1

Knowledge Based Systems

(CM0377)
Lecture 9

(Last modified 11th April 2002)

2

Reasoning about programs

- Prolog allows us to write programs that reason about - and even modify - themselves:
  - *Meta-predicates* take object-level clauses as arguments;
  - *Second-order predicates* take first-order predicates as arguments.
- We'll use such predicates to write:
  - various Prolog interpreters
  - a forward chaining inference engine for an expert system
  - interpreters for non-classical reasoning

3

Motivating Example

- Consider the following program:

```
parent_of(bill, jim).
parent_of(jane, jim).
parent_of(jim, john).
parent_of(john, fred).

male(bill). male(jim). male(john). male(fred).
female(jane).

father_of(F, C):- parent_of(F, C), male(F).
mother_of(M, C):- parent_of(M, C), female(M).
```

4

Motivating example (ctd.)

- Would be nice if we could take two lists such as [bill, jim, john] and [jim, john, fred], and check whether each person in first list is a parent of corresponding person in second list. One way is:

```
parents([], []).
parents([P | Ps], [C | Cs]):-
  parent_of(P, C), parents(Ps, Cs).
```

- e.g. ?- parents([bill, jim, john], [jim, john, fred])

5

Motivating example (ctd.)

- But would be nice if we could do the same for mother_of and father_of.
- We can use the 'univ' built-in predicate to build a term from a list, e.g.

```
?- Term =.. [parent_of, X, jim]
   Term = parent_of(X, jim)
?- parent_of(jane, Y) =.. List
   List = [parent_of, jane, Y]
```

6

Motivating example (ctd.)

- So we can define a *second-order* predicate ***rel*** that we can give the name of the predicate we actually want to test, e.g.

```
rel(parent_of, Ps, Cs)
```

- the code is:

```
rel(R, [], [])
rel(R, [X | Xs], [Y | Ys]):-
  Goal =.. [R, X, Y],
  Goal,
  rel(R, Xs, Ys).
```

7

Motivating example (ctd.)

- So the previous 'parents' goal is replaced by
  ?- rel(parent_of, [bill, jim, john], [jim, john, fred])
- And we could also try, e.g.
  ?- rel(mother_of, [jane], [jim]).
- (And, of course, queries that would fail.)
- Unassessed exercise: draw SLD trees for these queries.

8

Finding all solutions

- Now suppose we want to find *all* children. One way would be to add clause:

```
find_kids:-parent_of(P, C),
  write(C), nl, fail.
find_kids.
```

- But this only writes them out!

9

The findall predicate

- To get a list of all the children, use **findall**:

```
| ?- findall(C, parent_of(Parent, C), Children).
Children = [jim,jim,john,fred]
```

- Note this doesn't find a specific value for Parent: it tries each in turn.
- See Flach pp. 67-68 for other related predicates, **bagof** and **setof**, if interested

10

Manipulating clauses

- We have already seen how *call* and *not* can be used to treat terms as if they are atoms, and use them as queries. Prolog also allows us to:
  - Construct and add a new clause to a program dynamically
  - Remove a clause dynamically
  - 'Dissect' a clause

11

Clauses and terms

- A clause such as:

```
father_of(X, Y):-
  parent_of(X, Y), male(X).
```

- is identical, internally, to:

```
':-'(father_of(X, Y),
  ','(parent_of(X, Y),
      male(X))).
```

- Using **assert** we can construct a term then add it to a program ...

12

Using **assert**

- Example:

```
| ?- A=father_of(X, Y), B=parent_of(X, Y),
  C=male(X), assert((A:-B, C)).

A = father_of(X,Y),
B = parent_of(X,Y),
C = male(X) ?
yes
| ?- listing.
...
father_of(A, B) :-
        parent_of(A, B),
        male(A).
```

13

Using **assert** (ctd.)

- Alternatives:

```
| ?- A=father_of(X, Y),
     B=parent_of(X, Y), C=male(X),
     assert(':-'(A, ','(B, C))).
```

- and:

```
| ?- assert((father_of(X, Y):-
  parent_of(X, Y), male(X))).
```

14

Retract ...

- Similarly, can remove one or more clauses using **retract**. `retract(Clause)` retracts the first clause which unifies with `Clause` from our program. On backtracking, it'll remove the next one, etc. E.g. with the 'motivating example' given earlier, we could do the following:

```
| ?- Z=parent_of(X, Y), retract(Z), write(Z), nl, fail.
parent_of(bill,jim)
parent_of(jane,jim)
parent_of(jim,john)
parent_of(john,fred)

no
| ?-
```

15

The **clause** predicate

- **clause** allows us to find out what clauses are in a program. A goal of the form `clause(H,B)` will unify `H` with the head and `B` with the body of the first clause that unifies with `H:-B`. E.g., in the above program, `clause(parent_of(X, Y), Z)` would have `X=bill, Y=jim, Z=true` as first solution.

16

BUT...

- This is only guaranteed to work (as is assert/retract) if any predicate is declared as dynamic in the program. E.g., at start of program:

```
:- dynamic(parent_of/2).
```

- The number after the '/' is the *arity* of the predicate so declared.
- Alternatively, load a program making *everything* dynamic by using:

```
?- load_files([<list of files to load>],
  [compilation_mode(assert_all)]).
```

- e.g.

```
?- load_files([famil],
  [compilation_mode(assert_all)]).
```

17

A Prolog interpreter in Prolog!

- We can define an interpreter for 'pure' Prolog in Prolog itself:

```
prove(Goal):-                    (i)
  clause(Goal, Body),
  prove(Body).
prove((Goal1, Goal2)):-          (ii)
  prove(Goal1), prove(Goal2).
prove(true).                     (iii)
```

18

Example

- Suppose we have the program:

```
male(mark).  parent_of(mark, jim).
father_of(X, Y):-parent_of(X, Y), male(X).
```

- Then the query `?-prove(father_of(X, Y))` is processed thus:
  - Clause (i) finds a clause, `Goal=father_of(X, Y)`, `Body=','(parent_of(X, Y), male(X))`.
  - To prove `Body`, clause (ii) is used. It tries to prove `parent_of(X, Y)` and `male(X)`.
  - To prove `parent_of(X, Y)`, clause (i) finds a clause, `Goal=parent_of(mark, jim)` and `Body=true`.
  - To prove `true`, clause (iii) succeeds trivially.
  - ETC.

19

A spanner in the works

- `clause(true, Body)` will cause an error (similarly for any built-in predicate). A small modification fixes this:

```
prove(Goal):-
  \+(Goal  = true),
  clause(Goal, Body),
  prove(Body).
prove((Goal1, Goal2)):-
  prove(Goal1), prove(Goal2).
prove(true).
```

20

A meta-circular interpreter

- If our first interpreter had worked, it could have interpreted itself. To make this new one interpret itself, it also has to be able to handle *not (\+),* = and *','* specially:

```
prove(clause(X, Y)):-clause(X, Y).
prove(A=A).
prove(\+ A):- \+ prove(A).
prove(Goal):-
   \+(Goal = true), \+(Goal = ','(_, _)), \+(Goal =
   \+(_)),
   \+(Goal = '='(_, _)), \+(Goal = clause(_, _)),
   clause(Goal, Body), prove(Body).
prove((Goal1, Goal2)):-
   prove(Goal1),
   prove(Goal2).
prove(true).
```

21

The meta-circular interpreter

- Now the goal `?-prove(prove(father_of(X, Y)))` will give the correct answer, as will `?-prove(prove(prove(father_of(X, Y))))`, etc.
- These techniques will be useful for implementing alternative inference strategies and expert systems.
- Further reading: see Flach, pp. 66-74, especially 66/67 and 70/71. Also, see pp. 63-66 for some useful list-handling techniques.