

数理逻辑

如何验证数学证明的正确性

作者：逯晓零

组织：万物联盟数学分部

时间：2023/08/01

版本：1.0



ElegantLaTeX Program

数学之所以比一切其它科学受到尊重，一个理由是因为他的命题是绝对可靠和无可争辩的，而其它的科学经常处于被新发现的事实推翻的危险。—Albert Einstein(万物联盟自然部常驻大使)

目录

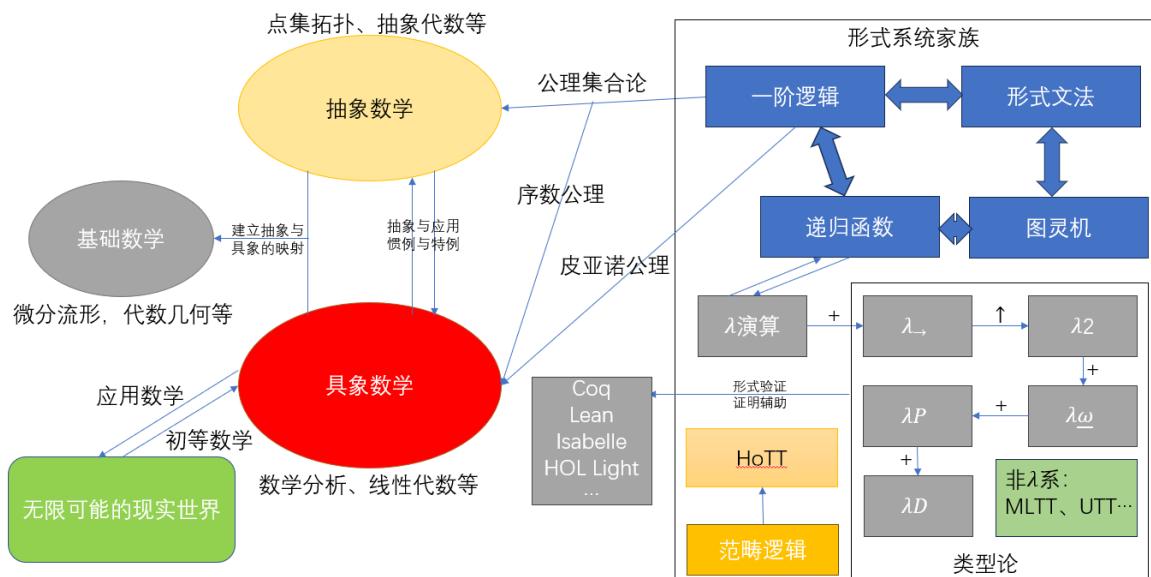
第一章 导言	1
第二章 逻辑入门	2
2.1 基础知识	2
2.2 不完全性	8
第三章 模型论简介	11
3.1 基础知识	11
3.2 非标准模型	23
第四章 计算与递归	27
4.1 几个系统	27
4.2 经典问题	37
第五章 数学基础其一：公理集合论	43
5.1 ZFC 公理	43
5.2 其它系统	47
5.3 连续统	48
第六章 数学基础其二：类型论	49
6.1 基础知识	49
6.2 逻辑内涵	52
6.3 集合论	54
第七章 证明辅助工具	56
7.1 Coq	56
7.2 Lean	72
7.3 Isabelle	77
7.4 HOL Light	82
7.5 Metamath	85
7.6 Mizar	86
7.7 ProofPower	86
7.8 ACL2/nqthm	87
7.9 PVS	87
7.10 NuPRL/MetaPRL	87
7.11 Haskell	88
7.12 Agda/Idris	89
7.13 Arend	89
7.14 F*	89
参考文献	90

第一章 导言

数理逻辑¹虽然是数学的基础，但我个人并不推荐直接开学，而是拥有了一定的数学基础再去考虑，如果实在没什么可以看的书，可以看看我以前的文章 [11–13]。值得注意的是，“数理逻辑与基础”²和哲学中“逻辑学”是两个不同东西，前者偏向于数学本身，而后者更偏向于思辨，笔者我作为一个融会贯通的读书人，本来是十分厌倦分类这件事的，但是科目的分类对于书籍的寻找是十分重要的，对于一些十分相近的名称，你很有可能找到一些不是你想要的参考书籍，比如你单纯使用“逻辑学”去找书籍的话，极大概率找到的都是思辨类的书籍，而不是我在这篇文章中所讨论的内容。数学并未抢到“逻辑学”这个名称，主要还是因为历史原因，或者说只是单纯的时间前后问题，“数理逻辑”正式成型是在 20 世纪，至于“逻辑学”的历史就不用多说了。虽然数理逻辑现在基本都是作为哲学系的课程，但作为纯正的数学生了解一下还是有必要的。

数理逻辑本质上是一门研究形式系统的学科，什么是形式系统？我们可以把它理解为一种“关于语言的规则游戏”，例如我们定义这个系统中有什么符号（字）、这些符号该如何组成原子对象（词）、这些原子对象该如何形成公式（句）、这些公式之间能怎么用起来或玩起来的规则。这里面的最后一步就十分重要，比如逻辑推导，它实际上就是给出一些基本公式（公理）、再告诉你公式的演化规则（推导规则），从而得到其它公式的游戏；再比如计算机，其相当于提前预定好游戏规则（程序）、从而对于每一个输入给出输出（计算），从而实现文字转化的游戏。形式系统最显著的特点就是不需要任何的前置知识，这意味着人人都可以玩起来，人人都可以设计属于自己的系统，不过难点在于你是否有足够的想象力来支撑你的设计内容，因此一个形式系统理论能否发展起来主要取决于，它有没有用？有没有人愿意使用它？有用体现在两个方面，其在一个新的领域完成了自己的工作，又或者在旧领域上有改革性的创新。

我们作为研究数学的人，对于形式系统的理解自然应该偏向于数学方向，那么我们真正需要理解的无非就是逻辑这件事了。在本篇文章的第 2 章到第 5 章，我们主要探究传统数学的纸面文字游戏，也就是形式化的严格语言，其主要包括一阶逻辑、形式文法、递归函数和图灵机这几大等价工具，虽然我们有着众多形式的严格语言，但其依旧停留在纸面上，这意味着我们依旧可能存在书写的漏洞，我们需要有个东西来检验形式化语言中并不存在书写的错误。这就是第 6 章和第 7 章所讨论的内容了，我们作为 21 世纪的人，有一个时刻伴随我们发展的工具——计算机，是不能被忽略的，虽说其也可能存在漏洞，但其严格性绝对是当前之最，此时数学形式化的主流逐渐从纸面走向计算机，此时我们对数学严格性的信赖程度至少有百分之九十九了。



¹Open Logic Project: <https://openlogicproject.org/>, 数理逻辑资料汇总网站

²参考 MSC(Mathematics Subject Classification System) 中的 “Mathematical logic and foundations”

第二章 逻辑入门

我们先来介绍数理逻辑最基础的内容，并引入一些基本的符号和认知，它是后面所有内容的基础，所以要十分谨慎地去对待，我所采用的教材是这本 [9]，虽然是哲学系的教材，但我觉得是目前所见中讲得最清晰的一本书了。

2.1 基础知识

命题逻辑

形式(逻辑)系统一般由四个部分组成，分别是：初始符号、形成规则、初始公式、变形规则。当我们将其四条进行具体化时，就能形成一种逻辑，比如大家最常见的逻辑系统就是下面的命题逻辑。

定义 2.1 (命题逻辑)

(1)[初始符号]

(i)(不定形符号，无穷个) $p, q, r, p_1, p_2, p_3 \dots \in \mathcal{L}_0$

(ii)(定形符号，4个) $\neg, \rightarrow, (,)$

(2)[形成规则]

(i) 任意 $p \in \mathcal{L}_0$ 是公式

(ii) 若 $A, B \in \mathcal{L}_0$ 是公式，则 $\neg A, (A \rightarrow B) \in \mathcal{L}_0$ 是公式

(iii) 所有的公式均由(i)和(ii)生成

(3)[初始公式]

(P₁) $A \rightarrow (B \rightarrow A)$

(P₂) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

(P₃) $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$

(4)[变形规则]

(分离规则, MP) 如果 $A, A \rightarrow B$ ，则有 B 。



希望读者不要对我们稍微使用了一下集合的符号而感到不适，我们只是想单纯地表示归属， \mathcal{L}_0 用来表示所有公式构成的整体，以方便在(1)(i)所说的不定形符号的选出。我们先来聚焦于(1)(2)两个部分，来探究一下 \mathcal{L}_0 意味着什么，我们必须清楚它不是集合，因为它的元素完全不确定，比如我们写 $A, (A \rightarrow B) \in \mathcal{L}_0$ ，但实际我们可能有

$$A = p \rightarrow q, A = \neg q, A = p \rightarrow (q \rightarrow r), \dots (A \rightarrow B) = ((p \rightarrow q) \rightarrow B), (A \rightarrow B) = (\neg q \rightarrow B), \dots$$

通过“形成规则”，公式本身就是一种符合某些规则的样式，比如下面这些样式就不属于公式集

$$(p, pq \rightarrow, p \neg, \neg q), \wedge \rightarrow \vee, \vee \rightarrow pq \wedge qp, \dots$$

对于样式，我们自然有一些基础的“简写法”，以得到我们常见的“与”“或”符号

$$p \wedge q := \neg(p \rightarrow \neg q), p \vee q := \neg p \rightarrow q, p \leftrightarrow q := (p \rightarrow q) \wedge (q \rightarrow p)$$

在公式中，括号主要用来改变结合的优先级，如果我们不加括号，则默认的优先级为：(从高到低) $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ ；连续的 \rightarrow 从后往前结合。下面是几个加括号的例子

$$r \vee \neg p \vee q, (r \vee (\neg p)) \vee q$$

$$r \rightarrow p \rightarrow p \leftrightarrow \neg \neg p \vee q, (r \rightarrow (p \rightarrow p)) \leftrightarrow ((\neg(\neg p)) \vee q)$$

不过以我个人观点来看，除非是一元连接符 $\neg p$ ，就最好都用上括号，一方面是比较清晰，另一方面我们都把它放到符号中去了，就没有不用的理由。在有些书籍中，会将“原子命题” p, q, r, \dots 和“公式” $A \wedge q, p \vee q, r \vee \neg B \vee q, \dots$ （小写表示原子命题，大写表示公式，公式中带公式表示一类公式）区分成两个东西，这样可以方便地讨论真值问题，但我个人觉得完全没有必要，因为无论是公式还是赋值本质都是递归定义的，这种递归性我们后面会详细讨论，现在就先放一放，以“替换”的思想来理解即可。

定义 2.2 (赋值)

所谓赋值指映射 $v : \mathcal{L}_0 \rightarrow \{0, 1\}$ 满足

- (1) 任意 $A \in \mathcal{L}_0$ 都有 $v(\neg A) = 0 \Leftrightarrow v(A) = 1$
- (2) 任意 $A, B \in \mathcal{L}_0$ 都有 $v(A \rightarrow B) = 0 \Leftrightarrow v(A) = 1, v(B) = 0$



上面讲得或许有些抽象，但其实很好理解，赋值其实就是把公式同构于布尔代数，虽然一般布尔代数得系统形式为 $\langle B, \wedge, \vee, \neg \rangle$ ，但其实可以简化为系统 $\langle B, \wedge, \neg \rangle$ 或 $\langle B, \vee, \neg \rangle$ ，并且和下面得系统都是等价的

$$\langle B, \neg, \rightarrow \rangle, \langle B, \neg, \leftrightarrow \rangle, \langle B, \neg, + \rangle, \langle B, | \rangle, \langle B, \downarrow \rangle, \langle B, \perp, \rightarrow \rangle$$

上面几个特殊符号的定义如下

$$A + B := (A \vee B) \wedge \neg(A \wedge B), A | B := \neg(A \wedge B), A \downarrow B := \neg(A \vee B), A \perp B := 0$$

这些事实只需考察布尔函数的决定因素即可，所以我们说的等价指真值表上是可以互相表示的，在没有赋值的情况下，符号就是符号，不存在等价的概念。

定义 2.3

对任意公式 $A \in \mathcal{L}_0$

- (1) 如果存在赋值 v 使得 $v(A) = 1$ ，则称 A 是可满足的
- (2) 如果不存在赋值 v 使得 $v(A) = 1$ ，则称 A 是不可满足的
- (3) 如果对所有的赋值 v 均有 $v(A) = 1$ ，则称 A 是重言式，并记 $\models A$
- (4) 如果对所有的赋值 v 均有 $v(A) = 0$ ，则称 A 是矛盾式，并记 $\not\models A$



这样看来赋值好像不止一个，没错，赋值其实并不代表指派真假值的规则，而是一次真假值的指派，例如一个简单的情形 $p \wedge q$ 就有四种赋值

$$p = 1, q = 1, v_1(p \wedge q) = 1; v_2(p \wedge q) = v_3(p \wedge q) = v_4(p \wedge q) = 0$$

如果我们引入“原子命题”概念，那么一个赋值实际就代表对这些原子命题进行了一次真假值指派，并且通过赋值满足的性质唯一延拓到了所有的公式上。我们注意到替代这种行为并不会改变重言式的性质，而后面最需要的也就是重言式和矛盾式了，所以我才会说引入原子命题的概念是完全没必要的。考虑完单个公式，自然需要考虑公式集了 $\Gamma \subset \mathcal{L}_0$ ，我们简记

$$v(\Gamma) = 1 \Leftrightarrow A \in \Gamma, v(A) = 1$$

定义 2.4

设 $\Gamma \subset \mathcal{L}_0, A \in \mathcal{L}_0$ ，对任意赋值 v 如果有 $v(\Gamma) = 1$ 则有 $v(A) = 1$ ，则称 Γ 重言蕴含 A ，并记为 $\Gamma \models A$ 。



证明某个公式是否为重言式，无非就是写出形式并进行真假值指派地验证即可，并且重言式和矛盾式有个最简单的对应，即重言式 A 和矛盾式 $\neg A$ 。到目前我们并未涉及证明和定理，而这就是(3)(4)的内容了，我们把 $P_1, P_2, P_3 \in \mathcal{L}_0$ 称为命题逻辑 $L = (\mathcal{L}_0, P = \{P_1, P_2, P_3\}, MP)$ 的公理，并引入证明的概念

定义 2.5 (证明)

设命题逻辑 $L = (\mathcal{L}_0, P, MP)$ 和公式集 $\Gamma \subset \mathcal{L}_0$ ，如果有序公式列 A_1, \dots, A_n 满足对任意的 $A_i \in \mathcal{L}_0, 1 \leq i \leq n$

下面之一成立 (1) $A_i \in P$

(2) $A_i \in \Gamma$

(3) A_i 由 $A_j, A_k (j < i, k < i)$ 通过 MP 得到

则称 A_1, \dots, A_n 是以 Γ 为前提的演绎，或称为 A_n 在 L 中以 Γ 为前提的证明，记做 $\Gamma \vdash_L A_n$ 。



如果有 $\emptyset \vdash_L A$ (可以简记为 $\vdash_L A$)，我们就把 A 称为 L 中的定理。显然与之对应的 $\emptyset \models A$ 表示 A 是重言式，它与定理的区别在于其不需要“公理”和“证明”，只需要单纯的真假值验证，因此对于一个形式系统，我们需要考虑哪些“公理”和“规则”可以证明出所有的重言式，这就是形式系统的可靠性和完全性。

定理 2.1 (命题逻辑)

(1)[可靠性] 如果 $\vdash_L A$ ，则有 $\models A$

(2)[一致性] 对任意的 $A \in \mathcal{L}_0$, $A, \neg A$ 不能都是 L 中的定理

(3)[紧致性] 公式集 Γ 是可满足的当且仅当 Γ 的每个有限子集是可满足的

(4)[完全性] 如果 $\models A$ ，则有 $\vdash_L A$



一些证明的例子和上面四个性质的证明就看我开头说的参考书，后面也是同理，我们的目标在于更高更远的地方，读者只需迅速吃透这些思想即可，说到底数理逻辑主要还是为了我们数学的证明做支撑，并说明证明可能产生的一些现象。

一阶逻辑

定义 2.6 (谓词逻辑)

(1)[初始符号]

(i)(常量，无穷个) $c_1, c_2, c_3, \dots \in C$

(ii)(变量，无穷个) $x_1, x_2, x_3, \dots \in V$

(iii)(谓词，无数个) $P_1^1, P_1^2, \dots; P_2^1, P_2^2, \dots \in \mathcal{P}$

(iv)(函数，无数个) $F_1^1, F_1^2, \dots; F_2^1, F_2^2, \dots \in \mathcal{F}$

(v)(定形符号，5个) $\neg, \rightarrow, (,), \forall$

(2)[形成规则]

(i) 任意 $c \in C, x \in V$ 是项 $c, x \in \mathcal{I}$

(ii) 对任意 $F_i^n \in \mathcal{F}$ 和 $t_1, \dots, t_n \in \mathcal{I}$ ，有 $F_i^n(t_1, \dots, t_n) \in \mathcal{I}$ 是项

(iii) 所有的项均由 (i) 和 (ii) 生成

(iv) 对任意 $P_i^n \in \mathcal{P}$ 和 $t_1, \dots, t_n \in \mathcal{I}$ ，有 $P_i^n(t_1, \dots, t_n) \in \mathcal{L}_1$ 是公式

(v) 若 $x \in V$ 且 A, B 是公式，则 $\neg A, (A \rightarrow B), (\forall x A) \in \mathcal{L}_1$ 是公式

(vi) 所有的公式均由 (iv) 和 (v) 生成

(3)[初始公式]

(P_1) $A \rightarrow (B \rightarrow A)$

(P_2) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

(P_3) $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$

(代入公理, S) $(\forall x A) \rightarrow A(x; ; t)$

(分配公理,D) $\forall x(A \rightarrow B) \rightarrow (\forall xA \rightarrow \forall xB)$

(概括公理,C) $A \rightarrow \forall xA, x \notin \text{Fr}(A)$

(4)[变形规则]

(分离规则,MP) 如果 $A, A \rightarrow B$, 则有 B 。



谓词逻辑也可以称为一阶逻辑, 是我们目前最常使用的逻辑, 通过它即可以构造出集合论, 并走向整个数学, 不过这都是后话, 我们先来着眼于当前的内容。在一阶逻辑中, 我们发现了“谓词”和“公式”的概念正好对应了, 前面我们所说的命题逻辑中的“原子命题”和“公式”, 同样我们先来考察(1)(2)中的项和公式, 下面是常用符号“存在”的缩写

$$\exists x A := \neg \forall x(\neg A)$$

在优先级上, \forall 大于 \neg , 因此更详细的写法为 $\exists x A := \neg(\forall x(\neg A))$ 。另外读者要注意了, 我们在无限的符号中都加了下标, 并不是要表明它们都是可数的, 只是想要单纯地区分不同的符号罢了, 而在函数和谓词上特地加的上标则用来表明其变量的个数, 所谓的“项”其实就是“数值”或“变量”, 只有套上了谓词才能真正进入公式体系

项: $c_1, x_4, F_1^2(c_2, x_9), F_1^3(F_1^1(x_2), c_2, x_6), \dots$

公式: $P_1^2(c_2, x_3), P_2^3(F_1^2(c_2, x_9), c_2, x_3) \rightarrow \neg A, \forall x_3(P_2^3(F_1^2(c_2, x_9), c_2, x_3) \rightarrow (B \rightarrow A)), \dots$

理解起来并不困难, 这里的大写 A, B 均代表某个公式, 我们记 $\text{sig}\mathcal{L}_1 = C \cup \mathcal{P} \cup \mathcal{F}$, 并把基数 $|\text{sig}\mathcal{L}_1|$ 称为 \mathcal{L}_1 的大小, 其决定了形式系统是否为有限的、可数的或不可数的。通上面的内容可知, 任何一个公式 $A \in \mathcal{L}_1$ 都应该是存在变量的, 但如果我们不把它显示出来就难以知道其情况, 故我们通过递归的方式把这些变量给放进一个集合。

定义 2.7

(1) 设 $t \in \mathcal{I}$ 是项, $A \in \mathcal{L}_1$ 是公式, 则变量集 $\text{Vr}(t)$ 由下面递归定义

- (i) 如果 $t \in \mathcal{V}$, 则 $\text{Vr}(t) = \{t\}$
- (ii) 如果 $t \in C$, 则 $\text{Vr}(t) = \emptyset$
- (iii) 如果 $t = F_i^n(t_1, \dots, t_n)$, 则 $\text{Vr}(t) = \text{Vr}(t_1) \cup \dots \cup \text{Vr}(t_n)$

变量集 $\text{Vr}(A)$ 由下面递归定义

- (i) 如果 $A = P_i^n(t_1, \dots, t_n)$, 则 $\text{Vr}(A) = \text{Vr}(t_1) \cup \dots \cup \text{Vr}(t_n)$
- (ii) 如果 $A = \neg B$, 则 $\text{Vr}(A) = \text{Vr}(B)$
- (iii) 如果 $A = B \rightarrow C$, 则 $\text{Vr}(A) = \text{Vr}(B) \cup \text{Vr}(C)$
- (iv) 如果 $A = \forall x B$, 则 $\text{Vr}(A) = \text{Vr}(B) \cup \{x\}$ 。



变量集 $\text{Vr}(t) \subset \mathcal{V}$ 本身的意义其实不大, 我们需要的是它的一个子集, 即考察哪些变量是自由的

定义 2.8 (自由变量)

(1) 设 $t \in \mathcal{I}$ 是项, $A \in \mathcal{L}_1$ 是公式, 则自由变量集 $\text{Fr}(t)$ 由下面递归定义

- (i) 如果 $t \in \mathcal{V}$, 则 $\text{Fr}(t) = \{t\}$
- (ii) 如果 $t \in C$, 则 $\text{Fr}(t) = \emptyset$
- (iii) 如果 $t = F_i^n(t_1, \dots, t_n)$, 则 $\text{Fr}(t) = \text{Fr}(t_1) \cup \dots \cup \text{Fr}(t_n)$

自由变量集 $\text{Fr}(A)$ 由下面递归定义

- (i) 如果 $A = P_i^n(t_1, \dots, t_n)$, 则 $\text{Fr}(A) = \text{Fr}(t_1) \cup \dots \cup \text{Fr}(t_n)$

- (ii) 如果 $A = \neg B$, 则 $\text{Vr}(A) = \text{Vr}(B)$
- (iii) 如果 $A = B \rightarrow C$, 则 $\text{Vr}(A) = \text{Vr}(B) \cup \text{Vr}(C)$
- (iv) 如果 $A = \forall x B$, 则 $\text{Vr}(A) = \text{Vr}(B) - \{x\}$ 。

显然差别只是扣除 $\forall x$ 的变量, 但如果其摆脱了 $\forall x$ 的控制就又会变成自由的, 下面是几个简单的例子

$$A = \forall x_1 (P_1^2(x_2, x_3) \rightarrow P_1^2(c_2, c_3)) \rightarrow P_1^2(x_1, c_2), \text{Fr}(A) = \text{Vr}(A) = \{x_1, x_2, x_3\}$$

$$A = \forall x_1 (P_1^2(x_1, c_1)) \rightarrow \neg(P_1^2(F_1^2(x_2, x_3), c_1)), \text{Fr}(A) = \{x_2, x_3\} \neq \text{Vr}(A)$$

与命题逻辑不同的是, 我们可以给予 $\text{sig}\mathcal{L}_1$ 具体的内容并添加一定数量的公理, 以形成一阶理论, 数论、集合论、实数等都是具体化后的理论, 更一般地我们有下面的定义

定义 2.9 (解释)

如果结构 $M = (M, I)$ 满足, $M \neq \emptyset$ 且映射 $I : \text{sig}\mathcal{L}_1 \rightarrow M$ 有

- (1) 任意 $c \in C$ 均有 $I(c) \in M$
- (2) 任意 $P_i^n \in \mathcal{P}$ 均有 $I(P_i^n) \subset M^n$
- (3) 任意 $F_i^n \in \mathcal{P}$ 均有 $I(F_i^n) : M^n \rightarrow M$

则称 M 有 \mathcal{L}_1 - 结构, 并把 M 称为 M 的论域, I 称为 M 的解释。

所以说解释的含义其实就是, 将 $\text{sig}\mathcal{L}_1$ 的常数对应到一个集合 M , 谓词对应到 M 上的关系, 函数对应到 M 上的映射, 对于“公理添加”的操作我们后面再讨论。在命题逻辑中, 我们没有给出原子命题, 所以赋值只需要依赖于布尔代数即可, 而到了一阶逻辑上, 我们给出了类似于项的概念, 因此赋值则需要依赖于一个具体的解释。

定义 2.10 (赋值)

(1) 对于 \mathcal{L}_1 - 结构 $M = (M, I)$, 我们把映射 $v : \mathcal{V} \rightarrow M$ 称为一个 M - 赋值

(2) 一阶逻辑 \mathcal{L}_1 的一个 M - 赋值, 表示映射 $\bar{v} : \mathcal{I} \rightarrow M$ 满足

- (i) 如果 $t \in \mathcal{V}$, 则 $\bar{v}(t) = v(t)$
- (ii) 如果 $t \in C$, 则 $\bar{v}(t) = I(t)$
- (iii) 如果 $t = F_i^n(t_1, \dots, t_n)$, 则 $\bar{v}(t) = I(F_i)(\bar{v}(t_1), \dots, \bar{v}(t_n))$

(3) 对于公式 $A \in \mathcal{L}_1$, 我们称它在解释 $M = (M, I)$ 和赋值 $v : \mathcal{V} \rightarrow M$ 是可满足的, 并记为 $\models_{(M, v)} A$, 其递归定义如下

(i) 如果 $A = P_i^n(t_1, \dots, t_n)$, 则有 $\models_{(M, v)} A \Leftrightarrow (\bar{v}(t_1), \dots, \bar{v}(t_n)) \in I(P_i^n)$

(ii) 如果 $A = \neg B$, 则有 $\models_{(M, v)} A \Leftrightarrow \not\models_{(M, v)} B$

(iii) 如果 $A = B \rightarrow C$, 则有 $\not\models_{(M, v)} A \Leftrightarrow \not\models_{(M, v)} B, \not\models_{(M, v)} C$

(iv) 如果 $A = \forall x B$, 则有 $\models_{(M, v)} A \Leftrightarrow m \in M, \models_{(M, v_{x \mapsto m})} B$ 。此处的 $v_{x \mapsto m} : \mathcal{V} \rightarrow M$ 是诱导 M - 赋值, 定义如下

$$v_{x \mapsto m}(y) = \begin{cases} m & y = x \\ v(y) & y \neq x \end{cases}$$

(4) 对于公式 $A \in \mathcal{L}_1$, 只要存在一个解释 $M = (M, I)$ 和赋值 $v : \mathcal{V} \rightarrow M$ 使得 $\models_{(M, v)} A$, 就称 A 是可满足的

(5) 对于公式 $A \in \mathcal{L}_1$, 如果任意的解释 $M = (M, I)$ 和赋值 $v : \mathcal{V} \rightarrow M$ 均有 $\models_{(M, v)} A$, 就称 A 是有效式, 并记 $\vdash A$

(5) 对于公式 $A \in \mathcal{L}_1$, 如果任意的解释 $M = (M, I)$ 和赋值 $v : \mathcal{V} \rightarrow M$ 均有 $\not\models_{(M, v)} A$, 就称 A 是矛盾式,

并记 $\not\models A$



因此一阶逻辑中的有效式对应了命题逻辑中的重言式, 我们记一阶系统 $L = (\mathcal{L}_1, \text{sig} \mathcal{L}_1, P = \{P_1, P_2, P_3, S, D, C\}, \text{MP})$, 则接下来逻辑蕴含(对应重言蕴含) $\Gamma \models A$ 、前提证明 $\Gamma \vdash_L A$ 、定理 $\vdash_L A$ 的概念与之前完全一样, 需要注意的是在公理中出现了代入 $A(x; t)$, 它的递归定义如下

定义 2.11 (代入)

设 $t \in \mathcal{I}, x \in \mathcal{V}$

(1) 对于项 $t_0 \in \mathcal{I}$ 有

$$(i) \text{ 如果 } t_0 \in \mathcal{V}, \text{ 则有 } t_0(x; t) = \begin{cases} t & t_0 = x \\ t_0 & t_0 \neq x \end{cases}$$

$$(ii) \text{ 如果 } t_0 \in C, \text{ 则有 } t_0(x; t) = t_0$$

$$(iii) \text{ 如果 } t_0 = F_i^n(t_1, \dots, t_n), \text{ 则有 } t_0(x; t) = F_i^n(t_0(x; t_1), \dots, t_0(x; t_n))$$

(2) 对于公式 $A \in \mathcal{L}_1$ 有

$$(i) \text{ 如果 } A = P_i^n(t_1, \dots, t_n), \text{ 则有 } A(x; t) = P_i^n(t_1(x; t), \dots, t_n(x; t))$$

$$(ii) \text{ 如果 } A = \neg B, \text{ 则有 } A(x; t) = \neg B(x; t)$$

$$(iii) \text{ 如果 } A = B \rightarrow C, \text{ 则有 } A(x; t) = (B(x; t) \rightarrow C(x; t))$$

$$(iv) \text{ 如果 } A = \forall y B, \text{ 则有 } A(x; t) = \begin{cases} \forall y B & y = x \\ \forall y B(x; t) & y \neq x \end{cases}$$



代入和存在一样也是一种公式的记号, 而“代入” $A(x; t)$ 其实就是字面的意思, 将公式 A 中所有自由变量 x 替换成 t , 当然如果 $x \notin \text{Fr}(A)$ 不是自由变量, 自然有 $A(x; t) = A$ 。但这样代入有一定风险, 比如我们将 x 替换成了非 $\text{Fr}(A)$ 中的元素, 从而使得公式发生了含义的改变, 因此我们需要进一步限制 t , 从而引入自由代入 $A(x; ; t)$ 的概念, 或者说 t 在 A 中相对于 x 是自由的。

定义 2.12 (自由代入)

(1) 对任意项 $t_0 \in \mathcal{I}$, t 在 t_0 中相对于 x 都是自由的

(2) 对于公式 $A \in \mathcal{L}_1$ 有

(i) 如果 $A = P_i^n(t_1, \dots, t_n)$, 则 t 在 A 中相对于 x 是自由的

(ii) 如果 $A = \neg B$ 且 t 在 B 中相对于 x 是自由的, 则 t 在 A 中相对于 x 是自由的

(iii) 如果 $A = B \rightarrow C$ 且 t 在 B, C 中相对于 x 是自由的, 则 t 在 A 中相对于 x 是自由的

(iv) 如果 $A = \forall y B$ 且 $x \notin \text{Fr}(A)$, 则 t 在 A 中相对于 x 是自由的

(v) 如果 $A = \forall y B$ 且 $x \in \text{Fr}(A), y \notin \text{Fr}(t)$ 且 t 在 B 中相对于 x 是自由的, 则 t 在 A 中相对于 x 是自由的



简单来讲“自由代入”其实就是限制了 t 不能是任意的项, 我们需要保证非自由变量不能出现在 t 中。对于一阶逻辑 $L = (\mathcal{L}_1, \text{sig} \mathcal{L}_1, P, \text{MP})$, 剩下的就是和命题逻辑一样的可靠性和完全性了, 由下面的定理完成概括。

定理 2.2

(1)[可靠性] 如果 $\vdash_L A$, 则有 $\models A$

(2)[完全性] 如果 $\models A$, 则有 $\vdash_L A$

(3)[紧致性] 公式集 Γ 是可满足的当且仅当 Γ 的每个有限子集是可满足的



读者需要注意一阶逻辑不一定具有一致性(无矛盾性¹), 但比命题逻辑多了一个“可判定性”, 这个我们后面会进行讨论。哥德尔在证明上述两性时是建立在 $\text{sig} \mathcal{L}_1$ 是可数的情况下, 但后来马尔西夫和亨金都将其推广到

¹准确来讲是我们无法证明一阶逻辑的一致性

了不可数的情形，所以我们之前说下标并不是要说明无穷符号是可数的。

高阶逻辑

至于更高阶的逻辑，比如二阶逻辑、三阶逻辑等，其运用并不深入，故本身也没什么探究的必要，但作为介绍我们还是讲讲“阶”意味着什么？在一阶语言中 $\forall x$ 中只能有 $x \in \mathcal{V}$ ，如果我们让函数谓词 P_i^n 和 F_i^n 也可以放在其中形成语句 $\forall P_i^n$ 和 $\forall F_i^n$ ，从而使得其可以表达更多的语句，其中最大的 $n+1$ 就是语言的阶数，比如下面是一个二阶逻辑的公式

$$\forall P \exists x (P(x) \rightarrow \forall x P(x))$$

按照传统的理解可以是，“对任意的关系（或函数）满足 XXX”。而这类东西，一旦我们建立集合的观念，那么关系（或函数）这种东西，就可以视为关系（或函数）论域中的常数，就自然地归属到一阶逻辑中，因此大多书籍都不会过于详细地讨论高阶逻辑，想要研究也不是不行，只是必要性不高罢了（地址）。类似地，还有多值逻辑（Many-valued logic）、模态逻辑（Modal logic）等形式系统，但对于理解的意义不大，等我们需要的时候再提吧。

2.2 不完全性

一阶理论

接下来，我们以一阶逻辑 $L = (\mathcal{L}_1, \text{sig} \mathcal{L}_1, P, \text{MP})$ 为讨论前提。如果公式 $A \in \mathcal{L}_1$ 满足 $\text{Fr}(A) = \emptyset$ ，我们就把 A 称为语句（即不包含自由变量），我们把所有语句构成的整体记为 $\text{Se}(\mathcal{L}_1)$ 。

定义 2.13 (一阶理论)

设 $T \subset \text{Se}(\mathcal{L}_1)$ ，如果公式 $A \in \mathcal{L}_1$ 满足 $T \vdash A$ 就有 $A \in T$ ，则称 T 是一阶理论，或称为 \mathcal{L}_1 -理论。

所谓的一阶理论难道就是指一个语句集 T ？比如下面的两个例子

$$\Gamma \subset \text{Se}(\mathcal{L}_1), \text{Th}(\Gamma) = \{A \in \text{Se}(\mathcal{L}_1) \mid \Gamma \vdash A\}$$

$$\mathcal{L}_1 = \{A \in \text{Se}(\mathcal{L}_1) \mid \vdash A\} = \text{Th}(\emptyset)$$

实际上我们可以用另一种观点来看待，假设一阶理论 T 只需依靠几个公理就能推出语句集 T 中的所有元素，即存在有限语句集 Γ 使得 $T = \text{Th}(\Gamma)$ ，则称 T 可被 Γ 有限公理化。那么我们可以在一阶逻辑 L 的初始公式中将语句集 Γ 加上，从而形成一个新的形式系统，这样一阶理论 T 就相当于这个形式系统的定理集。我们之所以能这样做的原因是因为 $\text{sig} \mathcal{L}_1 = C \cup \mathcal{P} \cup \mathcal{F}$ 的元素没有被具体化，导致实际能推出的定理是不足的，如果我们通过引入一些限制谓词和函数的公理，那么系统就会被相应的扩大，定理也会相应的变多。比如，在许多数学理论中都要用到的“等号”，我们将其引入谓词 $P_1^2(x_1, x_2)$ 并简单地记为

$$x_1 = x_2 := P_1^2(x_1, x_2)$$

从而得到一阶（带等号）理论，其引入了下面的五条相等公理

$$(E_1)t = t \quad (E_2)t_1 = t_2 \Rightarrow t_2 = t_1 \quad (E_3)t_1 = t_2 \rightarrow (t_1 = t_3 \rightarrow t_2 = t_3)$$

$$(E_4)t = u \rightarrow F_i^n(..., t, ...) = F_i^n(t; u) \quad (E_5)t = u \rightarrow (P_i^n(..., t, ...) \rightarrow P_i^n(t; u))$$

就如我们之前所说下标只是为了区分出 \mathcal{P} 中的元素，其记号是取之不尽用之不竭的，所以对于其它谓词写

成 $P_1^2(t_1, t_2)$ 也是无所谓的，因为这个特定的谓词已经被写成了 $= (t_1, t_2)$ 。虽然一阶逻辑 L 无法讨论一致性²，但我们可以考虑公式集 $\Gamma \subset \mathcal{L}_1$ 的一致性。如果对任意公式 $A \in \mathcal{L}_1$, $\Gamma \vdash_L A$ 和 $\Gamma \vdash_L \neg A$ 不能同时成立，就称 Γ 在 L 中是一致的，我们实际可以把 \emptyset 在 L 中的一致性视为 L 的一致性，类似地把理论 T 在 L 中的一致性视为 T 的一致性。

定理 2.3 (不一致性)

对于公式集 $\Gamma \subset \mathcal{L}_1$, 下面的命题等价

- (1) Γ 在 L 中不是一致的
- (2) $\Gamma \vdash \perp$
- (3) $\Gamma = \mathcal{L}_1$
- (4) 任意 $A \in \mathcal{L}_1$ 都有 $\Gamma \vdash A$
- (5) 存在公式 $A \in \mathcal{L}_1$ 使得 $\Gamma \vdash A \wedge \neg A$
- (6) 任意 $A \in \mathcal{L}_1$ 都有 $\Gamma \vdash A \wedge \neg A$



在一阶逻辑中，不一致性是十分奇怪的，我们一旦在公理中加入矛盾命题 $\{A, \neg A\}$ ，那么所有公式 \mathcal{L}_1 就会直接全变成定理，不一致的情形只有 \mathcal{L}_1 ，但问题在于我们不知道我们的系统是否为不一致的，对此我们有必要马上进入下一个课题。

定理 2.4 (一致性)

公式集 $\Gamma \subset \mathcal{L}_1$ 是一致的当且仅当 Γ 是可满足的。



哥德尔不完全性定理

接下来，我们要在一阶(带等号)理论上继续扩充，为了表示一阶理论与一阶逻辑 L 的差距，我们将其通过公理特殊化后的常数、谓词和函数写进其中，比如一阶(带等号)理论我们记为 $(L, =)$ ，表示在一阶逻辑 L 的基础上还有一些有关等号“=” 的公理。目前大部分的初等数学理论，都是建立在初等数论的基础上，其公理系统在形式系统中对应的就是一阶算术理论 $N = (L, =, 0, S, +, \times)$ ，其在一阶(带等号)理论的基础上，添加了一个常量 $0 := c_1$ 和三个函数

$$S(x) := F_1^1(x), x_1 + x_2 := F_1^2(x_1, x_2), x_1 \times x_2 := F_2^2(x_1, x_2)$$

我们把这三个函数称为后继、加法和乘法，添加一个符号缩写 $x \neq y := \neg(x = y)$ ，相应的限制公理为

$$(N_1) \forall x(S(x) \neq 0) \quad (N_2) \forall x \forall y(S(x) = S(y) \rightarrow x = y)$$

$$(N_3) \forall x(x + 0 = x) \quad (N_4) \forall x \forall y(x + S(y) = S(x + y))$$

$$(N_5) \forall x(x \times 0 = 0) \quad (N_6) \forall x \forall y(x \times S(y) = x \times y + x)$$

$$(N_7) \forall x(x \neq 0 \rightarrow \exists y(x = S(y)))$$

注意上面的是罗宾森算术，而不是皮亚诺算术，而且函数的结合优先级大于谓词。哥德尔不完全性定理的证明涉及后面的理论和技巧，所以怎么证明我是不会讲的，现在不会以后也不会，主要还是基本任何一本数理逻辑的书都会讲，实在过于常见了，我们的目的是严格地叙述它。

² 哥德尔第二不完全性定理，我们后面会讨论

定理 2.5 (哥德尔-罗瑟第一不完全性定理)

设 T 是包含一阶算术理论 \mathcal{N} 且是可计算公理化的理论。如果 T 是一致的，则存在语句 $\rho \in \text{Se}(\mathcal{L}_1)$ 使得 $\not\vdash_T \rho$ 且 $\not\vdash_T \neg\rho$ 。



这里的可计算公理化指其生成公理集 $T = \text{Th}(\Gamma)$ 的 Γ 是可计算的，后面我们会详细说明，一个通俗的例子就是 Γ 是有限的且每个语句都是有限长度的，或者我们直接使用一阶算术理论 \mathcal{N} 。另外，原始的哥德尔第一不完全性定理要求理论 T 是 ω -一致，它比一致更强，是一致的特殊情况，所以我们干脆就跳过算了。由于在一阶逻辑中，任意公式 $\rho \in \mathcal{L}_1$ 均满足 $\models \rho$ 和 $\models \neg\rho$ 有且仅有一个为真，因此上面定理说明了，我们没有 $\models \rho \Rightarrow \vdash_T \rho$ ，即理论 T 没有完全性。我们更进一步来讨论，有关一致性的第二不完全性定理。设 I 是下面的公理形式(即数学归纳法)构成的全体 ($P \in \mathcal{P}$ 是任一谓词)

$$(P(0, \dots) \wedge \forall y(P(y, \dots) \rightarrow P(S(y), \dots))) \rightarrow \forall xP(x, \dots)$$

使用 $\{N_1, N_2, N_3, N_4, N_5, N_6\} \cup I$ 作为公理生成的理论，我们称为皮亚诺算术，并记作 PA，则有

$$\text{PA} = \text{Th}(\mathcal{N} \cup I)$$

皮亚诺算术 PA 和一阶算术理论 \mathcal{N} 都具备可靠性和 Σ_1 -完全性。为了说明形式不可证明，我们需要引入符号缩写

$$y \in \mathcal{V}, \square_T(y) := \exists x \mathbf{pf}_T(x, y); A \in \mathcal{L}_1, \square_T A := \square_T(\Gamma A \Box); \text{Con}(T) := \neg \square_T \perp$$

上面涉及十分多的内容，我们只能快速地讲一遍。首先，我们可以将所有的初始符号对应到一个自然数

$$g(\text{ }) = 3, g(\text{.}) = 5, g(\text{\neg}) = 7, g(\text{\rightarrow}) = 9, g(\text{\rightarrow\rightarrow}) = 11, g(\text{\forall}) = 13$$

$$g(x_k) = 7 + 8k, g(c_k) = 9 + 8k, g(F_k^n) = 11 + 2^n 3^k 8, g(P_k^n) = 13 + 2^n 3^k 8$$

再通过算术的唯一分解定理可以将所有公式 $A = u_1 \dots u_k \in \mathcal{L}_1$ 对应到唯一的一个自然数

$$g(u_1 \dots u_k) = 2^{g(u_1)} 3^{g(u_2)} \dots p_k^{g(u_k)}, p_k \text{ 是第 } k \text{ 个素数}$$

更进一步对于证明 A_1, \dots, A_k 也可以对应到唯一的一个自然数

$$g(A_1, \dots, A_k) = 2^{g(A_1)} 3^{g(A_2)} \dots p_k^{g(A_k)}, p_k \text{ 是第 } k \text{ 个素数}$$

公式和证明的区别在于 2 的幂指数一个为奇数一个为偶数，因此这个对应是单射且非满的，此时把这个数记为 $\#A = g(A) \in \mathbb{N}$ (A 是初始符号、公式、证明之一)，并把 $\#A$ 称为 A 的哥德尔数(方案不是唯一的)。 $\mathbf{pf}_T(x, y)$ 可以视为 \mathbb{N}^2 的一个关系，或 \mathcal{N} 中的一个谓词，用来表示“哥德尔数为 y 的公式的证明的哥德尔数为 x ”，引入迭代符号和相应的缩写

$$S^1(0) = S(0), S^n(0) = S^{n-1}(0), \Gamma A \Box = S^{\#A}(0)$$

因此公式 $\text{Con}(T)$ 表示“ T 是一致的”，于是我们就有了一致性的证明定理。

定理 2.6 (哥德尔第二不完全性定理)

设 T 是包含皮亚诺算术 PA 且是可计算公理化的理论。如果 T 是一致的，则 $\not\vdash_T \text{Con}(T)$ 。



以通俗的话来说就是 T 不能证明自身的一致性，至此数理逻辑的入门内容结束了。

第三章 模型论简介

在一阶逻辑中，如果我们只看前两个部分(1)(2)所形成的公式系统 $L_1 = (\mathcal{L}_1, \text{sig } \mathcal{L}_1)$ ，而不考虑公理和证明的问题，就把 L_1 称为一阶语言，它只是单纯的公式生成系统，其逻辑判定特性，如有效性、可满足性等，就需要考虑相应的 \mathcal{L}_1 -结构 $M = (M, I)$ ，此时我们反过来把 M 称为一阶语言 L_1 的一个模型（或结构），而这正是我们讨论的起点。有关模型论，没有固定的参考书，其零散地分布在各种数理逻辑的书籍中 [1, 2]，由于其重要性我特别地将其提取出来以供读者参考。

3.1 基础知识

对于一个模型，我们实际只需要指出论域和相应的 sig(常数、谓词和函数)即可，对应就是随便的事情，比如下面这些例子。

- 示例 3.1 (1) 群 (G, e, \cdot) : 论域 G 、常数 $\{e\}$ 、谓词 \emptyset 、函数 $\{\cdot\}$
(2) 初等数论 $(\mathbb{N}, 0, S, +, \times)$: 论域 \mathbb{N} 、常数 $\{0\}$ 、谓词 \emptyset 、函数 $\{S, +, \times\}$
(3) 线序 $(X, <)$: 论域 X 、常数 \emptyset 、谓词 $\{<\}$ 、函数 \emptyset
(4) 有序域 $(F, 0, 1, +, \times, <)$: 论域 F 、常数 $\{0, 1\}$ 、谓词 $\{<\}$ 、函数 $\{+, \times\}$

它们都是特定一阶语言的模型，这里的特定主要指控制 $\text{sig } \mathcal{L}_1$ 中的元素个数以满足解释 I 确实能形成一个映射。我们继续探讨模型之间的关系

定义 3.1

设 $M = (M, I), N = (N, J)$ 为 L_1 的模型， $h : M \rightarrow N$ 为映射

- (1) 如果有
(i) 对任意的 $c \in C$, 有 $h(I(c)) = J(c)$
(ii) 对任意的 $P_i^n(t_1, \dots, t_n) \in \mathcal{P}$, 有 $I(P_i^n)(t_1, \dots, t_n) \Leftrightarrow J(P_i^n)(h(t_1), \dots, h(t_n))$
(iii) 对任意的 $F_i^n(t_1, \dots, t_n) \in \mathcal{F}$, 有 $h(I(F_i^n)(t_1, \dots, t_n)) = J(F_i^n)(h(t_1), \dots, h(t_n))$
则称 h 是同态映射
(2) 如果 h 是同态映射且是单射，则称 h 是嵌入映射
(3) 如果 h 是同态映射且是双射，则称 h 是同构映射，并记为 $h : M \cong N$
(4) 记 $\text{Th}(M) = \{A \in \text{Se}(\mathcal{L}_1) \mid \models_M A\}$ 。如果 $\text{Th}(M) = \text{Th}(N)$ ，则称 M 与 N 初等等价，并记为 $M \equiv N$

对于上面的两个定义有着相应简单的定理

定理 3.1

- (1) 如果 $M \cong N$, 则有 $M \equiv N$
(2) M 是有限的。如果 $M \equiv N$, 则有 $M \cong N$

下面是一些等价模型的例子

$$(\mathbb{N} - \{0\}, <) \equiv (\mathbb{N}, <), (\mathbb{Q}, <) \equiv (\mathbb{R}, <), (\mathbb{Q}, <, +, \times) \not\equiv (\mathbb{R}, <, +, \times)$$

为了理解线序模型都是等价的，我们需要引入一阶理论的模型的概念，即公式集 Γ 的模型

定义 3.2

- 设公式 $A \in \mathcal{L}_1$ 和公式集 $\Gamma \mathcal{L}_1$
(1) 如果 $\models_{(M, v)} A$, 则称 (M, v) 是 A 的模型

(2) 如果对任意 $A \in \Gamma$ 均有 $\models_{(\mathcal{M}, v)} A$, 则称 (\mathcal{M}, v) 是 Γ 的模型



理论的模型涉及了定理的概念, 因此必需在原来模型 \mathcal{M} 的基础上, 带上用于判定的赋值 v , 才能称为理论的模型。实际上, 我们可以证明“合同引理”, 其表明了赋值之间可以相互对应, 换言之考虑理论的模型时, 我们无需再考虑赋值, 因为其在合同意义下是唯一确定的, 这也是我们之前所说的“随便选对应”的意思。在理论 $(L, =)$ 的基础上, 我们添加谓词 $<$ 和下面的限制公理

$$(O_1) \forall x \neg(x < x) \quad (O_2) \forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z) \quad (O_3) \forall x \forall y (x < y \vee y < x \vee x = y)$$

$$(O_4) \forall x \forall y (x < y \rightarrow \exists z (x < z \wedge z < y)) \quad (O_5) \forall x \exists y \exists z (y < x \wedge x < z)$$

我们记理论 $DLO = (L, =, <)$, 并把它称为无端点稠密线序理论。容易验证, $(\mathbb{Q}, <), (\mathbb{R}, <)$ 都是它的模型, 实际上我们有一般的等价性定理。

定理 3.2

设 \mathcal{M} 和 \mathcal{N} 都是可数无穷模型。如果它们都是 DLO 的模型, 则它们是同构的。



我们举这个例子只是想读者对理论和模型有一点感性的认识, \mathbb{Q} 和 \mathbb{R} 的序理论是完全一样的, 但它们的域理论却是不同的, 一阶理论只是用来提供逻辑支撑, 而实际的数学实现则需要在各种模型中完成。比如, 我们可以证明理论 DLO 具有完全性, 因此 \mathbb{Q} 和 \mathbb{R} 的序理论的所有定理都可以被证明, 但大家同样意识到了 $\mathbb{N} \subset \mathbb{Q} \subset \mathbb{R}$, 而 $\mathbf{N} = (\mathbb{N}, 0, +, \times, S)$ 作为皮亚诺算术 PA 的模型却是不完全的, 这是因为 \mathbf{N} 显然不能作为 $(\mathbb{Q}, <)$ 的子模型, DLO 也不是 PA 的子理论, 所以在研究模型和理论时单纯地看集合的“大小”是没有任何意义的。

可表示性

模型的定义说明了, 理论的常数、谓词和函数可以通过解释在模型中表示出来, 因此我们自然需要讨论一个模型如何反作用于理论的概念, 即模型中的常数、关系和映射能否在理论中表示出来。设 T 是包含一阶算术理论 $\mathcal{N} = (L, =, 0, S, +, \times)$ 的理论, $(\mathbb{N}, 0, +, \times, S)$ 是皮亚诺算术 PA 的模型, 因此自然可以视为 \mathcal{N} 的模型(不一定可视为 T 的模型), 在 \mathcal{N} 中我们记 $\bar{n} = S^n(0)$, 在 \mathbb{N} 中我们记 $n = S^n(0)$ 。对于公式 $A \in \mathcal{L}_1$, 如果它有 $n = |\text{Fr}(A)|$ 个自由变量, 我们就记其为 $A(x_1, \dots, x_n)$, 并且相应的自由代入表示为 $A(\dots, t_k, \dots) := A(x_1, \dots, x_n)(x_k; t_k)$, 此时我们定义

定义 3.3

(1) 对于关系 $R \subset \mathbb{N}^k$, 如果存在公式 $A(x_1, \dots, x_k) \in \mathcal{L}_1$ 使得对任意的 $n_1, \dots, n_k \in \mathbb{N}$ 均有

$$(n_1, \dots, n_k) \in R \Rightarrow \vdash_T A(\bar{n_1}, \dots, \bar{n_k})$$

$$(n_1, \dots, n_k) \notin R \Rightarrow \vdash_T \neg A(\bar{n_1}, \dots, \bar{n_k})$$

则称 R 在 T 中是可表示的

(2) 对于函数 $f : \mathbb{N}^k \rightarrow \mathbb{N}$, 如果存在公式 $A(x_1, \dots, x_k, y) \in \mathcal{L}_1$ 使得对任意的 $n_1, \dots, n_k \in \mathbb{N}$ 均有

$$\vdash_T \forall y (A(\bar{n_1}, \dots, \bar{n_k}, y) \leftrightarrow y = \overline{f(n_1, \dots, n_k)})$$

则称 f 在 T 中是可表示的



有了上面的两个基础定义, 我们自然需要考虑什么样的关系 $R \subset \mathbb{N}^k$ 和函数 $f : \mathbb{N}^k \rightarrow \mathbb{N}$ 是可以被表示的, 为了响应需求, 对于理论 T , 我们在要求其包含 \mathcal{N} 的基础上, 进一步要求是可计算公理化的、一致的。

定义 3.4 (递归函数和递归关系)

(1) 基本函数指下面的三类函数(零值函数、后继函数、投影函数)

$$Z(x) = 0, S(x) = x + 1, U_i^n(x_1, \dots, x_n) = x_i$$

(2) 基本规则指下面的三种函数生成规则

(I. 复合) 对于函数 $g(x_1, \dots, x_j)$ 和函数 $h_1(x_1, \dots, x_k), \dots, h_j(x_1, \dots, x_k)$ 有

$$f(x_1, \dots, x_k) := g(h_1(x_1, \dots, x_k), \dots, h_j(x_1, \dots, x_k))$$

(II. 递归) 对于函数 $g(x_1, \dots, x_k)$ 和函数 $h(x_1, \dots, x_k, x_{k+1}, x_{k+2})$ 有

$$f(x_1, \dots, x_k, 0) := g(x_1, \dots, x_k)$$

$$f(x_1, \dots, x_k, y + 1) := h(x_1, \dots, x_k, y, f(x_1, \dots, x_k, y))$$

(III. 最小数) 函数 $g(x_1, \dots, x_k, x_{k+1})$ 满足 $\forall x_i, |\{y \mid g(x_1, \dots, x_k, y) = 0\}| > 0$ 有

$$f(x_1, \dots, x_k) = \min\{y \mid g(x_1, \dots, x_k, y) = 0\}$$

(3) 我们把基本函数通过有限此规则 I, II 得到的函数称为原始递归函数

(4) 我们把基本函数通过有限此规则 I, II, III 得到的函数称为递归函数

(5) 对于关系 $R \subset \mathbb{N}^k$, 我们定义它的特征函数为

$$\chi_R(x_1, \dots, x_k) = \begin{cases} 0 & (x_1, \dots, x_k) \notin R \\ 1 & (x_1, \dots, x_k) \in R \end{cases}$$

(6) 如果关系 R 的特征函数是递归的, 我们就称 R 是递归的

有了上面的两个基础, 我们就可以得到可表示的条件了。

定理 3.3 (可表示性)

(1) 关系 R 在 T 中是可表示的当且仅当它的特征函数 χ_R 在 T 中是可表示的

(2) 如果 f 是递归的, 则函数 f 在 T 中是可表示的



在这里, (1) 表明了关系的可表示性可以转化为函数的可表示性, (2) 表明了一类重要的可表示函数, 递归函数, 但读者需要注意反过来是不一定, 这其实很好理解, 如果理论 T 扩大, 其依旧满足前提, 但可表示的函数可能是会变多的, 至于使得所有可表示函数是递归函数的理论应该是什么? 只能说我不知道。但如过考虑稍微小点的关系, 确实有一个理论可以使得“所有可表示关系都是递归的”, 对于语句集 $\Gamma \subset \text{Se}(\mathcal{L}_1)$, 我们引入

$$\text{Cn}(\Gamma) := \{A \in \text{Se}(\mathcal{L}_1) \mid \Gamma \vDash A\}$$

由可靠性定理可知 $\text{Th}(\Gamma) \subset \text{Cn}(\Gamma)$, 我们引入 $A_E = (L, =, 0, +, \times, <, E)$, 则 $\text{Cn}(A_E)$ 就是我们想要的理论, 这里的 E 为二元函数, 是我们通常所见的指数运算 \exp , 有下面的两条公理

$$(E_1) E(x, 0) = S(0) \quad (E_2) E(x, S(y)) = E(x, y) \times y$$

可定义性

在模型中, 或者在一般的数学研究中, 我们一般不可能只停留在 \mathbb{N}, \mathbb{R} 等模型的基础符号上面, 都会遇到“定义”这样一件事, 以扩充符号体系。有关定义, 我在以前的文章曾说过一个例子, 就是为什么不能用集合来定义范畴, 定义在形式系统中的表现, 可能只是公式的简化, 又或者是增设前提进行演绎, 我们主要考虑后一种情形, 因此对于一个模型 $M = (M, I)$ 而言, 定义实际就是取出一个满足某些条件的子集 $X \subset M^n$ 。我们举一些简单的例子, 在初等数论 $(\mathbb{N}, 0, +, \times, S)$ 中, 我们定义整除为

$$a \mid b := \exists c (ac = b)$$

其相当于取出一个关系 $R \subset \mathbb{N}^2$ 有

$$(a, b) \in R \Leftrightarrow a \mid b$$

类似的，我们定义一个数论函数 $f : \mathbb{N} \rightarrow \mathbb{N}$ ，也相当于取出一个关系 $R \subset \mathbb{N}^2$ 有

$$(a, b) \in R \Leftrightarrow b = f(a)$$

接着，我们来将可定义的概念进行严格化表述。不过我们需要引入一个符号，设 $\mathcal{M} = (M, I)$ 是 \mathcal{L}_1 -结构，对于公式 $A(x_1, \dots, x_k) \in \mathcal{L}_1$ ，对任意 $a_1, \dots, a_n \in M$ ，我们使用下面的符号

$$\models_{\mathcal{M}} A[a_1, \dots, a_n]$$

表示“如果 \mathcal{M} -赋值 $v : \mathcal{V} \rightarrow M$ 满足 $v(x_i) = a_i (1 \leq i \leq n)$ ，就有 $\models_{(\mathcal{M}, v)} A$ ”。通过这个符号我们就无需反过来考虑常数 $a_i \in M$ 是否在理论 T 中可表达的问题了，此时有

定义 3.5

设 $\mathcal{M} = (M, I)$ 是 \mathcal{L}_1 -结构， $X \subset M$ ， $Y \subset M^n$

(1) 如果存在常数 $b_1, \dots, b_m \in X$ 和公式 $A(x_1, \dots, x_n, x_{n+1}, \dots, x_{n+m})$ 满足

$$(a_1, \dots, a_n) \in Y \Leftrightarrow \models_{\mathcal{M}} A[a_1, \dots, a_n, b_1, \dots, b_m]$$

就称 Y 可以通过 X 中的参数在 \mathcal{M} 中可定义，并记 $Y \in \text{Def}_n(\mathcal{M}, X)$

(2) 特别地，当 $X = \emptyset$ 且 $Y \in \text{Def}_n(\mathcal{M}, \emptyset)$ 时，我们称 Y 在 \mathcal{M} 是可定义的

我们可以考虑一些简单的例子，在初等数论 $\mathbf{N} = (\mathbb{N}, 0, +, \times, S)$ 中

示例 3.2 (1) 数字一： $1 = S(0)$ ，可以视为 $\{1\} \subset \mathbb{N}^1$ ，其在 \mathbf{N} 中是可定义的

(2) 有限集： $\{n_1, \dots, n_k\} \subset \mathbb{N}$ ，在 \mathbf{N} 中是可定义的

(3) 奇数集： $\{2n+1 \mid n \in \mathbb{N}\} \subset \mathbb{N}$ ，在 \mathbf{N} 中是可定义的

(4) 小于： $a < b$ ，可以视为 $\{(a, b) \mid a < b\} \subset \mathbb{N}^2$ ，在 \mathbf{N} 中是可定义的

想要在通常的例子下找不可定义的概念是挺困难的，仔细想想，我们都把它定义出来了，再说它不可定义也太矛盾了，不过不可定义的例子也是有的。

定理 3.4 (算术真理不可定义的塔斯基定理)

对于初等数论 $\mathbf{N} = (\mathbb{N}, 0, +, \times, S)$ 。 $\# \text{Th}(\mathbf{N}) := \{\#A \mid \models_{\mathbf{N}} A\} \subset \mathbb{N}$ 在 \mathbf{N} 中是不可定义的。

我们稍微讲讲塔斯基定理的一般形式，我们将条件上升到一般的理论 T ，同样要求其包含一阶算术理论 \mathcal{N} 且是可计算公理化的、一致的。对任意公式 $A(x) \in \mathcal{L}_1$ ，我们定义公式集

$$\text{TB}(A) = \{A(x; \lceil B \rceil) \leftrightarrow B \mid B \in \text{Se}(\mathcal{L}_1)\}$$

这里的 $A(x; \lceil B \rceil)$ 是通过哥德尔数 $\lceil B \rceil = S^{\#B}(B) \in \mathcal{I}$ 和代入两个步骤形成的公式

定理 3.5 (塔斯基不可定义性定理)

不存在公式 $A(x) \in \mathcal{L}_1$ 使得 $\text{TB}(A) \subset T$ 。

这就是句法形式的塔斯基定理，因为可定义性本质就是去找一个可以成为定理的公式，而定理的概念会随理论而改变，此处的 $\text{TB}(A)$ 其实就是表示，如果 A 可以用来定义，那么需要的公式有哪些，而塔斯基定理则说

明了，这些待选公式并不能成为定理，因此塔斯基定理的通俗含义是“满足一定性质的理论 T 中不能定义什么是真”，也就是我们不能定义什么是真理，因此塔斯基定理也被称为“真理的不可定义性”。

模型存在性

仔细想想，我们说了这么多模型与理论之间的关系，却没有意识到这样一个问题，一个理论 T 是否存在模型？如果存在，那么存在怎样的模型？实际上，对于前者我们已经在一阶理论的紧致性中回答了，理论 T 存在模型实际就等价于 T 是可满足的，于是就有下面的紧致性定理

理论 T 有模型当且仅当 T 的每个有限子集有模型

接着我们通过证据理论和一阶逻辑的完全性就可以得到下面的模型存在性定理

定理 3.6 (广义完全性定理)

理论 T 是一致的当且仅当 T 是有模型的。



模型的大小是一个值得关注的问题，我们先回顾一些简单概念：(1) 如果理论 T 满足对任意公式 $A \in \mathcal{L}_1$ 使得 $\vdash_T A, \vdash_T \neg A$ 有且只有一个成立，就称 T 是完全的；(2) 如果理论 T 不存在公式 $A \in \mathcal{L}_1$ 使得 $\vdash_T A, \vdash_T \neg A$ 同时成立，就称 T 是一致的；(3) 对于理论 T ，如果所有满足 $\models_M T$ 的 \mathcal{L}_1 -结构都是互相同构的（即 $\models_M T, \models_N T \Rightarrow M \cong N$ ），就称 T 是范畴的。其中一致性和完全性的关系就是哥德尔第一不完全性定理，而哥德尔第二不完全性定理则说明了一致性不好考察，所以我们来考察完全性和范畴性的关系。首先我们需要将完全性转化为模型的条件，即下面的定理

定理 3.7

理论 T 是完全的当且仅当，所有满足 $\models_M T$ 的 \mathcal{L}_1 -结构都是互相初等等价的（即 $\models_M T, \models_N T \Rightarrow M \equiv N$ ）。



由于同构可以推出初等等价，因此我们可以得到“范畴的理论一定是完全的”，即范畴性强于完全性。此时为了将定理反过来，就只需用到模型的大小了，即有

对于有穷理论，其完全性和范畴性是等价的

此处理论的大小指公理集的大小，由于任意理论均有 $T = \text{Th}(T)$ ，所以我们无需担心理论不可被公理化。无穷时就没那么简单了，我们有下面的核心事实

定理 3.8

如果模型 M 是无穷的，那么存在 \mathcal{L}_1 -结构 N 使得， $M \equiv N$ 且 $M \not\cong N$ 。



这其实预示了非标准模型的存在，我们后面会详细讨论。接着，我们聚焦于有限基数以外的模型，即从可数基数 \aleph_0 开始的无穷基数，只需下面的两个定理即可

定理 3.9

(1)[无穷模型存在] 如果理论 T 有任意大小的有限模型，那么 T 也有无穷大的模型

(2)[Löwenheim-Skolem-Tarski] 如果理论 T 有无穷大的模型，那么对任意基数 $\kappa \geq |\mathcal{L}_1|$ ， T 有基数为 κ 的模型



这里 $|\mathcal{L}_1|$ 表示所有公式的基数，在一阶语言中它必定是个无穷基数，实际上如果 $\kappa = |\text{sig}(\mathcal{L}_1)|$ 是无穷基数，那么有

$$\kappa \leq |\mathcal{L}_1| \leq |\text{sig}(\mathcal{L}_1)^{\aleph_0}| = \kappa^{\aleph_0} = \kappa$$

从而有基数相等 $|\mathcal{L}_1| = |\text{sig}(\mathcal{L}_1)|$, 所以我们认为一阶语言的大小就是 $\text{sig}(\mathcal{L}_1)$ 的大小。为了适应无穷基数 κ 的丰富性, 我们引入 κ 范畴性: 对于理论 T , 如果所有满足 $\models_M T$ 的基数为 κ 的 \mathcal{L}_1 -结构都是互相同构的 (即 $\models_M T, \models_N T, |\mathcal{M}| = |\mathcal{N}| = \kappa \Rightarrow \mathcal{M} \cong \mathcal{N}$), 就称 T 是 κ 范畴的。此时我们有一个核心定理

定理 3.10 (Morley, 范畴性定理)

如果存在某个不可数基数 κ 使得理论 T 是 κ 范畴的, 那么对任意不可数基数 κ 理论 T 是 κ 范畴的。



如果 $\kappa \geq \aleph_1$ (即不可数基数) 且理论 T 是 κ 范畴, 我也称 T 是 ω -稳定的, 在 [2] 中, 用了六、七两个章节进行讨论, 我们的简介篇就不探究了, 我们需要做的就是理解这种简单的事情。

初等等价

上一部分我们专注于模型的同构问题引出的理论范畴性, 这一节我们集中讨论模型的初等等价问题。对于两个 \mathcal{L}_1 -结构 $\mathcal{M} = (M, I), \mathcal{N} = (N, J)$, 如果映射 $h: M \rightarrow N$ 满足对任意的公式 $A(x_1, \dots, x_n) \in \mathcal{L}_1$ 和 $a_1, \dots, a_n \in M$ 均有

$$\models_M A[a_1, \dots, a_n] \Leftrightarrow \models_N A[h(a_1), \dots, h(a_n)]$$

就称 h 是初等嵌入映射, 并记 $h: \mathcal{M} \hookrightarrow \mathcal{N}$ 。如果包含映射 $i: M \rightarrow N$ 是初等嵌入映射, 就称 \mathcal{M} 是 \mathcal{N} 的初等子模型, 并记 $\mathcal{M} \prec \mathcal{N}$ 。显然初等子模型是初等嵌入映射, 而初等嵌入映射与初等等价的关系如下

定理 3.11

如果存在初等嵌入映射 $h: \mathcal{M} \hookrightarrow \mathcal{N}$, 则有 $\mathcal{M} \equiv \mathcal{N}$ 。



有关初等子模型有下面的判定定理

定理 3.12 (Tarski–Vaught Test)

设 $\mathcal{M} \subset \mathcal{N}$, 即 $\mathcal{M} = (M, I)$ 是 $\mathcal{N} = (N, J)$ 的子模型, 即包含映射 $i: M \rightarrow N$ 是嵌入映射。此时, $\mathcal{M} \prec \mathcal{N}$ 当且仅当, 对任意公式 $\exists x A(x, x_1, \dots, x_n)$ 和任意的 $a_1, \dots, a_n \in M$ 有

$$\models_N \exists x A[x, a_1, \dots, a_n] \Rightarrow \exists a \in M \models_N A[a, a_1, \dots, a_n]$$



虽然在数理逻辑中用逻辑符号总觉得怪怪的, 但写“当且仅当”“如果..., 则...”“存在”“对任意”之类也挺麻烦的, 不如混用算了, 我们只需搞清楚, “公式 XXX”、“ \models 公式”等等里的符号是数理逻辑, 其它情况就是我们日常逻辑推理的使用即可。对于初等等价还有很多判定定理, 比如 Fraïssé 定理 (参考[这里](#)的 theorem 2.2.4) 和 Ehrenfeucht 定理 (参考 [2] 的 Proposition 2.4.5) 等, 但我个人觉得这些定理换汤不换药, 还要引入更多的记号只会徒增烦恼, 所以就不详细讨论了。初等等价最重要的作用就是与完全性进行联系, 就是我们之前所说的那个定理

理论是完全的当且仅当它的任意模型是互相初等等价的

因此研究完全性理论也相当于研究模型的初等等价问题, 值得注意的是理论有模型的条件是具备一致性, 因此通常我们在讨论模型论的时候可以选择性的忽视一致性的条件。

定理 3.13 (Vaught 判别法)

设理论 T 没有有限模型。如果对某个无穷基数 $\kappa \geq |\mathcal{L}_1|$ 理论 T 是 κ -范畴的, 则 T 是完全的。



通过这个性质我们很容易证明无端点稠密线序理论 DLO、特征为 0 的代数闭域理论 $(L, =, 0, 1, +, \times)$ 、无原子布尔代数理论 $(L, \wedge, \vee, \neg, 1, 0)$ 都是完全的。完全性是难得可贵的, 因此我们还有一个稍弱的条件: 对于理论 T , 如果其任意满足 $\mathcal{M} \subset \mathcal{N}$ 的模型均有 $\mathcal{M} \prec \mathcal{N}$, 则称 T 是模型完全的。其与完全性的关系如下

定理 3.14 (Robinson)

设 T 是模型完全的理论

- (1) 如果 T 的任意两个模型都能同构嵌入到 T 的一个模型中，则 T 是完全的
- (2) 如果 T 有一个模型能同构嵌入到 T 的任意模型中，则 T 是完全的



于是为了得到完全性，我们可以考虑模型完全性，而为了得到模型完全性，我们还得考察一点其它的东西

定理 3.15 (Lindstrom)

设 \mathcal{L}_1 是可数的， T 是 \mathcal{L}_1 -理论，如果

- (1) T 的每个模型都是无限的
- (2) T 的任意模型升链的并是 T 的模型
- (3) 存在无穷基数 κ 使得 T 是 κ 范畴的

则 T 是模型完全的。



通过上面的定理，我们可以知道无端点稠密线序理论 DLO、代数闭域理论、实闭有序域理论是模型完全的。模型完全理论的一个重要运用就是用来证明希尔伯特零点定理和希尔伯特第 17 问，在此之前我们来介绍一下域理论和闭域理论，我们的基础前提依旧是一阶带等号理论 $(L, =)$ ，我们特定化常数 0, 1 和谓词 $+, \times$ ，并使用如下公理进行限制

$$(F_1) \forall x_1 \forall x_2 (x_1 + x_2 = x_2 + x_1) \quad (F_2) \forall x_1 \forall x_2 \forall x_3 ((x_1 + x_2) + x_3 = x_1 + (x_2 + x_3))$$

$$(F_3) \forall x_1 (x_1 + 0 = x_1) \quad (F_4) \forall x_1 \exists x_2 (x_1 + x_2 = 0)$$

$$(F_5) \forall x_1 \forall x_2 (x_1 \times x_2 = x_2 \times x_1) \quad (F_6) \forall x_1 \forall x_2 \forall x_3 ((x_1 \times x_2) \times x_3 = x_1 \times (x_2 \times x_3))$$

$$(F_7) \forall x_1 (x_1 \times 1 = x_1) \quad (F_8) \forall x_1 ((x_1 \neq 0) \rightarrow (\exists x_2 (x_1 \times x_2 = 1)))$$

$$(F_9) \forall x_1 (x_1 \times 0 = 0) \quad (F_{10}) \forall x_1 \forall x_2 \forall x_3 (x_1 \times (x_2 + x_3) = x_1 \times x_2 + x_1 \times x_3) \quad (F_{11}) 1 \neq 0$$

即可得到域理论 $T_F = (L, =, 0, 1, +, \times)$ ，所以我们实际就是把域的定义给搬了过来，只不过运用等价稍微改变了些条件，但它们却是货真价实的一阶逻辑公式。在理论 T_F 的基础上，我们添加下面的公理

$$(P_n) \forall x_0 \forall x_1 \dots \forall x_n (x_n \neq 0 \rightarrow (\exists y (x_n \times y^n + x_{n-1} \times y^{n-1} + \dots + x_1 \times y_1 + x_0 = 0)))$$

即可得到代数闭域理论 $ACF = Th(T_F \cup \{P_n\})$ ，此处 $y^n = \underbrace{y \times \dots \times y}_n$ 是缩写并要求 $n \geq 1$ 。在理论 ACF 的基础上，我们添加下面的公理

$$(C_p) (1 + 1 \neq 0) \wedge \dots \wedge (\underbrace{1 + \dots + 1}_{p-1} \neq 0) \wedge (\underbrace{1 + \dots + 1}_p = 0)$$

即可得到特征为 p 的代数闭域理论 $ACF_p = Th(ACF \cup \{C_p\})$ 。如果在理论 ACF 的基础上，我们添加下面的公理

$$(C_0) \underbrace{1 + \dots + 1}_n \neq 0$$

则可得到特征为 0 的代数闭域理论 $ACF_0 = Th(ACF \cup \{C_0\})$ 。在理论 T_F 的基础上，引入谓词 $<$ 并添加下面的公理

$$(O_1) \neg(x_1 < x_1) \quad (O_2) x_1 < x_2 \wedge x_2 < x_3 \rightarrow x_1 < x_3 \quad (O_3) x_1 < x_2 \vee x_2 < x_1 \vee x_1 = x_2$$

$$(O_6)x_1 < x_2 \rightarrow (x_1 + x_3 < x_2 + x_3) \quad (O_7)(x_1 < x_2 \wedge 0 < x_3) \rightarrow (x_1 \times x_3 < x_2 \times x_3)$$

$$(O_8)(x_1 < x_2 \wedge x_3 < 0) \rightarrow (x_2 \times x_3 < x_1 \times x_3) \quad (O_9)0 < 1$$

为了便于阅读，此处我们使用了全称量词固化¹，此时我们把 $T_{OF} = (T_F, <)$ 称为有序域理论。对于实闭域大家可能不熟悉，我就稍微讲个概念，如果域 F 满足对任意的 $x_1, \dots, x_n \in F$ 均有

$$\sum_{i=1}^n x_i^2 \neq -1$$

我们就称 F 为实域；如果实域 F 的任意非平凡扩域都不是实域，就称 F 为实闭域。因此考虑实闭域理论，应该在理论 T_F 或 T_{OF} 的基础上，而相应添加的公理为

$$(C_0) \underbrace{1 + \dots + 1}_n \neq 0$$

$$(P_{2n+1}) \forall x_0 \forall x_1 \dots \forall x_{2n+1} (x_{2n+1} \neq 0 \rightarrow (\exists y (x_{2n+1} \times y^{2n+1} + x_{2n} \times y^{2n} + \dots + x_1 \times y_1 + x_0 = 0)))$$

$$(S_1) \forall x_1 (\exists x_2 (x_2 \times x_2 = x_1) \vee \exists x_2 (x_2 \times x_2 = -x_1))$$

$$(S_2) \forall x_1 \dots \forall x_n (x_1^2 + \dots + x_n^2 = 0 \rightarrow (x_1 = 0 \wedge \dots \wedge x_n = 0))$$

我们把 $\text{RCF} = \text{Th}(T_F \cup \{C_0, P_{2n+1}, S_1, S_2\})$ 和 $\text{RCF}_o = \text{Th}(T_{OF} \cup \{C_0, P_{2n+1}, S_1, S_2\})$ 分别称为实闭域理论和实闭有序域理论。实际上对于实闭有序域理论，由于序公理的存在，我们可以进行适当的简化，写入下面的公理

$$(S_3) \forall x_1 (0 < x_1 \rightarrow (\exists x_2 (x_2 \times x_2 = x_1)))$$

于是有 $\text{RCF}_o = \text{Th}(T_{OF} \cup \{P_{2n+1}, S_3\})$ 。实闭有序域理论的两个常见模型为

$$\mathbb{R}, \mathbb{R} \cap \overline{\mathbb{Q}}$$

有关实闭域理论的完全性我们之前已经讨论过了，我们来讲一个比较新鲜的性质。对于理论 T ，如果对任意的公式 $A(x_1, \dots, x_n) \in \mathcal{L}_1$ 都存在一个不包含量词（即 \forall ）的公式 $B(x_1, \dots, x_n) \in \mathcal{L}_1$ 使得

$$\vdash_T \forall x_1 \dots \forall x_n (A(x_1, \dots, x_n) \leftrightarrow B(x_1, \dots, x_n))$$

就称 T 是适合消去量词的。读者可能会好奇，为什么我就是不愿意像一般书籍那样把模型或理论写在 \vdash 的前面

$$T \vdash A, (\mathcal{M}, v) \vDash A, \mathcal{M} \vDash A$$

就算是理论 T 这种本身也等同于带前提证明的，我也没这么写，理由十分简单，希望读者将理论 T 和一阶逻辑 L 视为只是初始公式不同的形式系统。此时我们有

定理 3.16 (Tarski)

理论 RCF_o 适合消去量词，理论 RCF 不适合消去量词。



有了上面的基础知识就能来证明下面的两个定理了。

¹读者需要注意， $\forall x_1 \neg(x_1 < x_1)$ 和 $\neg(x_1 < x_1)$ 是完全不同的公式，而固化也不是什么合格的逻辑术语，只是因为我们理论的公理前总加 $\forall x_i$ 比较麻烦而进行的省略罢了

定理 3.17 (希尔伯特零点定理)

设 K 为代数闭域，则对于多项式 $p_1, \dots, p_k \in P = K[x_1, \dots, x_n]$ ，下面三个命题等价

- (1) p_1, \dots, p_k 在 K 中无公共解
- (2) 存在 $g_1, \dots, g_k \in P$ 使得 $p_1g_1 + \dots + p_kg_k = 1$
- (3) p_1, \dots, p_k 在 K 每个扩域中无公共解



证明 (2) \Rightarrow (1)，由题意可知 $(p_1, \dots, p_k) = 1$ ，故 p_1, \dots, p_k 在 K 中无公共解

(1) \Rightarrow (3)，使用反证法，我们将 K 看成代数闭域理论的一个模型，并将其有解扩域的代数闭包记为 K_1 ，同样也可以视为代数闭域理论的一个模型。我们选出下面的一个公式

$$A := \exists y_1 \dots \exists y_n (p_1(y_1, \dots, y_n) = 0 \wedge \dots \wedge p_k(y_1, \dots, y_n) = 0)$$

此时我们有

$$\models_{K_1} A$$

由于代数闭域理论是模型完全的且它们的论域满足 $K \subset K_1$ ，因此有 $K \prec K_1$ ，从而有

$$\models_K A$$

这与题设矛盾

(3) \Rightarrow (2)，使用反证法，我们记理想

$$I = \{f_1p_1 + \dots + f_kp_k \mid f_1, \dots, f_k \in P\}$$

则有 $1 \notin I$ ，从而存在 P 的一个极大理想 M 使得 $I \subset M$ 。此时，剩余域 $K_1 = P/M$ 满足 $K \subset K_1$ ，取出元素 $a_i = x_i + M \in K_1$ ，则有

$$p_i(a_1, \dots, a_n) = p_i(x_1, \dots, x_n) + M = 0 + M$$

从而 a_1, \dots, a_n 是 p_1, \dots, p_k 在 K_1 中的公共解，这与题设矛盾

定理 3.18 (希尔伯特第 17 问)

设 R 为实闭有序域，则对于有理分式 $q \in Q = R(x_1, \dots, x_n)$ ，下面三个命题等价

- (1) q 在 R 上是非负的，即 $q(x_1, \dots, x_n) < 0$ 恒不成立
- (2) 存在有限个 $q_1, \dots, q_k \in Q$ 使得 $q = q_1^2 + \dots + q_k^2$
- (3) q 在 R 的每个有序扩域上是非负的



证明 (2) \Rightarrow (1)，显然

(1) \Rightarrow (3)，使用反证法，我们将 R 看成实闭有序域理论的一个模型，并将其满足 $q < 0$ 有解的有序扩域的代数闭包记为 K_1 ，同样也可以视为实闭有序域理论的一个模型。我们记 $q(x_1, \dots, x_n) = \frac{f(x_1, \dots, x_n)}{g(x_1, \dots, x_n)}$, $f, g \in R[x_1, \dots, x_n]$ ，并选出下面的一个公式

$$A := \exists x_1 \dots \exists x_n (f(x_1, \dots, x_n) \times g(x_1, \dots, x_n) < 0)$$

此时我们有

$$\models_{K_1} A$$

由于实闭有序域理论是模型完全的且它们的论域满足 $K \subset K_1$ ，因此有 $K \prec K_1$ ，从而有

$$\models_K A$$

从而 $q < 0$ 在 R 中有解，这与题设矛盾

(3) \Rightarrow (2)，使用反证法，我们的目的是在 Q 上定义序结构，从而使得 Q 是 R 的有序扩域。定义集合

$$P_0 = \{\sigma_1 + \sigma_2(-q) \mid \sigma_1, \sigma_2 \text{ 是 } Q \text{ 中的平方和}\}$$

则 $P_0 \subset Q$ 满足 (p1) $\forall a \in Q, a^2 \in P$ (p2) $\forall a_1, a_2 \in P, a_1 + a_2 \in P, a_1a_2 \in P$ (p3) $-q \in P$ (p4) $-1 \notin P$ 。由于上界 Q 的存

在，我们可以令 P_0 保持这四个性质扩充为极大的 P ，则 P 满足一个新的性质 (p5) 对任意 $a \in Q$, $a \in P, -a \in P$ 有且只有一个成立。我们定义 Q 上的序关系为

$$u, v \in Q, u < v \Leftrightarrow u \neq v, v - u \in P$$

此时， Q 是 R 的有序扩域且 $q < 0$ 有解，这与题设矛盾

力迫法

关于模型论还有许多离散的课题，比如各种完全性理论（原子模型、素模型、饱和模型、可数模型等）、初等模型链（初等链定理、省略型定理、内插定理等）、模型自同构（Skolem 函数、不可辩元等）等，以我们简介的体量来说完全没有介绍的必要，但超积运算 \prod 还是值得一说。

定理 3.19 (Vaught 定理)

对任意理论 T ，有 $I(T, \aleph_0) \neq 2$



此处 $I(T, \kappa)$ 表示“理论 T 的互不同构的基数为 \aleph_0 的模型的个数”，因此 Vaught 定理（参考地址）表示“不存在恰有两个不同构可数模型的完备理论”。

定理 3.20 (Craig 内插定理)

对任意公式 $A, B \in \mathcal{L}_1$ ，如果有 $A \models B$ ，则存在公式 $C \in \mathcal{L}_1$ 使得

- (1) $A \models C$ 且 $C \models B$
- (2) C 中出现的常数、函数和谓词在 A, B 中均出现



更强的内插定理比如 Lyndon（参考 [1] 的 theorem 2.2.24），只不过是增强了条件 (2)，所以没啥好说的。

定理 3.21 (Ramsey 定理)

- (1) 如果 $k, n < \aleph_0$ ，则 $\aleph_0 \rightarrow (\aleph_0)_k^n$
- (2) 如果 $k, m, n < \aleph_0$ ，则存在 $l < \aleph_0$ 使得 $l \rightarrow (m)_k^n$



我们来解释一下上面的符号，对于集合 X 和基数 κ 我们记符号 $[X]^\kappa = \{Y \in 2^X \mid |Y| = \kappa\}$ ，即所有基数为 κ 的 X 的子集构成的集合。对于基数 κ, λ 和 $\eta, \mu < \aleph_0$ 符号 $\kappa \rightarrow (\eta)_\lambda^\mu$ 表示：“如果集合 X 满足 $|X| \geq \kappa$ 且有一个分配映射 $f : [X]^\kappa \rightarrow [0, \lambda - 1] \cap \mathbb{N} = \{0, 1, \dots, \lambda - 1\}$ ，则存在集合 $Y \subset X$ 使得 $|Y| \geq \eta$ 且 $\exists a < \lambda \forall A \in [Y]^\kappa, f(A) = a$ ”。最简单的情形是

$$6 \rightarrow (3)_2^2$$

其表示“6 个人中至少存在 3 人相互认识或者相互不认识”。设一个指标集 I （超积的基数可以为无穷）和一族一阶语言模型 $\mathcal{M}_i = (M_i, I_i), i \in I$ ， D 是 I 的一个超滤子，先定义一般的笛卡儿积 $M = (\prod_{i \in I} M_i)/D$ 作为新的论域，并且相应的解释 I 为

- (1) 常数 $c \in C$: $I(c) = [(I_1(c), \dots, I_i(c), \dots)]_D \subset M$
- (2) 谓词 $P^n \in \mathcal{P}$: $(f_1, \dots, f_n) \in I(P^n) \Leftrightarrow \{i \in I \mid (f_{1i}, \dots, f_{ni}) \in I_i(P^n)\} \in D$ （其中 $f_j = (f_{j1}, \dots, f_{ji}, \dots) \in M, i \in I$ ）
- (3) 函数 $F^n \in \mathcal{F}$: $I(F^n)(f_1, \dots, f_n) = [(I_1(F^n)(f_{11}, \dots, f_{n1}), \dots, I_i(F^n)(f_{1i}, \dots, f_{ni}), \dots)]_D \in M$

此时我们把模型 $\mathcal{M} = (M, I)$ 称为 $\mathcal{M}_i = (M_i, I_i), i \in I$ 的超积（模型）。超积其实就相当于模型的笛卡儿积，虽然看起来麻烦，但其实就是一个纸老虎，下面的定理表明了超积模型的合理性。

定理 3.22 (超积基本定理)

设模型 $\mathcal{M} = (M, I)$ 是模型簇 $\mathcal{M}_i = (M_i, I_i), i \in I$ 的超积， D 是 I 的超滤子。对任意的公式 $A(x_1, \dots, x_n) \in \mathcal{L}_1$

和元素 $f_1, \dots, f_n \in M$ 有

$$\vDash_M A[f_1, \dots, f_n] \Leftrightarrow \{i \in I \mid \vDash_{M_i} A[f_{1i}, \dots, f_{ni}]\} \in D$$



超积可以用来证明紧致性定理、研究初等类、讨论超幂，但也没什么有趣的内容，唯一能说的是一个与初等等价有关的定理（参考 [2] 的 Theorem 2.5.36）

定理 3.23 (Keisler-Shelah 定理)

对于两个一阶语言的模型 M 和 N , $M \equiv N$ 当且仅当存在一个指标 I 和它的一个超滤子 D 使得 $(\prod_I M)/D \cong (\prod_I N)/D$ 。



我们最后再来介绍一下模型论中“力迫法”，如果 $\text{sig}(\mathcal{L}_1) \subset \text{sig}(\mathcal{L}_1^*)$, 我们就称 \mathcal{L}_1^* 是 \mathcal{L}_1 的膨胀, \mathcal{L}_1 是 \mathcal{L}_1^* 的归约。我们考虑最简单的常数膨胀, 设 $C_\infty \cap C = \emptyset$, 有 $\mathcal{L}_1(C) = \mathcal{L}_1 \cup C_1$, 设 T 是 \mathcal{L}_1 的一个一致理论, 如果 $S \subset \mathcal{L}_1(C)$ 仅由原子公式或原子公式的否定组成且 $T \cup S$ 是一致的, 就把 S 称为一个 T -条件。

定义 3.6

对于公式 $A \in \mathcal{L}_1(C)$ 和 T -条件 S , 我们归纳定义 S 力迫 A (记 $S \Vdash A$) 为

- (1) 若 A 为原子公式, 则 $S \Vdash A$ 当且仅当 $A \in S$
- (2) $S \Vdash \neg A$ 当且仅当不存在 T -条件 S_0 使得 $S \subset S_0, S_0 \Vdash A$
- (3) $S \Vdash (A \vee B)$ 当且仅当 $S \Vdash A$ 或 $S \Vdash B$
- (4) $S \Vdash \exists x A(x)$ 当且仅当存在 $c \in C_1$ 使得 $S \Vdash A(c)$



此处我们使用 $\{\neg, \vee, \exists\}$ 代替 $\{\neg, \rightarrow, \forall\}$ 作为初始符号是合理的, 因为我们有

$$\forall x A := \neg \exists x (\neg A), A \rightarrow B := \neg A \vee B$$

如果 $S \Vdash \neg \neg A$, 我们就称 S 弱力迫 A , 并记 $S \Vdash^\omega A$, 下面是一些简单的性质罗列。

性质 设 $A \in \mathcal{L}_1(C)$ 是公式, S, S_1 是 T -条件

- (1) 若 $S \Vdash A$ 且 $S \subset S_1$, 则 $S_1 \Vdash A$
- (2) 若 $A \in S$, 则 $S \Vdash A$
- (3) $S \Vdash^\omega A$ 当且仅当对任意 $S \subset S_1$ 存在 T -条件 S_2 使得 $S_1 \subset S_2$ 且 $S_2 \Vdash A$
- (4) 若 $S \Vdash A$, 则 $S \Vdash^\omega A$
- (5) $S \Vdash A$ 和 $S \Vdash \neg A$ 最多只有一个成立
- (6) 若 A 是原子公式或原子公式的否定且 $S \Vdash A$, 则 $S \cup \{A\}$ 是 T -条件
- (7) $S \Vdash^\omega \neg A$ 当且仅当 $S \Vdash \neg A$
- (8) $S \Vdash \forall x A(x)$ 当且仅当对任意 $c \in C_1, S \Vdash^\omega A(c)$

为了以后的方便, 常数集就直接写出, 并把 \mathcal{L} 的常数集记为 $C_{\mathcal{L}} \subset \text{sig } \mathcal{L}$ 。对于一阶语言 \mathcal{L} , 我们引入下面的符号集 $\Delta_0 = \Sigma_0 = \Pi_0$ 为

- (1) 如果 $t_1, t_2 \in \mathcal{I}$ 是项, 则 $t_1 = t_2 \in \Delta_0$
- (2) 如果 $A, B \in \Delta_0$, 则 $\neg A, A \rightarrow B \in \Delta_0$
- 并递归地定义 $\Delta_n, \Sigma_n, \Pi_n$ 为
 - (1) 如果 $B \in \Pi_{n-1}, A = \exists x B$, 则 $A \in \Sigma_n$
 - (2) 如果 $B \in \Sigma_{n-1}, A = \forall x B$, 则 $A \in \Pi_n$
 - (3) 如果 $A \in \Pi_{n-1} \cap \Sigma_{n-1}$, 则 $A \in \Delta_n$

对于理论 T , 我们记

$$T_V = \{A \in \Pi_1 \mid \vdash_T A\}$$

性质 (1) S 是 T -条件当且仅当 S 是 T_V -条件

(2) 性质 (1) 的意思是, 在理论 T 下 $S \Vdash A$ 当且仅当在理论 T_V 下 $S \Vdash A$

(3) 对于 T -条件 S , 我们记 $T^f(S) = \{A \in \mathcal{L} \mid S \Vdash^\omega A\}$, 并将 $T^f = T^f(\emptyset)$ 称为 T 的有限力迫伴随理论, 则有

$$T^f = (T_V)^f$$

(4) 对于 $A \in \Pi_1$ 和 T -条件 S 有, $S \Vdash A$ 当且仅当 $T \cup S \vdash A$

在最后的命题中 $T \cup S$ 不一定是理论, 所以我们将其视为证明的前提。针对公式我们继续引入新概念

定义 3.7

设 G 是膨胀语言 $\mathcal{L}(C)$ 的仅由原子公式或原子公式的否定组成的公式集, 如果对于 \mathcal{L} 的理论 T 有

(1) G 的任意有限子集都是 T -条件

(2) 对任意公式 $A \in \mathcal{L}(C)$, 都有 $S \subset G$ 使得, $S \Vdash A$ 或 $S \Vdash \neg A$

则称 G 是 T 在 $\mathcal{L}(C)$ 中的兼纳集, 或简称 T -兼纳集。



在兼纳集定义中, $S \Vdash A$ 或 $S \Vdash \neg A$ 有且只有一个成立, 此时, 当存在 S 使得 $S \Vdash A$ 成立时我们记 $G \Vdash A$, 并称作兼纳集 **G 力迫 A**, 通过这个符号和兼纳集的定义可知, $G \Vdash A$ 和 $G \Vdash \neg A$ 有且只有一个成立。

定理 3.24 (兼纳集存在性)

若 \mathcal{L} 是可数一阶语言, C 是可数常量集, T 是 \mathcal{L} 的一致理论, S 是 $\mathcal{L}(C)$ 的 T -条件, 则存在 T -兼纳集 G 使得 $S \subset G$ 。



对于不可数语言是可以没有兼纳集, 下面是兼纳集的基础性质

性质 (1) 若 A 是 $\mathcal{L}(C)$ 的原子公式, 则 $\vdash A \Rightarrow G \Vdash A$

(2) $G \Vdash A$ 当且仅当 $G \nvDash A$

(3) $G \Vdash A \wedge B$ 当且仅当 $G \Vdash A$ 且 $G \Vdash B$

(4) $G \Vdash A \vee B$ 当且仅当 $G \Vdash A$ 或 $G \Vdash B$

(5) 若 $\vdash A \rightarrow B$ 且 $G \Vdash A$, 则 $G \Vdash B$

接着我们在从理论走向模型

定理 3.25 (兼纳模型的唯一存在性)

设 G 是 $\mathcal{L}(C)$ 的 T -兼纳集, 则存在 $\mathcal{L}(C)$ 在同构意义下唯一的模型 $\mathcal{M}(G) = (M, I)$ 使得 (1) $I(C) = M$

(2) 对任意公式 $A \in \mathcal{L}(C)$, $\models_{\mathcal{M}(G)} A$ 当且仅当 $G \Vdash A$



我们也把上面的唯一模型 $\mathcal{M}(G)$ 称为由 G 生成的 T -兼纳模型。对于兼纳模型, 其基础性质如下

性质 设 $\mathcal{M}(G), \mathcal{M}(G_1), \mathcal{M}(G_2)$ 是 T -兼纳模型, $S \subset G$ 是 T -条件

(1) 若 $S \Vdash^\omega A$, 则有 $\models_{\mathcal{M}(G)} A$

(2) 若 \mathcal{L} 可数, 则 $S \Vdash^\omega A$ 当且仅当对任意 $G \supset S$ 有 $\models_{\mathcal{M}(G)} A$

(3) 任意 $\mathcal{M}(G)$ 均存在 $\mathcal{M} \supset \mathcal{M}(G)$ 使得 $\models_{\mathcal{M}} T$, 即任意 T -兼纳模型可以扩充为 T 的模型

(4) 若 $\mathcal{M}(G_1) \subset \mathcal{M}(G_2)$, 则 $\mathcal{M}(G_1) \prec \mathcal{M}(G_2)$

(5) 任意 T -兼纳模型 $\mathcal{M}(G)$ 是 T -存在完备模型, 即对任意语句 $A \in \Sigma_1$ 有“如果 T 的模型 \mathcal{M} 满足 $\models_{\mathcal{M}} A$, 则有 $\models_{\mathcal{M}(G)} A$ ”

(6) 若理论 T 满足 $T \subset \Pi_2$, 则每一个 T -兼纳模型 $\mathcal{M}(G)$ 都是 T 的模型

兼纳模型理论可以用来证明省略型定理, 不过我们并不需要, 模型论的力迫法来自于 Robinson 在公理集合论中用于研究连续统的力迫法的移植, 因此我们相当于介绍了力迫法的一类思想, 在我们后续讨论连续统时会带来一定的便利。

3.2 非标准模型

模型论确实十分的抽象，给人一种毫无用处的感觉，因此我们决定考虑一些实例，以让读者体会模型论所带来的内容。对于理论 T 的各种模型，我们可以做一件事情，就是将符号某些条件的模型称为标准模型，并把其它的称为非标准模型，比如对于一阶算术理论 N ，我们把初等数论模型 $(\mathbb{N}, +, \times, S, 0)$ 称为 **标准算术模型**，并把与初等数论模型不同构的模型称为**非标准算术模型**，非标准算术模型的例子很好找，只要添加²无穷 ∞ 并引入下列的运算规则即可

$$\infty + \infty = \infty, \infty \times \infty = \infty, S(\infty) = \infty, \infty + 0 = \infty, \infty \times 0 = 0$$

如果上面的方式过于露骨了，我们可以用性质来定义标准模型。对于 N 的一个模型 $M = (M, I)$ 和元素 $a \in M$ ，如果存在 $n < \aleph_0$ 使得

$$I(S^n(0)) = a$$

就称 a 是**有限元**，否则称 a 是**非标准元**。对于模型 $M = (M, I)$ ，若论域 M 中的所有元素都是有限元，则称 M 是**标准算术模型**，若论域 M 存在非标准元，则称 M 是**非标准算术模型**，此时我们可以证明

定理 3.26

在同构意义下

- (1) N 只有唯一的标准算术模型，即初等数论模型 $(\mathbb{N}, +, \times, S, 0)$
- (2) N 有 $\aleph_1 = 2^{\aleph_0}$ 个非标准算术模型



这些事实告诉我们什么？告诉我们在算术中使用 ∞ 是逻辑上合法的，只不过我们使用的是非标准模型，而不是初等数论这种标准模型。玩算术的时候，我们可能遇不到这么多无穷的概念，但在分析中可就太多了，接下来让我们来讨论在分析中，如何使用无穷大和无穷小来进行分析。

标准分析

对于数学分析，我想读者应该是十分熟悉的，其有两者逻辑上的引入方式。一种是传统教材法，其默认大家清楚实数 \mathbb{R} 的性质，即以实闭有序域理论 RCF_o 的一个模型 $(\mathbb{R}, 0, 1, <, +, \times)$ 作为前提，通过 $\varepsilon - \delta$ 语言来严格定义极限，从而引出分析理论。另一种则是实数定义法，其认为实数的认知不够严谨，因此从皮亚诺算术 PA 的初等数论模型 $(\mathbb{N}, 0, +, \times, S)$ 开始，通过一层层的定义扩充

$$\mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q} \rightarrow \mathbb{R}$$

最终从 \mathbb{R} 的定义中自然得到极限，从而引出分析理论。我个人比较倾向于后一种认知，这两种理论的区别在于，前者中“柯西极限存在准则”作为定理存在，而后者中“柯西极限存在准则”作为定义存在，所以后者阐释了我一直以来的理念，即“分析学是实数的四则运算”，分析是实数的筋骨，没有分析就没有实数的意义，这就是我喜欢后者的原因。实际上，如果我们只从 $(\mathbb{R}, 0, 1, <, +, \times)$ 的公理出发，通过一系列的定义和证明，是不足以推出柯西极限存在准则的，我们总得引入一条说明了“实数连续性”的定理才行。比如陈纪修的数学分析一书中，证明“柯西极限存在准则”用了“有界数列必有收敛子列”（相当于 Weierstrass 聚点定理），证明“有界数列必有收敛子列”用了“区间套定理”，证明“区间套定理”用了“单调有界原理”，证明“单调有界原理”用了“确界原理”，我们知道这些其实都是实数完备性的等价定理。关键在于最后一个，如果我们使用戴德金分割作为实数的定义，那么确界原理就是戴德金分割的自然导出，但大多教材对于“确界原理”的证明就十分诡异了，

²请读者注意了，我们只是添加了一个符号以兼容算术的运算， $\infty \times 0 = 0$ 是我们规定的，它的意义并不重要，只要在逻辑上说得通即可

我个人觉得就别去看了，实际上，如果我们构造实闭有序域理论 RCF_o 的一个模型 $(\mathbb{R} \cap \overline{\mathbb{Q}}, 0, 1, <, +, \times)$ ，则我们随便取个超越数的截断小数，比如 π 的截断小数

$$\{3, 3.1, 3.14, 3.141, 3.1415, 3.14159, \dots\}$$

就在 $\mathbb{R} \cap \overline{\mathbb{Q}}$ 中没有上确界，因此“确界原理”在模型 \mathbb{R} 中是可满足的，而在模型 $\mathbb{R} \cap \overline{\mathbb{Q}}$ 中是不可满足的，但我们知道 RCF_o 是完全可靠的理论，这意味着“确界原理”在 RCF_o 中不是有效的，因此仅通过 RCF_o 的公理是无法证明确界原理的。但我们只需随便添加一个实数完备性的等价定理，就可以得到分析理论了，而这又相当于回到了第二种认知法。

超实数

标准分析作为一般的课程，我们无需做过多的讨论，因此我们马上就转向非标准分析的研究。如果你想要深入了解非标准分析，请看一些比较新的讲义或文章（例如参考 1, 参考 2, 参考 3, 参考 4），否则你会十分地折磨（例如 Abraham Robinson 的原书）。我们先来讲一下非标准分析的逻辑构造过程，首先我们把理论 RCF_o 的实分析模型 $\mathbf{R} = (\mathbb{R}, <, 0, +, \times)$ 称为标准分析，我们在常数中加入³无穷大 ∞ ，并使用下面公理生成一个公式集

$$\text{Th}(\mathbf{R} \cup \{r < \infty \mid r \in \mathbb{R}\})$$

根据紧致性定理可知，两个定理集的可满足性是一样的，因此存在上述公式集的一个模型 \mathbf{R}^* 使得它是 $\text{Th}(\mathbf{R})$ 的模型且有

$$\mathbf{R} \equiv \mathbf{R}^*, \mathbf{R} \prec \mathbf{R}^*$$

因此存在一个模型 \mathbf{R}' 使得

$$h : \mathbf{R}^* \cong \mathbf{R}'$$

且 \mathbf{R} 是 \mathbf{R}' 的子模型，我们把 \mathbf{R}^* 称为 RCF_o 的非标准分析，并把相应论域 \mathbb{R}^* 称为超实数。由同构我们可以得到常量、关系和函数的对应

$$h : (\mathbb{R}, 0, +, \times, <) \cong (\mathbb{R}', 0, +, \times, <) \rightarrow (\mathbb{R}^*, 0^*, +^*, \times^*, <^*)$$

方便起见，在 \mathbb{R}^* 中省略一切常数、关系和函数的星号，简写乘法 $x \times y \rightarrow xy$ ，并离开模型论，进入非逻辑研究。首先，我们引入子集

$$\mathbb{R}_{\text{fin}} = \{x \in \mathbb{R}^* \mid \exists n \in \mathbb{N}, |x| < n\}$$

$$\mathbb{R}_{\text{inf}} = \mathbb{R}^* - \mathbb{R}_{\text{fin}}$$

$$\mathbb{R}_{\text{sim}} = \{x \in \mathbb{R}^* \mid \forall n > 0, |x| < \frac{1}{n}\}$$

并把它们的元素分别称为有限数、无穷大量和无穷小量，显然我们有

$$\mathbb{R}_{\text{sim}} \subset \mathbb{R}_{\text{fin}}, \mathbb{R} \subset \mathbb{R}_{\text{fin}}, \mathbb{R} \cap \mathbb{R}_{\text{sim}} = \{0\}$$

性质 (1) 若 $x, y \in \mathbb{R}_{\text{fin}}$ ，则 $x \pm y, xy \in \mathbb{R}_{\text{fin}}$

(2) 若 $x, y \in \mathbb{R}_{\text{sim}}$ ，则 $x \pm y, xy \in \mathbb{R}_{\text{sim}}$

³实际上，也可以加入无穷小 $\text{Th}(\mathbf{R} \cup \{0 < \varepsilon < r \mid r \in \mathbb{R}\})$

(3) 若 $x \in \mathbb{R}_{\text{fin}}, y \in \mathbb{R}_{\text{sim}}$, 则 $xy \in \mathbb{R}_{\text{sim}}$

从环的观点来看, $\mathbb{R}_{\text{fin}}, \mathbb{R}_{\text{sim}}$ 是 \mathbb{R}^* 的子环, \mathbb{R}_{sim} 是 \mathbb{R}_{fin} 的理想, 于是我们可以考虑商环 $\mathbb{R}_{\text{fin}}/\mathbb{R}_{\text{sim}}$ 的含义, 此时有定义

定义 3.8 (无限接近)

如果 $x, y \in \mathbb{R}^*$ 满足 $x - y \in \mathbb{R}_{\text{sim}}$, 则称 x 与 y 无限靠近, 并记为 $x \approx y$



性质 (1) \approx 是 \mathbb{R}^* 上的等价关系

- (2) 如果 $u \approx v, x \approx y, u + x \approx v + y$ 且 $-x \approx -y$
- (3) 如果 $x, y, u, v \in \mathbb{R}_{\text{fin}}$ 且 $u \approx v, x \approx y, ux \approx vy$
- (4) 如果 $r, s \in \mathbb{R}$, 则有 $r \approx s \Leftrightarrow r = s$
- (5) 对任意的 $x \in \mathbb{R}_{\text{fin}}$ 均存在唯一 $r \in \mathbb{R}$ 使得 $x \approx r$

由性质 (5) 可知, 对任意 $x \in \mathbb{R}_{\text{fin}}$, 存在唯一分解

$$x = s + i, s \in \mathbb{R}, i \in \mathbb{R}_{\text{sim}}$$

我们把 s 称为 x 的标准部分, 并记为 $\text{st}(x) = s$ 。

性质 (1) $\text{st} : \mathbb{R}_{\text{fin}} \rightarrow \mathbb{R}$ 满射环同态

- (2) $\mathbb{R}_{\text{fin}}/\mathbb{R}_{\text{sim}} \cong \mathbb{R}$
- (3) \mathbb{R}^* 不满足“确界原理”, 从而是不完备的

仅有超实数 \mathbb{R}^* 是不够的, 我们还要选一个算术的非标准模型 \mathbb{N}^* , 由标准分析的第二种构造法可知 \mathbb{R}^* 自然引出了相应的非标准自然数 \mathbb{N}^* , 并有下面的定义和性质

$$\mathbb{N}_{\text{inf}} = \mathbb{N}^* - \mathbb{N} = \{x \in \mathbb{N}^* \mid \forall n \in \mathbb{N}, x > n\}$$

由于非标准自然数的内容并不多, 我们以后直接记 “ $N > \mathbb{N}$ ” 表示 “ $N \in \mathbb{N}_{\text{inf}} = \mathbb{N}^* - \mathbb{N}$ ”。

分析学

在上面我们理清了超实数的概念和性质, 并且模型论保证了其存在性, 数理逻辑保证了其和标准分析的初等等价, 接着我们应该就此来讨论分析学了, 需要研究的对象无非就三个“数列极限”、“函数微积分”和“级数”, 让我们来一一讨论。

定理 3.27

- (1) 对于实数列 $a_n : \mathbb{N} \rightarrow \mathbb{R}$, $\lim_{n \rightarrow \infty} a_n = a$ 当且仅当 $\forall N > \mathbb{N}, a_N \approx a$
- (2) 对于实函数 $f : D \subset \mathbb{R} \rightarrow \mathbb{R}$, $\lim_{x \rightarrow a} f(x) = b$ 当且仅当 $x \approx a \Rightarrow f(x) \approx b$
- (3) 对于级数 $\sum_{i=0}^{\infty} a_i, \sum_{i=0}^{\infty} a_i = a$ 当且仅当 $\forall N > \mathbb{N}, \sum_{i=0}^N a_i \approx a$



上面给出了数列极限、函数极限和级数极限的转换原理, 一旦得到了极限的概念, 我们就可以引入微积分了。

定义 3.9 (微分)

对于函数 $f : D \subset \mathbb{R} \rightarrow \mathbb{R}$ 和 $a \in D, b \in \mathbb{R}$, 如果对任意 $\varepsilon \in \mathbb{R}_{\text{sim}} - \{0\}$ 均有 $\frac{f(c+\varepsilon)-f(c)}{\varepsilon} \approx b$, 就称 b 是 f 在 a 处的导数, 并记

$$df := f(x + \varepsilon) - f(x), dx := \varepsilon, f'(x) = \text{st}\left(\frac{df}{dx}\right)$$



在非标准分析中, 我们可以摆脱 $\varepsilon - \delta$ 语言的掌控, 而直接计算导数, 比如 $f(x) = x^2$ 有

$$f'(x) = \text{st}\left(\frac{df}{dx}\right) = \text{st}\left(\frac{(x+dx)^2 - x^2}{dx}\right) = \text{st}(2x + dx) = 2x$$

这其实就是微积分最初的形态，可谓是返朴归真了，更多微分的基础性质和传统分析基本一样，无非就是证明有些不同，所以就不讨论了，最后再看一下积分。对于函数 $f : [a, b] \rightarrow \mathbb{R}$ 和 $\Delta x \in \mathbb{R}_{\text{sim}}^{>0}$ ，我们记

$$U(f, \Delta x) = \sum_{i \in \mathbb{N}^*} \max\{f([x_i, x_{i+1}])\} \Delta x, x_{i+1} - x_i = \Delta x$$

$$L(f, \Delta x) = \sum_{i \in \mathbb{N}^*} \min\{f([x_i, x_{i+1}])\} \Delta x, x_{i+1} - x_i = \Delta x$$

定义 3.10 (积分)

如果对任意 $\Delta x \in \mathbb{R}_{\text{sim}}^{>0}$ 均有 $U(f, \Delta x) \approx L(f, \Delta x)$ ，则称 f 是黎曼可积的，并记

$$\int_a^b f(x) dx = \text{st}(U(f, \Delta x)) = \text{st}(L(f, \Delta x))$$



至此非标准分析已经建立完毕，可以看到非标准分析本质就是，对 $\varepsilon - \delta$ 语言做了一次直观的转化，所以在大多非标准书籍中都会提到“非标准分析与其说是一门学科，不如说是一种方法”。对于非标准分析一般有两种认知途径，一种就是传统方法，从模型论中推导出相应的非标准模型，比如我们介绍的 Robinson 构造和完全公理化的 Nelson 构造，另一种则是，将标准分析中的无穷小量和无穷大量添加到实数中，以形成超实数，并相应地把“相等”、“大小”、“和积”等推广上去，这其实就是大家经常玩分析时的思想，虽然很多证明中都不会把非标准分析放到明面上，但实际大家都在无形之中使用了非标准分析。我们从实数域 \mathbb{R} 开始，如果有极限 $\lim_{x \rightarrow 0} f(x) = 0$ 我们就把整体 $\lim_{x \rightarrow 0} f(x)$ 称为无穷小量，此时引入无穷小量集合为

$$\mathbb{R}_{\text{sim}} = \{0\} \cup \{\lim_{x \rightarrow 0} f(x) \mid \lim_{x \rightarrow 0} f(x) = 0\}$$

$$\lim_{x \rightarrow 0} f(x) + \lim_{x \rightarrow 0} g(x) := \lim_{x \rightarrow 0} (f(x) + g(x)), \lim_{x \rightarrow 0} f(x) \lim_{x \rightarrow 0} g(x) := \lim_{x \rightarrow 0} (f(x)g(x))$$

由无穷小量的性质可知上述加法和乘法的性质是合理的，从而 \mathbb{R}_{sim} 形成了一个环，我们让它和 \mathbb{R} 进行作用，得到有限数集合为

$$\mathbb{R}_{\text{fin}} = \{r + \varepsilon \mid r \in \mathbb{R}, \varepsilon \in \mathbb{R}_{\text{sim}}\}$$

$$r \in \mathbb{R}, \varepsilon = \lim_{x \rightarrow 0} f(x) \in \mathbb{R}_{\text{sim}}, r\varepsilon := \lim_{x \rightarrow 0} (rf(x)) \in \mathbb{R}_{\text{sim}}$$

$$(r_1 + \varepsilon_1) + (r_2 + \varepsilon_2) := (r_1 + r_2) + (\varepsilon_1 + \varepsilon_2), (r_1 + \varepsilon_1)(r_2 + \varepsilon_2) := r_1r_2 + (r_1\varepsilon_2 + r_2\varepsilon_1 + \varepsilon_1\varepsilon_2)$$

同样可以得到 \mathbb{R}_{fin} 形成一个环，此时我们只需补充定义无穷，即可得到超实数

$$\mathbb{R}_{\text{inf}} = \left\{ \frac{1}{\varepsilon} \mid \varepsilon \in \mathbb{R}_{\text{sim}} - \{0\} \right\}, \mathbb{R}^* = \mathbb{R}_{\text{fin}} \cup \mathbb{R}_{\text{inf}}$$

此处我们只做一个引导，深入内容留给读者自行探索。由于分析学在数学中的基础地位，非标准分析的方法是可以扩充到“实变函数”、“复变函数”和“泛函分析”等领域上的，并且有时还能在证明中取得极大简便，不过我只是一个领路人，就不做过多讨论了，值得一提的是非标准分析在希尔伯特第 5 问证明中有简化作用。

定理 3.28 (希尔伯特第 5 问)

如果拓扑群 G 是局部欧式化的，则 G 同构于一个 Lie 群。



详细可以参考陶哲轩的“Hilbert’s Fifth Problem and Related Topics”一书的 1.7 节。

第四章 计算与递归

上面我们讨论的语言、逻辑和模型其实都是比较偏向于哲学的内容，这次我们将目光放到计算机领域上，考虑一些偏向于计算的形式系统，并探究其与一阶语言的联系。这部分内容也同样会分布在各类数理逻辑的书籍中，但这次我们更推荐去阅读计算机相关的书籍，比如这一章中，我们的参考书是这本 [10]。虽然我好心地找了一本还不错的中译教材，但我觉得原文也值得读一读。

4.1 几个系统

自动机与文法

在计算机领域中，我们通常认为可枚举集就是指可数集，望读者警醒这个术语。我们先来讲一些比较基础的计算系统，自动机和形式文法，它们是编译原理和程序设计语言的基础理论，我们同样以形式化的方法进行介绍。没办法，这是我的特色，想要生动的讲解就去看原书吧。

定义 4.1 (有穷自动机)

有穷自动机由 5 部分组成 $(Q, \Sigma, \delta, q_0, F)$ ，其中

- (1) Q 是有限集，称为状态集
- (2) Σ 是有限集，称为字母表
- (3) $\delta : Q \times \Sigma \rightarrow Q$ 称为转移函数
- (4) $q_0 \in Q$ 称为起始状态
- (5) $F \subset Q$ 称为接受状态

嗯？你可能会有疑惑，所谓的状态机难道就是定义了一堆名词，确实如此，数学的形式系统通常就是搞成一堆术语，并进行一定程度的限制，比如 (3)(4)(5) 就是限制，它们分别限制了 δ 的定义域和值域、 q_0 是哪里的元素、 F 是谁的子集。一个具体的形式系统就是像下面一样的一堆符号

$$(1) Q = \{q_1, q_2, q_3\}$$

$$(2) \Sigma = \{0, 1\}$$

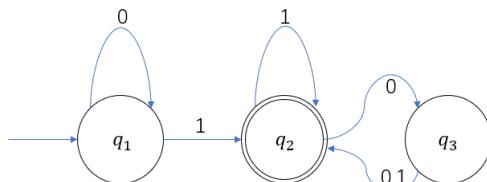
(3) δ 运算表

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

$$(4) q_0 = q_1$$

$$(5) F = \{q_2\}$$

一个具体形象地表示出一个有穷自动机的方法是图论法，步骤如下 (1) 在平面上画出与 Q 数量相当的圆用来表示所有状态 (2) 对于属于 F 的状态，再画上一层圆，来表示接受状态 (3) 画一个无源头箭头指向初始状态 q_0 ，来表示起始状态 (4) 对于转移函数 δ 的每个对应 $(q_i, b) \rightarrow q_j$ ，使用 q_i 到 q_j 的箭头并标上字母 b 。此时上面实例的系统就是下面的这张图



自动机拿来干嘛呢？当然是识别语言了，对于任意字母表 Σ ，我们把元素 $s \in \Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^n$ 称为一个字符串，即任意由字母构成的任意有限长的序列

$$(a_1), (a_1, a_2, a_3), (a_1, a_2, a_2, a_1), (a_1, a_1, a_4, a_4, a_3) \dots, a_1, a_2, a_3, a_4 \in \Sigma$$

有时为了方便我们直接省略 (,)，写成

$$a_1, a_1 a_2 a_3, a_1 a_2 a_2 a_1, a_1 a_1 a_4 a_4 a_3, \dots$$

其中字母的个数，或者序列的长度称为字母的长度，并记为 $|s|$ ，例如

$$|a_1| = 1, |a_1 a_2 a_3| = 3, |a_1 a_2 a_2 a_1| = 4, |a_1 a_1 a_4 a_4 a_3| = 5, \dots$$

特别地，我们引入空串 ε 满足 $|\varepsilon| = 0$ 。通常我们将字符串子集 $S \subset \Sigma^*$ 称为字母集 Σ 的一个语言，此时我们定义

定义 4.2 (接受)

对一有穷自动机 $M = (Q, \Sigma, \delta, q_0, F)$ 和字符串 $w = w_1 \dots w_n$ （对任意 $1 \leq i \leq n$ 有 $w_i \in \Sigma$ ）。如果存在状态序列 r_0, r_1, \dots, r_n 使得

- (1) $r_0 = q_0$
- (2) $\delta(r_i, w_{i+1}) = r_{i+1}$
- (3) $r_n \in F$

我们就称 M 接受 w 。我们将所有被 M 接受的字符串构成的集合记为 $L(M) \subset \Sigma^*$ ，并称 $L(M)$ 是机器 M 的语言，或称 M 识别语言 $L(M)$

同样是我们上面举得例子，它识别的语言是

$$\{w \mid w \text{ 至少含有一个 } 1 \text{ 并且最后一个 } 1 \text{ 的后面有偶数个 } 0\}$$

比如“111,11100,00010100”都可以被识别，“000,01000,0101010”都不可以被识别，我们将能被一台有穷自动机识别的语言称为正则语言。为了研究哪些语言是正则语言，我们有必要研究一下语言间的运算，并考察其是否能保持正则性。对于两个语言 $A, B \subset \Sigma^*$ ，我们定义下面的三种正则运算

(1) 并: $A \cup B = \{x \mid x \in A \vee x \in B\}$

(2) 连接: $AB = \{xy \mid x \in A, y \in B\}$

(3) 星号: $A^* = \{x_1 x_2 \dots x_k \mid \forall k \geq 0, x_i \in A\}$

前两个就跟它的名字一样，而最后一个相当于一个语言的任意自连接的并，只不过当 $k = 0$ 时，一定会把空串给引入进去

$$A^* = \{\varepsilon\} \cup (\bigcup_{i=1}^{\infty} \underbrace{A \dots A}_k)$$

显然它和 Σ^* 的含义互相兼容，我们可以证明

定理 4.1

任意正则语言经过有限次正则运算后依旧是正则语言。



基本思想就是考察基本运算“并”和“连接”，和基本语言 $\{\varepsilon\}$ 并设计有穷自动机的过程。只不过证明中，引入了一种新的形式系统，我们使用 2^X 来表示 X 的幂集，即 X 的所有子集构成的集合

定义 4.3 (非确定型有穷自动机)

非确定型有穷自动机由 5 部分组成 $(Q, \Sigma, \delta, q_0, F)$, 其中

- (1) Q 是有限集, 称为状态集
- (2) Σ 是有限集, 称为字母表
- (3) $\delta : Q \times \Sigma \rightarrow 2^Q$ 称为转移函数
- (4) $q_0 \in Q$ 称为起始状态
- (5) $F \subset Q$ 称为接受状态



它与有穷自动机的唯一区别就是转移函数的取值是非唯一的, 至于接受过程也是只改了一个符号

定义 4.4 (接受)

对一非确定型有穷自动机 $M = (Q, \Sigma, \delta, q_0, F)$ 和字符串 $w = w_1 \dots w_n$ (对任意 $1 \leq i \leq n$ 有 $w_i \in \Sigma$)。如果存在状态序列 r_0, r_1, \dots, r_n 使得

- (1) $r_0 = q_0$
- (2) $r_{i+1} \in \delta(r_i, w_{i+1})$
- (3) $r_n \in F$

我们就称 M 接受 w



既然它们这么地像自然就有一个等价性定理了

定理 4.2

每一台非确定型有穷自动机都等价于某一台确定型有穷自动机。



上述定理也是构造证明的, 其含义是“任意一台非确定型有穷自动机的语言都等于某一台确定型有穷自动机的语言, 反之亦然”。此时我们终于可以来探讨什么样的语言是正则语言, 答案是正则表达式, 名字都对应上了, 其严格定义为

定义 4.5

我们递归定义正则表达为

- (1) 如果 $a \in \Sigma$ 是字母, 则 $\{a\}$ 是正则表达
- (2) $\{\varepsilon\}$ 是正则表达
- (3) \emptyset 是正则表达
- (4) 如果 R_1, R_2 是正则表达式, 则 $R = R_1 \cup R_2$ 是正则表达
- (5) 如果 R_1, R_2 是正则表达式, 则 $R = R_1 R_2$ 是正则表达式
- (6) 如果 R_1 是正则表达式, 则 $R = R_1^*$ 是正则表达式



按照传统程序语言的习惯, 对于单个字母的语言 $\{a\}, \{\varepsilon\}$ 我们可以直接写成 a, ε , 另外我们还能定义 $R^+ := RR^*$ 表示一个或多个 R 中字符串连接组成的字符串。在程序语言中通过字符串后面加 $\{n\}, \{n,\}, \{n, m\}$ 来表示恰好 n 次、至少 n 次、至少 n 次且不超过 m 次, 在数学里我们就直接用指数的方法, 即 $R^n = \underbrace{RR\dots R}_n, R^{n, t} = \underbrace{RR\dots R}_k, k \geq n, R^{n, m} = \underbrace{RR\dots R}_t, n \leq t \leq m$ 。读者需要注意正则表达指一个语言, 即字符串的集合, 而不是一个字符串, 放到程序语言的说法就是, 其代表了一类符合某些特征的字符串, 使用字母表 $\Sigma = \{0, 1\}$, 则下面是一些简单的例子

- (1) $0^* 1 0^*$: 恰好有一个 1 的字符串
- (2) $\Sigma^* 1 \Sigma^*$: 至少有一个 1 的字符串
- (3) $\Sigma^* 001 \Sigma^*$: 含有子串 001 的字符串
- (4) $(01^+)^*$: 不含有字串 00 的字符串

(5) $(\Sigma\Sigma\Sigma)^*$: 长度为 3 的倍数的字符串

(6) $0\Sigma^*0 \cup 1\Sigma^*1 \cup 1 \cup 0$ (此处“连接”优先级大于“并”): 首尾相同字母的字符串
此时我们可以得到相应的等价性定理

定理 4.3

一个语言是正则语言当且仅当它是一个正则表达式。



虽然读起来有些绕口，但这套理论是程序语言中正则表达匹配设计的基础。我们很容易举出一个非正则语言的例子

$$B = \{0^n 1^n \mid n \geq 0\}$$

它可以看成无穷 $B_n = \{0^n 1^n\}, n \geq 0$ 的无穷并，这并不属于我们定义的范畴，显然有穷自动机的识别能力是有限的，我们是否可以定义一种识别无穷的语言呢？它是否叫无穷自动机呢？嗯，它其实就是形式文法，我们来严格定义它

定义 4.6 (形式文法)

(1) 形式文法由 4 部分组成 (V, Σ, R, S) , 其中

(i) V 是有限集, 称为变元集, 或称为非终结符号集

(ii) Σ 是有限集, 称为字母表, 或称为终结符号集

(iii) $R = \{\alpha \rightarrow \beta\}$ 是有限集, 称为规则集, 其中 $\alpha, \beta \in (V \sqcup \Sigma)^*$ 且 $\exists A \in V, A \in \alpha$

(iv) $S \in V$ 称为起始变元, 并且存在 $S \rightarrow \beta \in R$

我们也把形式文法称为零型文法

(2) 如果形式文法 (V, Σ, R, S) 满足任意的 $\alpha \rightarrow \beta \in R$ 有

$$|\alpha| \leq |\beta|$$

我们就把它称为一型文法

(3) 如果一型文法 (V, Σ, R, S) 满足任意的 $\alpha \rightarrow \beta \in R$ 有

$$|\alpha| = 1$$

我们就把它称为二型文法, 或称为上下文无关文法

(4) 如果二型文法 (V, Σ, R, S) 满足任意的 $\alpha \rightarrow \beta \in R$ 有

$$\beta = a \in \Sigma \text{ 或 } \beta = aA, a \in \Sigma, A \in V$$

我们就把它称为三型文法



显然形式文法的关键在于理解规则集 $R = \{\alpha \rightarrow \beta\}$, 它其实就是一个替换规则, 表示将 α 替换成 β , 随便乱替换是没有意义的, 因此最初的限制就是 α 中至少有一个变元, 这对后面二型文法稍微产生了一些影响, 因为此时 $|\alpha| = 1$ 且 α 中至少有一个变元, 因此我们只能有 $\alpha \in V$ 是一个变元。在 (1)(iii) 我们引入了一个新的符号, 即字母 $a \in \Sigma$ 属于字符串 $w \in \Sigma^*$ 的概念 $a \in w$, 它其实就是字面那样的意思, 即字符串 w 有形式

$$a \in w \Leftrightarrow w \in \Sigma^* a \Sigma^*$$

不得不说正则表达式真好用。其中二型文法最为常用, 我们举一个简单的实例

(1) 变元集: $V = \{S\}$

(2) 字母表: $\Sigma = \{0, 1\}$

(3) 规则集: $R = \{S \rightarrow \epsilon, S \rightarrow 0S1\}$

(4) 起始变元: $S \in V$

接着我们同样将其运用到语言中

定义 4.7 (派生)

对一形式文法 $G = (V, \Sigma, R, S)$

(1) 如果扩展字符串 $u, w \in (V \sqcup \Sigma)^*$ 满足, 存在规则 $\alpha \rightarrow \beta \in R$ 使得

$$\alpha \in u; \gamma, \omega \in (V \sqcup \Sigma)^*; u = \gamma\alpha\omega, w = \gamma\beta\omega$$

我们就称 u 生成 w , 并记为 $u \Rightarrow w$

(2) 如果扩展字符串 $u, w \in (V \sqcup \Sigma)^*$ 满足, $u = w$ 或者存在扩展字符串序列 $u_1, \dots, u_n \in (V \sqcup \Sigma)^*, n \geq 0$ 使得

$$u \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_n \Rightarrow w$$

我们就称 u 派生 w , 并记为 $u \Rightarrow w$ 。我们把语言

$$L(G) := \{w \in \Sigma^* \mid S \Rightarrow w\}$$

称为文法 G (生成) 的语言



容易发现我们上面给的二型文法实例的语言就是 $B = \{0^n 1^n \mid n \geq 0\}$, 显然形式文法的魅力在于可以处理无限的字符串集, 比有穷自动机更强大是毫无疑问的。有时为了文法表示的便利性, 对于两条规则 $\alpha \rightarrow \beta, \alpha \rightarrow \gamma \in R$, 我们简单记为 $\alpha \rightarrow \beta \mid \gamma$, 比如还是示例的二型文法, 它的规则可以简单写为

$$S \rightarrow \epsilon \mid 0S1$$

当然还可以复合更多规则, 例如语言 $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 2^n \mid n \geq 0\} \cup \{2^n 3^n \mid n \geq 0\}$ 的形式文法为

(1) 变元集: $V = \{S, S_1, S_2, S_3\}$

(2) 字母表: $\Sigma = \{0, 1, 2, 3\}$

(3) 规则集: $R = \{S \rightarrow S_1 \mid S_2 \mid S_3, S_1 \rightarrow \epsilon \mid 0S_11, S_2 \rightarrow \epsilon \mid 1S_22, S_3 \rightarrow \epsilon \mid 2S_33\}$

(4) 起始变元: $S \in V$

值得注意的是任一字符串 $w \in L(G)$ 的生成链可能是不唯一的, 其有两个方面, 一种是顺序不唯一, 这时我们最好的办法就是利用字符串自带的序结构从左到右进行, 并限制其为二型文法, 简单来讲就是在生成链 $u = u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_n = w$ 中的每一步

$$\alpha \in u_{i-1}; \gamma, \omega \in (V \sqcup \Sigma)^*; u_{i-1} = \gamma\alpha\omega, u_i = \gamma\beta\omega, 1 \leq i \leq n$$

均有

$$\forall \alpha_0 \rightarrow \beta_0 \in R, \alpha_0 \notin \gamma,$$

此时我们称其为最左派生 (若非二型文法, 则有可能在字符串 $\alpha\omega$ 的开头包含其它的生成头), 第二种是生成链的内容不一样, 这就是所谓的歧义

定义 4.8 (歧义)

如果字符串 w 在二型文法 G 中有至少两个不同的最左派生, 我们就称 G 歧义地产生字符串 w 。



有时一个语言被歧义地产生时, 我们有可能可以找到另外一个文法非歧义地产生这个语言, 这是程序设计语言的解释器制作时必需考虑的事。但有些语言只能被歧义地生成, 我们称其为固有歧义语言, 比如 $\{a^i b^j c^k \mid i = j \vee j = k\}$ 就是固有歧义语言。这告诉我们一件事实, 一个语言可能由多种二型文法产生, 我们可以试着找一些更规则的方式来缩小其可能性。如果二型文法 $G = (V, \Sigma, R, S)$ 的每一条规则 $\alpha \rightarrow \beta \in R$ 满足以下任意一条

(1) $\alpha = A \rightarrow \beta = BC; A \in V; B, C \in V - \{S\}$

(2) $\alpha = A \rightarrow \beta = a; A \in V; a \in \Sigma$

(3) $\alpha = S \rightarrow \beta = \epsilon$

则称 G 是一个乔姆斯基范式。我们可以证明

定理 4.4

任意一个二型文法生成的语言都可以用一个乔姆斯基范式生成。



实际上，各种计算模型都是可以升级的，比如我们让有穷自动机具备一个用于记忆的栈，就有了下推自动机，对于一个语言 $L, \varepsilon \notin L$ ，我们记 $L_\varepsilon = L \cup \{\varepsilon\}$ ，于是有

定义 4.9 (下推自动机)

下推自动机由 6 部分组成 $(Q, \Sigma, \Gamma, \delta, q_0, F)$ ，其中

- (1) Q 是状态集
- (2) Σ 是字母表
- (3) Γ 是有限集，称为栈字母表
- (4) $\delta : Q \times \Gamma_\varepsilon \times \Sigma_\varepsilon \rightarrow 2^{Q \times \Gamma_\varepsilon}$ 称为转移函数
- (5) $q_0 \in Q$ 称为起始状态
- (6) $F \subset Q$ 称为接受状态



没想到吧，这玩意也是非确定类型的，它对字符串 $w = w_1 w_2 \dots w_n \in \Sigma^*$ 的识别过程如下，即存在状态序列 $r_0, r_1, \dots, r_n \in Q$ 和栈字母字符串序列 $s_0, s_1, \dots, s_n \in \Gamma^*$ 使得

- (1) $r_0 = q_0, s_0 = \varepsilon$
- (2) $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ 且有 $s_i = at, s_{i+1} = bt; a, b \in \Gamma_\varepsilon, t \in \Gamma^*$
- (3) $r_m \in F$

其能力是大于有穷自动机，比如语言 $B = \{0^n 1^n \mid n \geq 0\}$ 就可以被下面的下推自动机识别

- (1) $Q = \{q_1, q_2, q_3, q_4\}$
- (2) $\Sigma = \{0, 1\}$
- (3) $\Gamma = \{0, \#\}$

(4) 我们用空白表示空集 $\emptyset \subset Q \times \Gamma_\varepsilon$ ，则 δ 的运算表为

$(\Sigma, \Gamma_\varepsilon)$	$(0, 0)$	$(0, \#)$	$(0, \varepsilon)$	$(1, 0)$	$(1, \#)$	$(1, \varepsilon)$	$(\varepsilon, 0)$	$(\varepsilon, \#)$	$(\varepsilon, \varepsilon)$
q_1									$\{(q_2, \#)\}$
q_2		$\{(q_2, 0)\}$		$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$					$\{(q_4, \varepsilon)\}$
q_4									

(5) $q_0 = q_1$

(6) $F = \{q_1, q_4\}$

你可能会觉得，如果我们把 $Q \times \Gamma_\varepsilon$ 视为一个状态集的话，不就变成了一个非确定型有穷自动机吗？其实并非如此，首先在字母表中，我们加入了空串 $\Sigma \neq \Sigma_\varepsilon$ ，其次 q_0, F 不属于栈字母表，在这两个限制下，它和非确定型有穷自动机是不同的。其实最大的区别还是在于识别过程，其中的字符串序列 $s_0, s_1, \dots, s_n \in \Gamma^*$ 就相当于一个栈的变化过程，虽然只有栈顶替换的操作，但实际“ a 替换 ε ”相当于“ a 出栈”，“ ε 替换 a ”相当于“ a 入栈”。这时我们就应该意识到文法其实也是一种替换，那么应该有着相应的等价，即有定理

定理 4.5

一个语言由二型文法生成当且仅当存在一个下推自动机识别它。



我们嗅到了一丝文法和自动机的关系，实际上，我们还有

定理 4.6

一个语言由三型文法生成当且仅当存在一个有穷自动机识别它。



另外，读者还要注意一个事实，我们知道有穷自动机的确定型和非确定型是一样的，但如果我们把下推自动机的转移函数改成 $\delta : Q \times \Gamma_\varepsilon \times \Sigma_\varepsilon \rightarrow Q \times \Gamma_\varepsilon$ ，即确定型，那么它的生成能力是下降的，即存在一个语言，它能由非确定型下推自动机生成却不能由确定型下推自动机生成。

图灵机

通过上面的探究，我们自然会想，有没有可以识别零型文法和一型文法的自动机，当然有了，对于前者就是大名鼎鼎的图灵机，后者则是图灵机的改造，让我们先从图灵机形式化开始吧。

定义 4.10 (图灵机)

图灵机由 7 部分组成 $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ ，其中

- (1) Q 是状态集
- (2) Σ 是输入字母表
- (3) Γ 是带字母表，且 $\Sigma \subsetneq \Gamma$ ，即至少有一个空白字符 $\sqcup \in \Gamma - \Sigma$
- (4) $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ 称为转移函数
- (5) $q_0 \in Q$ 称为起始状态
- (6) $q_{accept} \in Q$ 称为接受状态
- (7) $q_{reject} \in Q$ 称为拒绝状态，且 $q_{reject} \neq q_{accept}$



下一步自然是考察图灵机的识别问题了，图灵机 $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ 是一种运行机器， δ 可以视为它的程序，而它必需在内存上运行，即有一个带子 $R = (\dots, a_0, a_1, a_2, \dots, a_n, \dots) \in \prod_{i=-\infty}^{+\infty} \Gamma$ ，即左右有序无限延伸的标记了字母的带子，来同时作为输入和无限内存，然后就是程序的运行过程，首先图灵机指向带子的 a_0 位置并且处于状态 q_0 ，接着运行程序，对于每一个

$$\delta : (q_1, a_1) \mapsto (q_2, a_2, L)$$

表示“处于状态 q_1 且指向 a_1 时，将状态变成 q_2 、将带子上的 a_1 改成 a_2 并把指针向左移动一位”，类似地， $\delta : (q_1, a_1) \mapsto (q_2, a_2, R)$ 表示“处于状态 q_1 且指向 a_1 时，将状态变成 q_2 、将带子上的 a_1 改成 a_2 并把指针向右移动一位”。直到 q_{accept} 或 q_{reject} 时，图灵机停止运算，因此图灵机的输出结果是带子改变后的结果 $R = (\dots, a'_0, a'_1, a'_2, \dots, a'_n, \dots)$ 和一个停机状态，当然也有可能不停止永久运行下去。从图灵机对带子的改变可知，其具有计算能力，从图灵机停机的两种状态可知，其具有逻辑能力，因此我们把图灵机视为图灵机视为最强大的计算模型和逻辑模型是不为过的。对于语言的识别问题，我们可以将其视为输出 q_{accept} 或 q_{reject} 的逻辑问题，此时我们有定义

定义 4.11 (识别)

对一图灵机 $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

- (1) 我们将 $(q, i, R), q \in Q, R = (\dots, a_0, a_1, a_2, \dots, a_n, \dots) \in \prod_{i=-\infty}^{+\infty} \Gamma$ 视为一个格局，其中 q 是图灵机状态、 i 是指针位置， R 是带子内容。此时， $\delta : (q_1, a) \mapsto (q_2, b, L)$ 表示下面的格局变化

$$(q_1, i, R_1 = (\dots, a_i = a, \dots)) \rightarrow (q_2, i - 1, R_2 = (\dots, a_i = b, \dots))$$

$\delta : (q_1, a) \mapsto (q_2, b, R)$ 表示下面的格局变化

$$(q_1, i, R_1 = (\dots, a_i = a, \dots)) \rightarrow (q_2, i + 1, R_2 = (\dots, a_i = b, \dots))$$

- (2) 对于字符串 $w = w_1 \dots w_n$ （对任意 $1 \leq i \leq n$ 有 $w_i \in \Sigma$ ），我们任取一个空白字符 \sqcup 。如果一个格局序列

C_0, C_1, \dots, C_k 使得

(i) $C_0 = (q_0, 1, R = (\dots, a_0, a_1, \dots, a_n, \dots)), \forall i \leq 0 \vee i > n, a_i = \sqcup, \forall 1 \leq i \leq n, a_i = w_i$

(ii) $C_{i-1} \mapsto C_i \in \delta$, 即每一步格局由图灵机的程序推进

(iii) $C_k = (q_{accept}, i, R)$ 是接受格局

我们就称 T 接受 w 。我们将所有被 T 接受的字符串构成的集合记为 $L(T) \subset \Sigma^*$, 并称 $L(T)$ 是机器 T 的语言, 或称 T 识别语言 $L(T)$



实际上, 除了空白字符 $\sqcup \in \Gamma - \Sigma$ 以外, 我们还可以取出更多的字母, 读者可以把它想象成一种程序运行中的临时符号的感觉。如果语言 $L \subset \Sigma^*$ 可以被某台图灵机识别, 我们就称其是图灵可识别的。通过对识别的稍微改造, 我们可以得到语言的判定问题

定义 4.12 (判定)

对一图灵机 $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ 和字符串 $w = w_1 \dots w_n$ (对任意 $1 \leq i \leq n$ 有 $w_i \in \Sigma$), 我们任取一个空白字符 \sqcup 。如果一个格局序列 C_0, C_1, \dots, C_k 使得

(i) $C_0 = (q_0, 1, R = (\dots, a_0, a_1, \dots, a_n, \dots)), \forall i \leq 0 \vee i > n, a_i = \sqcup, \forall 1 \leq i \leq n, a_i = w_i$

(ii) $C_{i-1} \mapsto C_i \in \delta$, 即每一步格局由图灵机的程序推进

(iii) $C_k = (q, i, R)$ 满足 $q \in \{q_{accept}, q_{reject}\}$

我们就称 T 判定 w 。我们将所有被 T 判定的字符串构成的集合记为 $L^*(T) \subset \Sigma^*$, 并称 T 判定语言 $L^*(T)$



如果语言 $L \subset \Sigma^*$ 可以被某台图灵机判定, 我们就称其是图灵可判定的或简称为可判定的。判定无非就是在输出时多了一种“接受”或“拒绝”的状态, 但读者需要注意这意味着给出了更多的信息, 能使用的图灵机是变少的, 也就是说可判定的语言是图灵可识别的, 但反之不一定, 实际上确实可以找到一种语言是图灵可识别的却是不可判定的。图灵机的例子, 读者自己看着办, 我们先把其与文法的联系给出来

定理 4.7

一个语言由零型文法生成当且仅当存在一个图灵机识别它。



图灵机也有非确定型, 但可以证明它的识别能力和确定型图灵机是一样的, 所以就不做过多讨论了, 因为大家都喜欢确定的东西。接着, 我们来讲讲图灵机的变形, 通常我们有两种方向, 一种是限制图灵机的能力, 另一种是扩展图灵机的能力。我们先来讲一讲前一种, 首先我们可以限制输入带子为单向无限的 $(q, i > 0, R = (a_1, a_2, \dots, a_n, \dots)) \in \prod_{n=1}^{\infty} \Gamma$, 此时程序需要自行保证不会越过边界, 看起来能力似乎变弱了, 但实际上

定理 4.8

单向无穷图灵机等价于标准图灵机。



这里的等价有多重含义, 包括“由输入计算输出”、“识别语言”和“判定语言”等。其实“无穷”和“无穷”的计算能力一样, 并不值得惊讶, 如果我们进一步限制带子为有限的 $(q, 1 \leq i \leq n, R = (a_1, a_2, \dots, a_n)) \in \prod_{k=1}^n \Gamma$, 那么从“无穷”到“有穷”, 能力确实会下降, 我们把这个图灵机称为有界图灵机, 并且有

定理 4.9

一个语言由一型文法生成当且仅当存在一个非确定型有界图灵机识别它。



如果我们从另外一种角度进行限制, 比如两个字母 $\Sigma = \{0, 1\}$ 或者两种状态 $Q = \{q_0, q_1\}$, 结果会如何呢? 此时, 我们有

定理 4.10 (Shannon, 香农)

- (1) 只有两个字母的图灵机等价于标准图灵机。
 (2) 只有两个状态的图灵机等价于标准图灵机。



这其实并不值得惊讶，因为一旦我们限制了字母就可以让状态变多，而如果我们限制了状态就可以让字母变多，在无限内存的状态下是完全可以实现的。值得惊讶的是后者一种情况，我们发现，不论怎么增加图灵机的能力也不会有性能上的提升，例如我们给图灵机加一个不移动指针的操作

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

又或者我们让一条带子上有多个通道

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}$$

又或者我们让多条带子同时进行

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

又或者让它变成非确定型

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

图灵机的能力都不会发生改变，证明方法基本就是构造转化，有兴趣的读者可以自行看原书研究。图灵机还等价对应一种枚举器，它其实就是带打印机的图灵机，即我们增加了一个是否要把带上内容打印出来的选项

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} \times \{Y, N\}$$

其结果多了一个 $L_e(M) \subset \Sigma^*$ ，通常我们把它称为可枚举语言，可以证明可枚举语言和图灵可识别语言是等价的。在之前的内容中，我们曾稍微用过理论 $T = \text{Th}(\Gamma)$ 是可计算公理化的概念，即有一个公理集 Γ 是可计算的，现在我们来其严格化，取字母集 $\Sigma = \{0, 1, \wedge\}$ 和相应的二进制自然数 $\mathbb{N} = \{0\} \cup 1\Sigma^* \subset \Sigma^*$ ，对于一个函数 $f : \mathbb{N}^n \rightarrow \mathbb{N}$ ，我们将 $(x_1, \dots, x_n) \in \mathbb{N}^n$ 转为带上输入 $x_1 \wedge \dots \wedge x_n$ ，如果存在一个图灵机使得，对于有定义的输入 (x_1, \dots, x_n) 图灵机得到接受状态并在带上输出函数值，对于无定义的输入 (x_1, \dots, x_n) 图灵机得到拒绝状态，我们就称 f 是(图灵)可计算的。由哥德尔数给出了公式集到自然数的单射 $\Gamma \rightarrow \mathbb{N}$ ，此时我们将数集 $\#\Gamma = \{\#A \mid A \in \Gamma\} \subset \mathbb{N}$ ，如果函数 $f(x) = 1, x \in \#\Gamma; f(x) = 0, x \notin \#\Gamma$ 是可计算的，我们就称 Γ 是可计算的，这样我们的概念就完全了。

判定性

语言的识别问题我们已经讨论得差不多了，接下来我们聚焦于图灵机的逻辑能力 (q_{accept}, q_{reject}) ，即判定问题。首先是基础的归属问题

定理 4.11

由一型文法生成的语言是可判定语言。



同样我们可以找到可判定却不能由二型文法生成的语言，至此我们有下面的关系

$$\text{正则语言} (\Leftrightarrow \text{三型文法}) \Rightarrow \text{二型文法} \Rightarrow \text{一型文法} \Rightarrow \text{可判定语言} \Rightarrow \text{可识别语言} (\Leftrightarrow \text{零型文法})$$

定理 4.12

- (1) 有界图灵机的字符串识别问题、空语言问题和语言相等问题都是可判定的
- (2) 一阶谓词逻辑的恒真问题是不可判定的
- (3) 图灵机的停机问题是不可判定的



对于“XXX 问题”是不是可判定的该怎么理解呢？“可判定性”是很好搞定的，对于输入问题，你只需设计一个用于判定这个问题的图灵机，来解决问题即可，比如塔斯基证明的实数一阶理论是可判定的。难点在于“证明不可判定”这件事上，首先我们必需将待判定的问题转换成一个语言，接着我需要借助语言的相关判定理论来证明结论，后者是正常的数学问题，关键在于前一步。转换其实并不困难，最简单的办法就是直接用自然语言，比如一阶谓词逻辑的恒真问题，转化成语言 \mathcal{L}_1 ，图灵机则对每一个公式通过“接受”和“拒绝”来表示公式是有效的还是非有效的，更自然比如图灵机的停机问题，由于图灵机可以进行数学的形式化定义，那么它自然就可形成一个字符串，再并上输入带形成一个用于表示可运行图灵机的字符串，把这些字符串总结起来就变成了一个语言，它的字母表是“英语 \cup 汉语 \cup 数学符号”，此时图灵机对每一个输入的图灵机通过“接受”和“拒绝”来表示图灵机是会停止还是会停止。详细的证明，大家可以去看原书，我们给一个可判定的充要条件

定理 4.13

一个语言 $L \subset \Sigma^*$ 是可判定的当且仅当， L 和 $\Sigma^* - L$ 都是图灵可识别的。



我们该如何理解一阶逻辑是不可判定的呢？其实很好理解，一阶逻辑通过哥德尔数与递归函数产生了等价联系，通过后面的递归理论可知递归函数又可以与图灵可计算产生联系，图灵可计算包含了图灵机最大的能力，其对应语言的可识别性，而可识别性是大于可判定性的，所以我们有理由相信一阶谓词逻辑的恒真问题是不可判定的。

递归理论

对于图灵机而言，“判定”和“计算”是两件事情，判定只用到了图灵机的逻辑能力 (q_{accept}, q_{reject})，而计算还进一步使用了图灵机的输出带，因此可计算或递归可枚举等都是比可判定更弱的条件，因而包括的范围更大。

定义 4.13

- (1) 对于函数 $f : \Sigma^* \rightarrow \Sigma^*$ ，如果存在一个图灵机 M 使得，对于每一个输入 $w \in \Sigma^*$ ， M 停机且输出为 $f(w)$ ，我们就称 f 是可计算函数
- (2) 对于语言 A, B ，如果存在可计算函数 $f : \Sigma^* \rightarrow \Sigma^*$ 使得

$$w \in A \Leftrightarrow f(w) \in B$$

就称 A 可以映射归约到 B ，并记为 $A \leq_m B$



借助计算和映射归约的理论，我们可以轻松地得到一个判定性定理

定理 4.14

- (1) 如果 $A \leq_m B$ 且 B 是可判定的，则 A 是可判定的。
- (2) 如果 $A \leq_m B$ 且 A 是不可判定的，则 B 是可判定的。



我们接下来的目的是探究，什么样的函数是可计算的？可计算函数是什么样的？简单来讲其实就是我们之前讲的递归函数，但其只限于数论函数 $f : \mathbb{N} \rightarrow \mathbb{N}$ ，我们目的是将其推广到一般的计算函数理论上。首先，我们要知道图灵机可以打印可计算函数

定理 4.15 (递归定理)

对任意可计算函数 $t : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$, 存在一个计算函数 $r : \Sigma^* \rightarrow \Sigma^*$ 的图灵机 R 使得, 对任意 w 有

$$r(w) = t(\langle R \rangle, w)$$



此处的 $\langle R \rangle$ 表示图灵机在语言 Σ^* 中的描述, 换言之, 我们只需要在 t 中描述了图灵机 R 就可以在 t 中像 r 那样计算。利用这种思想, 由于我们通常假定 Σ 是有限的, 因此 Σ^* 实际是可数的, 所以我们自然可以进行一一对应 $\Sigma^* \rightarrow \mathbb{N}$, 这意味着可计算函数 $f : \Sigma^* \rightarrow \Sigma^*$ 可以等价为一个数论函数, 此时如果这个等价函数是递归的, 我们就称 f 是递归的, 此时我们有核心定理

定理 4.16

$f : \Sigma^* \rightarrow \Sigma^*$ 是可计算函数当且仅当 f 是递归的。



实际上, 对于递归函数, 还有一种等价的 λ 演算 (lambda calculus) 系统, 但我个人觉得没必要就不讨论了。最终, 我们可以得到多套系统的互相等价

$$\text{零型文法} \Leftrightarrow \text{图灵机} \Leftrightarrow \text{递归函数} \Leftrightarrow \text{一阶逻辑}$$

4.2 经典问题

希尔伯特第十问

对于计算的基本理论, 我们其实已经讲得差不多了, 所以我们就此来讨论一些经典问题。

定理 4.17

不存在统一的算法来判定所有多项式不定方程是否存在整数解。



对于这个定理的证明, 大家就别看开头参考书了, 去看这一本 [3], 又新又比较有数学味。这个问题其实是一个图灵机判定问题, 所以关键在于如何把它合理地转换成语言的判定问题, 前面我们一直说自然语言, 只是想要读者对这些判定和计算理论有个形象的理解, 实际转换的时候是不能这么直接的, 因为自然语言转化在数学严谨证明时不具有任何便利性, 接下来我将把这个转化过程详细地讲给你。

定义 4.14 (丢番图)

(1) 对于集合 $S \subset \mathbb{N}^n$, 如果存在整系数多项式 $P(x_1, \dots, x_n, y_1, \dots, y_m) \in \mathbb{Z}[x_1, \dots, x_n, y_1, \dots, y_m]$, $m \geq 0$ 使得

$$(x_1, \dots, x_n) \in S \Leftrightarrow \exists (y_1, \dots, y_m) \in \mathbb{N}^m, P(x_1, \dots, x_n, y_1, \dots, y_m) = 0$$

我们就称 S 是丢番图集

(2) 对于函数 $f : \mathbb{N}^n \rightarrow \mathbb{N}$, 如果

$$\{(x_1, \dots, x_n, y) \in \mathbb{N}^{n+1} \mid y = f(x_1, \dots, x_n)\}$$

是丢番图集, 就称 f 是丢番图函数



这里我们没有使用整数解 \mathbb{Z} , 而是自然数解 \mathbb{N} , 一方面它们都是可数无穷的, 在本质上是一致的, 另一方面, 我们有四平方和定理 “每个自然数均可表示为 4 个整数的平方和”, 此时我们可以得到

$$p(x_1, \dots, x_n) = 0 \text{ 有自然数解} \Leftrightarrow p(s_1^2 + t_1^2 + u_1^2 + v_1^2, \dots, s_n^2 + t_n^2 + u_n^2 + v_n^2) = 0 \text{ 有整数解}, x_i = s_i^2 + t_i^2 + u_i^2 + v_i^2$$

这意味着, 我们只需说明自然数解问题的不可判定, 就能说明整数解的一个子类不可判定, 从而得到整体的不可判定, 值得注意的是, 如果问题是可判定的就不能这么进行。丢番图集的概念其实很好理解, 就是指它能

否是一个整系数方程的自然数解，或部分自然数解，下面是几个例子

- (1) 偶数: $\{0, 2, 4, 6, \dots\}$, $f(x, y) = x - 2y$
- (2) 合数: $\{4, 6, 8, 9, 10, \dots\}$, $f(x, y_1, y_2) = x - (y_1 + 2)(y_2 + 2)$
- (3) 斐波那契数列: $\{1, 2, 3, 5, 8, \dots\}$ (集合没有重复元，故删除重复的 1), $f(x, y) = x^2 - xy - y^2 - 1$
- (4) 偶数项斐波那契数列: $\{1, 3, 8, 21, \dots\}$, $f(x, y) = 5x^2 + 4 - y^2$

对于丢番图集有一个看起来很惊讶的性质

性质 [Putnam] $S \subset \mathbb{N}$ 是丢番图集当且仅当，存在多项式 $P(x_1, \dots, x_n) \in \mathbb{Z}[x_1, \dots, x_n]$ 使得 $S = P(\mathbb{N}^n)$ ，即 S 是某个多项式的非负值域。

所以转而研究丢番图函数，对于概念没啥好说的，我们主要考察，哪些函数是丢番图函数，丢番图函数又有哪些性质。下面是一些简单的例子

性质 (1)[配对函数定理] 存在丢番图函数 $P(x, y), L(z), R(z)$ 使得, $\forall x, y, L(P(x, y)) = x, R(P(x, y)) = y, \forall z, P(L(z), R(z)) = z, L(z) \leq z, R(z) \leq z$

(2)[序列数定理] 存在丢番图函数 $S(i, u)$ 使得, $S(i, u) \leq u, \forall (a_1, \dots, a_N) \exists u, S(i, u) = a_i, 1 \leq i \leq N$

(3)[幂函数] $h(b, n) = (b+1)^n$ 是丢番图函数(利用 Pell 方程解上带的幂来实现构造证明)

(4)[组合数] $f(a, b) = \binom{a}{b}$ 是丢番图函数

(5)[阶乘] $f(a) = a!$ 是丢番图函数

(6)[和积] $h(a, b, y) = \prod_{k=1}^y (a + bk)$ 是丢番图函数

观察发现，它们似乎都具备递归定义的潜质，实际上，我们可以借助一阶逻辑的相关理论证明(实际过程很多，请自行看书理解)

定理 4.18

函数 $f: \mathbb{N}^n \rightarrow \mathbb{N}$ 是丢番图的当且仅当它是递归的。



我们竟然收回了前面的伏笔，属实令人惊讶。实际上，一旦我们得到了这个命题，问题就已经结束了，此时我们有下面的对应链

f 是丢番图的 $\Leftrightarrow f$ 是递归的 $\Leftrightarrow f$ 是可计算的

此时我们只需使用逻辑证明中最爱玩的自指命题，即可得到一个非丢番图数集，从而说明“希尔伯特第十问是不可解的”，具体做法是这样的。对于多项式的变量 $x_0, x_1, \dots, x_n, \dots$ 我们递归定义下面的多项式

$$P_0 = 1, P_{3i+1} = x_i, P_{3i+2} = P_{L(i)} + P_{R(i)}, P_{3i+3} = P_{L(i)}P_{R(i)}$$

由多项式的定义可知，我们枚举出了所有的多项式。然后我们枚举出所有的自然数的丢番图集

$$D_n = \{x_0 \mid \exists x_1, \dots, x_n, P_{L(n)}(x_0, x_1, \dots, x_n) = P_{R(n)}(x_0, \dots, x_n)\}$$

此时我们可以证明下面的通用性定理

定理 4.19 (Universality theorem)

$\{(n, x) \in \mathbb{N}^2 \mid x \in D_n\}$ 是丢番图集



此时我们通过逻辑学的对角线原理可以得到

定理 4.20 (非丢番图集的存在性)

$V = \{i \in \mathbb{N} \mid i \notin D_n\}$ 是非丢番图集



这是一个明显的自我否定，即我说我自己不是丢番图的，如果 $\exists k, V = D_k$ 则有

$$k \in D_k \Leftrightarrow i \in V \Leftrightarrow k \notin D_k$$

矛盾，即确实有一个非丢番图集。然后就能定义一个非递归函数了

$$g(n, x) = \begin{cases} 0 & x \notin D_n \\ 1 & x \in D_n \end{cases}$$

此时由于 $\{(x, n) \mid x \in D_n\}$ 是丢番图集，故存在多项式 $P(n, x, y_1, y_2, \dots, y_k)$ 使得

$$x \in D_n \Leftrightarrow \exists z_1, \dots, z_k, P(n, x, z_1, z_2, \dots, z_k) = 0$$

另一方面，如果希尔伯特第十问是可解的，那么我们有算法对于给定的 (n, x) 可以判定

$$P(n, x, z_1, z_2, \dots, z_k) = 0$$

是否有整数解。即对于给定 (n, x) ，可以判定 $x \in D_n$ 还是 $x \notin D_n$ ，这意味着 $g(n, x)$ 是可计算的，从而是递归的，与之前的结论矛盾了，因此希尔伯特第十问是不可解的。至此本部分内容结束了，更多的东西就给读者自行探索了，比如说明素数集是丢番图集等。

N 对 NP 问题

对于计算这件事，单纯讨论能不能计算是完全不够的，因为在现实中，我们还可能存在时间和空间的限制，因此算法复杂度或者图灵机的计算复杂度就是本部分的核心内容了。

定义 4.15 (时间复杂度)

对一图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ 和单词 $w \in \Sigma^*, |w| = n$ ，如果 M 对输入 w 可以停机，则把其运行的步数记为 $t_M(w) \in \mathbb{N}$ ，如果 M 对输入 w 不停机，则记 $t_M(w) = \infty$ ，我们进一步记

$$T_M(n) = \max\{t_M(w) \mid w \in \Sigma^*, |w| = n\}$$

并把函数 $T_M : \mathbb{N} \rightarrow \mathbb{N}$ 称为 M 的(最坏)运行时间，或者时间复杂度



在这里我们对 $t_M(w)$ 的计数就是图灵机执行了多少次转移函数 δ ，虽然这和传统编程对算法的观念有所不同，但后面我们会说明这两者其实是完全等价的，或者说在任何等价运行的变种图灵机上都存在复杂度的转化公式，对于表示复杂度的大 O 记号请自行去复习。

$$O(\ln n) < O(n) < O(n \ln n) < O(n^k) < O(k^n) < O(n!)$$

定义 4.16

设函数 $t : \mathbb{N} \rightarrow \mathbb{R}^+$ 和所有语言的集合 $\mathcal{L} = 2^{\Sigma^*}$ ，记语言子集

$$\text{TIME}(t(n)) = \{L(M) \in \mathcal{L} \mid T_M(n) = O(t(n))\}$$

把它称为时间复杂度为 $t(n)$ 的时间复杂性类。



一个简单的语言例子是 $A = \{0^k 1^k \mid k \geq 0\}$ ，则有 $A \in \text{TIME}(n \ln n)$ 。接着我们给出两个时间复杂度转化原理

定理 4.21

对于复杂度函数 $t(n), t(n) \geq n$

- (1) 每个 $t(n)$ 时间复杂度的多带图灵机都和某个 $O(t^2(n))$ 时间复杂度的标准图灵机等价。
- (2) 每个 $t(n)$ 时间复杂度的非确定型图灵机都和某个 $2^{O(t(n))}$ 时间复杂度的标准图灵机等价。



显然上面所说的图灵机在计算能力上，即能否计算的问题是一样的，但在时间复杂度上，即计算的时间，是不同的，这其实很好理解，这两种变形都相当于在用空间来换时间，这和传统的程序开发理念是一致的。接着我们来探讨有关时间复杂度的 N 对 NP 问题 (The P versus NP problem)，[这里](#)是官方的描述。

定义 4.17 (P and NP)

- (1) 我们记 P 类问题为

$$\mathbf{P} = \bigcup_{k=1}^{\infty} \text{TIME}(n^k)$$

即确定型单带图灵机在多项式时间内可识别的语言类。

- (2) 我们记 NP 类问题为

$$\mathbf{NP} = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k)$$

即非确定型单带图灵机在多项式时间内可识别的语言类。



由前面的结论可知 $\mathbf{P} \subset \mathbf{NP}$ ，于是我们的核心猜想为

命题 4.1 (N 对 NP 猜想)

$$\mathbf{P} = \mathbf{NP}?$$



直接解决这个猜想是困难的，我们有一个 NP 完全理论可以极大地简化问题。

定义 4.18

- (1) 对于语言 $A, B \in \mathcal{L}$ ，如果存在一个多项式时间内可计算的函数 $f : \Sigma^* \rightarrow \Sigma^*$ （即存在多项式时间复杂度的图灵机来计算这个函数）使得对每个 $w \in \Sigma^*$ 有

$$w \in A \Leftrightarrow f(w) \in B$$

我们就称 A 在多项式时间内可归约到 B ，并记为 $A \leq_P B$

- (2) 如果语言 $B \in \mathcal{L}$ 满足 $B \in \mathbf{NP}$ 且 $\forall A \in \mathbf{NP}, A \leq_P B$ ，就称 B 是 NP 完全的，我们把这类语言的集合记为 \mathbf{NPC}



接着 NP 完全的定义，我们有下面的核心定理

定理 4.22

- (1) 若 $A \leq_P B, B \in \mathbf{P}$ ，则 $A \in \mathbf{P}$
- (2) 若 B 是 NP 完全的且 $B \in \mathbf{P}$ ，则 $\mathbf{P} = \mathbf{NP}$
- (3) 若 B 是 NP 完全的且 $B \leq_P C, C \in \mathbf{NP}$ ，则 C 是 NP 完全的



这些东西的证明其实并不困难，关键在于 NP 完全的问题是否存在？一旦确实存在，我们的理论才能开始，也正因如此，我们才相信 $\mathbf{P} = \mathbf{NP}$ 是有可能的。

定理 4.23

- (1) 可满足性问题 $SAT = \{\varphi \mid \varphi \text{ 是可满足的}\}$ 是 NP 完全的
- (2) 分团问题是 NP 完全的
- (3) 顶点覆盖问题是 NP 完全的

- (4) 哈密顿路径问题是 NP 完全的
 (5) 子集问题是 NP 完全的



要想证明 $\mathbf{P} = \mathbf{NP}$, 我们只需“证明任一 NP 完全问题是 P 类问题即可”。但如果要证明 $\mathbf{P} \neq \mathbf{NP}$, 就比较困难了, 通常都是些数理逻辑的方法, 比如约化对角化, 即用于证明不可判定性的一般性理论, 比如停机问题的证明, 又或者是布尔电路下界问题等, 只能说这些比较未来可期。又或者, 我们可以考察其它的复杂度类来进行迫敛包含, 我们引入空间复杂性的概念

定义 4.19 (空间复杂度)

对一图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ 和单词 $w \in \Sigma^*, |w| = n$, 如果 M 对输入 w 可以停机, 则把其运行所经过的带上格数记为 $s_M(w) \in \mathbb{N}$, 如果 M 对输入 w 不停机, 则记 $s_M(w) = \infty$, 我们进一步记

$$S_M(n) = \max\{s_M(w) \mid w \in \Sigma^*, |w| = n\}$$

并把函数 $S_M : \mathbb{N} \rightarrow \mathbb{N}$ 称为 M 的(最多)使用空间, 或者空间复杂度



定义 4.20

设函数 $t : \mathbb{N} \rightarrow \mathbb{R}^+$ 和所有语言的集合 $\mathcal{L} = 2^{\Sigma^*}$, 记语言子集

$$\text{SPACE}(t(n)) = \{L(M) \in \mathcal{L} \mid S_M(n) = O(t(n))\}$$

把它称为时间复杂度为 $t(n)$ 的空间复杂性类。如果把图灵机换成非确定型, 则记 $\text{NSPACE}(t(n))$.



虽然在时间复杂度上, 确定型和非确定型图灵机很难有反向包含关系, 但在空间复杂度上却是存在的

定理 4.24

- (1) 对任意函数 $f(n) \geq n$ 有, $\text{NSPACE}(f(n)) \subset \text{SPACE}(f^2(n))$
 (2) 特别地, 如果我们记 $\text{PSPACE} = \bigcup_{k=1}^{\infty} \text{SPACE}(n^k)$ 和 $\text{NPSPACE} = \bigcup_{k=1}^{\infty} \text{NSPACE}(n^k)$, 则有

$$\text{PSPACE} = \text{NPSPACE}$$



感觉有点惊奇, 又感觉没那么惊奇, 我们使用 EXP 作为指数的前缀, LOG 作为对数的前缀, 由基本的计算量理论和上面的结论可知, 到目前有下面的包含关系

$$\text{LOGSPACE} \subset \text{NLOGSPACE} \subset \mathbf{P} \subset \mathbf{NP} \subset \text{PSPACE} = \text{NPSPACE} \subset \text{EXPTIME}$$

完全性是可以推广到各类语言中的, 比如形成 \mathbf{P} 空间完全类 PCSPACE、NLOG 空间完全类 NLOGCSPACE 等等, 但将太多太杂的东西没啥意义, 还是来考虑 $\mathbf{P} \neq \mathbf{NP}$ 吧, 我们有

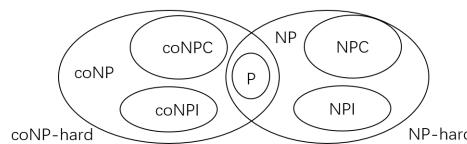
定理 4.25

使用 LINEAR-SIZE 表示可以由“只有两个字母 {0, 1} 且空间复杂度为 $O(n)$ 的图灵机”生成的语言类。若 $\mathbf{P} \subset \text{LINEAR-SIZE}$, 则 $\mathbf{P} \neq \mathbf{NP}$ 。



我们引入非 NP 完备类 $\mathbf{NPI} = \mathbf{NP} - (\mathbf{P} \cup \mathbf{NPC})$, 并使用 co 作为前缀来表示补语言, 则有

$$\text{NLOGSPACE} = \text{coNLOGSPACE}, \text{coP} = \mathbf{P}, \text{coNP} \neq \mathbf{NP} \Rightarrow \mathbf{P} \neq \mathbf{NP}$$



最外层的 NP-hard 已经可以不用理会了，它已经难过头了。图灵机还能改造出很多拥有不同复杂度的变种，我们来介绍一个对 NP-P 问题解决有利的系统。

定义 4.21 (概率图灵机)

(1) 概率图灵机 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ 是一种非确定型图灵机，对于一次运行 $\delta : (q_0, a_0) \mapsto \{(q_1, a_1, \pm 1), \dots, (q_n, a_n, \pm n)\}$ ，它进入每一个分支 $1 \leq i \leq n$ 的概率为

$$P(i) = \frac{1}{i}$$

(2) 此时对于输入 $w \in \Sigma^*$ ，概率图灵机 M 将以一定概率接受或拒绝它。对于语言 $A \in \mathcal{L}$ 和正数 $0 \leq \varepsilon < \frac{1}{2}$ ，如果任意的 $w \in \Sigma^*$ 有

$$w \in A \Rightarrow P(M \text{ 接受 } w) \geq 1 - \varepsilon; w \notin A \Rightarrow P(M \text{ 拒绝 } w) \geq 1 - \varepsilon$$

则称 M 以错误概率 ε 识别语言 A



大家可能对概率图灵机的识别有些不太理解，我们来回顾一下非确定型图灵机，它与确定型图灵机的区别只有一个，就是转移函数有多种选择

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

那么对于一个确定的输入，其运行的结果其实是不唯一的，在定义识别问题时，我们只需要求这些分支之中有一个给出接受，就称这个输入可以被识别。在没有引入概率的时候，对于输入 w 的运行时间 $t_M(w)$ 表示所有被识别分支中的最小时间，放到现实运行的计算机上就相当于，每运行到一次分支时，图灵机就分裂成相应分支个，然后再一并运行，直到有一个分裂后的图灵机成功识别了输入，就全部停止并记录时间，分裂过程相当于指数运算，也因此有了我们之前的 N 和 NP 之间的关系。在这种理念下，概率图灵机是一种结果唯一，但每次结果可能不同的图灵机，我们要求每次遇到分支时，都以均等的概率进入每一个分支，因此对于每一个结果都对应着一个概率值，此时我们要求图灵机在每一个分支都会停机，这样才能有概率关系

$$P(M \text{ 接受 } w) + P(M \text{ 拒绝 } w) = 1$$

那么对于输入 w ，概率图灵机的运行时间 $t_M(w)$ 该如何定义呢？其实我们就直接使用非确定型图灵机的运行时间即可。我们使用 BPP 表示多项式时间的概率图灵机以错误概率 $\frac{1}{3}$ 识别的语言类。我们给出下面的命题

命题 4.2 (命题 A)

存在语言 $L \in \text{EXPTIME}$ 和 $\varepsilon > 0$ ，使得对所有能识别 L 的电路族 $\langle B_n \rangle$ 和充分大的 n ， B_n 至少有 $2^{\varepsilon n}$ 个门电路。



此时我们有相应的判定定理

定理 4.26

如果 A 是真命题，则 $\text{BPP} = \text{P}$ 。如果 A 是假命题，则 $\text{P} \neq \text{NP}$ 。



这就是我们之前说的布尔电路下界问题。你问我布尔电路和电路簇是什么？其实就是通常电路的意思，它由输入端、导线和“与、或、非”门构成，并带有一个输出端，你可以直接把它理解成一个布尔函数 $F : \{0, 1\}^n \rightarrow \{0, 1\}$ ，而电路簇 $\langle B_n \rangle$ 指可数个电路或布尔函数，对于每个 B_n 有 n 个输入端或对应 n 元布尔函数，对于语言 $A \subset \Sigma^*$ 都存在一种二进制转化使得 $A \subset \{0, 1\}^*$ ，此时如果有

$$w \in A \Leftrightarrow |w| = n, B_n(w) = 1$$

我们就称电路簇 $\langle B_n \rangle$ 识别语言 A ，这样我们就把概念给理清了，所有的内容也就结束了。

第五章 数学基础其一：公理集合论

我们知道，大部分的数学理论都可以通过一阶逻辑，建立相应的一阶理论，稍微有表现力的像微积分就需要二阶逻辑，但这样的形式化是十分浪费逻辑资源的，我们的基本想法是，找到一个通用的基础理论来建立一阶逻辑形式化理论，而其它所有的理论都通过定义建立在其上，这就是数学基础所考虑的问题，即使用哪一个一阶理论作为数学大厦的基础。最原始的想法是，使用皮亚诺公理 **PA** 得到的初等数论系统 $(\mathbb{N}, 0, S, +, \times)$ ，作为一个不完全理论，其确实可以得到几乎所有扩张链 $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C} \subset \mathbb{H}$ 上的理论，包括微积分、实变函数、泛函分析等。但其不具备概念所应有的“概括性”，对于像群环域、线性空间、代数、拓扑、流形等特性描述对象，是无法定义出来的，因此我们有了集合论，而它可以定义出初等数论所需要的序数，且具有极强的概括能力，可以用来表示特性描述对象，因此就成了当代数学的基础理论。

5.1 ZFC 公理

你可能会觉得，集合论有什么好说的，基本每一本数学书的开头都是集合论，而且我们在描述逻辑的时候也都在用包含属于的符号，它几乎无处不在。但我必需告诉你，那不叫集合论，只能叫符号简记，因为我们没有触及到集合论的核心目的“研究无穷”。更会莫名其妙地产生可数 \aleph_0 和连续统 2^{\aleph_0} 之间是否存在其它基数 $\aleph_0 < \aleph_1 < 2^{\aleph_0}$ 的问题。为此，我们将从完全公理化的角度来研究集合论 ZFC，并在下一节中说明连续统假设是独立于 ZFC 公理的一条公式。集合论大家都太熟了，所以我们直接从公理开始研究，我们使用一阶带等号逻辑 $L = (\mathcal{L}_1, =)$ 作为基底，来生成 ZFC 理论。

定义 5.1 (ZFC)

在 L 的基础上引入一个谓词 \in 就足够了，此时 $ZFC = (L, \in)$ ，并有如下公理

(ZF1) 外延公理: $\forall x \forall y (\forall z (z \in x \leftrightarrow z \in y) \rightarrow x = y)$

(ZF2) 空集合存在公理: $\exists x \forall y (\neg y \in x)$

(ZF3) 有序对集合存在公理: $\forall x \forall y \exists z \forall u (u \in z \leftrightarrow u = x \vee u = y)$

(ZF4) 并集存在公理: $\forall x \exists y \forall z (z \in y \leftrightarrow \exists u (u \in x \wedge z \in u))$

(ZF5) 幂集合存在公理: $\forall x \exists y \forall z (z \in y \leftrightarrow \forall u (u \in z \rightarrow u \in x))$

(ZF6) 正则公理: $\forall x (\exists y (y \in x) \rightarrow \exists y (y \in x \wedge \forall z (z \in y \rightarrow \neg z \in x)))$

(ZF7) 无穷公理: $\exists x (\exists y (y \in x) \wedge \forall z (z \in x \rightarrow \exists t (t \in x \wedge (s \in t \leftrightarrow s = z \vee s \in z))))$

(ZF8) 替换公理模式: $\forall t (\forall x \in t \exists y A(x, y) \rightarrow \exists S \forall x \in t \exists y \in S A(x, y))$

公式缩略^a一: $\forall x \in y A := \forall x (x \in y \rightarrow A)$

公式缩略二: $\exists x \in y A := \exists x (x \in y \wedge A)$

(AC) 选择公理: $\forall x (\exists y (y \in x) \rightarrow \exists f^1 \forall y (y \in x \wedge \exists z (z \in y) \rightarrow f^1(y) \in y))$

公式缩略^b三: $\exists f^1 A := \exists f (\forall x \in f (\exists y \exists z (x = (y, z)) \wedge \forall y \in \text{dom } f \exists_1 z ((y, z) \in f)) \wedge A)$

公式缩略四: $\exists_1 y A(y) := \exists y (A(y) \wedge \forall z (A(z) \rightarrow z = y))$

项缩略一: $(y, z) := (u \in (y, z) \leftrightarrow (u = y \vee (t \in u \leftrightarrow (t = y \vee z))))$

项缩略二: $\text{dom } f := (u \in \text{dom } f \leftrightarrow \exists z ((y, z) \in f))$

^a为了避免使用二阶语言，我们将量词后加谓词视为某类公式的缩写

^b同样为了避免量词后加函数需要进行一些处理，此处函数非一阶逻辑中的函数，而要将其视为集合论中定义的函数，即二元关系的子集



由于一阶理论的一致性是无法系统内证明的，所以对于公理系统我们能做的无非就是验证各条公理之间的独立性，最基本的办法就是寻找模型，即说明添加公理以后依旧存在模型，而 ZFC 理论的一个模型就是我们通常所讨论的**朴素集合论**。读者需要知道这样一件事，建立一阶理论的目的无非就是说明一些逻辑问题，比如“XXX”

能不能推出 XXX”、“XXX 能不能在这个理论中被证明”、“XXX 能不能不能定义”等等，比如下面这个定理

定理 5.1 (选择公理)

- (1)[相容性] $\{\text{ZF1}, \dots, \text{ZF8}\} \not\vdash_L \text{AC} \vee \neg\text{AC}$
- (2) 在 ZF 下，选择公理 \Leftrightarrow 良序定理 \Leftrightarrow 佐恩引理



虽说我们把公理集合论作为数学基础，但这并不意味着我们以后的研究都必需在 ZFC 的逻辑框架之下，而应该是在其模型下进行研究。考虑朴素集合论模型 $\text{NST} := (\text{Set}, =, \in)$ ，我们来引出集合中的一些核心内容，首先任何理论模型下的任何谓词都只是公式的缩写，不存在能否定义的问题，例如

$$x, y \in \text{Set}, x \subset y := \forall t(t \in x \rightarrow y)$$

$$x, y \in \text{Set}, x \subseteq y := \forall t(t \in x \rightarrow y) \wedge x \neq y$$

因此我们讨论定义问题就是在模型论中的定义问题，即 $A \subset \text{Set}^n$ 的定义问题，由于集合自身的性质，我们只需考虑 $A \subset \text{Set}$ 的定义问题。对于 $A \subset \text{Set}$ ，其能被定义意味着，存在 $b_1, b_2, \dots, b_n \in \text{Set}$ 和公式 $A(x, x_1, x_2, \dots, x_n)$ 使得

$$a \in A \Leftrightarrow A[a, b_1, b_2, \dots, b_n]$$

先看 $n = 0$ 的无依赖情况，比如空集 $\emptyset \in \text{Set}$ ，考察公理 ZF2 显然有 $\text{Fr}(\text{ZF2}) = \emptyset$ ，但我们可以分离出定理 $A(x) = \forall y(x \subset y) = \forall y(\forall t(t \in x \rightarrow y))$ ，此时我们有

$$t \in \{\emptyset\} \Leftrightarrow A[t]$$

因此我们可以定义出空集 \emptyset 。接着我们考虑依赖情况，并一个个公理考察，对于 $\{x, y\} \in \text{Set}$ ，考察公理 ZF3 我们可以分离出 $A(z, x, y) = \forall t(t \in z \leftrightarrow t = x \vee t = y)$ ，从而有

$$t \in \{x, y\} \Leftrightarrow A[t, x, y]$$

实际上如果在理论 ZFC 有一条 $\forall x_1 \dots \forall x_n \exists y A$ 型的定理，我们都能定义让 y 依赖于 x_1, \dots, x_n 而在 Set 中定义出来。例如 ZF4 就表明了 $y = \cup_{t \in x} t$ 可以依赖 x 在 Set 中定义出来，接着我们利用 ZF3 和 ZF4， $z = x \cup y$ 可以依赖 x 在 Set 中定义出来，基本做法是我们先利用 ZF3 得到 $\{x, y\}$ ，再利用 ZF4 得到 $z = \cup_{t \in \{x, y\}} t$ 。之所以这么搞是因为利用 $x \cup y$ 是不能定义出 $\cup_{t \in x} t$ ，原因是十分简单的，因为 x 不一定是有限的，所以我们经常能看到依赖于指标集的并集定义法

$$\cup_{i \in I} A_i$$

接着看 ZF5，其表明了幂集 $y = 2^x$ 可以依赖 x 在 Set 中定义出来。ZF6 是用来消除罗素悖论的，它表示 $A = \{x \notin A\}$ 是不能定义的，ZF6 的通俗含义是

$$S \neq \emptyset, x \in S \Rightarrow S \cap x = \emptyset$$

罗素悖论的形式公式是

$$\exists x \forall t(t \in x \rightarrow \neg t \in x)$$

而我们可以在 ZFC 下证明

$$\neg(\exists x \forall t(t \in x \rightarrow \neg t \in x))$$

所以 ZF6 简单来讲就是，自指集合是不存在的。此处我们定义了交集 $z = x \cap y$ ，对此我们需要公式 $A(z, x, y) = \forall t(t \in z \leftrightarrow t \in x \wedge t \in y)$ ，并且可以得到

$$t \in z \Leftrightarrow A[z, x, y]$$

ZF7 用于表示可数基数的存在，对于非无穷的有限序数，我们可以以 ZF2 为基础并不断通过使用 ZF3 来叠加有限序数

$$0 = \emptyset, 1 = \{\emptyset\}, 2 = \{\emptyset, \{\emptyset\}\} \dots$$

在有限次运算且没有归纳法的情况下，我们只能引入公理来说明无限序数叠加的存在

$$|\mathbb{N}| = \{\emptyset, \{\emptyset, \{\emptyset, \dots\}\}\}$$

读者会有些其它的问题，为啥序数要这么定义，直接 $\{1, 2, 3, \dots\}$ 不行吗？当然不行，首先在公理化的朴素集合论下，我们必须将所有可定义的对象视为一个集合，因此序数 $1, 2, 3, \dots$ 等都必须代表一个集合，最好的办法就是利用有序对定义 $\{x, \{x, y\}\}$ 的方法，而这种依赖必需要有一个起始元素，而序数必需要从集合论自导出来才行，那么我们唯一能利用的只能是 ZF2 给出的空集合 \emptyset ，因此 ZF2+ZF3 共同给出了有限序数的定义，而 ZF7 给出了第一个无穷序数 $\aleph_0 = \{\emptyset, \{\emptyset, \{\emptyset, \dots\}\}\} \in \text{Set}$ 。最后的 ZF8 和 AC 就有点麻烦了，首先 ZF8 包含了无数条公理，因为谓词 $A(x, y)$ 是可以任意替换的，而我们并没有二阶语言 $\forall A$ ，所以它也被称为一个公理模式，实际可以证明 ZF8 不能被有限公理化，我们先来讲一下它的通俗含义，其告诉了我们一个很简单的事实，我们可以“用描述法定义集合”，其定义出的集合是 S ，依赖的集合是 t 和公式 $A(t, S)$ ，即下面的集合可以定义出来

$$S = \{x \in t \mid A(t, x)\}$$

此时你再仔细读读 ZF8 的含义，是不是就是如此。实际在集合描述上，ZF3+ZF4 告诉了我们可以用枚举法定义有限集，例如 $\{x_1, x_2, \dots, x_n\} \in \text{Set}$ 可以总结为以下的过程

$$x_1, x_2, \{x_1, x_2\}, x_3, \{x_3\}, \{\{x_1, x_2\}, \{x_3\}\}, \{x_1, x_2, x_3\}, x_4, \{x_4\}, \dots$$

这时你可能会疑惑单元素集合怎么叠加 $\{x\}$ ，实际上这利用了 ZF1，此时有 $\{x, x\} = \{x\}$ 。既然描述法这么强大，那前几个公理不都可以像下面这样描述出来吗？

$$\{x, y\} = \{t = x \vee t = y\}$$

$$x \cup y = \{t \in x \vee t \in y\}$$

$$2^x = \{t \subset x\}$$

首先第一种情况依赖了两个集合 x, y 并不符合 ZF8 的公理模式，在假设 $\{x, y\}$ 存在时，我们似乎可以只通过一个集合 $\{x, y\}$ 来描述并集

$$x \cup y = \{t \mid t \in x \in \{x, y\} \vee t \in y \in \{x, y\}\}$$

但问题是 t 从哪里来？ $t \in \{x, y\}$ 显然不行，但换成 $t \in \cup\{x, y\}$ 就又变成了 ZF4，实际上人家都已经证明了独立性，我们根本不用考虑这么多。不过借此我们可以给出交集补集等概念

$$x \cap y = \{t \in x \cup y \mid t \in x \wedge t \in y\}$$

$$x \subset y, y - x = \{t \in y \mid \neg t \in x\}$$

在第一个式子中，它看起来依赖于两个项 x, y ，但其实只依赖于一个 $x \cup y$ ，那公式 $A(t, x \cup y)$ 中出现了 x, y 项没有问题吗？当然没问题，因为它们并不是自由变量，其并不在 A 的考量之中。对于幂集合的定义也是显然不行的，因为它只给出了 $A(t, x)$ ，而没有 t 集合的来源集合，一旦回到了来源集合就又变成了 ZF5 公理。我们再来看最后的选择公理 AC，这条公理就比较特殊了，它的通俗含义为“一个非空集合簇中，我们可以从每个非空集合中选出一个元素”，由于此处涉及了映射的概念，我们先把它的含义给出来。我们先给出笛卡尔积的概念，它的基础是有序对 $(x, y) = \{x, \{x, y\}\}$ ，这需要第三种可形式定义集合的方法，根据前面的理论可知，我们需要一条 $\forall x \forall y \exists z A(z, x, y)$ 型的定理，就可以依赖 x 和 y 定义出 z ，因此我们在形式系统中只需证明下面的定理

$$\forall x \forall y \exists z (\forall u \forall v (u \in x \wedge v \in y \leftrightarrow \exists s (s \in z \wedge (t \in s \leftrightarrow (t = u \vee (u \in t \wedge v \in t))))))$$

从而可得到 $z = x \times y$ ，接着反复运用笛卡尔积就能得到任意有限笛卡尔积了，至于无限笛卡尔积目前我不知道怎么推出来，而且目前大部分的理论也没用到，就算是序列 (a_1, \dots, a_n, \dots) ，一般也是看成函数 $\mathbb{N} \rightarrow \{a_1, \dots, a_n, \dots\}$ ，而自然数集 \mathbb{N} 通过无穷公理得到，因此无穷笛卡尔积应该可以通过无穷公理来证明存在性，有兴趣的读者可以自行探索。一旦得到了笛卡尔积，映射 $f : x \rightarrow y$ 就可以直接给出来了

$$f = \{(a, b) \mid \forall a \in x \exists_1 b \in y ((a, b) \in x \times y)\}$$

这里我们用了符号缩略，比如直接把项用特征法 (a, b) 给写了出来，这样合理吗？当然合理了，因为 $t \in x \times y$ 是包含了一个定理 $\exists a \exists b (a \in x \wedge b \in y \wedge t = \{a, \{a, b\}\})$ 的，这是定义给出来的。只要一个集合 z 是可以被定义的，我们实际有

$$t \in z \Leftrightarrow A[t, x_1, \dots, x_n]$$

这里的 A 是定义公式， x_1, \dots, x_n 是依赖的元素。至于更简单的关系就不说了，最后我们来看选择函数，其相当于

$$f_x = \{(s, t) \mid \forall s \in x \exists_1 t \in \cup x (t \in s)\}$$

因此在 ZF 体系下，我们可以给出选择函数的定义，AC 并非说选择函数定义不出来，而是我们无法判定它是否存在。最通俗的例子就是，比如设 $x = \mathbb{R}$ ，我们可以给出定义

$$a < b, [b, a] = \{t \in \mathbb{R} \mid t \geq b \wedge t \leq a\}$$

但它并不是存在的，其存在性要我们通过逻辑推导来证明的，当然这个例子是可以证明的，但“AC 不能在 ZF 下证明”是已经被证明的。通过上面这么多的内容，我们差不多已经搞清了什么样的集合可以被定义出来

- (1) 有限枚举集： $\{x_1, x_2, \dots, x_n\}$
- (2) 公式描述集： $A_x = \{t \in x \mid A(t, x)\}$
- (3) 序数集： $0 = \emptyset, 1 = \{\emptyset, \{\emptyset\}\}, 2 = \{\emptyset, \{\emptyset, \{\emptyset\}\}\}, \dots, \aleph_0 = \{\emptyset, \{\emptyset, \{\emptyset, \dots\}\}\}, \dots$

对于一般抽象的理论，例如群我们该如何看待呢？其可以看成一个二元集 $G = \{x, \times\}$ ，其中 x 是群的元素集， \times 视为添加了限制的函数 $x \times x \rightarrow x$ 或者的三重笛卡尔积 $x \times x \times x$ ，限制实际就是使用方法(2)来描述子集，在群中包括了结合律、单位元存在和逆元存在，它们都是 ZFC 下的公式，从而限制出了乘法 \times ，当然有一个问题是群是否存在？此时我们可以利用(3)中的集合，构造出皮亚诺系统，从而得到初等数学理论，其中自然存在各种各样的群，因此集合论的基本思想就是“空集生万物，万物皆集合”。

5.2 其它系统

集合论确实美妙，但它并不具备凭空造集的能力，所有的集合都是从空集 \emptyset 通过公理中的几种运算生成的，我在以前范畴论的讨论文章 [11] 中说过，范畴是不能通过集合来定义的，并不是因为循环定义这种简单的问题，而是一阶逻辑对于范畴的判定能力不够，所以当时我们将范畴论视为一种逻辑理论，但如果我们在集合论的基础上，进一步给出类，一种可以凭空产生的对象，那么这个逻辑系统就能支撑范畴的定义了，这就是 NBG 系统，它的形式定义为

定义 5.2 (NBG)

在 L 的基础上引入三个谓词 \in, c, s , $x \in y$ 表示 x 属于 y , $c(x)$ 表示 x 是类 (class), $s(x)$ 表示 x 是集合 (set), 此时 NBG = (L, \in, c, s) 并有如下五组公理

- (A1) $\forall x(s(x) \rightarrow c(x))$: 集合都是类
- (A2) $\forall x \exists y(x \in y \rightarrow s(x))$: 类的元素都是集合
- (A3) $\forall X \exists Y(\forall x(x \in X \leftrightarrow x \in Y) \rightarrow X = Y)$: 外延公理
- (A4) $\forall s(x) \forall s(y) \exists s(z) \forall u(u \in z \leftrightarrow u = x \vee u = y)$: 无序对集合存在公理公式缩略一: $\forall s(x) A := \forall x(s(x) \rightarrow A)$
- (B1) $\exists X \forall x \forall y((x, y) \in X \leftrightarrow x \in y)$: 存在关系描述类 $E = \{(x, y) \mid x \in y\}$
- (B2) $\forall X \forall Y \exists Z(\forall u(u \in Z \leftrightarrow u \in X \wedge u \in Y))$: 存在类的交 $Z = X \cap Y$
- (B3) $\forall X \exists Y(\forall x(x \in Y \leftrightarrow \neg x \in X))$: 存在类的补 $Y = X^c$
- (B4) $\forall X \exists Y(\forall x(x \in Y \leftrightarrow \exists y((x, y) \in X)))$: 存在类关系的投影类 $X = (Y, Z)$
- (B5) $\forall X \exists Y(\forall x \forall y((x, y) \in Y \leftrightarrow x \in X))$: 存在类的笛卡尔积 $Y = X \times Z$
- (B6) $\forall X \exists Y(\forall x \forall y((x, y) \in Y \leftrightarrow (y, x) \in X))$: 存在类的逆 $Y = X^{-1}$
- (B7) $\forall X \exists Y(\forall x \forall y \forall z((x, y, z) \in Y \leftrightarrow (z, x, y) \in X))$: 类的次序交换一
- (B8) $\forall X \exists Y(\forall x \forall y \forall z((x, y, z) \in Y \leftrightarrow (x, z, y) \in X))$: 类的次序交换二
- 项缩略一: $(x, y) := \{x, \{x, y\}\}$ $(x, y, z) := (x, (y, z))$
- (C1) $\exists s(x)(\exists y(y \in x) \wedge \forall z(z \in x \rightarrow \exists t(t \in x \wedge (s \in t \leftrightarrow s = z \vee s \in z))))$: 无穷公理
- (C2) $\forall s(x) \exists s(y) \forall z(z \in y \leftrightarrow \exists u(u \in x \wedge z \in u))$: 并集存在公理
- (C3) $\forall s(x) \exists s(y) \forall z(z \in y \leftrightarrow \forall u(u \in z \rightarrow u \in x))$: 索引集合存在公理
- (C4) $\forall X(\forall s(t)(\forall x \in t \exists y((x, y) \in X) \rightarrow \exists S \forall x \in t \exists y \in S((x, y) \in X)))$: 替换公理
- (D) $\forall X(\exists y(y \in X) \rightarrow \exists y(y \in X \wedge \forall z(z \in y \rightarrow \neg z \in X)))$: 正则公理
- (E) $\exists X(\forall x \exists y((x, y) \in X \wedge \forall z(z \in x \rightarrow \exists y \in x((x, y) \in X)))$: 选择公理

另外我们还记 $GB = NBG - \{E\}$, NBG 系统的最大特色就是引入了集合之上一个类构造 (A1-A4)，在这个类构造中，可以凭空进行各种运算 (B1-B8)，并且保持了集合论的基本公理 (C1-C4)，而且通过凭空构造消去了公理模式 (D)(从而是有限公理化的)，并给出了简化的选择公理 (E)，显然 GB 是 ZF 的一个扩充，实际还能证明 GB 是 ZF 的一个保守扩充，即在 GB 中的定理都可以在 ZF 中证明，另外对于原来的选择公理 AC 来说有

$$NBG \vdash AC$$

由于 GB 和 ZF 的推理能力一样，所以也没啥好讲的，但由于 (B1-B8) 的存在，在 GB 系统下定义是比 ZF 方便很多的，甚至可以在纯类上讨论范畴论，所以讨论朴素集合论时，我们最好将其视为 GB 的模型会更好一些，这也是我们通常所用的属于符号。

定理 5.2

GB 是一致的当且仅当 ZF 是一致的。



在机器形式证明上，使用概括性的原则会更便于计算机理解，在 [14] 一书中，作者通过证明工具 Coq 来将大部分集合论的结论进行形式化，而他采用的公理集合论系统是 Morse-Kelley 系统，为了得到这个系统，首先

我们有概括原则

$$(CP) \exists X \forall x_1 \dots \forall x_n ((x_1, \dots, x_n) \in X \leftrightarrow A(x_1, \dots, x_n))$$

这是一个公理模式，说明有一个大类 X 可以描述所有的关系 A ，各种关系的拆分实际就是 (B1-B8)，显然有 $CP \vdash (B1 - B8)$ ，另一方面我们可以证明

$$\vdash_{GB} CP$$

这意味着在公理 $GB^* = GB - (B1 - B8)$ 下有

$$(B1 - B8) \Leftrightarrow CP$$

此时我们有一个新的系统 $MK = GB^* + CP$ ，这就是 Morse-Kelley 集合论。值得注意等的是 MK 不是 ZF 的保守扩充，显然 MK 的问题在于基数描述，我们在朴素集合论中引入一个新的概念，如果基数 κ 满足

- (1) $\kappa > \aleph_0$ (可数无穷基数由无穷公理给出)
- (2) 对任意序数 $n \in \mathbb{N}$ (在 ZF 下视为一个集合) 有，如果 $\forall a (a \in n \rightarrow a < \kappa)$ 且 $s < \kappa$ ，则 $\cup s < \kappa$
- (3) $\forall x (x < \kappa \rightarrow 2^x < \kappa)$

我们就称 κ 是不可达基数。集合虽然是可定义的，但不可达基数的存在性是独立于 ZF 的。我们把蕴含存在无穷多个不可达基数的强无穷公理添加到 ZF 得到的系统记为 ZM，则 ZM 反而变成了 MK 的扩充，即 MK 中的定理都是 ZM 的定理，至于其与选择公理的关系暂不明朗，因此在 [14] 中，并没有将实变函数等更复杂的理论给形式化。实际上，除了上面这些，还有更多的集合论，比如 Tarski-Grothendieck 集合论等，但从作为数学证明的基础上来讲，并不是十分刚需的，一些逻辑诡异也只能用在数理逻辑上，只能说食之无味弃之可惜，所以就随便提一提。

5.3 连续统

最后，我们再讲一下与集合论相关的连续统假设，它是关于基数的一个命题，在 ZFC 的公理之中，我们已经定义出了有穷基数 $n \in \mathbb{N}$ 和可数无穷基数 $|\mathbb{N}| = \aleph_0$ ，并且可以证明它们之间的无穷基数等于可数无穷基数。另外我们发现，通过幂集得到的基数一定是变大的

$$|x| < |2^x|$$

比如常规实例 $|\mathbb{N}| < |\mathbb{R}| = |2^{\aleph_0}|$ ，将其运用到可数基数上，我们就可以构造出无数的非可数无穷基数

$$n \in \mathbb{N}, \aleph_0 = |\mathbb{N}|, 2^{\aleph_0}, 2^{2^{\aleph_0}}, 2^{2^{2^{\aleph_0}}}, \dots$$

我们的问题是，是否存在一个基数在 \aleph_0 和 2^{\aleph_0} 之间？这就是连续统假设 (CH)。我们发现，无论怎么努力，都无法证明这个定理，此时我们自然会疑惑这个性质真的能证明吗？我们之所以会产生这个想法，是因为之前就有哥德尔不完全性定理指出，确实存在无法证明真假的命题。为了达到这个目的，我们必须将集合论形式化为一个一阶理论，于是就有了我们的 ZFC 系统。而我们的最终结论是

定理 5.3

$ZFC \not\vdash CH$ 且 $ZFC \not\vdash \neg CH$



即在 ZFC 系统下，我们既无法证明 CH 是真，也无法证明 CH 是假，换言之连续统假设 CH 相对于 ZFC 系统是独立的。证明的核心思想并不复杂，就是使用模型论的方法构造两个模型，使得一个模型下 $ZFC+CH$ 成立，且另一个模型中 ZFC 成立但 CH 不成立，详细参考 [7]，虽然老了点，但还不错。

第六章 数学基础其二：类型论

实际上，有助于数学研究的数理逻辑大框架已经被我们讨论完了，剩下的就是留给读者去探索各个领域的更深层次内容。数理逻辑作为一个研究“证明”的学科，我们不好好研究证明这件事本身，反而去探究各种不可能性多少有些奇怪，但是研究这方面的往往不止数学，所以导致体系十分紊乱，例如还有证明论 [5, 8] 和代数逻辑 [6] 等，不过其并没有带来什么值得回味的东西，所以个人觉得没什么好探究的。真正值得注意的是计算机科学所推崇的类型论，虽然并不是全部，但基本所有计算机辅助证明的理论基础都是更符合直觉的类型论，所以这就构成了本章内容，参考书目为 [4] 和[网址](#)，都是我很喜欢且比较新的书。另外国内也有一个讨论类型论的组织，有兴趣的读者可以去观摩一下。

6.1 基础知识

类型论和公理集合论一样也是一个形式系统，它们的区别在于，集合论需要一阶逻辑作为演绎系统和 ZFC 作为公理系统，而类型论则只有一个规则系统，类型 (types)，无论是命题还是演绎都视为一种类型，这其实有一点程序语言中面向对象的思想，类型论虽然是一个形式系统，但按照某空间的说法，我们只能得到各种具体的类型论，实际上就是将各种规则进行具体化，不同类型论之间存在规则的细微差异。我们将介绍目前主流工具 Coq 和 Lean 所采用的 CoC(Calculus of Constructions) 类型论，它的堆叠过程如下

$$\lambda C = \lambda 2 + \lambda \omega + \lambda P$$

但我们一般不考察各 λ 系统之间的关系，而是直接给出 CoC 的构造。所谓的项 (term) 指

- (1) 类型 (type): \square (一个)、命题 (proposition): $*$ (一个)、变量 (Variables): x, y, z, x_1, x_2, \dots (无数个)
- (2) 如果 A 和 B 是项，则 AB 是项
- (3) 如果 A 和 B 是项且 x 是变量，则 $\lambda x : A.B$ 和 $\Pi x : A.B$ (也可以写做 $\forall x : A.B$) 都是项

我们以后用小写 x, y, z, \dots 表示变量，用大写 A, B, C, \dots 表示项，另外对于项 B ，如果项 A 包含变量 x ，则 $A[x:=B]$ 表示将全部变量 x 替换成 B 所得到的项(但注意不能产生变量冲突，读者应该可以自行理解)，如果项 A 不包含变量 x ，则 $A[x:=B]=A$ 。如果两个项 A 和 B 完全一样，我们就记 $A \equiv B$ ；对于两个项 A 和 B ，引入 β -约化 $A \rightarrow_{\beta} B$ 为

- (1)(基础步) 对任意项 L , $(\lambda x : L.M)N \rightarrow_{\beta} M[x := N]$
- (2)(递归步) 对任意项 L , 如果 $M \rightarrow_{\beta} N$, 则有 $ML \rightarrow_{\beta} NL$ 、 $LM \rightarrow_{\beta} LN$ 、 $(\lambda x : L.M) \rightarrow_{\beta} (\lambda x : L.N)$ 、 $(\lambda x : M.L) \rightarrow_{\beta} (\lambda x : N.L)$ 、 $(\Pi x : L.M) \rightarrow_{\beta} (\Pi x : L.N)$ 、 $(\Pi x : M.L) \rightarrow_{\beta} (\Pi x : N.L)$

并引入 β -转化 $M =_{\beta} N$, 表示存在一系列的项 $M_0 \equiv M, M_1, \dots, M_n \equiv N$ 使得对任意 $0 \leq i < n$ 有, $M_i \rightarrow_{\beta} M_{i+1}$ 或 $M_{i+1} \rightarrow_{\beta} M_i$ 。我们把“项 1: 项 2”形式(例如, $* : \square$ 、 $x : B$ 、 $\lambda x : A.B : C$ 等)称为一个陈述 (statement)，其意思为“元项 1 具有类型项 2”；如果形式“项 1: 项 2”满足“项 1”是变量(例如, $x : A$ 、 $x : *$ 等)，则称它是一个声明 (declaration)；我们将一列声明“变量 1: 项 1, 变量 2: 项 2,..., 变量 n: 项 n”称为一个上下文 (context)，并在以后使用大写希腊字母(例如 Γ 等)来表示它们，并特别使用 \emptyset 来表示空的上下文，其中“变量 1,..., 变量 n”互不相同，并且对于变量 x 和上下文 Γ ，用 $x \in \Gamma$ 表示变量 x 在声明 Γ 的变量表中；最后，对于上下文 Γ 和陈述 $M : N$ ，我们把

$$\Gamma \vdash M : N$$

称为一个判断 (judgement)，其意思为“在语境 Γ 下，元 M 具有类型 N ”。后面我们会稍微使用一下集合的符号，比如 $s \in \{\square, *\}$ 表示 $s \equiv \square$ 或 $s \equiv *$ ，我想有了上一章的铺垫，读者很容易理解这不叫集合论，所以使用的时候请不要介意。我们将下面的形式

$$(name) \frac{\text{判断 1 判断 2 ... 判断 n}}{\text{判断}}, \text{restrict}$$

称为一个推导规则 (Derivation rule 或 Inference rule)，其中“判断 1,..., 判断 n”被称为假设 (premiss)，“判断”被称为结论 (conclusion)。此时，CoC 类型论或 λC 系统的推导规则为

$$\begin{array}{c} (\text{sort}) \frac{}{\emptyset \vdash * : \square} \\ (\text{var}) \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad x \notin \Gamma, s \in \{\square, *\} \\ (\text{weak}) \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad x \notin \Gamma, s \in \{\square, *\} \\ (\text{form}) \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2} \quad s_1, s_2 \in \{\square, *\} \\ (\text{appl}) \frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \\ (\text{abst}) \frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B} \quad s \in \{\square, *\} \\ (\text{conv}) \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma \vdash A : C} \quad B =_{\beta} C, s \in \{\square, *\} \end{array}$$

我想读者应该可以理解为什么类型论自带逻辑系统了吧，显然推导规则就已经有条件证明 $\Gamma \vdash \varphi$ 的意思了，但其在推导规则中将条件证明 $\Gamma \vdash \varphi$ 具化成了一个判断，CoC 实际就是一个“判断的推导系统”，和一阶逻辑通过“初始公式”和“变形规则”来得到是定理的公式类似，在类型论中，我们则需要通过“推导规则”来得到一个个的“判断”，为了靠近一阶逻辑，对于项，我们引入下面的缩写

$$\begin{aligned} \perp &\equiv \Pi x : *.x \\ A \rightarrow B &\equiv \Pi x : A.B \quad x \notin B \\ A \wedge B &\equiv \Pi x : *. (A \rightarrow B \rightarrow x) \rightarrow x \\ A \vee B &\equiv \Pi x : *. (A \rightarrow x) \rightarrow (B \rightarrow x) \rightarrow x \\ \neg A &\equiv \Pi x : *. (A \rightarrow x) \\ \exists x : A.B &\equiv \Pi y : *. (\Pi x : A. (B \rightarrow y)) \rightarrow y \\ \text{bool} &\equiv \Pi x : *. x \rightarrow x \rightarrow x \end{aligned}$$

我知道你现在一头雾水，但你先别急，我们先把 λC 系统搞清楚再说。第一条规则 sort 表示“命题具有类型的类型”，读着绕口无所谓，关键在于它可以得到推理的起点，除了 sort 其它所有的规则都是有假设的，这其实和空集在集合论中作为构造起点的作用是类似的。第二条 var 和第三条 weak 都是变量的引入规则，告诉我们在推理过程中，该如何引入变量，并且引入的变量不能在上下文中出现过，这和程序引入变量的过程有些类似。此时我们以 sort 作为起点，不断利用 var 和 weak，我们就能得到判断

$$x : * \vdash x : *$$

$$x : * \vdash * : \square$$

$$x : *, y : x \vdash x : *$$

$$x : *, y : x \vdash y : x$$

$$x : *, y : x \vdash * : \square$$

$$x : *, y : * \vdash x : *$$

$$x : *, y : * \vdash y : *$$

$$x : *, y : * \vdash * : \square$$

其就是把上下文放到了结果中，而上下文¹可以产生的格式为

$$x_1 : *, x_2 : *, \dots, x_n : *, y_1 : x_{i_1}, y_2 : x_{i_2}, \dots, y_n : x_{i_n}$$

上面 y_j 是可以往前靠的，只要保证与之相关的 x_{j_i} 已经出现过即可²，为了方便我们用 | 来分割 \vdash 后面的结果，于是有一个通用判断

$$x_1 : *, \dots, x_n : *, y_1 : x_{i_1}, y_2 : x_{i_2}, \dots, y_n : x_{i_n} \vdash * : \square | x_1 : * | \dots | x_n : * | y_1 : x_{i_1} | y_2 : x_{i_2} | \dots | y_n : x_{i_n}$$

虽然目前基本导不出什么判断，但这两条规则有着很大的后劲。第四条规则 form 表示谓词形成原理，首先对于 $\Pi x : A.B$ ，我们应该视为 $(\forall x : A)B$ ，即对于集合 A 中的元素 x 谓词 B 都是成立的，虽然我们看不出 B 像一个谓词，但我们知道 B 是一个项，并且可能包含变量 x，于是确实有谓词 $B(x)$ 的那味了。通过这个规则，再结合前面的 sort,var 和 weak，我们就能得到大量的判断了

$$\emptyset \vdash \Pi x : *.x : *$$

$$y : * \vdash \Pi x : *.x : *$$

$$y : \Pi x : *.x \vdash * : \square | y : \Pi x : *.x$$

$$\emptyset \vdash \Pi x : *.x : \square$$

$$y : * \vdash \Pi x : *.x : \square$$

$$\emptyset \vdash \Pi y : *. \Pi x : *.x : \square$$

显然，我们只需要反复循环的“form->weak->form->weak...”并使用 sort 作为另一个判断就能得到无限个判断，并且如果我们借助前面 $A \rightarrow B$ 的缩写，这无限个判断就相当于

$$\emptyset \vdash * \rightarrow * \rightarrow * \rightarrow \dots \rightarrow * : \square$$

这里的箭头 \rightarrow 和一阶逻辑一样是从右往左结合，但我们只要借助这些判断并通过 form 规则就可以得到任意结合的判断了，另一方面，如果我们以 $x : *, y : * \vdash x : * | y : *$ 作为起点，使用同样的循环就能得到另一类任意结合的判断了

$$x_1 : *, x_2 : *, \dots, x_n : * \vdash x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n : *$$

生成也不过如此，因此与之相对的第五条规则 appl 表示谓词使用原理，它实际相当于代入的形式书写， $M : \Pi x : A.B$ 表示 M 是一个 $\Pi x : A.B$ 类型的谓词，则对于元素 $N : A$ ，将 N 代入 M 的结果 MN 是 $B[x := N]$ 类型。当然仅由现在的公理，我们无法生成谓词的实例 $M : \Pi x : A.B$ ，此时就需要利用第六条规则 abst，来生成函数作为谓词实例，这里的 $\lambda x : A.B$ 是函数的 λ 演算形式，其表示变量为 x，定义域为 A，值域为 B。我们容易得到下面两个判断

$$x : * \vdash x \rightarrow x : * \equiv x : * \vdash \Pi z : x.x : *$$

¹读者要注意上下文是有序的，并且相同的判断可以同时作为 weak 的假设

²关键其实在于，我们得到 $\Gamma \vdash * : \square$ 以后， Γ 就能作为万用上下文来进行结论复现

$$\emptyset \vdash * \rightarrow * : \square \equiv \emptyset \vdash \Pi z : x.x : \square$$

此时，我们只要用 abst 就能得到一个函数实例了

$$\emptyset \vdash \lambda x : *.x \rightarrow x : * \rightarrow *$$

展开箭头 \rightarrow 可得

$$\emptyset \vdash \lambda x : *. \Pi z : x.x : \Pi z : x.x$$

由于 $\Pi z : X.Y$ 中的变量 z 是非自由的³，其出不出现是无所谓的，因此以后我们均采用简写 $X \rightarrow Y$ ，此时我们对任意 \square 类型的对象都能得到一个任意结合的函数实例

$$\emptyset \vdash \lambda x : *.x \rightarrow x \rightarrow \dots \rightarrow x : * \rightarrow * \rightarrow \dots \rightarrow *$$

如果想要箭头中的对象多元化，要么可以申明在判断的上下文中，要么多叠几个 λ 函数，既然得到了函数，那么就该用一用了，我们先随便加个上下文 $y : * \vdash \lambda x : *.x \rightarrow x : * \rightarrow *$ ，结合 $y : * \vdash y : *$ 并利用 appl 规则，即可得到判断

$$y : * \vdash (\lambda x : *.x \rightarrow x)y : *[z := y] \equiv y : * \vdash (\lambda x : *.x \rightarrow x)y : *$$

这样一搞其实就回到了 λ_\rightarrow 系统的内容了。最后一个规则 conv 就没啥好讲的了，简单来说就是把 A 的类型 B 换成可以 β -转化的类型 C，一个 β -约化（转化）的实例是

$$(\lambda x : *.x \rightarrow x)y \rightarrow_\beta (x \rightarrow x)[x := y] \equiv y \rightarrow y$$

$$(\lambda x : *.x \rightarrow x)y =_\beta y \rightarrow y$$

为了使用这个规则，我们需要先把函数移到陈述的后面，做法是利用 weak 引入变量

$$y : *, z : (\lambda x : *.x \rightarrow x)y \vdash * : \square$$

设定上下文 $\Gamma \equiv y : *, z : (\lambda x : *.x \rightarrow x)y \vdash *$ ，则可以得到

$$\Gamma \vdash z : (\lambda x : *.x \rightarrow x)y$$

$$\Gamma \vdash y \rightarrow y : *$$

此时就能使用 conv 规则了

$$\Gamma \vdash z : y \rightarrow y$$

这难道是函数指针？就留给读者自行体会了。此时我们好像得到了强化的 $\lambda 2, \lambda \omega$ 系统，还有一个遗憾是没有把谓词系统 $\Pi x : A.B$ 给充分利用起来，单看形式系统，我们是难以理解的 λC 系统的，所以我们下一步来讨论缩写形式所代表的逻辑内涵吧。

6.2 逻辑内涵

我们先来看命题逻辑，抽取最简单的（与 CoC 联系上最简单）两个生成符 $\{\perp, \rightarrow\}$ ，在命题逻辑中我们有

³另外， $\lambda x : X.Y$ 中的 x 也是非自由的，因此可以发现判断的上下文中并不需要变量声明 $x : * \vdash \dots$

$$\neg A \equiv A \rightarrow \perp, A \wedge B \equiv \neg(A \rightarrow \neg B), A \vee B \equiv \neg(\neg A \rightarrow \neg B)$$

这与我们上面给出的定义缩写是相匹配的⁴，因此 CoC 的逻辑符比单纯的命题逻辑有着更丰富的内涵。对于一阶逻辑，其多出了一个全称量词 \forall 和缩写

$$\exists x A \equiv \neg(\forall x \neg A)$$

基本对应为 $\forall x \in A(B(x)) \equiv \Pi x : A.B$ ，因此 CoC 中的全称量词还具有范围限制，同样比一阶逻辑更加丰富。接着考察在 CoC 中逻辑相关的判断，由于上述操作相当于在缩写，因此并不影响推断，于是只要有判断 $\Gamma \vdash A : * | B : *$ ，我们就能相应地得到判断

$$\Gamma \vdash \neg A : * | A \wedge B : * | A \vee B : * | \forall x : A.B : * | \exists x : A.B : *$$

接着我们需要像一阶逻辑一样考察原子命题，即说明“谓词在项上的类似物”也是命题类型 $*$ ，这样我们就能说明 $*$ 确实是命题类型。函数对应着没有自由变量的 λ 表达式， $F(x_1, \dots, x_n) \equiv \lambda x_1 : *. \lambda x_2 : *. \dots \lambda x_n : *. (\dots)$ ，它的定义域和值域都是 $*$ 类型，而整个函数是一个星号 $*$ 箭头式的类型，不全代入值时，依旧是一个星号 $*$ 箭头式的类型，当完全代入时就变成了一个 $*$ 类型，这样一阶逻辑中的常量在此被视为 $*$ 号类型。与之对应，没有自由变量的 Π 表达式， $P(x_1, \dots, x_n) \equiv \Pi x_1 : *. \Pi x_2 : *. \dots \Pi x_n : *. (\dots)$ ，对应了一阶逻辑中的谓词，对应了星号 $*$ 类型。对于一阶逻辑，我们还需要考虑真值问题

$$\begin{aligned} \text{bool} &\equiv \Pi x : *. x \rightarrow x \rightarrow x \\ 0 &\equiv \lambda b : *. \lambda x, y : b. x \\ 1 &\equiv \lambda b : *. \lambda x, y : b. y \end{aligned}$$

这里使用了一个同定义域的缩写 $\lambda x, y : A.B \equiv \lambda x : A. \lambda y : A. B$ ，通过简单的推导可知

$$\emptyset \vdash \text{bool} : *$$

$$\emptyset \vdash 0 : \text{bool}, 1 : \text{bool}$$

即 bool 值集是一个命题类型，而布尔值“0=False, 1=True”是一个布尔值类型。利用推导规则，我们可以将上述逻辑连接词实例化成一个函数（它们都具有命题类型 $*$ 是可以做到的）

$$\neg \equiv \lambda x : *. (x \rightarrow \perp)$$

$$\wedge \equiv \lambda x, y : *. \Pi z : *. (x \rightarrow y \rightarrow z) \rightarrow z, \vee \equiv \lambda x, y : *. \Pi z : *. (x \rightarrow z) \rightarrow (y \rightarrow z) \rightarrow z$$

此时如果有命题类型 $A : *, B : *$ 就可以得到（此处是项的相乘）

$$\neg A \equiv (\lambda x : *. (x \rightarrow \perp)) A \rightarrow_{\beta} A \rightarrow \perp$$

$$\wedge AB \equiv (\lambda x, y : *. \Pi z : *. (x \rightarrow y \rightarrow z) \rightarrow z) AB \rightarrow_{\beta} \Pi z : *. (A \rightarrow B \rightarrow z) \rightarrow z$$

$$\vee AB \equiv (\lambda x, y : *. \Pi z : *. (x \rightarrow z) \rightarrow (y \rightarrow z) \rightarrow z) AB \rightarrow_{\beta} \Pi z : *. (A \rightarrow z) \rightarrow (B \rightarrow z) \rightarrow z$$

因此逻辑的代入替换思想已经包含在了其中，接着我们只需要把上面的定义域给限制起来，就可以使得上述连接符成为一个布尔函数

⁴需注意有 $x \notin (\Pi x : *. x)$ ，原因是非自由

$$\neg \equiv \lambda x : \text{bool}.(x \rightarrow \Pi z : \text{bool}.z)$$

$$\wedge \equiv \lambda x, y : \text{bool}.\Pi z : \text{bool}.(x \rightarrow y \rightarrow z) \rightarrow z, \vee \equiv \lambda x, y : \text{bool}.\Pi z : \text{bool}.(x \rightarrow z) \rightarrow (y \rightarrow z) \rightarrow z$$

显然进行代入以后得到的也是布尔类型。因此，* 确实就是命题类型的意思，而另一个□呢？它的目的和集合论中的空集类似，目的是无中生有，并且涉及我们接下来的课题了。

6.3 集合论

通过上面的内容我们可以知道，在 CoC 中定义实际就是确定一个项的形式，或者说确定一种类型，而其相应的函数实例化就是这个类型所能取到的值，比如前面的 $\text{bool} = \{0, 1\}$ 。为了将定义作为上下文减少书写，并使用程序的模块化思想，我们需要将 λC 系统扩充为 λD ，为此我们先附带一堆常量 (Constants，注意这不是项): a, b, c, a_1, a_2, \dots (无数个)，并扩充项为

- (1) □、*、变量 (Variables): x, y, z, x_1, x_2, \dots (无数个)
- (2) 如果 A 和 B 是项，则 AB 是项
- (3) 如果 A 和 B 是项且 x 是变量，则 $\lambda x : A.B$ 和 $\Pi x : A.B$ (也可以写做 $\forall x : A.B$) 都是项
- (4) 如果 A_1, \dots, A_n 是项且 c 是常量，则 $c(A_1, \dots, A_n)$ 是项

方便起见我们进行如下简写， $\bar{A} \equiv A_1, \dots, A_n$ 、 $\bar{x} \equiv x_1, \dots, x_n$ 、 $\bar{x} : \bar{A} \equiv x_1 : A_1, \dots, x_n : A_n$ ，以此类推。此时，对于变量 x_i 、常量 a 和项 A_i, M, N ，我们把记号

$$\bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N$$

称为一个描述性定义 (descriptive definition)，进一步我们把

$$\bar{x} : \bar{A} \triangleright a(\bar{x}) := \perp : N$$

称为一个原子性定义 (primitive definition)，描述性或原子性定义统称为定义 (definition)， $\bar{x} : \bar{A}$ 称为这个定义的上下文， $a(\bar{x})$ 称为这个定义的定义元 (definiendum)， $M : N$ 或 $\perp : N$ 称为这个定义的陈述 (statement)；我们将一列定义 “定义 1,..., 定义 n” 称为环境 (environment)，此时 λD 的一个判断 (judgement) 指

$$\Delta; \Gamma \vdash M : N$$

其中 Δ 是环境、 Γ 是上下文、 $M : N$ 是陈述。此时对于 β -约化有稍许的改动

(1)(基础步) 对任意项 L ， $(\lambda x : L.M)N \rightarrow_{\beta} M[x := N]$

(2)(递归步) 对任意项 L, U_i, V_i 和常数 c ，如果 $M \rightarrow_{\beta} N$ ，则有 $ML \rightarrow_{\beta} NL$ 、 $LM \rightarrow_{\beta} LN$ 、 $(\lambda x : L.M) \rightarrow_{\beta} (\lambda x : L.N)$ 、 $(\lambda x : M.L) \rightarrow_{\beta} (\lambda x : N.L)$ 、 $(\Pi x : L.M) \rightarrow_{\beta} (\Pi x : L.N)$ 、 $(\Pi x : M.L) \rightarrow_{\beta} (\Pi x : N.L)$ 和 $c(\bar{U}, M, \bar{V}) \rightarrow_{\beta} c(\bar{U}, N, \bar{V})$

而且我们还需要引入一个 δ -约化 (即由定义而产生的约化)，其需要描述性定义 $\Gamma \triangleright a(\bar{x}) := M : N$ 作为环境 Δ 的一员来作为前提

(1)(基础步) $c(\bar{U}) \rightarrow_{\Delta} M[\bar{x} := \bar{U}]$

(2)(递归步) 如果 $M \rightarrow_{\Delta} N$ ，则有 $ML \rightarrow_{\Delta} NL$ 、 $LM \rightarrow_{\Delta} LN$ 、 $(\lambda x : L.M) \rightarrow_{\Delta} (\lambda x : L.N)$ 、 $(\lambda x : M.L) \rightarrow_{\Delta} (\lambda x : N.L)$ 和 $c(\bar{U}, M, \bar{V}) \rightarrow_{\Delta} c(\bar{U}, N, \bar{V})$

这里的 $M[\bar{x} := \bar{U}]$ 相当于多次代入 $M[x_1 : U_1][x_2 := U_2]...[x_n : U_n]$ 的简写。此时的 $\beta\delta$ -转化，实际就是两者的复合，即 $M \stackrel{\Delta}{=} N$ 当且仅当，存在一系列的项 $M_0 \equiv M, M_1, \dots, M_n \equiv N$ 使得对任意 $0 \leq i < n$ 有， $M_i \rightarrow_{\beta} M_{i+1}$ 或 $M_{i+1} \rightarrow_{\beta} M_i$ 或 $M_i \rightarrow_{\Delta} M_{i+1}$ 或 $M_{i+1} \rightarrow_{\Delta} M_i$ 。此时，我们就足以叙述出推导规则了

$$\begin{array}{c}
(sort) \frac{}{\emptyset; \emptyset \vdash * : \square} \\
(var) \frac{\Delta; \Gamma \vdash A : s}{\Delta; \Gamma, x : A \vdash x : A} \quad x \notin \Gamma, s \in \{\square, *\} \\
(weak) \frac{\Delta; \Gamma \vdash A : B \quad \Delta; \Gamma \vdash C : s}{\Delta; \Gamma, x : C \vdash A : B} \quad x \notin \Gamma, s \in \{\square, *\} \\
(form) \frac{\Delta; \Gamma \vdash A : s_1 \quad \Delta; \Gamma, x : A \vdash B : s_2}{\Delta; \Gamma \vdash \Pi x : A. B : s_2} \quad s_1, s_2 \in \{\square, *\} \\
(appl) \frac{\Delta; \Gamma \vdash M : \Pi x : A. B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash MN : B[x := N]} \\
(abst) \frac{\Delta; \Gamma, x : A \vdash M : B \quad \Delta; \Gamma \vdash \Pi x : A. B : s}{\Delta; \Gamma \vdash \lambda x : A. M : \Pi x : A. B} \quad s \in \{\square, *\} \\
(conv) \frac{\Delta; \Gamma \vdash A : B \quad \Delta; \Gamma \vdash C : s}{\Delta; \Gamma \vdash A : C} \quad B \stackrel{\Delta}{\equiv}_{\beta} C, s \in \{\square, *\} \\
(def) \frac{\Delta; \Gamma \vdash K : L \quad \Delta; \bar{x} : \bar{A} \vdash M : N}{\Delta, \bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N; \Gamma \vdash K : L} \quad a \notin \Delta \\
(def - prim) \frac{\Delta; \Gamma \vdash K : L \quad \Delta; \bar{x} : \bar{A} \vdash N : s}{\Delta, \bar{x} : \bar{A} \triangleright a(\bar{x}) := \perp : N; \Gamma \vdash K : L} \quad a \notin \Delta, s \in \{\square, *\} \\
(inst) \frac{\Delta; \Gamma \vdash * : \square \quad \Delta; \Gamma \vdash \bar{U} : \overline{A[\bar{x} := \bar{U}]}}{\Delta; \Gamma \vdash a(\bar{U}) : N[\bar{x} := \bar{U}]} \quad \bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N \in \Delta \\
(inst - prim) \frac{\Delta; \Gamma \vdash * : \square \quad \Delta; \Gamma \vdash \bar{U} : \overline{A[\bar{x} := \bar{U}]}}{\Delta; \Gamma \vdash a(\overline{U}) : N[\bar{x} := \bar{U}]} \quad \bar{x} : \bar{A} \triangleright a(\bar{x}) := \perp : N \in \Delta \\
(par) \frac{\Delta; \bar{x} : \bar{A} \vdash M : N}{\Delta, \mathcal{D}; \bar{x} : \bar{A} \vdash a(\bar{x}) : N} \quad \mathcal{D} \equiv \bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N, a \notin \Delta
\end{array}$$

系统 λD 的前 7 条，除多了一个环境 Δ 以外，和 λC 系统的推导规则是基本一样的。规则 def 和 def-prim 告诉我们如何定义，并把其引入环境中，规则 inst 和 inst-prim 告诉我们如何使用环境中的定义，最后一个规则 par 其实也是定义的引入规则，为了看清三个定义引入规则 (def, def-prim, par)，我们再来理解一下定义

$$\bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N$$

这里的 $\bar{x} : \bar{A}$ 是定义的依赖，即为了定义 a 需要的一些前置定义，通常它们都是每个类型 A_i 的实例 x_i ； $a(\bar{x})$ 相当于这个定义的名称，实际上只要一个 a 就足够了，但在数学上，为了体现它的依赖就把相应的实例写进名称中；最后的 $M : N$ 才是真正的定义描述。我们来举一个数学基础集合论的示例，我们使用 $*_s$ 表示集合类型，使用 $*_p$ 表示命题类型，但它们实际都是命题类型 * 并无区别，此时⁵

$$\begin{aligned}
S : *_s \triangleright ps(S) &:= S \rightarrow *_p : \square \\
S : *_s, x : S, V : ps(S) \triangleright element(S, x, V) &:= Vx : *_p \quad note : x \in_S V \equiv element(S, x, V) \\
S : *_s, x : S, V : ps(S), W : ps(S) \equiv \Gamma & \\
\Gamma \triangleright \subset (S, V, W) &:= \forall x : S. (x \in_S V \rightarrow x \in_S W) : *_p \\
\Gamma \triangleright \cup (S, V, W) &:= \lambda x : S. (x \in_S V \vee x \in_S W) : ps(S)
\end{aligned}$$

这时你可能会有疑惑，虽然好多事都能干起来了，但似乎不像宣传的那样直观？实际上，这类函数式的类型论主要是适用于计算机的，如果想要直觉主义的类型论，那么应该考虑 Martin-Löf 类型论，详细可以看这篇[文章](#)，至于我们的目的本来就是计算机，所以就不做过多讨论了。

⁵详细可以看 [4] 的 13 章的各个图 (Figure)，其将上下文写得更贴近于程序，但含义和我们所写是一样的

第七章 证明辅助工具

光说不练假本事，因此我们来详细探究一下目前主流的一些证明辅助工具 (Proof Assistant)。工具的参考目录来自于[这里](#)，可以看到基于 CoC 系统的 Coq 和 Lean 拥有目前最强的推导能力，因此它们的使用将成为我介绍的重点，当然我也会涉及其它工具的说明。而我们所介绍的工具主要是[此处](#)列举的几个，主要还是因为以问题为导向是十分重要的，如果形式化证明工具没办法形式化一些重要结论，那这形式化工具不要也罢。

7.1 Coq

安装与运行

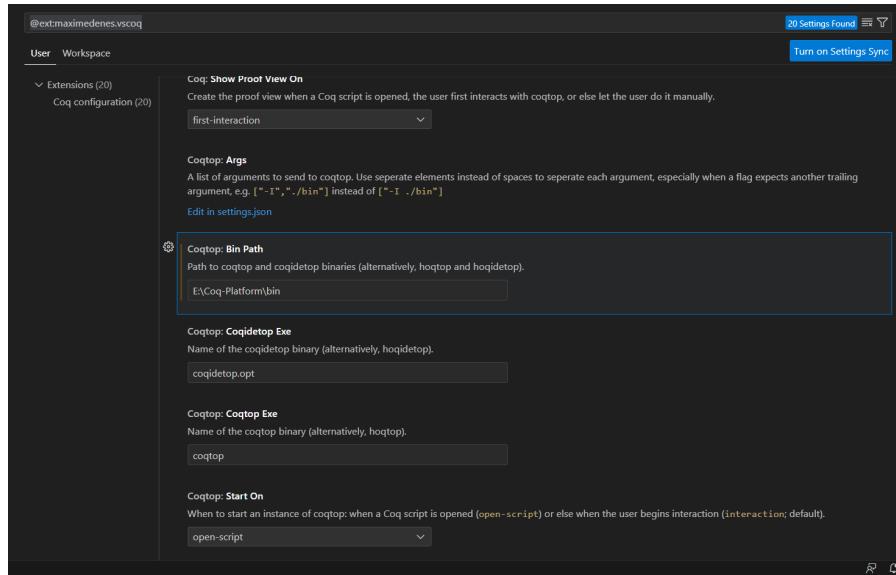
软件的安装与运行真的需要教吗？其实是不需要的，这实在过于基本了。此部分我想要介绍的是我所使用的 Coq 环境情况，对于开发类的说明讲解，统一运行环境是十分重要的，否则相同的代码很容易产生不同的结果。首先，我们进入[Coq 首页](#)；接着在“Running Coq”一栏中点击“Install Coq on your computer”；然后在“The Coq Platform (recommended)”一栏中点击“Windows and macOS installers”。此时会进入一个[github](#)仓库

The screenshot shows the GitHub repository page for the Coq project. The main navigation bar includes 'Code', 'Issues (42)', 'Pull requests (11)', 'Actions', 'Projects', 'Security', and 'Insights'. Below the navigation is a search bar with placeholder 'Type [] to search' and a 'Compare' button. The main content area displays the '2022.09.1 (Latest)' release. It includes a note from the maintainer 'MSoeigetropIMC' released on Jan 16, 2022, with 106 commits since the release. The release date is 2022.09.1 and the commit hash is 2397a58. A 'Compare' button is also present. Below the release notes, there's a section for 'Recommended binary installers' which lists Windows (64 bit) installer for Coq 8.16, macOS (Intel) installer for Coq 8.16, macOS (M1/M2/Apple Silicon/ARM) installer for Coq 8.16, and Linux (Snap) installer for Coq 8.16. There's also a 'General information' section with links to the README, Charter, CEP52, and detailed installation instructions for macOS, Linux, and Windows.

我们采用基本所有人都有的 window10 系统。这时你会发现电脑没有任何的变化，难道 Coq 就只是一个命令行程序吗？根据[官方文档](#)，似乎有一个 CoqIDE，的确我们可以在安装目录的 bin 下找到“coqide.exe”，不过个人体验下来不咋地，因此还是来用我最爱的 VSCode 吧。我们需要安装 VsCoq 扩展

The screenshot shows the VsCoq extension page in the VS Code Marketplace. The top navigation bar includes 'EXTENSIONS: MARKETPLACE' and 'VS CODE'. The main search bar contains 'reference.bb'. The results show the 'VsCoq' extension by 'Maxime Déénès'. The extension has a rating of 5 stars and 20,040 installs. It is described as an extension for Visual Studio Code with support for the Coq Proof Assistant. The 'DETAILS' tab is selected, showing the version v0.3.8, build passing, contributions, website, code of conduct, chat on zulip, and Visual Studio Marketplace v0.3.8. The 'open vsix v1.9.1' link is also present. The 'FEATURE CONTRIBUTIONS' tab is shown below. The 'Categories' section includes 'Keymaps' and 'Extension Resources' with links to 'Marketplace', 'Repository', and 'segbeill'. The 'More Info' section provides details about the extension's developer, last release date (2016-11-29), identifier, and general keybindings for vscoq. The 'Features' section lists asynchronous proofs, syntax highlighting, commands for step forward, interrupt, and more. The 'Requirements' section specifies VS Code or VSCode 1.38.0 or more recent, and Coq 8.7.0 or more recent.

由于 Coq 安装时，并没有产生环境变量，因此为了让插件找到 Coq，我们需要在插件设置“Coqtop:Bin Path”里填入 Coq 二进制程序的位置，即安装目录下的 bin 路径



最后就是运行了，要像一般开发语言来个“Hello World”的吗？当然没必要了。Coq 和开发语言不同，功能是比较单一的，主要做数学验证，不具备各种 IO 接口，这意味着 Coq 的实例实际只要一个就足够了，它并不像开发语言那样有着各种变形，通常我们只要一个范例就能搞懂用法，剩下的就是数学的东西了。例外情况是，你对这个证明辅助工具的理论基础并不清晰，比如使用 Coq 的时候，你不懂类型论，那这确实就需要像学习开发语言一样去一点点搞懂各个指令的含义和用法了。对于我和我的读者来说，当然使用的是数学角度，这意味着我默认你懂得这个证明辅助工具的理论基础，所以以后介绍用法的时候，我基本都是以实例作为导向的。

实例学习

我们以证明“根号 2 是无理数”为例，代码位于[这里](#)，下面是截取的主要片段

```

124 Theorem comparison4 : 3 * q - 2 * p < q.
125   apply Nat.add_lt_mono_l with (2 * p).
126   rewrite Nat.add_comm; rewrite Nat.sub_add;
127   | try (simple apply Nat.lt_le_incl; auto with arith).
128   replace (3 * q) with (2 * q + q); try ring.
129   apply Nat.add_lt_le_mono; auto.
130   repeat rewrite (Nat.mul_comm 2); apply mult_lt; auto with arith.
131 Qed.
132 
133 Remark mult_minus_distr_l : forall a b c : nat, a * (b - c) = a * b - a * c.
134 intros a b c; repeat rewrite (Nat.mul_comm a); apply Nat.mul_sub_distr_r.
135 Qed.
136 
137 Remark minus_eq_decompose : forall a b c d : nat, a = b -> c = d -> a - c = b - d.
138 intros a b c d H H0; rewrite H; rewrite H0; auto.
139 Qed.
140 
141 Theorem new_equality : (3 * p - 4 * q) * (3 * p - 4 * q) = 2 * ((3 * q - 2 * p) * (3 * q - 2 * p)).
142   repeat rewrite sub_square_identity; auto with arith.
143   repeat rewrite square_recompose; rewrite mult_minus_distr_l.
144   apply minus_eq_decompose; try rewrite hyp_sqrt; ring.
145 Qed.
146 
147 End sqrt2_decrease.
148 
149 #[local] Hint Resolve Nat.lt_le_incl comparison2: sqrt.
150 
151 Theorem sqrt2_not_rational : forall p q : nat, q <> 0 -> p * p = 2 * (q * q) -> False.
152 intros p q; generalize p; clear p; elim q using (well_founded_ind lt_wf).
153 clear q; intro q Hrec p Hneq;
154   pose proof Hneq as Hlt_0_q; apply Nat.neq_0_lt_0 in Hlt_0_q;
155   intros Heq.
156   apply (Hrec (3 * q - 2 * p) (comparison4 _ _ Hlt_0_q Heq) (3 * p - 4 * q)).
157   apply sym_not_equal; apply lt_neq; apply Nat.add_lt_mono_l with (2 * p);
158   | rewrite <- plus_n_O; rewrite Nat.add_comm; rewrite Nat.sub_add; auto with *.
159   apply new_equality; auto.
160 Qed.
```

此处根据我的个人习惯进行了缩进，在使用和我相同环境的情况下，可以直接运行到底部，换言之证明是成功的，接下来，我将从头解析所有的证明代码。第 1-14 行是开源协议的说明，此处告诉了我们注释代码为

(* 注释内容 *)

紧跟其后的是 Coq 标准库的使用

```
1 From Coq Require Import ArithRing.
2 From Coq Require Import Compare_dec.
3 From Coq Require Import Wf_nat.
4 From Coq Require Import Arith.
5 From Coq Require Import Lia.
```

格式和 Python 基本一样，这些库我们可以在“安装目录/lib/coq/theories”下找到，在 Coq 中此处相当于引入一些前置的定理/定义，具体是什么，我们根据后面的运用来讨论。

```
1 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
2   intros a; elim a; auto.
3   intros n' Hrec b; case b; auto.
4 Qed.
5
6 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
7   intros x; ring.
8 Qed.
9
10 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
11   unfold not in |- *; intros x y H1; elim (Nat.lt_irrefl x);
12     pattern x at 2 in |- *; rewrite H1; auto.
13 Qed.
```

最先进入眼里的三个定理证明，没错是三个。根据官方文档

Assertions and proofs

An assertion states a proposition (or a type) for which the proof (or an inhabitant of the type) is interactively built using [tactics](#). Assertions cause Coq to enter [proof mode](#) (see [Proof mode](#)). Common tactics are described in the [Basic proof writing](#) chapter. The basic assertion command is:

`Command thm_token ident_decl binder* : type with ident_decl binder* : type*`

```
thm_token ::= Theorem
           | Lemma
           | Fact
           | Remark
           | Corollary
           | Proposition
           | Property
```

After the statement is asserted, Coq needs a proof. Once a proof of `type` under the assumptions represented by `binders` is given and validated, the proof is generalized into a proof of `forall binder*, type` and the theorem is bound to the name `ident` in the global environment.

These commands accept the `program` attribute. See [Program Lemma](#).

我们写 Theorem(定理)、Lemma(引理)、Fact(事实)、Remark(注记)、Corollary(推论)、Proposition(命题)、Property(性质)都是进入证明模式，而 Qed 则是来表示证明模式结束的意思，当然确实要证明结束才行，否则是运行不了 Qed 的。几乎所有定理的写法都是

Theorem 定理命名: forall 涉及变量: 所属的类型, 命题表述.

前两个属于程序特有，从 `forall` 开始就相当于类型论中的一个判断¹，环境 Δ 由“From ... Require Import ...”引入，第一句中的 `nat` 类型就是由“Coq/Init/Datatypes”库给出，由于其在 `Init` 中，所以无需我们主动引入。`nat` 表示皮亚诺自然数（起始元：0，后继： $n = S^n(0)$ ），这样上面三个定理的含义如下

$$\forall a, b, c \in \mathbb{N}, a - b - c = a - (b + c)$$

$$\forall x \in \mathbb{N}, 2x = x + x$$

$$\forall x, y \in \mathbb{N}, x < y \Rightarrow x \neq y$$

虽然一看都很显然，但我们还是得看看它们证明了什么。我们先来看第一个证明，Coq 是指令式运行的，有点像汇编，其中分号“;”和句号“.”在观看上的作用是一样的，只是运行的时候，每一次都是到句号“.”。为了理解其运行原理，我们先全改成句号式。

```

21 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
22 intros a.
23 elim a.
24 auto.
25 intros n' Hrec b.
26 case b;auto.
27 Qed.
28
29 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
30 | intros x; ring.
31 | Qed.
32
33 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
34 | unfold not in |- *; intros x y H; elim (Nat.lt_irrefl x);
35 | pattern x at 2 in |- *; rewrite H; auto.
36 | Qed.
37
38
39 #[local] Hint Resolve lt_neq : core.
40

```

当我们运行定理申明以后，视图中会多一句话，用来表示我们当前的证明目标。下一步执行“intros a”

```

21 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
22 intros a.
23 elim a.
24 auto.
25 intros n' Hrec b.
26 case b;auto.
27 Qed.
28
29 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
30 | intros x; ring.
31 | Qed.
32
33

```

实际就是进行了一次目标转化，将原来要证的判断

$$\Delta; \emptyset \vdash \Pi a, b, c : \mathbb{N}. a - b - c = a - (b + c)$$

转化成了判断

$$\Delta; a : \mathbb{N} \vdash \Pi b, c : \mathbb{N}. a - b - c = a - (b + c)$$

因此视图竖线上方表示上下文 Γ ，intro/intros 的作用就是把 Π 中的非自由变量，转化为上下文中的自由变量，这实际是类型论中的 $\beta\delta$ -约化

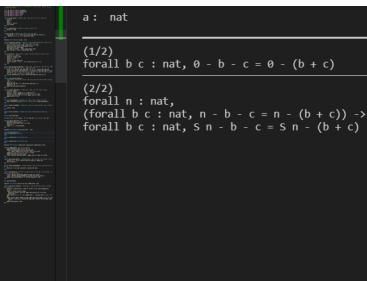
```

20 From Coq Require Import Lia.
21
22 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
23 intros a.
24 intros b.
25 intros c.
26 elim a.
27 auto.
28 intros n' Hrec b.
29 case b;auto.
30 Qed.
31
32 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
33 | intros x; ring.
34 | Qed.
35

```

¹即 $\Pi x, y, z : A. B$ ，其中 $A \equiv \mathbb{N}$, $B \equiv a - b - c = a - (b + c)$ ，加号“+”、等号“=”、减号“-”都是自然数中的定义

我们还可以直接三次约化变成一个无量词命题。下一步执行“elim a”



```

21 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
22   intros a.
23   elim a.
24   auto.
25   intros n' Hrec b.
26   intro.
27   case b;auto.
28 Qed.
29
30 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
31   intros x; ring.
32 Qed.
33
34 Theorem lt_neq : forall x y : nat, x < y -> x < y.
35   unfold not in |- *; intros x y H1; elim (Nat.lt_irrefl x);
36   pattern x at 2 in |- *; rewrite H1; auto.
37 Qed.
38

```

即对类型为 nat 的自由变量 a 进行归纳，这会根据类型的不同导出不同的结果



```

E : Coq-Platform > lib > coq > theories > Init > Datatypes.v
155 (** [nat] is the datatype of natural numbers built from [0] and successor
156 [S];
157
158 ∨ (** [nat] is the datatype of natural numbers built from [0] and successor
159 [S];
160   note that the constructor name is the letter O.
161   Numbers in [nat] can be denoted using a decimal notation;
162   e.g. [3%nat] abbreviates [S (S (S O))] *)
163 ∨ Inductive nat : Set := 
164   | O : nat
165   | S : nat -> nat.
166

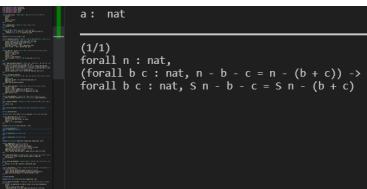
```

皮亚诺自然数主要是初始元 0 和后继 S，因此要证明的命题被拆成了两个

$$\Delta; a : \mathbb{N} \vdash \Pi b, c : \mathbb{N}. 0 - b - c = 0 - (b + c)$$

$$\Delta; a : \mathbb{N} \vdash \Pi n : \mathbb{N}. ((\Pi b, c : \mathbb{N}. n - b - c = n - (b + c)) \rightarrow (\Pi b, c : \mathbb{N}. S(n) - b - c = S(n) - (b + c)))$$

下一步执行“auto”，即自动证明



```

21 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
22   intros a.
23   elim a.
24   auto.
25   intros n' Hrec b.
26   intro.
27   case b;auto.
28 Qed.
29
30 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
31   intros x; ring.
32 Qed.
33
34 Theorem lt_neq : forall x y : nat, x < y -> x < y.
35   unfold not in |- *; intros x y H1; elim (Nat.lt_irrefl x);
36   pattern x at 2 in |- *; rewrite H1; auto.
37 Qed.
38

```

可以看到直接消去了归纳的初始步。这可不是什么自动证明，还没有如此高级的东西

Tactic auto `nat_or_var?` `auto_using?` `hintbases?` %

auto_using ::= using `one_term` +
hintbases ::= with *
| with `ident` +

Implements a Prolog-like resolution procedure to solve the current goal. It first tries to solve the goal using the `assumption` tactic, then it reduces the goal to an atomic one using `intros` and introduces the newly generated hypotheses as hints. Then it looks at the list of tactics associated with the head symbol of the goal and tries to apply one of them. Lower cost tactics are tried before higher-cost tactics. This process is recursively applied to the generated subgoals.

nat_or_var
Specifies the maximum search depth. The default is 5.

using `one_term` +
Uses lemmas `one_term` + in addition to hints. If `one_term` is an inductive type, the collection of its constructors are added as hints.

Note that hints passed through the `using` clause are used in the same way as if they were passed through a hint database. Consequently, they use a weaker version of `apply` and `auto`. `using one_term` may fail where `apply one_term` succeeds.

with *

根据官方文档的说明，实际就是根据环境和上下文消去一些简单的东西，复杂的东西它可完成不了，那么

$$\forall b, c \in \mathbb{N}, 0 - b - c = 0 - (b + c)$$

是个简单的结论吗？我们先来看这几个运算的定义

```
E: > Coq-Platform > lib > coq > theories > Init > ≡ Nat.v
43   end.
44
45 Register pred as num.nat.pred.
46
47 Fixpoint add n m :=
48   match n with
49   | 0 => m
50   | S p => S (p + m)
51   end
52
53 where "n + m" := (add n m) : nat_scope.
54
55 Register add as num.nat.add.
56
57 Definition double n := n + n.
58
59 Fixpoint mul n m :=
60   match n with
61   | 0 => 0
62   | S p => m + p * m
63   end
64
65 where "n * m" := (mul n m) : nat_scope.
66
67 Register mul as num.nat.mul.
68
69 (** ** Constants *)
70
71 Local Notation "0" := 0.
72 Local Notation "1" := (S 0).
73 Local Notation "2" := (S (S 0)).
74
75 Definition zero := 0.
76 Definition one := 1.
77 Definition two := 2.
78
79 (** ** Basic operations *)
80
81 Definition succ := S.
82
83 ∨ Definition pred n :=
84   (** Truncated subtraction: [n-m] is [0] if [n<=m] *)
85
86 Fixpoint sub n m :=
87   match n, m with
88   | S k, S l => k - l
89   | _, _ => n
90   end
91
92 where "n - m" := (sub n m) : nat_scope.
93
94 Register sub as num.nat.sub.
```

它们都是递归的，对于 $n+m$ ，把 n 回退成 0 后进行 n 次后继即是结果，对于 $n-m$ ，则是 m 与 n 一起往 0 靠，如果 n 先到 0 就让结果为零，如果 m 到 0 就让此时的 n 作为结果，因此根据定义有 $0-b-c = 0-c = 0 = 0-(b+c)$ ，所以在 Coq 看来起始步可以直接完成。下一步执行“intros n' Hrec b”

```
20 From Coq Require Import Ltac.
21
22 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
23 intros a.
24 elim a.
25 auto.
26 intros n' Hrec b.
27 case b;auto.
28 Qed.
29
30 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
31 intros x; ring.
32 Qed.
33
34 Theorem lt_neq : forall x y : nat, x < y -> x < y.
35 unfold not in |- *; intros x y H H1; elim (Nat.lt_irrefl x);
36 pattern x at 2 in |- *; rewrite H1; auto.
37 Qed.
```

这时你可能会感慨，变化也太大了吧。首先我们要清楚，指令后的参数只是为了起上下文变量名用的

```
20 From Coq Require Import Ltac.
21
22 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
23 intros one.
24 elim one.
25 auto.
26 intros two three four.
27 case four;auto.
28 Qed.
29
30 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
31 intros x; ring.
32 Qed.
33
34 Theorem lt_neq : forall x y : nat, x < y -> x < y.
35 unfold not in |- *; intros x y H H1; elim (Nat.lt_irrefl x);
36 pattern x at 2 in |- *; rewrite H1; auto.
37 Qed.
```

只要格式符合，我们同样能完成证明。实在搞不懂的话，我们就把三次脱壳分开写

The screenshot shows three separate proof sessions in the CoqIDE. Each session starts with the same code:

```

22 Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
23   intros a.
24   elim a.
25   auto.
26   intros n'.
27   intros Hrec.
28   intros b.
29   case b;auto.
30 Qed.

```

Session 1 (top): The proof is completed with a single step. The right panel shows the goal and its simplification:

$$\frac{}{(1/1) \forall b c : \text{nat}, n' - b - c = n' - (b + c) \rightarrow \forall b c : \text{nat}, S n' - b - c = S n' - (b + c)}$$

Session 2 (middle): The proof is completed with two steps. The right panel shows the goal and its simplification:

$$\frac{\begin{array}{l} a, n' : \text{nat} \\ Hrec : \forall b c : \text{nat}, n' - b - c = n' - (b + c) \end{array}}{(1/1) \forall b c : \text{nat}, S n' - b - c = S n' - (b + c)}$$

Session 3 (bottom): The proof is completed with three steps. The right panel shows the goal and its simplification:

$$\frac{\begin{array}{l} a, n' : \text{nat} \\ Hrec : \forall b c : \text{nat}, n' - b - c = n' - (b + c) \\ b : \text{nat} \end{array}}{(1/1) \forall c : \text{nat}, S n' - b - c = S n' - (b + c)}$$

可以看到，第一次我们前置了 n 的全称量词，第二次我们前置了箭头式 $A \rightarrow B$ 的前提，第三次我们又前置了 b 的全称量词，从直觉上，这些都是合情合理的，所以也无需讲太多。下一步执行“case b”

The screenshot shows the proof after executing “case b”. The right panel shows the goal and its simplification:

$$\frac{\begin{array}{l} a, n' : \text{nat} \\ Hrec : \forall b c : \text{nat}, n' - b - c = n' - (b + c) \\ b : \text{nat} \end{array}}{(1/2) \forall c : \text{nat}, S n' - 0 - c = S n' - (0 + c)}$$

Below this, another branch is shown:

$$(2/2) \forall n c : \text{nat}, S n' - S n - c = S n' - (S n + c)$$

一眼望去似乎和 `elim` 一样做了一次归纳，但实际只是对 b 进行了自动分类讨论，它从定义上有两种情况，一是作为初始元 0，二是作为所有自然数的后继 $S(n)$ ，后者是不包含零情况的。最后两个“auto”消去这两个简单结论，从而完成证明

The screenshot shows the final completed proof. The right panel shows the goal and its simplification:

$$\frac{\begin{array}{l} a, n' : \text{nat} \\ Hrec : \forall b c : \text{nat}, n' - b - c = n' - (b + c) \\ b : \text{nat} \end{array}}{(1/1) \forall n c : \text{nat}, S n' - S n - c = S n' - (S n + c)}$$

Below this, a message indicates the proof is finished:

Proof finished

从定义上是很容易验证的

$$S(n1) - 0 - c = S(n1) - c = S(n1) - (0 + c)$$

$$S(n1) - S(n) - c = \begin{cases} 0 - c = 0 = S(n1) - S(n + c) = S(n1) - (S(n) + c) & n1 < n \\ S(n1) - S(n + c) = S(n1) - (S(n) + c) & n1 \geq n \end{cases}$$

我们似乎在第一个定理的讲解上使用了过多的笔墨，想必你对 Coq 的形式化证明已经有了一定的感觉，这意味着预热结束了，接下来要加速了。

```
27 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
28   intros x.
29   ring.
30 Qed.
31
32 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
33   unfold not in |- *; intros x y H HI; elim (Nat.lt_irrefl x);
34   | pattern x at 2 in |- *; rewrite H; auto.
35 Qed.
36
37 Remark expand_mult2 : forall x : nat, 2 * x = x + x.
38   intros x.|_
39   ring.
40 Qed.
41
42 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
43   unfold not in |- *; intros x y H HI; elim (Nat.lt_irrefl x);
44   | pattern x at 2 in |- *; rewrite H; auto.
45 Qed.
46
47 ✓ Remark expand_mult2 : forall x : nat, 2 * x = x + x.
48   intros x.
49   ring.|_
50 Qed.
51
52 ✓ Theorem lt_neq : forall x y : nat, x < y -> x <> y.
53   unfold not in |- *; intros x y H HI; elim (Nat.lt_irrefl x);
54   | pattern x at 2 in |- *; rewrite H; auto.
55 Qed.
```

基础的全称量词前置操作没啥好说的，下一个指令“ring”(来自环境“From Coq Require Import ArithRing”)，实际就是一个环多项式方程验证，我们自己也能做到，根据定义不断递归就行了

$$2*x = S(S(0))*x = S(0)*x + x = 0*x + x + x = x + x$$

下一个定理就复杂了不少

```
32 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
33 unfold not in |- *.
34 intros x y Hlt.
35 elim (Nat.lt_irrefl x).
36 pattern x at 2 in |- *.
37 rewrite Hlt.
38 auto.
39 Qed.

32 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
33 unfold not in |- *.
34 intros x y Hlt.
35 elim (Nat.lt_irrefl x).
36 pattern x at 2 in |- *.
37 rewrite Hlt.
38 auto.
39 Qed.

32 Theorem lt_neq : forall x y : nat, x < y -> x <> y.
33 unfold not in |- *.
34 intros x y Hlt.
35 elim (Nat.lt_irrefl x).
36 pattern x at 2 in |- *.
37 rewrite Hlt.
38 auto.
39 Qed.
```

(1/1)
forall x y : nat, x < y -> x <> y

x, y : nat
H1 : x < y
H2 : x = y

(1/1)
x < x

x, y : nat
H1 : x < y
H2 : x = y

(1/1)
x < y

x, y : nat
H1 : x < y
H2 : x = y

Theorem lt_neq : forall x y : nat, x < y -> x <> y.
unfold not in |- *.
intros x y Hlt.
elim (Nat.lt_irrefl x).
pattern x at 2 in |- *.
rewrite Hlt.
auto.
Qed.

Theorem lt_neq : forall x y : nat, x < y -> x <> y.
unfold not in |- *.
intros x y Hlt.
elim (Nat.lt_irrefl x).
pattern x at 2 in |- *.
rewrite Hlt.
auto.
Qed.

Theorem lt_neq : forall x y : nat, x < y -> x <> y.
unfold not in |- *.
intros x y Hlt.
elim (Nat.lt_irrefl x).
pattern x at 2 in |- *.
rewrite Hlt.
auto.
Qed.

x, y : nat
H1 : x < y
H2 : x = y

(1/1)
(Fun n : nat -> x < n) x

Proof finished

第一句“unfold 定义名称 in 哪里要展开”就是将定义展开的意思，此处涉及两个符号

```
E: > Coq-Platform > lib > coq > theories > Init > ┌ Logic.v
32
33 (** [not A], written  $\neg A$ , is the negation of  $[A]$  *)
34 Definition not (A:Prop) := A -> False.
35
36 Notation " $\neg x$ " := (not x) : type_scope.
37 |
E: > Coq-Platform > lib > coq > theories > Init > ┌ Logic.v
384
385 Notation " $x = y$ " := (eq x y) : type_scope.
386 Notation " $x \neq y$ " := ( $\neg (x = y)$ ) : type_scope.
387 Notation " $x \leftrightarrow y$ " := ( $\neg (x \neq y)$ ) : type_scope.
388
```

由于 Notation 相当于记号缩写，所以我们无需特别申明，它会跟随定义的展开而展开，上述展开在逻辑上的含义为

$$x \neq y \equiv x = y \rightarrow \perp$$

在代码中为了便于计算机阅读，我们把恒假符 \perp 写成 False，实际上对于箭头 \rightarrow ，我们也是用杠加大于构成的 \rightarrow 。下一步进行了四次约化，分别对应两个全称量词和两个箭头，没啥好说的，此时需要根据上下文证明 False，实际就是反证法，说明上下文中有关矛盾的结论。elim 在无参数时表示对变量 x 使用定义，此时我们加了一个定理 $\forall x \in \mathbb{N}, \neg x < x$ （来自环境“From Coq Require Import Arith”）则表示对变量 x 使用定理，而 False 真值对应 $x < x$ 。下一个 pattern 有点复杂

Tactic pattern `pattern_occ`; `occurrences`?

Performs beta-expansion (the inverse of beta-reduction) for the selected hypotheses and/or goals. The `one_terms` in `pattern_occ` must be free subterms in the selected items. The expansion is done for each selected item `T` for a set of `one_terms` in the `pattern_occ` by:

- replacing all selected occurrences of the `one_terms` in `T` with fresh variables
- abstracting these variables
- applying the abstracted goal to the `one_terms`

For instance, if the current goal `T` is expressible as $\phi(t_1 \dots t_n)$ where the notation captures all the instances of the t_i in ϕ , then `pattern t1 ... tn` generates the equivalent goal $(\text{fun } (x_1:A_1) \dots (x_n:A_n) \Rightarrow \phi(x_1 \dots x_n)) t_1 \dots t_n$. If t_i occurs in one of the generated types A_j (for $j > i$), occurrences will also be considered and possibly abstracted.

This tactic can be used, for instance, when the tactic `apply` fails on matching or to better control the behavior of `rewrite`.

See the example [here](#).

它表示反向 β -约化，这里的 `fun` 对应类型论中的 λ ，即此处给出了

$$(\lambda n : \mathbb{N}. x < n)x \rightarrow_{\beta} x < x$$

这里的 2 表示我们把式子中的第 2 个 `x` 转化为函数的变量。接着通过 `rewrite` 进行项替换，此处把自由变量 `x` 替换成 `y`，即逻辑中的

$$(\lambda n : \mathbb{N}. x < n)x[x := y] \equiv (\lambda n : \mathbb{N}. x < n)y \rightarrow_{\beta} x < y$$

最后结论在上下文中，通过 `auto` 即可自动验证结束。

```

1 #[local] Hint Resolve lt_neq : core.

2

3 Theorem monotonic_inverse : forall f : nat -> nat, (forall x y : nat, x < y -> f x < f y) ->forall x y
4   : nat, f x < f y -> x < y.
5   intros f Hmon x y Hlt; case (le_gt_dec y x); auto.
6   intros Hle; apply Nat.lt_eq_cases in Hle.
7   destruct Hle as [Hlt'|Heq].
8   elim (Nat.lt_asymm _ _ Hlt'); apply Hmon; auto.
9   elim (lt_neq _ _ Hlt); rewrite Heq; auto.
10 Qed.

11 Theorem mult_lt : forall a b c : nat, c <> 0 -> a < b -> a * c < b * c.
12   intros a b c; elim c.
13   intros H; elim H; auto.
14   intros c'; case c'.
15   intros; lia.
16   intros c'' Hrec Hneq Hlt;
17   repeat rewrite <- (fun x : nat => mult_n_Sm x (S c'')). 
18   lia.
19 Qed.

20

21 Remark add_sub_square_identity : forall a b : nat, (b + a - b) * (b + a - b) = (b + a) * (b + a) + b *
22   b - 2 * ((b + a) * b).
23   intros a b; rewrite (Nat.add_comm b a) at 1 2; rewrite Nat.add_sub.
24   repeat rewrite Nat.mul_add_distr_r || rewrite <- (Nat.mul_comm (b + a)).
25   replace (b * b + a * b + (b * a + a * a) + b * b) with (b * b + a * b + (b * b + a * b + a * a));
     try (ring; fail).
   rewrite expand_mult2; repeat rewrite Nat.add_sub; auto with *.

```

26

Qed.

同样是三个定理，把它们转化为通俗语言就是

$$\forall f : \mathbb{N} \rightarrow \mathbb{N}, x < y \rightarrow f(x) < f(y) \Rightarrow f(x) < f(y) \rightarrow x < y$$

$$\forall a, b, c \in \mathbb{N}, c \neq 0, a < b \Rightarrow ac < bc$$

$$\forall a, b \in \mathbb{N}, (b + a - b)^2 = (b + a)^2 - 2(b + a)b + b^2$$

都是直觉上的显然结论。第 1 行 “Hint Resolve” 是与 auto 相关的指令，前面的 “[local]” 表示只在当前的文件生效，详细看[官方说明](#)，总之目的就是为了使用前面已经证明了的定理。第 4 行首先进行了 5 次 intros，分别对应全称量词、箭头、全称量词 x2、箭头，引入上下文

$$\Gamma \equiv f : \mathbb{N} \rightarrow \mathbb{N}, \text{Hmon} : \Pi x, y : \mathbb{N}. x < y \rightarrow f(x) < f(y), x : \mathbb{N}, y : \mathbb{N}, \text{Hlt} : f(x) < f(y)$$

则输出判断为

$$\Gamma \vdash x < y$$

接着针对变量 x 和 y 使用性质 le_gt_dec 进行分类，让后自动消去。这里的 le_gt_dec 要求两个变量，含义为 “ $\forall x, y \in \mathbb{N}, y \leq x \vee y > x$ ”，因此分类后的结果为

$$\Gamma \vdash y \leq x \rightarrow x < y \quad \Gamma \vdash y > x \rightarrow x < y$$

通过 auto 可以消去后面的自反性，从而只剩下前一种情况了。第 5 行首先通过 intros 将箭头移开

$$\Gamma, \text{Hle} : y \leq x \vdash x < y$$

接着对 Hle 使用性质 Nat.lt_eq_cases，其含义为 “ $\forall x, y \in \mathbb{N}, y \leq x \leftrightarrow y < x \vee y = x$ ”，即把小等号展开。第 6 行的 destruct，作用相当于 case，即对 Hle 进行分类讨论

$$\Gamma_1 \equiv \Gamma, \text{Hlt}' : y < x \vdash x < y \quad \Gamma_2 \equiv \Gamma, \text{Heq} : y = x \vdash x < y$$

而 destruct 会进行变量约简。接下来的两行分别解决两种情况，第 7 行的 Nat.lt_asymm 表示不等式的自反性 “ $x < y \leftrightarrow \neg y < x$ ”，两条横线表示不对性质输入参数，而直接作用于 Hlt，因此可得

$$\Gamma_1 \vdash x < y \equiv \perp \equiv f(y) < f(x)$$

接着 “apply Hmon” 表示使用上下文中的 Hmon 命题，从而可推出

$$\Gamma_1 \vdash y < x$$

结论在上下文中 auto 结束。第 8 行也是类似的，lt_neq 表示小于推不等 “ $x < y \rightarrow x \neq y$ ”，作用于 Hlt 可得

$$\Gamma_2 \vdash x < y \Rightarrow \Gamma_2 \vdash \neg x = y \equiv \perp \equiv f(x) = f(y)$$

接着通过 rewrite 自由代入 Heq 可得

$$\Gamma \vdash (f(x) = f(y))[y := x] \equiv f(x) = f(x)$$

最后 auto 验证结束。接着看第二个定理，第 12 行先将三个全称量词前置，接着 elim 对 c 使用定义，即归纳法。此时我们会默认先证明第一个基础步目标

$$\Gamma \equiv a : \mathbb{N}, b : \mathbb{N}, c : \mathbb{N} \vdash 0 \neq 0 \rightarrow a < b \rightarrow a0 < b0$$

第 13 行一串基本操作证明基础步

$$\Gamma, H : 0 \neq 0 \vdash a < b \rightarrow a0 < b0 \Rightarrow \Gamma, H : 0 \neq 0 \vdash 0 = 0$$

此时只剩归纳步了

$$\Gamma \vdash \Pi n : \mathbb{N}. ((n \neq 0 \rightarrow a < b \rightarrow ac < bc) \rightarrow (S(n) \neq 0 \rightarrow a < b \rightarrow aS(n) < bS(n)))$$

第 14 行，我们前置全称量词，再对其进行归纳，于是又分出了基础步和归纳步，对于基础步

$$\Gamma_0 \equiv \Gamma, c' : \mathbb{N} \vdash (0 \neq 0 \rightarrow a < b \rightarrow a0 < b0) \rightarrow (1 \neq 0 \rightarrow a < b \rightarrow a1 < b1)$$

第 15 行通过单个 intros 会将可前置的全部前置，即 4 个箭头，值得注意的是 $0 \neq 0 \rightarrow a < b \rightarrow a0 < b0$ 不属于第一层箭头会被整体前置到上下文中

$$\Gamma_0, H : 0 \neq 0 \rightarrow a < b \rightarrow a0 < b0, H1 : 1 \neq 0, H2 : a < b \vdash a1 < b1$$

紧跟的 lia(来自环境 “From Coq Require Import Lia”)，即线性自动求解，其会化为定义自动消去基础步

$$a1 < b1 \Leftrightarrow aS(0) < bS(1) \Leftrightarrow a0 + a < b0 + b \Leftrightarrow 0 + a < 0 + b \Leftrightarrow a < b$$

于是我们还剩下新的归纳步没有解决

$$\Gamma_0 \vdash \Pi n : \mathbb{N}. ((S(n) \neq 0 \rightarrow a < b \rightarrow S(n) < bS(n)) \rightarrow (S(S(n)) \neq 0 \rightarrow a < b \rightarrow aS(S(n)) < bS(S(n))))$$

第 16 行的 intros 前置了一个全称量词和三个箭头，此时环境为 $\Gamma_1 \equiv \Gamma_0, c'' : \mathbb{N}, \text{Hrec} : S(c'') \neq 0 \rightarrow a < b \rightarrow S(c'') < bS(c''), \text{Hneq} : S(S(c'')) \neq 0, \text{Hlt} : a < b$ ，要推出的判断为

$$\Gamma_1 \vdash aS(S(c'')) < bS(S(c''))$$

第 17 行的 repeat，不用多想就是重复的意思，其结束条件为错误或者输出没有改变的时候，此处执行 rewrite，将乘法按定义写出，即

$$a * S(S(c'')) = a * S(c'') + a$$

由于后面的限制，我们只化到 $a * S(c'')$ ，因此这里实际执行了两次 rewrite，这意味着此处可以用两句 rewrite 来代替。第 18 行的 lia 和上面一样地进行线性自动求解，实际上我们无需执行 “repeat rewrite”，现在的 Coq 版本可以直接通过 lia 自动化简并验证求解。

```

45
46 Theorem mult_lt : forall a b c : nat, c > 0 -> a < b -> a * c < b * c.
47   intros a b c; elim c.
48   intros H; elim H; auto.
49   intros c'; case c'.
50   intros; lia.
51   intros c'' Hrec Hneq Hlt;
52   lia.
53 Qed.
54

```

The screenshot shows the Coq proof assistant interface. On the left, the proof script is displayed with numbered lines 45 through 54. Lines 45-53 correspond to the proof of the theorem, and line 54 shows the command 'Qed.'. On the right, the proof state is shown with a green bar indicating progress. The text 'Proof finished' is visible at the top right of the proof area.

最后看第三个定理，第 22 行开始先来一个 intros 进行全称量词前置，接着通过加法交换律 Nat.add_comm 重写第 1 和第 2 个 “ $b+a$ ” 可得

$$a : \mathbb{N}, b : \mathbb{N} \vdash (a + b - b)(a + b - b) = (b + a)(b + a) + bb - 2((b + a)b)$$

然后通过加减性质 Nat.add_sub “ $a+b-b=a$ ” 再次复写可得

$$a : \mathbb{N}, b : \mathbb{N} \vdash aa = (b + a)(b + a) + bb - 2((b + a)b)$$

第 23 行使用分配律 Nat.mul_add_distr_r 和乘法交换律 Nat.add_comm 重复复写，主要是等式后面，从而可得

$$a : \mathbb{N}, b : \mathbb{N} \vdash aa = bb + ab + (ba + aa) + bb - 2(bb + ab)$$

这里之所以有个括号是因为我们没有使用结合律

$$(b+a)(b+a) = b(b+a) + a(b+a) = (b+a)b + a(b+c) = bb + ab + a(b+c) = bb + ab + (b+c)a = bb + ab + (ba + aa)$$

此处的执行过程为“分配-> 交换-> 分配-> 交换-> 分配”，之所以要交换再分配是因为定理 Nat.mul_add_distr_r 实际是右分配律。第 24 行先执行 “replace term1 with term2”，即把要证结论中的 term2 换成 term1，当然乱换肯定是不行的，因此我们会产生一个需要证明的子结论

$$a : \mathbb{N}, b : \mathbb{N} \vdash \text{term1} = \text{term2}$$

此时通过 ring 将这个结论消去。于是要证的结论变成了

$$a : \mathbb{N}, b : \mathbb{N} \vdash aa = bb + ab + (bb + ab + aa) - 2(bb + ab)$$

最后是 25 行，先使用前面证明的 expand_mult2 “ $2x=x+x$ ” 进行第一次复写

$$bb + ab + (bb + ab + aa) - 2(bb + ab) \equiv bb + ab + (bb + ab + aa) - (bb + ab + (bb + ab))$$

下一步复写没什么作用，因为式子中看不到 “ $m+n-n$ ” 形式的句子，最后的 “auto with *” 表示启动最大功率的约化，显然结论只要通过结合律加交换律就能完成，都是预设内容。如今的你，对于 Coq 形式化证明应该已经如火纯青了，所以我们要开启最大功率了，为了给后面的内容留点空间。

```

1 Theorem sub_square_identity : forall a b : nat, b <= a -> (a - b) * (a - b) = a * a + b * b - 2 * (a *
2   b).
3   intros a b H.
4   apply Nat.sub_add in H; rewrite Nat.add_comm in H.
5   rewrite <- H.
6   apply add_sub_square_identity.
7   Qed.
8
9 Theorem square_monotonic : forall x y : nat, x < y -> x * x < y * y.
10  intros x; case x.
11  intros y; case y; simpl in |- *; auto with *.
12  intros x' y Hlt; apply Nat.lt_trans with (S x' * y).
13  rewrite (Nat.mul_comm (S x') y); apply mult_lt; auto.
14  apply mult_lt; lia.
15  Qed.
```

```

15
16 Theorem root_monotonic : forall x y : nat, x * x < y * y -> x < y.
17   exact (monotonic_inverse (fun x : nat => x * x) square_monotonic).
18 Qed.
19
20 Remark square_recompose : forall x y : nat, x * y * (x * y) = x * x * (y * y).
21   intros; ring.
22 Qed.
23
24 Remark mult2_recompose : forall x y : nat, x * (2 * y) = x * 2 * y.
25   intros; ring.
26 Qed.

```

这里有 5 条定理，分别对应 ($\forall a, b, x, y \in \mathbb{N}$)

$$b \leq a \Rightarrow (a - b)^2 = a^2 + b^2 - 2ab$$

$$x \leq y \Rightarrow x^2 \leq y^2$$

$$x^2 \leq y^2 \Rightarrow x \leq y$$

$$xy(xy) = xx(yy)$$

$$x(2y) = x2y$$

定理本身在直觉上都是显然的，过程也是我们熟知的，唯一要提的是第三个定理的第 17 行使用了 `exact`，即对后面的项相乘使用 β -约化

$$\begin{aligned} & (\Pi f : \mathbb{N} \rightarrow \mathbb{N}. (\Pi x, y : \mathbb{N}, x < y \rightarrow f(x) < f(y)) \rightarrow \Pi x, y : \mathbb{N}, f(x) < f(y) \rightarrow x < y) (\lambda x : \mathbb{N}. x^2) (\Pi x, y : \mathbb{N}, x < y \rightarrow x^2 < y^2) \\ & \rightarrow_{\beta} ((\Pi x, y : \mathbb{N}, x < y \rightarrow x^2 < y^2) \rightarrow \Pi x, y : \mathbb{N}, x^2 < y^2 \rightarrow x < y) (\Pi x, y : \mathbb{N}, x < y \rightarrow x^2 < y^2) \\ & \rightarrow_{\beta} \Pi x, y : \mathbb{N}, x^2 < y^2 \rightarrow x < y \end{aligned}$$

从而得到我们需要的结论。接着有一段

```

Section sqrt2_decrease.

...
End sqrt2_decrease.

```

它将一部分内容给圈起来，作用主要是好使用这块区域的全局变量，而不污染整个证明的全局变量，比如紧接着它进行了变量的引入

```
Variables (p q : nat) (pos_q : 0 < q) (hyp_sqrt : p * p = 2 * (q * q)).
```

总共引入了四个共同的上下文变量 `p,q`, `pos_q` 和 `hyp_sqrt`。然后就是区域中的大块证明

```

1 Theorem sqrt_q_non_zero : 0 <> q * q.
2   generalize pos_q; case q.
3   intros H; elim (Nat.nlt_0_r 0); auto.
4   intros n H.
5   simpl in |- *; discriminate.
6 Qed.

```

```

7
8 #[local] Hint Resolve sqrt_q_non_zero : core.
9
10 Ltac solve_comparison :=
11   apply root monotonic; repeat rewrite square_recompose; rewrite hyp_sqrt;
12   rewrite mult2_recompose; apply mult_lt; auto with arith.
13
14 Theorem comparison1 : q < p.
15   replace q with (1 * q); try ring.
16   replace p with (1 * p); try ring.
17   solve_comparison.
18 Qed.
19
20 Theorem comparison2 : 2 * p < 3 * q.
21   solve_comparison.
22 Qed.
23
24 Theorem comparison3 : 4 * q < 3 * p.
25   solve_comparison.
26 Qed.
27
28 #[local] Hint Resolve comparison1 comparison2 comparison3: arith.
29
30 Theorem comparison4 : 3 * q - 2 * p < q.
31   apply Nat.add_lt_mono_l with (2 * p).
32   rewrite Nat.add_comm; rewrite Nat.sub_add;
33   try (simple apply Nat.lt_le_incl; auto with arith).
34   replace (3 * q) with (2 * q + q); try ring.
35   apply Nat.add_lt_le_mono; auto.
36   repeat rewrite (Nat.mul_comm 2); apply mult_lt; auto with arith.
37 Qed.
38
39 Remark mult_minus_distr_l : forall a b c : nat, a * (b - c) = a * b - a * c.
40   intros a b c; repeat rewrite (Nat.mul_comm a); apply Nat.mul_sub_distr_r.
41 Qed.
42
43 Remark minus_eq_decompose : forall a b c d : nat, a = b -> c = d -> a - c = b - d.
44   intros a b c d H H0; rewrite H; rewrite H0; auto.
45 Qed.
46
47 Theorem new_equality : (3 * p - 4 * q) * (3 * p - 4 * q) = 2 * ((3 * q - 2 * p) * (3 * q - 2 * p)).
48   repeat rewrite sub_square_identity; auto with arith.
49   repeat rewrite square_recompose; rewrite mult_minus_distr_l.
50   apply minus_eq_decompose; try rewrite hyp_sqrt; ring.
51 Qed.

```

主要证明了一个不等式 comparison4 和一个等式 new_equality，写成一般的形式是

$$p, q \in \mathbb{N}, q > 0, p^2 = 2q^2 + 3q - 2p < q$$

$$p, q \in \mathbb{N}, q > 0, p^2 = 2q^2 \vdash (3p - 4q)^2 = 2(3q - 2p)^2$$

上面大多内容，想必你一看就懂，所以我就稍微提几个关键点。第2行的“generalize pos_q”表示使用条件pos_q来将结论一般化，即将结论A变成“pos_q推出A”。第5行的“simpl in |- *”表示在结论中进行一定程度的可读性转化，本质上没起作用，删掉照样可以运行，而紧跟其后的“discriminate”表示等号验证，虽然预证结论是不等号，但展开后就是等号

$$0 \neq S(n)S(n) \equiv \neg 0 = S(n)S(n)$$

第10行的“Ltac名称(solve_comparison):=证明片段”是证明片段的引入，后面我们只需执行“solve_comparison”相当于运行了一次证明片段，后面的三个定理都用了类似的证明，所以在前面进行了一次证明片段简称。其它都是常规操作，就没啥好说的了，此时我们的主角终于登场了。

```

1 Theorem sqrt2_not_rational : forall p q : nat, q <> 0 -> p * p = 2 * (q * q) -> False.
2   intros p q; generalize p; clear p; elim q using (well_founded_ind lt_wf).
3   clear q; intros q Hrec p Hneq;
4   pose proof Hneq as Hlt_0_q; apply Nat.neq_0_lt_0 in Hlt_0_q;
5   intros Heq.
6   apply (Hrec (3 * q - 2 * p) (comparison4 _ _ Hlt_0_q Heq) (3 * p - 4 * q)).
7   apply sym_not_equal; apply lt_neq; apply Nat.add_lt_mono_l with (2 * p);
8   rewrite <- plus_n_0; rewrite Nat.add_comm; rewrite Nat.sub_add; auto with *.
9   apply new_equality; auto.
10 Qed.
```

首先“根号2是无理数”的形式表述为

$$\forall p, q \in \mathbb{N}, q \neq 0 \Rightarrow p^2 \neq 2q^2$$

由于在环境中，我们没有有理数的概念，所以命题必需要以自然数的形式进行叙述，此处的“A ≠ B”可以进行多次展开

$$A \neq B \equiv \neg A = B \equiv A = B \rightarrow \perp$$

于是就得到了上面的表述。里面的每个指令都是熟悉的，但这里的证法与传统的不同，由于互质的概念不在环境和上下文中，所以假设(p,q)=1，然后推出矛盾(p,q)=2是不行的，因此我来稍微把上面的内容翻译成你懂的语言，或许这样你就能把形式证明读得更加顺畅了。

证明 [证明翻译] 前提: $p, q \in \mathbb{N}, q \neq 0$

结论: $p^2 \neq 2q^2$

假设结论不成立，从而引入前提 $p^2 = 2q^2$ 。此时我们可以得到

$$\begin{aligned} p^2 = 2q^2 &\Rightarrow q < p \Rightarrow 3q - 2p < q \\ (3p - 4q)^2 &= 9p^2 - 24pq + 16q^2 = 18q^2 - 24pq + 8p^2 = 2(3q - 2p)^2 \end{aligned}$$

这意味着我们可以找到两个整数 $m = 3p - 4q, n = 3q - 2p < q$ 使得

$$m^2 = 2n^2$$

从而我们可以反复得到

$$m_i^2 = 2n_i^2, n_i < n_{i-1}$$

由于自然数是有下界的，所以不可能一直进行下去，从而矛盾，定理得证。

实际上，我们相当于证明了不定方程 $p^2 = 2q^2$ 没有正整数解，使用的方法是无穷下降法，这对应第 2 行中的 well_founded_ind 性质²。这么看环境 “From Coq Require Import Compare_dec” 和 “From Coq Require Import Wf_nat” 没有起到任何作用，实际上删除以后可以照样运行。

标准库一览

在证明中，环境或者证明依赖的定义和定理是十分重要的，这意味着我们需要在什么层次上证明这个定理，因此了解 Coq 给我们提供了什么是十分重要的，即 Coq 的标准库有哪些？我们并不推荐看官方的资料，而推荐看“安装路径/lib/coq/theories”下的内容，一方面其比较完整，另一方面官方资料有的东西在这里也有，但反过来就不一定了。

首先 “Init” 中的文件会默认加载，无需我们特别引入，其主要提供了我们需要用到的一些基本类型，不过像比特 byte、浮点数 decimal(hexadecimal) 之类计算机相关的类型我们基本用不到，真正值得注意的是 Datatypes、Logic、Nat、Peano 所提供的类型，在第一个文件中，我们得到了布尔和自然数这两个重要的类型，在第二个文件中，我们得到了各种逻辑表达式，在第三个文件中我们定义了自然数中的各种基本运算，第四个文件完善了自然数的皮亚诺公理，那集合呢？就如我们对 λD 系统的认知一样，它是理论自带的基础类型，类型 \square 、命题 $*_p$ 、集合 $*_s$ ，只不过在 Coq 中使用名称写出类型

类型：Type、命题：Prop、集合：Set

因此光一个初始 Init，就已经包含了类型论的基本内容，并且还引入了皮亚诺算术和集合论这两个数学基础，我们实际已经可以做不少事了，但这还不够，因为大量的基本性质还没有引入，这也就有了其它的库。比如我们在证明“根号 2 是无理数”的代码中，所用到的下面这些性质

$$\begin{aligned}
 & (\text{Nat.lt_irrefl}) \forall x \in \mathbb{N}, \neg x < x \\
 & (\text{Nat.lt_eq_cases}) \forall m, n \in \mathbb{N}, n \leq m \leftrightarrow n < m \vee n = m \\
 & (\text{Nat.lt_asymm}) \forall m, n \in \mathbb{N}, m < n \rightarrow \neg n < m \\
 & (\text{Nat.add_comm}) \forall m, n \in \mathbb{N}, m + n = n + m \\
 & (\text{Nat.mul_add_distr_r}) \forall m, n, p \in \mathbb{N}, (m + n)p = mp + np \\
 & (\text{Nat.add_sub}) \forall m, n \in \mathbb{N}, n + m - m = n \\
 & \dots
 \end{aligned}$$

都来自 “Arith.Arith.v” 库，实际上 Arith 包下的库都是有关算术的，里面有一个除 2 库 Div2 和偶数库 Even，还有欧几里得辗转相除库 Euclid 和良序关系库 Wf_nat，不知是不是太难形式化了，我们并没有看到整除相关的库。我个人觉得 Coq 的计算机气息有点太重了，比如各种二进制算术库 PArith、NArith、ZArith，个人觉得在数学上似乎起不到什么作用。回头看一下官网的介绍

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the certification of properties of programming languages (e.g. the CompCert compiler certification project, the Verified Software Toolchain for verification of C programs, or the Iris framework for concurrent separation logic), the formalization of mathematics (e.g. the full formalization of the Feit-Thompson theorem, or homotopy type theory), and teaching.

²即自然数子集的良序关系，其表示自然数子集存在最小元，从而表明了无穷下降法的可行性

它自称形式证明管理系统，而典型应用的第一个不是数学形式化，而是 the certification of properties of programming languages(编程语言性质的认证)，并且还给出了CompCert 计划。这样看来，虽然 Coq 还提供了有理数库 QArith、实数库 Reals、向量库、列表库、集合库等等，但我觉得没必要再深入了解了，主要因为里面的许多标准库的定义并不令人满意，更何况还包含了很多计算机特性的库 Unicode、Program、ssr、Sorting，我不能理解其在数学证明中的作用，而只会把它们视为理论精简性的障碍。正因如此 Lean，它来了。

7.2 Lean

安装与运行

Lean 和 Coq 使用相同的理论基础，所以我们没必要讲理论，可以从应用直接讲起。首先，我们需要进入[lean 社区首页](#)；接着在“Installation”一栏中点击“Windows installation”；最后按照里面的教程完成安装即可。我们总共安装了这些内容：Lean4(基础环境)、Lake(Lean4 构建系统与包管理工具)、elan(一键工具)、mathlib4(数学库，也是标准库)、lean4 的 VSCode 插件。我们可以看到 Lean 相较于 Coq 有更大的数学倾向，并且在众多数学家的参与下有一个庞大的符合数学习惯的[数学库](#)，而且 Lean 还能使用像 \mathbb{N}, \mathbb{R} 之类的符号。

如果你不想要 elan 的一键安装，想要了解各个过程，我也可以稍微讲一讲。首先去[这里](#)下载 Lean4 的压缩包；然后把它解压到你喜欢的目录，并在环境变量的 PATH 中添加“安装目录/bin”；最后在 VSCode 的插件市场中安装 lean4 即可，它会根据环境变量 PATH 自动找到我们的 Lean4。然后测试运行看是否成功

The screenshot shows the Lean 4 IDE interface. On the left, there is a code editor window with the following Lean 4 code:

```

Lean > #eval Lean.leanVersionString
1  eval Lean.leanVersionString
2
3  /- 定义一些常数 -/
4
5  def m : Nat := 1      -- m 是自然数
6  def n : Nat := 0
7  def b1 : Bool := true -- b1 是布尔型
8  def b2 : Bool := false
9
10 /- 检查类型 -/
11
12 #check m           -- 输出: Nat
13 #check n           -- Nat
14 #check n + 0        -- Nat
15 #check m * (n + 0) -- Nat
16 #check b1           -- Bool
17 #check b1 & b2      -- "&&" is the Boolean and
18 #check b1 || b2     -- Boolean or
19 #check true         -- Boolean "true"
20
21 /- 求值 (Evaluate) -/
22
23 #eval 5 * 4          -- 20
24 #eval m + 2          -- 3
25 #eval b1 & b2        -- false

```

On the right, there is a status bar showing the current file and commit information, along with several tabs for different Lean files (main.lean:1:0, main.lean:12:0, main.lean:13:0, main.lean:14:0, main.lean:15:0, main.lean:16:0, main.lean:17:0, main.lean:18:0) and a message list tab.

当然为了享受已经形式化的内容，我们还需要安装 mathlib4 库。Lean4 实际还是一种函数式编程语言，有一套完善的开发工作流程，还能编译成二进制文件，比如随便找个空文件夹，通过命令行运行“lake init foo”、“lake build”，并运行“./build/bin/foo”就能实现最基础的“Hello, world!”，但我们不需要这么复杂的内容，只需要它能服务于数学即可。但我们实际需要做的事情并不多，首先下载[mathlib4](#)库并解压；接着进入文件夹，打开命令行运行“lake build”，等待编译结束；最后使用 VSCode 打开这个文件夹，就可以像看书一样查看各个定理了。我们只需要把光标放到任意行的结尾就能看程序运行到这里时的状态

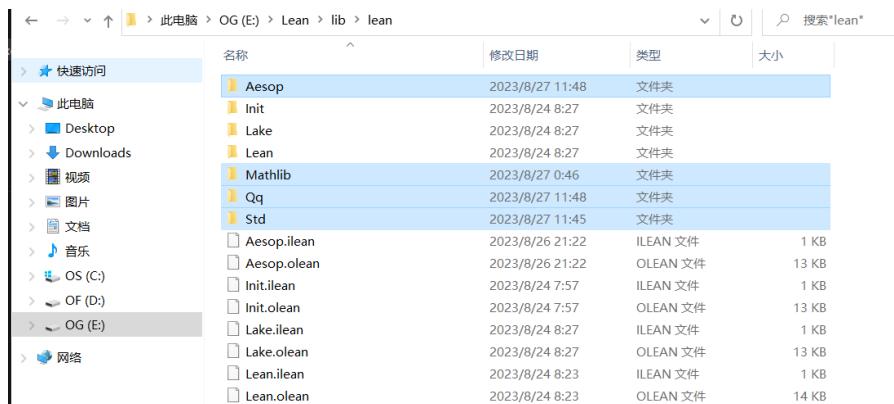
The screenshot shows the VSCode IDE with the Denumerable.lean file open in the editor. The Explorer sidebar on the left shows the project structure under the MATHLIB4 folder. The Editor tab shows the code for the Denumerable.lean file, which includes imports from Mathlib, Data.Rat, and other Lean standard library files. The Leainfowview tab on the right shows the current goal state and the type of the variable T. The status bar at the bottom indicates the file is saved.

如果你没进行预先编译，则你在查看的时候也会后台进行编译，此时你无法进行控制也无法看到进度，所以我们推荐提前做好编译。mathlib4 已经准备好了，接下来就是该如何引用它，最简单的办法就是利用好 mathlib4

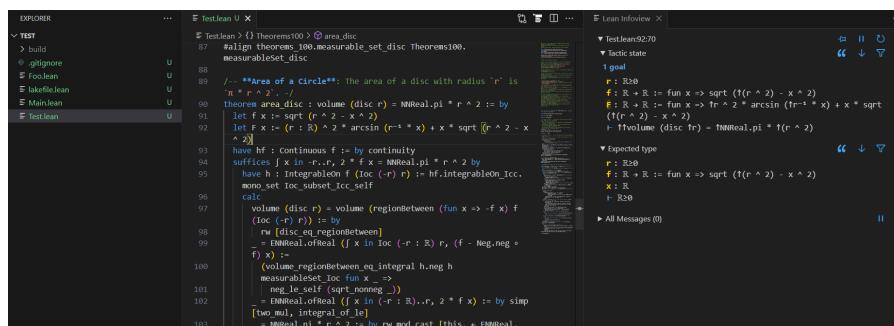
的环境，直接在里面创建文件写代码，顺便还能试着参与 mathlib4 项目；另一种方法是一键流，随便找个地方执行指令

```
lake new 项目名 math
cd 项目名
lake update
lake exe cache get
```

第一个指令表示创建一个 lake 项目并使用依赖 mathlib4，第二个指令表示进入你的项目目录，第三个指令会自动更新 mathlib4 仓库到 lake-packages 目录下，第四个指令会下载编译好的文件，这样在这个项目下就能使用 mathlib4 了，这样做的缺点是每个项目都会重新下载一遍 mathlib4，极其消耗内存，有一股 Node.js 的味道了。还有一种方法是把编译好的库放到“Lean 解压目录/lib/lean”下，让它成为系统库，目前我个人比较喜欢用这个方法，不过记得把依赖一起带进去



这样就能引入 mathlib4 库了，让我们随便抄个例子看看效果



实例学习

不同于 Coq 我们以证明“代数基本定理：非常数复多项式至少有一个复根”为例，代码位于 mathlib4 的“Mathlib/Analysis/Complex/Polynomial.lean”中，从一开头我们就了解到了 Lean 的注释方式为

/- 多行注释 -/

-- 单行注释

注意在单行注释中两个“-”是连在一起的。我们最开始遇到的代码当然是库的引入和使用了

```
1 /-
2 Copyright (c) 2019 Chris Hughes All rights reserved.
3 Released under Apache 2.0 license as described in the file LICENSE.
4 Authors: Chris Hughes, Junyan Xu
```

```

5  -/
6 import Mathlib.Analysis.Complex.Liouville
7 import Mathlib.FieldTheory.IsAlgClosed.Basic
8 import Mathlib.Analysis.Calculus.Deriv.Polynomial
9 import Mathlib.Topology.Algebra.Polynomial
10
11 #align_import analysis.complex.polynomial from
12   "leanprover-community/mathlib@\"17ef379e997badd73e5eabb4d38f11919ab3c4b3"
13
14 /-!
15 # The fundamental theorem of algebra
16
17 This file proves that every nonconstant complex polynomial has a root using Liouville's theorem.
18
19 As a consequence, the complex numbers are algebraically closed.
20 -/
21
22 open Polynomial
23
24 open scoped Polynomial

```

最开始的四个 import 表示引入外部定义和定理，主要是两个，即刘维尔定理 Liouville 和多项式 Polynomial，后面我们具体遇到再具体说。“#align_import...” 主要表示与 lean3 的对应，没啥用可以直接删除。“open Polynomial” 表示打开命名空间 Polynomial，简单来讲就是 C++ 中的 “using ...”，这样我们在使用命名空间的对象时就可以不用加前缀了。“open scoped Polynomial” 表示使用命名空间 Polynomial 中的记号 (notation, 例如 \mathbb{C})，这东西会传递的所以删掉其实也无所谓，不过为了预防以后可能出现的问题最好加上。

The screenshot shows the Lean 3 code editor with the following code:

```

Mathlib > Analysis > Complex > ℂ Polynomial.lean > {} Complex > exists_root
25 namespace Complex
26
27 /-- **Fundamental theorem of algebra**: every non constant complex polynomial
28   has a root -/
29 theorem exists_root {f : ℂ[X]} (hf : 0 < degree f) : ∃ z : ℂ, IsRoot f z := by
30   contrapose! hf
31   have : Metric.Bounded (Set.range (eval • f)⁻¹)
32   · obtain ⟨z₀, h₀⟩ := f.exists_forall_norm_le
33   · simp only [Pi.inv_apply, bounded_iff forall_norm_le, Set.forall_range_iff, norm_inv]
34   · exact (eval z₀ f)⁻¹, fun z => inv_le_inv_of_le (norm_pos_if_pos_2 < hf z₀) (h₀ z)
35   obtain ⟨c, hc⟩ := (f.differentiable.inv hf).exists_const_forall_eq_of_bounded this
36   · obtain rfl : f = c c⁻¹ := Polynomial.funext fun z => by rw [eval_c, ← hc z, inv_inv]
37   · exact degree_c_le
38   #align complex.exists_root Complex.exists_root
39
40 instance isAlgClosed : IsAlgClosed ℂ :=
41   IsAlgclosed.of_exists_root _ fun _p_ hp => Complex.exists_root <|
42     degree_pos_of_irreducible hp
43   #align complex.is_alg_closed Complex.isAlgClosed
44
45 end Complex

```

The right side of the interface shows the tactic state and goal:

- Tactic state**
- goal**: $f : \mathbb{C}[X]$, $hf : \forall (z : \mathbb{C}), \neg \text{IsRoot } f z$, $\vdash \text{degree } f \leq 0$
- Expected type**: $f : \mathbb{C}[X]$, $hf : 0 < \text{degree } f$, $\vdash 0 < \text{degree } f$
- All Messages (0)**

核心的证明部分倒是挺短的，命名空间 namespace 没啥可讲的，和 C++ 的基本一样。定理的基本格式如下

theorem 定理名称上下文: 描述:= by

此处的上下文其实就是一系列的类型判断，我们还能将其理解为调用定理时的输入参数，其中花括号 “{}” 表示默认隐式参数，如果是括号 “()” 的话我们在调用时可以使用下划线 “_” 来表示隐式参数，隐式参数表示上下文可推断的参数，我们拿目前这个例子来举例，它的形式描述为

$$f : \mathbb{C}[X], hf : \deg f > 0 \vdash \exists z : \mathbb{C}. f(z) = 0$$

显然我们只要给出第二个假设 $\deg f > 0$ ，它就默认包含了多项式 f ，也就是说这个参数 $f : \mathbb{C}[X]$ 可以从上下文中推断出来，最后的 by 则是说明我们要通过多个策略 (tactics) 来完成证明，这里的策略和 Coq 中的一条条指

令的含义一致，Lean 与 Coq 的区别在于没有“.”作为结束符，而是使用一行表示一个策略，但类似地 Lean 也可以使用分号“;”来表示组合策略，当策略状态 (tactic state) 中的目标 (goal) 为零时表示证明结束，退出策略模式。指令“`contrapose! hf`”表示针对假设 `hf` 使用反证法，即我们假设结论不成立，并推出与 `hf` 矛盾，对应形式描述为(以后我们不再使用 Π 而使用更符合数学习惯的 \forall)

$$\Gamma \equiv f : \mathbb{C}[x], \text{hf} : \forall z : \mathbb{C}. f(z) \neq 0 \vdash \deg f \leq 0$$

指令“`have 名称: 描述`”表示产生一个子目标，并将其放到当前目标的上下文中，如果我们不写名称，其默认使用 `this` 作为名称，此处的子目标简单来说就是“ $f^{-1}(z)$ 是有界的”，它的形式描述为

$$\Gamma \vdash \exists C : \mathbb{C}. \forall z : \mathbb{C}. |f^{-1}(z)| \leq C$$

接着点号“`·`”表示进入子目标的证明，这样可以使右边的视图只有这一个目标，从而排除上一个目标的干扰，但这个子目标结束证明时，就会回到原来的目标。指令“`obtain`”顾名思义就是获得的意思，此处表示从定理“`exists_forall_norm_le`”获得两个输出，并以 z_0, h_0 的元放到上下文中，定理“`exists_forall_norm_le`”的形式描述为

$$p : R[X] \vdash \exists x : R. \forall y : R. \|f(x)\| \leq \|f(y)\|$$

这里的 R 表示完备赋范空间，其实我们没必要想这么多，直接考虑复数 $R = \mathbb{C}$ ，则相应的范数为复数的模 $z \in \mathbb{C}, \|z\| = |z|$ ，因此这个定理表示“ $|f(z)|$ 有最小值”，我们取出来的 z_0 表示最小值点， h_0 则是性质的描述。指令“`simp only`”表示我们只用方括号中的定理化简结论，其中的“`Pi.inv_apply`”、“`bounded_iff_forall_norm_le`”、“`Set.forall_range_iff`”都是用来把前面 `have` 所使用的函数进行展开用的，得到的结果就是我们前面所给的形式描述，定理“`norm_inv`”表示“倒数的模等于模的倒数”可以将结论进一步化简为

$$\Gamma_0 \equiv \Gamma, z_0 : \mathbb{C}, h_0 : \forall y : \mathbb{C}. |f(z_0)| \leq |f(y)| \vdash \exists C : \mathbb{C}. \forall z : \mathbb{C}. |f(z)|^{-1} \leq C$$

即把倒数提到模外。简便起见，对于同类型的量词我们可以进行合并书写“ $\exists C \forall z : \mathbb{C} \equiv \exists C : \mathbb{C}. \forall z : \mathbb{C}$ ”。指令“`exact e`”表示通过 `e` 来结束证明，实际上我们想要的“模倒函数 $|f(z)|^{-1}$ 的最大值”就是上下文中的“函数模 $|f(z)|$ 的最小值”，即

$$\frac{1}{|f(z)|} \leq \frac{1}{|f(z_0)|} = C$$

我们需要证明的命题是如下形式

$$\exists C : \mathbb{C}. A \equiv \forall z : \mathbb{C}. |f(z)|^{-1} \leq C$$

而 `exact` 的参数表“`exact <C, A>`”用于匹配上述的变量，运行时它会自动匹配 `C` 的类型 \mathbb{C} 和 `A` 的类型 `Prop`，第一个参数 $C \equiv |f(z_0)|^{-1}$ 容易读懂，所以我们来看第二个参数，首先“`fun`”对应类型论中的 λ 表达式，此处的看法为

$$\lambda z : \mathbb{C}. B$$

$$B \equiv \text{inv_le_inv_of_le} (\text{norm_pos_iff}.2 <| \text{hf} z_0) (h_0 z)$$

后面涉及了几个定理的复合，其中两个带名字的定理的形式描述为

$$(\text{inv_le_inv_of_le}).ha : 0 < a, h : a \leq b \vdash b^{-1} \leq a^{-1}$$

$$(\text{norm_pos_iff}) \vdash 0 < |a| \leftrightarrow a \neq 0$$

先看第一个参数，其中“.2”表示从等价中提取第二个项作为上下文，即有

$$(\text{norm_pos_iff.2}) a \neq 0 \vdash 0 < |a|$$

接着“<| term”是代入参数的另一种写法，其参数是一个整体使用“ $\text{hf } z_0 \equiv (\forall z : \mathbb{C}. f(z) \neq 0) z_0 \rightarrow_{\beta} f(z_0) \neq 0$ ”，于是就能得到

$$(\text{norm_pos_iff.2} <| \text{hf } z_0) \equiv |f(z_0)| > 0$$

然后是第二个参数，即一次使用

$$h_0 z \equiv (\forall y : \mathbb{C}. |f(z_0)| \leq |f(y)|) z \rightarrow_{\beta} |f(z_0)| \leq |f(z)|$$

将这两个参数代入定理“inv_le_inv_of_le”即可得

$$0 < |f(z_0)|, |f(z_0)| \leq |f(z)| \Rightarrow |f(z)|^{-1} \leq |f(z_0)|^{-1}$$

这样我们就完成了一个局部目标。指令“obtain”用来获取参数，我们主要来看后面；定理 differentiable.inv 表示倒函数的可微性，即“如果 $f(z)$ 不恒为零则 $f^{-1}(z)$ 可微”，它需要输入不恒为零的条件 hf，并且返回相应的倒函数；定理 exists_const_forall_eq_of_bounded 表示刘维尔定理，即“有界可微复变函数恒为常数”，它需要输入有界的条件 this，并且返回“函数恒为常数”，其形式表达为

$$\exists c \forall z : \mathbb{C}. f^{-1}(z) = c$$

因此获得参数匹配后要证的结论为

$$\Gamma_1 \equiv \Gamma, c : \mathbb{C}, \text{hc} : \forall z : \mathbb{C}. f^{-1}(z) = c \vdash \deg f \leq 0$$

后面的点号其实没必要，但需要注意 Lean 在策略模式下要求语句严格对齐，删掉点号后需要把同一个目标的策略全部对齐才行。接着又是指令 obtain，此时我们获得了一个假设 rfl，即函数 $f(z)$ 为常数，这里的“ $C c^{-1}$ ”表示将复数类型的常数 c^{-1} 转化为多项式类型 $c^{-1} \in \mathbb{C}[X]$ ，这样同种类型下就能直接写等号了 $f(z) = c^{-1}$ 。为了得到这个假设，我们需要利用函数相等的定理

$$(\text{Polynomial.funext})\text{ext} : \forall r : \mathbb{C}. p(r) = q(r) \vdash p = q$$

为了让假设 hc 匹配这个定理的参数，我们需要通过复写“by rw”来定义函数，这里需要提醒一下读者，我们需要使用“函数 fun”来作为“全称量词 \forall ”类型的参数，如果不能理解的话，可以回去看看类型论。复写“by rw”相当于一个等式，我们可以使用连等来进行推断

$$f(z) = (f(z)^{-1})^{-1} = c^{-1} = c^{-1}(z)$$

使用连等推断时，要把复写“by rw”后的内容反过来看，“inv_inv”表示倒数的倒数等于本身“ $(f^{-1})^{-1} = f$ ”、“hc z”是条件 hc 的代入结果 $(\forall z : \mathbb{C}. f^{-1}(z) = c) z \rightarrow_{\beta} f^{-1}(z) = c$ ，为了将等号方向进行对应我们使用反箭头“ $\leftarrow (f^{-1}(z) = c) \equiv (c = f^{-1}(z))$ ”、“eval_C”为常值函数的常值性“ $c(z) = c$ ”，这样我们就得到了假设 rfl。并且目标变为

$$\Gamma_1 \vdash \deg(c^{-1}(z)) \leq 0$$

最后利用“degree_C_le”即“常值函数的次数小于零”即可完成证明，至此证明结束。这个证明实际就是一般复分析书籍对代数基本定理的证明，可见 Lean 在证明上是十分偏向于数学习惯的，相比之下，Coq 对于定理的描述都有些难以理解

```

187
188 lemma FTA : forall f : CCX, nonConst _ f -> {z : CC | f ! z [-] [0]}.
189 Proof.
190 intros.
191 elim (Cpoly_ex_degree _ f). intro n. intros. (* Set_ not necessary *)
192 apply FTA' with n; auto.
193 Qed.

```

<https://github.com/coq-community/corn/blob/master/fta/FTA.v#L188>

小结

对于 Lean 的库其实没什么好介绍的，我们所说的库主要指数学库 mathlib4，实际上 Lean4 本来是一种可编译的函数式编程语言，因此自身本来有着过硬的基底，但不知为何变成了数学的证明辅助工具，结果导致 mathlib4 变成标准库，基本所有的文档都是针对数学证明的，而 Lean4 本来的[文档](#)有一大堆 TODO 没写，所以仿佛[Lean4 社区](#)才是官网。如果从这个角度来看待 Lean4 的话，那么它的[官方库文档](#)是十分详尽且结构清晰，一目了然的，所以我无需像 Coq 那般提醒你哪些是与数学相关的内容。

在所有的证明辅助工具中，我最爱用的是 Lean，它的强大有目共睹，并且受到了大多数数学家的偏爱。需要提醒读者的是，这并不意味着我们能抛弃像 Coq 之类的其它证明辅助工具，由于 Coq 的“历史悠久”，导致有些定理只有 Coq 有，例如四色定理 (Four Color Theorem) 和 Feit-Thompson 定理 (奇数阶群必为可解群)。其实还有一个问题是，大家好像都比较懒，对于已经被某个证明辅助工具形式化的定理，如果其不具备应用意义的话，一般就不会移植到其它的证明辅助工具中，比如 Wiedijk100 中的第五个，素数定理 (Prime Number Theorem)，显然像 Coq、Lean 之类的都具备需求核心的分析库，但却只有 Isabelle、HOL Light 之类的才有。我们可以看到有不少定理被 Isabelle 和 HOL Light 给独占了，因此除了学习 Coq 和 Lean 以外，也有必要了解一下它们，这样我们只需四大证明辅助工具“Coq、Lean、Isabelle、HOL Light”就能吃透 Wiedijk100 中的 99 个定理，唯一没有被形式化的是 33，费马大定理 (Fermat's Last Theorem)，这玩意搞个完整的证明都难，更别说形式了。

7.3 Isabelle

对于 Isabelle 的安装比较简单，直接去[官网](#)下载并安装即可，而且还自带了一个 UI 交互界面，所以我们可以直接拿例子来进行讲解，读者需要注意 Isabelle，或者 HOL 家族的证明辅助工具，所依赖的理论基础是简单类型论 $\lambda\text{-}$ ，这意味着项的合格形式为：变量 x 、箭头 “ $\text{var1} \rightarrow \text{var1}$ ”、应用 “ term1 term2 ” 和函数 “ $\lambda x.\text{term}$ ”，而一阶逻辑、高阶逻辑等都是作为库在简单类型论中实现出来的，结合起来就是官方文档所给出的等式

$$\text{HOL} = \text{Functional Programming} + \text{Logic}$$

这次我们选择证明“有理数集是可数的”来作为例子，其位于“安装目录/src/HOL/Library/Countable.thy”中。

```

begin
lemma Rats_eq_range_of_rat_o_nat_to_rat_surj:
  "Q = range (of_rat o nat_to_rat_surj)"
  using surj_nat_to_rat_surj
by (auto simp: Rats_def image_def surj_def) (blast intro: arg_cong)
lemma "r ∈ Q ⟹ ∃n. r = of_rat (nat_to_rat_surj n)"
by (simp add: Rats_eq_range_of_rat_o_nat_to_rat_surj image_def)
end
instance rat :: countable
proof
show "∃to_nat::rat ⇒ nat. inj to_nat"
proof
have "surj nat_to_rat_surj"
  by (rule surj_nat_to_rat_surj)
then show "inj (inv nat_to_rat_surj)"
  by (rule surj_imp_inj_inv)
qed
qed
theorem rat_denum: "∃f :: nat ⇒ rat. surj f"
using surj_nat_to_rat_surj by metis
end

```

在 Isabelle 中以一个理论为文件单位，文件后缀 thy 实际就是理论 theory 的缩写，相应的基本格式为

```

1 theory Countable
2 imports Old_Datatype HOL.Rat Nat_Bijection
3 begin
4 ...
5 end

```

其中的第一行“theory Countable”表示理论的名称是可数性 (Countable)，第二行表示可数性理论所依赖的理论为：Old_Datatype、HOL.Rat、Nat_Bijection，一眼望去第一个提供像集合之类的类型，第二个提供有理数以备我们的主题，第三个提供自然数双射以给出可数的定义。最开始的 section 和后面的 subsection 像注释一般的起不到作用，删掉照样可以运行，虽然注释的格式应该是“(* 注释内容 *)”。class 是一种可继承的定义，其基本格式为

```

1 class 名称 =
2   fixes 名1 :: 箭头式 (infixl 缩写 优先级)
3   assumes 名2: 假设公式

```

在 Isabelle，我们把箭头式类型 “ $x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n \rightarrow y$ ” 称为函数，其中 x_1, \dots, x_n 说输入变量类型， y 是输出变量类型，如果只输入一部分得到的也是一个箭头类型，即函数。fixes 的内容相当于说明假设公式所依赖函数，infixl 则用于函数符号的缩写，如果有多个依赖函数就可以多写几个 fixes，assumes 也是类似的。假设公式使用高阶逻辑 HOL 的公式，即存在量词 \exists 和全称量词 \forall 的后面可以加上谓词和函数，谓词和函数均用箭头类型，由于我们在 Isabelle 定义了基础类型 bool，因此我们把谓词 $P(x_1, \dots, x_n)$ 视为函数 $x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n \rightarrow \text{bool}$ ，这样我们就把理论给搞清楚了，回到原类。

```

1 class countable =
2   assumes ex_inj: "\<exists>to_nat :: 'a \<Rightarrow> nat. inj to_nat"
3
4 lemma countable_classI:
5   fixes f :: "'a \<Rightarrow> nat"
6   assumes "\<And>x y. f x = f y \<Rightarrow> x = y"
7   shows "OFCLASS('a, countable_class)"
8 proof (intro_classes, rule exI)
9   show "inj f"
10  by (rule injI [OF assms]) assumption
11 qed

```

假设 ex_inj 内容的形式化表述为 $\exists f : x \rightarrow \text{nat}. \text{inj}(f)$ ，在这里一瞥在字母前表示变量类型，如 “'a,'b,...”，而正常写出则表示一般的由项得到的类型，如“布尔类型 bool，自然数类型 nat,...”，inj 是单射谓词，用来表示输入函数 f 是否为单射，因此此处假设的含义为“存在到自然数的单射”。接着是定理的基本格式，当然以 lemma、theorem 等开头的本质都是一样的

```

1 theorem 名称 =
2   fixes 名1 :: 箭头式 (infixl 缩写 优先级)
3   assumes 假设公式
4   show 结论公式
5   proof
6   ...
7 qed

```

显然 Isabelle 中的定理十分的语言化，不过这里的假设有点怪，其中 “ $\wedge x_1 x_2 \dots x_n. A$ ” 的前部分用来声明公式 A 中涉及的变量，官方解释为“for an arbitrary but fixed”，我们应该理解为高阶逻辑中的常量申明，只有这样我

们才能让函数 “f:a->nat” 作用在上面。在结论中 OFCLASS 是一个系统符 (其输入不是两个类型的实例)，其中“名称 _class” 用来表示名称为名称的 class 逻辑物，前部分用来填充类型，而整体形成一个公式

$$\begin{aligned}
 prop &= (\ prop) \\
 &\mid prop^{(4)} :: type && (3) \\
 &\mid any^{(3)} == any^{(3)} && (2) \\
 &\mid any^{(3)} \equiv any^{(3)} && (2) \\
 &\mid prop^{(3)} \&&& \&& prop^{(2)} && (2) \\
 &\mid prop^{(2)} ==> prop^{(1)} && (1) \\
 &\mid prop^{(2)} \implies prop^{(1)} && (1) \\
 &\mid [| prop ; \dots ; prop |] ==> prop^{(1)} && (1) \\
 &\mid [\![prop ; \dots ; prop]\!] \implies prop^{(1)} && (1) \\
 &\mid !! idts . prop && (0) \\
 &\mid \bigwedge idts . prop && (0) \\
 &\mid \text{OFCCLASS} (type , logic)
 \end{aligned}$$

于是结论的含义为变量 a 具有可数类型，这么一看定理 countable_classI 实际就是把单射展开成了公式形式。intro_classes 表示类的引入规则，此处意味着我们将可数类定义中的假设拉到证明的假设中，“rule exI” 表示使用规则 exI，它可以将常量表达式转化为存在量词表达式 “A(a) $\vdash \exists a. A(a)$ ”。在形式书写中我们要倒着看，此处的存在量词来自于可数类的定义，这意味着我们的目标转为证明单射 “inj f”，“proof(规则 1,..., 规则 n)” 为从结论的假设的逆向推理，“show 结论 by (规则 1,..., 规则 n)” 则是从假设到结论的正向推理。“injI [OF assms]” 表示规则 “OF assms” 复合规则 injI，“OF 公式” 表示将公式转化为规则，例如此处 assms 代指假设，因此得到规则 $f(x) = f(y) \vdash x = y$ ，injI 为单射定义规则，因此结合起来即得到了结论 “inj f”。

```

1 definition to_nat :: "'a::countable <=> nat" where
2   "to_nat = (SOME f. inj f)"
3
4 definition from_nat :: "nat <=> 'a::countable" where
5   "from_nat = inv (to_nat :: 'a <=> nat)"
6
7 lemma inj_to_nat [simp]: "inj to_nat"
8   by (rule exE_some [OF ex_inj]) (simp add: to_nat_def)
9
10 lemma inj_on_to_nat [simp, intro]: "inj_on to_nat S"
11   using inj_to_nat by (auto simp: inj_on_def)
12
13 lemma surj_from_nat [simp]: "surj from_nat"
14   unfolding from_nat_def by (simp add: inj_imp_surj_inv)
15
16 lemma to_nat_split [simp]: "to_nat x = to_nat y <=> x = y"
17   using injD [OF inj_to_nat] by auto
18
19 lemma from_nat_to_nat [simp]:
20   "from_nat (to_nat x) = x"
21   by (simp add: from_nat_def)

```

开头的 definition 由于定义非递归函数，基本格式为“definition 名称:: 箭头式 where 定义法”，在 to_nat 中“SOME 变量. 公式”表示希尔伯特截面算子，所以函数“to_nat”表示可数定义中的可数集合与自然数的对应，与之对应的是 from_nat，其中 inv 表示反转函数，接着我们来看几个引理。第一个表示 to_nat 是单射，“inj_on”是一个二元谓词，要求输入一个函数和集合，此处 S 应该是一个集合实例，但 Isabelle/HOL 没办法溯源查找，如果去文档里 find 会匹配到一堆 S，根本毫无用处，只能说确实有点垃圾，不过这一堆引理其实就是像告诉我们最后的结论，即可数的对应是集合与自然数的双射，且 to_nat 与 from_nat。另外此处的 lemma 证明使用缩写，对于可以一步到位的结论是可行的。

```

1 subclass (in finite) countable
2 proof
3   have "finite (UNIV::'a set)" by (rule finite_UNIV)
4   with finite_conv_nat_seg_image [of "UNIV::'a set"]
5   obtain n and f :: "nat \<Rightarrow> 'a"
6     where "UNIV = f ` {i. i < n}" by auto
7   then have "surj f" unfolding surj_def by auto
8   then have "inj (inv f)" by (rule surj_imp_inj_inv)
9   then show "\<exists>to_nat :: 'a \<Rightarrow> nat. inj to_nat" by (rule exI[of inj])
10  qed

```

此处“subclass (in 类型 1) 类 1”表示类型验证，即类型 1 是类 1，也就是说类型 1 是类 1 的子类，因此此处即为证明有限类型是可数的。“have...by...”和“show...by...”是完全一样的，而“with 定理公式”为“this using 定理公式”的一种写法，this 放在结论处表示下一步的条件处内容，如果 this 放在条件处则表示上一步的结论处内容，换言之 this 起到上下文连通的管道作用。“obtain 变量 where 内容 by 规则”表示我们获得有限类型的长度 n 和相应的编号映射“f:nat->'a”，获取的条件由上文的 this 传递下来。then 为“from this”的缩写，放到整体中我们有“have/show 结论 by 规则 using 定理公式 ≡from 定理公式 have/show 结论 by 规则”，因此此处接连三个“then have/show”用来表示正向推理，直到最后一行得到了可数的条件。接下来的三个“subsection”是各种对象的可数性验证，与我们的主题没有关系，可以直接跳过，让我们从 282 行开始看起。

```

1 definition nat_to_rat_surj :: "nat \<Rightarrow> rat" where
2   "nat_to_rat_surj n = (let (a, b) = prod_decode n in Fract (int_decode a) (int_decode b))"
3
4 lemma surj_nat_to_rat_surj: "surj nat_to_rat_surj"
5   unfolding surj_def
6 proof
7   fix r::rat
8   show "\<exists>n. r = nat_to_rat_surj n"
9   proof (cases r)
10    fix i j assume [simp]: "r = Fract i j" and "j > 0"
11    have "r = (let m = int_encode i; n = int_encode j in nat_to_rat_surj (prod_encode (m, n)))"
12      by (simp add: Let_def nat_to_rat_surj_def)
13    thus "\<exists>n. r = nat_to_rat_surj n" by (auto simp: Let_def)
14  qed
15 qed
16
17 lemma Rats_eq_range_nat_to_rat_surj: "\<rat> = range nat_to_rat_surj"
18   by (simp add: Rats_def surj_nat_to_rat_surj)

```

最开始的定义 nat_to_rat_surj 表示自然数到有理数的函数，此处涉及有理数的定义，给出一个有理数相当于给出一个自然数对 $(a,b) \equiv \frac{a}{b}$ ，其中 int_decode 为整数的自然数编码，其在 244 行证明整数是可数时也有用到，

`prod_decode` 为笛卡尔积的自然数编码，此处“`prod_decode n in 笛卡儿积`”相当于从笛卡儿积中取出序号为 `n` 的对象，`Fract` 的含义虽然是分式，但本质上表示一个自然数对，从而生成一个笛卡儿积，所以此处的有理数编码实际就是我们常用的循环斜线。接下来的定义要我们证明对应函数是一个满射，也就是说没一个有理数都有一个编号，“`unfolding`” 将满射谓词展开，从而形成后面书写的内容“`fix 有理数 show ...`”。接着“`cases r`” 即根据有理数定义分情况（实际只有一种情况）证明，此时结论的形式内容为“`fix 整数 i 整数 j assume r=i/j 并且 j>0`”。“`thus 结论`” 为“`from this show 结论`” 的缩写，此处表示我们根据上文的结论，得到了我们需要证明的结论。最后一个引理表示，我们对应函数的值域为有理数，这是满射的自然推论。

```

1 context field_char_0
2 begin
3
4 lemma Rats_eq_range_of_rat_o_nat_to_rat_surj:
5   "\<rat> = range (of_rat \<circ> nat_to_rat_surj)"
6   using surj_nat_to_rat_surj
7   by (auto simp: Rats_def image_def surj_def) (blast intro: arg_cong[where f = of_rat])
8
9 lemma surj_of_rat_nat_to_rat_surj:
10   "r \<in> \<rat> \<Longrightarrow> \<exists>n. r = of_rat (nat_to_rat_surj n)"
11   by (simp add: Rats_eq_range_of_rat_o_nat_to_rat_surj image_def)
12
13 end

```

此处的 `context` 相当于文件局部定义定理的申明

```

1 context 名称
2 begin
3 ...
4 end

```

详细可以看官方参考文档 locals 的 2.2。此处证明的两个引理，一眼望去就是在说复合函数“`of_rat(nat_to_rat_surj n)`” 可以同样实现自然数到有理数的对应，这里的 `of_rat` 是有理数到特征为 0 的素域的对应。这么一搞，其实也没什么，无非就是换了一种描述有理数的方法，这里的记号 \mathbb{Q} 和上一个引理的看法稍有不同，即上一次视为由整数构造的有理数，此处视为特征为 0 的素域。

```

1 instance rat :: countable
2 proof
3   show "\<exists>to_nat::rat \<Rightarrow> nat. inj to_nat"
4   proof
5     have "surj nat_to_rat_surj"
6       by (rule surj_nat_to_rat_surj)
7     then show "inj (inv nat_to_rat_surj)"
8       by (rule surj_imp_inj_inv)
9     qed
10    qed
11
12 theorem rat_denum: "\<exists>f :: nat \<Rightarrow> rat. surj f"
13   using surj_nat_to_rat_surj by metis

```

“`instance 元:: 类型`” 是类型判断，此处表示有理数是可数类型。这似乎和 `subclass` 没区别吗？并非如此，`rat` 是具体类型，而 `finite` 为抽象类型，通俗来讲 `finite` 类型只是说明了有限，其中元是无限制的，因此我们可以说某些集合是有限类型，而 `rat` 中元是固定为有理数的，这意味着我们只能说某个有理数具有 `rat` 类型，实际上如果

我们找具有 finite 类型的实例，比如 “ \mathbb{F}_p :finite”，我们就能引出判断 “instance $\mathbb{F}_p :: \text{countable}$ ” 了。“show” 展示了为了得到可数所需的条件，此处我们使用前面构造的函数 “nat_to_rat_surj” 来作为存在量词的参数，再通过前面证明的单满射引理即可完成结论的证明。至此证明完毕，我们也注意到了 Isabelle 的证明并非指令式的策略，而是逻辑式的 “have/show 结论 by 规则 using 定理公式”，结论和定理公式都是用高阶逻辑描述的语句，而规则则是类型论中的推断规则，通常我们只要熟悉了 Isabelle 的推断方法，它的证明看起来和传统数学还是有些像的。

7.4 HOL Light

接下来我们来认识一下 HOL 家族，实际有了 Isabelle 的基础，理解起来并不困难。[这里是 HOL System](#)、[这里是 HOL4](#)、[这里是 HOL Light](#)，它们的基本思想都是通过类型论在函数编程的基础上添加高阶逻辑以实现证明辅助，而它们的推理内核都是极其简单的，大量的内容都是通过标准库来实现的，我们在 Isabelle 的库中可以清晰地看到这一点。根据 wiki 的说法，HOL Light 属于 “A thriving ”minimalist fork””，简单来讲它的内核是 HOL 家族中最简单的，所以我们就来介绍它了。在官网的[这篇文章](#)中我们可以看到其内核的推断规则，另外我们可以了解到 HOL Light 实际相当于 OCaml 的库，这一点可以从它的 github 仓库看到，它的大量证明内容都是以 ml 为后缀的 OCaml 代码，也就是说我们直接省去了 HOL 加法成分中的函数式编程语言，而不像 Isabelle 一样需要自带函数式编程语言作为内核的一部分。由于个人工具的限制，这次就直接在 github 中看证明了，我们以证明“勾股定理：直角三角形的三边长度 a,b (斜边), c , 满足 $a^2 + c^2 = b^2$ ”为例，其位于[这里](#)

The screenshot shows a GitHub repository interface for 'hol-light / 100 / pythagoras.ml'. The left sidebar lists files like platonc.ml, print.ml, polyhedron.ml, primerecip.ml, ptolemy.ml, and pythagoras.ml. The right pane displays the code for 'pythagoras.ml'.

```

(* ===== *)
(* A "proof" of Pythagoras's theorem. Of course something similar is *)
(* implicit in the definition of "norm", but maybe this is still nontrivial. *)
(* ===== *)

needs "Multivariate/misc.ml";;
needs "Multivariate/vectors.ml";;

let PYTHAGORAS = prove
  ('(A B C)real2',
   orthogonal (A - B) (C - B)
   --> norm(C - A) pow 2 + norm(B - A) pow 2 = norm(C - B) pow 2',
   REWRITE_TAC[NORM_POW_2; orthogonal; DOT_LSRW; DOT_SWR; DOT_SYM] THEN
   CONV_TAC REAL_RNG);

```

```

1  (* ===== *)
2  (* A "proof" of Pythagoras's theorem. Of course something similar is *)
3  (* implicit in the definition of "norm", but maybe this is still nontrivial. *)
4  (* ===== *)

6  needs "Multivariate/misc.ml";;
7  needs "Multivariate/vectors.ml";;

```

最开始的两句用来引入外部依赖，值得注意的是在解析几何的加持下，我并不需要复杂的几何公理。首先平面上的点视为实数笛卡尔积的元素 $A, B, C \in \mathbb{R}^2$ ，相应的距离通过向量的范数进行诱导

$$d(A, B) = \sqrt{\|A - B\|}, \|x, y\| = \sqrt{x^2 + y^2}$$

此时直角 $\angle ABC$ 的定义为 $(A - C)(C - B) = 0$ (此处乘法为向量的内积)，相应的结论为

$$\|C - A\|^2 = \|B - A\|^2 + \|C - B\|^2$$

```

1  (* ----- *)
2  (* Direct vector proof (could replace 2 by N and the proof still runs). *)

```

```

3 (* ----- *)
4
5 let PYTHAGORAS = prove
6   (``!A B C:real^2.
7     orthogonal (A - B) (C - B)
8     ==> norm(C - A) pow 2 = norm(B - A) pow 2 + norm(C - B) pow 2``,
9      REWRITE_TAC[NORM_POW_2; orthogonal; DOT_LSUB; DOT_RSUB; DOT_SYM] THEN
10     CONV_TAC REAL_RING);;

11
12 (* ----- *)
13 (* A more explicit and laborious "componentwise" specifically for 2-vectors. *)
14 (* ----- *)
15
16 let PYTHAGORAS = prove
17   (``!A B C:real^2.
18     orthogonal (A - B) (C - B)
19     ==> norm(C - A) pow 2 = norm(B - A) pow 2 + norm(C - B) pow 2``,
20      SIMP_TAC[NORM_POW_2; orthogonal; dot; SUM_2; DIMINDEX_2;
21        VECTOR_SUB_COMPONENT; ARITH] THEN
22      CONV_TAC REAL_RING);;

```

显然此处给出了两个证明，证明的基本格式为

```
let 名称 = prove(` 命题`, 证明);;
```

从注解可以知道，第二个证明是明确而费力且只针对二维情况的，而第一个则是一般性的直接证明且能推广到 n 维情形，所以我们选取第一个证明来进行讲解。在 HOL Light 的公式语法中，“!”表示全称量词 \forall ，“?”表示存在量词 \exists ，“~”表示否定 \neg 。首先“REWRITE_TAC [定理 1,..., 定理 n]”表示根据定理进行复写，括号中定理的含义如下

$$\begin{aligned}
 & (\text{NORM_POW_2}) \forall x \in \mathbb{R}^2, \|x\|^2 = x \cdot x \\
 & (\text{orthogonal})(A - B) \cdot (C - B) = 0 \\
 & (\text{DOT_LSUB}) \forall x, y, z \in \mathbb{R}^2, (x - y) \cdot z = x \cdot z - y \cdot z \\
 & (\text{DOT_RSUB}) \forall x, y, z \in \mathbb{R}^2, x \cdot (y - z) = x \cdot y - x \cdot z \\
 & (\text{DOT_SYM}) \forall x, y \in \mathbb{R}^2, x \cdot y = y \cdot x
 \end{aligned}$$

于是要证的结论化为

$$\begin{aligned}
 & \|C - A\|^2 = \|B - A\|^2 + \|C - B\|^2 \\
 & \equiv (C - A) \cdot (C - A) = (B - A) \cdot (B - A) + (C - B) \cdot (C - B) \\
 & \equiv C \cdot (C - A) - A \cdot (C - A) = B \cdot (B - A) - A \cdot (B - A) + C \cdot (C - B) - B \cdot (C - B) \\
 & \equiv C \cdot C - C \cdot A - (A \cdot C - A \cdot A) = B \cdot B - B \cdot A ...
 \end{aligned}$$

然后 THEN 没啥好说的表示继续证明，后面的“CONV_TAC REAL_RING”是一个整体表示根据实数环的法则进行转化验证，简单来讲就是化简验证相等的意思。我们可以在根目录的文件“tactics.ml”和“calc_rat.ml”中分别找到它们的定义

```

329 (* ----- *)
330 (* Create tactic from a conversion. This allows the conversion to return *)
331 (* |- p rather than |- p = T on a term "p". It also eliminates any goals of *)
332 (* the form "T" automatically. *)
333 (* ----- *)
334
335 let (CONV_TAC: conv -> tactic) =
336   let t_tm = `T` in
337   fun conv ((as1,w) as g) ->
338     let th = conv w in
339     let tm = concl th in
340     if aconv tm w then ACCEPT_TAC th g else
341     let l,r = dest_eq tm in
342     if not(aconv l w) then failwith "CONV_TAC: bad equation" else
343     if r = t_tm then ACCEPT_TAC(EQT_ELIM th) g else
344     let th' = SYM th in
345     null_meta,[as1,r],fun i [th] -> EQ_MP (INSTANTIATE_ALL i th') th;;
346

```



```

calc_rat.ml
471 (* ----- *)
472 (* Basic ring and ideal conversions. | *)
473 (* ----- *)
474
475 let REAL_RING,real_ideal_cofactors =
476   let REAL_INTEGRAL = prove
477     (`(!x. &0 * x = &0) /\
478      (!x y z. (x + y = x + z) <=> (y = z)) /\
479      (!w x y z. (w * y + x * z = w * z + x * y) <=> (w = x) \vee (y = z))`,
480      REWRITE_TAC[MULT_CLAUSES; EQ_ADD_LCANCEL] THEN
481      REWRITE_TAC[GSYM REAL_OF_NUM_EQ];
482      | GSYM REAL_OF_NUM_ADD; GSYM REAL_OF_NUM_MUL] THEN
483      ONCE_REWRITE_TAC[GSYM REAL_SUB_0] THEN
484      REWRITE_TAC[GSYM REAL_ENTIRE] THEN REAL_ARITH_TAC)
485 and REAL_RABINOWITSCH = prove
486   (`!x y:real. ~(x = y) <=> ?z. (x - y) * z = &1`,
487    REPEAT GEN_TAC THEN
488    GEN_REWRITE_TAC (LAND_CONV o RAND_CONV) [GSYM REAL_SUB_0] THEN
489    MESON_TAC[REAL_MUL_RINV; REAL_MUL_LZERO; REAL_ARITH `~(&1 = &0)`])
490 and init = GEN_REWRITE_CONV ONCE_DEPTH_CONV [DECIMAL]
491 and real_ty = `:real` in
492 let pure,ideal =
493   RING_AND_IDEAL_CONV
494     (rat_of_term,term_of_rat,REAL_RAT_EQ_CONV,
495      `(`--`):real->real`, `(+`):real->real->real`, `(-`):real->real->real`,
496      `(inv`):real->real`, `(*)`:real->real->real`, `(/`):real->real->real`,
497      `(pow`):real->num->real`,
498      REAL_INTEGRAL,REAL_RABINOWITSCH,REAL_POLY_CONV) in
499     (fun tm -> let th = init tm in EQ_MP (SYM th) (pure(rand(concl th))),  

500      (fun tms tm -> if forall (fun t -> type_of t = real_ty) (tm::tms)
501        then ideal tms tm
502        else failwith
503          | "real_ideal_cofactors: not all terms have type :real");;

```

因此最后一步其实就是只根据 \mathbb{R}^2 的环性质进行化简得过程

$$\begin{aligned}
& \|C - A\|^2 = \|B - A\|^2 + \|C - B\|^2 \\
& \equiv C \cdot C - C \cdot A - (A \cdot C - A \cdot A) = B \cdot B - B \cdot A \dots \\
& \equiv C \cdot C - 2C \cdot A + A \cdot A = B \cdot B - 2B \cdot A + A \cdot A + C \cdot C - 2C \cdot B + B \cdot B \\
& \equiv -2C \cdot A = B \cdot B - 2B \cdot A - 2C \cdot B + B \cdot B \\
& \equiv 2B \cdot B - 2B \cdot A - 2C \cdot B + 2C \cdot A = 0 \\
& \equiv B \cdot B - B \cdot A - C \cdot B + C \cdot A = 0
\end{aligned}$$

最后的结论和条件中的 “ $\text{orthogonal}(A - B)(C - B) \equiv (A - B) \cdot (C - B) = 0$ ” 一致，从而完成了证明，机器验证基本都是展开验证，是没有因式分解能力的，除非进行人工引导。至此证明完毕，显然 Isabelle 在持有相同的理论基础上比其它 HOL 系列的工具更加清晰明了，这也是 wiki 所说的 “successor of HOL”，即 HOL 的后继者。

7.5 Metamath

实际上持有了四大证明辅助工具的我们早就已经无所畏惧了，后面的内容基本都是介绍性的，目的是为了填充目录，使其看起来丰满一些。**Metamath**虽然比 Coq、Isabelle 之类的工具新一些，但却是一套复古的证明辅助系统，它的理论基础是“一阶逻辑 + 集合论”，不过这里的集合论有稍作调整，比如有一个 Tarski–Grothendieck 公理，详细可以看[这里的“The Axioms”部分](#)。我们可以来观赏一下它是如何证明“素数定理： $\lim_{x \rightarrow \infty} \pi(x) \ln x / x = 1$ ”（位于[这里](#)）的

Theorem pnt 25885			
Description: The Prime Number Theorem: the number of prime numbers less than x tends asymptotically to $x / \log(x)$ as x goes to infinity. This is Metamath 100 proof #5. (Contributed by Mario Carneiro, 1-Jun-2016.)			
Assertion			
Ref			Expression
pnt $\vdash (x \in (1,+\infty) \rightarrow ((g^*x) / (x / (\log^*x))) \rightarrow_s 1)$			
Proof of Theorem pnt			
Dummy variables w, y, z are mutually distinct and distinct from all other variables.			
Step	Hyp	Ref	Expression
1	1xr 10488	$\vdash 1 \in \mathbb{R}^*$	
2	1lt2 11056	$\vdash 1 < 2$	
3	df-ioc 12545	$\vdash (.) = (x \in \mathbb{R}, y \in \mathbb{R}^* \rightarrow z \in \mathbb{R}^* \mid (x < z \wedge z < y))$	
4	df-ico 12547	$\vdash (.) = (x \in \mathbb{R}, y \in \mathbb{R}^* \rightarrow z \in \mathbb{R}^* \mid (y \leq z \wedge z \leq y))$	
5	krfltr 12354	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w))$	
6	3_4_5 12559	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
7	1_2_6 12560	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
8	resmp1 1733	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
9	7_8 12561	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
10	z 12562	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
11	ioss1 12601	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
12	11 12602	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
13	10_12 12603	$\vdash ((1 \in \mathbb{R}^* \wedge z \in \mathbb{R}^* \wedge w \in \mathbb{R}^*) \rightarrow ((1 < z \wedge z \leq w) \rightarrow 1 < w)) \subseteq (1,+\infty)$	
14	gtl 12647	$\vdash z > 2 \in \mathbb{R}$	
15	gtl2 12648	$\vdash z > 2 \in \mathbb{R}^*$	
16	pnfrx 12680	$\vdash ((2 \in \mathbb{R} \wedge z \in \mathbb{R}^*) \rightarrow (x \in (2,+\infty) \rightarrow (x \in \mathbb{R} \wedge 2 \leq x < z)))$	
17	elico2 12693	$\vdash ((2 \in \mathbb{R} \wedge z \in \mathbb{R}^*) \rightarrow (x \in (2,+\infty) \rightarrow (x \in \mathbb{R} \wedge 2 \leq x < z)))$	
18	mp2an 12698	$\vdash ((x \in (2,+\infty) \rightarrow 2 \leq x) \rightarrow ((x \in (2,+\infty) \rightarrow (x \in \mathbb{R} \wedge 2 \leq x < z))) \rightarrow ((x \in (2,+\infty) \rightarrow (x \in \mathbb{R} \wedge 2 \leq x < z)))$	
19	chtrpl 12647	$\vdash ((x \in (2,+\infty) \rightarrow 2 \leq x) \rightarrow ((x \in \mathbb{R} \wedge 2 \leq x) \rightarrow (0^*x) \in \mathbb{R}^*))$	

我们的屏幕太小，不足以把它全部显示完，但已经足以窥见其基本思想了，比如“ $a(,)b$ ”实际是一个函数，它将输入的常量 a,b 输出为一个集合“ (a,b) ”

Definition df-ioc 12545	
Description: Define the set of open intervals of extended reals. (Contributed by NM, 24-Dec-2006.)	
Assertion	
Ref	Expression
df-ioc $\vdash (.) = (x \in \mathbb{R}^*, y \in \mathbb{R}^* \mapsto [z \in \mathbb{R}^* \mid (x < z \wedge z < y)])$	
<i>Distinct variable group:</i> x,y,z	

当然我们这里说的函数是集合论中的函数，而不是 $\lambda \rightarrow$ 系统中的函数，也不是一阶逻辑中的函数，其主要通过笛卡儿积来进行定义，即 \mapsto 的定义为

Definition df-mpo 6965	
Description: Define maps-to notation for defining an operation via a rule. Read as “the operation defined by the map from x, y (in $A \times B$) to $C(x, y)$ ”. An extension of df-mpt 4989 for two arguments. (Contributed by NM, 17-Feb-2008.)	
Assertion	
Ref	Expression
df-mpo $\vdash (x \in A, y \in B \mapsto C) = \{ \langle x, y, z \rangle \mid ((x \in A \wedge y \in B) \wedge z = C)\}$	
<i>Distinct variable groups:</i> x, z , y, z , A , B , C <i>Allowed substitution hints:</i> $A(x,y)$, $B(x,y)$, $C(x,y)$	

再接再厉里面还有一个趋于无限的收敛符号，其相当于一个谓词，定义如下

Definition df-rlim 14691	
Description: Define the limit relation for partial functions on the reals. See rlim 14697 for its relational expression. (Contributed by Mario Carneiro, 16-Sep-2014.)	
Assertion	
Ref	Expression
df-rlim $\vdash \rightsquigarrow_r = \{(\langle f, x \rangle \mid ((f \in (C \uparrow_{pm} \mathbb{R}) \wedge x \in C) \wedge \forall y \in \mathbb{R}^+ \exists z \in \mathbb{R} \forall w \in \text{dom } f (z \leq w \rightarrow (\text{abs}'((f^*w) - x)) < y)))\}$	
<i>Distinct variable group:</i> x, w, y, z, f	

其使用的是标准 $\varepsilon - \delta$ 法，只不过量的名字不是 $\varepsilon - \delta$ 而是 $y - z$ 罢了。当然素数定理的证明不可能像这个页面一样只有区区 89 行，实际上我们在第 43 行看到了 pnt2 定理，即

$$\lim_{x \rightarrow \infty} \frac{\theta(x)}{x} = 1$$

我们曾在上一篇文章中讨论过，它其实是素数定理的等价命题。或许到这里我们的讨论就已经结束了，虽然“一阶逻辑 + 集合论”是数理逻辑的初始形式，但在形式验证上并不是一个好的选择，我个人不是很理解为

什么有些人会排斥类型论，而且也不是很看好不基于类型论的证明辅助工具，事实上历史的发展也说明了这点，社区最大的极大证明辅助工具都是基于类型论的，虽然它们的具体有各种形式，但依旧不影响它们处于类型论的框架之下。

7.6 Mizar

接下来的工具是我目前所见中最古老的证明辅助工具了，即Mizar了，虽然好像之前有一个 Automath 但它只是一种形式语言而没有相应的程序工具，Mizar 和 Metamath 持有相同的理论基础，所以我们就同样地来直接欣赏它的证明。这次选择的并不是哥德尔不完全性定理，因为其没有在 Mizar 中被形式化，所以我们选择“二次互反律： $(\frac{p}{q})(\frac{q}{p}) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}$ ”（位于[这里](#)）来撇一瞥

```
theorem Th48: :: INT_5:48
  for X being finite set
    for f being FinSequence of bool X st ( for d, e being Nat st d in dom f & e in dom f & d < e holds
      f . d misses f . e ) holds
      card (union (rng f)) = Sum (Card f)
  proof end;

Lm4: for fp being FinSequence of NAT holds Sum fp is Element of NAT
;

:: WP: The law of quadratic reciprocity
theorem Th49: :: INT_5:49
  for p, q being Prime st p > 2 & q > 2 & p < e q holds
    (Lege (p,q)) * (Lege (q,p)) = (-1) |^ ((p - 1) div 2) * ((q - 1) div 2)
  proof end;

theorem :: INT_5:50
  for p, q being Prime st p > 2 & q > 2 & p < e q & p mod 4 = 3 & q mod 4 = 3 holds
    Lege (p,q) = - (Lege (q,p))
  proof end;
```

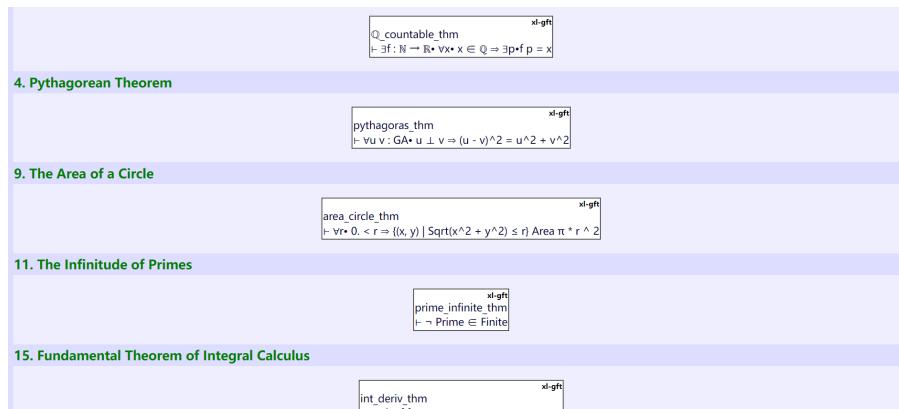
开动我们的超级大脑来观察 Th49，即我们的主题定理。“for”应该表示全称量词 \forall ，“being”是集合论的属于“ \in ”，“Prime”是素数的集合，之前我们说过虽然 “ $\forall x, y \in \text{Prime} A$ ”看起来像高阶逻辑，但它其实是公式 “ $\forall x \forall y (x \in \text{Prime} \wedge y \in \text{Prime} \wedge A)$ ”的简记，“st”表示满足实际就是把全称量词后的括号给省略掉了，“holds”显然是前提推理中的 \vdash ，后面的公式显然也包含了众多缩写，但我们稍微看看名称就差不多能理解了，问题是好像这里只是陈述了一下定理，而并没有证明。实际上，在前面定义完勒让德符号以后，就证明了一系列的等式，此时当我描述完定理时，Mizar 好像会自动来一次类似于其它工具的 auto，而通过上面已经证明的等式可以自动推出二次互反律的等式，从而就可以直接“proof end”了。我建议是跳过 MizarSystem 吧，实在没啥意思。

7.7 ProofPower

不知不觉我们又回到了 HOL 家族的证明辅助工具ProofPower，虽然我很想介绍它，但是它的工具定格在了 2017 年

Name	Last Modified	Size
Parent Directory		
OpenProofPower-2.7.6.tgz	2005-10-21 08:44	2808k
OpenProofPower-2.7.7.tgz	2007-06-16 10:25	2860k
OpenProofPower-2.7.8.tgz	2007-11-27 13:23	2888k
OpenProofPower-2.8.1p2.tgz	2009-10-11 13:26	3108k
OpenProofPower-2.9.1w2.tgz	2011-02-01 15:32	2928k
OpenProofPower-2.9.1w5.tgz	2013-01-06 14:24	2976k
OpenProofPower-2.9.1w8.tgz	2013-08-04 14:23	2980k
OpenProofPower-3.1w1.tgz	2014-04-23 13:17	2984k
OpenProofPower-3.1w2.tgz	2014-06-15 14:48	4616k
OpenProofPower-3.1w3.tgz	2015-03-14 16:53	4668k
OpenProofPower-3.1w4.tgz	2015-03-14 16:54	4668k
OpenProofPower-3.1w5.tgz	2015-04-17 15:49	4796k
OpenProofPower-3.1w6.tgz	2015-06-26 14:48	4812k
OpenProofPower-3.1w7.tgz	2016-03-12 16:43	4820k
OpenProofPower-3.1w8.tgz	2017-06-19 11:27	4824k

而且虽然我们有相应的定理描述



比如“任意半径为 $r>0$ 的圆的面积为 πr^2 ”，但是我们无法看到它的证明过程，那么我到底该怎么进行实例讲解呢？可见问题不在于我，而在于 ProofPower 自身的发展十分乏力，若不是它在 100 问中被提及，我都不知道有这东西，更不会在这里提起它了。

7.8 ACL2/nqthm

由于 ACL2 的本意为，nqthm(Boyer-Moore theorem prover) 的工业增强版，因此我们在工具 ACL2 后面附上一个 nqthm 以表敬意。我们本来想用它来讲解“哥德尔不完全性定理：包含数论的一致系统存在无法证明真假的命题”的，但奈何我们无法找到任何与之相关的参考资料就只能鸽掉了。

7.9 PVS

值得注意的是，上一个 ACL2 系统、现在要讲的 PVS 系统和下一个 NuPRL 系统都是基于 Common Lisp 实现的。Lisp 啊，一个特别老的高级语言了，在计算机技术高速发展的现在，除非语言的社区十分庞大（比如 C/C++），我还是持有推新不推旧的观点。与 ACL2 不同，PVS 的数学库倒是挺丰富的，这次我们欣赏“有无穷个素数”的证明（[在这里可以找到](#)）

The screenshot shows a PVS library page for the proof of the infinitude of primes:

Code

```

1 infinite_primes: THEORY
2 BEGIN
3   IMPORTING intagdivides_lees,
4           realagproduct_fseq_posnat,
5           prime_factorization
6   Primes: set[nat] = {n: nat | prime?(n)}
7
8   n: VAR posnat
9   prime_divides: LEMMA NOT prime?(n) AND n > 1 IMPLIES
10    (EXISTS (p: nat): prime?(p) AND divides(p, n))
11
12   i1: VAR nat
13   divides_a_product: LEMMA FORALL (f: fseq[posnat]),
14    (i1: below(length(f)));
15    divides(seq(f)(i1), product(f))
16
17   primes_infinite: THEOREM NOT is_finite(Primes)
18
19 END infinite_primes

```

说实话这个证明挺丑的，连高亮都没有。我尽力地看大概读出了，引理 prime_divides 表示“合数有一个素因子”，剩下的读不懂了，不过使用应该是传统的数论方法，不涉及解析理论。太难太累了，跳过吧。

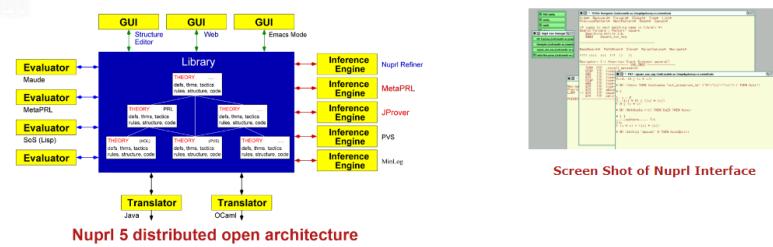
7.10 NuPRL/MetaPRL

这两个工具都隶属于 PRL 计划，其中 NuPRL 是 PRL 系统的核心工具，而 PRL 的最初思想来自于 MetaPRL 系统（官网已经废弃了），所以我们同样在 NuPRL 后面附上一个 MetaPRL 以表敬意。

Formal Digital Library (FDL)

FDL (Formal Digital Library) was a generic name for Nuprl 5.

The FDL was intended to be a web service.



Nuprl 4

Nuprl 4 is no longer supported. If you have Nuprl 4 theories which you would like migrated to Nuprl5, please send an email to nuprl@cs.cornell.edu for assistance.

Nuprl4 shares many refiner and editor features with Nuprl5. The Nuprl4 documentation may provide some information not yet included in the nuprl5 docs.

Nuprl 4 User Documentation

MetaPRL (metaprl.org)

MetaPRL was designed and implemented by Jason Hickey as part of his PhD research. Their system is described in detail in his PhD thesis and in several subsequent papers with users and collaborators. MetaPRL is a logical framework in the spirit of Isabelle and Twelf with the added capability of relating the logics. Jason first implemented the type theory of Nuprl, Computational Type Theory. MetaPRL is implemented in OCaml, Alexey Nogin and Alexei Koplyov used MetaPRL extensively and contribute to its evolution. It is now used at Hughes Research Laboratory and at Moscow State University among other places. The MetaPRL system combines the properties of an interactive LCF-style tactic-based proof assistant, a logical framework, a logical programming environment, and a formal methods programming toolkit. MetaPRL is distributed under an open-source license and can be downloaded from <http://metaprl.org/>.

MetaPRL-A Modular Logical Environment

This paper provides an overview of the system focusing on the features that did not exist in the previous generations of PRL systems.

根据官网内容，MetaPRL 已经作为 NuPRL 的一部分推断引擎而存在了，所以我们要以标题的第一名称为准。这次我们稍微看看“三角数倒数和公式： $\sum_{i=1}^{\infty} \frac{2}{n(n+1)} = 2$ ”³

The screenshot shows the NuPRL Project website's Proof Library section. The top navigation bar includes links for PRL Home, Introduction, Math Library, Publications, Projects, System, Seminar, People, The Book, Lectures, and Other Groups. The main content area is titled "Browse NuPRL Library Items by Name" and shows a search bar with "Freek 100" and "Index". Below the search bar, it says "Highlights: Minimal First Order Logic". A step of a lemma is shown in the code editor:

```
Step _ of Lemma triangular-reciprocal-series-sum
Σₙ. (r1/r(t(n + 1))) = r(2)
BY
{ Assert ``lim n→∞.r(2) - (r(2)/r(n + 2)) = r(2)``
  1.....assertion.....
  lim n→∞.r(2) - (r(2)/r(n + 2)) = r(2)
  2
  1. lim n→∞.r(2) - (r(2)/r(n + 2)) = r(2)
  ⊢ Σₙ. (r1/r(t(n + 1))) = r(2)

Latex:
\mSigma \emptyset \Sigma \Sigma . (r1/r(t(n + 1))) = r(2)
```

一眼鉴定是裂项相消法，只不过有点小难看，算了还是 Pass 吧。

7.11 Haskell

实际上，任何一门函数式编程语言的理论基础都是 λ 系的类型论，因此很多函数式语言都能成为交互式证明助手，或者用于实现编译器或解释器在编译期的检查。那么作为纯函数式编程语言的代表，Haskell，就值得稍微提一提了，不过我们确实就只是提一提了，我们很少有数学证明是 Haskell 上进行验证的，而是基于 Haskell 开发的工具之上，也就是我们将要引出的 Agda。

³位于 <https://www.nuprl.org/wip/>，但其是动态页面，需要点击 Browse Index by Name 后的 Freek 100，并找到第 42 问才行

7.12 Agda/Idris

工具Agda和Idris虽然没有什么本质关系，但它们的理论基础都是类似于 Martin-Löf 类型论 (MLTT, 也可以称为 Intuitionistic type theory, ITT) 的 UTT(unified theory of dependent types, 来自LuoZhaoHui论文“Computation and Reasoning: A Type Theory for Computer Science”), 而 Idris 在使用和 Agda 类似的类型系统时，采用了类似于 Coq 的证明系统，即策略 (tactics) 推导，而 Agda 就比较原始了，在证明上比较像函数式编程。Idris 虽好但社区好像不咋活跃，所以对于想了解 MLTT 系类型论的人选择 Agda，并且其还有相应的同伦类型论 (HoTT, Homotopy Type Theory, 参考[这里](#))，目前HoTT就只有在 Coq 和 Agda 上实现。虽然我们确实有很多的类型论，但实际上它们就只是改变了项的形成规则、判断的推导规则等形式内容，并且有时为了适应计算机的运作而做出调整罢了，不论做出多少的改变，类型论的内容框架“enviroment;context;element:Type”和推理框架“ $\frac{\text{premiss.1 premiss.2 ... premiss.n}}{\text{conclusion}}$ ”是永恒不变的。

7.13 Arend

实际上，各种类型论之间都存在一定的互译能力，例如基于 MLTT 系类型论 UTT 的 Agda 和基于 λ 系类型论 CoC 的 Coq，都实现了 HoTT 类型论，那么 HoTT 是什么东西呢？简单来讲就是基于范畴的观点建立类型论和同伦类之间的联系，不用质疑这里的同伦论就是代数几何中的那个，但这里的范畴论不是大家通常认识的那个，而是具有严格逻辑形式的范畴论 (参考书籍“Categorical Logic and Type Theory”)，我以前就说过范畴论的严格性必需来自逻辑学。这么一看形式系统的理论实在太多又太杂了，这也正常本来这就是针对“书写的演算”的研究，因此除了数学家，搞计算机的、搞哲学的、搞逻辑的、搞语言学的等等的人都混了进来，这也是无可奈何的事情。回到 HoTT，Arend就自称是基于 HoTT 的定理证明器 (theorem prover, [Wiki](#))，比较遗憾的是我并没有在官网找到相关的证明例子，自然也就只能提一下而不了了之了。

7.14 F*

当时我就觉得F*的样子也太吸引人了吧，就想着去了解一下，不过也就是看了一下官方介绍。

F* (pronounced F star) is a general-purpose proof-oriented programming language, supporting both purely functional and effectful programming. It combines the expressive power of dependent types with proof automation based on SMT solving and tactic-based interactive theorem proving.

F* programs compile, by default, to OCaml. Various fragments of F* can also be extracted to F#, to C or Wasm by a tool called [KaRaMeL](#), or to assembly using the [Vale](#) toolchain. F* is implemented in F* and bootstrapped using OCaml.

F* is open source on [GitHub](#) and is under active development by [Microsoft Research](#), [Inria](#), and by the community.

读者也就同样地自行观看一下，我们要在这里进行小结了。在[这里](#)，FreekWiedijk 列举了大量与数学相关的计算机工具，不过大多数基本都是要么销声匿迹要么十分小众，实际上由于存在大量可参考的先例，设计开发一门语言并制作编译器并不困难，编程最重要的是社区，放到证明辅助工具上也是一样的。

在证明辅助工具上，我最喜欢使用的 Lean4，不论是写代码还是数学库还是社区活跃度都比较符合我的口味，四大工具中的其它三个 Coq、Isabelle 和 HOL Light，只是我拿来看些独占定理用的，如果不想使用类型论了，想回归纯正的“一阶逻辑 + 集合论”，我会选择 metamath，它的数据库和索引都做得比较好，此外的工具于我而言真就不怎么重要了，人的精力是有限的，学会适当的取舍是十分有必要的。

参考文献

- [1] Chen C. Chang and H. Jerome Keisler. *Model Theory*. North Holland, 1990.
- [2] David Marker. *Model theory : An Introduction*. Springer New York, NY, 2002. URL: <https://doi.org/10.1007/b98860>.
- [3] M. Ram Murty and Brandon Fodden. *Hilbert's Tenth Problem: An Introduction to Logic, Number Theory, and Computability*. STUDENT MATHEMATICAL LIBRARY Volume 88. American Mathematical Society, 2019.
- [4] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.
- [5] Sara Negri and Jan Von Plato. *Structural Proof Theory*. Cambridge University Press, 2008.
- [6] Hiroakira Ono. *Proof Theory and Algebra in Logic*. Short Textbooks in Logic. Springer, 2019.
- [7] Raymond M. Smullyan and Melvin Fitting. *Set theory and the continuum problem*. OXFORD LOGIC GUIDES: 34. Clarendon Press, Oxford, 1996.
- [8] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.
- [9] 余俊伟, 赵晓玉, 裴江杰, 张立英. 数理逻辑. 新编 21 世纪哲学系列教材. 中国人民大学出版社, 2020.
- [10] Michael Sipser, 著. 唐常杰, 陈鹏, 向勇, 刘齐宏, 译. 计算理论导引: *Introduction to the Theory of Computation*. 计算机科学丛书. 机械工业出版社, 2006.
- [11] 逯晓零. 代数分析几何简介. <https://lixing48.gitee.io/download/AlgebraicAnalyticGeometry.pdf>.
- [12] 逯晓零. 数论大观园. <https://lixing48.gitee.io/download/NumberTheory.pdf>.
- [13] 逯晓零. 朗兰兹纲领简介. <https://lixing48.gitee.io/download/LanglandsProgram.pdf>.
- [14] 郁文生, 孙天宇, 付尧顺. 公理化集合论机器证明系统. 数学机械化丛书. 科学出版社, 2019.