5.6 Meta-Programming

Freitag, 19. Juni 2015 08:30

Prolog can manipulate terms (Sect 5.6.1) and programs (Sect 5.6.2). In particular, a program can manipulate itself while it is running.

5.6.1. Manipulation of terms and formulas

pre-defined predicates to manipulate/access/recognize certain forms of terms:

- number/1 : checks whether arg. is a number
- var/1 : var(t) is true iff t is an (uninstantiated) variable

```
?- var(X).
true
?- X=2, var(X).
false
```

- nonvar/1 : nonvar(t) is true iff t is no variable

```
?- nonvar(a).
true
?- X=2, nonvar(X).
X=2
?- nonvar(X).
```

— although there is an instantia-

?- nonvar(X).
false

— although there is an instantiation of X such that nonvar is true

- atomic/1: atomic(t) is true iff t is a fct/pred symbol of arity 0 or a number

?- atomic(a).
true

?- atomic(-2).
true

?- atomic(a(a)).
false

?-atomic(X).
false

- compound/1: compound(t) is true iff t is a term/formula which does not just consist of a symbol of arity 0 or a number or a variable

?- compound(a).
false

?- compound(X).
false

?- compound(1+2).
true

?- compound(a(a)).
true

These predicates can be used to recognize certain forms of terms. But we also want to extract certain parts of terms (decomposition) and to construct new terms.

Solution: transform terms to lists or vice versa

e.g. $f(a,b)$ can be transformed to

$[f, a, b]$

f/2

f/0

Pre-defined predicate: $=..$ / 2

(infix notation)

$t =.. l$ is true iff

l is the list-representation of the term t

?- f(a,b) =.. L.

L = [f,a,b]

+(1,2)

?- 1+2 =.. L.

L = [+, 1, 2]

?- f(g(a), b) =.. L.

L = [f, g(a), b]

?- T =.. [f, a, b].

T = f(a, b)

?- T =.. [f].

T = f.

?- X =.. Y.

error

?- X =.. [Y, a, b].

error

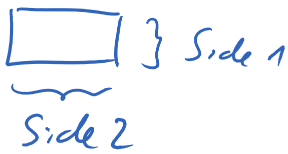
?- X =.. [f | L].

error

error if the leading fct. symbol and its arity are indetermined

Example for using =.. : Represent and enlarge different geometrical figures

square(Side) ← term to represent 

rectangle(Side₁, Side₂) ← 

triangle(Side₁, Side₂, Side₃)

circle(Radius)

Prog: enlarge(square(Side), Factor, square(NSide)) :- NSide is Factor * Side.

enlarge(rect(S_1, S_2), F, rect(NS_1, NS_2)) :- NS_1 is F * S_1, NS_2 is F * S_2.

```
enlarge(triangle(S1,S2,S3), ... ) :- ...
enlarge(circle(R), ... ) :- ...
```

Disadvantage: new enlarge-clause for each geometrical figure, although all these clauses essentially do the same.

Better solution:

```
enlarge(Fig, Factor, NFig) :- Fig =.. [Type | Param],
                              multlist(Param, Factor, NParam),
                              NFig =.. [Type | NParam].

multlist([], _, []).
multlist([X|L], Factor, [NX|NL]) :- NX is Factor * X,
                                    multlist(L, Factor, NL).
```

There are additional predicates to access/manipulate parts of terms:

- functor/3 :   functor(t, f, n) is true iff f/n is the leading fct/pred symbol of t

```
?- functor(g(f(X), X, g), F, N).
F = g, N = 3

?- functor(T, g, 3).
```

```
T = g(X, Y, Z).
```

- arg/3: arg(n, t, a) is true iff a is the n-th argument of t

```
?- arg(3, g(f(X), X, g), A).
A = g

?- functor(D, date, 3),
   arg(1, D, 19),
   arg(2, D, 6),
   arg(3, D, 2015).

D = date(19, 6, 2015).
```

Ex: Predicate to check whether a term is variable-free:

```
ground(T) :- nonvar(T),
             T =.. [Functor | Argumentlist],
             groundlist(Argumentlist).

groundlist([ ]).
groundlist([T | Ts]) :- ground(T), groundlist(Ts).
```

5.6.2 Manipulation of Programs

Prolog-prog $\hat{=}$ data base of clauses which can be read and modified

?- clause (t_1, t_2).

is true iff there is a program clause

B :- $C_1, \ldots, C_k$ ← $k=0$ is possible, needed for facts: B :- true.

such that

clause (t_1, t_2) unifies with

clause $(B, (C_1, \ldots, C_k))$.

Ex: times(_, 0, 0).

times(X, Y, Z) :- Y > 0, Y1 is Y-1, times(X, Y1, Z1), Z is Z1 + X.

?- clause(times(X, Y, Z), Body).

Y = 0, Z = 0, Body = true ;

Body = (Y > 0, Y1 is Y-1, ..., Z is Z1 + X).

While "clause" can be used to read the code of the running program, there also exist predicates that can modify the text of the running program:

assert/1 and retract/1

?- assert(t).

Proof always succeeds, but as a side-effect, the clause t is added at the <u>end</u> of the program.

(The predicate asserta(t) adds the clause t at the <u>beginning</u> of the program. The pred. assertz is like assert.)

<u>Ex</u>:

```
?- assert(p(0)).
true
?- p(X).
X = 0
?- clause(p(X), B).
X = 0, B = true

?- assert(square(X,Y) :- times(X,X,Y)).
true
```

clauses built with "clause" cannot be asserted. "clause" is <u>static</u>

Predicates can be static or dynamic.
By default, all predicates in the prog. are static.

Clauses for static predicates cannot be added or removed by assert + retract.

Predicates introduced by "assert" are dynamic.
Moreover, predicates in the program can be declared to be dynamic by a corresponding directive:

Ex:
```
:- dynamic times/3.
times(_, 0, 0).
times(X, Y, Z) :- Y>0, ... .
```

```
?- times(2,3,Z).
Z=6
?- asserta(times(X,1,X)).
true
?- clause(times(X,Y,Z), B).
X=Z, Y=1, B=true
```

?- retract(t)

prof succeeds iff there is a prog. clause that unifies with t. As a side effect, this prog. clause is removed.

Ex:
```
?- retract(times(X,Y,X) :- Body).
Y=1, Body=true;
```
← removes times(X,1,X)

X = 0, Y = 0, Body = true ; ← removes times(_,0,0)
Body = ... ← removes times(X,Y,Z) :- Y>0, ...

assert + retract can lead to completely non-understandable programs ⇒ use them only for certain purposes.

Sensible use of assert + retract: compute results and store them for later use.

Ex: Store results of computations in a table to re-use these results later on and avoid their repeated re-computation.

*	0	1	...	9
0	0	0		0
1	0	1		9
⋮	0	...	...	
9	0	9		81

```
member(X, [X|L]).
member(X, [_|L]) :-
        member(X, L).
```

```
maketable :- L = [0,1,2,3,4,5,6,7,8,9],
             member(X,L),
             member(Y,L),
             Z is X*Y,
             assert(mult(X,Y,Z)),
             fail.          ← enforces backtracking
```

⤷ X and Y will range over all numbers from 0,...,9

⤷ 100 new facts are added to the program.

```
?- maketable.
false
?- mult(X,Y,8).
X=1, Y=8 ;
X=2, Y=4 ;
X=4, Y=2 ;
X=8, Y=1.
```

There exists a pre-defined predicate

findall/3

which finds <u>all</u> solutions to a query (i.e., without needing the user to press ;).

findall(t, g, l) is true iff the following holds:

- Prolog tries to prove the query g and builds up the full SLD tree.
- Then it collects all answer substitutions

$\sigma_1, \ldots, \sigma_n$ (in left-to-right depth-first search).

- Then findall(t, g, l) is true iff l is the list $[\sigma_1(t), \sigma_2(t), \ldots, \sigma_n(t)]$.

Ex: family program including the rule

fatherOf(F,C) :- married(F,W), motherOf(W,C).

?- findall(C , fatherOf(gerd,C), L).

$$L = [\underbrace{\text{susanne}}_{\sigma_1(C)}, \underbrace{\text{peter}}_{\sigma_2(C)}]$$

?- findall (fatherOf(gerd,C)), fatherOf(gerd, C), L).

$$L = [\underbrace{\text{fatherOf(gerd, susanne)}}_{\sigma_1(\text{fatherOf(gerd,C)})}, \underbrace{\text{fatherOf(gerd, peter)}}_{\sigma_2(\text{fatherOf(gerd,C)})}]$$

findall could be programmed ourselves using assert and retract:

```
findall( X , Query, Xlist) :- Query,
                               assert(answer(X)),
```

```
collectAnswers([X|Rest]) :- retract(answer(X)),
                            !,
                            collectAnswers(Rest).
collectAnswers([]).
```

(Upper part of the clause as written above it:)

```
fail
;
collectAnswers(Xlist).
```

Since Prolog-programs can also be regarded as terms, one can use Prolog to write

meta-programs (programs that operate on programs, e.g., compilers and interpreters)

and

meta-interpreters (interpreter for a prog. language that is written in this prog. language).

In particular, one can also easily write interpreters for variants of Prolog.

Simplest meta-interpreter (: Meta-Interpreter 0)

```
prove(Goal) :- Goal.
```

If prog. contains p(0)

```
?- prove(p(X)).
X=0
```

Meta-Interpreter 1 (for pure logic programs)

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Variant of this meta-interpreter where composed goals are handled from right to left:

Meta-Interpreter 2

Variant of meta-interpreter 1 which also returns the length of the proof:

Meta-Interpreter 3

```
?- prove(fatherOf(gerd, C), N).
C = Susanne, N = 3
```