Lecture 10

Logic Programming

Contents

- Clausal form
- Resolution
- Horn clauses
- Prolog as Horn clauses
- Prolog search as refutation proof

(See Clocksin & Mellish, Chapter 10)

Logic and Prolog

- The meaning of a Prolog program can be explained very naturally using First Order Predicate Logic (FOPL).
- This gives a theoretical basis for Prolog and related computational systems.
- There are some simplifications and assumptions needed.
- Many practical Prolog facilities are not expressible in FOPL.

Some terminology

A *literal* is either a predicate applied to arguments, e.g.:

$p(a, b, c, d)$
$mother(fred, rita)$

or the negation of a predicate applied to arguments, e.g.

$\rightharpoonup p(a, b, c, d)$
$\rightharpoonup mother(fred, rita)$

A *conjunction* is a sequence of expressions linked by " $\wedge$ " (for "and").

$p(a, b, c, d) \ \wedge \ q(c, d) \ \wedge \ p(a, f, c, d)$
$mother(fred, rita) \ \wedge \ father(fred, bill)$

A *disjunction* is a sequence of expressions linked by " $\vee$ " (for "or").

$p(a, b, c, d) \vee \ \rightharpoonup q(c, d) \ \vee \ p(a, f, c, d)$
$mother(fred, rita) \vee \ \rightharpoonup father(fred, bill)$

A *clause* is a disjunction of literals.

Functions and Skolemisation

A *function term* is an argument to a predicate in the form of a functor and arguments; e.g.

$likes(mother(fred), father(bill))$
$greater(succ(succ(x)), succ(x))$

In rearranging FOPL formula, it is possible to re-express existential quantifiers ($\exists$) with constants or function terms ("Skolemisation" – details omitted.)

$\forall Y \exists X : likes(Y, X)$

becomes:

$likes(Y, skolem1(Y))$

(Names of the Skolem functions are chosen to be different in different expressions.)

Clausal form

Any expression in FOPL can be rearranged into *clausal form*: a conjunction of disjunctions of literals, with all variables treated as *universally* quantified, and any existential variables represented as Skolem terms.

$$\forall M(man(M) \supset (\exists W(woman(W) \ \wedge \ likes(M, W))))$$

becomes:

$$(\rightharpoonup man(M) \ \vee \ woman(skolem2(M))) \wedge$$
$$(\rightharpoonup man(M) \ \vee \ likes(M, skolem2(M)))$$

So any formula can be seen as a collection of clauses, viewed as being conjoined, each clause being a disjunction of literals.

Hence, any *set* of formulae can be thus arranged, by combining the sets for each formulae.

Resolution

One simple inference rule – *resolution* – for use in a set of clauses.

Given a set of clauses, if there is a pair of clauses which contain two literals, one positive and one negated, which – apart from the negation symbol – unify (as in Prolog unification), then the following clause is a logical consequence:

- unify the two matching literals
- make a new clause by joining the two old clauses *apart from the two matching literals*
- propagate any variable bindings achieved in the unification throughout the new clause.

Resolving:

$\rightharpoonup parent(X, Z) \vee \ \rightharpoonup \ ancestor(Z, Y) \ \vee \ ancestor(X, Y)$

and:

$\rightharpoonup ancestor(peter, W)$

yields:

$\rightharpoonup parent(peter, Z) \vee \ \rightharpoonup \ ancestor(Z, W)$

This can be repeated, creating an expanding collection of clauses, always logically a consequence of the initial set.

Refutation

Notice that if the clauses being resolved are simply P and $\rightharpoonup P$, the result will be an empty clause (no literals); i.e. inconsistency produces the empty clause.

To determine if a given literal Q is a consequence of a (consistent) set of clauses C:

- negate Q (forming $\rightharpoonup Q$)
- add this item to C, forming $\{\rightharpoonup Q\} \cup C$
- repeatedly perform the resolution inference step within the set of clauses, adding the results back into the set
- if a resolution results in an empty clause, $\{\rightharpoonup Q\} \cup C$ was inconsistent
- and in this case $C \supset Q$

Horn clause

A *Horn clause* is a clause where at most one of the literals is positive (not negated).

$$\rightharpoonup parent(X, Z) \vee \rightharpoonup ancestor(Z, Y) \vee ancestor(X, Y)$$

A Horn clause can be rearranged into an implication between a conjunction of (positive) literals and a single (positive) literal:

$$(parent(X, Z) \wedge ancestor(Z, Y)) \supset ancestor(X, Y)$$

This is just like aProlog "clause", with the comma for "conjunction" and "`:-`" as (reverse) implication:

```
ancestor(X,Y) :- parent(X, Z) , ancestor(Z,Y)
```

So a Prolog program can be thought of as a collection of Horn clauses.

Resolution with Horn clauses/Prolog

Resolution with Horn clauses is simple – each clause has at most one positive literal, so resolution is always between that unique positive literal and one of the negated literals from the other clause.

In the Prolog format, the positive literal is the head of the clause.

So resolution consists of unifying a negative literal with the head of a clause.

View negated literals as "goals"– Prolog clause bodies become goals when their head matches.

The program is the base set, C, of clauses.

The top level goal the user supplied is the initial literal Q, which is implicitly negated then added to C.

Finding a clause whose head matches the goal is finding a clause whose one positive literal matches the negated literal that is the current focus.

The original program (C) is the only place where positive literals can be found – results of resolution are always entirely negated literals.

Prolog search as resolution + refutation

Prolog's computation is then a particular search strategy:

- for a current goal (clause of exactly one negated literal), find a clause whose head (unique positive literal) unifies;
- do the unification and resolution;
- resulting clause will be just the body of the previous clause (goal and head matched and so are omitted);
- go to work on this new clause from left-to-right – treat each literal in turn as a goal.
- keep track of pending goals (clause parts to be processed)
- if a resolution ever produces no pending goals, success.

Some impurities

Some Prolog facilities are not properly logical:

Negation: The "`\+`" symbol (or "not" in Clocksin & Mellish) means "not be successfully computable" (not "not logically true"): "negation as failure" – see earlier lecture.. Also

```
\+ racoon(rocky).
```

is not acceptable as a program clause.

The cut: The "`!`" operator acts on the search space, not on the objects being computed upon. It can cause logically valid deductions to be overlooked.

Any genuinely "meta" predicates: `functor`, `var`, `=`, etc.

"Evaluable" predicates: `write`, `read`, `tab`, etc. (Predicates like "`<`" are also evaluable, but fit in neatly with the logical account.)

Summary

- Horn clauses are a general logical representation
- Resolution (with refutation) is a general inference mechanism
- Pure Prolog can be viewed as a particular search process for a Horn clause resolution theorem-prover
- There are impure aspects