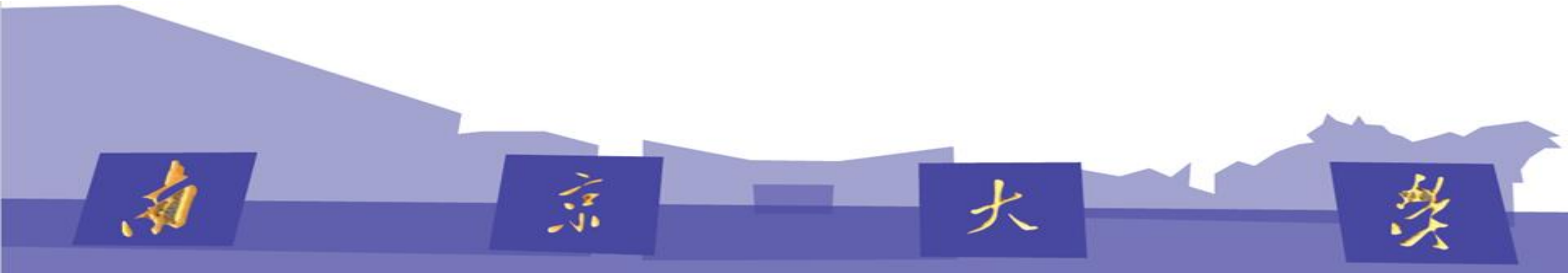
LAMDA
Learning And Mining from DatA

南京大學
NANJING UNIVERSITY 1902

Higher Order Programming in Prolog

Presented by Xin-Hao Zhu



Content

- Second-order in Logic & Programming
- Higher-order Predicates
- All-solution Predicates

Note that Prolog is rooted in **first-order logic** (FOL).

By *"higher-order"*, we mean …

- [√] extending the flexibility of Prolog programming via *higher-order predicates*
- [×] building Prolog based on syntax & semantics of second-order logic

Second Order Logic

First-order logic **quantifies** only variables that range over individuals

$$\forall x\ P(x)$$

Second-order logic, in addition, also **quantifies** over relations

e.g., *law of excluded middle (排中律):* $\forall P\ \forall x\ (P(x) \vee \neg P(x))$

With increased expressivity, it's generally hard to reason on second-order clauses

- Second-order logic (with standard semantics) does not admit a complete proof theory
- **First-order logic** is the strongest logic satisfying both *compactness* and *completeness*.
  - subject to certain restrictions; aka Lindström's theorem

Second-order in Prolog

From a mathematical perspective…

- ***First-order clauses*** describe relationships between entities, e.g., `like(x, y)`
- ***Second-order clauses*** describe relationships between relations

Functionally, **second-order predicates** accept predicates as its arguments

- similar to "passing function as argument" in Python

Second-order in Prolog cont.

From a computational perspective…

- ***First-order procedure*** operates over a single path of the tree structure
- ***Second-order procedure*** operates over multiple (or all) paths of the tree structure

Functionally, **second-order predicates** finds all proofs of a goal

- thus called *all-solution predicates*
- implemented based on other higher-order predicates

Meta Programming

Meta-programming is writing programs that manipulate programs.

- so-called “code as data”
- functionally, writing programs that **read, generate, or transform** other programs (or itself).

Examples:

- **Python**: run-time definition of **classes**, or **functions** (e.g., `exec()`)
- **Prolog**: run-time construction of **goals** when answering a query (e.g., `call()`)

Partial Goals

A **partial goal** is a **goal** to which **zero or more arguments are added** before it is called.

- similar to **partial function** in Python

Consider a predicate `like/2`, and the following **partial goals:**

- `like`
- `like(mike)`
- `like(mike, apple)`

A partial goal **`C`** called with **`N`** additional arguments is often denoted as **`C_N`**.

Mode Indicators

The **mode indicators** for program comments as follows

+ indicates an **input** argument.

- must be instantiated to a term

  satisfying some type specification

- but **not necessarily be ground**

: indicates a **meta** argument

- e.g., a term that can be called as partial goal

- indicates an **output** argument.

- may or may not be bound at call-time

```
[Def] sum(+X, +Y, -S)

[Use] ?- sum(1, 2, S)
```

```
[Def] call(:Goal, +ExtraArg1)

[Use] ?- call(male, mike)
```

```
[Def] bagof(+Template, :Goal, -Bag)

[Use] ?- Bagof(Y, like(X,Y), Ys)
```

call/N

`call(:Goal, +ExtraArg1, ...)`

- call a *partial goal* **dynamically**
- appends extra arguments to `Goal`, then call the result

```
?- call(plus, 1, 2, X).
X = 3.

?- call(plus(1), 2, X).
X = 3.
```

`call/N` is normally used for **goals that are not known at compile time**.

- The SWI-Prolog toplevel essentially performs `read(Goal), call(Goal).`

`call/N` is the **building block** of other meta-predicates.

maplist/N

maplist/2 and **maplist/3** are among the **most** frequently used meta-predicates.

`Maplist/2(Pred_1, Ls)`

- operates like `map(Pred_1, list)` in Python
- `true` **iff** `call(Pred_1, L)` is `true` for each element `L` in the list `Ls`.

`Maplist/3(Pred_2, As, Bs)`

- operates like `map(Pred_2, zip(As, Bs))` in Python
- `true` **iff** `call(Pred_2, A, B)` is `true` for each pair of elements `A, B` in `As, Bs`

`maplist/N` cont.

For example, we can state that an arbitrary term `E` is **different** from **all** elements in `Ls`

```
?- maplist(dif(E), Ls).
  Ls = []
; Ls = [_A], dif(E, _A)
; Ls = [_A, _B], dif(E, _A), dif(E, _B)
; Ls = [_A, _B, _C], dif(E, _A), dif(E, _B), dif(E, _C)
; … .
```

This works in **all directions**, even if `Ls` is not instantiated.

foldl/N

In the `foldl/N` family, the most frequently used predicate is `foldl/4`

`foldl(Pred_3, Ls, S0, S)` operates like Python `reduce()`

- describes a **fold** from the left of the list `Ls`
- where **`S0`** is the initial state
- and **`S`** is the final state

[Power] For each element `L` in `Ls` and intermediate state `Sn`, `call(Pred_3, L, Sn, Sn+1)` is invoked to relate the current list element `L` and intermediate state `Sn` to the **next** intermediate state `Sn+1`

`foldl/N` cont.

Calculate `sum` of elements in list

```
integer_sum(A, B, S) :- S is A + B.

list_sum(Ls, S) :- foldl(integer_sum, Ls, 0, S).
```

```
?- list_sum([2,1,3], S).
S = 6.
```

Calculate `max` of elements in list

```
integer_max(A, B, M) :- M is max(A, B).

list_max([L|Ls], M) :- foldl(integer_max, Ls, L, M).
```

```
?- list_max([2,1,3], S).
S = 3.
```

if_/3

The meta-predicate `if_/3` is a recent and important development of Prolog.

- It's popular in Prolog community since 2016, see this post
- Use `if_/3` in SWI-Prolog via module library(reif) (i.e., "refined-if")

`if_/3` combines desirable declarative properties with good performance in many situations.

```
hailstone(N, N).

hailstone(N0, N) :-
    N0 #= 2*N1,
    hailstone(N1, N).

hailstone(N0, N) :-
    N0 #= 2*_ + 1,
    N1 #= 3*N0 + 1,
    hailstone(N1, N).
```

```
hailstone(N, N).

hailstone(N0, N) :-
    R #= N0 mod 2,
    if_(R = 0,
        N0 #= 2*N1,
        N1 #= 3*N0 + 1),
    hailstone(N1, N).
```

All-solution Query: Example

A short **example** of all-solution predicates …

Database

```
like(alice, apple).
like(alice, pear).
like(bob, apple).
like(bob, banana).
```

[temporal representation]

Query A

```
?- like(X, Y).

X = alice,
Y = apple ;
X = alice,
Y = pear ;
X = bob,
Y = apple ;
X = bob,
Y = banana.
```

[spatial representation]

Query B

```
?- bagof(Y, like(X, Y), Ys).

X = alice,
Ys = [apple, pear] ;
X = bob,
Ys = [apple, banana].
```

All-solution Predicates

Frequently used ***all-solution predicates***:

1. **`bagof/3`**
2. `setof/3`
3. `findall/3`: the building block of `bagof/3` and `setof/3`

```
bagof(+Template, :Goal, -Bag)
```

[SWI-Prolog] backtrack over the alternatives of the free variables in `Goal`, unifying `Bag` with the corresponding alternatives of `Template`

Example: Setting

Consider the database:

```
has_course(nju, ai, ml).
has_course(nju, ai, alg).
has_course(nju, cs, alg).
```

To list courses available for each university & school …

```
SELECT Course
    FROM has_course(University, School, Course)
    GROUP BY University, School
```

bagof/3

Usage of `bagof(+Template, :Goal, -Bag)`

Database

```
has_course(nju, ai, ml).
has_course(nju, ai, alg).
has_course(nju, cs, alg).
```

Query

```
?- bagof(C, has_course(U, S, C), Cs).

U = nju,
S = ai,
Cs = [ml, alg] ;

U = nju,
S = cs,
Cs = [alg].
```

Existential Variables

`bagof/3` results are grouped by **free variables (**in goal)

However, if we want to **skip** some variables when grouping …

- E.g., find all courses in NJU, regardless of schools

Then, some variables can be declared as existential

- We only need to know a variable **exists**; we need not to know its value

Syntax: `bagof(+Template, +Var^Goal, -Bag)`

- Functionally, Prolog do not use Var to group results
- Essentially, Prolog do not bound Var when backtracking

Existential Variables cont.

Query **without** existential variable

```
?- bagof(C, has_course(U, S, C), Cs).

U = nju,
S = ai,
Cs = [ml, alg] ;

U = nju,
S = cs,
Cs = [alg].
```

Query **with** existential variable

```
?- bagof(C, S^has_course(U, S, C), Cs).

U = nju,
Cs = [ml, alg, alg].
```

Here, S is declared as an existential variable

- remains unbound while backtracking
- does not show in the result

Different Types of Variables

```
bagof(C, S^has_course(U, S, C), Cs)
```

1. Captured Variables:
   - appear in the *Template*
   - **presented as elements** in *Bag*
2. Existential Variables:
   - appear as *Var1^Var2^...^Goal*
   - **skipped** in backtracking; **not show** in proofs
3. Free Variables
   - appear in *Goal*, but not declared as existential
   - **bound** in backtracking; used to **group** proofs

findall/3

Functionally, `findall/3` is `bagof/3` with **all variables are existential**

```
?- bagof(C, U^S^has_course(U, S, C), Cs).
Cs = [ml, alg, alg].
```

```
?- findall(C, has_course(U, S, C), Cs).
Cs = [ml, alg, alg].
```

- Note: `findall/3` yields `true` if bag is empty, where `bagof/3` would yield `false`
- However, `bagof/3` is actually implemented based on `findall/3`

[SWI-Prolog] `findall/3` is equivalent to `bagof/3` with all free variables appearing in Goal scoped to the Goal with an existential (caret) operator ^, except that `bagof/3` fails when Goal has no solutions.

setof/3

`bagof/3` may return duplicate items in *Bag*

If you want to **remove duplicate entries**, use `setof/3` instead

```
?- bagof(C, S^has_course(U, S, C), Cs).

U = nju,
Cs = [ml, alg, alg].
```

```
?- setof(C, S^has_course(U, S, C), Cs).

U = nju,
Cs = [ml, alg].
```

- `bagof/3` sort and group proofs with `keysort/2`, which **retains duplicates**
- `setof/3` uses `sort/2` instead, so as to **remove duplicates**

Implementation of All-solution Predicates

In SWI-Prolog, `findall/3` is implemented by low-level **C code**

- searching in a while-loop
- collects all proofs of goal via backtracking

Mechanism of `bagof/3` and `setof/3`

1. **assign** existential and free variables
2. **call** `findall/3` to obtain all proofs for *Goal* without existential variables
3. **sort** proofs using free variables as keys
4. **group** the sorted list into blocks

```
bagof(Templ, Goal0, List) :-
    '$free_variable_set'(Templ^Goal0, Goal, Vars),
    (   Vars == v
    ->  findall(Templ, Goal, List),
        List \== []
    ;   alloc_bind_key_list(Vars, VDict),
        findall(Vars-Templ, Goal, Answers),
        bind_bagof_keys(Answers, VDict),
        keysort(Answers, Sorted),
        pick(Sorted, Vars, List)
    ).
```

Use All-solution Predicates with Caution

Each of these predicates implies **negation as failure**, implicitly or explicitly

The “not provable” predicate `(\+)/1` can be implemented via `findall/3`

```
\+ Goal :- findall(., Goal, []).
```

Negation as failure is **not** a **sound** implementation of logical negation (not)

- except when the goal is ground

Use all-solution predicates with caution, as they may destroy **monotonicity** of program

- after adding a clause, goals that previously succeeded may fail