# Lean 是如何工作的

August 1, 2024

# 数学的基础与类型

# 数学的基础

我们需要写出完全形式化、严格的证明，那么我们就需要有表示数学对象的方法. 我们不得不开始谈论数学的基础，有了数学的基础，我们才可以构造并讨论千变万化的数学对象.

- 我们需要有一个逻辑框架，在里面谈论形式化的数学理论.
  - 系统本身指定了一些演绎方式，称为推理规则.
  - 我们常常还需要在系统内额外添加一些假设的陈述，称为公理.

# 集合论

当谈论数学的基础时，集合论指用公理刻画集合和它们之间的属于关系. 将各种数学对象都编码为集合，用对集合的操作来完成对数学对象的操作，从而发展数学理论. 以下以 ZFC 集合论为例:

- ZFC 集合论建立在一阶逻辑上，推理规则由一阶逻辑提供.
- ZFC 集合论在一阶逻辑内使用若干条公理刻画集合.

# 类型论

类型论允许定义类型，得到和使用类型中的值. 并且常常能实现一些简单的自动计算.

- 类型论本身就是推理系统，用推理规则进行类型的操作.
- 需要额外添加的公理通常较少.
- 有一些额外的规则帮助自动进行简化、类型判断与定义相等判断.

# 为什么是类型论

为什么要选择类型论而不是集合论？

- 集合论“没有类型检查”，所有对象都是集合. 可以合法地写出意料之外的陈述.
  - 可以讨论 $\pi \cap \sin$ 是什么、$\mathbb{R} \in 0$ 是否正确. 在一些常见的编码方式下可以证明 $2 = (0, 0)$、并且是 $1$ 上的拓扑.
- 多数数学家不会将数学对象视为集合编码，思考方式更加接近类型论，即用万有性质进行定义.
- 类型论天然适合用于描述计算过程，并能自动简化得到结果. 而集合论的函数是一堆特殊的集合有序对.

# 什么是类型论

正如有很多不同的集合论公理系统一样，类型论也有非常多的变种. 它们常常有以下特征.

- 足够强的类型论可以作为数学的基础使用. 可以在 Lean 中构造 ZFC 的模型.
- 类型论本身可以提供足够强大的逻辑系统.
- 类型论也是编程语言的类型系统的数学模型. 相关的研究指导着编程语言的设计.

接下来将介绍一些类型论中的常见的概念，以及它是如何作为逻辑系统使用的.

# 类型

每个类型有一些值，对一个表达式也可以确定它的类型. 当你想显式标记 $a$ 的类型为 $A$ 时，可以记 $a : A$.

- 可以类比集合论，如 $0 \in \mathbb{N}$，$\pi \in \mathbb{R}$.
  - 但需要注意在集合论中 $\in$ 是一个关系，可以集合论里面讨论某个对象是否属于某个集合. 而在类型论中 $:$ 是元语言层面的判断.
- 可以类比通用编程语言，如在 C++ 中，有 `0 : int`、`0.1 : double`、`std :: cin : std :: istream`.

# 类型

在类型论的各种推理规则中，重点关注这 2 个问题：如何得到这个类型的值？ 如何使用这个类型的值？

- 得到一个类型的值的方法称为构造规则.
- 使用一个类型的值的方法称为消去规则.

以下将介绍 Lean 的类型论中的一些类型.

# 函数类型

- 对于类型 $A, B$，我们有函数类型 $A \to B$.
- 若对于 $a : A$，有可能包含 $a$ 的表达式 $\Phi : B$，那么可以得到函数 $f(a) := \Phi$，$f : A \to B$.
- 例如，对 $n : \mathbb{N}$ 有函数 $f(n) := n + n$，此时 $f : \mathbb{N} \to \mathbb{N}$.
- 有时将上面的函数写为 $\lambda n.\, n + n$，可以类比数学家喜欢用的 $n \mapsto n + n$.
- 函数的消去规则当然是函数应用：对 $f : A \to B$ 和 $a : A$，我们将二者直接并列表示函数应用，即 $f\, a : B$.

函数类型相当基本重要：现代类型论始于 1940 年 Church 提出简单类型 $\lambda$ 演算，而 $\lambda$ 演算正是基于函数抽象这一概念.

# 积类型

- 对于类型 $A, B$，我们有积类型 $A \times B$.
- 类比集合论，对集合 $A, B$，所有 $a \in A, b \in B$ 的二元组 $(a, b)$.
- 构造规则：$A \to B \to A \times B$.
- 消去规则：$\mathsf{fst} : A \times B \to A$，$\mathsf{snd} : A \times B \to B$.

熟悉范畴论的同学可能会立即发现，如果将类型看作范畴中的对象而函数看作态射，那么 $A \times B$ 正是 $A$ 与 $B$ 的积对象.

# 和类型

- 对于类型 $A, B$，我们有和类型 $A + B$（一些资料中可能会使用其他记号）.
- 类比集合论，对集合 $A, B$，所有 $a \in A$ 形成的二元组 $(0, a)$ 与所有 $b \in B$ 形成的二元组 $(1, b)$ 构成了无交并 $A \sqcup B$.
- 构造规则：$\mathsf{inl} : A \to A + B$，$\mathsf{inr} : B \to A + B$.
- 消去规则：$(A \to C) \to (B \to C) \to (A + B \to C)$.

熟悉范畴论的同学可能会立即发现，$A + B$ 正是 $A$ 与 $B$ 的余积对象. 因此它也被称为余积类型.

# $\Pi$ 类型

- 对于类型 $A$，$B : A \to \mathrm{Type}$，我们有 $\Pi$ 类型 $\prod_{a:A} B\, a$.
- 可以视为函数类型的推广，返回的类型由输入的值决定.
- 可以视为 $a : A$ 索引的一族类型 $B\, a$ 的积，当 $A$ 仅有两个值时相当于积类型.
- 向量场就是一个 $\Pi$ 类型，每个点的向量空间一般不同.
  - 用日常一点的类比就是地球上每一点的风速（假装你对每一点有个风速类型）.

熟悉范畴论的同学可能会立即发现，$\prod_{a:A} B\, a$ 正是一族对象的积.

# $\Sigma$ 类型

- 对于类型 $A$，$B : A \to \mathrm{Type}$，我们有 $\Sigma$ 类型 $\sum_{a:A} B\, a$.
- 可以视为积类型的推广，右侧的类型由左侧值决定.
- 可以视为 $a : A$ 索引的一族类型 $B\, a$ 的和，当 $A$ 仅有两个值时相当于和类型.
- 熟悉微分几何的同学可以发现，向量丛就是一个 $\Sigma$ 类型.
  - 向量丛的截面，即上一页所说向量场. $\Sigma$ 类型与 $\Pi$ 类型也可以类比这一“截面”的关系.

熟悉范畴论的同学可能会立即发现，$\sum_{a:A} B\, a$ 正是一族对象的余积.

# 空类型

空类型即没有任何值的类型. 这里使用记号 $\mathbf{0}$ 表示.

- 没有任何构造规则.
- 消去规则是？
- 由空类型可以得到任何类型. 拥有空类型的值是荒谬的.

熟悉范畴论的同学可能会立即发现，空类型正是范畴中的始对象.
范畴的终对象是？
单元素类型. 这里使用记号 $\mathbf{1}$ 表示.

# 归纳类型

在 Lean 中，$\Pi$ 类型是基本类型，函数类型是它的一个特例. 而上文介绍的其他类型都是归纳类型的特例.

- 构造规则为指定的一些函数，可以输入自身.
- 消去规则为“归纳”或“递归”. 它保证了空语境中归纳类型的每个值一定是由构造规则构成的表达式树.
  - 例：自然数.
  - 例：列表.
  - 例：二叉树.

附加上指定的构造函数的类型构成了一个范畴，归纳类型正是范畴中的始对象.

# 如果时间充裕的话讲讲这些例子

- 自然数可以表示有限类型的势.
- 上文介绍的一些类型的势可以由自然数上的简单运算得到. 我们发现这些计算方法非常符合这些类型的名称.
  - 在一般的情况下对应基数的运算.
- 试试将自然数的运算律类比到类型上.

# 命题、宇宙层级、公理

# 命题与类型

- 以蕴涵为例，若我们有 $p \to q$ 的证明和 $p$ 的证明，那么我们可以得到 $q$ 的证明.
- 以合取为例，若我们有 $p$ 的证明和 $q$ 的证明，那么我们可以得到 $p \wedge q$ 的证明.
- 以析取为例，若我们有 $p$ 的证明，那么我们可以得到 $p \vee q$ 的证明，有 $q$ 的证明同理.
- 若我们有 $\bot$ 的证明，根据爆炸原理，我们可以得到任意命题的证明.
- 类比函数类型、积类型与和类型的构造规则、空类型的消去规则.

因此在类型论中我们可以将 命题视为类型 (propositions as types)、证明视为程序 (proofs as programs). 检查程序的类型就是检查证明无误.

# 命题与类型

谓词的类型是什么？ 类型到命题的函数.

| 类型论 | | 逻辑 | |
|---|---|---|---|
| 函数类型 | $A \to B$ | $P \to Q$ | 蕴涵 |
| 积类型 | $A \times B$ | $P \wedge Q$ | 合取 |
| 和类型 | $A + B$ | $P \vee Q$ | 析取 |
| $\Pi$ 类型 | $\prod_{a:A} B\, a$ | $\forall x, P(x)$ | 全称量化 |
| $\Sigma$ 类型 | $\sum_{a:A} B\, a$ | $\exists x, P(x)$ | 存在量化 |
| 空类型 | $\mathbf{0}$ | $\bot$ | 假 |
| 单元素类型 | $\mathbf{1}$ | $\top$ | 真 |

这一类型系统与逻辑的自然演绎之间的对应关系称为 Curry-Howard 对应.

# 否定

表上似乎没有看见否定？
否定可以定义为蕴涵假，类型论中即为 $P \to \mathbf{0}$.
试从势的角度说明合理性.
试从与假等价的角度说明合理性.

# 证明无关性

我们并不关心一个命题是如何被证明的. 命题的任何证明（即值）应当相等.

- 在 Lean 中，Prop 中的类型被视为命题. 与其他类型不同，命题有证明无关性.
- 一个命题的不同证明是定义相等的，没有任何区别. 用户不需要关注命题是如何证明的.
- 因此与一般的类型不同，命题要么为空，要么仅有一个值.

# 宇宙层级

允许在 Type 内部对 Type 进行量化会导致类似 Cantor 集合论中的悖论.

- 需要让 Type 居住在另一个类型里，称为 Type 1，同理还应有 Type 2、Type 3 等，称为宇宙层级.
- 考虑到我们还需要 Prop 命题宇宙，可以让 Prop 的类型为 Type. Prop 和 Type 一并称为 Sort.
  - Sort $0$ = Prop，Sort $u+1$ = Type $u$.
- 一般来说，Sort $u$ 到 Sort $v$ 的 $\Pi$ 类型应居住在 Sort $\max u\, v$ 中以避免悖论. 但当 Sort $v$ 为 Prop 时，$\Pi$ 类型可以在 Prop 中.
  - 这一一般性质称为直谓性，而违反的这一性质的 Prop 则称为非直谓的命题宇宙.
  - 非直谓的命题宇宙符合 Curry-Howard 对应.

# 命题的消去

- 一般来说，Type 中的类型可以消去到任何一个宇宙，而命题有证明无关性，应只能消除到 Prop.
- 但当命题明显只能通过 0 种或 1 种方式构造时，可以允许命题消去到任何一个宇宙，称为 subsingleton elimination.
- `False` 为 0 种的情况.
- `Eq` 与 `Acc` 为 1 种的情况. 前者用于替换，后者用于一般的递归定义的终止证明.

# 公理

Lean 中数学证明主要使用 3 个额外的公理.

- 命题外延性 `propext`，即等价的命题完全相等.
- 等价值的等价类相等 `Quot.sound`.
- 类型论选择公理 `Classical.choice`.

`Quot.sound` 可以推出函数外延性 `funext`. 由 `propext`、`funext`、`Classical.choice` 可得出排中律. 使用 `Classical.choice` 并不相当于使用 ZFC 集合论中的选择公理. 类型论的数学基础与 ZFC 集合论有较大区别.

# 一致性强度

- $n+2$ 个宇宙的 Lean 可以证明 ZFC $+$ $n$ 个不可达基数的一致性.
- ZFC $+$ $n$ 个不可达基数 $+$ 全局选择公理可以证明 $n+1$ 个宇宙的 Lean 的一致性.
  - 当然也可以直接用 ZFC $+$ $(n+1)$ 个不可达基数. 需要全局选择公理是因为 Lean 的 `Classical.choice` 直接以全局选择公理的形式提供.
- 因此 Lean 的一致性相当于（不准确地称为）ZFC $+$ $\omega$ 个不可达基数的一致性.

关于 Lean 的类型论的更多细节，参见 [The Type Theory of Lean].

# 类型类层次结构

# 数学结构与类型类

在不同的阶段，我们对数学对象有着不同的了解.

- 如整数和实数上的序、代数、拓扑、测度结构.
- 结构也可能会满足一些额外的性质.

谈论某个特定的数学对象时，我们常常会默认它带有一些附加的结构，而不总是每次都指定附加上什么结构.
我们需要支持为数学对象附加上默认的结构，谈论它也是在谈论它上面的结构. 这在 Lean 等 proof assistant 中常常以类型类的形式体现.

# 数学结构与类型类

类型类可以视为与一些类型或值有关的附加信息. 可以在全局定义一个类型类的实例，并在需要的使用时候让 Lean 自动查找.

- 也可以在指定的某个“范围”内定义，并且在需要的时候设定讨论的范围.

类型类系统不仅仅是简单查找，还需要有一定搜索合成能力.

- 如积类型上的序、代数、拓扑结构.

# 数学结构与类型类

类型类还用来完成一些其他事情，如类型转换.

- 当类型不符合预期时，Lean 会尝试插入一个类型转换.
- 当一个非函数的值被用作函数时，Lean 会尝试插入一个到函数的类型转换.
- 当一个非类型的值被用作类型时，Lean 会尝试插入一个到类型的类型转换.
- 这三种转换所使用的函数都通过类型类的方式进行搜索，并将所用函数插入到表达式中.
- 如果所用函数标记为类型转换，Lean 会将它显示为箭头.
- 更具体的介绍见 [Theorem Proving in Lean 4].

# 歧义

一般来说，我们需要保证类型类查找到的结果是唯一的（所有可能的结果都是按定义相等）.

- 如果我们在整数上定义了两个加法，那我们谈论整数上的加法时，到底在谈论什么呢？
- 需要考虑不同的类型类推导路径，而不仅仅是不去写多个定义.
- Prop 值的类型类没有这个问题.

# 搜索合成效率

Mathlib 的类型类系统已经极为庞大，推导过程可能极为复杂，在依赖较多的文件中可能导致证明检查超时.

- 前几个月我尝试对代数结构的类型类搜索顺序做了一些调整，让 mathlib 构建时间中的类型类搜索时间加速了 $25\%$，整体加速了 $10\%$. 但是影响范围较大，可能短时间内无法合并.
- 类型类的搜索合成算法与 mathlib 的类型类设计可能都还有很大的改进空间.

# 结构的层次

数学结构形成了一个复杂的层次关系.

- 一个结构可以附加很多性质. 不同的性质组合会得不同的概念.
- 带有某些结构的对象之间的互动.
- 结构之间的互动.
- 结构之间可能有天然的依赖关系或自然诱导的关系.

# 例：Mathlib3 中赋范域 normed_field 下的层次

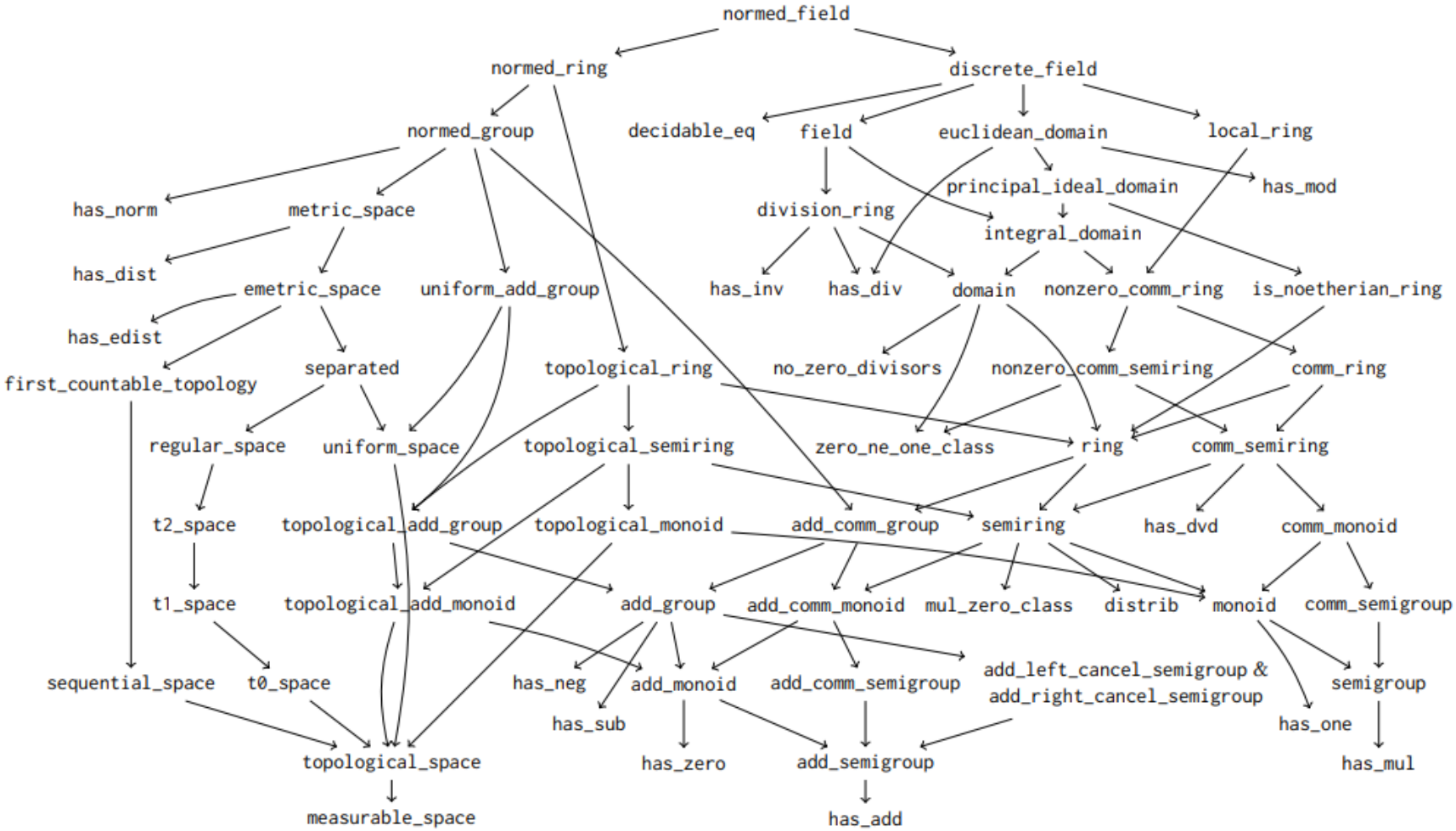


**Figure 1.** The structure hierarchy underlying `normed_field`. An edge from `S` to `T` indicates that an instance of `T` can be derived

# Bundle

很多简单而基本的数学结构都被设计为一个单独的类型类. 如 `Group` `LinearOrder` 等.
一般是通过 extend 更加基本的类型类最终得到的. 扩展关系形成了一个层次结构.
每一种性质组合都需要一个单独的类型类，甚至可能不得不添加一些并不会实际使用到的组合.

- `Group` → `DivInvMonoid`, `GroupWithZero` → `DivInvMonoid`
- `NormedLinearOrderedField` → `NormedField`, `NormedLinearOrderedField` → `LinearOrderedField`

# Mixin

在要求已有一些类型类的基础上，引入新的类型类要求一些额外的性质. 如 `IsDomain T2Space LocallyConvexSpace` 等.
更典型的例子是“带有某些结构的对象之间的互动”.

- `Module R M` 首先要求 `R` 是一个 `Semiring`，`M` 是一个 `AddCommMonoid`.

对于“一个结构上有很多可附加的性质”的情况，mixin 可以让引入的类型类数量不至于发生组合爆炸. 但引入的类型类参数会变得更多，条件会变得混乱.

# 对性能的影响

两种方式都可能会减慢类型类搜索速度.

- 在层次结构中扩展出新的类型类时，每次找某个子类时都可能会检查到这个新的类型类.
- mixin 会增加表达式的大小，增大搜索范围. 此外能推出一些很基本的类型类时，可能会引起大量不相关的类型类搜索.

# Mathlib 的一些基本现状

根据之前的讨论，一个整体性比较强的结构应该使用捆绑的方法，而结构上一些比较次要的性质、不同对象和对象上有较弱互动的不同结构则应该使用 mixin.

- 拓扑代数结构已经是 mixin 的形式.
- 赋范代数结构还是整体捆绑的形式，但重构为 mixin 的建议已经被多次提及，在 mathlib3 时期就已有一个未完成的 PR.
- 有序代数结构的有两套 API 并存，Damiano 等人和我完成了一套很底层的 mixin API，原有的 API 仍然存在并多次提出改为 mixin 的计划.

# 消除 diamond 的一些例子

对于一个 instance，如果它不是为特定类型进行的定义，而是一个一般的推导，那么应当确保它给出的定义足够自然、典范. 否则应该再三确认把它设置为 instance 是否真的合适.
还有一种情况是通常来说足够典范，但在一些特定情况下会引起麻烦. 通常的解决方案是改为它的条件中的类型类的一部分，将原本的值设为默认值，在必要的时候手动定义.

# 度量空间 → 拓扑空间

这一诱导关系极为重要. 至少我们不可能对度量结构重新写一遍拓扑结构的 API.
但是这个定义有时又不够典范，度量本身和拓扑结构相对独立.

- 积空间有积拓扑，积度量同样能诱导拓扑，它们不定义相等.
- 在不同时候我们会考虑矩阵的不同的范数，它们诱导的度量不一样，但为了方便应该保持拓扑相同.

Mathlib 的 `MetricSpace` 中定义了默认的拓扑结构，但允许在需要时手动提供定义.

# $\{\mathbb{N}, \mathbb{Z}, \mathbb{Q} \geq 0, \mathbb{Q}\}$-类型转换

自然数到半环、整数到环有自然的嵌入和作用，并且在 mathlib 中最终定义了相应的代数 instance.
在 2 年前，它们的类型转换是一个单独的定义，引起了很大的麻烦. 这导致了它们到自身的代数与 Algebra.id 的定义不一样，需要常常进行转换.
后来相关的定义改成了 `AddMonoidWithOne SubNegMonoid DivisionSemiring DivisionRing` 的一部分. 解决了这一问题.
但后来又引起了一些麻烦，见 [lean4#2950] [#6262] [#12608]

# 元编程

# 元编程

元编程是 Lean 的极为重要的功能，可以说没有 Lean 的元编程就没有 Lean.
很遗憾的是完全不可能在有限的时间内详细讲解 Lean 的元编程. 想要讲元编程可能得一开始把上午下午上课变成上午下午晚上上课.
有兴趣的同学可以尝试阅读 [Metaprogramming in Lean 4].

# 语法和表达式

语法和表达式是元编程主要操作的对象.

- Lean 允许定义语法和相应的语义.
- Lean 用归纳类型表示解析出的语法和表达式.
- Lean 将最终的表达式交给 kernel 进行类型检查，将一切证明的正确性转变为几百行 kernel 代码的正确性.

# Macro & Elaboration

- Macro 将语法变为另一个语法，常用于定义新的语法和语法糖.
  - 各种运算符、if-then-else 语法等都是用 macro 定义的.
- Elaborator 用于解读语法，多数时候解读为表达式.
  - 展示一些 term elaborator.
    - [#15192].

# Pretty Printing

- 展示一些 delaborator.

# Tactic

- 解释一些 tactic 的工作原理.

# 计算

# 计算

Lean 有两种意义上的计算.

- 根据类型论的规则对表达式进行化简.
  - 只需信任 kernel 类型检查器.
  - #reduce、decide 使用这种计算.
- 编译成更高效的代码执行.
  - 可以将编译器生成的代码替换掉.
    - 替换为更高效的实现，如 Nat、Array、csimp.
    - 调用各种与现实世界交互的常规编程功能.
  - 元编程代码通过这种方式执行.
  - 需要额外信任编译器和替换的代码，但仍然要尽可能保证结果与类型论一致.
  - #eval、native_decide 使用这种计算.

# reduce

- 公理与不透明或标记为 irreducible 的定义会阻止 reduce.
  - 有时为了加速类型检查会使用 irreducible.
- decide tactic 将相应的 `Decidable` instance reduce 为 isFalse 或 isTrue 来判定命题.
  - 如果使用了公理或不透明的定义则无法判定.
  - 如果使用了 irreducible 的定义则需使用 with_reducible.

# 有限集的势

根据对计算的需求，API 也会分为可计算和不可计算两类：

| | 可计算 | 不可计算 |
|---|---|---|
| 类型 | `Fintype`, `Fintype.card` | `Finite`, `Nat.card` |
| 集合 | `Finset`, `Finset.card` | `Set.Finite`, `Set.ncard` |

注意 `Finite` 是 Prop 而 `Fintype` 不是，若能使用 `Finite` 则尽量使用，并且不要定义不可计算的 `Fintype` instance.

# eval

- 证明和类型不参与计算，可以利用 Lean 的类型论做一些特有的优化.
- 生成的代码速度很快，在多数测试上超过了 Haskell 和 OCaml.
- 没有提供相应代码的公理会阻止 eval.
  - 无法生成对应代码的定义都会被标记为 `noncomputable`.
- 未被替换实现的定义最终会编译到 C，然后由 C 编译器编译出可执行文件.

# partial

- Lean 的类型论中只能写出能证明停机的函数.
- 实际使用的程序中有些函数未必停机，或无法证明停机，或难以证明停机.
- partial 定义允许用户编写不提供终止证明的函数并生成相应代码.
- 在类型论中是一个不透明的定义.
- 不透明的定义需要提供 `Nonempty` instance，所以不会产生矛盾.

# 我们可以用 Lean 算什么

- 当成一般的函数式编程语言使用.
  - Lean4 的编译器主要就是用 Lean 实现的.
- 进行元编程，写 tactic.
  - Lean 检查的是程序算出的表达式或“证书”，元程序本身没有任何终止或正确性要求.
- 写可以验证正确性的程序.
  - 想在写出高效程序的同时能证明正确性很有挑战性.
  - 亚马逊云科技已经在一些软件中使用 Lean 验证了关键部分.
- 将形式证明与自动定理证明器与计算机代数系统结合起来.
  - 是 Lean 社区正在做的事.

# 广告时间

我的本科毕业设计是在 Lean 中实现一些数据结构与算法并证明它们的正确性.

- [Algorithm].
- 作为一个库还在很早期的开发阶段.
- ~~大家快来给我点 star! !1~~

# Lean 与计算

- [leansat] Lean 社区正在尝试用 Lean 实现经过验证的 SAT 求解器.
- 社区讨论过若干次用 Lean 实现计算机代数系统或让计算机代数系统生成用 Lean 检查的证书.
- 可视化证明或者代码库的内容，例如 [lean-graph]

# 谢谢