Meta-programming in Prolog (2)

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

Contents

• Summary of previous lecture
• Prolog meta-programming facilities
Summary of previous lecture

• Meta-programming is programming where the data represents a program

• We say that the data is the name of the program it represents

• In Prolog, we usually use as a name a term which is identical to the named program itself

• In Prolog, we usually use a non-ground representation – ie we use variables to name variables

• A better approach is a ground representation – but this has the disadvantage that we have to handle unification and variable bindings explicitly
Summary of previous lecture (2)

• It is possible to represent the Prolog execution mechanism as a meta-program, which refers directly to the Prolog database.

• It is possible to extend and adapt the execution mechanism in various ways, such as to change the execution rule, and to admit multiple databases.

• Meta-programming can allow significant increases in the reasoning power of the language, and can allow us to solve complex problems merely by stating them as logic programs.
Meta-programming predicates in Prolog can be divided into several headings:

- term/instantiation testing
- term manipulation
- database access
- database manipulation
- meta-level identity testing
Instantiation testing

- These predicates are used for testing if the meta-level representation of a program variable (which is itself a variable) has a value or not.
  
  `var(X)` succeeds if X is an uninstantiated variable;
  
  `nonvar(X)` succeeds if X is not an uninstantiated variable;
  
  `ground(X)` succeeds if X is a fully instantiated term.
Instantiation testing (2)

- For example, a correct use of `var/1` might be in an explicit unification algorithm, where we need to know if an object is a variable or a term.

- They are often abused for preventing object-level calls to predicates where instantiation is crucial, *eg*:

  ```prolog
  p(X, Y) :- ground(Y), X is Y.
  ```
Syntax checking

• These predicates are used for testing the syntactic shape of a meta-level term

\( \text{var}(X) \) succeeds if \( X \) is a variable;
\( \text{atom}(X) \) succeeds if \( X \) is a ground atom
\( \text{float}(X) \) succeeds if \( X \) is a ground floating-point number (\( \text{real}/1 \) in some implementations)
\( \text{integer}(X) \) succeeds if \( X \) is a ground integer
\( \text{number}(X) \) succeeds if \( X \) is a ground number
\( \text{atomic}(X) \) succeeds if \( X \) is a number or an atom
\( \text{simple}(X) \) succeeds if \( X \) is either a variable or atomic
\( \text{compound}(X) \) succeeds if \( X \) is neither a variable nor atomic
\( \text{callable}(X) \) succeeds if \( X \) is a well-formed Prolog goal
Prolog term manipulation

• These predicates are very useful for constructing meta-level terms, but also as short-cuts for building object-level terms when we really need to

functor(Term, Name, Arity) succeeds when Term has functor Name and arity Arity

arg(ArgNo, Term, Arg) succeeds when Arg is the Argnoth argument of Term

Term =.. List (univ) succeeds when List is of the form [F | L] and F is the functor of Term and L is a list of its arguments

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

L = [p,a,b]

yes

=../2 is often the most useful meta-predicate, as it can remove the need to write infinite numbers of clauses!
Prolog term manipulation (2)

name(const, CharList) succeeds if const is an atomic constant whose name is spelled in CharList

- Notation: we use the term "\texttt{abc}\" to denote the \textit{string} [97, 98, 99], which is the list of the ASCII codes for the letters $a$, $b$, $c$

\begin{verbatim}
?- name( 'Geraint', _GList ),
    name( 'Wiggins', _WList ),
    append( _GList, _WList, _Name ),
    name( Whole, _Name ).

Whole = 'GeraintWiggins'
yes
\end{verbatim}

\begin{verbatim}
?- name( 'Geraint', "Geraint" ).
\end{verbatim}
Prolog database access and manipulation

- We can access the contents of our program database thus:

\[
\text{clause}(\text{Head}, \text{Body}) \text{ succeeds if there is a clause unifiable with } \text{Head} \ :- \ \text{Body}. \text{ in the database. If } \text{Head} \text{ is a fact in the database, then } \text{Body} \text{ is set to true}
\]
Some of these predicates allow us to change the contents of the Prolog database. If you do this in an arbitrary way, you will probably damage the logic of your program.

- `assert(Clause)` adds `Clause` to the database.
- `asserta(Clause)` adds `Clause` to the database as the first clause of the predicate so defined.
- `assertz(Clause)` adds `Clause` to the database as the last clause of the predicate so defined.
- `retract(Clause)` removes the first clause to match `Clause` from the database.
- `retractall(Head)` removes all the clauses whose head match `Head` from the database.
Prolog database access/manipulation (3)

- In order to store (parts of) a predicate, the predicate must be dynamic.

- If you build a new predicate, Prolog will make it dynamic automatically; otherwise (i.e., if it is in your program file) you must declare it as dynamic:

  ```prolog
  :- dynamic p/1.
  p(2).
  ?- assertz( p(1) ),
     asserta( p(3) ), listing( p ).
  p(3).
  p(2).
  p(1).
  yes
  ```
Meta-level identity testing

• There are two predicates for testing syntactic identity at the meta-level
  
  \(==/2\) succeeds if its arguments are syntactically identical
  
  \(\not=\)/2 succeeds if its arguments are not syntactically identical

• Note the difference between \(=\) and \(==\):

  ?- X = Y.
  X = Y
  yes

  ?- X == Y.
  no

  ?- X = X.
  yes

  ?- X == X.
  yes

  ?= f(X) = f(1).
  X = 1
  yes

  ?- f(X) == f(1).
  no

"Meta-programming in Prolog (2)", CIS335: Logic Programming, Goldsmiths’ College, University of London
Meta-level identity testing (2)

• Note also that \( \neq \) does not mean “not equals”; it means “syntactically different from” and its result depends on the instantiation of its variables.

• For example,

\[
\begin{align*}
? - \ \ + \ X & = \ X. \\
\text{no} & \\
\end{align*}
\]

\[
\begin{align*}
? - \ \ + \ X & = \ Y. \\
\text{no} & \\
\end{align*}
\]

\[
\begin{align*}
? - \ X & \neq X. \\
\text{no} & \\
\end{align*}
\]

\[
\begin{align*}
? - \ X & \neq Y. \\
\text{yes} & \\
\end{align*}
\]

• It is incorrect to use \( \neq \) as object-level “not equals” because it depends on instantiation.