ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog Meta-Interpreters

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

The starting point

- In Prolog, no difference between programs and data, or, using the Prolog terminology, …
- …No difference (in the representation) between predicates and terms.

- We can ask the Prolog interpreter to provide us the clauses of the (currently loaded) program
  - **`clause(Head, Body).`**

- We can also ask the interpreter to "execute" a term
  - **`call(T).`**

Meta-interpreters

- Meta-interpreters as meta-programs, i.e., programs who execute/works/deal with other programs
  - Programs as input of meta-programs

- Used for rapid prototyping of interpreters of symbolic languages

- In Prolog, a meta-interpreter for a language L is defined as an interpreter for L, but written in Prolog

- Given the premises, would it be possible to write a Prolog interpreter for the language Prolog?

Meta-interpreter for Pure Prolog
aka the vanilla meta-interpreter

- Define a predicate solve(goal) that answers true if Goal can be proved using the clauses of the current program

```
solve(true) :- !.
solve( (A,B) ) :- !, solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

- Notice: it does not deal with pre-defined predicates…
  - For each pre-defined predicate, needs to add a specific solve clause that deal with it

Meta-interpreter for Pure Prolog aka the vanilla meta-interpreter

```
solve(true) :- !.
solve( (A,B) ) :- !, solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

- Notice: no need to "call" any predicate. The vanilla meta-interpreter explores the current program, searching for the clauses, until it can prove the goal, or it fails.

- As it is, the vanilla meta-interpreter mimic the standard behaviour of the Prolog interpreter…
- … but now, we can modify it and get different behaviours

Meta-interpreters for Pure Prolog – Example
Right-most selection rule

Example: define a Prolog interpreter that adopts the calculus rule "right most":

```
solve(true) :- !.
solve( (A,B) ) :- !, solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

```
solve(true) :- !.
solve( (A,B) ) :- !, solve(B), solve(A).
solve(A) :- clause(A,B), solve(B).
```

Meta-interpreters for Prolog – Example

Define a Prolog interpreter **`solve(Goal, Step)`** that:

- It is true if **`Goal`** can be proved
- In case **`Goal`** is proved, **`Step`** is the number of resolution steps used to prove the goal
  - In case of conjunctions, the number of steps is defined as the sum of the steps needed for each atomic conjunct

Meta-interpreters for Prolog – Example

Given the program:

```
a :- b, c.
b :- d.
c.
d.

?- solve(a, Step)
yes  Step=4
```

Why?

Meta-interpreters for Prolog – Example

Define a Prolog interpreter **`solve(Goal, Step)`** that:

- It is true if **`Goal`** can be proved
- In case **`Goal`** is proved, **`Step`** is the number of resolution steps used to prove the goal
  - In case of conjunctions, the number of steps is defined as the sum of the steps needed for each atomic conjunct

```
solve(true,0):-!.
solve((A,B),S) :- !, solve(A,SA),
                     solve(B,SB),
                     S is SA+SB.
solve(A,S) :-  clause(A,B),
               solve(B,SB),
               S is 1+SB.
```

Meta-interpreters for Prolog – Example

Let us suppose to represent a knowledge base in terms of rules, and for each rule we have also a "certainty" score (between 0 and 100).

Example:

```
rule(a, (b,c), 10).
rule(b, true, 100).
rule(c, true, 50).
```

Meta-interpreters for Prolog – Example

- Define a meta-interpreter **`solve(Goal,CF)`**, that is true if **`Goal`** can be proved, with certainty **`CF`**.

- For conjunctions, the certainty is the minimum of the certainties of the conjuncts

- For rules, the certainty is the product of the certainty of the rule itself times the certainty of the proof of the body (eventually divided by 100).

Meta-interpreters for Prolog – Example

```
rule(a, (b,c), 10).
rule(a, d, 90).
rule(b,true, 100).
rule(c,true, 50).
rule(d,true, 100).

?-solve(a,CF).
yes CF=5;
yes CF=90
```

Meta-interpreters for Prolog – Example

```
solve(true,100):-!.
solve((A,B),CF) :- !, solve(A,CFA),
                      solve(B,CFB),
                      min(CFA,CFB,CF).
solve(A,CFA) :- rule(A,B,CF),
                solve(B,CFB),
                CFA is ((CFB*CF)/100).

min(A,B,A) :- A<B,!.
min(A,B,B).
```

Meta-interpreters – a simple Expert System

- Let us suppose we have a knowledge base, and we want to query it looking to prove/verify something

```
good_pet(X) :- bird(X), small(X).
good_pet(X) :- cuddly(X), yellow(X).
bird(X) :- has_feathers(X), tweets(X).
yellow(tweety).
```

We want to know if **`tweety`** is a good pet:

```
?- good_pet(tweety).
ERROR: Undefined procedure: has_feathers/1
```

- Does it mean that we do not know if **`tweety`** has feathers?
- Does it mean that we do not know anything about the concept "having feathers"?

Meta-interpreters – a simple Expert System

- Idea: extend your **KB** and/or **reasoning tool**, so that:
  - It tries to prove a Goal using the given KB
  - If it fails, the reasoner could also ask help to the user

(alternative) Solutions:

1. Modify our knowledge base
2. [Implement a new/extend existing] reasoning tool

Example taken from:

https://swish.swi-prolog.org/example/expert_system.pl

Meta-interpreters – a simple Expert System

- Idea: extend your **KB** and/or **reasoning tool**, so that:
  - It tries to prove a Goal using the given KB
  - If it fails, the reasoner could also ask help to the user

```
prove(true) :- !.
prove((B, Bs)) :- !,
  prove(B),
  prove(Bs).
prove(H) :-
  clause(H, B),
  prove(B).
prove(H) :-
  write('Is '), write(H), writeln(' true?'),
  read(Answer), get_code(_),
  Answer = yes.
```

`=/2` tries to unify the two arguments. If it succeeds, the two arguments are unified later.

`get_code(_)` is needed due to the implemented behaviour of read/1, that leaves a char in the buffer. Notice: it behaves differently on windows, and on the swish web app.

Meta-interpreters – a simple Expert System

- But… asked predicate can be asked in this way… cannot we limit the user question to a user-specified list of predicates?

```
prove(true) :- !.
prove((B, Bs)) :- !,
  prove(B),
  prove(Bs).
prove(H) :-
  clause(H, B),
  prove(B).
prove(H) :-
  askable(H),
  write('Is '), write(H), writeln(' true?'),
  read(Answer), get_code(_),
  Answer == yes.
```

```
% Only askable predicates can be
 asked to the user
askable(tweets(_)).
askable(small(_)).
askable(cuddly(_)).
askable(has_feathers(_)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The KB
good_pet(X) :-
 bird(X), small(X).
good_pet(X) :-
 cuddly(X), yellow(X).
bird(X) :-
 has_feathers(X), tweets(X).

yellow(tweety).
```

Meta-interpreters – Exercise

- Write a meta-interpreter for the Prolog language that prints out, before and after the execution of a subgoal, the subgoal itself. Example:

```
p(X) :- q(X).
q(1).
q(2).

?- solve(p(X)).
yes   X/1
Solving: p(X_e0)
Selected Rule: p(X_e0):-q(X_e0)
Solving: q(X_e0)
Selected Rule: q(1):-true
Solved: true
Solved: q(1)
```

Meta-interpreters – Exercise – Solution

- The solution is obtained by simply modifying the meta interpreter vanilla

```
solve(true):- !.
solve((A,B)):- !, solve(A), solve(B).
solve(A) :-
  write('Solving: '), write(A), nl,
  clause(A,B),
  write('Selected Rule: '), write(A), write(':-'), write(B), nl,
  solve(B),
  write('Solved: '), write(B), nl.
```

Meta-interpreters – Exercise (variation)

- Write a meta-interpreter for the Prolog language that prints out, before and after the execution of a subgoal, the subgoal itself. Subgoals should also be "tabbed" on the right depending on the depth of the resolution tree. Example:

```
p(X) :- q(X).
q(1).
q(2).


?- s(p(X)).
yes   X/1
Solving: p(X_e0)
   Selected Rule: p(X_e0):-q(X_e0)
   Solving: q(X_e0)
      Selected Rule: q(1):-true
      Solved: true
   Solved: q(1)
```

Meta-interpreters – Exercise (variation) – Solution

```
s(true, N):- !.
s((A,B), N):- !, s(A, N), s(B, N).
s(A, N) :-
        tt(N), write('Solving: '), write(A), nl,
        clause(A,B),
        N1 is N+1,
        tt(N1), write('Selected Rule: '), write(A), write(":-"), write(B), nl,
        s(B,N1),
        tt(N1), write('Solved: '), write(B), nl.

tt(0).
tt(N):-
        N>0,
        tab(3),
        N1 is N-1,
        tt(N1).
```

Dynamically modifying the program

- When a Prolog program is consulted/loaded, its representation in terms of data structures (terms) is loaded into a table in memory
- Such table is often referred as the program database
  - Indeed, it is managed using DBMS techniques for increasing performances
  - For example, functors of the heads are indexed, to speed-up the search for possible candidates for unification with a goal

- If it is a table, can we change it?
  - Add entries to the table?
    Means adding new clauses for a predicate
    In procedural terms, it would be like adding new methods
  - Remove entries from the table?

Dynamically modifying the program – `assert`

`assert(T)`

Clause **`T`** is added to the database program.

- When **`assert`** is evaluated, **`T`** must be instantiated to a term denoting a clause (either a fact or a rule).
- **`T`** is added to the database program in a non-specified position.
- In backtracking, **`assert`** is ignored
  - Non declarative behaviour
  - Added clauses are not removed by backtracking
- For efficiency reasons, functors of predicates that will be added must be declared as "dynamic“:
  `:- dynamic(foo/1)).`

Dynamically modifying the program – `assert`

`assert(T)`

Clause **`T`** is added to the database program.

- However, the order of the clause definitions in Prolog does have a (important!) meaning
- Two variations available:

- **`asserta(T)`**
  T is added at the beginning of the database
- **`assertz(T)`**
  T is added ad the end of the database

- The behaviour can greatly change…

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dynamically modifying the program – `assert`

```
?-dynamic(a/1).
a(1).
b(X):-a(X).
```

```
?-   assert(a(2)).
```

```
a(1).
a(2).
b(X):-a(X).
```

```
?-   asserta(a(3)).
```

```
a(3).
a(1).
a(2).
b(X):-a(X).
```

```
?-   assertz(a(4)).
```

```
a(3).
a(1).
a(2).
a(4).
b(X):-a(X).
```

Dynamically modifying the program – `retract`

`retract(T)`

The first clause in the database that unifies with **`T`** is removed.

- When evaluated, **`T`** should be instantiated to a term denoting a clause
- If more than one clauses unify with **`T`**, the first one is removed; some Prolog implementations keep tracks with a backtrackable choice point.

- Some Prolog implementations provide the predicate **`abolish/retract_all`**, that remove all the occurrences of the specified **`term`** with **`arity`**

Dynamically modifying the program – `retract`

```
?-dynamic(a/1).
?-dynamic(b/1).
a(3).
a(1).
a(2).
a(4).
b(X):-c(X),a(X).
```

```
?-  retract(a(X)).
yes X=3
```

```
a(1).
a(2).
a(4).
b(X):- c(X),a(X).
```

```
?-  abolish(a,1).
```

```
b(X):- c(X),a(X).
```

```
?- retract((b(X):-BODY))
yes BODY=c(X),a(X)
```

Dynamically modifying the program – `retract`

```
a(3).
a(1).
a(2).
a(4).
b(X):-c(X),a(X).
```

```
?-   retract(a(X)).
yes X=3;
```

```
a(1).
a(2).
a(4).
b(X):- c(X),a(X).
```

```
yes X=1;
```

```
a(2).
a(4).
b(X):- c(X),a(X).
```

```
yes X=2;
```

```
a(4).
b(X):- c(X),a(X).
```

```
yes X=4;
no
```

```
b(X):- c(X),a(X).
```

`assert` and `retract` – few issues…

- When using assert and retract, the declarative semantics of Prolog is lost

- Consider an empty program, and the following queries:

```
?- assert(p(a)), p(a).
?- p(a), assert(p(a)).
```

- The first query succeeds; the second query fails.

- The order of the literals plays a fundamental role (but the same holds for the cut, for the negation with unbound variables, etc. etc.)

`assert` and `retract` – few issues…

Another example:

```
a(1).                                      (P1)
p(X)  :-  assert((b(X))), a(X).

a(1).                                      (P2)
p(X)  :-  a(X), assert((b(X))).
```

The query **`:- p(X).`** produces the same answer, but two different database modifications/

- in P1 , **`b(X)`** is added to the database: ∀**`X p(X)`**
- in P2 , **`b(1)`** only is added to the database.

`assert` and `retract` – few issues…

- A further problem is about the quantification of variables
- Variables in clauses are quantified universally…
- Variables in queries are quantified existentially.

Consider the query `:- assert((p(X))).`

- **`X`** is existentially quantified.
- However, the database is extended with the clause
  `p(X).`
- Formula: **$\forall$`X p(X)`**

`assert` and `retract` – example: the lemma generation

- Simple recursive solutions for computing Fibonacci are usually very inefficient

```
% fib(N,Y) "Y is the Nth Fibonacci number"

fib(0,0)   :-   !.
fib(1,1)   :-   !.
fib(N,Y)   :-    N1 is N-1, fib(N1,Y1),
                 N2 is N-2, fib(N2,Y2),
                 Y is Y1+Y2.,
                 generate_lemma(fib(N,Y)).
```

`assert` and `retract` – example: the lemma generation

```
generate_lemma (T) :- asserta(T).
```

- Alternatively:

```
generate_lemma (T) :- clause(T,true), !.
generate_lemma (T) :- asserta(T).
```

- The second solution checks that the same lemma is not added multiple times to the database