Logic Programming

Lecture 8: Term manipulation & Meta-programming

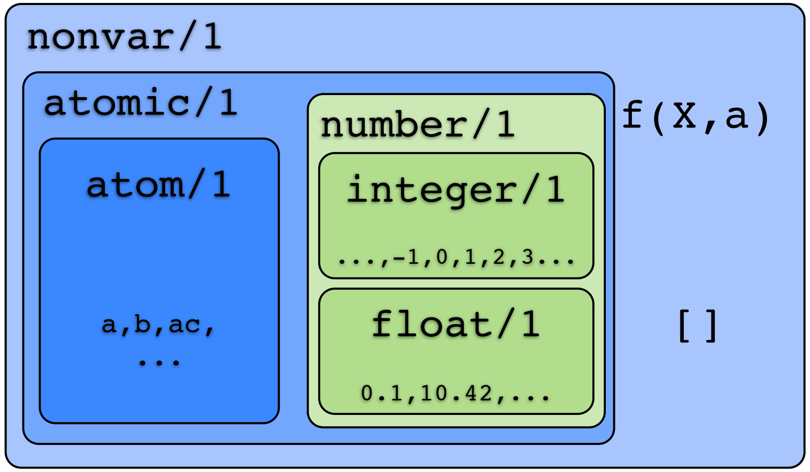
Outline for today

- Special predicates for examining & manipulating terms
  - `var, nonvar, functor, arg, =..`
- Meta-programming
  - `call/1`
  - Symbolic programming
  - Prolog self-interpretation

`var/1` and `nonvar/1`

- `var(X)` holds
- `var(a)` does not hold
- `nonvar(X)` does not hold
- `nonvar(a)` holds
  - (tutorial questions...)
- None of these affects binding

Other term structure predicates



Structural equality

- 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
```


From last week: 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 implementation

```
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).
```

A difference list implementation

```
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).
```

\== means "not equal as term"

functor/3

- Takes a term T, gives back the **atom** and **arity**

```
?- functor(a,F,N).
F = a
N = 0
?- functor(f(a,b),F,N).
F = f
N = 2
```


arg/3

- Given a number N and a complex term T
- Produce the Nth argument of T

```
?- arg(1,f(a,b),X).
X = a
?- arg(2,f(a,b),X).
X = b
```


=../2 ("univ")

- We can convert between terms and "universal" representation

```
?- f(a,f(b,c)) =.. X.
X = [f,a,f(b,c)]
?- F =.. [g,a,f(b,c)].
F = g(a,f(b,c))
```

Variables as goals

- Suppose we do

  `X = append([1],[2],Y), X.`

- What happens? (In Sicstus at least...)

  `Y = [1,2]`

- Clearly, this behavior is nonlogical
  - i.e. this does not work:

    `X, X = append([1],[2],Y).`

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])
```

- A variable goal X is implicitly treated as call(X)

Call + =..

- Can do some evil 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

- Logical formulas

```
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)

Prolog in Prolog

- 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
```


OK, but so what?

- Prolog interpreter already runs programs...
- Self-interpretation is interesting because we can **examine** or **modify** behavior 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

- 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),
   \+(member(rule1,Prf)).
Prf = [rule2,[rule3, rule4]].
```


Other 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

- Further reading:
  - LPN, ch. 9
  - Bratko, ch. 23
- Next time
  - Constraint logic programming
  - Course review