数理逻辑

08 - 一阶逻辑的语法

(Press ? for help, n and p for next and previous slide)

戴望州

南京大学智能科学与技术学院

2026年 - 春季

https://daiwz.net

前情提要

命题逻辑

- 命题逻辑的语法
- 命题逻辑的语义
- 命题逻辑的公理系统和推理
- 命题逻辑的可靠性
- 命题逻辑的完备性
  - 一致性
  - 紧致性
  - 可满足性
  - Lindenbaum引理

语法

一阶语言

什么是**一阶（first-order）**？

一阶语言

逻辑符号

1. 标点符号：“(”，“)”，“$,$”
2. 联结词：$\to$，$\neg$
   - » 其它联结词：$\wedge, \vee, \leftrightarrow$
3. 变元（可数无穷个）：$\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots$
4. 量词：$\forall$
   - » 其它量词：$\exists$
5. 等词（可选）：$=$

非逻辑符号 （signature）

1. n元谓词（可数无穷个），$n \geq 1$：$A_0^n, A_1^n, \ldots$
2. 常元（可数无穷个）：$c_0, c_1, \ldots$
   - » 特殊常元：$\top, \bot, T, F$
3. n元函数（可数无穷个）：$f_0^n, f_1^n, \ldots$

一阶语言的例子

纯一阶逻辑（*pure First-Order Logic, FOL*）

常元	$c_0, c_1, \ldots$
n元谓词	$A_0^n, A_1^n, \ldots$
n元函数	$f_0^n, f_1^n, \ldots$
等词	无

一阶语言的例子

初等数论

常元	$\mathbf{0}$
谓词	$<$
1元函数	$\mathbf{S}$
2元函数	$+, \times, \mathbf{E}$
等词	有

- 注：$\mathbf{S}, \mathbf{E}$分别为后继和指数函数

FOL之于数学

因为：

- 一阶语言包含集合论语言
- 一切数学都可被嵌入在集合论中

所以：

1. 集合论的**一阶语言**能够描述任意数学
2. 一切数学定理都来自对集合论公理的**逻辑推导**

FOL 的例子 [Enderton, pp.73]

项

定义 2.1（项，*term*）[Enderton, pp.74]：

每个n元函数符号f对应一个n元项构造算子$\mathcal{F}_f$：

$$\mathcal{F}_f(\epsilon_1, \cdots, \epsilon_n) = f\epsilon_1 \cdots \epsilon_n$$

一阶逻辑的项的集合是常元和变元符号经过（0次或多次）$\mathcal{F}_f$运算得到的表达式集合

原子公式

定义 2.2（原子公式，*atomic formula*，*atom*）[Enderton, pp.74]：

一阶逻辑的原子公式是如下形式的表达式

$$Pt_1 \cdots t_n$$

其中 P 是 n 元谓词，$t_1, \ldots, t_n$ 是项

合式公式

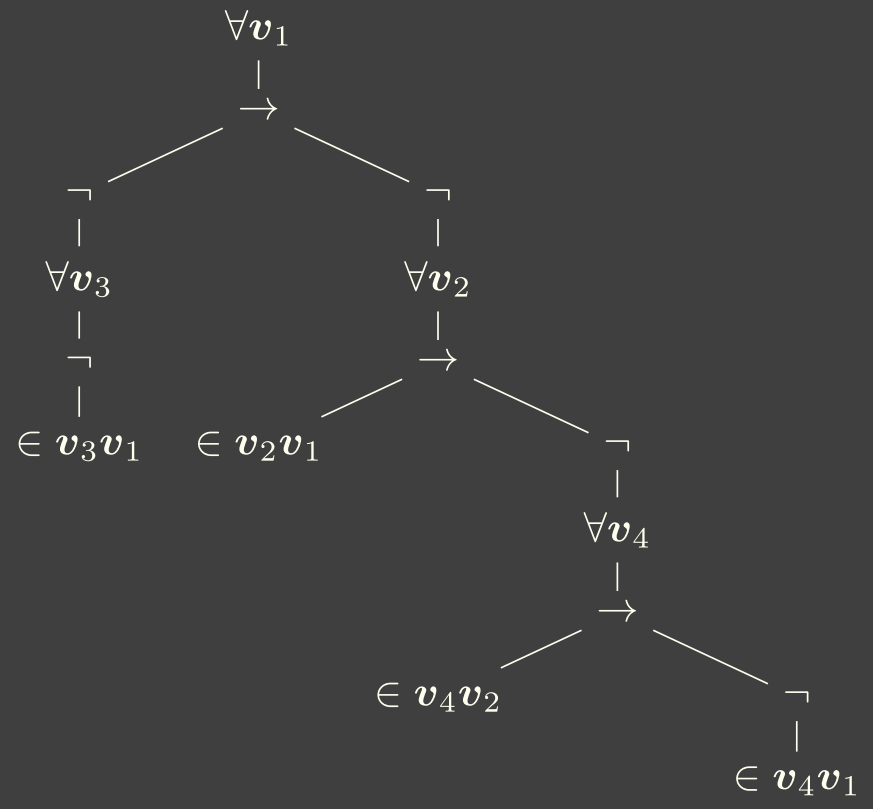
定义 2.3（合式公式，*well-formed formula*，*wff*）[Enderton, pp.75]：

一阶逻辑的合式公式集合是原子公式通过运用 0 次或多次公式构造算子 $\mathcal{E}_{\neg}, \mathcal{E}_{\leftarrow}, \mathcal{Q}_i (i = 1, 2, \ldots)$ 形成的表达式集合。其中

$$\begin{aligned} \mathcal{E}_{\neg}(\gamma) &= (\neg\gamma) \\ \mathcal{E}_{\leftarrow}(\gamma, \delta) &= (\gamma \leftarrow \delta) \\ \mathcal{Q}_i(\gamma) &= \forall \boldsymbol{v}_i \gamma \end{aligned}$$

合式公式

$$\forall \boldsymbol{v}_1((\neg\forall \boldsymbol{v}_3(\neg \in \boldsymbol{v}_3\boldsymbol{v}_1)) \to (\neg\forall \boldsymbol{v}_2(\in \boldsymbol{v}_2\boldsymbol{v}_1 \to (\neg\forall \boldsymbol{v}_4(\in \boldsymbol{v}_4\boldsymbol{v}_2 \to (\neg \in \boldsymbol{v}_4\boldsymbol{v}_1))))))$$



自由变元

- $\forall \boldsymbol{v}_2 \in \boldsymbol{v}_2 \boldsymbol{v}_1$
- $(\neg \forall \boldsymbol{v}_1 (\neg \forall \boldsymbol{v}_2 \in \boldsymbol{v}_2 \boldsymbol{v}_1))$

自由变元

$$\varphi \equiv \sum_{j=0}^{k} a_j$$

自由变元

定义 2.4（自由出现，*occur free*）[Enderton, pp.76]：

考虑一个变元x，我们递归地定义：

1. 对于原子公式α，x在α中自由出现当且仅当x在α中出现
2. x在$(\neg\alpha)$中自由出现当且仅当x在α中自由出现
3. x在$(\alpha \to \beta)$中自由出现当且仅当x在α中自由出现或在β中自由出现
4. x在$(\forall v_i \alpha)$中自由出现当且仅当x在α中出现且$x \neq v_i$

若一个变元如果不是自由（*not free*）的，则我们称它为（受）约束的（*bounded*）。若一个wff没有自由出现的变元，则称它是闭公式（*closed wff*）或语句（*sentence*）

变元替换

$$(\varphi)_2^k = (\sum_{j=0}^{k} a_j)_2^k = \sum_{j=0}^{2} a_j$$

变元替换

$$(\varphi)_i^j = (\sum_{j=0}^{k} a_j)_i^j = \sum_{i=0}^{k} a_i$$

变元替换

定义2.5a（替换，*substitution*）[Enderton, pp.112]

wff中的变元替换（substitution）可递归地定义如下：

1. 对任意原子公式α，α_t^x是用项t代替α中出现的所有x后所得的表达式
2. $(\neg\alpha)_t^x = (\neg\alpha_t^x)$
3. $(\alpha \to \beta)_t^x = (\alpha_t^x \to \beta_t^x)$
4. $(\forall y\,\alpha)_t^x = \begin{cases} \forall y\,\alpha, & \text{if } x = y \\ \forall y\,(\alpha)_t^x, & \text{if } x \neq y \end{cases}$

我们记这个替换算子为$\theta = [t/x]$，则$\alpha_t^x = \alpha \circ \theta = \alpha\theta$

变元替换

$$[\sum_{j=0}^{n}(k \cdot a_j)]_{f(j)}^{k} \stackrel{?}{=} \sum_{j=0}^{n}(f(j) \cdot a_j)$$

可替换

定义2.5b（可替换，*substitutable*）[Enderton, pp.112]

我们定义项t对于α中的变元x是可替换的（substitutable）如下：

1. 对任意原子公式α，项t对于α中出现的所有x都是可替换的（原子公式无量词）
2. 项t对于$(\neg\alpha)$中的x是可替换的，当且仅当它对α中出现的x是可替换的
3. 项t对于$(\alpha \to \beta)$中的x是可替换的，当且仅当它对α和β中出现的x均是可替换的
4. 项t对于$\forall y\,\alpha$中的x是可替换的当且仅当：
   - x在$\forall y\,\alpha$中是约束出现的（为保证替换后不影响原公式意义），或者
   - y在t中未出现（为保证t中的变元不在替换后被$\forall y$量化）且t对于α中的x是可替换的

替换的例子

一阶逻辑的Hilbert系统里有一条公理模式：

$$\forall x\, \alpha \to \alpha_t^x$$

其中t对于α中的x是可替换的，那么：

一阶逻辑的片段：Prolog

Horn子句

至多只有一个肯定文字的**子句**

$$H \vee \neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_n = H \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

其中H和B_i均为**原子公式**

Prolog规则

在逻辑程序（**Pro**gramming with **Log**ic, Prolog）中，我们用：

- `":-"`（“colon dash”）代替“$\leftarrow$”
- `","`代替“$\wedge$”
- `"."`代表句子完结

Prolog 的例子

```
% 曹操：
%   车胄有八万精兵驻防徐州，八万呢！
%   你就算是八万个馒头，刘备也得啃上半个月！
%   怎么可能说丢就丢？
%
% 荀彧：
%   完全有可能
%   刘备手里有曹操颁给的兵符。
%   车胄见了兵符，以为刘备是曹操派来的。
%   因此刘备可能骗开徐州城门。

% ---------- 事实 ----------

stationed('车胄', '徐州').
elite_troops('车胄', 80000).

has_token('刘备', '曹操兵符').
issued_by('曹操兵符', '曹操').
guard_general('徐州', '车胄').
loyal_to('车胄', '曹操').
```

N-QUEENS


```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   N Queens animation.

   https://www.metalevel.at/queens/
   ================================

   Written Feb. 2008 by Markus Triska (triska@metalevel.at)
   Public domain code. Tested with Scryer Prolog.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

:- use_module(library(clpz)).
:- use_module(library(lists)).
:- use_module(library(format)).
:- use_module(library(dcgs)).
:- use_module(library(freeze)).
:- use_module(library(charsio)).

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Constraint posting.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

Sudoku

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Solving Sudoku with Prolog
   https://www.metalevel.at/sudoku/

   Written Feb. 2008 by Markus Triska  (triska@metalevel.at)
   Public domain code. Tested with Scryer Prolog.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

:- use_module(library(clpz)).
:- use_module(library(lists)).
:- use_module(library(format)).
:- use_module(library(dcgs)).
:- use_module(library(freeze)).
:- use_module(library(charsio)).

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Constraint posting
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
```

元解释器

- **解释器**（interpreter）：“A program that evaluates programs”
- **元解释器**（meta-interpreter）：可以用于解释自身实现语言构成的程序
  - 我们可以用Prolog写一个解释Prolog语言的程序！

为什么是Prolog?

1. Prolog是一种同像性（homoiconic）语言
   - Prolog程序就是Prolog terms
2. Prolog有许多解释器能利用的隐式功能
   - 基于SLD-归结的深度优先搜索
3. Prolog语句非常简单
   - `Head :- Body`

元调用（Meta-call）

若我们有一个程序，

```
natnum(0).
natnum(s(X)) :-
    natnum(X).
```

Prolog能够动态地运行它：

```
?- Goal = natnum(X), call(Goal).
Goal = natnum(0), X = 0 ;
Goal = natnum(s(0)), X = s(0) ;
Goal = natnum(s(s(0))), X = s(s(0)) ;
Goal = natnum(s(s(s(0)))), X = s(s(s(0))) ;
Goal = natnum(s(s(s(s(0))))), X = s(s(s(s(0)))) ;
...
```

自我解析

也叫做自省（reflection and introspection）

若我们有一个程序，

```
:- dynamic complicated_clause/1. % 需要声明这是公共谓词

complicated_clause(A) :-
    goal1(A),
    goal2(A),
    goal3(A).
complicated_clause(1).
```

Prolog能够用`clause/2`解析它：

```
?- clause(complicated_clause(Z), Body).
Body =  (goal1(Z), goal2(Z), goal3(Z)) ;
Z = 1, Body = true.
```

最基础的元解释器

通过meta-call和`clause/2`实现SLD-归结：

```
prove(true).
prove((A,B)) :-
    prove(A),
    prove(B).
prove(Goal) :-
    Goal \= true,
    Goal \= (_,_),
    clause(Goal, Body),
    prove(Body).
```

用在`natnum/1`中：

```
?- prove(natnum(X)).
X = 0 ;
X = s(0) ;
X = s(s(0)) ;
... .
```

最基础的元解释器

通过meta-call和`clause/2`实现SLD-归结：

```
prove(true).
prove((A,B)) :-
    prove(A),
    prove(B).
prove(Goal) :-
    Goal \= true,
    Goal \= (_,_),
    clause(Goal, Body),
    prove(Body).
```

解释不了自己：

```
?- prove(prove(natnum(X))).
ERROR: No permission to access private_procedure `(\=)/2'
```

自定义语法

比如，新定义一个谓词`head_body(Head, Goals)`

```
head_body(natnum(0), []).
head_body(natnum(s(X)), [natnum(X)]).
```

新的SLD-归结元解释器：

```
prove([]).
prove([G|Gs]) :-
    head_body(G, Goals),
    prove(Goals),
    prove(Gs).
```

自定义语法2

利用list difference定义谓词`head_body(Head, Goals0, Goals)`

- `Head`成立当且仅当`Goals0-Goals`成立

```
head_body(natnum(0), Rs, Rs).
head_body(natnum(s(X)), [natnum(X)|Rs], Rs).
```

新的SLD-归结元解释器：

```
prove([]).
prove([G|Gs]) :-
    head_body(G, Goals, Gs),
    prove(Goals).
```

解释解释器的解释器

`prove/1`可以表达为：

```
head_body(prove([]),Rs,Rs).
head_body(prove([G|Gs]),[head_body(G,Goals,Gs),prove(Goals)|Rs], Rs).
```

`head_body/3`自己也可以表达为：

```
head_body(head_body(Head,Goals0,Goals), Rs, Rs) :-
    head_body(Head, Goals0, Goals).
```

解释解释器的解释器

```
% Meta-Interpreter
prove([]).
prove([G|Gs]) :-
    head_body(G, Goals, Gs),
    prove(Goals).

head_body(prove([]),Rs,Rs).
head_body(prove([G|Gs]),[head_body(G,Goals,Gs),prove(Goals)|Rs],Rs).

head_body(head_body(Head,Goals0,Goals), Rs, Rs) :-
    head_body(Head, Goals0, Goals).

% Program
head_body(natnum(0), Rs, Rs).
head_body(natnum(s(X)), [natnum(X)|Rs], Rs).

?- prove([prove([natnum(X)])]).
X = 0 ;
X = s(0) ;
X = s(s(0)) ;
X = s(s(s(0))) .
```

小结

一阶逻辑的语法

- 量词
- 变元
- 替换
- 逻辑程序