Logic Programming: Term manipulation, Meta-Programming

Alan Smaill

Nov 21, 2013

Today

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

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

- \==/2: tests that two terms are not identical

```
?- X \== X.
no
?- X \== 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 `\==/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.

`call/1`:

- Given a Prolog term `G`, solve it as a goal

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

```
?- read(X), call(X).
|: member(Y,[1,2]).
X = member(1,[1,2])
```

call with =..

...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].
```

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).
```

Formula Simplification

```
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).
...
```

Satisfiability checking

- Given a formula, find a satisfying assignment for the atoms in it;
- Assume atoms given `[p1,...,pn]`.
- A valuation is a list `[(p1,true|false),...]`.

```
gen([],[]).
gen([P|Ps], [(P,V)|PVs]) :-
         (V=true;V=false),
          gen(Ps,PVs).
```

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) :- gen(Ps,V),
                   sat(V,Q).
```

- (On failure, backtrack & try another valuation)

- 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.

```
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).
```

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

```
prolog_pf(Goal,[Tag|Proof]) :-
      rule_pf(Goal,Body,Tag),
      prologs_pf(Body,Proof).
prologs_pf([],[]).
prologs_pf([Goal|Goals],[Proof|Proofs]) :-
      prolog_pf(Goal,Proof),
      prologs_pf(Goals,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.
- 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

- Material covered in LPN, ch. 1-6:
- Terms, variables, unification (+/- occurs check)
- Arithmetic expressions/evaluation
- Recursion, avoiding non-termination
- Programming with lists and terms
- Expect ability to solve problems similar to those in tutorial programming exercises (or textbook exercises)

Review

- Material covered in LPN, ch. 7-11:
- Definite clause grammars
- Difference lists
- Non-logical features ("is", cut, negation, assert/retract)
- Collecting solutions (findall, bagof, setof)
- Term manipulation (var, =.., functor, arg, call)
- Expect ability to explain concepts & use in simple Prolog programs

Review

- Advanced topics (Bratko ch. 11-12, 14, 23)
- Search techniques (DFS, IDS, BFS)
- Symbolic programming & meta-programming
- Expect understanding of basic ideas
- not ability to write large programs from scratch under time pressure
- **Not** higher-order programs (may appear in theory exam, though)

Some exam info

- Programming exam: 2 hours
- DICE machine with SICSTUS Prolog available
- (Documentation won't be, but exam will not rely on memorizing obscure details)
- Sample exam on course web page
- Some exams are on ITO web page; questions similar but different format.

Other resources

There is a lot more to logic programming!

- Books: "The Art of Prolog", Sterling & Shapiro, MIT Press
- Association for Logic Programming
- Journals: Theory and Practice of Logic Programming; main journal before 2001 was Journal of Logic Programming
- Main conferences:
    - International Conference on Logic Programming (ICLP) - main annual conference.
    - Principles and Practice of Declarative Programming (PPDP) - covers LP and other "declarative" paradigms

Honours/MSc projects?? Let me know.