5.6.2 Meta-Programming: Manipulating Programs

Montag, 3. Juli 2017 09:11

A Prolog program is a data base of clauses. It can read its own code during runtime using the pre-defined predicate clause/2.

Here, clause(t_1, t_2) is true iff

there is a prog. clause B :- $C_1, \ldots, C_k$ such that clause(t_1, t_2) unifies with

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

Ex: Prog contains 2 clauses:

```
times(_, 0, 0).          ← is regarded as  times(_, 0, 0) :- true.
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, times(X, Y1, Z1), Z is Z1 + X)
```

↑ stands for

```
Y>0, (Y1 is ... , (times(X ...), Z is ...))
```

Up to now, we considered the data base of prog. clauses to be static. But it is possible to

modify a Prolog-prog. during its execution
(⇒ dynamic).

assert/1: proving assert(t) always succeeds and as a side-effect, the clause t is added at the end of the program.

Ex: times-prog.

```
?- assert(p(0)).        ← fact p(0). is added at the end of the prog.
true

?- p(X).

X = 0.

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

?- square(3,Y).
Y = 9.
```

There are also predicates asserta/1 and assertz/1.

asserta/1: adds clause at the beginning of the prog

assertz/1: ... at the end of the prog. (like assert)

One can only modify prog. clauses for dynamic predicates: All predicates introduced by assert are dynamic. But the pred. in the program are static unless they are explicitly declared as dynamic.

To this end, one needs a corresponding directive in the program:

```
:- dynamic times/3.
times(_, 0, 0).
times(X, Y, Z) :- Y > 0, Y1 is Y-1, times(X, Y1, Z1),
                  Z is Z1 + X.
```

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

retract/1 is used to remove clauses (of dynamic predicates)

retract(t) succeeds iff a prog. clause unifies with t. As a side effect, the first prog. clause that unifies with t is removed.

Ex: times

?- retract(times(X,Y,Z) :- B).

first clause for times is deleted.

By pressing ; several times, one can delete all clauses for times.

Instead:

?- retract(times(X,Y,Z)).

would only delete facts of times

Prog clauses can also contain assert and retract → prog. can modify itself while it is running. ⇒ Can lead to non-understandable programs (use assert + retract with care).

A useful application of assert + retract is to save intermediate results for later use.

Ex: We want to save intermediate results of multiplication.

⇒ Create table which stores X * Y for all numbers X and Y between 0 and 9.

```
maketable :- L = [0,1,2,3,4,5,6,7,8,9],
             member(X,L),
             member(Y,L),
```

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

```
member(Y, L),
Z is X*Y,
assert(times(X, Y, Z)),
fail.
```

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

↑ creates 100 times-facts due to backtracking

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

Ex: Predicate findall should not only find the first solution to a query (and wait for the user to press ";") but it should find <u>all</u> solutions to a query. This predicate is predefined in Prolog, but we could define it ourselves with assert and retract.

findall(t, g, l) is true iff the following holds:
One builds up the complete SLD tree for the query g and computes all answer substitutions $\sigma_1, \dots, \sigma_n$.
Then findall(t, g, l) is true iff l is the list $[\sigma_1(t), \dots, \sigma_n(t)]$.

E.g.: Consider the family-example.

```
?- findall( Y , fatherOf(gerd, Y), L).
L = [susanne, peter].

?- findall(fatherOf(gerd, Y), fatherOf(gerd, Y), L).
```

L = [fatherOf(gerd, susanne), fatherOf(gerd, peter)].

Now we implement findall ourselves:

```
findall(X, Query, Xlist) :- Query,                 ← first solve the query
                                                     and compute answer subst σ
                                                     for X
                            assert(answer(X)),      ← assert the
                                                     fact answer(σ(X))
ensures that we                                      where 'answer'
build up the whole  →       fail                     is a fresh
SLD tree for Query                                   pred. symbol
                            ;
                            collect_answers(Xlist).

collect_answers([X|Rest]) :- retract(answer(X)),   ← instantiates X
                             !                        with the first
                             ;                        found solution
                             collect_answers(Rest).   and removes
                                                      the first answer
                                                      clause
collect_answers([]).
```

Another application of meta-programming is the use of Prolog to write interpreters for programming languages. In particular, one can implement interpreters for Prolog (or variants of Prolog) in Prolog easily.

- Simple Meta-Interpreter for Prolog:

```
prove(Goal) :- Goal.     ← useless interpreter which
                           delegates all work to the
                           original Prolog interpreter
```

- More sophisticated interpreter for pure logic programs. This will serve as the basis for other interpreters for variants of logic programming.

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

Slide 39

[uses built-in unification of Prolog, but implements treatment of clauses itself

- Now one can modify this interpreter to experiment with alternative semantics of logic programming. We now implement an interpreter which handles sequences of literals from right to left:

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

- Another possibility is a meta-interpreter which also computes the length of proofs (i.e., the number of needed resolution steps):

```
prove(true, 0) :- !.
prove((Goal1, Goal2), N) :- !, prove(Goal1, N1), prove(Goal2, N2),
                             N is N1 + N2.
prove(Goal, N) :- clause(Goal, Body), prove(Body, N1),
                  N is N1 + 1.
```