1

Knowledge Based Systems

(CM0377)

Lecture 7

(Last modified 5th March 2002)

2

Motivation

- How to express:
  - Everyone has a mother
- Answer - by introducing *abstract names,* e.g.
  `has_mother(X, mother_of(X))`
- So Andrew's mother is
  `mother_of(andrew)` and
  `has_mother(andrew, mother_of(andrew))`
  is a logical consequence

3

Full clausal logic

- The structure illustrated above is not part of relational clausal logic: *full clausal logic* allows the introduction of such 'nested' terms.

4

Full clausal logic - syntax

- *Simple terms* - constants and variables
- *Complex terms - functor,* followed by a sequence of terms in brackets separated by commas
- Otherwise as relational clausal logic

5

Semantics

- A *term* denotes an individual; an *atom* denotes a proposition about individuals
- *Herbrand Universe* - Set of ground terms that can be constructed from the constants and functors in a program.
- As before,
  - *Herbrand base* of a program *P* is set of ground atoms that can be constructed from predicates in *P;*
  - A *Herbrand interpretation* of *P* is a subset of the H. base;
  - An interpretation is a *model* of a program if it's a model of every ground instance of every clause.

6

Example

- Let *P* be the program:

```
integer(0).
integer(s(X)):-integer(X).
```

- Herbrand universe of *P:*

```
{0, s(0), s(s(0)), ...}
```

- Herbrand base of *P:*

```
{integer(0), integer(s(0)),
integer(s(s(0))), ...}
```

7

Example (ctd.)

- A model of *P* must be a model of every ground instance of every clause, i.e. of
  - `integer(0)` (i)
  - `integer(s(0)):-integer(0)` (ii)
  - `integer(s(s(0))):-integer(s(0))` (iii)
  - ...
- So `integer(0)` must be **true** in any model (from (i)); `integer(s(0))` must be **true** in any model (from (ii), since `integer(0)` is **true**); similarly `integer(s(s(0)))` must be **true** in any model (from (iii)), etc.
- So any model of this program is infinite!

8

Another example

- Consider the program *P*:

```
ancestor_of(X, Y, none):-parent_of(X, Y).     (i)
parent_of(bill, jim).                         (ii)
ancestor_of(X, Y, list(Z, Intermediates)):-
    parent_of(X, Z),
    ancestor_of(Z, Y, Intermediates).         (iii)
```

- Let's try to build a *minimal* model, where nothing is assigned to **true** unless it *has* to be.
- First observe that, in any model, `parent_of(bill, jim)` must be assigned to **true** (clause (ii)).

9

Another example (ctd.)

- Now suppose `parent_of(X, Y)` is assigned to **false** for all X, Y other than `{X->bill, Y->jim}`. Then all ground instances of (i) are clearly necessarily true in such an interpretation, except for:

```
ancestor_of(bill, jim, none):-parent_of(bill, jim).
```

- So `ancestor_of(bill, jim, none)` must also be assigned to **true** in any model of *P*.

10

Another example (ctd.)

- Similarly, all ground instances of (iii) are necessarily **true** except those matching:

```
ancestor_of(bill, Y, list(jim, Intermediates)):-
    parent_of(bill, jim),
    ancestor_of(jim, Y, Intermediates).
```

- In fact, *all* ground instances of the above clause are also true in the interpretation we are building, as long as `ancestor_of(jim, Y, Intermediate)` is **false** in the interpretation for all `Y` and for all `Intermediate`. In other words, our minimal model is:

```
{parent_of(bill, jim), ancestor_of(bill, jim, none)}
```

- Note that this is *finite!*

11

Proof theory

- Resolution is the same as for relational clausal logic, but with unification redefined to find a *most general unifier* of 2 terms by:
  - renaming variables so that the two terms have none in common;
  - finding first subterm pair where the two atoms differ;
  - if neither subterm is a variable, unification fails;
  - else substitute the other term for all occurrences of the variable;
  - repeat with next pair of differing subterms until no more differences found
- The set of partial substitutions arising from this process comprises the mgu

12

Example

- Unify:

```
greater_than(s(X), X)
```

  and

```
greater_than(s(s(Z)), s(0))
```

- The mgu is `{X -> s(Z), Z-> 0}`
- Giving

```
greater_than(s(s(0)), s(0))
```

13

Example (ctd.)

- Now consider:
  ```
  greater_than(s(X), X).
  :- greater_than(Y, Y).
  ```
- Unifying the two greater_than atoms involves the substitution `{Y -> s(X)}`
- So now we're unifying:
  ```
  greater_than(s(X), X).
  :- greater_than(s(X), s(X)).
  ```
- So apply substitution `{X -> s(X)}`
- But the 'X' in s(X) therefore needs to be replaced by s(<whatever X is replaced by>), i.e. `{X -> s(s(s(...)))}`

14

Problems ...

1. This is an infinite term - not defined in the Herbrand base
2. `greater_than(Y, Y)` *isn't* a logical consequence of `greater_than(s(X), X).`

- Include *occurs check* (X can't unify with any term containing X as a subterm) to make resolution sound again.
- NB Prolog *doesn't* have occurs check as standard, because it's expensive computationally. So need to write programs bearing this in mind!!

15

Meta-theory

- Full clausal logic is sound (if unification includes the occurs check), refutation complete but not complete.
- It's *semi-decidable:* if *C* is a logical consequence of *P* then we can determine it in a finite time. But if not, no algorithm is guaranteed to halt establishing this for arbitrary *P* and *C*.
- So take care to avoid looping in Prolog!

16

Definite clause logic

- Restricting clauses to have *one* literal in the head ...
  - yields much greater efficiency
  - reduces expressivity
  - can be used in conjunction with negated literals in the body to partially solve the expressivity problem

17

Example

- Consider

```
get_wet(X); umbrella(X):-rains_on(X).   (i)
rains_on(fred).                         (ii)
:-umbrella(bill).                       (iii)
```

- Logical consequences include:

```
get_wet(fred); umbrella(fred)           (iv)
get_wet(bill):-rains_on(bill)           (v)
```

- Notice (iv) is indefinite - we don't know if Fred gets wet or he has an umbrella.
- Also, it would be much more efficient if we always knew in advance which literal to resolve on in each clause.

18

Solution

- Definite clausal logic - every clause has precisely one literal in head, i.e. of form:
  $A :- B_1, \ldots, B_n$
- And the clauses will only be used to prove *A,* and will do so by proving each of B_1, ..., B_n in turn. Much more efficient - hence use in Prolog.

19

Reintroducing 'indefiniteness'

- Consider:
  `get_wet(X); umbrella(X):-rains_on(X).`
- To prove `gets_wet(X)` need to prove `rains_on(X)` is **true** *and* that `umbrella(X)` is **false**.
- Similarly, to prove `umbrella(X)` need to prove `rains_on(X)` is **true** *and* that `gets_wet(X)` is **false**.

20

General clauses

- Can extend definite clauses by introducing *not,* retaining the above information:
  `get_wet(X):-rains_on(X), not umbrella(X).`
  `umbrella(X):-rains_on(X), not gets_wet(X).`
- We shall soon see that Prolog implements *not* as *negation as failure.*

21

Extra reading

- To help you to see the relationship of clausal logic to standard propositional and predicate logic, read sec. 2.5 of Flach's book.

22

Resolution and Prolog

- Prolog implements resolution by implementing a particular strategy, SLD (**S**election rule **L**inear resolution **D**efinite clause) resolution.
- So we can't write a Prolog program without thinking *how* it will run!

23

Example

```
customer_of(P, M):-owns(P, O), makes(M, O).
owns(jim, pavillion).
owns(fred, zx81).
owns(jane, spectrum).
makes(sinclair, spectrum).
makes(hp, pavillion).
makes(sinclair, zx81).
```

- Query:

```
?- customer_of(P, sinclair).
```

- 2 possible answers `{P->fred}, {P->jane}`

24

So how does it work?

- First resolve the query with first clause:
  `:-owns(P, O), makes(sinclair, O).`
- Which to resolve on now? Our Prolog *selection rule* chooses the left-most, and searches top-down, reaching `owns(jim, pavillion)` first. Resolving we get:

```
:- makes(sinclair, pavillion).
```

- ...dead-end!

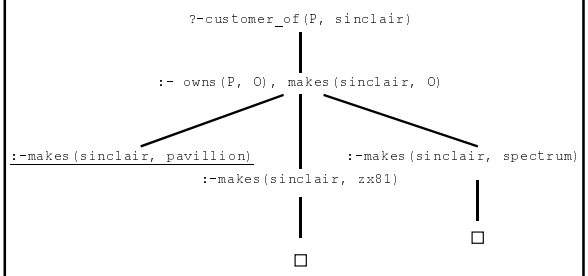
25

How it works (ctd.)

- Backtrack to next `owns` clause, `owns(fred, zx81)`, yielding:
  `:- makes(sinclair, zx81).`
- Searching now *succeeds.* etc.
- An SLD-tree can be drawn of all resolvents to show the search ...

26

SLD tree



27

Clause order

- Because Prolog effectively searches an SLD tree from left to right, there's a problem if the leftmost branches are infinite. e.g.

```
connected_to(X, Y, Road):-
    connected_to(Y, X, Road).
connected_to(cardiff, london, m4).
```

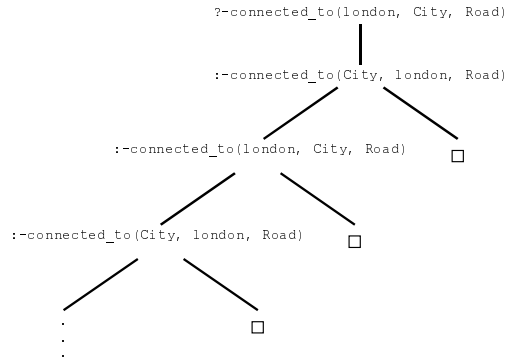
- the query:

```
?-connected_to(london, City, Road).
```

has answer `{City->cardiff, Road->m4}`
but will never be found by Prolog!

28

SLD tree



29

Partial solution

- Reorder clauses:

```
connected_to(cardiff, london, m4).
connected_to(X, Y, Road):-
    connected_to(Y, X, Road).
```

- (Exercise: draw SLD tree for the previous query against the above program)
- But this is still only a *partial* solution to the problem - still infinite number of solutions; won't halt on query that has *no* answer!

30

Some conclusions

- May never reach a success branch in the SLD tree, being trapped in some infinite subtree. *Prolog is incomplete.*
- Any infinite SLD tree causes the Prolog interpreter to loop if no (more) answers are to be found. *Semidecidability of full clausal logic.*
- *Coming soon to a lecture theatre near you ... modifying the search tree using the* ***cut.***