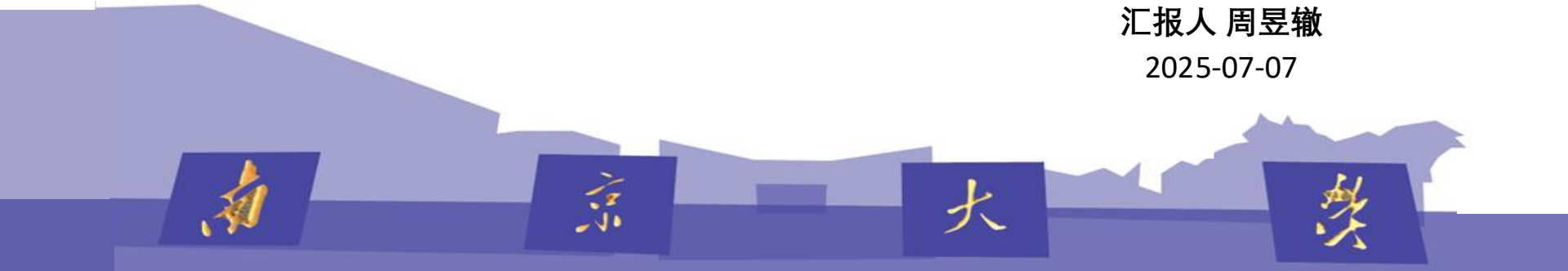
第三次 Prolog 读书班

汇报人 周昱辙

2025-07-07



目录

一阶谓词

我们通常编写的谓词

- 一阶谓词的参数是**数据项 (Terms)**，例如原子、数字、列表或结构。

示例:

```
Parent(john, marry).
Append([a,b], [c,d], L).
Member(X, [1,2,3]).
```

- 在这些例子中，`john`, `[a,b]`, `X` 等分别是原子、列表和变量。
- 谓词操作的是**数据**。

高阶谓词

定义

一个**高阶谓词**是一个以其他谓词作为其参数的谓词。

- 它操作的是**行为**或**逻辑**，而不仅仅是数据。
- **核心机制: $call/N$**
- Prolog内置的 $call/N$ 谓词允许我们将一个项作为目标来执行。

```
?- Goal = member(X, [a,b,c], call(Goal)).
Goal = member(a, [a,b,c]), X = a;
Goal = member(b, [a,b,c]), X = b;
Goal = member(c, [a,b,c]), X = c.
```

maplist/N

目的: 将一个操作（谓词）应用于列表中的每一个元素。

示例: 将列表中的每个原子转换为其长度。

```
?- maplist(atom_length, [prolog, is, powerful], Lengths).
Lengths = [6,2,8].
```

示例: 检查列表中的所有元素是否都是数字。

```
?- maplist(number, [1, 2, 3]).
true.
```

*map_list/3*的实现

```
our_maplist(_, [], []).
our_maplist(Pred, [H1|T1], [H2|T2]) :-
    Goal =.. [Pred, H1, H2],
    call(Goal),
    our_maplist(Pred, T1, T2).
```

- `=..`(univ) 操作符用于将一个目标解构或构建成一个列表。
- successor(1, 2) =.. [successor, 1, 2]

```
?- our_maplist(atom_length, [prolog, is, powerful], Lengths).
Lengths = [6,2,8].
```

findall/3

目的: 将一个目标的所有可能解收集到一个列表中。

`findall(Template, Goal, List)`

- 找到使 Goal 成功的所有方式，并根据 Template 收集结果。

示例: 找到一个家庭中的所有孩子。

```
parent(john, mary).
parent(john, tom).
parent(susan, mary).

?- findall(Child, parent(john, Child), Children).
Children = [mary,tom].
```

foldl/4

目的: 将一个列表“折叠”或“归约” (reduce) 成一个单一的值。
foldl(Predicate, List, InitialValue, Result)

核心思想: 它像滚雪球一样工作。
1.从一个**初始值**（小雪球）开始。
2.遍历列表，每次都将当前元素与“雪球”结合，形成一个**新的、更大的雪球**。
3.列表遍历完毕后，最后的“雪球”就是最终结果。

示例: 计算列表所有元素的和。

```
plus(Elem, AccIn, AccOut) :-
    AccOut is AccIn + Elem.
?- foldl(plus, [10, 20, 30], 0, Sum).
Sum = 60.
```

使用高阶谓词的优势

抽象 (Abstraction)
- 隐藏了递归、回溯和结果收集的细节。

代码重用 (Code Reuse)
- 编写一次 maplist，可以在任何需要列表映射的地方使用。减少了重复的“样板代码”。

声明性 (Declarativeness)
- 代码更贴近于描述“做什么”，而不是“如何一步步地做”。
- maplist比手写递归循环更清晰地表达了意图。

减少错误 (Fewer Bugs)
- 通用的、经过良好测试的高阶谓词（如库函数）通常比我们自己为每个特定任务编写的递归代码更可靠。

目录

Pure predicate

定义

一个**纯粹的**Prolog谓词（predicate）具有以下特征:

1.逻辑一致性: 其成功或失败仅取决于其参数的逻辑定义。

2.无副作用 (Side Effects): 它不会做任何超出计算结果之外的事情。

1. 无输入/输出 (I/O) 操作。
2. 不修改数据库 (assert/retract)。
3. 不改变任何外部状态。

纯粹谓词的特性

引用透明性 (Referential Transparency)

- 一个表达式（如 `member(a, [a,b,c])`）可以用其结果（true）替换，而不改变程序的逻辑。

顺序无关性 (Commutativity)

- `goal1, goal2` 的逻辑结果应该和 `goal2, goal1` 相同。

多向性 (Multi-directionality)

- 可以用于不同的参数模式。
- 例如，`append(L1, L2, [a,b,c])` 可以用来：
  - 检查 (L1, L2 已知)。
  - 连接 (L1, L2 已知)。
  - 分解 (L3 已知)。

纯粹谓词示例member/2

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

多向性使用:

```
?- member(b, [a,b,c]).
true.
?- member(X, [a,b,c]).
X = a ; X = b ; X = c.
```

顺序无关:

`member(X, [a,b]), number(X)` 与 `number(X), member(X, [a,b])` 在逻辑上等价。

不纯粹谓词

不纯 = "超逻辑" (Extra-logical) 谓词

当一个谓词的行为超出纯粹的逻辑领域时，它就是不纯的。

主要来源:

1.输入/输出 (I/O): *write/1*, *read/1*

2.数据库修改: *asserta/1, assertz/1*

3.执行控制: *!/0* (the Cut), *->/2* (if-then-else)

4.元逻辑 (Meta-logical): *var/1, nonvar/1*

这些谓词检查参数的当前状态，而不是它们的逻辑内容。

不纯谓词: The Cut *(!/0)*

Cut 的作用: 修剪搜索树。

Green Cut: 提升效率。

- 用于修剪那些我们从逻辑上已知必然会失败的分支。
- 不改变程序的逻辑结果。
- 示例: 在函数式谓词中，一旦找到一个匹配的子句，就无需尝试其他子句。

Red Cut: 改变逻辑。

- 修剪那些**可能包含解**的分支。
- 使程序不完整。
- 强烈依赖于子句的顺序。

! 是Prolog中最强大也最容易被误用的工具。

相同谓词的不纯粹与纯粹实现

目标：计算一个列表中正数数量。

```
:- dynamic counter/1.

count_positives_impure(List, Count) :-
    % 1. 初始化计数器
    asserta(counter(0)),
    % 2. 调用辅助谓词遍历列表
    process_list(List),
    % 3. 收回最终计数值并清理
    retract(counter(Count)).
```

```
% 辅助谓词，遍历列表并修改计数器
process_list([]).
process_list([H | T]) :-
    ( H > 0 ->
        retract(counter(Old)),
        New is Old + 1,
        asserta(counter(New))
    ; true
    ),
    process_list(T).
```

相同谓词的不纯粹与纯粹实现

目标：计算一个列表中正数数量。

```
count_positives_pure(List, Count) :-
    count_positives_acc(List, 0, Count).

count_positives_acc([], Acc, Acc).

count_positives_acc([H | T], AccIn, FinalCount) :- H > 0,
    NewAcc is AccIn + 1,
    count_positives_acc(T, NewAcc, FinalCount).

count_positives_acc([H | T], AccIn, FinalCount) :- H =< 0,
    count_positives_acc(T, AccIn, FinalCount).
```

为什么使用不纯谓词？

- **与世界互动**: 现实世界的程序需要处理I/O。
- **效率**: Cut和数据库操作可以用来实现缓存（记忆化）或避免重复计算。
- **控制**: 实现具有确定性行为的控制流
- **表达状态**: 模拟状态变化，尽管通常有更纯粹的替代方案（如传递状态变量）。

目录

一个看似正确的程序

计算列表长度

```
list_length([], 0).
list_length([_|Ls], N) :-
    N #> 0,
    N #= N0 + 2,
    list_length(Ls, N0).
```

它在某些情况下似乎能正常工作:

- 基本情况成功: `?- list_length([], 0).` -> **true.**
- 它可以生成列表: `?- list_length(Ls, _).` -> `Ls = [], Ls = [_], ...`

但这些简单的检查还不够。这个程序是**错误**的。

终止性

测试期望的属性

我们不只测试答案，更要测试**属性**。

1. **终止性(Termination):** 我们知道 *list_length/2* 应该能生成无限的列表，所以最通用的查询应该**不会**终止。

我们可以用查询 `?- Q, false.` 来测试。只有当Q通用终止时，这个查询才会终止。

```
?- list_length(Ls, L), false. % 程序无限运行
```

程序正确地表现出了不终止的行为，因此它通过了这项属性测试。但这仍然不足以发现错误。

寻找反例

寻找反例

我们可以用具体的例子或一个可信的参考实现来进行测试。

1. 具体测试用例: 我们可以让Prolog找出一个由a构成的列表，使得我们的谓词对其**不**成立。

```
?- maplist(=(a), Ls), \+ list_length(Ls, _).
% 未找到反例。
```

2. 参考built-in实现:

```
?- length(Ls, A), list_length(Ls, B), A #\= B.
A = 1, B = 2, Ls = [_] ; ...
```

这表明，对于长度为1的列表，我们的谓词计算出的长度是2。

测试谓词基本属性

测试“长度”的一个基本属性

如果没有参考实现怎么办？可以利用关系本身的一个已知**属性**。

属性: 如果一个列表 Ls 的长度是 N，那么 [_|Ls] 的长度必须是 N+1。

搜索这个属性的一个反例：

```
?- list_length(Ls, N), list_length([_|Ls], N1), N1 #\= N + 1.
Ls = [], N = 0, N1 = 2
;…
```

目录

tracing不是最佳选择

习惯于过程式语言的程序员，可能会想到用 *trace* 来调试。

```
?- trace, your_goal.
```

但这种方法有几个严重缺点：

1.tracers本身也有错误

2.trace过程迅速变得复杂：对于稍复杂的程序，trace信息会变得难以阅读和理解。

3.鼓励过程化思维：过度依赖trace会阻碍学习更优雅、更强大的声明式调试技术。

我们的目标：关注“为什么”出错，而不是“如何”执行。

错误的两种类型

一个纯粹的Prolog程序可能犯两种错误

1.过于泛化 (Too General)

1. **症状**: 产生了不应该有的解。
2. **修正**: 需要添加约束。

2.过于特化 (Too Specific)

1. **症状**: 缺失了本应该有的解。
2. **修正**: 需要将程序泛化。

我们的 list_length/2 例子

它同时犯了两种错误:

- **过于泛化**: ?- list_length([_], 2). -> true. (错误地成功)
- **过于特化**: ?- list_length([_], 1). -> false. (错误地失败)

*泛化

一种定位错误的声明式技术

由 Ulrich Neumerkel 提出，核心是使用 * 将目标“泛化掉”。

1. 定义操作符:

```
:- op(920,fy, *).
*_.
```

2. 调试步骤:

- 我们从一个**错误地失败**的查询开始: `?- list_length([_], 1). -> false.`
- 用*注释掉所有子句体中的目标，使其变得**足够泛化**以至于查询能够成功:

```
list_length([_|Ls], N) :-
* N #> 0,
* N #= N0 + 2,
* list_length(Ls, N0).
```

- **逐个恢复**目标，直到查询再次失败。导致失败的那个目标就是错误的根源。
  - 当我们恢复到 N #= N0 + 2 时，查询再次失败。**因此，找到错误**

*泛化

技术背后的基本原则

移除一个目标，最多只会让一个谓词变得**更泛化**，绝不会使其变得更特化。

- 这个属性（称为**单调性 Monotonicity**）对于pure的Prolog代码成立。
- 这是Prolog语言一个几乎独一无二的强大特性。