CS206 Lecture 16

Term Rewriting in Prolog

G. Sivakumar
Computer Science Department
IIT Bombay
siva@iitb.ac.in
http://www.cse.iitb.ac.in/∼siva

Thu, Feb 13, 2003

Plan for Lecture 17

- Extra-logical features of Prolog
  - Examining "Types" and Structures
  - Cuts (for controlling search)
  - Negation
- Naive Rewriting using Prolog

List to Set conversion

Suppose we wish to write a predicate `setform(L,S)` which means "S is the set of elements in L" (no duplicates).

Naive attempt

```
setform([],[]).
setform([A|R],Ans) :- setform(R,Ans1),
                      ins(A,Ans1,Ans).


ins(A,[],[A]).
ins(A,[B|R],Ans) :- ?????
```

How to complete the definition of "ins"?

We need to check for "inequality" or "non-membership" i.e. some **negative** property.

Prolog provides some "extra-logical" features to make life easy for programmer (although Prolog is Turing-complete without such features).

Examining Structures

Prolog has several built-in predicates that allow us to examine structures. Why?

- Symbolic computation (as opposed to numerical) is one of the areas where Prolog can be used.
- Check the calling pattern in the body of a rule and behave accordingly.

Var predicate

`var(X)`

Succeeds if `X` is currently uninstantiated (i.e. X is still a variable); otherwise it fails.

Examples:

```
| ?- var(X).
yes
| ?- var([X]).
no
| ?- var(f(Y,Z)).
no
| ?- var((X)).
yes
```

Opposite is `nonvar(X)`.

Typical Usage

If we want to invert `plus` using regular numbers (not 0,s notation).

```
plus(X,Y,Z) :- nonvar(X), nonvar(Y),
               Z is X + Y.
plus(X,Y,Z) :- nonvar(X), nonvar(Z),
               Y is Z - X.
plus(X,Y,Z) :- nonvar(Z), nonvar(Y),
               X is Z - Y.
```

We can call this `plus` with any two arguments instantiated and it will work! But, above not sufficient to call with all three unbound and generate answers. Or, only one variable bound.

Other Type Predicates

- integer(X) Succeeds if X is currently instantiated to an integer; otherwise it fails.
- real(X) Succeeds if X is currently instantiated to a floating point number; otherwise it fails.
- number(X) Succeeds if X is currently instantiated to either an integer or a floating point number (real); otherwise it fails.

Atomic structures

`atom(X)` checks if `X` is bound to a non-numeric constant.
`atomic(X)` checks if `X` is bound to a non-numeric constant or a number.
Examples:

```
| ?- atomic(10).
yes
| ?- atomic(p).
yes
| ?- atomic(h).
yes
| ?- atomic(h(X)).
no
| ?- atomic("foo").
no
| ?- atomic('foo').
yes
| ?- atomic(X).
no
| ?- atomic(X((Y))).
no
```

Compound Structures

`compound(X)` Succeeds if `X` is currently instantiated to a compound term (with arity greater that zero), i.e. to a nonvariable term that is not atomic; otherwise it fails. Examples:

```
| ?- compound(1).
no
| ?- compound(foo(1,2,3)).
yes
| ?- compound([foo, bar]).
yes
| ?- compound("foo").
yes
| ?- compound('foo').
no
| ?- compound(f(a,b)).
yes
```

Taking apart compound terms

`functor(Term, Functor, Arity)`

Succeeds if the *functor* of the Prolog term `Term` is `Functor` and the *arity* (number of arguments) of `Term` is `Arity`. Examples:

```
| ?- functor(p(f(a),b,t), F, A).
F = p
A = 3

| ?- functor(T, foo, 3).
T = foo(_595708,_595712,_595716)

| ?- functor(foo, F, 0).
F = foo

| ?- functor("foo", F, A).
F = .
A = 2
```

Picking out one argument

`arg(Index, Term, Argument)`

Unifies `Argument` with the `Index`th argument of `Term`, where the index is taken to start at 1. Initially, `Index` must be instantiated to any integer and `Term` to any non-variable Prolog term.

Examples:

```
| ?- arg(2, p(a,b), A).
A = b

| ?- arg(1, h(a,b), A).
A = a

| ?- arg(0, foo, A).
no

| ?- arg(2, [a,b,c], A).
A = [b,c]
```

Copying Terms

`copy(Term, Copy)`

Makes a `Copy` of `Term` in which all variables have been replaced by brand new variables which occur nowhere else. It can be very handy when writing (meta-)interpreters for logic-based languages.

Examples:

```
| ?- copy_term(X, Y).

X = _598948
Y = _598904

| ?- copy_term(f(a,X), Y).

X = _598892
Y = f(a,_599112)
```

Cuts and Negation

To control Backtracking

- Reorder rules
- Reorder clauses in the body of a rule

But is this enough? Consider some **mutually exclusive rules**
Suppose school uniforms are chosen as follows. If a child is in class 1-5, then blue. If a child is in class 6-9, then red. If a child is in class 10-11, then white.
We can code this as follows.

```
uniform(X,blue) :- class(X,Y),  Y < 6.
uniform(X,red) :- class(X,Y), Y > 5, Y < 10.
uniform(X,white) :- class(X,Y), Y > 9.
```

Unnecessary Search

Assume also

```
happy(Child) :- uniform(Child, Colour),
                likes(Child, Colour).


likes(ramesh,red).
class(ramesh,4).
```

Try the goal `happy(ramesh)`.
We get 2 subgoals,
`uniform(ramesh,Colour)`, `likes(ramesh, Colour)`
Prolog will try all three clauses for `uniform` and will fail 3 times.
Search tree is unnecessarily big.

Cuts to express determinism

Prolog allows programmer to improve such situations using cut denoted by "!"

We write

```
happy(Child) :- uniform(Child, Colour), !,
                likes(Child, Colour).
```

The meaning is that-

Once the clause uniform(Child,Colour) has given one answer, **do not try for any other answer.**

Cut the choice points left for uniform and proceed to next clause.

We can do this, because we know uniform colour has only one answer since clauses are mutually exclusive.

Green Cuts

A cut as used above is a GREEN cut (or safe cut).
Removing the cut will not cause any problems.
Only make Prolog search unnecessarily in parts of the tree where there is no answer. **Red Cuts**
Now that *cut* is provided, programmers may be tempted to use it as below.
Instead of writing

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

they may write

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

reasoning that if first clause succeeds it is the only answer and we will never backtrack to second. So, why do the test X $<$ Y?
Is this fine?

Red Cuts (ctd.)

As long as `max` is called with only first two arguments instantiated this is fine. We will get only correct answers.

But, suppose we call as below, `max(5,2,2)`

This goal will succeed in the version with *cut* (why?) but not in the original version.

Such cuts are called RED (or unsafe).

Should be used only when certain calling patterns are guaranteed.

```
ins(X,[],[X]).
ins(X,[B|R],[B|R]) :- member(X,[B|R]),!.
ins(X,[B|R],[X | [B | R]]).
```

is fine provided we always call ins with first two arguments bound.

Definition of Cut

- Parent Goal

  The goal that matched the head of the clause containing the cut.

- Solving a cut

  When cut is encountered as a goal, it succeeds immediately.

- Effect of cut

  System commits to all choices made between the time the parent goal was invoked and the time the cut was solved. That is, all remaning alternatives between the parent goal and the cut are discarded.

Cut (ctd.)

Consider the clause

```
H :- B1, B2, B3, !, B4, B5, B6.
```

If goal G matches H.
We solve B1, B2, B3 as usual (which gets solutions one at a time).
When we reach the !, we commit to this solution for B1,B2,B3 and will not try any other answers for B1, B2, B3.
Further, G itself becomes committed to this choice and we will not try to match G with the head of any other clause.

Illustrative example for cut

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
```

When solving A, we will try C after getting the first solution for B.

When solving C, when we get the first solution for P,Q,R, we commit to this and do not examine other choices for P,Q,R. We only return all the answers obtained from S,T,U. We also do not try `C :- V`.

Cut is invisible to A and all choices for B will be tried.

Negation

Cuts provdide a weak form of negation to Prolog.
Consider `not` defined as follows.

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

To check if a Goal cannot be satisfied, we try the Goal.
If we find one answer, we cut and fail.
If we find no answer (the search tree is finite and all branches fail) then not(Goal) succeeds.

Example usage of not

Consider `member` defined as follows.

```
member(X,[X|Y]).
member(X,[U|V]) :- member(X,V).
```

Using this, we can write a insertion predicate to a set (no duplicates).

```
ins(A,[],[A]) :- !.
ins(A,B,[A|B]) :- not(member(A,B)).
```

Trace `ins(2, [3,4,5], Ans)`.

Other Features

- Global variables/state

  `assert, retract`

- Collecting solutions together

  `setof, bagof, findall`

All these features to be used with care.
Makes **performance** better.
Strategy: First understand the **logic** and write a **correct** program.
Then, use "extra-logical" features carefully, if necessary, to improve performace.