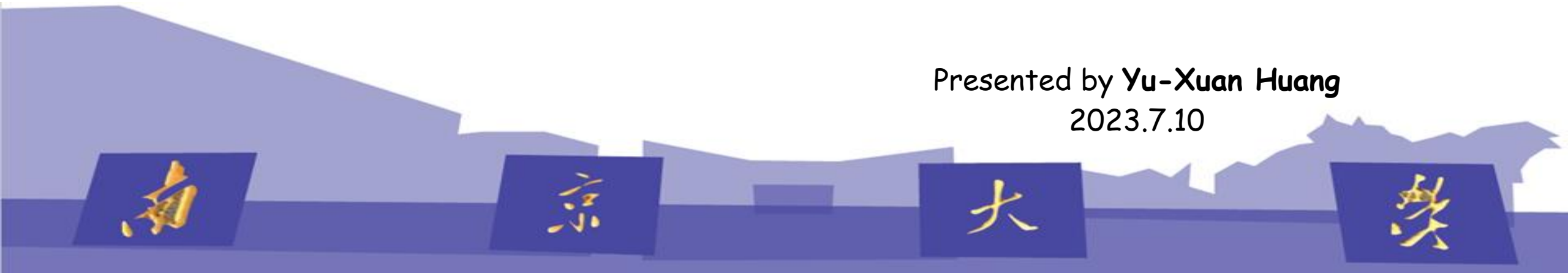
LAMDA
Learning And Mining from DatA

南京大學
NANJING UNIVERSITY
1902

(Meta-)Interpreters

Presented by **Yu-Xuan Huang**
2023.7.10



Interpretation is pervasive

Most programs are interpreters for specific languages.

Examples:

- a web browser interprets HTML, JavaScript, HTTP, ...
- 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

Meta-programs treat other programs as data. They analyze, transform, and interpret other programs.

An ***interpreter*** is a program that evaluates programs.

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

Why write MI: Create **special-purpose** interpreters

The idea of MI is used in Metagol

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 Simplest Meta-interpreters

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

or

```
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**

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 (Tracing)

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

```
member(X,[X|Xs]).
member(X,[Y|Ys]) ← member(X,Ys).
```

Program 3.12 Membership of a list

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

The results are the same, but we can manually control the process.

```
solve(member(X,[a,b,c]))
    clause(member(X,[a,b,c]),B)                    {X=a,B=true}
    solve(true)
        true          Output:  X=a
                               ;
    solve(true)
        clause(true,T)     f
    clause(member(X,[a,b,c],B)                     {B=member(X,[b,c])}
    solve(member(X,[b,c]))
        clause(member(X,[b,c]),B1)                 {X=b,B1=true}
        solve(true)
        true          Output:  X=b
                               ;
        solve(true)
            clause(true,T)     f
        clause(member(X,[b,c]),B1)                 {B1=member(X,[c])}
        solve(member(X,[c]))
            clause(member(X,[c],B2)                {X=c,B2=true}
            solve(true)
            true          Output:  X=c
                                   ;
            solve(true)
                clause(true,T)     f
            clause(member(X,[c],B2))               {B2=member(X,[ ])}
            solve(member(X,[ ]))
                clause(member(X,[ ]),B3)     f
                  no (more) solutions
```

Figure 17.2 Tracing the meta-interpreter

Alternative representations of clauses

Assume clause "**`A :- B1,B2,…,Bn`**" is represented by **`rule(A, [B1,…,Bn])`**

```
solve(Goal) ← solve(Goal,[ ]).

solve([ ],[ ]).
solve([ ],[G|Goals]) ← solve(G,Goals).
solve([A|B],Goals) ← append(B,Goals,Goals1), solve(A,Goals1).
solve(A,Goals) ← rule(A,B), solve(B,Goals).
```

Program 17.6 A meta-interpreter for pure Prolog in continuation style

Handle built-in predicates

Built-in predicates are not defined by clauses in the program and need different treatment.

```
builtin(A is B).          builtin(A > B).
builtin(read(X)).         builtin(write(X)).
builtin(integer(X)).      builtin(functor(T,F,N)).
builtin(clause(A,B)).     builtin(builtin(X)).
```

Figure 17.3 Fragment of a table of builtin predicates

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

Or use `predicate_property(G, built_in)`

Handle cut

- Handling cut correctly in a meta-interpreter is tricky
- Usually relies on technical details of the scope of cut in a particular implementation of Prolog.
- Refer to Art 17.4 , Craft 7.5-7.6

Examples

Example: A Simple Tracer

```
solve_trace(Goal) ← solve_trace(Goal,0).

solve_trace(true,Depth) ← !.
solve_trace((A,B),Depth) ←
    !, solve_trace(A,Depth), solve_trace(B,Depth).
solve_trace(A,Depth) ←
    builtin(A), !, A, display(A,Depth), nl.
solve_trace(A,Depth) ←
    clause(A,B), display(A,Depth), nl, Depth1 is Depth + 1,
    solve_trace(B,Depth1).

display(A,Depth) ←
    Spacing is 3*Depth, put_spaces(Spacing), write(A).

put_spaces(N) ←
    between(1,N,I), put_char(' '), fail.
put_spaces(N).

between(1,N,I) ← See Program 8.5.
```

Program 17.7 A tracer for Prolog

- The **display** is between **clause** and **solve_trace**, ensuring the goal is displayed **each time** backtracks to choose another clause.
- If the **clause** and **display** are swapped, only the **initial call** of the goal is displayed.

Example: Depth Limit

```
mi_limit(Goal, Max) :-
        mi_limit(Goal, Max, _).

mi_limit(true, N, N).
mi_limit((A,B), N0, N) :-
        mi_limit(A, N0, N1),
        mi_limit(B, N1, N).
mi_limit(g(G), N0, N) :-
        N0 #> 0,
        N1 #= N0 - 1,
        mi_clause(G, Body),
        mi_limit(Body, N1, N).
```

+

```
mi_id(Goal) :-
        length(_, N),
        mi_limit(Goal, N).
```

= iterative deepening

- '#>' '#=' are defined in CLP(FD)

Example: Reverse Order

Reverse the default execution order of goals:

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

Example: Building A Proof Tree

solve(Goal,Tree) ←
    Tree is a proof tree for *Goal* given the program defined by *clause/2*.

```
solve(true,true) ← !.
solve((A,B),(ProofA,ProofB)) ←
    !, solve(A,ProofA), solve(B,ProofB).
solve(A,(A←builtin)) ← builtin(A), !, A.
solve(A,(A←Proof)) ← clause(A,B), solve(B,Proof).
```

Program 17.8 A meta-interpreter for building a proof tree

?- solve(son(lot , haran) ,Proof)

```
Proof = (son(lot,haran) ←
         ((father(haran,lot)←true),
         (male(lot)←true))).
```

Explanation

Example: Uncertainty Reasoning

A logic program with certainties: a set of ordered pairs (Clause,Factor), where Clause is a clause and Factor is a certainty factor in [0,1].

solve(Goal,Certainty) ←
    Certainty is our confidence that *Goal* is true.

```
solve(true,1) ← !.
solve((A,B),C) ←
    !, solve(A,C1), solve(B,C2), minimum(C1,C2,C).
solve(A,1) ← builtin(A), !, A.
solve(A,C) ← clause_cf(A,B,C1), solve(B,C2), C is C1 * C2.

minimum(X,Y,Z) ← See Program 11.3.
```

Program 17.9 A meta-interpreter for reasoning with uncertainty

Example: Uncertainty Reasoning with Threshold

solve(Goal,Certainty,Threshold) ←
    Certainty is our confidence, greater than *Threshold*, that *Goal* is true.

```
solve(true,1,T) ← !.
solve((A,B),C,T) ←
    !, solve(A,C1,T), solve(B,C2,T), minimum(C1,C2,C).
solve(A,1,T) ← builtin(A), !, A.
solve(A,C,T) ←
    clause_cf(A,B,C1), C1 > T, T1 is T/C1,
    solve(B,C2,T1), C is C1 * C2.
minimum(X,Y,Z) ← See Program 11.3.
```

Program 17.10 Reasoning with uncertainty with threshold cutoff

- The new threshold is the quotient of the previous threshold by the certainty of the clause.

Other Examples

- Art 17, Craft 7, Power 18
- Reifying backtracking (Power)

Q & A

Thanks!