1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

https://hdl.handle.net/10945/36984

A more general logic programming

Prolog is a language for *logic programming*. That is, it's a programming language that looks and works somewhat like formal logic. To be efficient and easy to use, Prolog omits some important features of formal logic. We'll discuss now these omissions, and present a more general and powerful--but slower--approach to logic programming. This approach is called *resolution*, and it emphasizes declarative (instead of procedural) meanings of logical formulas.

Logical limitations of Prolog

Prolog can do many things. But it has four fundamental logical weaknesses:

1. Prolog doesn't allow "or"d (disjunctive) facts or conclusions--that is, statements that one of several things is true, but you don't know which. For instance, if a light does not come on when we turn on its switch, we can conclude that either the bulb is burned out or the power is off or the light is disconnected.

2. Prolog doesn't allow "not" (negative) facts or conclusions--that is, direct statements that something is false. For instance, if a light does not come on when we turn on its switch, but another light in the same room comes on when we turn on *its* switch, we can conclude that it is false that there is a power failure.

3. Prolog doesn't allow most facts or conclusions having existential quantification--that is, statements that there exists some value of a variable, though we don't know what, such that a predicate expression containing it is true. (Prolog does have a limited form of existential quantification for local variables in rules, as discussed in Section 4.1.) For instance, if we know that something is wrong with an appliance, then there exists a component X of the appliance such that X has a fault in it.

4. Prolog doesn't directly allow *second-order logic*, predicate names as variables--that is, statements about P where P stands for any predicate name. We can get something close with the trick of the **inherits** predicate in the car frame hierarchy of Section 12.11, the idea of rewriting facts to include extra predicate-name arguments. You can also approach second-order logic using the built-in **clause** predicate discussed in Section 7.13. So this weakness of Prolog is less serious than the others, and we won't say anything further about it in this chapter.

Notice that these are logical issues, not efficiency issues. Chapter 6 discussed how Prolog isn't an efficient control structure for reasoning about some problems. But these four points are deeper and more serious weaknesses: they represent things Prolog can't do at all even working slowly.

The logical (declarative) meaning of Prolog rules and facts

To better understand Prolog's limitations and how to get around them, let's examine more carefully what Prolog rules and facts mean.

Chapter 4 explained how rules and facts all go into a Prolog database of true statements. Alternatively, a Prolog database can be seen as a single logical statement representing the conjunction ("and"ing) of the statements of each rule and fact. We'll now show how the declarative or logical meaning (not the same as the procedural meaning) of any Prolog database can be expressed entirely as a set of statements using only "or"s and "not"s.

To do this we must first remove the "**:-**" symbol from rules, because that is not a logic symbol. In Section 4.1 we said to think of it as a backward arrow or backward implication. In logic, an implication (or, strictly speaking, a *material* implication) is equivalent to an "or" (a disjunction, symbolized in Prolog by "**;**") of the pointed-to side of the arrow with the negation of the other side of the arrow. So the rule

```
a :- b.
```

is equivalent in logical (declarative) meaning to

```
a; not(b).
```

which reads as "a or not b". Figure 14-1 shows the truth table for the two forms, to better convince you.

An immediate generalization comes from taking predicate **b** to be an "and" (conjunction) itself. For instance

```
a :-  c, d, e.
```

is equivalent to

```
a; not(c,d,e).
```

which by DeMorgan's Law is the same as:

```
a; not(c); not(d); not(e).
```

So we can express the logical equivalent of any Prolog rule by "or"ing the left side with the **not**s of each expression that is "and"ed on the original right side. This is called the *clause form* of the rule.

But logical equivalence is not complete equivalence of meaning because it only covers the declarative meaning of rules. Prolog rules also have the "procedural" interpretation of "If you believe that these things hold, then believe that this thing holds". So rules involve a causation, a moving from a cause (a belief in several things) to an effect (a belief in something else); and the effect comes after the cause in time. Causations imply a direction of reasoning, while an "or" doesn't necessarily. For instance,

```
a :- b.
```

models a causation from **b** to **a**. But its logical equivalent in clause form

```
a; not(b).
```

can also be used from **a** to **b**: if we are told **a** is false, then **b** must be false too so the "or" will be true. That's the *contrapositive* direction of reasoning, and Prolog rules can't be used that way even though it's logically valid | REFERENCE 1|. .FS | REFERENCE 1| Don't confuse the backward reasoning of the contrapositive with backward chaining. Backward chaining reasons about the left side of some rule, whereas contrapositive reasoning reasons about the *opposite* of the left side. .FE

But clause form has advantages too. The second limitation on reasoning of Prolog interpreters mentioned in Section 14.1 concerned **not**s: they mean "impossible to prove", not "proved false". But when we write things in clause form we can interpret **not**s more precisely, to reach new conclusions not otherwise possible. For instance, in the previous contrapositive reasoning example, when **a** is false, **b** is proved false, not just proved to be impossible to succeed. So clause form can provide "true nots". But there is an associated danger: when the Prolog rule itself has a **not**, we must make sure that we can interpret it as provably false in the real world. Otherwise, the clause form only covers part of its meaning.

Extending Prolog rules

Using the preceding clause-form conversion for Prolog rules lets us give meaning to new kinds of rules, rules not legal in Prolog. For instance this "pseudo-Prolog"

```
(a; b) :- c.
```

which means that either of **a** or **b** is true whenever **c** is true, becomes in clause form

```
a; b; not(c).
```

And this pseudo-Prolog

```
not(a) :- b.
```

which means **a** is false whenever **b** is true, becomes

```
not(a); not(b).
```

Notice that the first clause-form formula has two unnegated expressions, and the second has no unnegated expressions. In general, any Prolog rule without **not**s becomes a clause form having one and only one unnegated expression, what's called a *Horn clause*.

Clause form for a rule can require more than one "or"ed formula. As a more complicated example, consider this pseudo-Prolog

```
(a; (b, c)) :- d, not(e).
```

which has the logical equivalent

```
a; (b, c); not(d); e.
```

To get rid of the "and", we can use the distributive law for "and" over "or". This gives two separate statements (*clauses*), each of which must be true:

```
a; b; not(d); e.
a; c; not(d); e.
```

And that's the clause form for the original rule.

Rewriting rules in clause form answers some puzzling questions of why rules sometimes seem "and"ed together and other times "or"ed together. Suppose we have two rules

```
a :- b.
```

```
a :- c.
```

The logical equivalent form is

```
(a; not(b)), (a; not(c)).
```

or:

```
a; (not(b), not(c)).
```

using the distributive law of "and" over "or". This can be rewritten as a single rule

```
a :- (b;c).
```

using DeMorgan's Law. So an "and" in the one sense--the "and" of the logical truth of separate rules--is an "or" in another--the "or" of the right sides of rules with the same left side.

More about clause form

So a clause is an "or" of a bunch of things, each of which is either a single predicate expression or the **not** of one. As usual, expressions can have arguments. For instance, this "santa clause":

```
santa(joe); santa(tom); not(santa(bill)).
```

Any statement we can express in first-order logic has a logical equivalent in a set of clauses. Why is this important? Because there's a simple yet powerful inference method that can be applied to clauses, *resolution*. It can be proved that *any* inference that logically follows from a set of statements can be found by using resolution on the clause forms of those statements. So resolution is more powerful than Prolog-style backward chaining.

But to use resolution, everything must be in clause form. We've already illustrated a procedure for translating "pseudo-Prolog" with "or"s, "not"s, and arbitrarily complex formulas on the left sides of rules: just rewrite in *disjunctive normal form* as an "or" of "and"s, using the laws of logic. That covers the first two limitations of Prolog cited in Section 14.1. But what about existential quantifiers? They can get complicated.

The simplest case for existential quantifiers is when we want to assert there exists some variable value such that a predicate expression mentioning that variable is true. Then we can substitute a constant for the variable, provided that constant can't be confused with an actual value of the variable, like a nonsense word. For instance, if we want to say that there exists an X such that **p(X)** is true, then we can could assert fact **p(zzxxy)** provided that **zzxxy** is not a symbol for any of the actual values for X. We can then use this fact **p(zzxxy)** in chains of reasoning, just remembering that this value doesn't really mean anything.

But now suppose we want to say that for every Y there exists X such that **p(X,Y)** holds. Now we can't just substitute a constant for X because X may depend on Y: that is, X is a function of Y. This function is a *Skolem function*. We need one whenever we are trying to represent in clause form a statement containing both existential and universal quantification. Standard techniques exist for situations needing Skolem functions, but they are too complicated to discuss here. See books on "theorem proving" if you're interested.

Resolution

Resolution is an inference technique that takes two clauses as input, and produces a clause as output. The output clause, the *resolvent*, represents a true statement consistent with the input clauses, the result of *resolving* them. In other words, the resolvent is one conclusion we can draw. If the resolvent is a fact, then we've proved a fact. If the resolvent is the clause consisting of no expressions, the *null clause*, we've proved a contradiction. Resolution is particularly efficient for proof by contradiction: we assume the opposite of some statement we wish to prove, and see if we can prove the null clause from it.

Resolution requires pairs of opposites in the two input clauses. That is, one input clause must contain a predicate expression--call it P--for which **not(Q)** occurs in the other input clause and where P can match Q by binding variables as necessary. (Formally, P matches Q if the expression **P=Q** can succeed.) Then the resolvent of the two input clauses is the "or" of everything besides P and **not(Q)** in the two clauses, eliminating any duplicate expressions. We say that the P and the **not(Q)** "cancel". For instance, if the input clauses are

```
a; b; not(c); d.
e; not(b); a; f.
```

then the resolvent (output) clause is

```
a; not(c); d; e; f.
```

where we eliminated the opposites **b** and **not(b)** and a duplicate **a** fact.

Inference by resolution becomes most useful when we do several resolutions in succession. Here's an example. Let's use Horn clauses (clauses with one and only one unnegated expression), because we already know how Prolog interpreters handle them, and let's avoid variables. Suppose we have these rules and facts:

```
a :- b, c.
c :- d.
b.
d.
```

Rewriting in clause form, we get these Horn clauses:

```
a; not(b); not(c).
c; not(d).
b.
d.
```

(Prolog facts are identical in clause form.) Now suppose we want to prove **a** (see Figure 14-2):

1. To do proof by contradiction, we add **not(a)** to the other four clauses in our database of true statements.

2. Resolving **not(a)** with the first of the four original statements, the **a** and **not(a)** expressions cancel, and the resolvent is **not(b); not(c)**. We can add that new clause to the others.

3. Resolving this new clause with the third of the original clauses, the **b** and **not(b)** cancel, and the resolvent is just **not(c)**.

4. Resolving this in turn with a second of the original four clauses, we cancel the **c** and **not(c)**, giving as resolvent **not(d)**.

5. Finally we resolve this with the last of the original four clauses, and **d** and **not(d)** cancel, leaving us with a null clause.

6. Therefore we can prove anything if we assume that **a** is false. So **a** must be true. (This assumes that the original set of rules and facts was not self-contradictory, something we could verify by doing all possible resolutions among them.)

There is a one-to-one correspondence of the steps in the previous *resolution proof* and the steps that Prolog interpreters follow in backward chaining. To prove **a**, they would:

1. Take **a** as the goal (query).

2. Find a rule for **a**: the first rule. This says to prove **b**, then **c**.

3. But **b** is a fact (the first fact).

4. To prove **c**, use the second rule, which says to prove **d**.

5. But **d** is a fact (the second fact).

6. Therefore **c** is true and **a** is true.

In general, resolution can do anything backward chaining can do, but not the other way around. Resolution is a more general and flexible form of inference, because it can resolve clauses in many different orders; backward chaining is more rigid. And every new resolvent clause can be used many ways in new resolutions, so resolution possibilities keep increasing--it's a *monotonic* search process.

Resolution with variables

When predicates have variable arguments, resolution becomes a little more complicated: we still look for a pair of opposites, but Prolog-style binding of the variables can be done to make the canceling expressions "match". As with Prolog, bindings made to variables apply to any other occurrences of the variables within their original clauses, so if a **p(X)** in the first input clause matches a **p(4)** in the second input clause, any other **X** in the first clause becomes a 4. Variables can also be bound to other variables. Important note: it's essential that each input clause have different variable names before resolving.

Here's an example of resolution with variables. Suppose the two clauses are

```
a(3); b(Y); not(c(Z,Y)).
not(a(W)); b(dog); c(W,cat).
```

The **a** expressions can cancel with **W** bound to 3, giving:

```
b(Y); not(c(Z,Y)); b(dog); c(3,cat).
```

The **b(dog)** is redundant with **b(Y)**, so we can improve this clause to:

```
b(Y); not(c(Z,Y)); c(3,cat).
```

But we could resolve the original two clauses another way. The **c** expressions could cancel, with **Z** being bound to **W** and with **Y** being bound to **cat**, giving:

```
a(3); b(cat); not(a(W)); b(dog).
```

This is a completely different resolvent, representing a different conclusion possible from the two clauses. Notice that we can't eliminate anything here; **b(cat)** and **b(dog)** aren't redundant, nor are **a(3)** and **not(a(W))**.

Note that bindings are transitive: if **A** is bound to 9, and **B** is bound to **A**, then **B** is bound to 9 too. So several reasoning steps may be necessary to determine a variable binding.

Three important applications of resolution

Resolution is a powerful inference technique that can supplant other inference techniques. Three important special cases of resolution are summarized in Figure 14-3: backward chaining, forward chaining, and rule collapsing. To use resolution for backward chaining as in Section 14.5, one starting clause is always the negation (opposite) of something to be proved. To use resolution for forward chaining, one input clause for every resolution is always a fact. Rule collapsing is a way to make rules more efficient, not a control structure itself: it takes two rules, one whose left side is on the other's right side, and combines them into a single new rule; this is equivalent to resolving the rules' clauses. Procedure collapsing is important to compilers for programming languages.

Resolution search strategies

Reasoning by resolution means performing a series of resolution operations. This often means more things to choose from than conflict resolution in rule-based systems, because you must pick pairs of clauses to resolve, and typically there are many. Facts are clauses too, as are the resolvents (results) from past resolutions. So resolution-based reasoning is a kind of search with a high branching factor. This means a breadth-first resolution strategy (resolving every pair of original clauses, then every new clause with either an original clause or a new clause, and so on) is very slow. In principle, a breadth-first resolution control structure can prove anything that is a logical consequence of particular facts and rules; but that's no good if it takes a hundred years to do so.

Often we know what we want to prove. Then we can use a *set-of-support* strategy, which we used without saying so in the example of Section 14.5. The idea is to assume the opposite of what we want to prove, and resolve it repeatedly with other clauses until we reach the null clause. If no possible resolutions remain at some point (when no other clause has an opposite that can "cancel"), back up to the last previous resulting clause for which there were alternative resolutions, and take an alternative. This is basically a depth-first resolution strategy starting from the negation of the proof objective.

If we don't have any one thing in particular we want to prove, but we prefer to prove facts, then a *unit-preference* resolution strategy may be good. The idea is to do first all resolutions involving facts, both positive and negative. If there aren't any facts, then perhaps prefer resolutions involving two-argument clauses, and so on. This strategy tends to keep resolvent clauses short, which often means we discover new facts fast.

If our clauses all represent rules, we may want to do the rule *collapsing* mentioned in the last section. A breadth-first resolution strategy could work for this, since there's no designated objective or facts to work

from. For efficiency, we might try instead best-first search with the evaluation function the total number of expressions in the input clauses, which tends to discourage less useful resolutions. In any event, we should arbitrarily stop the search at some point, since there can be enormous numbers of possible resolutions, and we don't want to try them all.

Domain-dependent heuristics can help considerably when reasoning with resolution. Heuristics can enhance any of the search strategies mentioned.

Implementing resolution without variables (*)

Resolution without variables is simple to implement in Prolog. Represent the clauses as list arguments to **clause** facts, so for instance

```
a; not(b); not(c); d.
```

is represented as

```
clause([a,not(b),not(c),d]).
```

Notice lists require commas, so the comma here actually means "or". Then query the predicate **go**. Everything new proved will be printed out. Here's the program:

```
/* Resolution without variables */
go :- resolution(C1,C2,Cnew), !, write(Cnew), nl,
  not(Cnew=[]), go.


resolution(C1,C2,Cnew) :- clause(C1), clause(C2), not(C1=C2),
  matched_items(C1,C2,C1item,C2item), delete(C1item,C1,C1d),
  delete(C2item,C2,C2d), union(C1d,C2d,Cnew), not(clause(Cnew)),
  not(tautology(Cnew)), not(some_superset(Cnew)),
  asserta(clause(Cnew)).


matched_items(C1,C2,C1item,not(C1item)) :- member(C1item,C1),
  member(not(C1item),C2).
matched_items(C1,C2,not(C2item),C2item) :-
  member(not(C2item),C1), member(C2item,C2).


some_superset(C) :- clause(C2), subset(C2,C).


tautology(C) :- member(X,C), member(not(X),C).
```

Notice that we check new clauses to make sure they're neither previously found, nor tautologies (always-true statements), nor immediately derivable from other clauses by removal of items; only then do we assert a new clause. The assertion uses **asserta**, so a depth-first strategy much like set-of-support will be used.

This requires the **member**, **delete**, and **subset** predicate definitions from Sections 5.5, 5.6, and 5.7 respectively, plus the **union** predicate from Section 11.4 (which is closely related to the **append** of Section 5.6). Using **union** instead of **append** prevents repeated expressions in the resolvent.

```
/* Utility functions for resolution */
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


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


delete(X,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).


subset([],L).
subset([X|L1],L2) :- member(X,L2), subset(L1,L2).
```

For a demonstration, suppose we have these clauses:

```
clause([a,not(b),not(c)]).
clause([b]).
clause([d,not(a)]).
clause([c,e,f]).
```

Here's what happens when we run the program:

```
?- go.
[a,not(c)]
[not(c),d]
[d,e,f]
[a,e,f]


no
```

Unfortunately, implementing resolution with variables in Prolog is very tricky. The problem is in handling redundancies, the **tautology** and **subset** predicates of the program, for which Prolog's normal variable-binding must be subverted. Such full resolution inferencers can be written more easily in more flexible programming languages like Lisp.

.SH Keywords:

logic programming
disjunctive fact
negative fact
existential quantifier
second-order logic
clause form
Horn clause
Skolem function
resolution
breadth-first resolution strategy
set-of-support resolution strategy
unit-preference resolution strategy

Exercises

14-1. (A) Resolve all possible ways and list bindings:

```
state(3,A,X); possible(X,X,A).
not(possible(5,Y,Z)); state(3,6,Y).
```

14-2. (R,A) Suppose you are working for a research organization. Suppose you can get travel money if your department chairman approves and your sponsor approves. Alternatively, you can get travel money if your department chairman approves, the boss over him or her approves, and there are discretionary department funds available.

(a) Represent the preceding as two Prolog rules. (Hint: use one-letter predicate names because you'll have to write them many times in this problem.)

(b) Represent the two rules in clause form.

(c) Suppose that these are the only two ways that you can get travel money. Therefore if you do get travel money, certain things must have been true. Write this implication in "pseudo-Prolog" as a rule with the symbol "**:-**", but with "and"s, "or"s, and "not"s on its left side. Your rule must cover *all* the implications of having travel money.

(d) Convert this pseudo-Prolog rule to clause form. (Hint: clause form here is three clauses.)

(e) Suppose you can get travel money. What new clauses logically follow from this? (Make sure there are no unnecessary extra expressions in the clauses.)

(f) Suppose you can't get travel money. What new clauses logically follow from this? (Make sure there are no unnecessary extra expressions in the clauses.)

14-3. (a) Represent the following in clause form:

> C1: Block A is on the table.
> C2: Block B is on block A.
> C3: Block C is on block A.
> C4: Block D is on block C.
> C5: Block A is blue.
> C6: Block B is blue.
> C7: Block C is red.
> C8: Block D is green.
> C9: A block is above another block X if it is on that block or else if it is on a block which is above X.

(b) Prove by resolution that block D is above a blue block. Give numbers of statements you resolve, and label your resolvents.

14-4. Suppose we're told to resolve the clauses

```
a; b.
not(a); not(b).
```

Can we simultaneously cancel out both matched pairs, getting the null clause as resolvent? Why or why not?

14-5. (R,A) By the definition of resolution given in this chapter, if we resolve the clauses

```
a; b.
not(a); c.
```

we get

```
b; c.
```

But this doesn't seem to make sense if we show what's happening in a Venn diagram (See Figure 14-4). Here the region marked with lines running from southwest to northeast represents the first clause, and the region marked with lines running southeast to northwest represents the second clause. Any pair of clauses that are each individually true can be considered to be "and"ed together. But the region that has both markings (the cross-hatched region) does not correspond to the preceding resolvent clause. What's wrong?

14-6. (E) Suppose we have these Prolog rules:

```
a :- b, c.
a :- not(b), d.
```

Suppose that **not** can also be interpreted as a "real" not, one that insists that negative evidence is present.

(a) Write the two rules in clause form.

(b) Now resolve the two clauses from part (a), and write a new Prolog rule without **not**s equivalent to the resolvent clause.

(c) Explain how the idea of the preceding could be generalized to a useful trick to improve rule-based expert systems, one that applies to rules of any length. Explain (i) how you pick appropriate (good) pairs of rules, (ii) how you can avoid converting to clause form, and (iii) under what circumstances you can delete the original rules.

14-7. Prolog interpreters are Horn-clause theorem-proving systems. But consider the following fallacy.

(a) Represent in clause form this pseudo-Prolog:

```
(weekend(T); holiday(T)) :- empty(spanagel421,T).
```

which means that if Spanagel 421 is empty, it must either be a weekend or a holiday.

(b) Convert this into an equivalent Prolog rule with only one expression, **holiday(T)**, on its left side.

(c) Show that the original rule and the rule for part (b) have equivalent truth value by showing their truth tables.

(d) Part (b) is a legal Prolog rule because it has only one predicate expression on its left side. So it seems we can always implement a non-Horn clause in Prolog. What's wrong with this argument?

14-8. (E) Suppose we wish Prolog interpreters to use mathematical induction proof techniques automatically. Discuss what is wrong with just adding an induction rule to all Prolog programs, saying in essence

```
"p(X) is true if p(1) is true and if for N>1, p(N) implies p(N+1)."
```

14-9. (A) Consider proving things by resolution as a search problem with one operator: resolve two clauses to get a new clause. Each state can be described as a set of clauses given or proved. Suppose you have N clauses to begin.

(a) What is an upper bound on the initial branching factor?

(b) What is an upper bound on the branching factor after K resolutions have been done?

14-10. (P) The resolution program can be made more efficient.

(a) Improve it so it doesn't examine every pair of clauses, but only pairs in which the first clause precedes the second in the database.

(b) Improve it to remove any clause that *becomes* redundant, in that the expressions of the last clause discovered are a subset of the expressions of that clause.

Go to book index