Meta-programming in Prolog

Prof. Geraint A. Wiggins
Centre for Cognition, Computation and Culture
Goldsmiths College, University of London

Contents

- The idea behind meta-programming
- Representations
- Some examples

The idea behind meta-programming

- Meta-programming is one of the underpinning ideas of Logic Programming

- Meta-programming allows us
  - to manipulate the programs we write to improve their behaviour
  - to simulate execution and develop formal theories about programs (*eg* debugging)

- Meta-programming is easy and elegant in Prolog because the term syntax is identical with the clause syntax. . .

  . . . so we can use terms (data) to *name* clauses (programs)

Representations

- In order to build a program which takes as data another program, we have to decide on a representation

- We have already said that a term in Prolog is distinct from a literal with the same functor

- So at any time, we can tell whether something is a program or a piece of data which *names* a program, from the context in which we find it

- This distinction, between program syntax and the name of a program is the distinction between *object*- and *meta-levels*, respectively

Representations (2)

- Think back to the unification problem in exercise 2

- There, we simulated Prolog unification, but using a representation which made things easier:

  Variables were named by a Prolog structure, `+variable`, where `variable` was an atom

  Terms were constructed like `functor-Args`, where `functor` was an atom and `Args` was a (Prolog) list of terms

- So, for example, we represented the Prolog term

```
elephant( big, Forgot )
```

  as the term name

```
elephant-[big-[],+forgot]
```

Representations (3)

- In that example, the idea was to find a representation which could be accessed by unification alone, to avoid the use of real meta-predicates

- In real meta-programming, we don't want to have to mess about with special representations. . .

  . . . rather, we want simply to let a term represent or name itself

- There are two problems with this:
  1. We cannot manipulate the terms by unification alone
  2. We have to use Prolog variables to represent variables, but, from within the program, we can never know their names

Representations (4)

- We call this the *non-ground* representation – we represent a variable with a variable (*ie* a non-ground term)
- The alternative, which is better, but not straightforwardly available in Prolog, is the *ground* representation, where we use a ground term to represent each variable
- One advantage of the non-ground representation is that we can let Prolog look after unification by default, though this can easily lead to confusion!
- With the ground representation, variable bindings have to be handled explicitly

Example 1: the `solve` interpreter

- This is a standard approach to simulating and understanding the Prolog execution rule

- Here is an interpreter for Prolog with conjunction between literals and disjunction between clauses

```
solve( true ).
solve( Goal )  :- \+ Goal = ( _, _ ),
                  clause( Goal, Body ),
                  solve( Body ).
solve(( Goal1, Goal2 )) :-
                  solve( Goal1 ),
                  solve( Goal2 ).
```

- `clause/2` is a built-in predicate succeeds if its arguments are the head and body of a clause in the database

Example 1: the `solve` interpreter (2)

- We could rewrite the interpreter to make our computation rule work right-most first:

```
solve( true ).
solve( Goal )  :- \+ Goal = ( _, _ ),
                  clause( Goal, Body ),
                  solve( Body ).
solve(( Goal1, Goal2 )) :-
                  solve( Goal2 ),
                  solve( Goal1 ).
```

- And of course, we can do cleverer things, as in exercise 4

Example 2: breadth-first execution

- Another version of `solve`:

```
solve( Goal ) :-
        conj2list( Goal, List ).

solve2( [] ).
solve2( [true|T] ) :-
        solve2( T ).
solve( [Goal|Rest] ) :-
        \+ Goal = (_,_),
        clause( Goal, Body ),
        conj2list( Body, List ),
        append( Rest, List, New ),
        solve2( New ).
solve2( [(G1,G2)|Rest] ) :-
        solve2( [G1,G2|Rest] ).
```

(`conj2list/2` converts a conjunction to a list)

Example 3: reasoning about agents' knowledge

- The `demo/2` predicate is like `solve/1` but it is able to distinguish between different databases

```
demo( _AnyDB, true ).
demo( DB, Goal ) :-
        \+ Goal = ( _, _ ),
        demo_clause( DB, Goal, Body ),
        demo( DB, Body ).
demo( DB, (Goal1, Goal2)) :-
        demo( DB, Goal1 ),
        demo( DB, Goal2 ).
```

- `demo_clause/3` may be constructed using `clause/2` and the Prolog module system

- But we can extend this in very useful ways

Example 3: reasoning about agents' knowledge (2)

- In multi-agent reasoning systems, there is usually some common knowledge
- So, assuming we have a database `ck` containing this, we can add it in to our `demo/2` program:

```
demo( DB, Goal ) :- \+ DB = ck,
                    demo( ck, Goal ).
```

- We can add a meta-meta rule, which says that if something is common knowledge, then everyone knows that it is common knowledge:

```
demo( ck, demo( ck, Goal )) :-
        demo( ck, Goal ).
```

Example 3: reasoning about agents' knowledge (3)

- 1990, Kim & Kowalski used these methods to produce an automated proof of the "Three wise men problem":

  A king, wishing to determine which of his three wise men is the wisest, puts a white spot on each of their foreheads, and tells them that at least one of the spots is white. The king arranges the wise men in a circle so that they can see and hear each other (but cannot see their own spot) and asks each man in turn what is the colour of his spot. The first two say they don't know, and the third says that his spot is white.

  How does the third wise man reach his conclusion?

Example 3: reasoning about agents' knowledge (4)

- The solution runs as follows:

- The king poses his question to man 1, who answers "I don't know", so
  1. All know that man 1 knows that one spot is white;
  2. All know that man 1 does not know if his spot is white;

Example 3: reasoning about agents' knowledge (5)

- Next, the king poses his question to man 2, who also answers "I don't know", so

3. All know that man 1 knows what colours the others' spots are;
4. From 1, 2, 3, all know that man 1 knows that man 2 or man 3 has a white spot, or both;
5. From 4, man 2 knows that man 1 knows that man 2 or man 3 has a white spot or both;
6. From 5, man 2 knows that man 2 or man 3 has a white spot or both;
7. Man 2 does not know if his spot is white;

Example 3: reasoning about agents' knowledge (6)

- Next, the king poses his question to man 3, who answers "my spot is white", so

8. Man 2 knows what colour man 3's spot is;
9. From 6, 7, 8, man 2 knows that man 3's spot is white;
10. Man 3 knows that man 3's spot is white.