Programming Languages

Lecture 8: Logic Programming Languages

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Blacksburg, Virginia 24061 USA

History and Goals of Logic Programming

- Competitor to LISP for AI programming in 80's
- Adopted by Japanese for Fifth Generation Computing Project (Prolog).
- What is logic programming?
  - Programming based on the notion of logical deduction in symbolic logic.
  - Implementation typically based on mechanisms for automatic theorem proving.

History and Goals of Logic Programming (cont)

- "A constructive proof that for every list L there is a corresponding sorted list S composed of the same elements as L yields an algorithm for sorting a list."
- Philosophy is shared by others not working on "logic programming"
  - Constable at Cornell, Martin-Löf in Sweden, Calculus of Constructions group in France.
  - These groups want to extract (more traditional) program from constructive proofs.
- Very-High-Level Languages - non-procedural
- State what must be done, not how to do it. Idea is to separate logic from control.

Introduction to Prolog

- Prolog (PROgramming in LOGic), first and most important logic programming language.
- Developed in 1972 by Alain Colmerauer in Marseilles.
- Relational rather than functional programming language
- Often best to start out as thinking of Prolog in terms of language for working with a data base.

Pure Prolog

- Three types of statements:
  1. Facts (or hypotheses):
     ```
     father(albert,jeffrey).
     ```
  2. Rules (or conditions):
     ```
     grandparent(X,Z) :- parent(X,Y),parent(Y,Z).
     ```
     (read "`:-`" as "if")
  3. Queries (or goals):
     ```
     ?-grandparent(X,jeffrey).
     ```
- Should think of language as non-deterministic (or non-procedural). Looks for all answers satisfying query.

Example: Facts

```
father(ralph,john).
father(ralph,mary).
father(bill,ralph).

mother(sue,john).
mother(sue,mary).
mother(joan,ralph).

male(john).
male(ralph).
male(bill).

female(mary).
female(sue).
```

Example: Rules

```
is_mother(M) :- mother(M,Y).

parent(M,X) :- mother(M,X).
parent(F,X) :- father(F,X).

parents(M,F,X) :- mother(M,X), father(F,X).

sister(X,S) :- female(S), parents(M,F,X), parents(M,F,S), S\=X.

ancester(X,Y) :- parent(X,Y).
ancester(X,Y) :- parent(X,Z),ancester(Z,Y).
```

Selection Sort

```
sel_sort([],[]).
sel_sort(L,[Small|R]) :-
        smallest(L,Small) ,
        delete(Small,L,Rest) ,
        sel_sort(Rest,R) .

/* smallest(List, Small) results in Small being the smallest element in List. */
smallest([Small],Small) .
smallest([H|T],Small) :-
          smallest(T,Small) ,
          Small=<H.
smallest([H|T],H).

/* delete(Elt,List,Result) has Result as List after deleting Elt. */
delete(Elt,[],[]) .
delete(Elt,[Elt|T],T) .
delete(Elt,[H|T],[H|NuTail]) :-
        delete(Elt,T,NuTail) .
```

Insertion Sort

```
ins_sort([],[]) .

ins_sort([H|T],Sorted) :-
        ins_sort(T,Rest),
        insert(H,Rest,Sorted) .

/* insert(Elt,List,Result) - if List is sorted, Result is obtained by putting
     Elt where it fits in order in List. */

insert(Elt,[],[Elt]) .

insert(Elt,[H|T],[Elt,H|T]) :-
        (Elt=<H) .

insert(Elt,[H|T],[H|Sorted]) :-
        insert(Elt,T,Sorted) .
```

QuickSort

```
/* sep(List,Key,Less,More) separates the List into the set of elements less
   than or equal to Key (in Less) and those greater than or equal to Key
   (in More) */
sep([],Key,[],[]) .
sep([H|T],Key,Less,[H|More]) :-
        (H>Key) ,
        sep(T,Key,Less,More) .
sep([H|T],Key,[H|Less],More) :-
        (H=<Key) ,
        sep(T,Key,Less,More) .

quick_sort([],[]) .
quick_sort([H|T],Sorted) :-
        sep(T,H,Less,More) ,
        quick_sort(Less,L_sorted) ,
        quick_sort(More,M_sorted) ,
        concat(L_sorted,[H|M_sorted],Sorted) .

/* concat(First, Second, List) results in List = concatenation of First
and Second. */
concat([],L,L) .
concat([H|T],L,[H|C]) :- concat(T,L,C) .
```

Permutations

Can take advantage of reversibility since computing with relations rather than functions.

```
append([],L,L) .
append([H|T],L,[H|R]) :- append(T,L,R) .

permute([],[]) .
permute(L,[H|T]) :-
        append(V,[H|U],L) ,
            /* V,U stand for the part of L before and after H */
        append(V,U,W) ,
        permute(W,T) .
```

Logical vs Procedural Reading of Programs

- Can read Prolog program as specifying solution or computation.
- How does a prolog program compute an answer?

```
father(ralph,john).  father(ralph,mary).  father(bill,ralph).
mother(sue,john).  mother(sue,mary).  mother(joan,ralph).
parent(M,X) :- mother(M,X).
parent(F,X) :- father(F,X).
ancester(X,Y) :- parent(X,Y).
ancester(X,Y) :- parent(X,Z),ancester(Z,Y).

?-ancester(joan,X).
```

- Backtracking
  - First succeeds with `X = ralph`
  - Second `X= john`
  - Third `X = mary`

Operators in Prolog

- Usually used as prefix, can force to be in infix or postfix and give precedence as well.
- Example: arithmetic operators: `2+3*5` abbreviates `+(2,*(3,5))`
- Operations are not evaluated
- Better to think of operations as forming tags on values
  - Forming records
  - Don't really compute anything
  - Typically uninterpreted
- Relations `=`, `\=`, `<`, `>`, `=<`, `>=` (note order of composite relations) are evaluated.

```
  digit(N):- N>=0,N<10.
```

Example of Using Operations

- Trees
  - Tree is either nil or is of form `maketree(tree1,X,tree2)`.
  - Programs to manipulate trees use pattern matching like in ML
- Arithmetic
  - If actually wish to calculate, must use `is`.
  - Ex. `area(L,W,A) :- A is L*W.`
  - Can only compute `A` from `L`,`W` — not reversible.

Graphs

- Look at program to find paths in graph!

```
edge(a,b).
edge(a,f).
edge(b,c).
edge(c,a).
edge(d,e).
edge(e,a).
edge(e,c).

dumb_path(Start,Finish) :- edge(Start,Finish).
dumb_path(Start,Finish) :- edge(Start,Next),dumb_path(Next,Finish).
```

- What happens if type:

```
?- dumb_path(a,c).
?- dumb_path(a,e).
```

- Problem is continues to go through same vertex multiple times.

Graphs (cont)

- Smarter program keeps track of vertices which are visited.

```
path(Start,Finish) :- smart_path(Start,Finish,[]).

smart_path(Current,Target,Visited) :- edge(Current,Target).

smart_path(Current,Target,Visited) :-
        edge(Current,Next),non_member(Next,Visited),
        smart_path(Next,Target,[Next|Visited]).

non_member(Elt,[]).
non_member(Elt,[Hd | Tl]) :- Elt \== Hd, non_member(Elt,Tl).
```

Adjacency Lists

- Note that database representation of graph is not only possibility.
- Can use adjacency list representation.
- Write graph as list of vertices and corresponding edges:
  - Each vertex included with list of all edges going from it.
  - E.g. node a represented by

    ```
    v(a,[b,f])
    ```
  - Whole graph given by

    ```
    [v(a,[b,f]), v(b,[c]), v(c,[a]), v(d,[e]),v(e,[a,c])].
    ```

Adjacency Lists (cont)

- Advantage - can add vertices and edges during a computation.
- Write:
  - `vertex(Graph,Vertex)` which tells if `Vertex` is in `Graph`:

```
vertex([v(Vert,Edge) | Rest],Vert).
vertex([_ | Rest],Vert) :- vertex(Rest,Vert).
```

  - `edge(Graph,X,Y)` true if there is an edge from `X` to `Y`.

```
edge(Graph,X,Y) :- member(v(X,Edges),Graph),
                   member(Y,Edges).
edge(Graph,X,Y) :- member(v(Y,Edges),Graph),
                   member(X,Edges).
```

- Rest of program for paths is as before.

Russian Farmer Puzzle

- Variation of missionary and cannibals

  Farmer taking goat and (giant) cabbage to market. Wolf following farmer. Come to river with no bridge, but only tiny boat big enough to hold farmer and one object. How can farmer get all three across river without goat eating cabbage or wolf eating goat?

- Specify solution
  - At any time during solution, describe current state by noting which side of river each of farmer, goat, cabbage and wolf are on (call them north/south).
  - Can write down all states and all allowable transitions and then find path. (Like finding path in graph!)

Representation

- Really only 4 possible actions: Farmer crosses with one of goat, cabbage, and wolf, or farmer crosses alone.
- Write rules for each.
- Describe states by terms — `state(F,G,C,W)` where each of `F`, `G`, `C`, `W` is one of **`north`**, **`south`**.
- Need predicate “opposite”

```
opposite(north,south).
opposite(south,north).
```

Axiomatizing Actions

- Action 1: Farmer crosses with goat — need farmer and goat to start on same side, no danger of wolf eating cabbage.

```
transition(state(F0,F0,C,W),state(F1,F1,C,W)) :-
      opposite(F0,F1).
```

- Action 2: Farmer crosses with cabbage — need farmer and cabbage to start on same side, wolf and goat must be on opposite sides of river so goat is safe.

```
transition(state(F0,G,F0,W),state(F1,G,F1,W)) :-
       opposite(F0,F1), opposite(G,W).
```

Axiomatizing Actions

- Action 3: Farmer crosses with wolf — need farmer and wolf to start on same side, goat must be on opposite side from cabbage.

```
transition(state(F0,G,C,F0),state(F1,G,C,F1)) :-
      opposite(F0,F1), opposite(G,C).
```

- Action 4: Farmer crosses alone — need cabbage and wolf on same side, goat on opposite side.

```
transition(state(F0,G,C,C),state(F1,G,C,C)) :-
      opposite(F0,F1),opposite(G,C).
```

- Finding solution to problem is like finding path in graph.

Cut

- Cut — curtails backtracking.

$$\texttt{pred(...):-cond}_1, \ldots, \texttt{cond}_k, \texttt{!}, \texttt{cond}_{k+1}, \ldots, \texttt{cond}_n.$$

- Cut `!` is always satisfied — freezes all previous choices
- If get to point where no more solutions to $\texttt{cond}_{k+1}$`,...,` $\texttt{cond}_n$ then all of `pred(...)` fails and no other clause or rule for `pred` will hold.
- Backtracks until `!` satisfied and then never backtracks over it.

Uses of Cut

1. Only want one rule to be applicable, so keep from trying another.
2. Cut indicates have reached a point where cannot succeed.
3. Only want one solution — avoid generating others.

Cut Examples

- `sum_to(N,X)` should give $1 + 2 + ... + N$.

```
sum_to(1,1).
sum_to(N,Sum) :- Pred is N-1, sum_to(Pred,Partial_sum),
                 Sum is Partial_sum + N.
```

- First answer for `sum_to(1,X)` is sum
- Second attempt goes into infinite loop.
- Most likely happens as part of resatisfying something complex. e.g. `sum_to(1,X),foo(apples)`
- Fix by putting in `sum_to(1,1) :- !.`

Fail

- If can exclude some cases early on. Can use cut to terminate search.

```
Possible_Pres(willy) :- !,fail.
Possible_Pres(X) :- NativeBorn(X).
```

- Fail is predicate that is never satisfied.

Problems with Cut

- Sometimes get strange behavior if not careful.
- Suppose have following program:

```
likes(joe,pizza) :- !.
likes(jane,Anything).
```

- What happens if put in query: `?- like(Person,pizza).`
- Get answer of `joe`, but not `jane`. Yet `likes(jane,pizza)` is true!

Cut and Reversibility

- Using cut may hinder reversibility.
- Example:

```
append([],X,X) :- !.
append([A|B],C,[A|D]) :- append(B,C,D).
```

- Now only get one answer (when running either direction).

Formal Basis for Prolog

- Prolog based on resolution theorem proving!
- Understand program as either proof or set of procedure calls.
  - The fact `A(a,b,c)` is just treated as an atomic statement which is asserted to be true.
  - The rule `A(X,Y) :- B(X,Y),C(X,Y).` is understood as logical statement $\forall X, Y.(B(X,Y) \wedge C(X,Y) \rightarrow A(X,Y))$

Horn Clauses

- Formulas of the form $\forall X_1, \ldots, X_m.(B_1(X_1, \ldots, X_m) \wedge \ldots \wedge B_n(X_1, \ldots, X_m) \rightarrow A(X_1, \ldots, X_m))$ are said to be Horn clause formulas (notice fact is degenerate case where $n = 0$).
- Program is understood as a collection of Horn clause formulas.
- Query `?- D(X,Y)` is understood as $\exists X, Y.D(X, Y)$.

Resolution Theorem Proving

- Is there a proof of $\exists X, Y . D(X, Y)$ from statements in program?
- Resolution theorem proving works by attempting to show that hypotheses and negation of conclusion (i.e. $\forall X, Y . \sim D(X, Y)$) generate a contradiction.
- Contradiction is essentially found by finding values for X, Y such that $D(X, Y)$.

Prolog and Horn Clauses

- Prolog is a restriction of Horn clause logic.
- Clause contains at most one positive atomic formula.
- Prolog does not implement resolution correctly.
- Omits the "occurs check" during unification.
- Occurs check prevents matching a variable with a term that contains the same variable
- For example unifying `a(Z,Z)` with `a(X,successor(X))`
- Avoids infinite solutions to equations

```
X = successor(successor(successor(successor(successor(...
```

- Can lead to incorrect conclusions.

Negation in Prolog

- Negation based on assumption of complete knowledge: if system cannot prove statement is true, then it must be false.
- Three possible outcomes of proof: Succeeds, fails, doesn't terminate.
- Returns false only if it fails (finite failure).
- If attempt never terminates then don't report it false!.
- Note that this is a non-monotonic rule of reasoning.
- In family program from above, since there is no fact corresponding to `father(shezad,kim)`, system deduces that it is false.

Negation

- Built-in predicate `not` defined so that `not(X)` succeeds if an attempt to satisfy `X` fails.
- `not(X)` fails if an attempt to satisfy `X` succeeds.

  ```
  not(X) :- X,!,fail.
  not(X).
  ```

  Thus

  ```
  ?- not(father(shezad,kim)).
  ```

  reports true.
- However if add fact, `father(shezad,kim)`, then will reverse answers.

Not Strangeness

- Suppose define the following rule

  ```
  childless(X) :- not(father(Y,X)),not(mother(Z,X)).
  ```

  - If `fred` not mentioned `childless(fred)` will return yes
  - But `childless(Z)` returns no.
- Define the following rule but no rules for `young`

  ```
  old(X) :- not(young(X)).
  ```

- If write the other direction,

  ```
  young(X) :- not(old(X)).
  ```

  get the opposite response, everything is `young`!

`Not` Cautions

- `not` does not always behave properly — in particular, `not(not(term))` does not usually return same answers as `term`.
- Safest if only use `not` on predicates which have no free variables (i.e., make sure variables instantiated before calling it!).

Evaluation of Logic Programming and Prolog

- Keep in mind difference between Prolog and pure (Horn clause) logic programming
- Prolog has many faults
- Idea of logic programming may still be quite promising if can overcome deficiencies of Prolog. (For example, see uses of Datalog in knowledge-based database.)
- If when programming, can ignore issues of control and just worry about logic, then when works can worry about control to optimize without destroying correctness.
- Very high level — self documenting.
- Efficiency is problem with Prolog. (Can optimize with tail-recursion as in ML!)

Evaluation of Logic Programming and Prolog

- Effectiveness limited to two classes of programs:
  1. Where efficiency not a consideration.
  2. Where too complex for conventional language.
- Retains useful role as well for prototyping or as specification language.

Evaluation of Logic Programming and Prolog

- One of reasons Japanese chose Prolog is belief that it can be speeded up by use of highly parallel machines. OR-parallelism seems useful (work on different branches of search tree in parallel), but AND-parallelism (trying to satisfy various terms of right side of rule in parallel) seems more problematic.
  - One of few examples of non-procedural languages. (well, sort of)
  - Generalize by replacing unification by, e.g., solving systems of inequalities. Constraint logic programming.