ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog – Meta-predicates

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

Meta-predicates

- In Prolog predicates (i.e., programs) and terms (i.e., data) have the same syntactical structure…
- … as a consequence, predicates and terms can be exchanged and exploited in different roles!!!

- There are several pre-defined predicates that allows to deal with these structures, and work with them

The `call` predicate

- If terms are predicates:
  - I could “prepare/create” a term in a ”proper” way…
  - … and then… EXECUTE IT!

- **`call(T)`**: the term **`T`** is considered as a atom (a predicate), and the Prolog interpreter is requested to evaluate (execute) it.
- Obviously, at the moment of the evaluation, **`T`** must be a non-numeric term
  - A number is a constant, and cannot be “evaluated” in logical sense

The `call` predicate

- The predicate **`call`** is considered to be a meta-predicate since:
  - its evaluation directly interfere with the Prolog interpreter underneath, within the same evaluation instance
  - It directly "alterates" the program.

- The predicate **`call`** takes as input a term that can be interpreted as a predicate:

```
p(a).
q(X):-  p(X).

:- call(q(Y)).
yes Y = a.
```

When executed, **`call`** asks to the Prolog interpreter of proving **`q(Y)`**

The `call` predicate

- The predicate call can be used also within programs:

```
p(X):-  call(X).
q(a).

:- p(q(Y)).
yes Y = a.
```

- Some Prolog interpreters allow the following notation

```
p(X):-  X.
```

- X is also said to be a *meta-logic variable*

The `call` predicate – Example

- Define a program that behaves as a procedural if_then_else construct

`if_then_else(Cond,Goal1,Goal2)`

  - If Cond is evaluated to true, then Goal1 is evaluated
  - If Cond is evaluated to false, then Goal2, is evaluated

```
if_then_else(Cond,Goal1,Goal2):-
        call(Cond), !,
        call(Goal1).
if_then_else(Cond,Goal1,Goal2):-
        call(Goal2).
```

The `fail` predicate

- **`fail`** takes no arguments (its arity is zero)

- Its evaluation always fails!
- As a consequence, it forces the interpreter to explore other alternatives…
- … in other words, it explicitly activates the backtracking

- Why on the earth should we force the failure of a proof?
  - To obtain some form of iteration over data;
  - To implement the negation as failure;
  - To implement a logical implication

The `fail` predicate – the iteration

- Let us consider a KB with facts **`p/1`**. Apply a predicate **`q(X)`** on all **`X`** that satisfy **`p(X)`**.
- A possible solution (not the only one) :

```
iterate :- call(p(X)),
           verify(q(X)),
           fail.
iterate.

verify(q(X)):- call(q(X)), !.
```

The `fail` predicate – the negation as failure

- Implement the negation as failure through a predicate **`not(P)`**
- **`not(P)`** is true if **`P`** is not a consequence of the program

```
not(P) :- call(P), !, fail.
not(P).
```

Combining `fail` and `cut`

- The sequence !, fail is often used to force a global failure of a predicate p (and not only backtracking)
- For example, define the fly property, that is true for all the birds except penguins and ostrich…
- … in other words, write a KB able to deal with default, but supporting exceptions.

```
fly(X)   :-   penguin(X), !, fail.
fly(X)   :-   ostrich(X), !, fail.
  ....
fly(X)   :-   bird(X).
```

Predicates `setof` and `bagof`

- In Prolog, the usual query **`:- p(X).`** is interpreted with **`X`** existentially quantified. As result, it is returned a possible substitution for variables of **`p`** such that the query is satisfied.

- Sometimes, it might be interested to answer to queries like "**`which is the set S of element X such that p(X) is true?`**" (second-order query)

- Some Prolog interpreters support this type of second-order queries by providing pre-definite predicates.

Predicates `setof` and `bagof`

- **`setof(X,P,S).`**
  S is the set of instances X that satisfy the goal P

- **`bagof(X,P,L).`**
  L is the list of instances X that satisfy the goal P

- In both cases, if there are no X satisfying P, the predicates fail.
- Which is the difference? **`bagof`** returns a list possibly containing repetitions, **`setof`** should not contain repetitions. Not always true: it depends by the implementation.

Predicates `setof` and `bagof` – examples

Knowledge Base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

```
:- setof(X, p(X), S).
 yes S = [0,1,2]
              X = X
:- bagof(X, p(X), S).
 yes S = [1,2,0,1]
              X = X
```

NOTICE: variable X, at the end, has not been unified with any value.

Predicates `setof` and `bagof` – examples

Parameters are reversible. Knowledge Base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

```
:- setof(X, p(X), [0,1,2]).
 yes   X = X

:- bagof(X, p(X), [1,2,0,1]).
 yes X = X
```

Predicates `setof` and `bagof` – examples

Conjunction of goals. Knowledge Base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

```
:- setof(X, (p(X),q(X)), S).
 yes S = [2]
         X = X
:- bagof(X, (p(X),q(X)), S).
 yes S = [2]
         X = X
```

```
:- setof(X,(p(X),r(X)),S).
 no

:- bagof(X,(p(X),r(X)), S).
 no
```

Predicates `setof` and `bagof` – examples

Non existing goals. Knowledge Base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

```
:- setof(X, s(X), S).
 no

:- bagof(X, s(X), S).
 no
```

NOTICE: this is the expected behaviour…
however, some interpreters return an error
`calling an undefined procedure s(X)`

Predicates `setof` and `bagof` – examples

Complex terms in the answer. Knowledge Base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

```
:- setof(p(X), p(X),S).
 yes S=[p(0),p(1),p(2)]
     X=X
:- bagof(p(X), p(X), S).
 yes S=[p(1),p(2),p(0),p(1)]
     X=X
```

Predicates `setof` and `bagof` -- quantificaiton of variables

- Knowledge base:

```
father(giovanni,mario).
father(giovanni,giuseppe).
father(mario, paola).
father(mario,aldo).
father(giuseppe,maria).
```

I was expecting all the X for which father(X,Y) is true…

INSTEAD, it returned those X for which, for the same value of Y, father(X,Y) is true.

?????

```
:-  setof(X, father(X,Y), S).
      yes    X=X    Y= aldo        S=[mario];
             X=X    Y= giuseppe    S=[giovanni];
             X=X    Y= maria       S=[giuseppe];
             X=X    Y= mario       S=[giovanni];
             X=X    Y= paola       S=[mario];
      no
```

Predicates `setof` and `bagof` -- quantificaiton of variables

- Knowledge base:

```
father(giovanni,mario).
father(giovanni,giuseppe).
father(mario, paola).
father(mario,aldo).
father(giuseppe,maria).
```

We need to specify that Y has to be quantified existentially…

```
:-  setof(X, Y^father(X,Y), S).
 yes  [giovanni,mario,giuseppe]
      X=X
      Y=Y
```

Predicates `setof` and `bagof` -- quantificaiton of variables

- Compound terms in the answer. Knowledge base:

```
father(giovanni,mario).
father(giovanni,giuseppe).
father(mario, paola).
father(mario,aldo).
father(giuseppe,maria).
```

```
      :-  setof((X,Y), father(X,Y), S).
       yes  S=[(giovanni,mario),(giovanni,giuseppe),
               (mario, paola), (mario,aldo),
               (giuseppe,maria)]
            X=X
            Y=Y
```

Predicate `findall`

- Often, we are interested in setof and bagof, with the semantics of the variables not appearing in the first argument being quantified existentially…

`findall(X, P, S)`

- true if S is the list of instance X (without repetitions) for which predicate P is true.

- If there is no X satisfying P, then findall returns an empty list.

Predicate `findall` – example

- Knowledge base:

```
father(giovanni,mario).
father(giovanni,giuseppe).
father(mario, paola).
father(mario,aldo).
father(giuseppe,maria).
```

```
      :-  findall(X, father(X,Y), S).
       yes  S=[giovanni, mario, giuseppe]
            X=X
            Y=Y
```

Equivalent to:

```
      :-  setof(X, Y^father(X,Y), S).
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`Bagof`, `setof`, and `findall` are not limited to facts

- Predicates **`setof`**, **`bagof`** e **`findall`** works also when the property to be verified is not a simple fact, but it si defined by rules.

```
p(X,Y):- q(X),r(X).
q(0).
q(1).
r(0).
r(2).
```

```
:-  findall(X, p(X,Y), S).
 yes  S=[0]
      X=X
      Y=Y
```

Verification of implication through `setof`

- Let us have predicates of the type **`father(X,Y)`** and **`employee(Y)`**
- We want to verify if it is true that for every **`Y`** for which **`father(p,Y)`** holds, then **`Y`** is an employee (all the sons of p are employees)
  father(p,Y) → employee(Y)

```
imply(Y):- setof(X, father(Y,X), L), verify(L).

verify([]).
verify([H|T]):- employee(H), verify(T).
```

Iteration through `setof`

- Execute the procedure q on each element for which p is true

```
iterate:- setof(X, p(X),L),
             filter(L).
  filter([]).
  filter([H|T]):- call(q(H)),
                          filter(T).
```

- Which difference with the implementation made through fail? What about backtrackability?

Verifying properties of terms

- **`var(Term)`**
  true if Term is currently a variable

- **`nonvar(Term)`**
  true if Term currently is not a free variable

- **`number(Term)`**
  true if Term is a number

- **`ground(Term)`**
  true if Term holds no free variables.

Accessing the structure of a term

- Term =.. List
  [SWI documentation]
  - List is a list whose head is the functor of Term and the remaining arguments are the arguments of the term.
  - Either side of the predicate may be a variable, but not both.

```
?- foo(hello, X) =.. List.
List = [foo, hello, X]

?- Term =.. [baz, foo(1)].
Term = baz(foo(1))
```

Accessing the clauses of a program

- In Prolog, terms and predicates are represented with the same structure
- In particular, a clause (a query) is represented as a term
- Example: given

```
h.
h :- b1, b2, …, bn.
```

- They correspond to the terms:

```
(h, true)
(h, ','(b1, ','(b2, ','( ...','( bn-1, bn) ...)))
```

Accessing the clauses of a program – the predicate `clause`

- **`clause(Head, Body)`**
  true if (Head, Body) is unified with a clause stored within the database program

- When evaluated
  – Head must be instantiated to a non-numeric term
  – Body can be a variable or a term describing the body of a clause

- Its evaluation opens choice points, if more clauses with the same head are available

The predicate `clause` – example

- Program:

```
?-dynamic(p/1).
?-dynamic(q/2).
p(1).
q(X,a) :- p(X), r(a).
q(2,Y) :- d(Y).
```

```
?-   clause(p(1),BODY).
          yes     BODY=true
?-   clause(p(X),true).
          yes     X=1
?-   clause(q(X,Y), BODY).
          yes     X=_1    Y=a     BODY=p(_1),r(a);
                  X=2     Y=_2    BODY=d(_2);
          no
?-   clause(HEAD,true).
           Error - invalid key to data-base
```

Other amenities… Loading modules and libraries

- Modern Prolog interpreters come equipped with a huge library of code.
- To load a library:
  `use_module(library(XXX)).`

- Example (SWI Prolog):
  `:- use_module(library(lists)).`
  Load the pre-defined predicates for dealing with lists

- library(aggregate)
- library(ansi_term)
- library(apply)
- library(assoc)
- library(broadcast)
- library(charsio)
- library(check)

Other amenities… Loading modules

- library(**clpb**)
- library(**clpfd**)
- library(**clpqr**)
- library(csv)
- library(**dcgbasics**)
- library(dcghighorder)
- library(debug)
- library(dicts)
- library(error)
- library(explain)
- library(help)
- library(intercept)
- library(summaries.d/intercept.tex)
- library(iostream)
- library(lists)
- library(main)
- library(occurs)
- library(option)
- library(optparse)
- library(ordsets)
- library(persistency)
- library(predicate_options)
- library(prologjiti)
- library(prologpack)
- library(prologxref)
- library(pairs)
- library(pio)
- library(random)
- library(readutil)
- library(record)
- library(registry)
- library(settings)
- library(simplex)
- library(ugraphs)
- library(url)
- library(www_browser)
- library(solution_sequences)
- library(thread)
- library(thread_pool)
- library(varnumbers)
- library(yall)

Other amenities… packages

- SWI-Prolog Semantic Web Library 3.0
- Constraint Query Language A high level interface to SQL databases
- SWI-Prolog binding to GNU readline
- SWI-Prolog ODBC Interface
- SWI-Prolog binding to libarchive
- Transparent Inter-Process Communications (TIPC) libraries
- JPL: A bidirectional Prolog/Java interface
- Pengines: Web Logic Programming Made Easy
- SWI-Prolog SSL Interface
- Google's Protocol Buffers Library
- SWI-Prolog Natural Language Processing Primitives
- Prolog Unit Tests
- SWI-Prolog Unicode library
- SWI-Prolog YAML library
- SWI-Prolog HTTP support

Other amenities… packages

- SWI-Prolog Regular Expression library
- Managing external tables for SWI-Prolog
- A C++ interface to SWI-Prolog
- SWI-Prolog SGML/XML parser
- SWI-Prolog binding to zlib
- Paxos -- a SWI-Prolog replicating key-value store
- SWI-Prolog Source Documentation Version 2
- SWI-Prolog C-library
- SWI-Prolog binding to BSD libedit
- SWI-Prolog RDF parser