Chapter 11 :: Logic Languages

Programming Language Pragmatics

Michael L. Scott

ELSEVIER

Logic Programming

- Based on predicate calculus
- Predicates - building-blocks $P(a_1,a_2,...,a_K)$
  - limit(f, infinity, 0)
  - enrolled(you, CS xxx)
  - These are interesting because we attach meaning to them, but within the logical system they are simply structural building blocks, with no meaning beyond that provided by explicitly-stated interrelationships

Logic Programming Concepts

- Operators
  - conjunction, disjunction, negation, implication
- Universal and existential quantifiers
- Statements
  - sometimes true, sometimes false, often unknown
  - axioms - assumed true
  - theorems - provably true
  - hypotheses (goals) - things we'd like to prove true

Logic Programming Concepts

- Example statements:

```
all f, l [
   limit(f, x0, l) <=>
        (all e [
             e > 0 => (exists d [
                  d > 0 and all x [
   (((|x-x0| < d) => (|f(x)-l|) < e)])])]

all f, g [f = O(g) <=
   (exist c, n0 [
        all n [
             n > n0 => f(n) < cg(n)]])]
```

Logic Programming Concepts

- Most statements can be written many ways
- That's great for people but a nuisance for computers
  - It turns out that if you make certain restrictions on the format of statements you can prove theorems mechanically
  - That's what logic programming systems do
  - Unfortunately, the restrictions that we will put on our statements will not allow us to handle most of the theorems you learned in math, but we will have a surprising amount of power left anyway

Logic Programming Concepts

- We insist that all statements be in the form of HORN CLAUSES consisting of a HEAD and a BODY
  - The head is a single term
  - The body is a list of terms
  - A term can be a constant, variable, or STRUCTURE consisting of a FUNCTOR and a parenthesized list of arguments

Logic Programming Concepts

- A structure can play the role of a data structure or a predicate
  - A constant is either an ATOM or a NUMBER
  - An atom is either what looks like an identifier beginning with a lower-case letter, or a quoted character string
  - A number looks like an integer or real from some more ordinary language
  - A variable looks like an identifier beginning with an upper-case letter
  - There are no declarations
  - All types are discovered implicitly

Logic Programming Concepts

- The meaning of the statement is that the conjunction of the terms in the body implies the head
  - A clause with an empty body is called a FACT
  - A clause with an empty head is a QUERY, or top-level GOAL
  - A clause with both sides is a RULE
- The Prolog interpreter has a collection of facts and rules in its DATABASE
  - Facts are axioms - things the interpreter assumes to be true

Prolog

- Prolog can be thought of declaratively or imperatively:
  - We'll emphasize the declarative semantics for now, because that's what makes logic programming interesting
  - We'll get into the imperative semantics later
- Prolog allows you to state a bunch of axioms
  - Then you pose a query (goal) and the system tries to find a series of inference steps (and assignments of values to variables) that allow it to prove your query starting from the axioms

Prolog

- The meaning of the statement is that

```
mother(mary, fred).
    % you can either think of this as  % an
predicate asserting that mary
    % is the mother of fred -
    % or a data structure (tree)
    % in which the functor (atom)
    % mother is the root,
    % mary is the left child, and
    % fred is the right child
fat(albert).
rainy(rochester).
```

Prolog

- Rules are theorems that allow the interpreter to infer things
- To be interesting, rules generally contain variables

```
employed(X) :- employs(Y,X).
```

can be read:

```
for all X, X is employed if there
exists a Y such that Y employs X
```

- Note the direction of the implication:
  - The example does NOT say that X is employed ONLY IF there is a Y that employs X

Prolog

- The scope of a variable is the clause in which it appears
  - Variables whose first appearance is on the left hand side of the clause have implicit universal quantifiers
  - Variables whose first appearance is in the body of the clause have implicit existential quantifiers
- Similarly:

Prolog

```
grandmother(A, C) :- mother(A, B),
                     mother(B, C).
```

can be read:

```
for all A, C [A is the grandmother of C
if there exists a B such that A is the
mother of B and B is the mother of C].
```

We probably want another rule that says

```
grandmother(A, C) :- mother(A, B),
                     father(B, C).
```

Prolog

- To run a Prolog program, one asks the interpreter a question
  - This is done by stating a theorem - asserting a predicate - which the interpreter tries to prove
    - If it can, it says *yes*
    - If it can't, it says *no*
    - If your predicate contained variables, the interpreter prints the values it had to give them to make the predicate true.

Prolog

- The interpreter works by what is called BACKWARD CHAINING
  - It begins with the thing it is trying to prove and works backwards looking for things that would imply it, until it gets to facts
- It is also possible in theory to work forward from the facts trying to see if any of the things you can prove from them are what you were looking for - that can be very time-consuming
  - Fancier logic languages use both kinds of chaining, with special smarts or hints from the user to bound the searches

Prolog

- The predicate you ask for is the interpreter's original GOAL
  - In an attempt to SATISFY that goal, it looks for facts or rules with which the goal can be UNIFIED
    - Unification is a process by which compatible statements are *merged*
  - Any variables that do not yet have values but which correspond to constants or to variables with values in the other clause get INSTANTIATED with that value
  - Anyplace where uninstantiated variables correspond, those variables are identified with each other, but remain without values

Prolog

- The interpreter starts at the beginning of your database (this ordering is part of Prolog, NOT of logic programming in general) and looks for something with which to unify the current goal
  - If it finds a fact, great; it succeeds
  - If it finds a rule, it attempts to satisfy the terms in the body of the rule depth first
  - This process is motivated by the RESOLUTION PRINCIPLE, due to Robinson:
    - It says that if C1 and C2 are Horn clauses, where C2 represents a true statement and the head of C2 unifies with one of the terms in the body of C1, then we can replace the term in C1 with the body of C2 to obtain another statement that is true if and only if C1 is true

Prolog

- When it attempts resolution, the Prolog interpreter pushes the current goal onto a stack, makes the first term in the body the current goal, and goes back to the beginning of the database and starts looking again
- If it gets through the first goal of a body successfully, the interpreter continues with the next one
- If it gets all the way through the body, the goal is satisfied and it backs up a level and proceeds

Prolog

- If it fails to satisfy the terms in the body of a rule, the interpreter undoes the unification of the left hand side (this includes uninstantiating any variables that were given values as a result of the unification) and keeps looking through the database for something else with which to unify (This process is called BACKTRACKING)
- If the interpreter gets to the end of database without succeeding, it backs out a level (that's how it might fail to satisfy something in a body) and continues from there

Prolog

- We can visualize backtracking search as a tree in which the top-level goal is the root and the leaves are facts (see Figure 11.2 - next slide)
  - The children of the root are all the rules and facts with which the goal can unify
  - The interpreter does an OR across them: one of them must succeed in order for goal to succeed
  - The children of a node in the second level of the tree are the terms in the body of the rule
  - The interpreter does an AND across these: all of them must succeed in order for parent to succeed
  - The overall search tree then consists of alternating AND and OR levels

Prolog

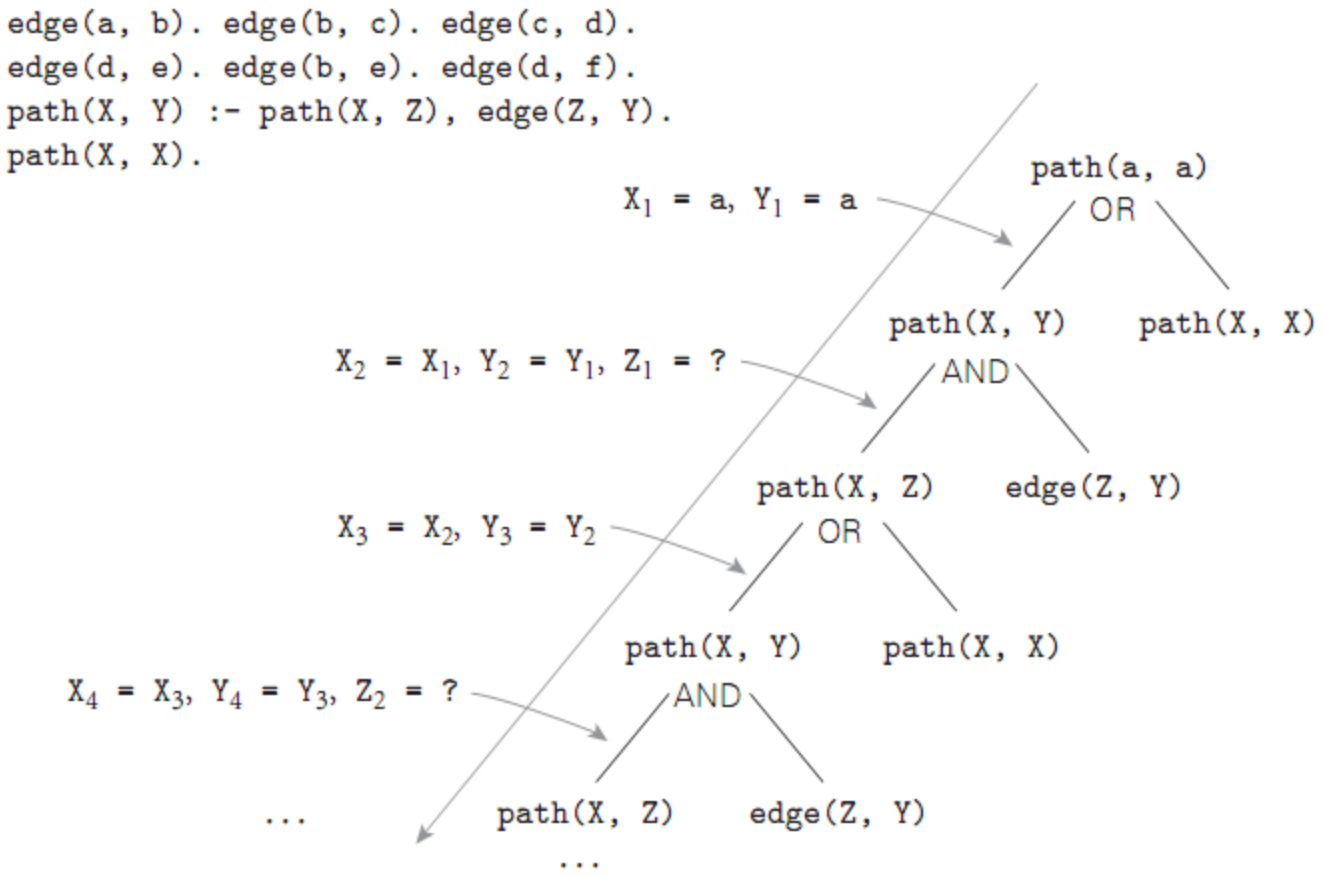

Figure 11.2 **Infinite regression in Prolog.** In this figure even a simple query like `?- path(a, a)` will never terminate: the interpreter will never find the trivial branch.

Prolog

- PROLOG IS ***NOT*** PURELY DECLARATIVE
  - The ordering of the database and the left-to-right pursuit of sub-goals gives a deterministic imperative semantics to searching and backtracking
  - Changing the order of statements in the database can give you different results
    - It can lead to infinite loops
    - It can certainly result in inefficiency
- **Tree relationships:**

Prolog

```
parent(a,b).      % a is the parent of b
parent(a,d).
parent(a,k).
parent(k,l).
parent(k,m).
parent(b,e).
parent(b,f).
parent(f,g).
parent(f,h).
parent(f,i).

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

Prolog

- Then the question

```
?- ancestor(U,h).
```

generates the answers

```
U = f;
U = b;
U = a;
no
```

- The question

```
?- ancestor(b,U).
```

generates all nodes in the subtree rooted in b

<< trace >>

Prolog

- If we change the order of the two ancestor rules, we get different execution orders:

```
?- ancestor(U,h).
U = a;
U = b;
U = f;
no
```

- If we change the order of the subgoals in the compound rule,

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

we run into an infinite loop (see also Figure 11.2)

Prolog

- **Arithmetic**: The '=' operator determines whether its operands can be unified

```
?- A = 37.
A = 37
yes

?- 2 = 2.
yes
```

Math operators are functors (structure names), not functions

```
?- (2+3) = 5
no
```

Prolog

- For math we use the built-in operator *is*

```
?- is(X, 1+2).
X = 3
yes

?- X is 1+2.
X = 3
yes
```

% LHS of 'is' must be as-yet uninstantiated

```
?- 1+2 is 4-1.
no
```

% RHS of 'is' must already be instantiated

```
?- X is Y.
<error>
```

Prolog

- **Tic-tac-toe** (see Figure 11.3 on next slide)
  - This program finds the next move, given a board configuration
  - It does not play a whole game (see the book for an extended version that does)
  - It depends on the ordering of rules
    - move(A) is the root rule
    - A is a result parameter
  - No winning strategy
    - each player can force a draw

Prolog

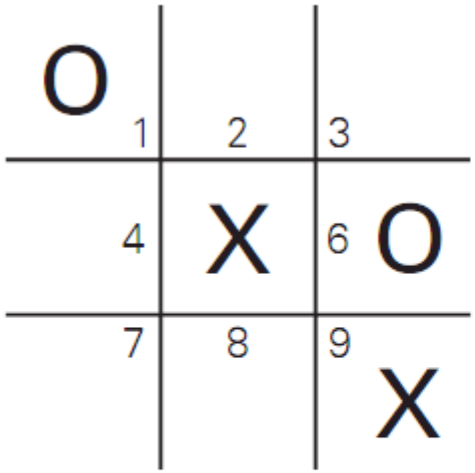

Figure 11.3 **A "split" in tac-tac-toe.** If X takes the bottom center square (square 8), no future move by O will be able to stop X from winning the game—O cannot block both the 2–5–8 line and the 7–8–9 line.

Logic Programming Examples

% express that three given squares lie in a line

```
ordered_line(1,2,3).   ordered_line(4,5,6).
ordered_line(7,8,9).   ordered_line(1,4,7).
ordered_line(2,5,8).   ordered_line(3,6,9).
ordered_line(1,5,9).   ordered_line(3,5,7).
line(A,B,C) :- ordered_line(A,B,C).
line(A,B,C) :- ordered_line(A,C,B).
line(A,B,C) :- ordered_line(B,A,C).
line(A,B,C) :- ordered_line(B,C,A).
line(A,B,C) :- ordered_line(C,A,B).
line(A,B,C) :- ordered_line(C,B,A)
```

% we assume a not so perfect opponent

Prolog

```
% the following rules work well
move(A) :- good(A), empty(A).

full(A) :- x(A).
full(A) :- o(A).
empty(A):- not full(A)

% strategy (key is ordering following five rules)
good(A) :- win(A).
good(A) :- block_win(A).
good(A) :- split(A).
good(A) :- block_split(A).
good(A) :- build(A).
```

Prolog

```
    % first choice is to win(1)
win(A) :- x(B), x(C), line(A,B,C).

    % block opponent from winning (2)
block_win(A) :- o(B), o(C), line(A,B,C).

    % opponent cannot block us from winning next
    % see Figure 11.6 before for this case(3)
split(A) :- x(B), x(C), different(B,C),
                    line(A,B,D), line(A,C,E),
                    empty(D), empty(E).
same(A,A).
different(A,B) :- not same(A,B).
```

Prolog

```
% prevent opponent from creating a split  (4)
block_split(A) :- o(B), o(C), different(B,C),
                  line(A,B,D), line(A,C,E),
                  empty(D), empty(E).


% pick a square toward three in a row  (5)
build(A) :- x(B), line(A,B,C), empty(C).


% if non of the five, final defaults (in this order)
good(5).
good(1).    good(3).    good(7).    good(9).
good(2).    good(4).    good(6).    good(8).
```