EECS 3401 — AI and Logic Prog. — Lecture 9

Adapted from slides of Yves Lesperance

Vitaliy Batusov
vbatusov@cse.yorku.ca

York University

October 19, 2020

Prolog

- Today: **Prolog**. Control flow, Negation, Second-Order programming, Tail recursion
- Required reading: Clocksin & Mellish Chapters 3, 4, 6, 10

Simplicity Hides Complexity

- Simple composition (via $\wedge$ and $\vee$) of goals hides complex control patterns
- Not easily represented by traditional flowcharts
- May not be a bad thing
- Want important aspects of logic and algorithm to be clearly represented and irrelevant details to be left out

Semantics: Procedural and Declarative

- Prolog programs have both a declarative (logical) semantics and a procedural semantics

- **Declarative**: query holds if it is a logical consequence of the program
- **Procedural**: query succeeds if a matching fact or rule succeeds
  Defines order in which goals are attempted, what happens when they fail, etc.

Conjunction and Disjunction

- Prolog's *and* (the comma `,`) and *or* (semicolon `;`)[1] are not purely logical operations
- It is often important to consider the order in which goals are attempted
  - Left-to-right for `,` and `;`
  - Top-to-bottom for facts/rules

[1] Recall, this is syntactic sugar for having multiple rules defining same goal

Conjunction is not always commutative

- From last time:

```
?- cousinOf(X,Y), american(X).
```

  vs

```
?- american(X), cousinOf(X,Y).
```

- Also from an earlier lecture:

```
?- parent(P, C1), parent(P, C2), not(C1 = C2).
```

  vs

```
?- parent(P, C1), not(C1 = C2), parent(P, C2).
```

Conjunction is not always commutative

Querying the same knowledge base:

```
?- parent(P, C1), parent(P, C2), not(C1 = C2).
P = 'Mary',
C1 = 'Elizabeth',
C2 = 'Margaret' .
```

```
?- parent(P, C1), not(C1 = C2), parent(P, C2).
false.
```

Why?

Uses of `;`

- Disjunction operator `;` can be used to regroup several rules with the same head

```
parent(X,Y) :- mother(X,Y); father(X,Y).
```

- Can improve efficiency by avoiding redoing unification
- Important: `;` has lower precedence than `,`, just like in logic

Prolog negation

- Prolog uses **Negation As Failure**, not logical negation!
- Operator: `\+` — "not provable"
- The operator `not` is a deprecated synonym for `\+`, feel free to use either in this course
- Negation As Failure means: **if the statement cannot be proven by Prolog, assume it's false**
- `\+ goal` succeeds if `goal` fails.
- Interpreting `\+` as negation amounts to making the **closed-world assumption** (CWA)

NAF Example

- KB:

```
human(ulyssus).
human(penelope).
mortal(X) :- human(X).
```

- Query:

```
?- \+ human(jason).
true.
```

- In proper logical semantics, these axioms do not bear out $\neg human(jason)$.

Semantics of free variables under `\+` is weird

- Normally, variables in a query are exitstentially quantified from outside
- Query `?- p(X), q(X).` means "there exists `X` such that `p(X)` and `q(X)`".
- But query `?- \+(( p(X), q(X) )).` means "it is not the case that there exists `X` such that `p(X)` and `q(X)`".

  Contrast with "there exists `X` for which neither `p(X)` nor `q(X)` holds" — a totally different meaning

To Avoid Pitfalls

- `\+` works correctly if its argument is instantiated
- For example, in the rule
  `intersect([X|L],Y,I):- \+ member(X,Y), intersect(L,Y,I).`
  `X` and `Y` should both be instantiated.

Another Example

- Program: `animal(cat). vegetable(turnip).`
- Queries:

```
?- \+ animal(X), vegetable(X).
false.
/* Why??? */

?- vegetable(X), \+ animal(X).
X = turnip.
```

Guarding the "else"

- Suppose you have two rules for same predicate, which cover two mutually exclusive cases
- Can't rely on implicit negation in predicates that can be redone
- Whenever there are alternative rules and backtracking, each rule should be logically valid
- Safe bet: in the "else" rule, repeat the guarding condition with negation

Example: intersect

Bad:

```
intersect([], _, []).
intersect([X|L],Y,[X|I]):- member(X,Y), intersect(L,Y,I).
intersect([X|L],Y,I):- /* nothing here */ intersect(L,Y,I).

?- intersect([a], [b, a], []).
true.  /* Why??? */
```

Good:

```
intersect([], _, []).
intersect([X|L],Y,[X|I]):- member(X,Y), intersect(L,Y,I).
intersect([X|L],Y,I):- \+ member(X,Y), intersect(L,Y,I).

?- intersect([a], [b, a], []).
false.
```

Example: `intersect` using the Cut

Also good (and more efficient), using the **Cut**:

```
intersect([], _, []).
intersect([X|L],Y,Z):- member(X,Y), !,
                        Z=[X|I], intersect(L,Y,I).
intersect([X|L],Y,I):- /* no guard */ intersect(L,Y,I).


?- intersect([a], [b, a], []).
false.
```

Why more efficient?

Cut used to define useful features

- Built-in goal `fail` **always fails**.

```
?- fail.
false.
```

- If goal `g` should be false when `c_1`, . . . , `c_n` holds, can write

```
g :- c_1, ... , c_n, !, fail.
```

- With this pattern, we can actually define Prolog negation:

```
\+ g :- g, !, fail.
\+ g.
```

Some Control Predicates

- `true` — bult-in goal, always succeeds
- `fail` — always fails
- `repeat` — always succeeds, infinite number of choice points

```
loopUntilNoMore :- repeat, doStuff, checkNoMore.
```

but tail recursion is cleaner:

```
loop :- doStuff, (checkNoMore; loop).
```

Forcing All Solutions

- Program:

```
test :- member(X, [1,2,3]), nl, print(X), fail.
```

- `print` and `nl` have no alternative solutions, but `member` does:

```
?- test.


1
2
3
false.
```

Second-Order Features: `bagof` and `setof`

- `bagof(T,G,L)` — instantiates `L` to the list of all instances of `T` for which goal `G` succeeds
- Example:

```
?- member(X,[2,5,7,3,5]),X >= 3.
X = 5 ;
X = 7 ;
X = 3 ;
X = 5.

?- bagof(X, (member(X,[2,5,7,3,5]),X >= 3), L).
L = [5, 7, 3, 5].
```

Second-Order Features: `bagof` and `setof`

- `setof` is similar to `bagof`, except it removes duplicates from the output list:

```
?- bagof(X, (member(X,[2,5,7,3,5]),X >= 3), L).
L = [5, 7, 3, 5].

?- setof(X,(member(X,[2,5,7,3,5]),X >= 3),L).
L = [3, 5, 7].
```

- Can also collect values of several variables by putting them in a struct:

```
?- bagof(pair(X,Y),
         (member(X,[a,b]),member(Y,[c,d])),
         L).
L = [pair(a, c), pair(a, d), pair(b, c), pair(b, d)].
```

Second-Order Features

- `setof` and `bagof` are called "Second-Order" features because they are queries about the value of a **set or relation**, as opposed to measly individuals
- In logic, this would be quantification over predicates
- Not allowed in FOL; this is what Second-Order Logic is for

Tail recursion optimization in Prolog

- Suppose we have:
  - Goal A
  - Rule A' `:-` B_1 `,` B_2 `,` $\ldots$ `,` B_{n-1} `,` B_n `.`
  - Goal A unifies with head A'
  - Sub-goals B_1 `,` B_2 `,` $\ldots$ `,` B_{n-1} all succeed.
- If there are no alternatives[2] left for A and for B_1 `,` B_2 `,` $\ldots$ `,` B_{n-1}, then we can simply **replace** the goal A by sub-goal B_n on execution stack
- In such cases, the predicate A is **tail-recursive**
- Whether B_n succeeds or fails, there is nothing left to do in A, so we can replace the call stack frame for A by that of B_n. Then, the recursion can be as space efficient as iteration

[2] i.e., ways of proving

Example: factorial

- Recall this implementaion of factorial:

```
f(0,1).
f(N,F):- N>0, M is N-1, f(M,F1), F is N*F1.
```

- Close to mathematical definition
- Not tail-recursive
- Requires $O(N)$ in stack space

Example: factorial

- A better implementation

```
f(N,F):- f1(N,1,F). /* alias */

f1(0,F,F).
f1(N,T,F):- N>0, T1 is T*N, N1 is N-1, f1(N1,T1,F).
```

- Uses an accumulator
- Is tail-recursive and each call can replace the previous call
- Can prove correctness

Example: `append`

```
append([],L,L).
append([X|R],L,[X|RL]):- append(R,L,RL).
```

- `append` is tail-recursive if the first argument is fully instantiated
- Prolog must detect the fact that there are no alternatives left; may depend on clause indexing mechanism used

Example: `split`

```
split([],[],[]).
split([X],[X],[]).
split([X1,X2|R],[X1|R1],[X2|R2]):- split(R,R1,R2).
```

Tail-recursive!

Example: `merge`

```
merge([],L,L).
merge(L,[],L).
merge([X1|R1],[X2|R2],[X1|R]) :-
        order(X1,X2), merge(R1,[X2|R2],R).
merge([X1|R1],[X2|R2],[X2|R]) :-
        not order(X1,X2), merge([X1|R1],R2,R).
```

Tail-recursive, but the lack of alternatives may be hard to detect (can use cut to simplify)

Example: `mergesort`

```
mergesort([],[]).
mergesort([X],[X]).
mergesort(L,S):- split(L,L1,L2),
        mergesort(L1,S1), mergesort(L2,S2), merge(S1,S2,S).
```

A really cool example: Finite State Automata

A Finite State Automaton $(\Sigma, S, s_0, \delta, F)$ is a representation of a machine as

- a finite set of states S
- an input alphabet Σ
- a state transition relation/table δ, where the current state $(\in S)$ and current input symbol $(\in \Sigma)$ are mapped to next state $(\in S)$
- an initial state s_0
- a set of final states F

A FSA *accepts* an input sequence over alphabet Σ if, starting in the designated starting state s_0, scanning the input sequence leaves the automaton in a final state $(\in F)$

FSA: example

Consider:

- An automaton that accepts strings of symbols over the alphabet $\{x, y\}$ which contain an even number of x's and an odd number of y's.
- Idea: keep track of whether we've seen an even or odd number of each symbol
- $S = \{ee, eo, oe, oo\}$
- $s_0 = ee$
- $\delta = \{(ee, x, oe), (ee, y, eo), \ldots\}$
- $F = \{eo\}$

Implementation

- `fsa(Input)` succeeds if and only if the FSA accepts or recognizes the sequence (list) `Input`
- Initial state represented by a predicate `initial_state(State)`
- Final state represented by a predicate `final_states(List)`
- State transition table represented by a predicate
  `next_state(State, InputSymbol, NextState)`
  Note: `next_state` need not be a function

Implementing `fsa/1`

```
fsa(Input) :- initial_state(S), scan(Input, S).

scan([], State) :- final_states(F), member(State, F).
scan([Symbol | Seq], State) :-
        next_state(State, Symbol, Next), scan(Seq, Next).
```

Result Propagation

- `scan` uses "pumping"/result propagation
- Carries around current state and the remainder of the input sequence
- If FSA is deterministic, when the end of input is reached, can make an accept/reject decision immediately; tail recursion optimization can be applied
- If FSA is non-deterministic, may have to backtrack; must keep track of remaining alternatives on execution stack

Non-determinism

- A non-deterministic FSA accepts an input sequence if there exists *at least one sequence* which leaves the automaton in one of its final states
- `?- fsa(Input).`
- `scan` searches through all possible choices for `Symbol` at each state, fails only if no sequence leads to a final state

Representing tables

- Can use a binary connector, e.g. `A-B-C` instead of `next_state(A,B,C)` — looks cleaner, may help in spotting typos
- Program may look something like

```
ee-x-oe.  /* Insted of next_state(ee,x,oe) */
ee-y-eo.
oe-x-ee.
oe-y-oo.

scan([], State) :- final_states(F), member(State, F).
scan([Symbol | Seq], State) :-
        State-Symbol-Next, scan(Seq, Next).
```

End of Lecture

- Next time: **Search**