2025 Lean 与数学形式化讲义(1B)

上海交通大学 AI4MATH 团队

1 初识 Lean

我们现在要初步地探索 Lean 交互式形式化定理证明器这一强大工具. 我们不妨先简单体验一下使用 Lean 进行形式化数学证明的感受. 我们会在之后逐步深入探索背后的原理和工作细节.

1.1 认识 Lean 的 VS Code 界面

使用 VS Code 编辑器打开 Mathematics In Lean 项目, 定位至以下位置:

`mathematics_in_lean/MIL/C01_Introduction/S02_Overview.lean`

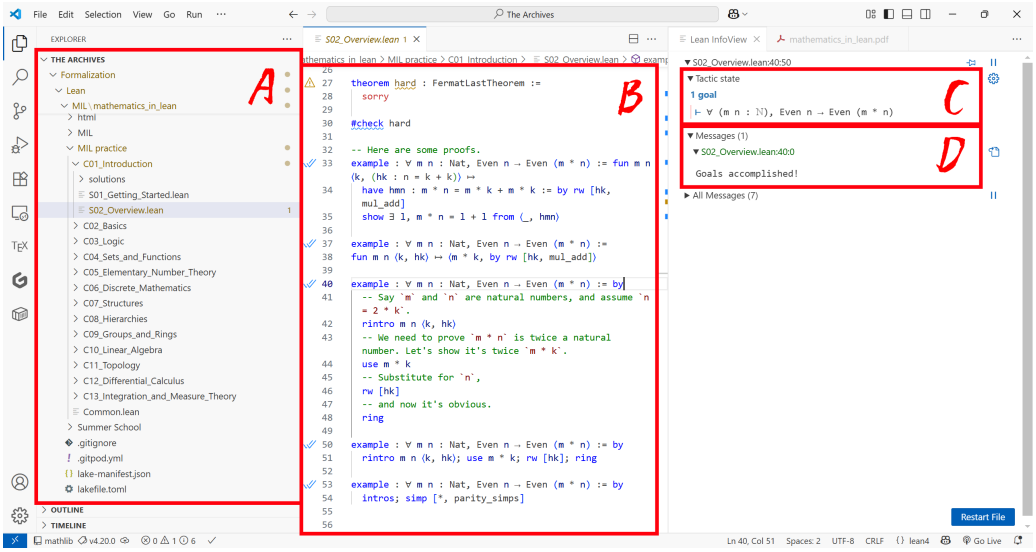


图 1: 使用 VS Code 编辑器编写 Lean 的界面

如图1.1所示, 默认布局下, 界面将以三大纵栏呈现. [1]

第一纵栏, 即图中 A 区域, 是 VS Code 自带的**文件浏览器 (File Explorer)**. 通常我们会打开一个 Lean 项目所在根目录, 并通过文件浏览器来定位至所需查看或编辑的文件.

第二纵栏, 即图中 B 区域, 是 VS Code 自带的**代码编辑器 (Code Editor)**. 使用者操作 Lean 工具的方式是使用 Lean 语言编写代码.

第三纵栏, 是 VS Code Lean 4 语言扩展提供的功能, 称为 **Lean 信息视图 (Lean InfoView)**, 主要用于展示 Lean 程序内核通过**语言服务器协议 (Language Server Protocol (LSP))**返回给使用者的各类信息. 其中 D 区域是一个常驻栏目, 用于展示任何对应当前光标位置的**消息 (Messages)**. 特别地, 图中 C 区域依据光标所在位置的语境将展示特别的信息: 在进行定理的形式化证明时, 将展示当前定理的**证明状态 (Proof State / Tactic State)**; 当进行对象的类型检查时, 将展示被检查对象的类型; 当进行定理检索时, 将展示检索结果或是推荐的下一步证明步骤, 等等.

现在在 `mathematics_in_lean` 文件夹下新建一个 `test` 文件夹, 在其中新建一个 `test.lean` 文件, 并输入以下代码来导入 Mathlib 库 (这可能会比较慢, 建议每节课前先完成这一步):

```
import Mathlib
```

[1] 鉴于并非所有人都足够熟悉 VS Code 编辑器, 我们也简作介绍, 同时进行一些术语的统一.

1.2 初识类型论

Lean 依赖的形式化理论框架是**依值类型论 / 依赖类型论 (Dependent Type Theory (DTT))**, 简称**类型论 (Type Theory)**. [2] 我们在正式上手使用 Lean 之前, 先学习一些简单的类型论[3]概念. 请注意, 这只是在初步建立一种"直觉性"的认识, 在第三章的学习中我们会进行更为系统的学习.

概念 1.2.1 对象的类型 (Type of an Object)
在类型论语言下, 任何**对象 (Object)** 有其**类型 (Type)**.
若对象 a 的类型为 A, 记作 $a : A$.
此时称对象 a 是类型 A 的一个**元素 / 实例 / 项 (Element / Instance / Term)**. [a]

[a] 这个术语很麻烦, 称"元素"容易与"集合-元素"混淆, "实例"容易与"类-实例"混淆, "项"则没有强调对象相对其类型的关系.

语法 1.2.2 类型检查 (Type Check)
在 Lean 中, 使用 `#check` 命令可以在 InfoView 中查看一个对象的类型. 设 $a : A$, 那么:

```
#check a        -- A
```

我们上述讨论的"对象", 是类型论语言框架之内被描述的"对象". 而 Lean 本身作为一个面向"对象"编程得到的程序, 许多"对象"是程序意义下的"对象", 而非类型论语言描述的"对象". 这通常包括:

反例 1.2.1 Lean 作为元语言自身的语法成分
如 `#check` 命令本身. 查看 "`#check`"[a] 命令的类型是没有意义的, 只会报错:

```
#check #check   -- error
```

[a] 这里的引号不代表字符串. 由于 Lean 同时可承担编程语言的身份, 对字符串进行类型检查是合法的, 结果为 "`String`".

反例 1.2.2 类型论作为元语言自身的语法组成部分
事实上可归为上一类. 但由于它通常和类型论语言所描述的对象一起出现, 可能更容易引起混淆. 这包括用于类型声明的冒号 ":", 定义号 ":=" 等等. 查看 ":", ":=" 的类型也是没有意义的:

```
#check :        -- error
#check :=       -- error
```

反例 1.2.3 Lean 的"语法糖"
主要指 Lean 中为了方便人理解而通过 Unicode 符号来表示函数的机制. 例如用加号 "+" 指代 "Add" 函数[a]等等. 查看 "+" 的类型会报错, 但可以查看 "Add" 函数的类型.

```
#check +        -- error
#check Add      -- Type u → Type u
```

[a] 实际上是某种为了处理符号运算而定义的类型类的实例, 距离具体的函数实现还要再套几层. 这是个很麻烦的问题.

[2] 严格来讲, Lean 中并不是所有东西都以类型论为基础, 但这是个很复杂的问题.
[3] "简单的依值类型论知识", 不是 Russell 先生的"简单类型论".

可能还有很多其他例外情况, 不再一一列举. 现在我们举一些真正类型论语言框架下讨论的“对象”:

例 1.2.1 自然数

默认情况下, Lean 会将数字解读为自然数:

```
#check 1        -- ℕ
#check ℕ        -- Type
```

例 1.2.2 命题

命题证明的类型是命题, 命题的类型是命题类型 (我们稍后会介绍这是怎么回事):

```
#check rfl      -- ∀ {α : Sort u} {a : α}, a = a
#check ∀ {α : Type} {a : α}, a = a    -- Prop
```

语法 1.2.3 对象的定义

在 Lean 中声明一个类型为 A 的对象 a, 可以通过 `def` 关键字实现:

```
def a : A := sorry
```

其中 “`:`” 是类型声明符, “`:=`” 是定义符, `sorry` 是占位符, 在对象的具体构造未被给出的情况下, `sorry` 可以给出任何一个类型的对象, 但并无实际意义.

例 1.2.3 定义一个自然数对象

可以用以下代码来定义一个值为 1 的自然数对象 n:

```
def n : ℕ := 1
def m : ℕ := n + 1
```

类型的概念并不足以建立对象之间的关系. 在类型论中, 通过“函数”的概念来刻画对象之间的关系:

概念 1.2.4 函数类型 (Function Type)

若 A, B 是类型, 那么 $A \to B$ 也是类型, 称为由 A 类型映射到 B 类型的**函数类型 (Function Type)**. 其元素称为**函数 (Functions)**.

若 f 是一个从 A 类型映射到 B 类型的函数, 即 $f : A \to B$; 此外, a 是 A 类型的一个元素, 那么 $f(a)$ 是一个 B 类型的元素.

概念 1.2.5 函数的 Curry 化 (Currying)

规定函数类型中 “$\to$” 算子是右结合的, 即用 $A \to B \to C$ 表示 $A \to (B \to C)$. 这种表示习惯称为**函数的 Curry 化 (Curry)**, 这样我们可以 “链式” 地表示多元函数. 参数很多时, 一个函数类型像是长长的链条一般, 因此也成为 **Curry 链 (Curry Chain)**.

例 1.2.4 Curry 链

我们通常习惯将自然数上的加法视作一个二元函数 f_+:

$$f_+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\forall a, b : \mathbb{N}, f_+(a, b) = a + b$$

Curry 化的函数 g_+ 则是这样的:

$$g_+ : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

$$\forall a : \mathbb{N}, g_+(a) : \mathbb{N} \to \mathbb{N}$$

$$\forall b : \mathbb{N}, (g_+(a))(b) = a + b$$

此时我们将 g_+ 写为 Curry 链的形式:

$$g_+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

第三章我们还会简单了解 λ-演算语言, 那时我们就能更好地表示函数对象.

语法 1.2.6 函数的定义与应用

在 Lean 中声明一个类型为 $A \to B$ 的函数 f 有两种写法: [a]

```
def f : A → B := sorry
def f (a : A) : B := sorry
```

一般计算机语言中, 对函数应用参数使用括号表示. 但在 Lean 中, 遵循函数式编程语言的习惯, 函数应用参数使用空格. 若 $f : A \to B$, $a : A$, 那么:

```
#check f (a)    -- B
#check f a      -- B
```

[a] 事实上, 第二种写法被用来兼容依值函数 (将在第三章讲到), 更加简洁常用.

语法 1.2.7 求值

使用 `#eval` 命令可以在 InfoView 中查看可计算表达式的值:

```
#eval 1 + 1          -- 2
```

例 1.2.5 定义一些简单的自然数上的函数

可以用两种方法定义一个自加函数 `My_self_add`:

```
def My_self_add_1 : ℕ → ℕ := fun n ↦ n + 1    -- λ-abstraction
def My_self_add_2 (n : ℕ) : ℕ := n + 1
#check My_self_add_1     -- ℕ → ℕ
#check My_self_add_2     -- ℕ → ℕ
#eval My_self_add_1 2    -- 3
#eval My_self_add_2 5    -- 6
```

语法 1.2.8 隐式参数
若有些参数的值可以通过上下文自动推断得到, 可以用 "_" 占位符隐式传递, Lean 会自动推理出参数的值. 更进一步, 可以在函数声明时用"{ * }" 将参数声明为隐式参数, Lean 会自动用占位符填充隐式参数. 隐式参数依然可以通过 (x := _) 强制性地显式传递.

```
def h := add_comm 1 2
#check h          -- 1 + 2 = 2 + 1
def f (a : ℕ) (b : ℕ) (h : a + b = b + a) : ℕ := a
#eval f _ _ h     -- ℕ
def g {a : ℕ} {b} (h : a + b = b + a) : ℕ := a
#eval g h         -- ℕ
def g' {a : ℕ} {b} (h : a + b = b + a) := h
#check g' (a := 1) (b := 2) sorry    -- 1 + 2 = 2 + 1
```

作为一门支持数学形式化证明的语言, 所需处理的最特殊的一类数学对象即是命题. 但假如我们用类型论讨论命题对象的目的只是为了描述清楚命题, 那么它们也就没有什么特殊性可言了. 在 Lean 中, 类型论不仅是一门负责把对象描述清楚的语言, 还要借助它"描述清楚"的能力来兼顾逻辑推理的任务. 这样我们不仅要描述清楚命题, 还要研究命题的证明. 因此, 每个命题可以被视为一个类型, 其元素是该命题的证明. 这样, 证明一个命题就等价于构造一个该命题对象的元素, 在一些已知命题的条件下证明一个新命题等价于用已知命题的元素来构造新命题的元素.

概念 1.2.9 命题类型 (Proposition Type)
在 Lean 中, **命题类型 (Proposition Type)** 是一个特殊的类型, 记作 `Prop`. 命题类型的元素称为**命题 (Propositions)**. 一个命题对象的元素称为该命题的**证明 (Proof)**.

但是一个命题可能可以通过多种方式构造出证明. 我们在使用已知命题的证明来构造新命题的证明时, 事实上并不关心这个已知命题是通过哪种方式证明得到的. 这就引出了证明无关性的公设.

关于 Lean 这样设计的巧思, 思想内涵是 **Curry-Howard 同构**, 我们将在第三章再详细介绍.

公理 1.1 证明无关性 (Proof Irrelevance)
若 α 是一个命题, h_1, h_2 都是 α 的证明, 那么将 h_1, h_2 视为是等同的.

某种意义上, 我们可以将命题对象本身视为它的"灵魂", 而将证明视为它的"本体". 每个"灵魂"只能用唯一的"本体"来承载. 我们通常把"本体"命名为它"灵魂"的内涵, 也即将定理的证明变量命名为其内容.

例 1.2.6 平方非负定理
在 Mathlib 库中, `pow_two_nonneg` 是"一个数的平方非负"这个定理的证明.

```
#check pow_two_nonneg
-- expected type: ∀ {R : Type u} [inst : Semiring R] [inst_1 :
   LinearOrder R] [IsRightCancelAdd R] [ZeroLEOneClass R] [
   ExistsAddOfLE R] [PosMulMono R] [AddLeftStrictMono R] (a : R)
   , 0 ≤ a ^ 2
```

可以看到定理的内容事实上非常复杂, 主要是一些可以自动合成的代数结构.

1.3 使用 Lean 进行形式化证明

语法 1.3.1 定理声明 (Theorem Declaration)
使用 theorem 关键字可以声明一个定理. 实际上它与 def 关键字的作用相同, 但声明定理要求定理的类型是一个命题. 定理声明的语法如下:

```
theorem th_name (var : varType) (h : hyp) : Goal := sorry
```

theorem 关键字还有若干变体, 都是为了优化 Lean 作为数学工具的体验, 本质上仍是 def 的变体:

1. lemma: 与 theorem 作用原理完全相同, 只是习惯上用于不太重要的引理声明;
2. example: 用于声明一个临时的例子, 与 theorem 作用原理也基本相同, 但不允许给对象起名, 也就无法在其他定理中调用通过 example 声明的对象;
3. axiom: 用于声明公理. 公理的声明不需要给出具体实现即可调用, 对应了数学公理“不证自明”的特点.

由于 theorem 本质上做着和 def 相同的工作, 证明命题的本质是构造一个命题对象的元素, 即给出某个命题函数的实现, 这种证明称为**项证明 (Term Proof)**. 但这种思维方式与一般的分步数学证明相去甚远, 于是 Lean 引入了 **策略 (Tactic)** 工具, 将命题对象的构造过程分解为若干类操作, 逐步给出命题的证明. 这种分步证明的方式称为**策略证明 (Tactic Proof)**, 与一般的数学证明的书写思路更加贴近.

语法 1.3.2 策略证明 (Tactic Proof)
使用 by 关键字可以开始一个 Tactic 证明:

```
theorem th_name (var : varType) (h : hyp) : Goal := by sorry
```

概念 1.3.3 证明状态 (Proof State)
将光标放在 by 关键字后, 可以看到 InfoView 中会显示当前定理的**证明状态 (Proof State)**.
一个证明状态的信息包括**证明目标 (Goal)**, 通常显示在 “⊢” 符号后; 以及一些“素材”, 包括**变量 (Variables)** 和**假设 (Hypotheses)**, 分别对应类型的类型是 Type 和类型的类型是 Prop 的对象[a].
事实上一个证明状态是一个依值函数类型, 用证明状态的全部信息可以用于完成一个等同的定理声明.
为了和 InfoView 在形式上统一, 我们之后暂时用蓝色的 ⊢ 符号来书写一个“证明状态”:

$$(x_i : T_i), (h_i : P_i) \vdash G$$

一个由策略和相关参数构成的**证明步骤 (Proof Step)** 构成一个证明状态的变换. 如果一个证明步骤将证明状态 S_1 变成了 $S_2, \ldots, S_n$, 事实上是在说: 如果提供 $S_2, \ldots, S_n$ 的证明, 那么我们能证出 S_1. 即一个证明步骤用 $S_2, \ldots, S_n$ 的证明对象结合参数合成出 S_1的证明对象.

[a] 实际上类型论语言中这两者是一样的, 我们只关心对象的相互依赖关系.

可以将定理的“实现”想象为一个货物交付的过程, 如图1.3所示. 其中 theorem 声明的定理相当于一个货物的订单, 项证明要构造出一个满足订单需求的对象来交付货物, 就必须以“手工作坊”的方式自己手搓一个对象交付.

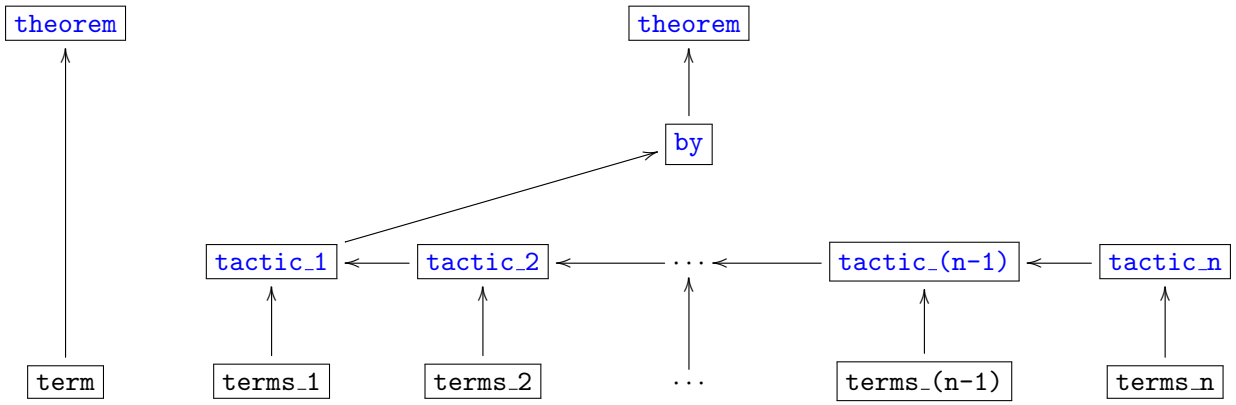


图 2: Lean 中项证明与策略证明的对比

这通常会非常非常麻烦, 因为我们很难搞清为了合成函数类型的对象而书写的各个 λ-表达式中引入的诸多变量之间的关系，这就像一个复杂的工业产品很难由一个人手工完成并交付.

策略证明则更像是一个生产流水线. `by` 相当于一个中介机构, 用于和流水线进行对接. 每一个策略相当于一个流水线的环节, 我们使用一些项来设置这个环节的生产功能, 这样一个生产环节就能根据自己的能力和本环节订单需求推测出它需要的原材料, 成为下一个环节的订单需求. 这样, 每个环节的交付目标都会成为上一个环节的原材料, 直到最后一个环节交付最终的货物.

对于复杂的证明, 项证明会变得非常复杂, 我们要一次性提供一个巨大无比的对象, 这会变成一项几乎不可能手动完成的任务. 但是使用策略证明, 每一个步骤用到的对象可以变得非常简单, 我们只需递归地分步甚至可以分工完成一个个简单的任务即可.

策略 1 `rw` (`rewrite`)
`rw` (重写)策略拿入一个等式或等价命题的证明, 在指定的项(证明目标或是假设)中寻找左式对应的表达式, 并将其替换为右式.
通过 `at` 关键字指定策略作用的目标, `at *` 表示作用于所有项.
可以通过 $\leftarrow$ 符号更改重写的方向, 即将右式替换为左式.
假设 A 是一个表达式, h 是等式 $A = B$ 或等价命题 $A \leftrightarrow B$ 的证明, 那么:

```
-- Proof state: (h1 : A) ⊢ A
rw [h]
-- Proof state: (h1 : A) ⊢ B
rw [h] at h1
-- Proof state: (h1 : B) ⊢ B
rw [← h] at *
-- Proof state: (h1 : A) ⊢ A
```

我们不妨简单地接触一些 Lean 的策略证明, 以便感受一下策略证明的语言风格. 定位到以下文件:
`mathematics_in_lean/MIL/C02_Basics/S01_Calculating.lean`
这是 Mathematics In Lean 习题集中关于一些简单等式和不等式命题的证明. 我们现在从自然语言证明的角度入手, 尝试理解第一个例题做了一件什么事:

例 1.3.1 实数的交换环结构

第一个例题是如下陈述的:

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
  rw [mul_comm a b]
  rw [mul_assoc b a c]
```

需要注意的是, 在未考虑完全结合律的情况下, 乘法算子默认是左结合的, `a * b * c` 是 `(a * b) * c` 的简写, 而不能视为 `a * (b * c)` 的简写.

用自然语言来陈述, 定理陈述的内容是: 若 a, b, c 是实数, 那么恒有 $(a \cdot b) \cdot c = b \cdot (a \cdot c)$.

它对应一个证明状态:

$$(a, b, c : \mathbb{R}) \vdash (a \cdot b \cdot c = b \cdot (a \cdot c))$$

查看一下 `rw` 策略的参数在同样的证明语境下的类型:

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
  #check mul_comm a b        -- a * b = b * a
  rw [mul_comm a b]
  #check mul_assoc b a c     -- b * a * c = b * (a * c)
  rw [mul_assoc b a c]
```

追踪证明状态的变化:

$$\boxed{(a, b, c : \mathbb{R}) \vdash (a \cdot b \cdot c = b \cdot (a \cdot c))}$$

$$\downarrow$$

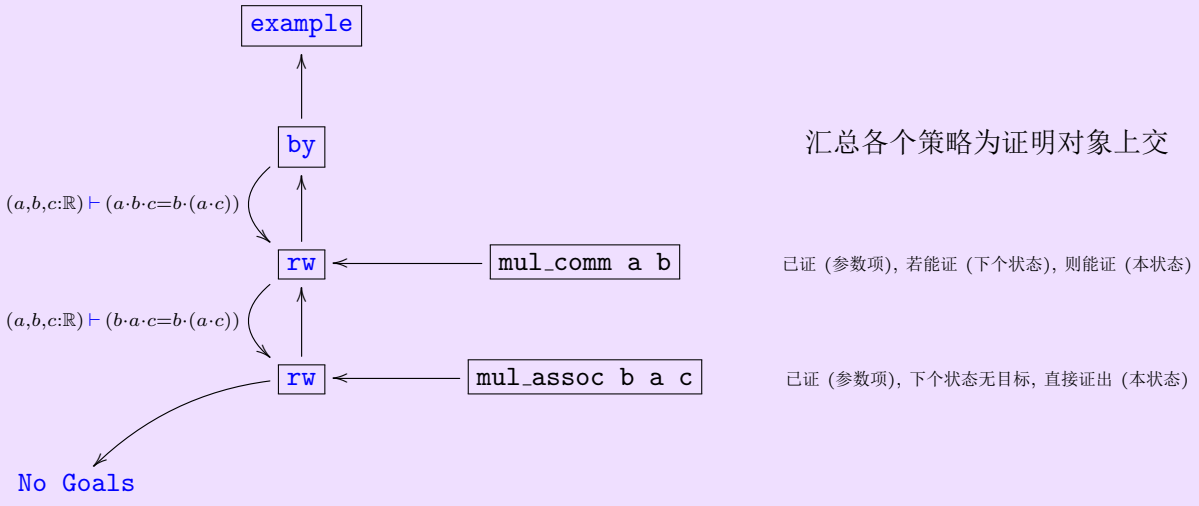
$$\boxed{(a, b, c : \mathbb{R}) \vdash (b \cdot a \cdot c = b \cdot (a \cdot c))}$$

$$\downarrow$$

$$\boxed{\texttt{No Goals}}$$

我们可以画出这样一张证明蓝图:

证明状态　　关键字 / 策略　　参数项　　干了什么



策略 2 calc

calc 策略主要用于处理各类传递的等式和不等式, 书写方式与递等式写法类似. 每行起始时可以用占位符 "_" 来表示左式, 因其必须与上式的右式相等, Lean 会自动推断出左式的值. 每行要提供一个本行等式或不等式成立的证明.

```
theorem th_name : a = z := by
  calc
    a = b := by sorry
    _ = c := by sorry
    -- ...
    _ = z := by sorry
```

calc 策略必须彻底完成目标算式的证明.

以下是一些处理表达式的"全自动"策略, 一般通过各类搜索算法或者项正规化的手段来自动完成证明, 通常需要导入 Mathlib 库才能使用.

策略 3 ring

ring 策略用于自动处理交换环结构上的等式证明.

ring 策略通过对交换环上的公式进行正规化来证明等式命题. [a]

[a] https://www.cs.ru.nl/~freek/courses/tt-2014/read/10.1.1.61.3041.pdf

策略 4 linarith

linarith 策略用于自动处理线性不等式证明.

策略 5 simp

自动化简策略, 一些在 Mathlib 库中标识了 simp 属性的定理会被自动应用, 以简化证明目标, 此外可以额外添加证明尝试化简.

策略 6 norm_num

通过对通常代数类型 (如 $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ 等) 上的四则运算, 取逆和幂运算表达式进行正规化实现自动化简, 可自动证明一些等式或不等式目标.

策略 7 aesop

aesop 策略是基于树搜索算法的自动化证明策略. [a]

[a] https://dl.acm.org/doi/pdf/10.1145/3573105.3575671

语法 1.3.4 #help tactic

使用 #help tactic 命令可以查看当前 Lean 版本支持的所有策略的详细信息.

也可访问 https://github.com/haruhisa-enomoto/mathlib4-all-tactics/blob/main/all-tactics.md 查询 tactic 信息, 或求助于大语言模型.