementation in Prolog.)

We define the negative ancestor rule as "If g is a subgoal which unifies with the negation of an ancestor, then we can mark g proven."

The proof procedure becomes: a goal g is proven if (1) there is a contrapositive form of an input clause that unifies with g, such that all of the literals on the right hand side of the contrapositive form are proven, or (2) g unifies with an ancestor literal $\neg g$ such that all substitutions are consistent. We can express this procedure

```
% prove(G,A) is true if and only if Clauses ⊨A ⊃G
prove(G,A)←
        member(G,A);
prove(G,A)←
        clause(G,B)
        neg(G,GN)
        proveall(B,[GN|A]);

% proveall(L,A) is true iff Clauses ⊨A ⊃Lᵢ for each Lᵢ∈L
proveall([],A);
proveall([G|R],A)←
        prove(G,A)
        proveall(R,A);

% neg(X,Y) is true if X is the negative of Y, both in their simplest form
neg(not(X),X)←
        ne(X,not(Y));
neg(X,not(X))←
        ne(X,not(Y));

% clause(H,B) is true if there is the contrapositive form of an input clause
%       such that H is the head, and B is the body.
%       in particular, we know Clauses ⊨B ⊃H
```

Figure 1 Full clausal theorem prover in Prolog

in terms of a Prolog provable relation (cf. [Bowen82, Bowen85]) as in fig. 1. The following theorem holds for the proof procedure of fig. 1:

Theorem 1. The proof procedure is correct with respect to the above semantics, and is complete in the sense that if *?* is logically entailed by a consistent set of clauses, then there is a proof for *?*.

Proof. (1) Correctness. To prove the correctness, we need to show that each of the rules above is true with respect to the intended interpretation. The first clause of the definition of *prove* is correct as it says, "If $G \in A$ then $L \supset A$," which is trivially correct. The second clause for *prove* says "If $Clauses \models B \supset G$ and $Clauses \models A \cup \{\neg G\} \supset B$ then $Clauses \models A \supset G$", which is true by transitivity of implication, and by noticing that "if $Clauses \models A \cup \{\neg G\} \supset G$ then $Clauses \models A \supset G$."

(2) Completeness follows directly from the completeness of MESON proof procedure [Loveland78].

5. Prolog compatible subsystems

Now arises the question that, if we have a set C of general clauses, must we always have available all contrapositives and search up the tree, or can we statically determine conditions under which we do not need to do these things? If so, a portion of our processing can be as efficient as Prolog. In particular, we would like to pay the extra cost only in proportion to our use of the extra-Prolog features. Of course, if all of the clauses are indeed definite then we do not require the modifications. However, we would also like to see if there is some larger class of clauses for which we do not need to use the contrapositives and search up the tree; i.e., to determine, if possible, some non-trivial subset of the clause set C for which the modifications can be ignored.

To do so, first define a literal L to be **relevant** within a set of clauses if L is provable within the context of its ancestors. That is L can be proven under the assumption of $\neg M$ for ancestors M of L. In other words, we can generate a set A such that $prove(L,A)$ can be derived.

Now consider the cases where we actually need to search up the tree from a goal g to find an ancestor unifying with $\neg g$. All of the three following conditions below must hold, for the ancestor search to be useful:

1. $\neg g$ is relevant;
2. $\neg g$ is askable (i.e., can be generated as a subgoal) and
3. within a subproof of $\neg g$ we can generate a subgoal of g.

If one of the above conditions cannot occur, then we do not need to consider searching for a negated ancestor of g.

The second concern is with conditions under which we need to form the contrapositive of a rule so that L is on the left hand side. This is required when:

1. L is askable and
2. L is relevant

If one of the conditions 1 or 2 cannot be the case, then we do not need the contrapositive forms. Of course the first case is uninteresting as if L is never asked the contrapositive will never be used.

Now the idea is to describe a way to statically determine, for some set of literals, that the contrapositive or the searching for a negated ancestor is unnecessary.

5.1. Statically determined classes of predicates

We define weaker notions of relevant and askable which can be computed at compile time, namely **potentially relevant** and **potentially askable**. If some goal is neither potentially relevant nor potentially askable, we do not need contrapositive forms where that literal is on the left hand side. We need only do an ancesor search if both it and it's negation are potentially relevant and potentially askable.

Potentially askable is a relation on signed predicate symbols and sets of signed predicate symbols. If L is a literal, the signed predicate symbol **corresponding** to L has the same sign as L and the predicate symbol of the atom composing L. Intuitively, the signed predicate symbol of L is obtained by stripping the arguments from the atom in L. Potentially relevant is a property of signed predicate symbols.

Potentially askable is defined as:

(1) $?$, exported symbols or signed predicate symbols are potentially askable relative to $\{\}$ (see discussion of section 5 for explanation of exported symbols);

(2) if p is potentially askable relative to S, and L is a literal corresponding to p, and $L \leftarrow L_1 \wedge \cdots \wedge L_k$ is the contrapositive of a clause, then each p_i corresponding to L_i such that $p_i \notin S$ is potentially askable relative to $\{p\} \cup S$.

Potentially relevant is defined as:

(1) if p is potentially askable relative to S and $\neg p \in S$ then p is potentially relevant;

(2) if $L \leftarrow L_1 \wedge \cdots \wedge L_k$ is the contrapositive of a clause such that L corresponds to p, and if the p_i corresponding to the L_i are potentially relevant, then so is p.

Theorem: if L is relevant and askable, then its corresponding signed literal is potentially relevant and potentially askable.

Proof: assume L is relevant and askable. Consider the proof tree containing L. Transform this into a tree containing the signed predicated corresponding to the literals in the proof tree. Remove all paths in the tree which form cycles (i.e., if there is a p as an ancestor of p, then remove that branch and all associated subtrees). This tree then shows that p is potentially askable and potentially relevant.

Corollary: Only if a literal is potentially relevant and potentially askable do we need consider contrapositives of rules with it on the left hand side.

Corrolary: We need only search ancestors for negated literals if both the literal and its negation are both potentially askable and potentially relevant.

Proposition: Given a set of definite clauses, the determination of potentially relevant and potentialy askable is adequate to show that we need only consider the normal Prolog search tree (i.e. we need never consider other contrapositive forms, or need to do an ancestor search).

Proof: the negative of a predicate symbol is never potentially askable. This is because given a subgoal which is a negative atom, we always produce one more subgoal which is a negative atom. All ancestors are negative so the negative ancestor rule never works, and we never have one potentially provable. The procedure terminates as there are only a finite number of signed predicates to consider.

6. Implementation

The properties potentially relevant and potentially provable are decidable and can be computed at input time. We need only consider is what elements are potentially askable relative to {}. There are three possibilities, depending on how much we know about potential queries:

1. If we know what forms a query may take, then we only need make the corresponding signed predictes potentially askable relative to {}, and deriving all other potentially askable things from this.
2. If we have a module system with restricted exports, we can make the syntactic restrictions for each module and make the classes local to each module. By restricting the exports from each module, we can then make the elements of the export list initially askable relative to {}. As all calls inside the module are assumed to be local, we need not worry about negative calls outside the module.
3. The other possibility is to consider each possible call that can be given the system. That is make every signed predicate symbol potentially askable relative to {}. This is not as bad as it may seem, as if some signed predicate has been considered askable relative to anything then we have no need to reconsider it as askable relative to the empty set. We can also stop attemtps to prove potential askability if we have found a subgoal which has already been proven to be potentially relevant.

7. Compiling Into Prolog

The way we have used this is to compile the clauses into Horn clauses. This is done as follows:

If some atom, L, and its negation are both potentially askable and potentially relevant, then we replace it with $pr(L,A)$ and its negation with $pr(n(L),A)$, and make the first rule in the definition of L and $n(L)$,

$pr(L,A) \leftarrow member(n(L),A);$
$pr(n(L),A) \leftarrow member(L,A);$

We then create contrapositives of all rules such that only potentially askable and provable predicates appear on the left hand side of the rule. Those rules with a *pr* on the left hand side and the right hand side must add the head element to the ancestor list on the right hand side. For example the clause $h \leftarrow t$, where h, $\neg h$, t and $\neg t$ are potentially askable and provable becomes

$pr(h,A) \leftarrow pr(t,[h|A]);$
$pr(n(t),A) \leftarrow pr(n(h),[n(t)|A]);$

8. Example

Consider the following set of clauses (with all arguments removed); note there is a recursive call between c, $\neg e$ and g.

$a \leftarrow b \wedge c$;
$a \vee b \leftarrow d$;
$c \vee e \leftarrow f$;
$\neg g \leftarrow e$;
$g \leftarrow c$;
g;
$f \leftarrow h$;
h;
d;

The following are both potentially askable and potentially relevant: a, $\neg a$, b, $\neg b$, c, d, $\neg e$, f, g, h. Therefore we need only check the contrapositive and ancestors of a and b. Note that all but the first two clauses can be implemented as Prolog clauses, even though they are not.

9. Conclusion

We have shown a general method for adding true negation and disjunction to a Horn Clause language. We have given a method for statically determining cases for which we do not need to use the extra machinery and can compute answers with the same machinery as for Horn Clause logic.

This technique is currently being implemented in the Theorist system [Poole86], which requires negation to prove that theories are inconsistent.

Acknowledgements

This research was supported by National Sciences and Engineering Research Council of Canada grant A0894.

References

[Bowen82] K. Bowen and R.A. Kowalski (1982), Amalgamating language and metalanguage in logic programming, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.-A. Tarnlund (eds.), Academic Press, New York, 153-172.

[Bowen85] K.A Bowen and T. Weinberg (1985), A meta-level extension of Prolog, *IEEE 1985 Symposium on Logic Programming*, July 15-18, Boston, Massachusetts, 48-53.

[Chang73] C.L. Chang and R.C.T. Lee (1973), *Symbolic Logic and Mechanical*

Theorem Proving, Academic Press, New York.

[Genesereth85] M.R. Genesereth (1985), The use of design descriptions in automated diagnosis, *Qualitative Reasoning about Physical Systems*, D.G. Bobrow (eds.), MIT Press, Cambridge, Massachusetts, 411-436.

[Loveland78] D.W. Loveland (1978), *Automated theorem proving: a logical basis*, North-Holland, Amsterdam, The Netherlands.

[Poole84] D. Poole (1984), Making "clausal" theorem provers "non-clausal", *Proceedings of the Fifth Biennal Conference of the Canadian Society for the Computational Studies of Intelligence*, May 15-17, University of Western Ontario, London, Ontario, 124-126.

[Poole86] D.L. Poole, R.G. Goebel, and R. Aleliunas (1986), Theorist: a logical reasoning system for defaults and diagnosis, *Knowledge Representation*, N.J. Cercone and G. McCalla (eds.), Springer-Verlag, New York [invited chapter, submitted September 10, 1985].

[Robinson65] J.A. Robinson (1965), A machine-oriented logic based on the resolution principle, *ACM Journal* **12**(1), 23-41.

[Robinson79] J.A. Robinson (1979), *Logic: Form and Function*, Artificial Intelligence Series 6, Elsevier North Holland, New York.

[Umrigar85] Z.D. Umrigar and V. Pitchumani (1985), An experiment in programming with full first-order logic, *IEEE 1985 Symposium on Logic Programming*, July 15-18, Boston, Massachusetts, 40-47.