Logic Programming:
Term manipulation, Meta-Programming

Alan Smaill

Oct 22, 2015
Today

- Reminder of term manipulation predicates
  - var/1, functor/2 etc
- Meta-programming
  - call/1
  - symbolic programming
  - Prolog in Prolog
Recall

- `var/1` holds if argument is Prolog variable, when called.
- `nonvar/1` holds if argument is not a variable when called (ground, or partially instantiated)

- None of these affect binding.
Other term properties

- number/1: -89, 1.007
- integer/1: -89, 6000
- float/1: 0.1, 67.6543
- atom/1: a,f,g1567
- atomic/1: a,f,g1567,1.007,-89
Structural equality

Can test for whether terms are identical:

- The \(==/2\) operator tests whether two terms are exactly identical,
- Including variable names! (No unification!)

?- X == X.
  yes
?- X == Y.
  no
Structural comparison

\( \neq/2 \): tests that two terms are not identical

?- X \( \neq \) X.
no
?- X \( \neq \) Y.
yes
Recall: breadth-first search

- Keep track of all possible solutions, try shortest ones first
- Maintain a “queue” of solutions

```
bfs([[Node|Path],_], [Node|Path]) :-
    goal(Node).
bfs([Path|Paths], S) :-
    extend(Path,NewPaths),
    append(Paths,NewPaths,Paths1),
    bfs(Paths1,S).
bfs_start(N,P) :- bfs([[N]],P).
```
A difference list version

Here is a more efficient way, using difference lists — the first two arguments to `bfs_dl/2` are thus a (difference) list of lists, and the associated difference list variable.

```
bfs_dl([[Node|Path]|_], _, [Node|Path]) :-
goal(Node).
bfs_dl([Path|Paths], Z, Solution) :-
  extend(Path,NewPaths),
  append(NewPaths,Z1,Z),
  Paths \== Z1, \%
  (Paths,Z1) is not empty DL
  bfs_dl(Paths,Z1,Solution).
```

```
bfs_dl_start(N,P) :- bfs_dl([[N]|X],X,P).
\== checks if terms `Paths,Z1` are identical as terms
```
Why more efficient?

For the \( \setminus = /2 \) test, recall that the empty difference list is represented as a pair \( X/X \) with two occurrences of the same variable.

Notice that, although the new version uses the usual append/3, its first argument is the list of new paths, not the list of current paths, which is usually much larger.
meta-Programming: `call/1`

`call/1`:  
- Given a Prolog term \( G \), solve it as a goal

```prolog
?- call(append([1],[2],X)).
X = [1,2].
```

```prolog
?- read(X), call(X).
|: member(Y,[1,2]).
X = member(1,[1,2])
```
...allows some devious things.

callwith(P,Args) :-
    Atom =.. [P|Args], call(Atom).

map(P,[],[]).
map(P,[X|Xs],[Y|Ys]) :-
    callwith(P,[X,Y]), map(P,Xs,Ys)

plusone(N,M) :- M is N+1.

?- map(plusone,[1,2,3,4,5],L).
L = [2,3,4,5,6].
Symbolic programming

Propositions

prop(true).
prop(false).
prop(and(P,Q)) :- prop(P), prop(Q).
prop(or(P,Q)) :- prop(P), prop(Q).
prop(imp(P,Q)) :- prop(P), prop(Q).
prop(not(P)) :- prop(P).
simp(and(true,P),P).
simp(or(false,P),P).
simp(imp(P,false), not(P)).
simp(imp(true,P), P).
simp(and(P,Q), and(P1,Q)) :- simp(P,P1).
simp(and(P,Q), and(P,Q1)) :- simp(Q,Q1).
...
Satisfiability checking

- Given a formula, find a satisfying assignment for the atoms in it;
- Assume atoms given \([p_1, \ldots, p_n]\).
- A valuation is a list \([(p_1, \text{true} | \text{false}), \ldots]\).

\begin{verbatim}
\text{gen([], []).}
\text{gen([P|Ps], [(P,V)|PVs]) :-}
  (V=true; V=false),
  \text{gen(Ps,PVs).}
\end{verbatim}
Evaluation

sat(V,true).
sat(V,and(P,Q)) :- sat(V,P), sat(V,Q).
sat(V,or(P,Q)) :- sat(V,P) ; sat(V,Q).
sat(V,imp(P,Q)) :- \+(sat(V,P)) ; sat(V,Q).
sat(V,not(P)) :- \+(sat(V,P)).
sat(V,P) :- atom(P),
member((P,true),V).
Satisfiability

- Generate a valuation
- Test whether it satisfies Q

\[
satisfy(Ps, Q, V) :- \text{gen}(Ps, V), \text{sat}(V, Q).
\]

- On failure, this backtracks & tries another valuation.

This exploits logic programming search in a useful and concise way.
Represent definite clauses

```
rule(Head,[Body,.....,Body]).
```

A Prolog interpreter in Prolog:

```
prolog(Goal) :- rule(Goal,BODY),
              prologs(BODY)
prologs([]).
prologs([Goal|Goals]) :- prolog(Goal),
                        prologs(Goals).
```
Example

```
rule(p(X,Y), [q(X), r(Y)]).
rule(q(1),[]).
rule(r(2),[]).
rule(r(3),[]).

?- prolog(p(X,Y)).
X = 1
Y = 2
```
So what?

- Prolog interpreter already runs programs...
- Self-interpretation is interesting because we can examine or modify behaviour of interpreter.
rules with “justifications”!

```
rule_pf(p(1,2), [], rule1).
rule_pf(p(X,Y), [q(X), r(Y)], rule2(X,Y)).
rule_pf(q(1), [], rule3).
rule_pf(r(2), [], rule4).
rule_pf(r(3), [], rule5).
```
Witnesses

Now we can produce proof trees showing which rules were used:

\[
\text{prolog\_pf}(\text{Goal},[\text{Tag}|\text{Proof}]) :- \\
\quad \text{rule\_pf}(\text{Goal},\text{Body},\text{Tag}), \\
\quad \text{prologs\_pf}(\text{Body},\text{Proof}).
\]

\[
\text{prologs\_pf}([],[]).
\]

\[
\text{prologs\_pf}([\text{Goal}|\text{Goals}],[\text{Proof}|\text{Proofs}]) :- \\
\quad \text{prolog\_pf}(\text{Goal},\text{Proof}), \\
\quad \text{prologs\_pf}(\text{Goals},\text{Proofs}).
\]
Witnesses

"Is there a proof of p(1,2) that doesn't use rule 1?"

?- prolog_pf(p(1,2),Prf),
\+(in_proof(rule1,Prf)).

Prf = [rule2,[rule3, rule4]].
Other applications

Iterative deepening interpreter:
  as we saw for general search, we can:
  – search exhaustively to a given depth;
  – if no solution found, increase depth bound and recurse.

This way, we are assured to find a solution if there is one.
Iterative deepening meta-interpreter

Prolog implementations allow inspections of the internal knowledge base of facts and rules.

To make use of this, need to make relevant predicates “dynamic”, e.g. by having a directive:

```prolog
:- dynamic(foo/2).
```

```prolog
foo(a,1).
foo(b,Y) :- foo(a,X), Y = X + 1.
```

The `clause/2` predicate then allows us to inspect clauses matching a given head pattern:

- returns an explicit `true` for an empty body (head is a fact).
- returns body as atom, or as compound term made up of pairs `(X,Y)`. 
We can query for clause information:

| ?- clause(foo(X,Y),Body).
X = a,
Y = 1,
Body = true ? ;
X = b,
Body = (foo(a,_A),Y=_A+1) ;
no

We can even query with more instantiated pattern:

| ?- clause(foo(c,Y), Body).
no
| ?- clause(foo(a,Y), Body).
Y = 1,
Body = true ?
We can give depth-bounded search for goals tagged with a depth bound, as follows:

```prolog
solve( true/_ ) :- !. %% base case
solve( (A,B) ) :- solve(A), solve(B). %% pair of goals
solve( Q/N ) :- 0<N, M is N-1,
%% unify with head of a clause;
%% note change of depth.
    clause( Q, Body ),
    tag(Body,M,Tagged),
    solve( Tagged ).
```

%%% tag(+,+,:) distributes depth label to subgoals
Now use iterative deepening wrapper:

\[
\text{idsolve(Query)} :\text{idsolve(Query,0)}. \\
\text{idsolve(Query,N)} :\text{idsolve(Query,N)} \\
\text{tag(Query,N,QQ), solve(QQ),} \\
\text{write('Solution found during search to depth ')} \\
\text{write(N).} \\
\text{idsolve(Query,N)} :\text{idsolve(Query,N)} \\
\text{M is N+1, write('Searching at depth ')} \\
\text{write(M),nl,} \\
\text{idsolve(Query,M).}
\]
Now look at cases where depth-first execution may be problematic, and compare `?- Query.` with `?- idsolve(Query).

- Where looping occurs, so losing solutions (incompleteness): iterative deepening search will find solutions (given enough resources).

- Where solutions with short derivations are found only after solutions with longer derivations: iterative deepening will find the former before the latter.

**BUT** iterative deepening itself will loop if there is no solution!
More applications

- Tracing
  - Can implement trace/1 this way
- Declarative debugging
  - Given an error in output, “zoom in” on input rules that were used
  - These are likely to be the ones with problems

For more on this, see LPN, ch. 9, and Bratko, ch. 23