Westlakelean2025

emailboxofjjd

December 2024

1 结构 (Structure)

现代数学中，数学结构的使用至关重要，这些结构封装了多种信息，Lean 中提供了多种定义此类结构并构造特定实例的方法。我们已经在 Lean 中看到了结构的例子，例如环和拓扑空间。这里我们将进一步解释神秘的方括号参数，如 `[Ring α]` 和 `[Lattice α]`。并进一步展示如何自己定义和使用代数结构。

1.1 定义结构

从广义上讲，**结构**是对数据形式的约定，可能还包含数据需要满足的约束条件。结构的**实例**是一组满足这些约束条件的特定数据。例如，我们可以指定一个（三维空间里的）点是三个实数的三元组：

```
@[ext]
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ
```

`@[ext]` 告诉 Lean 自动生成一些定理，这些定理可用于证明当两个结构实例的每部分信息分别相等时，这两个实例相等，这一性质称为外延性。

```
#check Point.ext

example (a b : Point) (hx : a.x = b.x) (hy : a.y = b.y) (hz : a.z = b.z) : a = b := by
  ext
  repeat' assumption
```

然后，我们可以定义 `Point` 结构的特定实例。Lean 提供了多种方法来实现这一点。

```
def myPoint1 : Point where
  x := 2
  y := -1
  z := 4

def myPoint2 : Point :=
  ⟨2, -1, 4⟩

def myPoint3 :=
  Point.mk 2 (-1) 4
```

在第一个例子中，结构的字段被显式命名。在 myPoint3 的定义中提到的函数 Point.mk 被称为 Point 结构的构造函数，因为它用于构造元素。您可以根据需要指定不同的名称，例如 build。

```
structure Point' where build ::
  x : ℝ
  y : ℝ
  z : ℝ

#check Point'.build 2 (-1) 4
```

接下来的两个例子展示了如何定义结构上的函数。第二个例子显式使用了 Point.mk 构造函数，而第一个例子为了简洁使用了匿名构造函数。Lean 可以从 add 的目标类型推断出相关的构造函数。通常，我们会将与 Point 这样的结构相关的定义和定理放在同名的命名空间中。在下面的例子中，因为我们打开了 Point 命名空间，add 的全名是 Point.add。当命名空间未打开时，我们必须使用全名。但请记住，使用匿名投影符号通常很方便，它允许我们写 a.add b 而不是 Point.add a b。Lean 将前者解释为后者，因为 a 的类型是 Point。

```
namespace Point

def add (a b : Point) : Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def add' (a b : Point) : Point where
  x := a.x + b.x
  y := a.y + b.y
  z := a.z + b.z

#check add myPoint1 myPoint2
#check myPoint1.add myPoint2

end Point

#check Point.add myPoint1 myPoint2
#check myPoint1.add myPoint2
```

下面我们将继续将定义放在相关的命名空间中，但我们会省略命名空间命令。为了证明加法函数的性质，我们可以使用 rw 展开定义，并使用 ext 将结构两个元素之间的等式简化为组件之间的等式。下面我们使用 protected 关键字，即使命名空间打开，定理的名称也是 Point.add_comm。这在我们希望避免与通用定理（如 add_comm）产生歧义时很有帮助。

```
protected theorem add_comm (a b : Point) : add a b = add b a := by
  rw [add, add]
  ext <;> dsimp
  repeat' apply add_comm

example (a b : Point) : add a b = add b a := by simp [add, add_comm]
```

由于 Lean 可以在内部展开定义，我们有以下的依定义相等。

```
theorem add_x (a b : Point) : (a.add b).x = a.x + b.x :=
  rfl
```

也可以使用模式匹配在结构上定义函数，类似于定义递归函数的方式。下面的定义 `addAlt` 和 `addAlt'` 本质上是相同的；唯一的区别是我们在第二个中使用了匿名构造函数符号。尽管有时以这种方式定义函数很方便，并且结构 eta-约简使这种替代定义在定义上等价，但在后续证明中可能会使事情变得不那么方便。特别是，`rw [addAlt]` 会留下一个包含 `match` 语句的更混乱的目标视图。

```
def addAlt : Point → Point → Point
  | Point.mk x₁ y₁ z₁, Point.mk x₂ y₂ z₂ => ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

def addAlt' : Point → Point → Point
  | ⟨x₁, y₁, z₁⟩, ⟨x₂, y₂, z₂⟩ => ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

theorem addAlt_x (a b : Point) : (a.addAlt b).x = a.x + b.x := by
  rfl

theorem addAlt_comm (a b : Point) : addAlt a b = addAlt b a := by
  rw [addAlt, addAlt]
  ext <;> dsimp
  repeat' apply add_comm
```

特别有用的是，结构不仅可以指定数据类型，还可以指定数据必须满足的约束条件。在 Lean 中，后者表示为 `Prop` 类型的字段。例如，标准的 2-单纯形被定义为满足 `x ≥ 0`、`y ≥ 0`、`z ≥ 0` 和 `x + y + z = 1` 的点集。这个集合是三维空间中顶点为 `(1, 0, 0)`、`(0, 1, 0)` 和 `(0, 0, 1)` 的等边三角形及其内部。我们可以在 Lean 中表示如下：

```
structure StandardTwoSimplex where
  x : ℝ
  y : ℝ
  z : ℝ
  x_nonneg : 0 ≤ x
  y_nonneg : 0 ≤ y
  z_nonneg : 0 ≤ z
  sum_eq : x + y + z = 1
```

请注意，最后四个字段引用了 `x`、`y` 和 `z`，即前三个字段。我们可以定义一个将 2-单纯形映射到自身的函数，交换 `x` 和 `y`：

```
def swapXY (a : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := a.y
  y := a.x
  z := a.z
  x_nonneg := a.y_nonneg
  y_nonneg := a.x_nonneg
  z_nonneg := a.z_nonneg
  sum_eq := by rw [add_comm a.y a.x, a.sum_eq]
```

更有趣的是，我们可以计算单纯形上两个点的中点。我们在文件的开头添加了 `noncomputable section` 以便在实数上使用除法。

```
noncomputable section
```

```
def midpoint (a b : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := (a.x + b.x) / 2
  y := (a.y + b.y) / 2
  z := (a.z + b.z) / 2
  x_nonneg := div_nonneg (add_nonneg a.x_nonneg b.x_nonneg) (by norm_num)
  y_nonneg := div_nonneg (add_nonneg a.y_nonneg b.y_nonneg) (by norm_num)
  z_nonneg := div_nonneg (add_nonneg a.z_nonneg b.z_nonneg) (by norm_num)
  sum_eq := by field_simp; linarith [a.sum_eq, b.sum_eq]
```

在这里，我们使用简洁的证明项建立了 x_nonneg、y_nonneg 和 z_nonneg，但使用 by 在策略模式下建立了 sum_eq。

结构可以依赖于参数。例如，我们可以将标准 2-单纯形推广到任意 n 的标准 n-单纯形。在这个阶段，您不需要了解 Fin n 类型的任何信息，只知道它有 n 个元素，并且 Lean 知道如何对其求和。

```
open BigOperators

structure StandardSimplex (n : ℕ) where
  V : Fin n → ℝ
  NonNeg : ∀ i : Fin n, 0 ≤ V i
  sum_eq_one : (∑ i, V i) = 1

namespace StandardSimplex

def midpoint (n : ℕ) (a b : StandardSimplex n) : StandardSimplex n
    where
  V i := (a.V i + b.V i) / 2
  NonNeg := by
    intro i
    apply div_nonneg
    · linarith [a.NonNeg i, b.NonNeg i]
    norm_num
  sum_eq_one := by
    simp [div_eq_mul_inv, ← Finset.sum_mul, Finset.sum_add_distrib,
      a.sum_eq_one, b.sum_eq_one]
    field_simp

end StandardSimplex
```

我们已经看到结构可以用于将数据和属性捆绑在一起。有趣的是，它们也可以用于将属性捆绑在一起而不包含数据。例如，下一个结构 IsLinear 将线性的两个组成部分捆绑在一起。

```
structure IsLinear (f : ℝ → ℝ) where
  is_additive : ∀ x y, f (x + y) = f x + f y
  preserves_mul : ∀ x c, f (c * x) = c * f x

section
variable (f : ℝ → ℝ) (linf : IsLinear f)

#check linf.is_additive
```

```
#check linf.preserves_mul

end
```

值得指出的是，结构并不是将数据捆绑在一起的唯一方式。`Point` 数据结构可以使用通用类型 `product` 定义，而 `IsLinear` 可以用一个简单的 `and` 定义。

```
def Point'' :=
  ℝ × ℝ × ℝ

def IsLinear' (f : ℝ → ℝ) :=
  (∀ x y, f (x + y) = f x + f y) ∧ ∀ x c, f (c * x) = c * f x
```

通用的结构甚至可以用于组件之间依赖关系的结构。例如，子类型构造将数据与属性结合在一起。`PReal` 可以视为正实数的类型。任何 `x : PReal` 都有两个组件：值和正数的属性。您可以访问这些组件为 `x.val`，它具有类型 `ℝ`，以及 `x.property`，它表示 `0 < x.val` 的事实。

```
def PReal :=
  { y : ℝ // 0 < y }

section
variable (x : PReal)

#check x.val
#check x.property
#check x.1
#check x.2

end
```

我们也可以使用子类型来定义标准 2-单纯形.

```
def StandardTwoSimplex' :=
  { p : ℝ × ℝ × ℝ // 0 ≤ p.1 ∧ 0 ≤ p.2.1 ∧ 0 ≤ p.2.2 ∧ p.1 + p.2.1 + p.2.2 = 1 }
```

但即使我们可以使用乘积、子类型来代替结构，使用结构有许多优点。定义一个结构抽象了底层表示，并为访问组件的函数提供了自定义名称。这使得证明更加稳定：仅依赖于结构接口的证明通常在我们更改定义时继续有效，只要我们根据新定义重新定义旧的访问器。此外，正如我们即将看到的，Lean 提供了将结构编织成一个丰富的、相互关联的层次结构的支持，并管理它们之间的交互。

1.2 代数结构

为了澄清“代数结构”这一短语的含义，我们将通过一些例子来说明。

1. 一个偏序集由一个集合 P 和一个在 P 上的二元关系 $\leq$ 组成，该关系是传递的和自反的。一个群由一个集合 G 和一个结合的二元运算、一个单位元 e 以及一个为每个 G 中元素 x 返回逆元的函数组成。如果该运算是交换的，则该群是阿贝尔群或交换群。

2. 一个格是一个带有交和并的偏序集。

3. 一个环由一个（加法写作的）阿贝尔群 R 连同结合的乘法运算和单位元 1 组成，使得乘法对加法分配。如果乘法是交换的，则该环是交换环。

4. 一个有序环 R 由一个环连同其元素上的偏序组成，使得对于 R 中的每个 a、b 和 c，$a \le b$ 意味着 $a + c \le b + c$，并且 $0 \le a$ 和 $0 \le b$ 意味着 $0 \le ab$。

5. 一个度量空间由一个集合 X 和一个函数 $d : X \times X \to \mathbb{R}$ 组成，满足以下条件：

   (a) 对于 X 中的每个 x 和 y，$d(x, y) \ge 0$。

   (b) $d(x, y) = 0$ 当且仅当 $x = y$。

   (c) 对于 X 中的每个 x 和 y，$d(x, y) = d(y, x)$。

   (d) 对于 X 中的每个 x、y 和 z，$d(x, z) \le d(x, y) + d(y, z)$。

6. 一个拓扑空间由一个集合 X 和一个 X 的子集集合 τ 组成，称为 X 的开子集，满足以下条件：

   (a) 空集和 X 是开的。

   (b) 两个开集的交是开的。

   (c) 任意开集的并是开的。

在这些例子中，结构的元素属于一个集合，即载体集，有时它代表整个结构。例如，当我们说“设 G 是一个群”然后说“设 $x \in G$”时，我们使用 G 来代表结构和它的载体。并非每个代数结构都以这种方式与单个载体集相关联。例如，二分图涉及两个集合之间的关系，伽罗瓦对应也是如此。一个范畴也涉及两个感兴趣的集合，通常称为对象和态射。

这些例子表明，证明助手为了支持代数推理需要做的一些事情。首先，它需要识别结构的具体实例。数系 $\mathbb{Z}$、$\mathbb{Q}$ 和 $\mathbb{R}$ 都是有序环，我们应该能够在这些实例中应用关于有序环的通用定理。有时，一个具体集合可能以多种方式成为结构的实例。例如，除了 $\mathbb{R}$ 上的通常拓扑（构成实分析的基础）之外，我们还可以考虑 $\mathbb{R}$ 上的离散拓扑，其中每个集合都是开的。

其次，证明助手需要支持结构上的通用符号。在 Lean 中，符号 `*` 用于所有常见数系中的乘法，以及通用群和环中的乘法。当我们使用像 `f x * y` 这样的表达式时，Lean 必须使用关于 `f`、`x` 和 `y` 的类型信息来确定我们所指的是哪种乘法。

第三，它需要处理结构可以以各种方式从其他结构继承定义、定理和符号的事实。一些结构通过添加更多公理来扩展其他结构。一个交换环仍然是一个环，因此任何在环中有意义的定义在交换环中也有意义，任何在环中成立的定理在交换环中也成立。一些结构通过添加更多数据来扩展其他结构。例如，任何环的加法部分是一个加法群。环结构添加了乘法和单位元，以及管理它们并将它们与加法部分关联的公理。有时我们可以用一个结构定义另一个结构。任何度量空间都有一个与之相关的规范拓扑，即度量空间拓扑，并且可以有任何线性序关联的各种拓扑。

最后，重要的是要记住，数学允许我们使用函数和运算来定义结构，就像我们使用函数和运算来定义数字一样。群的乘积和幂仍然是群。对于每个 n，模 n 的整数形成一个环，对于每个 n，该环上的多项式矩阵再次形成一个环。因此，我们可以像计算它们的元素一样轻松地计算结构。这意味着代数结构在数学中具有双重身份，作为对象集合上的信息和作为独立对象。证明助手必须适应这种双重角色。

当处理具有关联代数结构的类型的元素时，证明助手需要识别结构并找到相关的定义、定理和符号。所有这些听起来像是很多工作，确实如此。但 Lean 使用一部分基本机制来执行这些任务。本节的目标是解释这些机制并向您展示如何使用它们。

第一个要素几乎显而易见：正式地说，代数结构是上节中所述的结构。代数结构是对满足某些公理假设的数据的，我们在第上节中看到，这正是 `structure` 命令设计用来容纳的。这是天作之合！

给定一个数据类型 `α`，我们可以如下定义 `α` 上的群结构。

```
structure Group₁ (α : Type*) where
  mul : α → α → α
  one : α
```

```
  inv : α → α
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  inv_mul_cancel : ∀ x : α, mul (inv x) x = one
```

请注意，类型 α 是 Group₁ 定义中的一个参数。因此，您应该将对象 struc : Group₁ α 视为 α 上的群结构。

这个群的定义类似于 Mathlib 中 Group 的定义，我们选择了名称 Group₁ 以区分我们的版本。如果您编写 #check Group 并按住 ctrl 键点击定义，您将看到 Mathlib 版本的 Group 被定义为扩展另一个结构；我们未来将解释如何做到这一点。

让我们构造一个群，即 Group₁ 类型的一个元素。对于任何类型对 α 和 β，Mathlib 定义了类型 Equiv α β，表示 α 和 β 之间的等价关系。Mathlib 还为此类型定义了符号 α ≃ β。元素 f : α ≃ β 是 α 和 β 之间的双射，由四个组件表示：一个从 α 到 β 的函数 f.toFun，逆函数 f.invFun 从 β 到 α，以及两个属性，指定这些函数确实是彼此的逆。

```
variable (α β γ : Type*)
variable (f : α ≃ β) (g : β ≃ γ)

#check Equiv α β
#check (f.toFun : α → β)
#check (f.invFun : β → α)
#check (f.right_inv : ∀ x : β, f (f.invFun x) = x)
#check (f.left_inv : ∀ x : α, f.invFun (f x) = x)
#check (Equiv.refl α : α ≃ α)
#check (f.symm : β ≃ α)
#check (f.trans g : α ≃ γ)
```

要注意，f.trans g 需要以相反的顺序组合函数。Mathlib 已声明从 Equiv α β 到函数类型 α → β 的强制转换，因此我们可以省略编写 .toFun 并让 Lean 为我们插入它。

```
example (x : α) : (f.trans g).toFun x = g.toFun (f.toFun x) :=
  rfl

example (x : α) : (f.trans g) x = g (f x) :=
  rfl

example : (f.trans g : α → γ) = g ∘ f :=
  rfl
```

Mathlib 还定义了类型 perm α，表示 α 与自身之间的等价关系。

```
example (α : Type*) : Equiv.Perm α = (α ≃ α) :=
  rfl
```

显然，Equiv.Perm α 在等价关系的复合下形成一个群。我们将其定向为 mul f g 等于 g.trans f，其正向函数是 f ∘ g。换句话说，乘法就是我们通常认为的双射的复合。这里我们定义这个群：

```
def permGroup {α : Type*} : Group₁ (Equiv.Perm α)
    where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
```

```
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

现在我们知道如何在 Lean 中定义代数结构，并且我们知道如何定义这些结构的实例。但我们也希望将符号与结构关联起来，以便我们可以在每个实例中使用它。此外，我们希望安排它，以便我们可以在结构上定义一个操作并在任何特定实例中使用它，并且我们希望安排它，以便我们可以在结构上证明一个定理并在任何实例中使用它。

事实上，Mathlib 已经设置为对 `Equiv.Perm α` 使用通用群符号、定义和定理。

```
variable {α : Type*} (f g : Equiv.Perm α) (n : ℕ)

#check f * g
#check mul_assoc f g g⁻¹
#check g ^ n

example : f * g * g⁻¹ = f := by rw [mul_assoc, mul_inv_cancel, mul_one]

example : f * g * g⁻¹ = f :=
  mul_inv_cancel_right f g

example {α : Type*} (f g : Equiv.Perm α) : g.symm.trans (g.trans f) = f :=
  mul_inv_cancel_right f g
```

我们现在的任务是理解幕后发生的魔法，使 `Equiv.Perm α` 的示例如此便于使用。

Lean 需要能够使用我们键入的表达式中找到的信息来找到相关的符号和隐式群结构。类似地，当我们使用类型为 `ℝ` 的表达式 `x` 和 `y` 编写 `x + y` 时，Lean 需要将 `+` 符号解释为实数上的相关加法函数。它还必须将类型 `ℝ` 识别为交换环的实例，以便所有交换环的定义和定理都可用。再举一个例子，连续性在 Lean 中是相对于任何两个拓扑空间定义的。当我们有 `f : ℝ → ℂ` 并编写 `Continuous f` 时，Lean 必须找到 `ℝ` 和 `ℂ` 上的相关拓扑。

这种魔法是通过三者的结合实现的。

1. ** 逻辑 **。一个应该在任何群中解释的定义将群的类型和群结构作为参数。类似地，关于任意群元素的定理以对群的类型和群结构的全称量词开始。2. ** 隐式参数 **。类型和结构的参数通常被隐式化，因此我们不必编写它们或在 Lean 信息窗口中看到它们。Lean 会默默地为我们填充这些信息。3. ** 类型类 (type-class) 推断 **。也称为类推断，这是一种简单但强大的机制，使我们能够注册信息供 Lean 以后使用。当 Lean 被要求填充定义、定理或符号的隐式参数时，它可以使用已注册的信息。

虽然注释 `(grp : Group G)` 告诉 Lean 它应该期望显式给出该参数，注释 `{grp : Group G}` 告诉 Lean 它应该尝试从表达式中的上下文线索中推断出来，但注释 `[grp : Group G]` 告诉 Lean 应该使用类型类推断来合成相应的参数。由于使用此类参数的重点是我们通常不需要显式引用它们，Lean 允许我们编写 `[Group G]` 并匿名化名称。您可能已经注意到，Lean 会自动选择像 `_inst_1` 这样的名称。当我们使用带有 `variables` 命令的匿名方括号注释时，只要变量仍在范围内，Lean 会自动将参数 `[Group G]` 添加到任何提到 `G` 的定义或定理中。

我们如何注册 Lean 需要使用的信息以执行搜索？回到我们的群的例子，我们只需要做两个更改。首先，我们使用关键字 `class` 而不是 `structure` 来定义群结构，以指示它是类推断的候选者。其次，我们使用关键字 `instance` 而不是 `def` 来注册特定实例。与类变量的名称一样，我们允许实例定义匿名，因为通常我们希望 Lean 找到它并使用它，而不会用细节困扰我们。

```
class Group₂ (α : Type*) where
  mul : α → α → α
  one : α
```

```
  inv : α → α
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  inv_mul_cancel : ∀ x : α, mul (inv x) x = one

instance {α : Type*} : Group₂ (Equiv.Perm α) where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

以下说明了它们的用法。

```
#check Group₂.mul

def mySquare {α : Type*} [Group₂ α] (x : α) :=
  Group₂.mul x x

#check mySquare

section
variable {β : Type*} (f g : Equiv.Perm β)

example : Group₂.mul f g = g.trans f :=
  rfl

example : mySquare f = f.trans f :=
  rfl

end
```

`#check` 命令显示 `Group₂.mul` 有一个隐式参数 `[Group₂ α]`，我们期望通过类推断找到它，其中 `α` 是 `Group₂.mul` 参数的类型。换句话说，`{α : Type*}` 是群元素类型的隐式参数，而 `[Group₂ α]` 是 `α` 上群结构的隐式参数。类似地，当我们为 `Group₂` 定义一个通用平方函数 `my_square` 时，我们使用 `{α : Type*}` 作为元素类型的隐式参数，并使用 `[Group₂ α]` 作为 `Group₂` 结构的隐式参数。

在第一个示例中，当我们编写 `Group₂.mul f g` 时，`f` 和 `g` 的类型告诉 Lean，在 `Group₂.mul` 的参数 `α` 中必须实例化为 `Equiv.Perm β`。这意味着 Lean 必须找到一个 `Group₂ (Equiv.Perm β)` 的元素。之前的实例声明告诉 Lean 如何做到这一点。问题解决了！

这种用于注册信息的简单机制非常有用，以便 Lean 在需要时可以找到它。以下是它的一种应用方式。在 Lean 的基础中，数据类型 `α` 可能为空。然而，在许多应用中，知道一个类型至少有一个元素是有用的。例如，函数 `List.headI` 返回列表的第一个元素，可以在列表为空时返回默认值。为了实现这一点，Lean 库定义了一个类 `Inhabited α`，它所做的只是存储一个默认值。我们可以显示 `Point` 类型是一个实例：

```
instance : Inhabited Point where default := ⟨0, 0, 0⟩
```

```
#check (default : Point)

example : ([] : List Point).headI = default :=
  rfl
```

类推断机制也用于通用符号。表达式 `x + y` 是 `Add.add x y` 的缩写，其中`Add α` 是一个类，它存储了 `α` 上的二元函数。编写 `x + y` 告诉 Lean 找到一个已注册的 `[Add.add α]` 实例并使用相应的函数。下面，我们为 `Point` 注册加法函数。

```
instance : Add Point where add := Point.add

section
variable (x y : Point)

#check x + y

example : x + y = Point.add x y :=
  rfl

end
```

通过这种方式，我们也可以将符号 `+` 分配给其他类型上的二元操作。

但我们还可以做得更好。我们已经看到 `*` 可以在任何群中使用，`+` 可以在任何加法群中使用，并且两者都可以在任何环中使用。当我们在 Lean 中定义一个新的环实例时，我们不必为该实例定义 `+` 和 `*`，因为 Lean 知道这些是为每个环定义的。我们可以使用这种方法为我们的 `Group₂` 类指定符号：

```
instance hasMulGroup₂ {α : Type*} [Group₂ α] : Mul α :=
  ⟨Group₂.mul⟩

instance hasOneGroup₂ {α : Type*} [Group₂ α] : One α :=
  ⟨Group₂.one⟩

instance hasInvGroup₂ {α : Type*} [Group₂ α] : Inv α :=
  ⟨Group₂.inv⟩

section
variable {α : Type*} (f g : Equiv.Perm α)

#check f * 1 * g⁻¹

def foo : f * 1 * g⁻¹ = g.symm.trans ((Equiv.refl α).trans f) :=
  rfl

end
```

这种方法之所以有效，是因为 Lean 执行递归搜索。根据我们声明的实例，Lean 可以通过找到 `Group₂ (Equiv.Perm α)` 的实例来找到 `Mul (Equiv.Perm α)` 的实例，并且它可以找到 `Group₂ (Equiv.Perm α)` 的实例，因为我们已经提供了一个。Lean 能够找到这两个事实并将它们链接在一起。

类推断是微妙的，使用时必须小心，因为它无形中控制了我们键入的表达式的解释。然而，如果明智地使用，类推断是一个强大的工具。它使 Lean 中的代数推理成为可能。