2025 Lean 与数学形式化讲义(1C)

上海交通大学 AI4MATH 团队

2 一阶形式逻辑

2.1 一阶逻辑算子

我们以一种实用主义的姿态来认识一阶形式逻辑系统, 不去纠结具体算子的含义, 而主要关心如何使用它们来构造命题, 以及在 Lean 中如何处理这些算子来进行形式化证明.

这其中的原因是, Lean 在依值类型论框架下实现一阶形式逻辑, 这导致一些算子在 Lean 中的语义与传统的一阶逻辑不完全相同, 例如全称量词算子的实质是依值函数类型算子, 量化的对象不一定是命题, 可以是任何表达式; 而存在量词则是通过依值函数与归纳类型构造的依值对, 所以存在量词的实现 `Exists` 是有类型的类型论对象, 而全称量词则没有类型. 这其中的一致性通过 Curry-Howard 同构来解释.

无论如何, 我们先脱离语义地认识一下一阶逻辑算子. 这些算子多数使用 Unicode 字符表示, 通过鼠标悬浮可以从语言服务器协议返回的信息中查看它们的输入方式:

概念 2.1.1 蕴含 (Implication)
设 p, q 是命题, 则 $p \to q$ 是命题, 称为命题 p 与 q 的**蕴含**.

概念 2.1.2 等价 (Equivalence)
设 p, q 是命题, 则 $p \leftrightarrow q$ 是命题, 称为命题 p 与 q 的**等价**.

概念 2.1.3 合取 (Conjunction)
设 p, q 是命题, 则 $p \wedge q$ 是命题, 称为命题 p 与 q 的**合取**.

概念 2.1.4 析取 (Disjunction)
设 p, q 是命题, 则 $p \vee q$ 是命题, 称为命题 p 与 q 的**析取**.

概念 2.1.5 否定 (Negation)
设 p 是命题, 则 $\neg p$ 是命题, 称为命题 p 的**否定**.

概念 2.1.6 谓词 (Predicate)
设 α 是类型, 则 α 上的一元谓词是一个类型为 $p : \alpha \to \text{Prop}$ 的函数.

概念 2.1.7 全称量化 (Universal Quantification)
设 α 是类型, p 是 α 上的一元谓词, 则 $\forall x : \alpha, p(x)$ 是命题, 称为 p 的**全称量化**.

概念 2.1.8 存在量化 (Existential Quantification)
设 α 是类型, p 是 α 上的一元谓词, 则 $\exists x : \alpha, p(x)$ 是命题, 称为 p 的**存在量化**.

2.2 自然语言陈述的形式化

相信大部分同学初次正式接触一阶形式逻辑 (主要是量词) 是在数学分析中的 $\varepsilon-\delta$ 语言, 往往从数列极限讲起. Mathlib 中的数列极限定义比较复杂, 我们不妨用数学分析课程中学习的的 $\varepsilon-\delta$ 语言来写一个简单版本的实数列极限的定义. 为了保证不与 Mathlib 中已有的内容冲突, 我们将定义放在一个新的**命名空间 (Namespace)**中.

语法 2.2.1 命名空间 (Namespace)
在 Lean 中, 使用 `namespace` 关键字可以创建一个新的命名空间, 以避免与已有的内容冲突.
在命名空间中定义的内容可以通过 `MyAnalysis.` 前缀来访问, 也可以通过 `open` 关键词打开命名空间直接访问:

```
namespace MyNamespace
def a (α : Type*) : α := sorry
#check a
end MyNamespace
#check a                    -- error
#check MyNamespace.a
open MyNamespace
#check a
```

定义 2.2.2 实数列极限 (Limit of a Real Sequence)
设 a_n 是一个实数序列, a 是一个实数, 称 a_n 的**极限**为 a, 记作 $\lim\limits_{n\to+\infty} a_n = a$, 当且仅当:

$$\forall \varepsilon > 0, \exists N : \mathbb{N}, \forall n > N, -\varepsilon < a_n - a < \varepsilon$$

$\forall \varepsilon > 0, P(\varepsilon)$ 其实是 $\forall \varepsilon : \mathbb{R}, \varepsilon > 0 \to P(\varepsilon)$ 的简写; $\exists N > 0, P(N)$ 是 $\exists N : \mathbb{N}, N > 0 \wedge P(N)$ 的简写. Lean 中也支持这种写法, 但在使用证明策略进行处理时要注意这一点.
定义"数列极限"这个概念时, 首先注意到在指定了变量 a_n 作为实数列, 变量 a 作为实数的语境下, $\lim\limits_{n\to+\infty} a_n = a$ 是一个命题, 这说明数列极限是一个实数列类型与实数类型上的二元谓词.
实数序列在 Lean 中表示为类型为 $\mathbb{N} \to \mathbb{R}$ 的函数, 实数的类型为 $\mathbb{R}$, 二元谓词的返回值类型是命题类型 `Prop`, 命题的内容是上文用一阶逻辑算子写成的公式. 于是我们可以用 Lean 代码写出实数列极限的定义:

```
def lim (a_ : ℕ → ℝ) (a : ℝ) : Prop :=
  ∀ ε > 0, ∃ N : ℕ, ∀ n > N, -ε < a_ n - a ∧ a_ n - a < ε
#check lim (fun n => 1 / n) 0        --  Prop
```

定理 2.2.3 实数列极限唯一 (Uniqueness of Limit of a Real Sequence)
设 a_n 是一个实数序列, a, b 是实数, 如果 $\lim\limits_{n\to+\infty} a_n = a$, $\lim\limits_{n\to+\infty} a_n = b$, 那么 $a = b$.
对上述自然语言进行分解, 发现这个命题有三个变量参数, 分别是实数序列 a_n (类型为 $\mathbb{N} \to \mathbb{R}$), 和两个实数 a, b (类型为 $\mathbb{R}$), 这些信息都会被包含在假设中, 可以隐式声明; 还有两个假设参数, 分别是 $\lim\limits_{n\to+\infty} a_n = a$ 和 $\lim\limits_{n\to+\infty} a_n = b$ 两个命题的证明; 结论是 $a = b$, 分别将它们形式化, 得到:

```
theorem lim_uniq
  {a_ : ℕ → ℝ}
  {a b : ℝ}
  (ha : lim a_ a)
  (hb : lim a_ b)
: a = b
:= by sorry
```

对于较为复杂的数学陈述, 我们可能很难直接观察得到它们的句法结构, 甚至许多时候由于自然语言自身存在大量缺漏或冗余信息, 这些陈述本身可能就含有歧义或者极为模糊, 难以找到它们的形式语言表达. 但我们可以通过对语言分块提取主算子的方式进行形式化:

设 $(a_n$ 是一个实数序列$)_1$, $(a, b$ 是实数$)_2$, 如果 $\left(\lim_{n\to+\infty} a_n = a\right)_3$, $\left(\lim_{n\to+\infty} a_n = b\right)_4$, 那么 $(a = b)_5$.

我们可以将上述子块表达为一个树结构, 称为**算子树 (Operator Tree)**. 由于 Lean 语言声明函数与依值函数时有 Curry 链和变量声明两种语法, 我们可以画出两种结构:

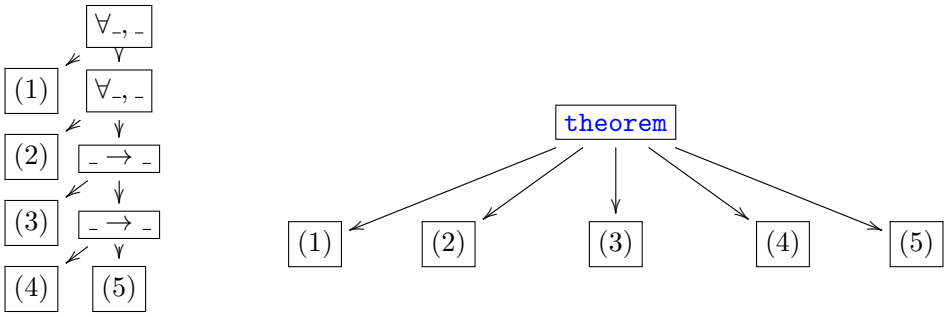

图 1: 初步复原得到的算子树框架

这两种结构本质上差不多, 前者是对象的类型, 后者是函数声明时的表达式, 但是我们一般更习惯后一种, 得到的算子树也更加美观. 不断递归地对每个子块分块并寻找主算子, 我们可以通过复原完整的算子树来得到自然语言陈述的形式化表达:

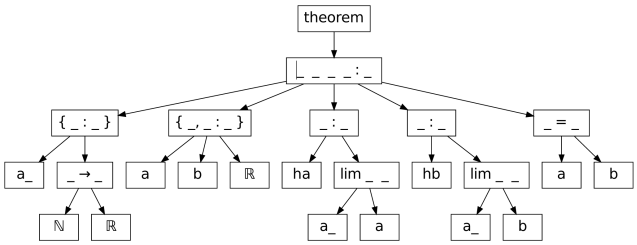

图 2: 完整复原的算子树

大语言模型 (Large Language Model (LLM)) 技术成熟后, 自动形式化技术兴起. 北京大学团队的 Herald 模型[1]与上海交大团队的 ATLAS 模型[2]是代表性的自动形式化模型.

[1]https://arxiv.org/pdf/2410.10878
[2]https://arxiv.org/pdf/2502.05567

2.3 证明状态变换

接下来我们将介绍一阶逻辑算子的证明状态变换, 以及在 Lean 中如何使用这些变换来进行形式化证明. 介绍一阶形式逻辑时, 用类似 Gentzen 相继式演算的书写方式表意, 但一些左右规则做了适当简化.

语法 2.3.1 变量声明

在 Lean 中, 使用 `variable` 关键字可以全局声明变量, 这样就无需在定理声明时反复声明相同的变量:

```
variable (p q : Prop)
theorem pq : p → q := by sorry
```

但需要注意, 这样声明的定理仍是依值函数, 在使用时仍需提供参数:

```
#check pq        --  ∀ (p q : Prop), p → q
theorem sth (hp : p) : q := pq hp  -- error
```

若隐式声明变量, 则不会有这个问题. 我们下文中默认 p, q, r 指代命题.

```
variable {p q r : Prop}
```

我们用蓝色 "$\vdash$" 表示 Lean 策略证明中的 "证明状态", "$\longrightarrow$" 表示证明状态变换, 含义为 "要证明 ... 只需证明 ...". 注意证明状态变化的方向和实际推理中证明合成的方向是相反的.

公理 2.1 切割规则 (Cut)

若语境下命题 p 有证明, 借助命题 p 的证明能构造命题 q 的证明, 则语境下能直接构造命题 q 的证明:

$$\frac{\Gamma \vdash (h_p : p) \qquad \Gamma, (h_p : p) \vdash (h_q : q)}{\Gamma \vdash (h_q : q)}$$

策略 1 引理引入

使用 `have` 策略可以构造一个新的引理以帮助证明目标:

```
theorem th_name {p : Prop} : q := by
  have hp : p := by sorry
  #check hp          --  p
  sorry
```

`have` 策略内是一个独立的证明状态, 其证明目标为 `p`, 证明完成后会将 `hp` 添加到当前语境中.

公理 2.2 合取右规则 (Right Rule of Conjunction)

若命题 p 和 q 均有证明, 则可构造一个命题 $p \wedge q$ 的证明:

$$\frac{\Gamma \vdash (h_p : p) \qquad \Gamma \vdash (h_q : q)}{\Gamma \vdash (h : p \wedge q)} (\wedge R)$$

策略 2 合取证明目标的处理
合取右规则对应以下证明状态的变化:

$$\Gamma \vdash (p \wedge q) \quad \longrightarrow \quad \begin{cases} (\Gamma \vdash p) \\ (\Gamma \vdash q) \end{cases}$$

可使用 `constructor` 策略拆分合取证明目标:

```
theorem Conj_constructor (hp : p) (hq : q) : p ∧ q := by
  constructor        -- Goal_1 : p, Goal_2 : q
  · exact hp
  · exact hq
```

也可使用 `refine` 策略与语法缺口来构造合取证明目标:

```
theorem Conj_refine (hp : p) (hq : q) : p ∧ q := by
  refine ⟨?_, ?_⟩     -- Goal_1 : p, Goal_2 : q
  · exact hp
  · exact hq
```

公理 2.3 合取左规则 (Left Rules of Conjunction)
若可用一个命题 p 的证明构造结论的证明, 则可用 $p \wedge q$ 的证明构造结论的证明:

$$\frac{\Gamma \vdash (h_p : p)}{\Gamma, (h : p \wedge q) \vdash G}(\wedge L_1)$$

若可用一个命题 q 的证明构造结论的证明, 则可用 $p \wedge q$ 的证明构造结论的证明:

$$\frac{\Gamma \vdash (h_q : q)}{\Gamma, (h : p \wedge q) \vdash G}(\wedge L_2)$$

策略 3 合取假设的处理
合取左规则对应以下证明状态变化:

$$\Gamma, (h : p \wedge q) \vdash r \quad \longrightarrow \quad \Gamma, (h : p \wedge q), (h_p : p), (h_q : q) \vdash r$$

可使用 `have` 策略解析合取假设:

```
theorem Conj_have1 (h : p ∧ q) : p := by
  have ⟨hp, hq⟩ := h
  exact hp

theorem Conj_have2 (h : p ∧ q) : p := by
  have hp : p := h.left
  have hq : q := h.right
  exact hp
```

```
theorem Conj_have3 (h : p ∧ q) : p := by
  have hp : p := h.1
  have hq : q := h.2
  exact hp
```

have 策略会保留原合取命题的证明 $h : p \wedge q$, 但下两种方式会将之丢弃, 即对应以下证明状态变化:

$$\Gamma, (h : p \wedge q) \vdash r \quad \longrightarrow \quad \Gamma, (h_p : p), (h_q : q) \vdash r$$

也可使用 obtain 和 rcases 策略解析合取假设:

```
theorem Conj_rcases (h : p ∧ q) : p := by
  rcases h with ⟨hp, hq⟩
  exact hp

theorem Conj_obtain (h : p ∧ q) : p := by
  obtain ⟨hp, hq⟩ := h
  exact hp
```

公理 2.4 蕴含右规则 (Right Rule of Implication)
若在有命题 p 的证明时可构造一个命题q 的证明, 则可构造一个命题 $p \to q$ 的证明:

$$\frac{\Gamma, (h_p : p) \vdash (h_q : q)}{\Gamma \vdash (h : p \to q)} (\to R)$$

策略 4 蕴含证明目标的处理
蕴含右规则对应证明状态变化:

$$\Gamma \vdash (p \to q) \quad \longrightarrow \quad \Gamma, (h_p : p) \vdash q$$

可使用 intro 策略来提取蕴含命题的假设:

```
theorem Imp_intro (hq : q) : p → q := by
  intro hp
  exact hq
```

也可使用 refine 策略与语法缺口来构造合取证明目标: [a]

```
theorem Imp_refine_left (hq : q) : p → q := by
  refine fun hp => ?_
  exact hq
```

[a] λ-抽象的演算语法将在第三章具体讲解.

公理 2.5 蕴含左规则 (Left Rule of Implication)
若命题 p 有证明, 则可用 $p \to q$ 的证明构造一个命题 q 的证明:

$$\frac{\Gamma \vdash (h_p : p)}{\Gamma, (h : p \to q) \vdash q}(\to L)$$

策略 5 蕴含假设的处理
蕴含左规则以下证明状态变化:

$$\Gamma, (h : p \to G) \vdash G \quad \longrightarrow \quad \Gamma, (h : p \to G) \vdash p$$

可使用 `apply` 策略来应用蕴含假设:

```
theorem Imp_apply (hp : p) (h : p → q) : q := by
  apply h
  exact hp
```

也可使用 `refine` 策略与语法缺口来更改证明目标:

```
theorem Imp_refine_right (hp : p) (h : p → q) : q := by
  refine h ?_
  exact hp
```

公理 2.6 析取右规则 (Right Rule of Disjunction)
若命题 p 有证明, 则可构造一个命题 $p \vee q$ 的证明:

$$\frac{\Gamma \vdash (h_p : p)}{\Gamma \vdash (h : p \vee q)}(\vee R_1)$$

若命题 q 有证明, 则可构造一个命题 $p \vee q$ 的证明:

$$\frac{\Gamma \vdash (h_q : q)}{\Gamma \vdash (h : p \vee q)}(\vee R_2)$$

策略 6 析取证明目标的处理
析取右规则对应以下证明状态变化:

$$\Gamma \vdash (p \vee q) \quad \longrightarrow \quad (\Gamma \vdash p); \qquad \Gamma \vdash (p \vee q) \quad \longrightarrow \quad (\Gamma \vdash q)$$

使用 `left` / `right` 策略选择析取的左 / 右分支:

```
theorem Disj_left (hp : p) : p ∨ q := by
  left           -- Goal : p
  exact hp

theorem Disj_right (hq : q) : p ∨ q := by
  right          -- Goal : q
  exact hq
```

公理 2.7 析取左规则 (Left Rule of Disjunction)
若用 p 和 q 可分别合成出 r 的证明, 则可用命题 $p \vee q$ 的证明构造一个命题 r 的证明:

$$\frac{\Gamma, (h_p : p) \vdash (h_r : r) \qquad \Gamma, (h_q : q) \vdash (h_r : r)}{\Gamma, (h : p \vee q) \vdash (h_r : r)} (\vee L)$$

策略 7 析取假设的处理
析取左规则对应以下证明状态变化:

$$\Gamma, (h : p \vee q) \vdash G \quad \longrightarrow \quad \begin{cases} (\Gamma, (h_p : p) \vdash G) \\ (\Gamma, (h_q : q) \vdash G) \end{cases}$$

可使用 `obtain` 策略或 `rcases` 策略对析取假设分类讨论:

```
theorem Disj_obtain (h : p ∨ q)(pr : p → r)(qr : q → r): r := by
  obtain hp | hq := h
  · exact pr hp
  · exact qr hq

theorem Disj_rcases (h : p ∨ q)(pr : p → r)(qr : q → r): r := by
  rcases h with hp | hq
  · exact pr hp
  · exact qr hq
```

公理 2.8 全称量词右规则 (Right Rule of Universal Quantifier)
若 a 是 α 的元素且不在语境中, p 是 α 上的一元谓词, 命题 $p(a)$ 有一个证明, 则可构造一个命题 $\forall x : \alpha, p(x)$ 的证明:

$$\frac{\Gamma, (a : \alpha) \vdash (h_p : p(a))}{\Gamma \vdash \forall x : \alpha, p(x)} (\forall R)$$

策略 8 全称量词证明目标的处理
全称量词右规则对应以下证明状态变化:

$$\Gamma \vdash (\forall x : \alpha, p(x)) \quad \longrightarrow \quad \Gamma, (a : \alpha) \vdash p(a)$$

可使用 `intro` 策略将全称量词目标实例化:

```
theorem Forall_intro : ∀ x, P x := by
  intro x        --  Goal: P x
  #check x       --  x : α
  sorry
```

也可使用 `refine` 策略创造语法缺口:

```
theorem Forall_refine : ∀ x, P x := by
  refine fun x => ?_   -- Goal: P x
```

```
    #check x              -- x : α
    sorry
```

公理 2.9 全称量词左规则 (Left Rule of Universal Quantifier)
若 p 是 α 上的一元谓词, a 是类型为 α 的元素, 命题 $p(a)$ 有证明, a 是 α 的元素, 则可构造一个命题 $p(a)$ 的证明:

$$\frac{\Gamma,(a:\alpha),(h_p:p(a))\vdash q}{\Gamma,(h:\forall x:\alpha,p(x)),(a:\alpha)\vdash q}(\forall L)$$

策略 9 全称量词假设的处理
全称量词左规则对应以下证明状态变化:

$$\Gamma,(a:\alpha),(h:\forall x:\alpha,P(x))\vdash p \quad\longrightarrow\quad \Gamma,(a:\alpha),(h:\forall x:\alpha,P(x)),(h_p:P(a))\vdash p$$

可以对全称量词假设代入具体变量:

```
theorem Forall_sub (h : ∀ x, P x) (a : α) : p := by
  #check h a          -- P a
  sorry
```

特别地, 当证明目标与全称量词代入的结果相同时, 可以直接使用 `apply` 策略. 原则上需要提供一个 α 类型的实例, 但实际上 Lean 会自动推断并直接完成证明:

$$\Gamma,(a:\alpha),(h:\forall x:\alpha,P(x))\vdash P(a) \quad\longrightarrow\quad \Gamma,(h:\forall x:\alpha,P(x))\vdash \alpha \quad\longrightarrow\quad \texttt{No Goals}$$

```
theorem Forall_apply (h : ∀ x, P x) (x : α) : P x := by
  apply h
```

公理 2.10 存在量词右规则 (Right Rule of Existential Quantifier)
若 a 是 α 的元素, p 是 α 上的一元谓词, 命题 $p(a)$ 有一个证明, 则可构造一个命题 $\exists x:\alpha,p(x)$ 的证明:

$$\frac{\Gamma(a:\alpha)\vdash(h_p:p(a))}{\Gamma(a:\alpha)\vdash(\exists x:\alpha,p(x))}(\exists R)$$

策略 10 存在量词证明目标的处理
以下证明状态变化:

$$\Gamma,(a:\alpha)\vdash(\exists x:\alpha,p(x)) \quad\longrightarrow\quad \Gamma,(a:\alpha)\vdash p(a)$$
$$\Gamma\vdash(\exists x:\alpha,p(x)) \quad\longrightarrow\quad (\Gamma\vdash(a:\alpha)),(\Gamma\vdash p(a))$$

可使用 `use` 策略提供实例:

```
theorem Exists_use (a : α) : ∃ x, P x := by
  use a                 -- Goal: P a
  sorry
```

也可使用 refine 策略创造语法缺口, 依值对的依赖关系会被保留:

```
theorem Exists_refine (a : α) : ∃ x, P x := by
  refine ⟨?_, ?_⟩
  · exact a              -- Remaining Goal: P a
  · sorry
```

使用 constructor 搭配 case 策略也可以实现同样效果: [a]

```
theorem Exists_constructor (a : α) (h : P a) : ∃ x, P x := by
  constructor
  case w => exact a
  · sorry
```

[a] 我没有搞明白为什么依值对中的变量会自动变成第二个目标.

公理 2.11 存在量词左规则 (Left Rule of Existential Quantifier)
若命题 $\exists x:\alpha, p(x)$ 有证明, 则可构造一个类型为 α 的元素 a 和一个命题 $p(a)$ 的证明:

$$\frac{\Gamma,(h:\exists x:\alpha,P(x))\vdash(h_p:p)}{\Gamma,(a:\alpha),(h:P(a))\vdash(h_p:p)}(\exists L)$$

策略 11 存在量词假设的处理
存在量词右规则对应以下证明状态变化:

$$\Gamma,(h:\exists x:\alpha,p)\vdash G \quad\longrightarrow\quad \Gamma,(a:\alpha),(h_p:p(a)),(h:\exists x:\alpha,p)\vdash G$$

使用 have, obtain 或 rcases 策略解析存在量词假设:

```
theorem Exists_have (h : ∃ x, P x) : p := by
  have ⟨a, ha⟩ := h
  sorry

theorem Exists_obtain (h : ∃ x, P x) : p := by
  obtain ⟨a, ha⟩ := h
  sorry

theorem Exists_rcases (h : ∃ x, P x) : p := by
  rcases h with ⟨a, ha⟩
  sorry
```

以上三种方式与合取假设的处理大同小异. 除 have 策略外, 其他策略会导致原存在量词假设丢失.

公理 2.12 矛盾律与排中律 (Contradiction and Law of Excluded Middle)
命题 $\neg p$ 定义为 p 蕴含矛盾:

$$\neg p := p \to \texttt{False}$$

若 p 和 $\neg p$ 同时有证明, 则可构造一个矛盾的证明, 进而可构造任何命题的证明:

$$\frac{}{\Gamma, (h_1 : p), (h_2 : \neg p) \vdash \texttt{False}}$$

p 和 $\neg p$ 中总有一个有证明, 即:

$$\frac{\Gamma, (h_1 : p) \vdash (h_q : q) \qquad \Gamma, (h_2 : \neg p) \vdash (h_q : q)}{\Gamma \vdash (h_q : q)}$$

策略 12 否定算子的处理

使用 intro 策略处理否定算子的定义: (不依赖排中律)

```
theorem Not_intro : ¬p := by
  intro h
  sorry
```

当假设中有矛盾时, 使用 contradiction 策略结束证明:

```
theorem Not_contradiction (h1 : p) (h2 : ¬p) : q := by
  contradiction

theorem False_contradiction (h : False) : q := by
  contradiction
```

排中律允许使用反证法证明命题 (contrapose! 策略和 by_contra 策略):

```
theorem Not_contrapose (h1 : p) (h2 : ¬p) : q := by
  contrapose! h2
  exact h1

theorem Not_by_contra : p := by
  by_contra h
  sorry
```

排中律允许使用 by_cases 策略进行分类讨论, 对应的证明状态变化:

$$\Gamma \vdash q \quad \longrightarrow \quad \begin{cases} (\Gamma (h : p) \vdash q) \\ (\Gamma (h : \neg p) \vdash q) \end{cases}$$

```
theorem By_cases {p : Prop} : q := by
  by_cases h : p
  · sorry
  · sorry
```