Meta-interpreters in Prolog

The Power of Prolog

2025-07-09

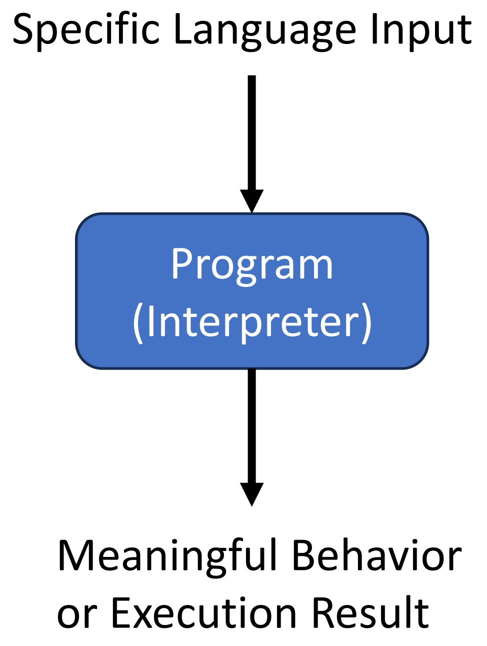
Presented by Xin-Shuang Zhang

Interpretation is pervasive

Most programs are interpreters for specific languages.

Examples:

- a web browser interprets HTML, JavaScript, HTTP...
- Python interpreter reads and executes Python code.
- many programs interpret command line arguments
- an editor interprets user input, regular expressions etc.
- configuration files and settings need to be interpreted
- documents and graphs are described in PostScript, Tex etc.
- ...



Meta-interpreters

A ***meta-interpreter (MI)*** interprets a language similar or identical to its own implementation language.

Why write *MI* in Prolog? ——Create **special-purpose** interpreters

The writing of meta-programs, is particularly easy in Prolog, because:

- The equivalence of programs and data: both are Prolog terms
- Prolog's implicit mechanisms can be used in interpreters
- Prolog is a simple language. Only construct "Head:-Body"

The Coarsest Meta-interpreters

```
prove(A) :- call(A).
```

- This is the **coarsest form** of meta-programming:
  By using call/1, we **absorb** backtracking, unification, handling of conjunctions, the call stack etc.

Absorption: *Implicitly* using features of the underlying engine

Reification: Making features *explicit*

- We can make these features **explicit** and subsequently adjust and extend them at will
- Differences in meta-interpreters can be characterized in their **granularity**

The Best Known Meta-interpreters

```
prove(true).
prove((A, B)) :- prove(A), prove(B).
prove(G) :- clause(G, Body), prove(Body).
```

Need cut or extra conditions like

```
G \= true,
G \= (_,_),
```

Preliminary: clause/2

`clause(Head, Body)` ⟺ There is a clause "`Head :- Body.`" (by unification)

Example:

```
h(X,Y):-f(X),g(Y).
h(u,v).
```

?- clause(h(A,B), Body).

```
  Body = (f(A),g(B))          % Unification is implicit
; Body = true, A = u, B = v   % The body of facts is true
```

Example:

```
complicated(A) :- g1(A), g2(A), g3(A).
```

?- clause(complicated (Z), Body).

Body = (g1(Z), g2(Z), g(Z)).

The Best Known/Vanilla Meta-interpreters

```
prove(true).
prove((A, B)) :- prove(A), prove(B).
prove(G) :- clause(G, Body), prove(Body).
```

Need cut or extra conditions like
`G \= true,`
`G \= (_,_),`

Explanation:

- The empty goal represented by the constant **`true`**
- A conjunction **`(A, B)`** is true if **A** is true and **B** is true
  (guarantees that the leftmost goal in the conjunction is solved first)
- A goal **G** is true if there is a clause **`G:-Body`** in program such that **Body** is true (responsible for giving different solutions on backtracking)

Example: Membership of A List

```prolog
% base
prove(true).

% 合取
prove((A,B)):-prove(A), prove(B).

% goal不是base也不是合取
% 递归：尝试Goal:-Body且prove(Body)
prove(G):-G \= true,
          G \= (_,_),
          clause(G,Body),
          prove(Body).

% member/2定义
%如果 X 是列表的第一个元素，则 X 在列表中。
%如果 X 不是第一个元素，就递归检查剩余部分 Ys。
member(X, [X|_]).
member(X, [_|Ys]):-member(X,Ys).
```

https://www.swi-prolog.org/

```prolog
member(X, [a,b,c]).
```

X	
a	1
b	2
c	3

false

```prolog
prove(member(X, [a,b,c])).
```

X	
a	1
b	2
c	3

false

Meta-interpreters Explicit Trace

```prolog
% base
prove(true):-
    writeln('成功: true').
% 递归
prove(G):-G \= true,
        G \= (_,_),
        clause(G,Body),
        format('尝试匹配规则: ~w :- ~w~n', [G, Body]),
        prove(Body).

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

```
prove(member(X, [a,b,c])).
尝试匹配规则: member(a,[a,b,c]) :- true
成功: true
X = a
尝试匹配规则: member(_736,[a,b,c]) :- member(_736,[b,c])
尝试匹配规则: member(b,[b,c]) :- true
成功: true
X = b
尝试匹配规则: member(_730,[b,c]) :- member(_730,[c])
尝试匹配规则: member(c,[c]) :- true
成功: true
X = c
尝试匹配规则: member(_730,[c]) :- member(_730,[])
false
```

- backtracking
- unification

Meta-interpreters Explicit Trace

```
?-prove(member(X, [a, b, c])).

    clause(member(X, [a, b, c]), B)          {X=a, B=true}
    prove(true)
        true      Output: X = a
    clause(member(X, [a, b, c]), B)          {B=member(X, [b,c])}
    prove(member(X, [b, c]))
    |
    +-> clause(member(X, [b, c]), B1)        {X=b, B1=true}
    |   prove(true)
    |       true      Output: X = b
    +-> clause(member(X, [b, c]), B1)        {B1=member(X, [c])}
        prove(member(X, [c]))
        |
        +-> clause(member(X, [c]), B2)       {X=c, B2=true}
        |   prove(true)
        |       true      Output: X = c
        +-> clause(member(X, [c]), B2)       {B2=member(X, [])}
            prove(member(X, []))
            |
            +-> clause(member(X, []), B3)    false
                  no (more) solutions
```

```
prove(true).
prove(G):-G \= true,
          G \= (_,_),
          clause(G,Body),
          prove(Body).

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

Inductive Logic Programming (ILP)

- A form of machine learning
- Given training examples, induce a hypothesis (a set of logical rules) that generalizes

Background knowledge:

$$reverse(X,Y){:}-\cdots$$
$$upper(X,Y){:}-\ldots$$
$$lower(X,Y){:}-\ldots$$

Positive examples:

$$\{target(['a','l','i','c','e'],['E','C','I','L','A']),$$
$$target(['a','l','i','c','e'],['E','C','I','L','A'])\}$$

Negative examples:

$$\{target(['a','l','i','c','e'],['a','l','i','c','e'])\}$$

Solution:

$$\boxed{target(X,Y) :- reverse(X,Z), upper(Z,Y).}$$

Metarules

- P, Q, R: second-order variables (i.e. can unify with predicate symbols)
- A, B, C: first-order variables (i.e. can unify with constant symbols)

```
metarule([P,Q], [P,A,B], [[Q,A,B]]). % identity
metarule([P,Q], [P,A,B], [[Q,B,A]]). % inverse
metarule([P,Q,R], [P,A,B], [[Q,A],[R,A,B]]). % precon
metarule([P,Q,R], [P,A,B], [[Q,A,B],[R,B]]). % postcon
metarule([P,Q,R], [P,A,B], [[Q,A,C],[R,C,B]]). % chain
```

```
P(A,B):-Q(A,B).
P(A,B):-Q(B,A).
P(A,B):-Q(A),R(A,B).
P(A,B):-Q(A,B),R(B).
P(A,B):-Q(A,C),R(C,B).
```

- A way to define the hypothesis space
- Restrict infinite hypothesis space to make the search feasible
- A hypothesis is an instance of the given metarules

Example

META-RULE	含义	示例
P(A,B) :- Q(A,B)	直接继承	parent(X,Y) :- father(X,Y)
P(A,B) :- Q(B,A)	反向继承	sibling(X,Y) :- sibling(Y,X)
P(A,B) :- Q(A), R(A,B)	前提 + 关系	can_teach(T,C) :- qualified(T), teaches(T,C)
P(A,B) :- Q(A,B), R(B)	关系 + 约束	can_drive(P,C) :- has_license(P,C), registered(C)
P(A,B) :- Q(A,C), R(C,B)	间接关系	grandparent(G,C) :- parent(G,P), parent(P,C)

Meta-Interpretive Learning

https://github.com/metagol/metagol

Professor Stephen H. Muggleton

FREng

Professor of Machine Learning

School of Artificial Intelligence
Nanjing University

- Metagol is an inductive logic programming (ILP) system based on meta-interpretive learning (MIL).
- Metagol is written in Prolog and runs with SWI-Prolog.
- The key idea of MIL is to **use metarules to restrict the form of hypothesis space**

Example: Learning the Grandparent Relation

The following code demonstrates **learning the grandparent relation** given the mother and father relations as background knowledge:

```
:- use_module('metagol').

%% metagol settings
body_pred(mother/2).
body_pred(father/2).

%% background knowledge
mother(ann,amy).
mother(ann,andy).
mother(amy,amelia).
mother(linda,gavin).
father(steve,amy).
father(steve,andy).
father(gavin,amelia).
father(andy,spongebob).

%% metarules
metarule([P,Q],[P,A,B],[[Q,A,B]]).
metarule([P,Q,R],[P,A,B],[[Q,A,B],[R,A,B]]).
metarule([P,Q,R],[P,A,B],[[Q,A,C],[R,C,B]]).
```

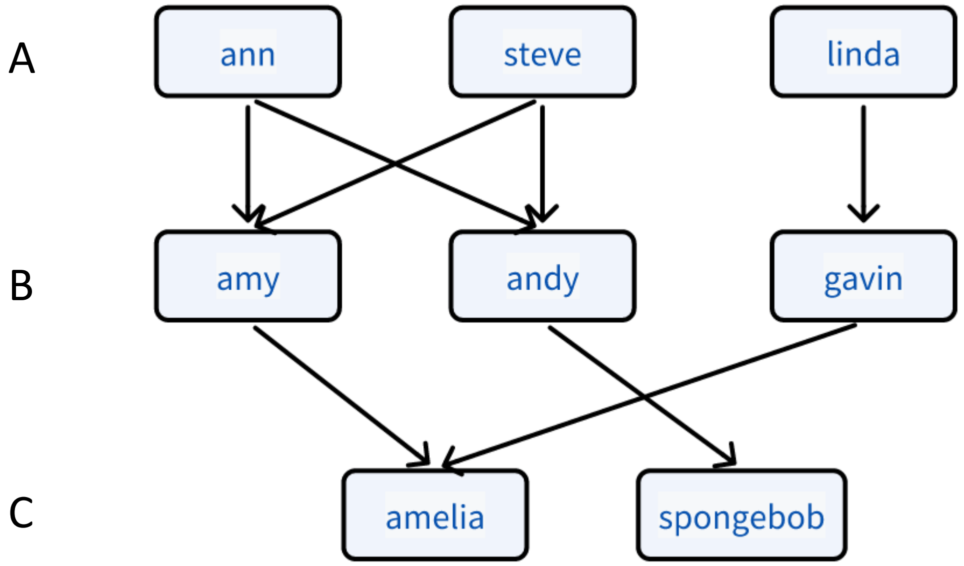
```
%% learning task
:-
  %% positive examples
  Pos = [
    grandparent(ann,amelia),
    grandparent(steve,amelia),
    grandparent(ann,spongebob),
    grandparent(steve,spongebob),
    grandparent(linda,amelia)
  ],
  %% negative examples
  Neg = [
    grandparent(amy,amelia)
  ],
  learn(Pos,Neg).
```

Running the above program will print the output:

next page

Output

```
% clauses: 1
% clauses: 2
% clauses: 3
grandparent(A,B):-grandparent_1(A,C),grandparent_1(C,B).
grandparent_1(A,B):-mother(A,B).
grandparent_1(A,B):-father(A,B).
```



Why metagol?

- **Predicate Invention →** `grandparent_1/2`
- **Recursion rule learning →**

```
ancestor(A, D) :- parent(A, D).
ancestor(A, D) :- parent(A, B), ancestor(B, D).
```

Hypothesising an Algorithm from One Example: the Role of Specificity (2023)

Expert Systems in Prolog

The Power of Prolog

2025-07-09

Presented by Xin-Shuang Zhang

Introduction

An expert system emulates the decision-making ability of a human expert.

Prolog is very well suited for implementing expert systems due to several reasons:

- Prolog itself can be regarded as a simple ***inference engine*** or ***theorem prover*** that derives conclusions from known rules. Very simple expert systems can be implemented by relying on Prolog's built-in search and backtracking mechanisms.
- Prolog data structures let us flexibly and conveniently represent ***rule-based systems*** that need additional functionality such as probabilistic reasoning.
- We can easily write ***meta-interpreters*** in Prolog to implement custom evaluation strategies of rules.

Example: Animal Identification

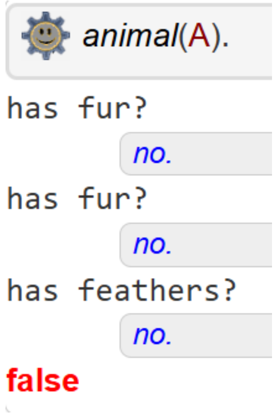
我们的目标是编写一个专家系统，用来帮助识别动物。假设我们已经掌握了以下关于动物的知识，也就是一些推理规则：

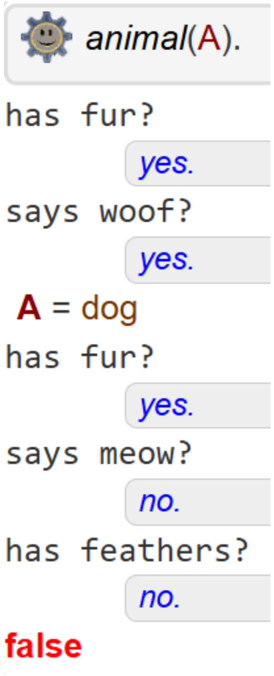
- 如果一个动物有毛皮 (*fur*) 并且会叫 “汪” (*woof*)，那么它是狗 (**dog**)。
- 如果一个动物有毛皮 (*fur*) 并且会叫 “喵” (*meow*)，那么它是猫 (**cat**)。
- 如果一个动物有羽毛 (*feathers*) 并且会叫 “嘎” (*quack*)，那么它是鸭子 (**duck**)。

这些规则并不完整，但它们作为一个示例用例，可以很好地说明专家系统的一些核心思想。

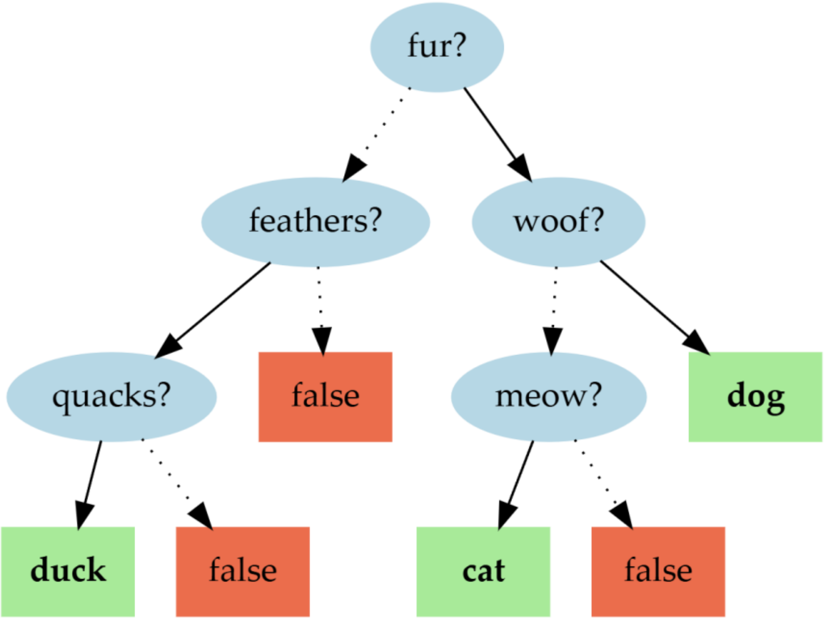
```
animal(dog)  :- is_true("has fur"), is_true("says woof").
animal(cat)  :- is_true("has fur"), is_true("says meow").
animal(duck) :- is_true("has feathers"), is_true("says quack").

is_true(Q) :-
        format("~s?\n", [Q]),
        read(yes).
```





Using a Domain-Specific Language (DSL)




```
animals([animal(dog, [is_true("has fur"), is_true("says woof")]),
        animal(cat, [is_true("has fur"), is_true("says meow")]),
        animal(duck, [is_true("has feathers"), is_true("says quack")])]).

tree(if_then_else("has fur",
                  if_then_else("says woof",
                               animal(dog),
                               if_then_else("says meow",
                                            animal(cat),
                                            false)),
                  if_then_else("has feathers",
                               if_then_else("says quack",
                                            animal(duck),
                                            false),
                               false))).
```

Using a Domain-Specific Language (DSL)

```
animal(A) :-
        tree(T),
        tree_animal(T, A).

tree_animal(animal(A), A).
tree_animal(if_then_else(Cond,Then,Else), A) :-
        (   is_true(Cond) ->
            tree_animal(Then, A)
        ;   tree_animal(Else, A)
        ).
```

Such trees can be interpreted in a straight-forward way

```
?- animal(A).
has fur?
        no.
has feathers?
        no.
false

?- animal(A).
has fur?
        yes.
says woof?
        no.
says meow?
        yes
A = cat
```