F28PL1 Programming Languages

Lecture 17: Prolog 2

Search summary

- *question* is:
  - *term* or a conjunction of *term*s
  - goal which the system tries to satisfy
- satisfying a goal will usually involve the satisfaction of sub-goals
- for a conjunction of *term*s, system attempts to satisfy each as a *sub-goal* from left to right

Search summary

- for a (sub-)goal:
  - the data base is searched for a *clause* with a *head* with a *functor* that matches the goal's *functor*
  - *arguments* of the *clause head* are then matched against the *arguments* of the goal
  - if the *clause* is a *rule* then an attempts to satisfy the *body* as a new sub-goal
  - *body* satisfaction may complete goal/*clause head* matching
  - matching results passed back to the parent goal

Search summary

- decisions made in satisfying a sub-goal:
  - carried over to subsequent sub-goals
  - can only be undone by the failure to satisfy a subsequent sub-goal
  - resulting in backtracking to that preceding sub-goal

Matching summary

	clause		
goal	atom/integer	variable	structure
atom/integer	fail if not same	instantiate (2)	fail
variable	instantiate (1)	share (3)	instantiate(1)
structure	fail	instantiate (2)	match (*)

(1) goal argument variable instantiated to clause atom, integer or structure

(2) clause argument variable instantiated to goal atom, integer or structure

(3) goal variable and clause variable share: as soon as one is instantiated so is the other

(*) structure matching is recursive

Variable summary

- Prolog has no concept of assignment as a change to a variable's value
  - variables are instantiated to values by matching
  - instantiation can only be undone by backtracking.
- all occurrences of a variable in a *term* are references to the same variable
- a variable may be instantiated as a result of the satisfaction of any sub-goal from the *term*

Variable summary

- the instantiation of a variable in a *rule body* results in:
  - all references to the variable accessing that value
  - the occurrences in the *rule head* are instantiated
- matching results in the instantiation in the corresponding goal argument through sharing
- variable/structure instantiation will delay until the variables in the structure are instantiated.

Example

- Phil & Robin are friends. Chris & Robin are friends. Phil & Pat are friends. Jo & Chris are friends.
- friendship is symmetric: if X is Y’s friend then Y is X’s friend.
- Phil has an invitation to the party. Pat has an invitation to the party.
- You can go to the party if you have an invitation or you have a friend who has an invitation.

Example

```
friends(phil,robin).
friends(chris,robin).
friends(phil,pat).
friends(jo,chris).
friends(F1,F2) :- friends(F2,F1).
invitation(phil).
invitation(pat).
party(P) :- invitation(P).
party(P) :- friends(P,F),invitation(F).
```

Example

- can Robin go to the party?

```
| ?- party(robin).
yes
```

- try: `party(robin) :- invitation(robin)`
  - try: `invitation(robin)`

```
...
```

  - fail & backtrack
- try: `party(robin) :- friends(robin,F),invitation(F)`
  - try: `friends(robin,F)`
  - `...`
  - fail & backtrack

Example

- try: `friends(robin,F2) :- friends(F2,robin)`
- try : `friends(F2,robin)`
  - `...`
  - matches: `friends(phil,robin)`
- try: `invitation(phil)`
- matches: `invitation(phil)`

Example

- can Chris go to the party?

`| ?- party(chris).`

`yes`

- try: `party(chris) :- invitation(chris)`
  - try: `invitation(chris)`

`...`

  - fail & backtrack
- try: `party(chris) :- friends(chris,F),invitation(F)`
  - try: `friends(chris,F)`
  - `...`
  - matches: `friends(chris,robin)`
  - try: `invitation(robin)`
  - `...`
  - fail & backtrack

Example

- try: `friends(chris,F2) :- friends(F2,chris)`
- try : `friends(F2,chris)`
  - `...`
  - matches: `friends(jo,chris)`
- try: `invitation(jo)`
- `...`
- fail & backtrack
- try: `friends(F2,chris) :- friends(chris,F2)`
- `...`

- but already failed with `friends(chris,F)`

```
friends(F1,F2) :- friends(F2,F1).
```

- this never terminates if right hand side fails...!

Limit choices

- only interested in two possibilities: `X` and `Y` are friends or `Y` and `X` are friends:

```
party(X) :- friends(X,Y), invitation(Y).

party(X) :- friends(Y,X), invitation(Y).
```

- but...
- if `invitation(Y)` in first clause fails then will try `invitation(Y)` again in 2nd clause

Refactor

- general case:

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

- if *c* succeeds but *b* fails in 1st clause will backtrack, match *d* and try to match *b* again in second clause
- gather together common sub-goals

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

- now, if first clause of *e* fails (*c*) will try second clause (*d*) but not retry *b*

Example

```
pals(X,Y) :- friends(X,Y).

pals(X,Y) :- friends(Y,X).

party(X) :- pals(X,Y), invitation(Y).
```

Cut

- ! - cut operator
- prevent backtracking where it is unnecessary or incorrect
- commits the system to any choices:
  - made since the start of the satisfaction of the goal
  - which matched the rule containing the cut
- backtracking over a cut causes that goal to fail

Cut

- someone is popular if they can go to the party and they don’t talk about computing

```
popular(P) :-
        party(P), no_computer_talk(P).
```

- consider Eric, who can go to the party but is a computer buff:

```
invitation(eric).
```

Cut

- try: `popular(eric)`
  - try: `party(eric)`
  - try: `invitation(eric)`
    - matches: `invitation(eric)`
  - try: `no_computer_talk(eric)`
  - fail: so backtrack
  - try: `party(eric)` again

Cut

- backtracking may be prevented by:

```
popular(P) :- party(P),!,no_computer_talk(P).
```

- if:

```
no_computer_talk(P)
```

- fails then the goal which matched the rule:

```
popular(P)
```

- will fail, in this case:

```
popular(erik)
```

Anonymous variable

- _ - underline
- matches anything
- nothing is shared or instantiated

Equality

X = *Y*

- compares *X* and *Y* for structural equality
- works for all terms
- = same as: `equal(X,X)`

X \= *Y*

- succeeds if *X* not equal to *Y*

Arithmetic expressions

+ - addition

- - subtraction

* - multiplication

/ - division

brackets:

(. . .)

used to impose an explicit evaluation order

Arithmetic expressions

- "arithmetic expressions" are just infix structures
- not normally evaluated
- may be treated in the same way as any other structure
  - e.g. pattern matching

```
| ?- operands(X+Y,X,Y).

| ?- operands(66+77,O1,O2).

O1 = 66

O2 = 77
```

Arithmetic evaluation

`is`

- operator to enforce evaluation

X `is` *Y*

- *X* is a variable
- *Y* is a term with all variables instantiated
- the “expression” *Y* is evaluated
- if the variable X is instantiated
  - then *X*'s value and the result are compared
- otherwise, *X* is instantiated to the result

Arithmetic evaluation

```
| ? - sumsq(X,Y,Z) :-
          Z is (X*X)+(Y*Y).
| ?- sumsq(3,4,25).
yes
| ?- sumsq(5,5,F).
F = 50
```

Arithmetic evaluation

- right hand side of `is` must be fully instantiated
- can’t use `is` to find left hand side values which make an “expression” evaluate to a right hand side value
- so, above example can be used to:
  - check that an `X`, `Y` and `Z` have the `sumsq` relationship
  - find `Z` from `X` and `Y`
- can’t be used to find `X` or `Y` from `Z`

Arithmetic evaluation

- `is` is not an assignment operator

`X is X+1`

- will always fail
- if `X` is uninstantiated
  - then `X+1` fails
  - `X` can’t be incremented
- if `X` is instantiated
  - then `X` can never match `X+1`

Numeric recursion

- find sum of first N integers:
- sum of first 0 is 0
- sum of first N is N more than sum of first N-1

```
sum(0,0).
sum(N,S) :- N1 is N-1, sum(N1,S1),S is S1+N.
```

- NB can't just invoke rule with expression argument
  - must evaluate expression explicitly

Numeric recursion

```
| ?- sum(3,S).
s = 6
```

- try: `sum(3,S) :- N1 is 3-1, sum(N1,S1),S is S1+3`
  - try: `N1 is 3-1` – `N1` is `2`
  - try: `sum(2,S1)`
  - try: `sum(2,S1) :- N1’ is 2-1,sum(N1’,S1’),S1 is S1’+2`
    - try: `N1’ is 2-1` – `N1’` is `1`
    - try: `sum(1,S1’)`
      - try: `sum(1,S1’) :- N1’’ is 1-1,sum(N1’’,S1’’),`
      - `S1’ is S1’’+1`

        try: `N1’’ is 1-1` – `N1’’` is `0`

        try: `sum(0,S1’’)`
        - matches: `sum(0,0)` – `S1’’` instantiated to `0`

Numeric recursion

        - try: `S1' is 0+1 – S1' is 1`
    - try: `S1 is 1+2 – S1 is 3`
- try: `S is 3+3 – S is 6`

Numeric comparison

= - equality

\= - inequality

> - greater than

< - less than

>= - greater than or equal to

=< - less than or equal to

- both operands must be instantiated to numbers
  - apart from = and \=

Database manipulation

`asserta(`*X*`)`

- *X* is an instantiated term
- adds *X* to the database
- before the other clauses with the same functor as *X*

`assertz(`*X*`)`

- adds *X* to the database
- after the other clauses with the same functor as *X*

Database manipulation

`retract(`*X*`)`

- *X* is a term
- removes first clause matching *X* from database
- NB in SICSTUS, cannot `assert`/`retract` clauses with functors like those loaded at start of program

Database manipulation

- e.g. count how often clauses with the functor
- `invitation` occur in the database
- need to repeatedly check database
- can’t use recursion to find invitations as each level will start from database beginning
- can’t combine backtracking with counting
  - each backtrack will reverse count
- keep count as clause in database

Database manipulation

```
check_invitations(N) :-
 asserta(count(0)),
 count_invitations(N).
```

- puts: `count(0)`into the database
- calls: `count_invitations(N)`

Database manipulation

```
count_invitations(N) :- invitation(_),
                        increment.
count_invitations(N) :- retract(count(N)).
```

- find an `invitation`
- call `increment`
  - add one to the count
  - fail & backtrack to find next `invitation`
- if finding `invitation` fails then:
  - backtrack to second option
  - retract: `count(N)` from the database
  - setting `N` to the final count

Failure

`fail`

- always fails
- backtrack to next option for previous sub-goal
- often use: `!,fail` to make current goal fail completely
- NB over use of `!,fail` can makes program sequential

Database manipulation

- to keep count:

```
increment :-
 retract(count(N)), N1 is N+1,
 asserta(count(N1)), !, fail.
```

- removes: `count(N)` from the database
  - setting `N` to the current count
- sets `N1` to `N+1`
- puts: `count(N1)` back into the database
- `fail` backtracks to `!` so `increment` fails

Database manipulation

`| ?- check_invitations(N).`

`N = 3`

- try: `check_invitations(N) :- assert(count(0)),count_invitations(N)`
  - try: `assert(count(0))`
  - `count(0)` now in database
  - try: `count_invitations(N) :- invitation(_), increment`
  - `try: invitation(_)`
    - matches: `invitation(pat)`

Database manipulation

- try: `increment :- retract(count(N)), N1 is N+1, assert(count(N1)),!,fail`
  - try: `retract(count(N))`
    - matches: `count(0) - N` is `0`
  - try: `N1 is N+1`
    - `N1` is `1`
  - try: `assert(count(N1))`
    - `count(1)` now in database
  - `!, fail - increment` fails – backtrack
- try: `invitation(_)`
  - matches: `invitation(phil)`

Database manipulation

- try: `increment :- retract(count(N)), N1 is N+1, assert(count(N1)),!,fail`
  - try: `retract(count(N))`
    - matches: `count(1) - N` is `1`
  - try: `N1 is N+1`
    - `N1` is `2`
  - try: `assert(count(N1))`
    - `count(2)` now in database
  - `!, fail - increment` fails – backtrack
- try: `invitation(_)`
  - matches: `invitation(eric)`

Database manipulation

- try: `increment :- retract(count(N)), N1 is N+1, assert(count(N1)),!,fail`
  - try: `retract(count(N))`
    - matches: `count(2) - N` is 2
  - try: `N1 is N+1`
    - N1 is 3
  - try: `assert(count(N1))`
    - `count(3)` now in database
  - `!, fail – increment` fails - backtrack
- try: `invitation(_)`
  - fail & backtrack

Database manipulation

- try: `count_invitations(N) :-`
  - `retract(count(N))`

    matches: `count(3) – N` is `3`
- imperative style of programming
- treating database as memory
- treating `assert/retract` as assign/get value