Prolog & the Resolution Method

Gunnar Gotshalks
1999 March

Contents

The relation of Prolog to logic

Notes on the resolution method

Background

Based on resoution proof method developed by Robinson in 1966

Tries to find a contradiction in a set of logical formulae

Complete system of proof with only one rule. If something can be proved resolution can do it. That is a contradication will be found if the set of logical formulae contain a contradiction.

Correct, only theorems will be proven correct, nothing else.

Propositional case

Have a collection of clauses in cunjunctive normal form

- each clause is a set of propositions connected with OR
- propostions can be negated.
- set of clauses is implicitly ANDed together

Example 1

```
A or B
C or D or ~E
F
```

Example 2

What happens if we have a contradiction in the database?

- Database is `P`
- Add `~P` to the database
- `P & ~P` is a contradiction
- Equivalent of `P & ~P => null` – cancelling each other and null (**empty clause**) which is false.

Resolution rule

Given the clause "`Q or ~R`" and the clause "`R or P`", then resolving the two clauses is the following.

```
(Q or ~R) and (R or P) => (P or Q)
```

In the general case Q and P can represent abitrary sets of disjunts

$Q = A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_q$

$P = B_1^1 \text{ or } B_2^2 \text{ or } \ldots \text{ or } B_p^q$

Then resolving the two clauses gives the following.

$(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_q \text{ or } \sim R) \text{ and } (R \text{ or } B_1 \text{ or } B_2 \text{ or } \ldots \text{ or } B_p)$

$\Rightarrow (A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_q) \text{ or } (B_1 \text{ or } B_2 \text{ or } \ldots \text{ or } B_p)$

$= A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_q \text{ or } B_1 \text{ or } B_2 \text{ or } \ldots \text{ or } B_p$

Resolution normally means clauses get longer. The difficulty in using resolution is to avoid exponential explosion in the number of clauses and and explosion in their length.

Example 3

Given the following set of cluases

1. `P`
2. `~P or Q`
3. `~Q or ~R`

4. R

Looking for a contradiction, means trying to generate the empty clause

5. Resolving 1 (P) and 2 (~P or Q) yields Q
6. Resolving 4 (Q) and 3 (~Q or ~R) yields ~R
7. Resolving 5 (~R) and 4 (R) yields the empty clause

Notation for if A then B

If A then B translated to conjunctive normal form is ~A or B.

In Prolog it is written as B |- A

A longer example is if A and B and C then D or E or F which becomes
~A or ~B or ~C or D or E or F. There is no equivalent in Prolog (see Horn clauses later in the document).

A set of clauses can be written as a collection if...then statements

If (the moon is made of green cheese) then (pigs fly) =>
~(the moon is made of green cheese) or (pigs fly).

A fact is just itself (the moon is made of green cheese) which comes from
if true then (the moon is made of green cheese) => ~true or (the moon is made of green cheese) => false or (the moon is make of green cheese) => (the moon is made of green cheese). In Prolog this would look like (the moon is made of green cheese) |- true. But since the "|- true" is redundant it is not used.

A query

A query is the equivalent of if A and B and C then false (in Prolog false if A and B and C)
~A or ~B or ~C or false => ~A or ~B or ~C
Again the true part is redundant and is not used.

Example: false if (pigs fly) ==> ~(pigs fly) or false => ~(pigs fly)

Prolog distinquishes between facts and queries depending upon the mode in which it is being used. In "consult" mode we are entering facts. In interactive mode we are entering queries.

Predicate Calculus case

Step up to predicate calculus as resolution is not interesting at the propositional level

First we can simplify our notation.

The universal quantifier, ∀X – forall X – used in expressions such as ∀X:P(X) – forall X it is the case that P(X) is true – can be written more simply by just having the variable – the for all is assumed. Thus we simply write P(X). For example person(X) implicitly means ∀X:person(X).

The existential quantifier ∃X – there exists an X – used in expressions such as ∃X:(PX) – there exists an X such that P(X) is true creates a constant P(a) – person (alice). The constant alice represents a value which makes the predicate preson true. If we do not know the value of the specific constant we give it a name, say a, where the name represents the value of the constant.

What happens if we nest quantifiers.

∃x ∃y : P(x,y) becomes P(a,b). Introduce two constants.

∃x ∀y : P(x,y) becomes P(a,y). One constant for all y values.

∀x ∃y : P(x,y) becomes P(x,f(x)). Could have a different constant for each value of x, so we have a function of x as the "constant" in the simplified notation.

∀x ∀y : P(x,y) becomes P(x,y). We have two independent variables.

$\forall x \; \forall y \; \exists z : P(z)$ becomes $P(g(x,y))$. The constant z depends upon the variables x and y, so we have a function of x and y to represent the constant.

Removing quantifiers as done in the above is called **Skolemization**. Removal of the existential quantifier ($\exists$) gives use constants. Removal of the universal quantifier ($\forall$) gives us variables and removal of nested quantifiers gives us functions – in Prolog called structures or compound terms). Each predicate is called a **literal**.

The transitive closure of the constants and the functions of constants is called the **Herbrand universe**.

When we have variables and constants then we use pattern matching to **unify** literals so they can cancel. **Unification** is the process of doing a pattern match on two literals.

Example 5

Given the following two clauses in a database.

```
person(bob)
~person(X) or mortal(X)   ;; for all X: if person(X) then mortal(X)
```

Lets make a query asking if bob is a person. The query adds the following clause to the database – bob is not a person.

```
~person(bob)
```

Resolution with `person(bob)` gives the empty clause. We have a contradiction, so bob is a person.

What if we ask if bob is mortal? This adds the clause – bob is not mortal to the database.

```
~mortal(bob)
```

Now there is no direct match but we can match `mortal(X)` with `~mortal(bob)` if we let X=bob. So resolving the query with the second clause in the database gives us the following.

```
~person(bob)
```

which if resolved with the first clause gives us the desired contradiction and so we know that bob is mortal.

On the same database let us ask if alice is a person by adding the following clause to the database.

```
~person(alice)
```

Now resoution cannot find a contradiction and eventually all possibilities are exhausted. Without a contradication we assume `~person(alice)` must be true and so alice is not a person. We have a **closed universe.** We only know what is in set of clauses in the database.

General resolution permits unifying several literals at once and includes **factoring,** unifying two literals in a clause. For example, if we have the following clause.

```
loves(X,bob) or loves(alice, Y)
```

We can factor the clause and produce the following clause by pattern matching the literals within the clause with the assignment X=alice and Y=bob. The factored clause is implied by the unfactored clause because, effectively, the factored clause represents a subset of the true cases in the unfactored clause.

```
loves(alice,bob)
```

Creating a database means converting general predicat calculus statements into conjunctive normal form. A large part of Chapter 10 describes how this can be done.

Horn clauses are ones where the consequent is a single literal. For example X is the consequent in "if A and B and C then X". Horn clauses are important because resolution, while it is complete, usually leds to clauses getting longer and longer while contradiction means getting the empty clause – need to shorten clauses.

Horn clauses have the property that every clause has at most one **positive** literal (unnegated) and zero or more **negative literals** (negated). Since facts are positive literals resolving with facts reduces the length of clauses, thus helping to get to an empty clause.

There is much research into getting resolution to be more efficient because it is so ameanable to automation.