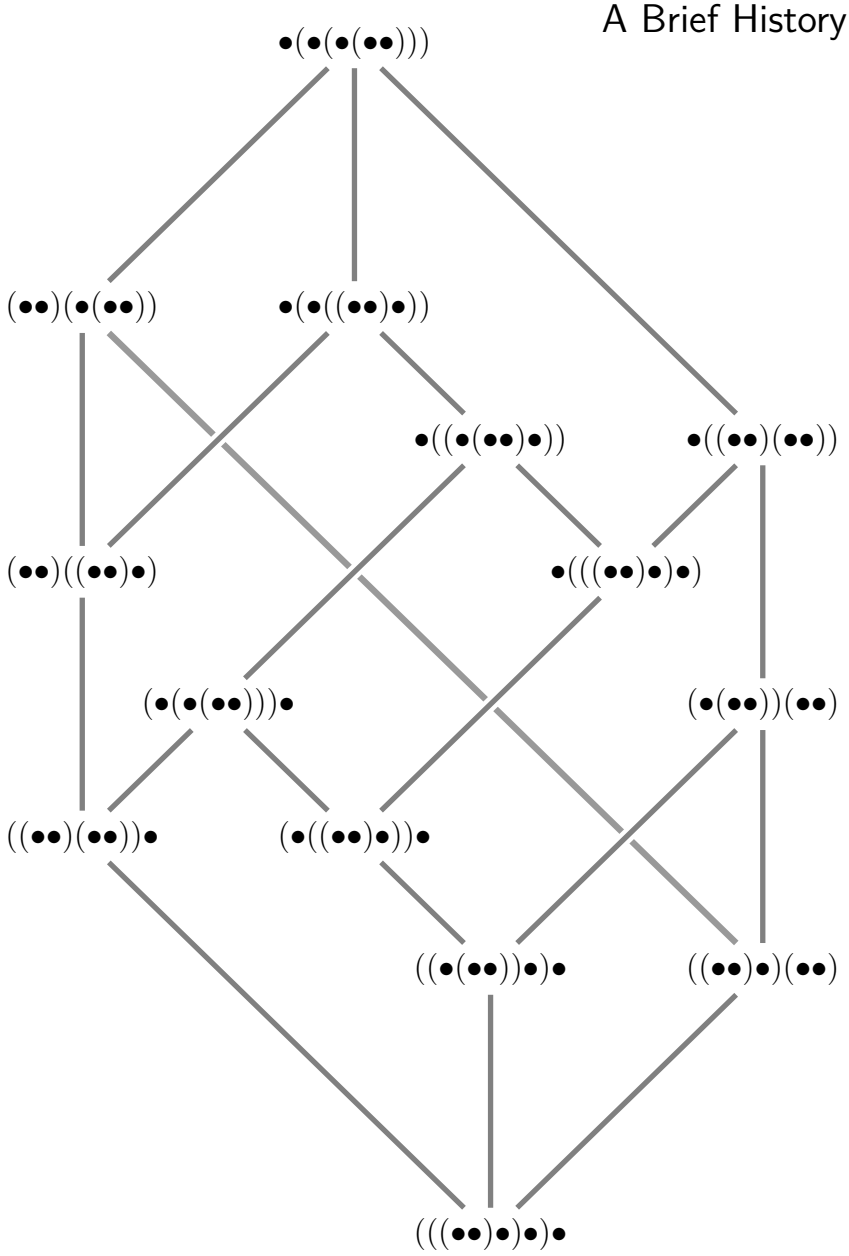
类型论简史

A Brief History of Type Theory

Trebor



Category theory implements the relative point of view by means of *slice categories*; but this language is greatly obfuscated in comparison to the simplicity of working with non-relative objects. The goal of type theory is to reconcile the expressivity of the relative point of view with the simplicity of the global point of view, by providing a language that makes movement between different slices (base change) seamless.

J. Sterling, *Towards a geometry for syntax*. [57]

目录

第一章 前言 **1**
1.1 什么是类型论? . . . 2
1.2 朴素的语法 . . . 3
1.3 客观逻辑与主观逻辑 . . . 4
1.4 类型论的语义 . . . 6
1.5 类型论的历史 . . . 9

第二章 起源 **11**
2.1 Russell 的类型论 . . . 11
2.2 标准语义与 Henkin 语义 . . . 13
2.3 λ-演算与组合子逻辑 . . . 14
2.4 简单类型 λ-演算的典范化 . . . 18
2.5 论域论 . . . 28

第三章 Curry–Howard 对应 **31**
3.1 简单类型 λ-演算与命题逻辑 . . . 31
3.2 依值类型 . . . 32
3.3 经典逻辑 . . . 36

第四章 Martin-Löf 类型论 **45**
4.1 归纳类型 . . . 45
4.2 相等类型 . . . 46
4.3 宇宙 . . . 48
4.4 自洽性 . . . 49
4.5 应用 . . . 50

第五章 范畴语义 **53**
5.1 范畴与模型 . . . 53

5.2 类型论的自然模型 54
5.3 融贯问题 58
5.4 范畴语义的历史 60
5.5 意象与内语言 60

第六章 同伦类型论 **61**
6.1 K 原理的独立性 61
6.2 同伦类型论 62
6.3 同伦类型论的语义 66
6.4 立方类型论 67

第七章 展望 **69**

第一章　前言

2020 年 12 月, Peter Scholze 发布了一项挑战 [52]. 在他的凝聚态数学前沿研究中有一条技术性定理, 对于这个领域有至关重要的作用:

定理 1.1 (Clausen, Scholze)**.** *设 $0 < p' < p \leq 1$ 为实数, 令 S 为投射有限集, V 为 p-Banach 空间. 记 $\mathcal{M}_{p'}(S)$ 为 S 上的 p'-测度构成的空间. 那么对于 $i \geq 1$ 有*

$$\mathrm{Ext}^i_{\mathsf{Cond(Ab)}}(\mathcal{M}_{p'}(S), V) = 0.$$

Scholze 希望这条定理的证明可以被完全严格形式化, 并且通过计算机的检查. 2022 年 7 月, Lean 社区宣布这个挑战正式完成. 换句话说, 即使是数学最尖端的结论, 也可以在一年半的时间内被转化成计算机可以理解并验证的代码. 这种成就是如何达成的呢? 答案是 Lean 的类型论.

在类型论中, 一切数学对象的含义都由它们从属的类型决定. 如类型 $\mathbb{N}$ 的元素是自然数, 而类型 $\alpha \times \beta$ 的元素是有序对, 类型 $\alpha \to \beta$ 的元素是映射, 等等. 而这些类型可以相互组合, 表达出复杂的含义, 这使得类型论有能力作为数学的基础, 与集合论的地位类似. 另一方面, 计算机科学中也有利用类型描述程序的传统. 因此, 类型论可以看作是数学与计算机之间的桥梁. 这也使得计算机定理验证成为了有可行性的工作: 证明可以写成计算机能解析的语言, 并且交由计算机验证正确性. 传统的使用一阶逻辑与集合论数学基础中, 一个初等的定理也可能需要冗长的说明才能完全严格的写出来, 因此尽管理论上可能, 实操上却非常困难. 类型论不仅可以验证数学定理. 现在, 类型论正广泛用于验证各种需要低出错率的软硬件, 如军事、医用、宇航电子仪器等设计无误.

这篇文章的主要目的是以历史作为连贯的主线将类型论的知识串联在一起, 为中文学习者提供些微的参考价值. **这不是类型论的教程**, 但是文章中介绍各个领域时都会提及其中优秀的参考书目或者其他文献资料, 供希望进一步了解的读者阅读. 对于一些没有标准翻译的名词, 会在括号中附上英文名称.

文章不要求任何类型论的前置知识, 并且尽量避免范畴论知识的使用, 但是会援引一般数学中的许多例子. 同时, 每一节的内容相对独立, 因此如果某一部分使用的前置知识较多, 读者可以直接跳过, 不影响后续阅读. 在使用范畴论知识时, 会使用如下的图标标示难度:

草莓籽表示不需要范畴论知识, 或者只需要文内已经提到的知识. 红色草莓表示需要读者对基础的范畴论定义有熟悉度. 金草莓意味着读者需要掌握范畴之间的常用操作 (米田引理等). 月莓图标表示读者需要对意象论的各种构造有直观理解.

在文章写作过程中, 类型论社区提供了巨大的帮助, 包括查找一些事实错误, 提供优秀的参考资料, 修改措辞等等. 特别地, *n*Lab 上记有许多令人醍醐灌顶的观点, 但是由于不是期刊书籍, 它的功劳很难以学术通用的形式得到应有的承认.

1.1 什么是类型论?

从最表层的角度说, 类型论研究的是对象和它们的类型之间的相互作用. 比如说 1 的类型是 $\mathbb{N}$, 而函数的类型是 $\mathbb{N} \to \mathbb{N}$; 一个泛函的类型则可能是 $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$. 大部分类型论的介绍都会采取同一个思路: 一套类型系统[1]就是一组规则的集合, 我们应用这些规则推导一些结论. 这些结论被称为**断言** (judgement). 譬如 "$1 + 1 = 2$" 就可以是某个类型论中的断言. 而类型论就是研究这些断言的学科.

但是纵观类型论 120 年的历史, 这其实是一个有失偏颇的概括. 它只注意到了类型论的**语法** (syntax), 或者称为 "语形" 的研究. 而语法与**语义** (semantics) 是分不开的. 在这里, 我提出这篇论文的中心:

类型论是语法与语义研究的对立统一.

这就将问题拆分为三个部分:

- (1.2 节) 什么是语法?
- (1.4 节) 什么是语义?
- (1.5 节) 二者的研究是如何对立统一的?

这一章剩下的部分将会简要地从当代视角解释这些问题, 读者也可以直接跳过阅读下一章. 类型论不仅局限于数学, 它在计算机科学与哲学中都有重要的地位. 因此这篇文章会尽可能多地谈及各方面的例子, 以让读者多了解不同学科视角下的思想.

[1] 类型系统也叫做类型论, 而上面说的是类型论这门学科, 就像拓扑这门学科研究的是拓扑这种数学对象一样.

1.2 朴素的语法

什么是语法? 最朴素, 也是人们最初的理解, 就是从所有的字符串 (字符串就是某个固定的字母表中的符号组成的有限长的列表) 中选出一个子集, 规定为 “符合语法” 的. 譬如我规定字母表是 $\{0,1,2,3,4,5,6,7,8,9,+,\times\}$, 然后写出一些规则, 使得 $1+3\times 25$ 是符合语法的, 而 $++3\times 03$ 不是.

这些规则一般是这样:

$$\frac{(n\in\{0,1,\dots,9\})}{n \text{ number}} \quad \frac{n \text{ number} \quad (m\in\{1,\dots,9\})}{mn \text{ number}}$$

$$\frac{n \text{ number}}{n \text{ valid}} \quad \frac{n \text{ valid} \quad m \text{ valid}}{n+m \text{ valid}} \quad \frac{n \text{ valid} \quad m \text{ valid}}{n\times m \text{ valid}}$$

这种写法来自逻辑学的传统, 上面是前提, 下面是结论. 如果用数学中更常见的写法, 就是譬如 “如果 n 和 m 是符合语法的表达式, 那么 $n\times m$ 也是符合语法的表达式.” 并且所有符合语法的表达式是在这些规则下封闭的最小集合.

这种原始的语法, 优点在于非常简单. 如果我们希望创造一种数学基础, 我们当然不希望定义这个基础的时候就用到大量的数学概念. 字符串和它们的拼接 —— 这再简单不过了, 任何人都能够确信自己可以正确理解并操作. 类型论创始时正是为了解决 Russell 悖论引起的基础危机, 所以这一点尤其重要.

然而这种语法的缺陷在正式应用时就会立刻体现出来. 譬如在处理变量的时候: 我们都很清楚 $\int x\,\mathrm{d}x$ 中没有**自由变量**, 只有**束缚变量** x. 因此它写成 $\int y\,\mathrm{d}y$ 是完全一样的. 同样的道理, 如果 $f(x)=\int_0^1 xy\,\mathrm{d}y$, 那么计算 $f(y^2)$ 的时候, 我们不能简单的代入

$$f(y^2)\overset{?}{=}\int_0^1 y^2\cdot y\,\mathrm{d}y.$$

正确的做法是先将束缚变量 y 改成另一个名字, 再代入. 把数学表达式作为字符串处理时, 这种问题是尤其头疼的. 更不用说还需要证明每个字符串至多有一种解读的方式, 等等.

然而这些问题都是一些鸡毛蒜皮的小事, 都可以通过机械而繁琐的步骤处理. 在度过基础危机时期之后, 我们理应放下过于谨慎的态度. 对于形式系统的研究, 我们绝大部分的精力都不应该放在处理这种事情上, 而应该讨论一些真正重要的问题: 这个系统有没有矛盾? 这个系统能不能证明这些重要的定理? 这些问题, 与用字符串时产生的这些技术细节是不相干的.

那么, 我们应当换一个视角看这个问题: 字符串本身并不是语法. *它只是表示语法的一种方式*. Klein 群并不是四个矩阵; 这四个矩阵只是表示这个群的一种方式. 我们的实数并不是 Cauchy 收敛的数列的等价类; 这些等价类只是编码实数的一种方式. 对于语法来说, 我们感兴趣的并不是这些字符串的性质, 而是这些字符串表示的东西. 换言之, 我们可以认为存在[2]一个 “理想” 的语法, 而我们使用字符串或者其他什么方式描述的只是表示这种语法的

[2] 当然, 一个数学对象是否 “存在” 是一个数学哲学问题, 这里不过多讨论.

一种方式, 我们只不过没有更好的办法描述这种理想的语法而已. 这就引出了**客观语法**的概念.

1.3 客观逻辑与主观逻辑

这两个概念源于 Hegel. Hegel 认为逻辑分为两种. 其中主观逻辑类似于我们数学中称的逻辑, 处理的是重言式而没有实质内容. 而客观逻辑则处理的是概念的发生学, 把哲学上 (如 Kant) 对概念的划分从任意武断的变为理性必然的.[3] 我们可以把客观逻辑的研究中遇到的问题编码成主观逻辑可以处理的问题, 进而用主观逻辑进行推导; 但是由于主观逻辑自身的局限性, 这些终究只是一种编码. (与之相对地, Russell 等人认为仅仅使用形式逻辑就可以研究一切的哲学问题, 这开创了分析哲学学派.)

数学家 Lawvere 将这两个概念拿到了数学中来. 他认为主观逻辑是客观逻辑的一部分, 主观逻辑反映了客观逻辑, 并且部分指导客观逻辑的构造.

> 五十年前, 源于几何学的需求, 范畴论创造了譬如伴随函子、意象、纤维化、闭范畴、2-范畴等等的事物. 这些事物是为了帮助阐明一些复杂但并非随意构造的概念, 以及这些概念之间的相互作用; 这些概念产生于对空间和数量的研究的需要. [...] 如果我们把 [上面这句话] 其中的 “空间和数量” 替换成 “任何一项严肃的研究”, 那么这就是我姑且对**客观逻辑**的定义. [41]

Lawvere 认为, 范畴论就是客观逻辑的一个数学模型. 我们在数学中的定义—定理—证明过程都是由主观逻辑 (即形式逻辑) 主导的, 但是这些定义的动机、定理的直观、证明的思路, 都是来自于客观逻辑. 主观逻辑不能刻画全部的客观逻辑, 而只是其一部分, 并且承担着指导作用, 正如铁轨指导火车的运行. 同时, 我们交流时不能直接交流客观逻辑中的理念, 而只能先将它们编码成主观逻辑中的命题.

回到语法上来. 我们自然可以如上一节所举的例子那样定义一套语法, 但是这一套语法背后反映的实际上是我们对于四则运算的概念. 因此譬如括号匹配, 优先级等等概念, 实际上都是这种具体的编码 — 也就是**主观语法** — 中的细节. 而这个主观语法所意图反映的概念才是我们真正希望研究的, 也即**客观语法**. 这与下面提到的语义的概念有所区别, 下一节介绍语义时再详细阐述. 与主客观逻辑类似, 客观的语法是不能落脚到计算机上来的, 因此主观语法仍然是有必要的. 对于主观语法和客观语法的研究, 构成了对立统一.

作为 Lawvere 思想的一种具体化, 客观语法可以由范畴论刻画, 特别地, 由意象 (topos) 刻画. 这一思想 (据笔者考证) 是由 Jonathan Sterling 提出的. 我们在之后的章节里会逐步介绍具体的细节. 当代的类型论研究重点已经逐渐从主观的语法转移到客观的语法上来. 正如 Sterling 所说:

[3] 笔者对哲学不完全了解, 如果有不妥请尽管指出.

> 这篇学位论文立足于一项重要的观察：类型论中的定理可以用一种在范畴的等价下不变的方式叙述；因此，我们没有必要过于关注具体的树状表示或者变量绑定的等等细节．现在看来，对这种细节的过度关注与本科计算机科学教育中对所谓“语法解析 (parsing)”，或者本科逻辑学教育中对括号的匹配计数过度强调的不良风气是一脉相承的．[56, (4.2∗3)]

这一节的最后，我们举一个例子：群的语法．它的主观语法比较好理解．首先我们有一些作为变量的符号 x, y, z 等．每个变量都是一个表达式．如果 M, N 是表达式，那么 $(M \cdot N)$ 和 M^{-1} 也是表达式．同时 1 也是表达式．这样，表达式就例如 $(1^{-1} \cdot ((x \cdot y^{-1})^{-1} \cdot x))$ 等等．

对于这套简单的理论，主观语法还是比较好处理的．我们可以归纳定义一个表达式的变量集 $\mathrm{Var}(M)$．上面那个表达式的变量集就是 $\{x, y\}$．我们可以证明每个表达式都只有唯一的一种方式从上面的规则中产生．

然后我们还需要描述群元素间的相等．这在一般的类型论中称为**判值相等** (judgemental equality)．是理论中内生的相等性．与它相关的辨析也会在第四章中讲述．我们定义表达式之间的一个等价关系，这次使用逻辑学家更熟悉的语言：

$$\frac{}{M = M} \quad \frac{M = N}{N = M} \quad \frac{M = N \quad N = P}{M = P}$$

这规定了 $=$ 是个等价关系．

$$\frac{M = N}{M^{-1} = N^{-1}} \quad \frac{M_1 = M_2 \quad N_1 = N_2}{(M_1 \cdot N_1) = (M_2 \cdot N_2)}$$

这规定了判值相等是**合同** (congruent) 关系，换言之，把一个表达式 M 中的一部分 N 替换成与之相等的另一个表达式 N'，得到的新表达式 M' 也等于原来的表达式 M．

$$\frac{}{(1 \cdot M) = M} \quad \frac{}{(M \cdot 1) = M}$$

$$\frac{}{((M \cdot N) \cdot P) = (M \cdot (N \cdot P))} \quad \frac{}{(M \cdot M^{-1}) = 1} \quad \frac{}{(M^{-1} \cdot M) = 1}$$

注意这些规则都是没有前提，因此是恒成立的．这就规定了群的主观语法．用代数的眼光看，这刚好是在构造由变量符号生成的自由群．这个视角是非常深刻的，后面还会多次碰见．

那么客观语法如何呢？这需要一定的范畴论工具．我们后面会详细介绍范畴论的知识．这里只预先为读者提供一个大略的图像．我们考虑一列群 $F_n (n \in \mathbb{N})$，即有 n 个生成元的自由群．它们之间的群同态构成一个范畴．我们取这个范畴的对偶 $\mathcal{T}$．这个范畴代表了上面的主观语法的客观化，也称为群论的**语法范畴**．我们会在下一节更仔细地讲述这个范畴的特性，以及它如何与主观语法相对应．但是现在可以为已经掌握了一定范畴论知识的读者罗列几点：

- 从 $\mathcal{T}$ 到集合范畴 Set 的保积函子与群一一对应．
- 从 $\mathcal{T}$ 到拓扑空间范畴 Top 的保积函子与拓扑群一一对应．

- 从 $\mathcal{T}$ 到光滑流形范畴 Man 的保积函子与 Lie 群一一对应.

这里保积函子指的是保持 Descartes 乘积的函子.

1.4 类型论的语义

语义是什么? 顾名思义, 就是指某种解释语言的方式. 我们还是以群的语言为例子. 群的语言中有 $M \times N$ 与 M^{-1} 的操作, 那么我们朴素的语义中就可以解释为一个集合 G 以及其上的两个函数 $m : G \times G \to G, i : G \to G$. 而表达式 1 就解释为 G 中选定的某个元素 e. (其实这可以认为是 “零元函数” , 因为 $G^n \to G$ 就是 n 元函数, 取 $n = 0$ 得到 $G^0 \to G$, 即从单点集出发的函数.) 这样, 任何表达式就都能通过这些函数解释成具体的运算结果. 我们进一步要求选定的函数需要满足语法中规定的所有等式. 这样得到的语义我们已经很熟悉了: 一个群恰好是一种语义. 同时这也告诉我们, 同一个语言也可以有不同的语义, 甚至不一定有典范的语义.

当然, 这只是把语言解释成了集合和函数. 我们没有理由把自己限制到这些事物上. 譬如说我们把 “集合” 改为 “光滑流形” , “函数” 改为 “光滑映射” . 读者可以发现这就是 Lie 群的定义. 这就促使我们思考, 怎样的数学结构就足够支撑语义的定义呢? 我们需要一个抽象的数学结构, 包含一些对象和它们之间的映射. 这就自然的产生了范畴的概念.

定义 1.1. 一个**范畴** $\mathcal{C}$ 包含

- 一个集合 $\mathrm{Obj}(\mathcal{C})$, 其中的元素称为范畴 $\mathcal{C}$ 的**对象**.[4]
- 对于每对对象 A, B, 一个集合 $\hom_{\mathcal{C}}(A, B)$, 称为从 A 到 B 的**态射**或者**箭头**. 对于每个对象 A, 都有一个态射 $\mathsf{id}_A \in \hom_{\mathcal{C}}(A, A)$, 称作**恒同态射**. 对于两个态射

$$A \xrightarrow{f} B \xrightarrow{g} C$$

有其**复合态射** $g \circ f \in \hom_{\mathcal{C}}(A, C)$.

它们满足一些等式. 恒同态射与其他箭头的复合满足 $f \circ \mathsf{id} = f = \mathsf{id} \circ f$. 并且三个顺次相连的态射复合满足结合律 $f \circ (g \circ h) = (f \circ g) \circ h$.

那么集合与函数就构成一个范畴 Set. 读者可以验证函数的复合满足所需要的等式. 群与群同态也构成一个范畴 Grp, 等等. 进一步, 范畴之间保持其结构的映射就称为函子.

定义 1.2. 一个**函子** $F : \mathcal{C} \to \mathcal{D}$ 包含其对象的映射 $\mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{D})$ 与每对对象之间的态射的映射 $\hom_{\mathcal{C}}(A, B) \to \hom_{\mathcal{D}}(F(A), F(B))$. 它满足 $F(\mathsf{id}_A) = \mathsf{id}_{F(A)}$ 与 $F(f \circ g) = F(f) \circ F(g)$.

[4] 有时候我们希望考虑如所有集合构成的范畴, 此时由于所有集合的集合不存在, 我们需要将这个定义放宽. 这里不考虑这些问题, 但是在严格处理的时候这些问题是非常重要的.

再进一步, 函子之间保持其结构的映射称为自然变换.

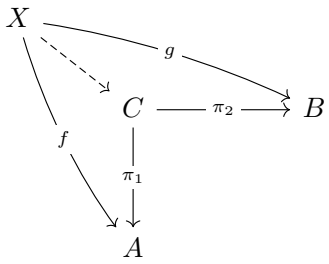
定义 1.3. 对于两个函子 $F, G : \mathcal{C} \to \mathcal{D}$, 一个**自然变换** $\eta : F \to G$ 为一族态射 $\eta_X : F(X) \to G(X)$, 满足对于任何 $f : X \to Y$, 下面的正方形图表交换 (即两条路径复合得到的是相等的态射).

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \eta_X}\downarrow & & \downarrow{\scriptstyle \eta_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

范畴论是非常丰富而深刻的学科. 读者可以阅读 [55] 或 [45] 进一步学习.

我们有一个范畴还不够, 我们需要有对象的乘积来表达二元函数 $G \times G \to G$. 在范畴中可以直接定义对象之间的乘积.

定义 1.4. 给定一个范畴中的两个对象 A, B, 如果有另一个对象 C 以及两个态射 $\pi_1 : C \to A$, $\pi_2 : C \to B$, 满足对于任何一对态射 $f : X \to A$, $g : X \to B$, 都存在一个态射 $u : X \to C$ 使得 $f = \pi_1 \circ u$, $g = \pi_2 \circ u$, 就称 C 为 A, B 的 **Descartes 乘积**, 记作 $A \times B$.



读者可以验证集合的乘积、群的直积、向量空间的直积等等都满足条件. 因此这个定义恰当地刻画了我们称为积的许多不同定义. 范畴乘积在同构意义下是唯一的.

同时, 我们还需要表达 “零元函数” 的概念.

定义 1.5. 如果一个范畴中有一个对象 1 满足任何对象 X 到 1 都有唯一的态射, 那么这个对象称作**终对象**. 对偶地, 如果有一个对象 0 满足它到任何对象都有唯一的态射, 那么这个对象称作**始对象**.

终对象可以看成是 “零元乘积” . 在集合范畴中, 这就是单点集的定义. 我们有 $1 \times X$ 与 X 同构等性质. 有了这些工具之后, 我们就可以在任何含有乘积和终对象的范畴中定义群的语义了. 我们把群的语义定义为一个对象 G, 态射 $i : G \to G$, $m : G \times G \to G$, $e : 1 \to G$. 它们需要满足群的等式. 如考虑 $\mathsf{id}, i : G \to G$ 这两个态射, 由范畴乘积的定义, 存在态射 $\langle \mathsf{id}, i \rangle : G \to G \times G$. 这个态射可以与 m 复合. 另一方面由终对象的定义, $G \to 1$ 有唯一的态射, 它可以与 e 复合. 我们要求这两个复合映射 $G \to G$ 相等. 这在集合范畴里说的就是 $x \cdot x^{-1} = e$. 其他的等式也可以类似地重新表述.

为什么要花费这么大功夫, 把原来的等式表达得这么复杂呢? 这是因为我们重新表述了这个等式后, 它就可以立刻解放而用到其他范畴中去.[5] 譬如我们现在可以考虑光滑流形与可微映射构成的范畴, 那么就立刻得到 Lie 群的定义. 如果在代数簇的范畴中, 就得到代数群的定义.

在 1963 年, Lawvere [40] 提出了一种更优雅的表述方式. 如果我们希望考虑一个群 G 中的两个元素 x, y 使得 $x^2y^2 = e$, 那么我们可以将其表述为从 $\langle x, y \mid x^2y^2\rangle$ 到 G 的群同态. 这是由 x, y 两个元素在满足我们需要的等式的前提下自由生成的群. 如果读者接触过逻辑学, 那么可以类比 Lindenbaum–Tarski 代数, 它也蕴含着类似的思想. 类似地, 我们现在希望考虑某个范畴中的一个对象以及态射 i, m, e, 满足一些等式. 因此我们应该寻找某个范畴 $\mathcal{T}$, 它由 i, m, e 三个态射在满足我们需要的等式前提下 “自由生成” .

我们考虑由一个对象 $\star$ 以及它自身的有限次乘积 $\star^n (n \in \mathbb{N})$ 构成的范畴. 我们考虑这样构成的范畴 $\mathcal{T}$ 到其他范畴的函子. 如果这个函子保持乘积结构, 那么它就由 $F(\star)$ 的值唯一确定. 正如 $\mathbb{Z}$ 到环 R 的环同态 f 由 $f(1)$ 的值唯一确定. 如果我们进一步在范畴 $\mathcal{C}$ 中自由加入一个态射 $m : \star^2 \to \star$, 同时不破坏乘积结构, 那么从这个范畴出发的 (保乘积结构的) 函子就会由 $F(\star)$ 与 $F(m) : F(\star) \times F(\star) \to F(\star)$ 唯一决定. 如果 m 在范畴 $\mathcal{T}$ 中满足一些等式, 那么 $F(m)$ 就会自然的满足这些等式.

那么我们就只需要解决两个问题: 如何具体构造这个范畴 $\mathcal{T}$, 以及定义 “保持乘积的函子” . 其中后者比较简单, 大致来说我们只需要 $F(A \times B)$ 与 $F(A) \times F(B)$ 同构即可.

定义 1.6. 给定两个范畴 $\mathcal{C}, \mathcal{D}$, 如果这两个范畴中任意两个对象都有乘积, 那么考虑任何一个函子 $F : \mathcal{C} \to \mathcal{D}$ 都有态射 $F(A \times B) \to F(A) \times F(B)$. 如果这些态射对于任何 $A, B \in \mathrm{Obj}(\mathcal{C})$ 都是同构, 那么就称 F **保持二元乘积结构**. 如果进一步 $F(1)$ 也是终对象, 那么就称 F **保持有限乘积结构**, 简称为**保积函子**.

接下来我们定义 $\mathcal{T}$. 这只需要定义 $\star^m$ 到 $\star$ 的态射. 根据上面的讨论, 我们考虑所有使用 m 个变量 $x_1, \ldots, x_m$, 用乘法、逆元、单位元运算得到的任意表达式. 如果有两个表达式 M, N 根据群满足的等式 (结合律等等) 可以证明相等, 就记作 $M \sim N$. 那么 $\sim$ 是一个等价关系, 我们将表达式的集合商去这个等价关系, 作为 $\hom(\star^m, \star)$ 的定义. 而一般的 $\hom(\star^m, \star^n)$ 可以定义为 n 个这样的表达式构成的有序对.

读者可能已经注意到了, 这样定义出来的正是 1.2 节中介绍过的群的语法! 这样看来, 群的语义可以看作是从群的语法构成的范畴出发的保积函子. 这就是 **Lawvere 函子语义**. 对于更复杂的理论, 我们也需要更加复杂的范畴结构, 因此考虑的函子也相应地需要保持这些结构.

在 1.2 节中, 我们还提到了另一种构造语法范畴的方式. 考虑 n 个生成元的自由群, 它们与其间的群同态构成一个范畴. 我们将所有的箭头反向, 就构成了语法范畴. 读者可以验证这样得到的是完全相同的范畴, 不过这种表述更加简练.

[5] 事实上我们在第 5.5 节中会介绍怎么既能得到好处, 又不使等式的表述过于复杂.

既然任何一个从群语法范畴出发的保积函子都可以看作是语义, 那么恒同函子 $\mathcal{T} \to \mathcal{T}$ 自然也是一套语义, 也就是说语法本身构成平凡的语义. 更进一步, 考虑所有的语义构成的范畴.[6] 在这个范畴中, 语法构成的平凡语义是始对象.

使用范畴表述的语法与语义, 在处理非常复杂的类型论 (如立方类型论) 时是降低复杂程度, 让证明简洁清晰的有力武器. 不过我们当然还需要证明这一套理论与传统的语法是等价的. 这类证明一般表现为两个定理, 分别称作**完备性**与**可靠性**. 我们在2.2节会再次提到.

1.5 类型论的历史

在类型论研究的历史中, 语法和语义的研究是对立统一的关系. 起初, 人们写下语法规则, 然后通过分析这些规则试图建立类型论的性质. 而选择这些规则的原因, 是人们脑中期望这些规则所拥有的语义. 在分析的过程中, 人们会发现为了证明类型论的一些性质, 需要考虑比一开始期望的语义更广义的其他语义. 而这些新的语义的发现又促使人们写下新的规则, 构造新的类型论. 这是一个事物走向自身的否定, 进而走向否定的否定的过程.

事实上, 类型论的发展像很多学科一样, 从某个主干开始不断产生不同的分支. 而每个时期中数学家对不同分支投入的精力不同. 我们会大致以每个分支产生的时间顺序讲述, 但是并非严格如此. 譬如较老的分支在后来的应用上产生了成果, 我仍然会在讲述这个分支的章节一并介绍.

如上文所述, 本文将会以对语法和语义的研究之间的矛盾运动为主线. 类型论有非常广泛的分支和应用, 本文不可能面面俱到. 如在计算机科学中用于对程序的正确性进行检查的类型论, 由于和数学中提到的类型论关系较远, 文章中就不做过多介绍. 我们从 Russell 的类型论出发, 以当代研究前沿的立方类型论为终点.

[6] 这个范畴中两个语义 $F : \mathcal{T} \to \mathcal{S}_1, G : \mathcal{T} \to \mathcal{S}_2$ 之间的态射是一个保积函子 $H : \mathcal{S}_1 \to \mathcal{S}_2$, 满足 $H \circ F = G$.

第二章　起源

2.1　Russell 的类型论

1899 年, Zermelo 发现了 Russell 悖论, 但是他并没有发表这个结论. 1901 年这个悖论由 Russell 独立发现. 它的论证非常简洁.

定理 2.1 (Russell)**.** *以下说法不可能成立: 任给一个含有变量 x 的命题 $P(x)$, 都有一个集合 $y = \{x \mid P(x)\}$, 使得 $z \in y$ 当且仅当 $P(z)$ 成立.*

证明. $P(x) = (x \notin x)$ 是含有变量 x 的命题. 而如果 $R = \{x \mid x \notin x\}$ 是集合, 那么 $z \in R$ 当且仅当 $z \notin z$. 特别地, $R \in R$ 当且仅当 $R \notin R$. 矛盾. □

因此, 我们习以为常的集合记号 $\{x \mid P(x)\}$ 是直接导致矛盾的. 这个悖论急迫要求数学家重新审视数学和逻辑的基础. 在这个基础上 Zermelo 等提出并逐渐完善的 ZFC 集合论, 大家或许已经有所了解. 但是 Russell 本人提出的解决方案则走了另一条思路. 这里我们介绍一个由 Alonzo Church 简化的版本. [22]

我们可以发现, 数学中我们根本不会需要讨论 $x \in x$ 这样的命题. 实数 x 属于实数集 $\mathbb{R}$, 而一些实数的集合 (如开集) 构成集合族 Ω. 我们偶尔还会讨论集合族的集合, 如所有超滤的集合 βX, 这在拓扑学的 Stone 表示定理中有提到. 注意这些数学对象之间都是有明确的层级的, 这一级的对象属于上一级的对象, 层级混乱的命题一般没有数学意义. 更严格来说, 对象之间是有分类的. 譬如虽然二维的点 $(\pi, 3)$ 和三维空间的点 $(0, 1, -1)$ 都是同一个层级的, 但是我们也基本不会把它们放在同一个集合中.

从这个观察出发, 我们可以把这种分类的概念严格化. 我们设当前研究的最基本的个体的分类称作 ι. 如我们研究数论的时候 ι 代表全体自然数, 研究分析学的时候就是全体实数, 等等. (对于更复杂的理论, 完全可以有多个基本个体的分类, 如平面几何学可能把点、直线、圆各分一类.) 这里我们不妨就考虑是自然数的情况. 其次我们需要有一类数学对象表示命题, 因此我们把所有命题构成的分类称作 o.

其次, 我们需要从基本的分类出发组成更复杂的分类. 譬如说可以用 $\iota \to \iota$ 表示自然数到自然数的函数. 为什么选择函数而不是集合呢? 注意要描述 A 的一个子集, 就等同于描述

各个元素是否属于这个子集. 换句话说, 我们给每个元素赋予一个命题. 因此 $\iota \to o$ 恰好就是自然数的子集的分类. 这样, 集合就可以看成是函数的特殊情况.

多元函数如何呢? 我们可以再引入一个代表集合 Descartes 乘积的分类. 但是由数学家 Moses Schönfinkel 提出了一种方便的技巧. 注意从 m 元集到 n 元集的函数恰好有 n^m 个. 而 $(n^m)^p = n^{mp}$. 而集合也有类似的等式: 值域是函数的函数 $A \to (B \to C)$ 和二元函数 $A \times B \to C$ 是一一对应的. 这其实在数学中出现过许多次. 譬如向量空间的张量积满足 $A \otimes B \to C$ 和 $A \to (B \to C)$ 是同构的. 带点拓扑空间上也有 $A \wedge B \to C$ 和 $A \to (B \to C)$ 同构, 这里 $\wedge$ 是压缩乘积 (smash product), 而 $\to$ 是连续函数空间. 因此, 我们不需要额外引入多元函数的概念. 这个技巧由 Haskell Curry 的使用而受到推广, 因此今天我们将其称作 **Curry 化**.

当然, 这只是一套能实现所有功能的方案. 我们也可以选择其他的组合. 譬如规定 $(s_1, s_2, \ldots, s_n), n \in \mathbb{N}$ 为 n 元关系的分类, 即含有 n 个变量的命题组成的分类. 其中第 i 个变量的分类是 s_i. 这样的话, 表示零元关系的 () 就是命题的分类, 而函数则在建立起逻辑之后定义为特殊的二元关系.

这些分类, 就是我们所称的类型. 这个类型论以及其变体被称为**简单类型论**. 当然, 只有类型还不行, 我们需要能谈论类型里的元素. 这里需要注意的是, 元素并不能独立于类型而存在. 我们不能先假设有一个元素, 然后讨论它属于什么类型. 这在 ZFC 集合论中是一样的, 我们不能假设有一个物体, 然后讨论它是不是集合, 因为一切可讨论的物体都是集合.[1] 在类型论中一切都首先归属于某个类型, 而后才能开始讨论其性质. 我们把 "E 的类型是 s" 写作 $E : s$.

在简单类型论中, 构造和使用元素的方法也非常简单. 首先, 对于函数类型, 我们可以用数学中常见的方法构造一个函数 —— 写出其解析式. 这种方法早在 Euler 的时代就已经熟知了:

$$x \mapsto \sin x^2 + 1.$$

这就规定了一个函数. 在逻辑学中, 我们更习惯用这样的符号:

$$\lambda x. \sin x^2 + 1.$$

不过这只是具体记号的细微差别. 更严格的说, 如果假设 x 是 s_1 类型的变量, 有一个含有 x 的表达式 E 是 s_2 类型的; 那么 $\lambda x.E$ 就是 $s_1 \to s_2$ 类型的表达式.

反过来, 我们有了一个函数, 要如何使用它呢? 唯一的方法当然是代入其自变量. 因此如果 $F : s_1 \to s_2$, 并且 $A : s_1$, 那么 $F(A) : s_2$. 这里因为我们经常会使用函数, 我们把 $a \to (b \to c)$ 简写为 $a \to b \to c$, 即默认是右结合的 (减法则是左结合的: $a-b-c = (a-b)-c$). 同时, 我们省略函数求值的括号, 并且默认左结合: $FAB = (F(A))(B)$.

[1] 实际上确实可能有这种语言, 例如有时会提到 "真类" 的概念. 但在 ZFC 中严格来说这就和我们现代使用 "设 ϵ 是任意小的正实数" 一样, 是严谨用语的一种缩写简化.

除了函数, 我们还有命题. 这时候我们就可以引入熟悉的逻辑结构了. 譬如说 “且” 命题就是 $\wedge : o \to o \to o$. 如上面所述这里表示一个二元函数. 而 “对于所有” 就是 $\forall : (s \to o) \to o$. 譬如说我有一族关于 n 的命题 $n^2 \neq 2$, 那么 $\forall(\lambda n.n^2 \neq 2)$ 就接受这一族命题作为参数, 而产生一个命题, 表示 “对于所有的 n 这些命题都成立” . 还有一个非常重要的东西是 $\epsilon : (\iota \to o) \to \iota$. 如果 $P : \iota \to o$ 是一族命题, 并且存在一个对象满足这个命题, 那么 $\epsilon(P)$ 表示某一个满足条件的对象. 有些版本中需要把这里的 “存在” 改成 “存在唯一” , 因为这里实际上默认了选择公理.

我们这里说的, 譬如 $\wedge$ 表示且命题, 是靠添加逻辑公理来保证的. 这和集合论是类似的: 集合论除了集合的公理之外, 也是默认了所有的逻辑公理的. 例如 “如果证明了 $p \Rightarrow q$, 并且证明了 p, 那么就可以证明 q” , 等等. 这些公理不涉及具体的数学对象.

至于那些涉及具体数学对象的公理, 就称为非逻辑公理. 譬如说在研究数论的时候, 我们就会规定一个元素 $0 : \iota$, 并且有一个函数 $S : \iota \to \iota$ 表示自然数的后继. 那么 $S(S(S(0)))$ 就表示 3. 我们还会添加数学归纳法作为公理:

$$\forall(\lambda P.P(0) \wedge \forall(\lambda n.P(n) \Rightarrow P(S(n))) \Rightarrow \forall(\lambda n.P(n))).$$

具体公理的细节比较乏味, 这里就省略了. 读者可以参考 [29]. 据笔者的阅读它关于命题逻辑的公理是有瑕疵的, 但是无伤大雅.

人们依据这种形式系统, 写出了计算机程序 Isabelle [49], 用计算机验证数学证明, 从而达到极高的准确性. 已经有不少数学理论被 Isabelle 形式化. 譬如质数定理 [27]. 这说明类型论与集合论一样, 有形式化数学的基本能力.

2.2 标准语义与 Henkin 语义

简单类型论常常被称作*高阶逻辑*. 这是相对于一阶逻辑说的. 一阶逻辑中一切的 $\forall, \exists$ 量词, 其作用的变量都是取值于同一个论域的. Peano 公理中这个论域就是自然数, ZFC 集合论中这个论域就是集合. 因此, Peano 公理对于归纳法做了特殊处理: 归纳法说的是 “对于所有含变量 n 的命题[2] $P(n)$, 如果 $P(0)$ 成立, 并且 $\forall n.P(n) \Rightarrow P(n+1)$, 那么就有 $\forall n.P(n)$.” 这里出现了 $\forall P$ 的表达. Peano 公理系统在一阶逻辑中无法做出这个表达, 因此实际上并不是添加了一条公理, 而是无数多条, 对于每一种可能的 $P(n)$ 都添加了一条独立的公理. 如果某个逻辑系统允许这样的量词, 就称之为二**阶逻辑**. 这里的命题含有谓词变量 P, 也就是关于谓词的谓词. 进一步, 二阶逻辑允许讨论关于谓词的谓词, 那么就会有关于 “关于谓词的谓词” 的谓词. 这就是三阶逻辑.

简单类型论中, 由于 $\forall : (s \to o) \to o$ 中 s 是任意的类型, 如果 $s = \iota$, 则这是一阶逻辑的量词; 如果 $s = (\iota \to o)$, 那么这就是二阶量词; 以此类推. 因此简单类型论包含了任意高阶的逻辑, 从而有非常强的表达力.

[2] 含有变量的命题称作**谓词**, 有 k 个变量就称为 k 元谓词.

与其它的高阶逻辑一样, 简单类型论有两种语义, 标准语义与 Henkin 语义. 其中前者是后者的特殊情况.

定义 2.1. 简单类型论的一个 **Henkin 模型** $M_\bullet$ 满足

- 对于每个类型 s, 选择一个集合 M_s;
- 对于函数类型 $s_1 \to s_2$, 选择的集合 $M_{s_1 \to s_2}$ 是函数集 $M_{s_1} \to M_{s_2}$ 的子集.

标准模型中要求 $M_{s_1 \to s_2} = (M_{s_1} \to M_{s_2})$. 这些集合的选取需要满足所有的公理.

因为语言是有限长的, 至多可以表达出可数个 $\mathbb{N} \to \mathbb{N}$ 函数, 绝大部分函数都是无法表达的. 因此 Henkin 模型更加灵活, 可以不包含不需要的函数, 在证明相关性质的时候更加有用. 每个命题在 Henkin 模型中都有一个真值. 对于了解逻辑学的读者, 这些事情应该不算新奇.

注意模型是在形式系统的外部进行的. 换句话说, 我们现在正在研究这个形式系统, 而不是使用这个形式系统; 因此我们研究时使用的形式系统可以任意选取, 比如选择 ZFC. 因此命题的真值也都是在这个外部的形式系统 (也叫**元逻辑**) 中得到的. 这个真值选取的正确性由一条定理保证:

定理 2.2 (Henkin 可靠性)**.** *如果某个命题在系统中可证, 那么它在所有 Henkin 模型中为真.*

一个重要的应用就是其逆否命题: 如果构造了某个 Henkin 模型, 其中我们关心的命题为假, 那么在系统中就不可证明这个命题. 注意 "不可证明" 与 "可证明其否定" 的区分. 类似的, 如果某个 Henkin 模型中我们关心的命题为真, 那么在系统中就不可证伪这个命题.

反过来的性质通常更难证明, 但是更加有用:

定理 2.3 (Henkin 完备性)**.** *某个命题在所有 Henkin 模型中都为真, 那么这个命题可证.*

我们在 1.4 节已经提到过这两个性质.

2.3 λ-演算与组合子逻辑

2.3.1 λ-演算

上面我们介绍了简单类型论. 而简单类型论中最核心的内容其实是关于如何形成和使用函数. 我们不妨把这部分内容抽象出来进一步研究其性质. 这一部分工作是由 Church 从他提出的类似的一套数学基础 [18] 中进一步提取出来的; 在 [19] 中他研究了 λ-演算的递归论性质.

剥去其他无关的细节后, 我们得到的系统尤其简洁. 它只有变量, 函数构造 $\lambda x.M$, 以及函数求值 $M(N)$. 它满足

$$(\lambda x.A)(B) = A[x/B].$$

其中 $A[x/B]$ 表示将 A 中包含的 x 变量全部代换成 B. 这个等式称作 **β-等价**. 正如我们在前言中所述, 这个描述看似简单, 但是变量的处理会引起一些麻烦, 在这里我们不做处理. 这个等式描述了构造一个函数, 而后马上对它求值, 会得到什么. 有时候我们也可以反过来考虑, 对一个函数求值, 然后以此构造一个函数, 会得到什么呢?

$$\lambda x.f(x) = f.$$

因为这实际上只是把原先 f 代表的函数包装了一下, 并没有改变它的实际性质, 因此我们认为这仍然是等于 f 的. 这个等式称作 **η-等价**. 当然, 这里也有变量处理的细节: f 的表达式中不能含有 x 本身.

仅仅有这些东西, 我们就能构造出一套非常丰富的结构了. 譬如说, 我们可以编码自然数: 将自然数 n 编码为

$$\lambda f.\lambda x.\underbrace{f(f(f(...}_{n个}(x))))$$

它的意义是 "把函数 f 迭代 n 次" , 用迭代次数来编码自然数. 那么我们可以立刻实现加法:

$$\mathsf{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx).$$

这是什么意思呢? 给定两个自然数 m, n 以及一个函数 f, 它先把 f 迭代了 n 次得到 nfx (因为 n 的定义就是把函数迭代 n 次的函数), 然后在此基础上又将 f 迭代了 m 次. 因此这就编码了两个自然数的加法.

乘法也是类似的, 我们有了一个函数 g 等于 f 的 n 次迭代后, 我们可以把 g 进行 m 次迭代, 就得到了 f 的 $m \cdot n$ 次迭代:

$$\mathsf{mul} = \lambda m.\lambda n.\lambda f.\lambda x.m(nf)x.$$

这里连着的 λ 已经有些多了, 我们把这些简写成 $\lambda mnfx.m(nf)x$, 即只用一个 λ 符号. 读者可以自行构造指数函数 m^n. Church 在论文 [19] 中, 进一步构造了判定自然数是否相等的函数, 从而能够在 λ-演算中编码 Diophantus 方程. 而在 [18] 中, 他与上面类似地添加了逻辑运算, 从而得到了逻辑的基础.

2.3.2 组合子逻辑

另一方面, 在 1920 年左右, 数学家 Moses Schönfinkel 提出了**组合子逻辑** [53]. 上面我们提到的简单类型论, 以及我们熟知的一阶逻辑等等, 其中一个最基本也是最容易被忽视的概念是变量. 在前言中, 我已经说明了处理变量的一部分困难. Schönfinkel 试图找到一种方式, 能够不使用变量而描述函数, 进而给出逻辑的更方便的定义. 原始论文中用的具体字母有些不同, 但是其余内容和我们下面介绍的是一致的.

既然我们关注的是如何不靠变量表达一切函数, 那么我们自然要先把具体的函数, 如 $\log, \sin$ 等等先放到一边, 关注函数运算本身的抽象结构. 我们语言中需要有函数求值, 把两

个表达式拼合成更大的表达式. 因为函数求值 $f(x)$ 本质上是个二元运算, 所以这里和乘法写得像一些: (fx), 强调其运算属性. 因为这里所有表达式都将不再有变量, 我们称它们为**组合子**. 注意这里的 x, y 等等都是代表一个完整的表达式. 它们并不是语言本身的变量, 而是为了描述语言而引入的记号. 我们称其为**元变量**.

首先我们有恒同函数: 它满足

$$(\mathsf{I}x) = x.$$

这很好理解. 那么譬如有 $(\mathsf{IK}) = \mathsf{K}$, 等等.

其次, 我们有常数函数: 对于任何表达式 x, 我们都希望有一个表达式, 它代表恒等于 x 的常函数. 换句话说, 如果这个表达式是 K_x 的话, 那么我们希望有等式 $(K_x y) = x$. 因此我们引入第一个原始组合子 K, 使得对于任何 x, $(\mathsf{K}x)$ 就是满足要求的常函数. 换句话说,

$$((\mathsf{K}x)y) = x.$$

这里括号已经开始有点多了. 我们和上面一样, 规定函数求值运算是左结合的. 上面的表达式也可以写成 $\mathsf{K}xy$, 表达相同的意思.

我们再引入函数复合: 给定两个函数 f, g, 我们需要能表述其复合 h, 满足 $hx = f(gx)$. 因此我们引入组合子 B, 使得 $\mathsf{B}fg$ 就是其复合. 换句话说,

$$\mathsf{B}fgx = f(gx).$$

再者, 对于一个二元函数 fxy, 我们希望能把它的第二个参数代入 gx. 换句话说, 我们希望找到函数 h 使得 $hx = fx(gx)$. 因此我们引入组合子 S, 使得

$$\mathsf{S}fgx = fx(gx).$$

这些东西有些无穷无尽了. 对于我们所希望的一切函数操作, 能不能只由有限个组合子互相组合表达呢? Schönfinkel 接下来指出, 只需要 S, K 两者就足够表达剩余的所有操作. 举个最简单的例子, 如何用它们表示 I 呢? 请看:

$$\begin{aligned} & \mathsf{SKK}x \\ (\mathsf{S}\text{ 的定义}) \ &= \mathsf{K}x(\mathsf{K}x) \\ (\mathsf{K}\text{ 的定义}) \ &= x. \end{aligned}$$

换句话说, SKK 这个函数, 在任何表达式 x 处求值的时候得到的都是 x 自身. 所以这个函数就是恒同函数! 关于这些组合子的更多性质, 读者可以参阅这本十分有趣的逻辑谜题书 [53].

这些组合子在 1927 年被 Haskell Curry 重新发现. 后来, 它们被计算机科学借鉴, 用来作为程序运行的一种简洁而完备的模型. 事实上, 当代计算机中有一类编程语言, 称为函数式语言, 它们在编译成程序的时候就经常会使用组合子作为中间步骤.

有了这些组合子, 我们就可以加入具体的函数来表达丰富的数学对象了. 对于逻辑来说, 我们可以加入组合子 $\forall$, 使得 $\forall A$ 表示 (在常规的有变量的语言中的) $\forall x.A(x)$. 作为例子, 现在我们可以重新表述我们在一阶逻辑中一条常见的公理:

$$(\forall x.A(x)) \implies A(c),$$

其中 c 是某个常数. 在组合子逻辑中, 这就可以写作

$$\mathsf{Im}(\forall A)(Ac)$$

其中 Im 组合子接受两个参数 p, q, 表示命题 $p \Rightarrow q$. 这样, 我们就完全避免了需要严格描述诸如 "将 x 代换为 c" 之类的麻烦. 这里再度提醒读者, A, c, p, q 在这里都是元变量, 只有 x 是真正的语言内变量.

2.3.3 Curry 悖论

然而, 把组合子或者 λ-演算直接用于逻辑基础是不可行的. 首先由 S. C. Kleene 与 J. B. Rosser 在 1935 年提出了组合子逻辑的悖论; 其次这个悖论被 Curry 简化. 为了理解这个悖论, 我们首先来看一个数学中经常会遇到的问题: 不动点问题.

假设我有一个任意的函数 f, 我希望求解方程 $x = f(x)$. 事实上, 递归函数也能这么定义. 假如我有递归定义 $f(x) = M$, 其中 M 是某个含有 f 的表达式, 那么我们就能写成 $f(x) = g(f, x)$. 换言之, 我们需要找到 $f = \lambda x.g(f, x)$ 的解; 再重写一下, 就是需要求 $G(f) = \lambda x.g(f, x)$ 的不动点. 由此可见, 能够求解不动点问题, 是语言表达力强的体现. 这里就体现了 λ-演算的强大能力.

定理 2.4. *对于任何函数 f, 存在表达式 M 满足 $M = f(M)$.*

证明. 取

$$M = (\lambda m.f(mm))(\lambda m.f(mm))$$

根据 β-相等的等式 $(\lambda x.A)B = A[x/B]$, 代入就得到

$$M = f(mm)[m/\lambda m.f(mm)] = f((\lambda m.f(mm))(\lambda m.f(mm))) = f(M).$$

恰好满足要求. □

这种递归的技巧的确非常天才, 读者可以再仔细研究这个表达式, 揣摩产生它的动机. 既然对于任何一个 f, 这个表达式都是其不动点, 那么我们自然可以提取出一个映射来:

$$\mathsf{fix}(f) = (\lambda m.f(mm))(\lambda m.f(mm)).$$

换句话说, $\mathsf{fix} = \lambda f.(\lambda m.f(mm))(\lambda m.f(mm))$ 就是**不动点组合子**.[3] 上一节中我们说过, 组合子演算可以表达所有的函数运算, 这个也不例外. 它对应的组合子为

$$\mathsf{Y} = \mathsf{S}(\mathsf{K}(\mathsf{SII}))(\mathsf{S}(\mathsf{S}(\mathsf{KS})\mathsf{K}))(\mathsf{K}(\mathsf{SII})).$$

这个复杂的表达式则是由将 λ-演算和组合子演算机械地互相转换的算法得到的.

读到这里, 谨慎的读者可能有疑问了: 万一某个表达式没有不动点怎么办? 如果有多个不动点, 算出来的会是哪一个呢? 我们来验证几个具体的例子.

首先对于常函数 $K_x(y) = x$, 我们计算

$$\mathsf{fix}(K_x) = K_x(\mathsf{fix}(K_x)) = x.$$

因此它计算出的的确是其唯一的不动点. 而对于恒同函数 $\lambda x.x$, 一切都是其不动点. 计算得到

$$\mathsf{fix}(\lambda x.x) = (\lambda x.xx)(\lambda x.xx).$$

我们记右边两个重复的表达式为 ω, 令 $\Omega = \omega\omega$. 那么能不能继续把这个表达式的具体值求出来呢? 读者如果自行试图继续化简的话, 就会发现无论怎么化简, 都会立刻兜圈回到原本的表达式. 换句话说, 这种表达式的求值操作是**不停机**的.

在逻辑中, 最简单的不存在不动点的函数是什么呢? 当然是命题的否定! 换句话说, 如果我们记 not 接受一个命题, 表示其否定, 那么 fix(not) 就等于其自己的否定, 从而导出了矛盾. 这就是 Curry 悖论.

2.4 简单类型 λ-演算的典范化

关于译名

对于已经了解这部分内容的读者, 有必要在这里澄清相关术语的翻译问题. 在重写系统下不可归约 (reduce) 的表达式称为既约形式 (normal form); 有强/弱停机性 (strong/weak normalization) 的概念. 而在类型论中, 一般而言有典范形式 (normal form), 无变量时称作闭典范形式 (canonical form). 注意英文中相同的单词有不同的含义, 我们翻译为不同的中文词汇.

我们上面已经看到, 不停机的 λ-演算或组合子很难成为逻辑的载体. 不过 Russell 的简单类型论目前来看还没有受到影响: 诸如 fix 的表达式在简单类型论中是不符合类型规则的. 那么对于添加了类型的演算, 能不能保证它不出现类似的问题呢? 在这个时期, 诞生了很多

[3] 虽然这并不是组合子演算的表达式, 但是广义来说, 我们把所有不含自由变量的 λ-表达式都称作组合子.

类型论以及对应研究它们的技术. 简单类型组合子或 λ-演算的相关性质可以用初等组合学的手段加以证明 [43], 但是我们这里选择更加展示其结构的一种思路 [56, §4.2].

首先, 我们想要的性质的精确定义是什么? Sterling 在 [56, §5.1] 中有非常精当的论述.

我们最常见到的概念是**停机性**. 使用有向图来描述 "化简" (术语是**归约**) 的方向, 我们可以定义不能进一步化简的表达式为**既约形式**. 如果对于每个表达式, 都存在一条路径达到既约形式, 则称这个有向图代表的归约系统是**弱停机**的; 如果所有的路径都必须在有限步内达到既约形式, 则称归约系统**强停机**. 假如从同一个表达式出发的任意两条归约路径, 都可以延长至共同的终点, 则称这个归约系统有**合流性** (confluence).

具体到类型论中, 我们可以规定一些归约的规则. 与上面抽象地使用有向图表示归约不同, 这里的规则可以对任何子表达式使用. 例如有规则 $1+1 \rightsquigarrow 2$, 那么就可以得到 $(1+1)+3 \rightsquigarrow 2+3$. 这样, 我们得到的就是一套**重写系统**. 我们可以给组合子演算写出一套重写系统:

$$\mathsf{I}x \rightsquigarrow x$$

$$\mathsf{K}xy \rightsquigarrow x$$

$$\mathsf{S}xyz \rightsquigarrow (xz)(yz)$$

那么设 $\omega = \mathsf{SII}$, 它对应 λ-演算中的 $\omega = \lambda x.xx$, 因为由归约规则得到

$$\omega x = \mathsf{SII}x \rightsquigarrow (\mathsf{I}x)(\mathsf{I}x) \rightsquigarrow xx.$$

请读者动笔归约表达式 $\Omega = \omega\omega$. 如果计算无误就会发现, Ω 无论如何归约, 几步之后就会回到自己. 因此 Ω 无法归约到既约形式. 换句话说, 无类型的组合子演算不是弱停机的, 从而也不可能是强停机的.

用重写系统处理简单类型组合子与 λ-演算, 最终可以证明它们是强停机的, 参见 [43]. 然而, 许多规则难以用重写系统表述. 譬如我们希望对于无序数对有 $\{x, y\} = \{y, x\}$. 使用重写系统处理这种等式显然有些困难. 当代研究的不少类型论无法使用重写系统描述. 所以我们有必要考虑别的方式. Sterling 在 [56] 中考虑了更加广义的概念, 在英文中仍然使用了 normalization 一词, 我们改译为**典范化**.

类型系统之间千差万别, 因此也极难对 "典范化" 一词做通用定义. 这里我们看几个例子.

2.4.1 简单类型组合子演算的典范化

在组合子演算中, 我们可以定义**典范形式**的概念. 典范化即为 "一切表达式都等价于某个典范形式". 我们定义不含 $\mathsf{S}ABC$ 或者 $\mathsf{K}AB$ 的表达式为典范的, 其中 A, B, C 为子表达式. 因此类似 $\mathsf{S}(\mathsf{K}(\mathsf{K}x))\mathsf{S}$ 的表达式是典范的.

我们类比简单类型 λ-演算的方案定义类型系统.

$$\mathsf{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

$$\mathsf{K} : \alpha \to \beta \to \alpha.$$

其中 α, β, γ 可以任意取值. 我们添加一个类型 $\mathbb{B}$ 作为基础类型, 有两个元素 $\mathsf{y}, \mathsf{n} : \mathbb{B}$. 注意这个类型中还有有其他元素, 如 Kyn 等. 但是在我们之前规定的运算的等价关系下, Kyn 等价于 y. 最后我们有规则

$$\frac{A : \alpha \to \beta \quad B : \alpha}{AB : \beta}$$

回忆分数线上方是前提, 下方是结论. 因为组合子演算中没有变量, 定义比 λ-演算方便许多. 直接对表达式长度归纳, 不难证明 $\mathbb{B}$ 类型中的典范形式只有 y 与 n.

由于无类型组合子演算没有典范化的性质, 我们必然需要在证明中利用类型. 因此我们可以对类型归纳：如果命题对 α, β 均成立可以推出对 $\alpha \to \beta$ 成立, 并且命题对 $\mathbb{B}$ 成立, 那么命题就对所有的类型均成立. 这种归纳法称为**结构归纳法**. 一般的归纳是沿着自然数的结构归纳, 而这里是沿着类型的结构归纳.

定理 2.5. 简单类型组合子演算中, 一切表达式都有一个等价的典范表达式.[4]

证明. 对于任何类型 α, 设 R_α 是这个类型中可典范化的表达式, T_α 为所有表达式. 我们需要证明的就是 $R_\alpha = T_\alpha$. 但是直接对 α 归纳是无法完成证明的, 我们需要加强归纳假设.

对于基础类型 $\mathbb{B}$, 我们设 $M_\mathbb{B} = R_\mathbb{B}$. 对于函数类型 $\alpha \to \beta$, 考虑

$$M_{\alpha\to\beta} = \{F : \alpha \to \beta \mid \forall A \in M_\alpha, FA \in M_\beta\}.$$

这样 $M_{\alpha\to\beta}$ 中的元素都对应一个 $M_\alpha \to M_\beta$ 函数. 可以发现这就是一个 Henkin 模型. 这样定义的 $M_\bullet$ 使得对类型的归纳可以顺利进行. 读者可以自行对类型归纳验证 M_α 均保持表达式上的等价关系. 对表达式长度归纳就可以证明 $M_\alpha = T_\alpha$. 最后只需要证明 $M_\alpha = R_\alpha$ 即可. 而这对类型归纳也很简单. □

可以注意到, 对表达式的长度归纳在这里实际上也是对表达式的结构归纳法：如果命题对 S, K 都成立, 并且如果对 A, B 成立则对 AB 成立, 那么命题对一切组合子表达式都成立.

2.4.2 简单类型 λ-演算的闭典范化

我们同样规定有一个基本类型 $\mathbb{B}$, 含有两个元素 y, n. 简单类型 λ-演算中可以含有自由变量, 为了严格说明这些自由变量的类型, 我们需要引入**语境** (context) 的概念. 在简单类型的情况下, 语境就是一些变量的列表, 附上每个变量的类型. 因此语境形如 $x{:}\alpha, y{:}\beta, \ldots, z{:}\gamma$. 我们用大写希腊字母表示语境. 如果某个表达式 M 在语境 Γ 中有类型 α, 那么我们写作 $\Gamma \vdash M : \alpha$.

[4] 之前所说的 “相等” 实际上是表达式之间的一个等价关系, 如 $\mathsf{K}xy$ 等价于 x, 我们介绍时为了方便写作 $\mathsf{K}xy = x$. 它们作为表达式显然是不相同的.

我们可以将类型规则用这个记号写出来：

$$\frac{(x{:}\alpha)\in\Gamma}{\Gamma\vdash x:\alpha}\qquad\frac{\Gamma,x{:}\alpha\vdash M:\beta}{\Gamma\vdash\lambda x^{\alpha}.M:\alpha\to\beta}\qquad\frac{\Gamma\vdash M:\alpha\to\beta\quad\Gamma\vdash N:\alpha}{\Gamma\vdash MN:\beta}.$$

注意这里严格来说 $\lambda x.M$ 实际上需要标注变量的类型, 但是一般情况下我们都省略了.

其次, 我们添加乘积类型. 有乘积类型的 λ-演算有闭典范化性质, 显然意味着不使用乘积类型的子集也有闭典范化性质. 而加入了乘积类型更能展现其本质结构.

$$\frac{\Gamma\vdash M:\alpha\quad\Gamma\vdash N:\beta}{\Gamma\vdash(M,N):\alpha\times\beta}\qquad\frac{\Gamma\vdash M:\alpha\times\beta}{\Gamma\vdash\pi_1(M):\alpha}\qquad\frac{\Gamma\vdash M:\alpha\times\beta}{\Gamma\vdash\pi_2(M):\beta}$$

乘积类型就是有序对的类型. 我们有 $\pi_1(M,N)$ 等价于 M, $\pi_2(M,N)$ 等价于 N. 反过来, 任何有序对 M 都等价于 $(\pi_1(M),\pi_2(M))$. 这两个等价关系与函数一样, 分别称作 β- 与 η-等价.

闭典范化说的是任何一个不含自由变量的表达式 $\vdash M:\mathbb{B}$ 都等价于 y,n 其中之一. 而更强一些的**典范化**说的是任何一个表达式 $\Gamma\vdash M:\alpha$ 都等价于某个典范表达式. 我们互相递归地定义典范表达式和中性表达式：

定义 2.2. 典范表达式要么是类型为 $\mathbb{B}$ 的中性表达式, 要么是 y,n 之一, 要么形如 $\lambda x.M$ 或 (M,N), 其中 M,N 是典范表达式. **中性表达式**要么是变量, 要么形如 $M(N)$ 或 $\pi_i(M)$, 其中 M 是中性表达式, N 是典范表达式.

为什么要如此定义呢? 这是从表达式的特征中总结的规律, 我们举一些例子来说明. 先不考虑乘积类型. $(\lambda x.x)y$ 很明显不能是典范的, 因为它可以化简为 y. 因此典范表达式不能形如 $(\lambda x.M)N$. 这样看来, λ 必须出现在外侧. 因此典范表达式最外层应该形如若干层 $\lambda x.\square$ 嵌套. 嵌套到最里层, 则可以出现函数求值操作 $M(N)$. 这里 M 不能含有 λ, 因此只能是进一步函数求值. 到最末层则是一个变量. 而 N 则可以是任何典范表达式. 换句话说, 只考虑函数类型时, 典范表达式应当形如

$$\lambda x_1.\lambda x_2.\lambda x_3.\cdots\lambda x_n.((x_iN_1)N_2)\cdots N_k,$$

其中 N_i 都是典范表达式. 把这个规律严格写出来就是定义 2.2 将关于 $\mathbb{B}$ 与乘积类型的部分去掉剩下的部分. 类似地对乘积类型进行分析, 就能导出完整的定义.

这里还有一个细节, 就是定义 2.2 要求上面的 $((x_iN_1)N_2)\cdots N_k$ 这个部分整体类型必须是 $\mathbb{B}$. 这是因为 η-等价. 如果变量 $f:\mathbb{B}\to\mathbb{B}$, 那么 f 等价于 $\lambda x.f(x)$. 我们认为后者才是真正的典范形式, 因为每个函数都是由 λ 构造的. 这在乘积类型中更加清晰：每个有序对显然都应该形如 (M,N), 因此如果有变量 $p:A\times B$, 它的典范形式也应当是 $(\pi_1(p),\pi_2(p))$.

无论是证明闭典范化还是典范化性质都需要不少功夫, 因为与组合子的情况类似, 简单地直接归纳是行不通的. 我们需要先通过归纳法证明一个更强的命题. 大致思路仍然是在每个类型 σ 上定义一个谓词 R_σ, 并归纳证明其性质. 因为集合 X 上的谓词 P 等价于它的子

集 $\{x \in X \mid P(x)\}$, 所以这两者可以交换使用. 如

$$R_{\sigma\to\tau} = \left\{ (\Gamma, f) \,\middle|\, \begin{array}{c} \Gamma \vdash f : \sigma \to \tau, \\ \forall (\Delta, s) \in R_\sigma, \Gamma \subseteq \Delta \implies (\Delta, f(s)) \in R_\tau \end{array} \right\}.$$

这比较不直观, 而且很难想到. 利用范畴语言, 可以给出更加系统的表述, 其中这些略微复杂的构造其实都是自然而唯一的.

为此, 我们首先需要将简单类型 λ-演算的语法表达为范畴语言. 我们令所有的语境 Γ 为范畴 $\mathcal{T}$ 中的对象, 而 Γ 到 Δ 的态射就是从 Γ 到 Δ 的**代换** (substitution). 例如从 $\Gamma = (x{:}A, y{:}A \to B)$ 到 $\Delta = (u{:}B, v{:}\mathbb{B})$ 的代换可以将 u 代换为 $y(x)$, 将 v 代换为 y. 我们把这个代换写作 $\sigma = [u/y(x), v/\mathsf{y}]$. 整体写作 $\Gamma \vdash \sigma : \Delta$.

如果 Γ, Δ 都只有一个类型, 那么 Γ 到 Δ 的代换就类似于这两个类型之间的函数. 使用语境与代换构成范畴, 与使用类型和函数构成范畴, 实际上是等价的. 但是前者在后续的理论中更加优雅, 并且能够推广. 比如某些类型论中可能没有函数类型, 等等.

我们认为 $\beta\eta$-等价的表达式给出相同的代换. 空的语境 () 在这个范畴中就是终对象, 因为一个代换 $\Gamma \vdash \sigma : ()$ 只需要给出零个表达式, 从而显然是唯一的. 因此我们也把空的语境按照终对象的记号写作 1.

我们实际上有非常好的方式刻画这个范畴. 我们已经介绍过了范畴中的 Descartes 乘积. 而函数也有范畴定义.

定义 2.3. 假设范畴中存在相关的乘积对象. 两个对象 A, B 的**函数对象** C, 或称为指数对象、幂对象, 配备了**求值态射** $\mathsf{ev} : A \times C \to B$, 使得对任何对象 C' 配备态射 $e : A \times C' \to B$, 都有唯一的态射 $u : C' \to C$ 使得以下图表交换.

$$\begin{array}{ccc} C' & \xrightarrow{\quad u \quad} & C \end{array}$$

$$\begin{array}{ccc} C' \times A & \xrightarrow{u\times\mathsf{id}} & C \times A \\ \downarrow{\scriptstyle e} & \swarrow{\scriptstyle \mathsf{ev}} & \\ B & & \end{array}$$

函数对象一般记作 $B^A, A \to B, [A, B]$ 等.

等价来说, 函数对象是满足 $\hom(-, B^A)$ 自然同构于 $\hom(- \times A, B)$ 的对象.

定义 2.4. 如果一个范畴包含所有的二元乘积、终对象与函数对象, 那么称这个范畴为**积闭范畴**. 如果一个函子 F 保持有限乘积结构, 同时

$$F(A) \times F(B^A) \xrightarrow{\cong} F(A \times B^A) \xrightarrow{F\mathsf{ev}} F(B)$$

这个复合态射由函数对象的性质诱导的态射 $F(B^A) \to F(B)^{F(A)}$ 是同构, 则称这个函子保持积闭结构, 即其为**保积闭函子**.

引理 2.1. *$\mathcal{T}$ 是由图表 $1 \rightrightarrows \mathbb{B}$ 生成的自由积闭范畴.*

“自由生成”在这里的含义或许需要阐明. 我们定义这样一种数学结构, 其中包含一个积闭范畴以及选定一个对象 B 和两个态射 $1 \rightrightarrows B$. 这些数学结构之间可以定义一些态射, 即保持选定的对象和态射的保积闭函子. 那么这就构成了一个范畴,[5] 姑且称作 $\mathsf{CartClosedCat}_{1 \rightrightarrows B}$. 那么其中的始对象就是我们所说的被自由生成的范畴.

证明. 把这个定理展开, 就可以发现它整齐地打包了很多预备的引理. 在这里我们只提到重要的引理, 其余部分都是直接展开定义即可.

- 首先我们需要证明 $\mathcal{T}$ 构成范畴, 需要验证结合律. 这就是**代换引理**. (很不幸的是这个词可以指代两个不同的引理.) 这可以通过对表达式的结构做归纳得到.

- 其次我们需要证明 $\mathcal{T}$ 是积闭范畴. 这实际上就是简单类型 λ-演算的语法决定的, 而其中那些自然同构的等式则是由 β, η 规则保证.

- 我们接下来需要证明, 对于任意一个这样的范畴 $\mathcal{C}$, 都能找到保持选定的对象和态射的保积闭函子 $F : \mathcal{T} \to \mathcal{C}$. 这之中需要证明的是代换与 $\beta\eta$-等价是可交换的. 而这函子的唯一性来源于其所有类型都是用 $\mathbb{B}, \times, \to$ 三者生成的. □

一个**范畴模型**就是从语法范畴出发的保持语法结构的函子. 在我们这里, 简单类型 λ-演算的模型就是从语法范畴 $\mathcal{T}$ 出发的保持 $\mathbb{B}, \mathsf{y}, \mathsf{n}$ 以及积闭结构的函子. 回顾上文, 就会发现我们刚刚已经证明的引理 2.1 可以表述成 “$\mathcal{T}$ 是简单类型 λ-演算的范畴模型构成的 2-范畴中的始对象”. 特别地, 语法范畴本身也是模型之一.

我们现在就可以迅速证明 $\mathsf{y} \neq \mathsf{n}$. 考虑所有集合的范畴 Set, 它是积闭的. 我们再选择 $1 \rightrightarrows B$ 为 $1 = \{0\}$ 到 $2 = \{0, 1\}$ 的两个态射即可. 如果 $\mathsf{y} = \mathsf{n}$, 那么说明引理 2.1 产生的函子 $\mathrm{s} : \mathcal{T} \to \mathsf{Set}$ 满足 $\mathrm{s}(\mathsf{y}) = \mathrm{s}(\mathsf{n})$, 进一步得到集合 2 的两个元素相等, 矛盾. 注意这个模型仍然无法证明闭典范性, 因为 s 仍然有可能把不相等的表达式 $M, N \in \hom_{\mathcal{T}}(1, \mathbb{B})$ 映射到相等的函数.

为了研究闭典范性, 我们需要先找出类型论中的闭元素, 即不包含自由变量的表达式. 这很容易, 我们只需要要求语境为空即可. 因此我们考虑 $\mathbf{\Gamma}(X) = \hom_{\mathcal{T}}(1, X)$. 这是一个 $\mathcal{T} \to \mathsf{Set}$ 的函子.

如上文所述, 我们一般的思路是为每一个类型赋予一个谓词. 在范畴论中, 我们可以使用一个 $f : Y \to X$ 的态射作为 X 上的 “谓词”. 大致来说, 我们认为如果 $f^{-1}(x)$ 非空, 那么这个谓词就取值为真, 否则为假. 当然这个大致说法只在集合范畴里成立, 其他范畴中只能起启发作用. 通常我们还需要对这些态射做一些要求, 但是这里不做任何要求更简洁. 因

[5] 实际上这是一个 2-范畴, 因为函子之间有自然变换, 也就是态射之间的态射. 我们考虑函子的时候一般不考虑相等, 而是自然同构.

此, 我们考虑这样一个范畴 $\mathcal{G}$, 它的对象是形如 $f : \Sigma' \to \mathbf{\Gamma}(\Sigma)$ 的态射,[6] 其中 Σ 是 $\mathcal{T}$ 中的对象, Σ' 是集合. 这个范畴里的态射是这样的交换图:

$$\begin{array}{ccc} \Sigma' & \xrightarrow{\sigma'} & \Delta' \\ \downarrow & & \downarrow \\ \mathbf{\Gamma}(\Sigma) & \xrightarrow{\mathbf{\Gamma}(\sigma)} & \mathbf{\Gamma}(\Delta) \end{array}$$

其中 σ 是 $\Sigma \vdash \Delta$ 的一个代换, σ' 是某个函数. 这样, $\mathcal{G}$ 就是每个类型的闭表达式集上所有可能的谓词的范畴. 如果读者熟悉范畴论, 就应该知道这是逗号范畴的特例 $\mathcal{G} = \mathrm{Id}_{\mathsf{Set}} \downarrow \mathbf{\Gamma}$. 它也可以看成是下面这个拉回:

$$\begin{array}{ccc} \mathcal{G} & \dashrightarrow & \mathsf{Set}^{\to} \\ {\scriptstyle \mathrm{gl}}\downarrow & \lrcorner & \downarrow{\scriptstyle \mathrm{cod}} \\ \mathcal{T} & \xrightarrow[\mathbf{\Gamma}]{} & \mathsf{Set} \end{array}$$

注意这时候我们自动有了一个函子 gl, 它把 $\mathcal{G}$ 中的对象 $\Sigma' \xrightarrow{f} \mathbf{\Gamma}(\Sigma)$ 映射到 Σ. 我们希望能够取 $\mathcal{G}$ 为一个模型. 回顾模型的定义, 我们的第一步当然是要证明:

引理 2.2. *$\mathcal{G}$ 是积闭范畴, 并且 gl 是保积闭函子.*

这个证明实际上对应的就是通常的证明中在类型上递归定义谓词. 但是注意这个命题本身已经完全限制了构造的可能性, 因此无需任何不自然的想法, 仅仅是展开定义即可. 这其中用到的一个重要性质是 $\mathbf{\Gamma}(1) \cong 1$ 与 $\mathbf{\Gamma}(\Sigma \times \Delta) \cong \mathbf{\Gamma}(\Sigma) \times \mathbf{\Gamma}(\Delta)$. 这可以抽出一条引理.

引理 2.3. *如果 $\mathcal{C}, \mathcal{D}$ 都有有限乘积, $F : \mathcal{C} \to \mathcal{D}$ 保持有限乘积, 那么 $\mathcal{G} = \mathrm{Id}_{\mathcal{D}} \downarrow F$ 也有有限乘积, 并且两个投影函子 $\mathcal{G} \to \mathcal{C}$, $\mathcal{G} \to \mathcal{D}$ 都保持有限乘积. 如果 $\mathcal{C}, \mathcal{D}$ 还有指数对象, 那么 $\mathcal{G}$ 也有指数对象, 并且 $\mathcal{G} \to \mathcal{C}$ 保持指数对象.*

注意这里不要求 F 保持指数对象, 并且 $\mathcal{G} \to \mathcal{D}$ 一般不保持指数对象.

证明. 参考 [39, 例子 A2.1.12]. 考虑对象 $(1 \to F(1)) \in \mathcal{G}$. 因为 $F(1) \cong 1$, 显然任何对象 $A' \to F(A)$ 到它都只有唯一的映射. 因此 $\mathcal{G}$ 有终对象, 并且两个投影函子保持终对象.

对于二元乘积, 如果有 $A' \to F(A), B' \to F(B)$, 那么这自然诱导映射 $A' \times B' \to F(A) \times F(B) \cong F(A \times B)$, 并且有两个投影

$$\begin{array}{ccccc} A' & \longleftarrow & A' \times B' & \longrightarrow & B' \\ \downarrow & & \downarrow & & \downarrow \\ F(A) & \longleftarrow & F(A \times B) & \longrightarrow & F(B) \end{array}$$

[6] 准确地说, 是一个三元组 (Σ', f, Σ).

我们验证这就是 $\mathcal{G}$ 中的二元乘积. 考虑交换图表

$$\begin{array}{ccccc} A' & \longleftarrow & C' & \longrightarrow & B' \\ \downarrow & & \downarrow & & \downarrow \\ F(A) & \longleftarrow & F(C) & \longrightarrow & F(B) \end{array}$$

则由上下两个乘积的泛性质分别得到唯一的映射 $C' \to A' \times B'$, $F(C) \to F(A \times B)$, 使得图表交换. 因此 $A' \times B' \to F(A \times B)$ 的确满足 $\mathcal{G}$ 中乘积的泛性质. 并且由构造得到两个投影函子也保持乘积.

对于函数对象来说, 考虑如图的拉回.

$$\begin{array}{ccccc} W & & \longrightarrow & & {Y'}^{X'} \\ \downarrow{\scriptstyle p} & \lrcorner & & & \downarrow \\ F(Y^X) & & \longrightarrow & F(Y)^{F(X)} & \longrightarrow F(Y)^{X'} \end{array}$$

其中左下的映射来自复合 $F(X) \times F(Y^X) \xrightarrow{\cong} F(X \times Y^X) \to F(Y)$. 不难验证 p 满足函数对象的泛性质. 由于一般而言 $W \ncong {Y'}^{X'}$, 因此投影映射 $\mathcal{G} \to \mathcal{D}$ 不保持指数对象; 但是 $\mathcal{G} \to \mathcal{C}$ 仍然是保持的. □

接下来, 我们只需要在 $\mathcal{G}$ 中选择一个对象 B 和两个态射 $1 \rightrightarrows B$. 选择了之后, 根据定理 2.1 我们立刻得知有唯一的保持它们的函子 $\mathrm{s} : \mathcal{T} \to \mathcal{G}$. 如果 gl 也保持它们的话, 那么这两个函子都在 $\mathsf{CartClosedCat}_{1 \rightrightarrows B}$ 中. 根据始对象的性质, $\mathrm{gl} \circ \mathrm{s} : \mathcal{T} \to \mathcal{T}$ 必然等于恒同函子 $\mathrm{Id}_{\mathcal{T}}$. 知道了这个, 我们自然要选择合适的对象. 这也不难, 如图构造即可:

$$\begin{array}{ccccc} 1 & \xrightarrow{\mathsf{y}} & \{\mathsf{y},\mathsf{n}\} & \xleftarrow{\mathsf{n}} & 1 \\ \downarrow & & \downarrow & & \downarrow \\ \mathbf{\Gamma}(1) & \xrightarrow[\mathbf{\Gamma}(\mathsf{y})]{} & \mathbf{\Gamma}(\mathbb{B}) & \xleftarrow[\mathbf{\Gamma}(\mathsf{n})]{} & \mathbf{\Gamma}(1) \end{array}$$

注意整个交换图都在 Set 中.

我们需要证明的是任何一个表达式 $M \in \mathbf{\Gamma}(\mathbb{B}) = \hom_{\mathcal{T}}(1, \mathbb{B})$ 都一定恰好是 y, n 其中之一. 因此我们考虑 $\mathrm{s}(M) \in \hom_{\mathcal{G}}(\mathrm{s}(1), \mathrm{s}(\mathbb{B}))$, 根据 $\mathcal{G}$ 中态射的定义, 这对应一个交换图:

$$\begin{array}{ccc} 1 & \overset{M'}{\dashrightarrow} & \{\mathsf{y},\mathsf{n}\} \\ \downarrow & & \downarrow \\ \mathbf{\Gamma}(1) & \xrightarrow[\mathbf{\Gamma}(M)]{} & \mathbf{\Gamma}(\mathbb{B}) \end{array}$$

由于 $\mathrm{gl} \circ \mathrm{s} = \mathrm{Id}$, 得到交换图的下沿必须是 $\mathbf{\Gamma}(1) \to \mathbf{\Gamma}(\mathbb{B})$. 而由于 s 保持终对象, 得到左上角是 1. 右侧则是因为 s 按照定义需要保持范畴中选定的 B 对象. 虚线表示的映射就选出了 y, n 的其中一个. 而由于正方形交换, 我们可以得出结论: M 必然是 y, n 其中之一.

2.4.3 简单类型 λ-演算的典范化

证明典范化显然要更加困难一些. 因为我们需要处理一切语境中的表达式, 因此 $\mathbf{\Gamma}(\Sigma) = \hom(1, \Sigma)$ 明显是不够的. 另一个自然的想法是改成使用 $よ(\Sigma) = \hom(-, \Sigma)$: 既然 1 不够, 就让它取遍所有的对象吧! 这样构造的新范畴 $\mathcal{G} = \mathrm{Id} \downarrow よ$ 的对象是形如 $\Sigma' \to よ(\Sigma)$ 的自然变换, 其中 Σ' 是某个函子 $\mathcal{T}^{\mathrm{op}} \to \mathsf{Set}$.

但是这样, 需要选定的对象 B 就难以构造了. 我们需要一个函子 $B' : \mathcal{T}^{\mathrm{op}} \to \mathsf{Set}$ 与自然变换 $B' \to よ(\mathbb{B})$, 我们仿照前面的做法希望 $B'(\Sigma)$ 应当是 Σ 下类型 $\mathbb{B}$ 中的典范表达式. 但是这样一来它的函子性就难办了. 例如若 $\Sigma = (f : \mathbb{B} \to \mathbb{B})$, 那么 $f(\mathsf{yes})$ 就属于典范形式. 但是考虑代换 $\sigma : 1 \to \Sigma$, 将 f 代入为 $\lambda x.x$, 那么 $(\lambda x.x)(\mathsf{yes})$ 就不再属于典范形式, 需要化简成 yes. 然而我们这里不能化简, 因为当前正是在证明这样的化简总是存在!

这里, 关键之处在于典范形式在代换下不稳定. 不过, 我们可以限制到一类特殊的代换, 使得它们稳定. 这种代换称作**更名代换**, 即只允许将变量换成别的变量, 不允许代换成更复杂的表达式. 语境与语境之间的更名代换构成 $\mathcal{T}$ 的宽[7]子范畴, 记作 $\mathcal{A}$, 记含入函子为 $\rho : \mathcal{A} \hookrightarrow \mathcal{T}$.

上面提到了 $\mathbf{\Gamma}$ 不足, 而 よ 有余. 因此我们可以取折中的 $\mathbf{N}(\Sigma) = \hom(\rho(-), \Sigma)$. 这是一个函子 $\mathcal{T} \to (\mathcal{A}^{\mathrm{op}} \to \mathsf{Set})$. 一般来说, 形如 $N(X) = \hom(F(-), X)$ 的函子在几何中称作**脉** (nerve) 函子. 定义范畴 $\mathcal{G} = \mathrm{Id} \downarrow \mathbf{N}$.

$\mathcal{G}$ 是积闭范畴仍然可以同理证明. 这是因为引理 2.3 中用到的关键性质仍然成立, 即 $\mathbf{N}$ 保持终对象和二元乘积. 最后, 我们需要选择 $1 \rightrightarrows B$. 依葫芦画瓢, 定义 $B'(\Sigma)$ 为 Σ 下类型 $\mathbb{B}$ 的典范形式集合 $\mathrm{nf}(\Sigma, \mathbb{B})$. 因为典范形式在更名代换下稳定, 不难证明 B' 构成函子 $\mathcal{A}^{\mathrm{op}} \to \mathsf{Set}$. 而典范形式 B' 到全体表达式的集合有显然的含入映射. 这就构造了 $B' \to \mathbf{N}(\mathbb{B})$. 因为 y, n 是一切语境下的典范元素, 也不难给出两个箭头 $1 \rightrightarrows B$. 这样, 由 $\mathcal{T}$ 的泛性质, 我们就得到了函子 $\mathrm{s} : \mathcal{T} \to \mathcal{G}$.

同理, 对于任何表达式 $M \in \hom(\Sigma, \alpha)$ 都可以应用函子 s, 就得到一个 $\mathcal{A}^{\mathrm{op}} \to \mathsf{Set}$ 中的交换图.

$$\begin{array}{ccc} \Sigma' & \xrightarrow{M'} & \alpha' \\ \downarrow & & \downarrow \\ \mathbf{N}(\Sigma) & \xrightarrow[\mathbf{N}(M)]{} & \mathbf{N}(\alpha) \end{array}$$

注意下方是由于 $\mathrm{gl} \circ \mathrm{s} = \mathrm{Id}$ 锚定的. 回忆 $\mathbf{N}(\Sigma) = \hom(\rho(-), \Sigma)$, 并且 $\rho : \mathcal{A} \to \mathcal{T}$ 是含入函子. 如果取它在 Σ 处的值 (因为括号太多, 写成下标), 就得到集合的映射 $\mathbf{N}_\Sigma(M) : \hom(\Sigma, \Sigma) \to \hom(\Sigma, \alpha)$, 具体来说是 $\sigma \mapsto M \circ \sigma$. 再代入 $\sigma = \mathsf{id}_\Sigma$ 就可以得到 M. 因此, 如果存在 $r \in \Sigma'_\Sigma$ 使得它被向下的映射对应到 $\mathsf{id}_\Sigma \in \mathbf{N}_\Sigma(\Sigma)$, 就可以得到 $M'(r) \in \alpha'_\Sigma$ 使得它对应到 $M \in \mathbf{N}_\Sigma(\alpha)$. 另一方面, 我们还需要利用归纳法确定上方的 Σ', α' 是什么.

[7] 即包含全体对象.

既然我们要证明典范性, 自然是希望能够归纳证明 α' 的元素对应 α 类型中的典范元素. 更精确的说, 我们希望给出一个映射 $\downarrow^{\alpha}_{\Sigma} : \alpha'_{\Sigma} \to \mathrm{nf}(\Sigma, \alpha)$ 使得纵向的映射 $\downarrow^{\alpha}_{\Sigma} : \alpha'_{\Sigma} \to \mathbf{N}_{\Sigma}(\alpha)$ 可以拆分成

$$\alpha'_{\Sigma} \xrightarrow{\downarrow^{\alpha}_{\Sigma}} \mathrm{nf}(\Sigma, \alpha) \hookrightarrow \hom(\Sigma, \alpha) = \mathbf{N}_{\Sigma}(\alpha).$$

对于 $\mathbb{B}$ 来说这已经成立了, 取恒同映射即可. 而对于 $\alpha \times \beta$ 来说, 根据引理 2.3 中的构造, 在上方的函子就是 $\alpha' \times \beta'$. 由归纳假设, α', β' 各自已经有了这种映射. 那么取

$$\downarrow^{\alpha\times\beta}_{\Sigma}(m, n) = (\downarrow^{\alpha}_{\Sigma}(m), \downarrow^{\beta}_{\Sigma}(n)).$$

而对于函数类型则困难一些. 我们可以填入部分答案

$$\downarrow^{\alpha\to\beta}_{\Sigma}(m) = \lambda x.\ \downarrow^{\beta}_{\Sigma,x:\alpha}(m(?)).$$

其中输入 $m \in (\alpha \to \beta)'_{\Sigma}$ 按照引理 2.3 的构造是 $\hom(よ_{\mathcal{A}}(\Sigma) \times \alpha', \beta')$ 的元素.[8] 我们需要依靠 m 得到某个 $\beta'_{\Sigma,x:\alpha}$ 的元素, 因此我们需要想办法输入某个 $\alpha'_{\Sigma,x:\alpha}$ 的元素.

事实上在归纳定义 $\downarrow$ 的同时, 我们需要同时归纳定义一个反向的映射 $\uparrow^{\alpha}_{\Sigma} : \mathrm{ne}(\Sigma, \alpha) \to \alpha'_{\Sigma}$, 从中性表达式映射到 α'_{Σ}. 同样这需要满足复合 $\mathrm{ne}(\Sigma, \alpha) \to \alpha'_{\Sigma} \to \mathbf{N}_{\Sigma}(\alpha) = \hom(\Sigma, \alpha)$ 恰好是中性表达式到全体表达式的含入映射. 对于 $\alpha = \mathbb{B}$ 来说 α'_{Σ} 就是典范表达式的集合, 因此 $\uparrow^{\mathbb{B}}_{\Sigma}$ 应当取从中性表达式映射到典范表达式的显然的映射. 对于乘积类型, 由归纳假设我们有 $\uparrow^{\alpha}_{\Sigma}, \uparrow^{\beta}_{\Sigma}$. 给定一个中性表达式 $\Sigma \vdash M : \alpha \times \beta$, 那么 $\pi_1(M), \pi_2(M)$ 各自也是中性表达式. 因此取

$$\uparrow^{\alpha\times\beta}_{\Sigma}(M) = (\uparrow^{\alpha}_{\Sigma}(\pi_1(M)), \uparrow^{\beta}_{\Sigma}(\pi_2(M)))$$

即可. 最后, 对于函数类型, 取

$$\downarrow^{\alpha\to\beta}_{\Sigma}(m) = \lambda x.\ \downarrow^{\beta}_{\Sigma,x:\alpha}(m(\uparrow^{\alpha}_{\Sigma,x:\alpha}(x))),$$

$$\uparrow^{\alpha\to\beta}_{\Sigma}(M) = (\sigma, n) \mapsto \uparrow^{\beta}_{\Sigma}(M(\downarrow^{\alpha}_{\Delta}(n)[\sigma])).$$

或许记号上有些令人眼花缭乱, 但是不难看出是唯一符合条件的答案. 注意上面用了 λ, 因为需要构造一个表达式. 而下面用了 $\mapsto$, 因为需要构造的是集合之间的映射: 输入任何 $\sigma : \Delta \to \Sigma$ 与 $n \in \alpha'_{\Delta}$, 我们需要输出 β'_{Σ} 的元素. 我们将 n 变为典范表达式后, 使用了更名代换, 仍然得到一个典范表达式 $\downarrow^{\alpha}_{\Delta}(n)[\sigma]$.

有了这两种映射, 我们现在可以完成证明的最后一步. 对于语境 $\Sigma = (x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n)$, 它同构于乘积 $\alpha_1 \times \cdots \times \alpha_n$. 考虑

$$r_{\Sigma} = (\uparrow^{\alpha_1}_{\Sigma}(x_1), \ldots, \uparrow^{\alpha_n}_{\Sigma}(x_n)) \in \Sigma'.$$

[8] 实际上是一个二元组 (m, M), 其中 M 是表达式 $\Sigma \vdash M : \alpha \to \beta$, 并且满足对于任何更名代换 $\sigma : \Delta \to \Sigma$ 与 $a \in \alpha'_{\Delta}$, $M(\downarrow^{\alpha}_{\Delta}(a)) = \downarrow^{\beta}_{\Delta}(m(\sigma, a))$. 我们直接用 m 表达 $\hom(よ_{\mathcal{A}}(\Sigma) \times \alpha', \beta')$ 的元素, 忽略 M. 同时 m 的第一个参数 σ 明确时也省略不写.

由于构造中保证了 $\nearrow^{\alpha}_{\Sigma}$ 复合上 $\downarrow^{\alpha}_{\Sigma} : \alpha'_{\Sigma} \to \mathbf{N}_{\Sigma}(\alpha)$ 后等于含入映射, 我们得到 r_{Σ} 在竖直映射下恰好对应到 $(x_1, \ldots, x_n) = \mathsf{id}_{\Sigma}$, 是恒同态射. 因此按照前面给出的交换图, 就得到 $M'(r_{\Sigma}) \in \alpha'_{\Sigma}$ 在竖直映射下对应 M. 而又因为这个竖直映射可以拆分成 $\wr^{\alpha}_{\Sigma}$ 复合上含入映射 $\mathrm{nf}(\Sigma, \alpha) \hookrightarrow \hom(\Sigma, \alpha)$, 这就得到了与 M 等价的正规形式 $\wr^{\alpha}_{\Sigma}(M'(r_{\Sigma}))$.

2.5 论域论

这一节主要参考 [1, 17]. 尽管无类型的 λ-演算与组合子演算不能直接用于逻辑, 但是仍然可以用来表示计算. 因此寻找它的一套语义表示也是有意义的. 一个非常重要的观察是, 无类型的本质是**单类型**. 换句话说, 实际上所有的表达式都有一个相同的类型. 在这里, 我们需要有一个函数类型 $\alpha \to \beta$, 但是一切的类型都相同, 所以我们就顺理成章地得到了一个等式:

$$(D \to D) \cong D.$$

因此, 无类型 λ-演算的语义大致上是要找到一个数学对象 D, 使得 $D \to D$, 也就是某种函数或者映射的空间, 满足这个同构. 其中我们的 λ 语法是从左往右的对应, 而函数求值则是从右往左的对应.

但是这种对象并不常见: 如果 D 是集合, 那么函数集 $D \to D$ 和 D 有双射, 当且仅当 D 只有一个元素. 因为如果 D 有至少两个元素, 那么 $D^D \geq 2^D > D$. 这里第二个不等式来源于 Cantor 的对角线论证. 而只有一个元素的集合过于平凡.

但是, 其他数学对象仍然是有机会的. 譬如假设 D 是一个拓扑空间, 那么 $D \to D$ 是连续函数构成的空间. 它至少在集合元素数量上是可以和 D 有双射的. 那么能不能构造一个拓扑空间使得 $D \to D$ 和 D 之间有同胚呢? 1960 年代, Dana Scott 为了解决这个问题, 提出了一系列数学对象, 主要来自于拓扑空间, 以及各种各样的序和格结构. 研究这些结构的理论被称为**论域论** (domain theory).

在此简单介绍一下这些数学对象的直观.

定义 2.5 ([1, §§2.1.5–2.1.6])**.** 给定一个集合 D 上的偏序 $\succeq$, 它的**定向子集** (directed subset) 为那些满足对于所有 $x, y \in A$, 存在 $z \in D$ 使得 $z \succeq x, y$ 的非空子集 A. 如果每个定向子集 A 的上确界都存在, 则称其为**定向完备偏序** (directed-complete partial order, 简写 dcpo). 两个 dcpo 之间的 **Scott-连续函数**为满足

$$f(\sup A) = \sup f(A)$$

的单调函数. 其中 A 取遍所有的定向子集. 注意单调函数满足定向子集的像仍然是定向子集.

可以给 dcpo 赋予一种拓扑, 称为 **Scott 拓扑**, 使得 Scott-连续函数就是在这些拓扑下连续的函数. Scott-连续函数的集合 $[A \to B]$ 逐点使用 B 上的偏序可以诱导出一个偏序. 在这个偏序下定向子集也都有上确界, 即逐点的上确界.

我们还可以在 dcpo 上添加各种各样的要求, 比如有最小元 $\bot$, 有连续性, 代数性等等; 就像环论中我们也会研究满足各种条件的环. 这些空间统称为**论域** (domain).[9] 而大致来说, 这些空间在求解类似 $[D \to D] \cong D$, 或者更一般的 $F(D) \cong D$ 方程时都表现了优良的性质. 我们可以找到最小的 D 满足这样的方程. 具体在 λ-演算上, 我们得到的偏序中每个点表示一组信息, 而在偏序中越大表示信息越详细. 而上面所说的 $\Omega = (\lambda x.xx)(\lambda x.xx)$ 则对应最小元 $\bot$. 我们甚至可以给出 Y 组合子的一种刻画.

引理 2.4 ([1, 定理 2.1.19]). *对于有最小元 $\bot$ 的 dcpo A, 存在 Scott-连续函数*

$$\mathsf{fix} : [A \to A] \to A$$

取出每个函数的最小不动点.

证明. 对于每个 Scott-连续函数, 考虑 $\bot, f(\bot), f(f(\bot)), \ldots$ 构成的序列. 由最小元的定义有 $\bot \preceq f(\bot)$. 再反复使用 f 的单调性得到 $f^{(n)}(\bot) \preceq f^{(n+1)}(\bot)$. 因此这是一条升链, 从而是定向子集. 它有上确界 $f^{(\infty)}(\bot) = \sup_{n\in\mathbb{N}} f^{(n)}(\bot)$. 那么由 f 的 Scott 连续性有

$$f(f^{(\infty)}(\bot)) = f(\sup_{n\in\mathbb{N}} f^{(n)}(\bot)) = \sup_{n\in\mathbb{N}} f(f^{(n)}(\bot)) = f^{(\infty)}(\bot).$$

故这的确是不动点, 并且不难看出这是最小的不动点. 另一方面, 考虑 n 次迭代操作 $f \mapsto f^{(n)}(\bot)$, 这是 Scott-连续函数, 从而这些迭代函数对 n 取逐点上确界也是 Scott-连续的, 所以 fix 是连续函数. □

我们构造出满足 $[D \to D] \cong D$ 的对象后, 可以证明 Y 的语义就是 fix 函数. 回忆我们认为 dcpo 中元素越大表示信息越详细, 而 fix 是最小不动点. 换句话说, 尽管某个函数可能有很多不动点, 但是 Y 总是取出定义最 “不明确” 的那个. 譬如 $\lambda x.x$ 的不动点为全体表达式. 而 $\mathsf{Y}(\lambda x.x) = \Omega$, 选出了信息 “最少” 的那个, 即 $\bot$.

论域论不仅提供了一种框架, 还将计算机科学、数学与逻辑学联系起来. 特别是计算机科学中经常需要处理不停机的递归, 以及集合论难以处理的函数. 最经典的例子就是下一章马上会介绍的 F 系统, 而不少编程语言正是以 F 系统为蓝本构建的.

[9] 和哲学或者逻辑学中的论域不同.

第三章　Curry–Howard 对应

3.1　简单类型 λ-演算与命题逻辑

经过上面的一些研究, 或许有些读者会产生一种模糊的感觉. 假如我们有 $f : \alpha \to \beta$, 以及 $a : \alpha$, 那么我们就能求值得到 $f(a) : \beta$. 这其实和逻辑里最古老的推理有些相像: 假如我们证明了 $p \Rightarrow q$, 并且我们证明了 p, 那么我们就可以证明 q.

这是否能推广一下呢? 完全可以! $p \wedge q$ 就对应了 Descartes 乘积 $\alpha \times \beta$. 这个类型的元素形如 (a, b), 其中 $a : \alpha, b : \beta$. 并且有两个投影函数 $\pi_1(a, b) = a, \pi_2(a, b) = b$. 它们的类型写出来就是 $\alpha \times \beta \to \alpha$ 以及 $\alpha \times \beta \to \beta$. 对应到逻辑里面, π_1 恰好就是 “如果 $p \wedge q$ 成立, 那么 p 成立” .

对偶地, $p \vee q$ 则对应了类型的不交并, 记作 $\alpha + \beta$. 它的元素要么形如 $\iota_1(a)$, 要么形如 $\iota_2(b)$. 而我们有一个函数对此分类讨论: 如果有 $f : \alpha \to \gamma$ 与 $g : \beta \to \gamma$ 分别处理两种情况, 那么

$$\mathsf{case}(f, g, c) = \begin{cases} f(a) & c = \iota_1(a) \\ g(b) & c = \iota_2(b). \end{cases}$$

而 case 的类型就是 $(\alpha \to \gamma) \to (\beta \to \gamma) \to (\alpha + \beta \to \gamma)$. (回忆我们之前说的用 $\alpha \to \beta \to \gamma$ 表示一个二元函数的技巧.)

我们总结一下上面的相似性, 可以发现可以把类型和命题做类比. 类型的元素就和命题的证明差不多, 因此命题的真假就可以解释为对应的类型是否有元素. 我们可以定义一个类型 $\mathbf{0}$ 没有任何元素, 代表假命题. 则 $\alpha \to \mathbf{0}$ 代表命题的否定, 缩写成 $\neg\alpha$. 我们对 $\mathbf{0}$ 的元素可以分类讨论, 但是由于只有零种情况, 因此得到

$$\mathsf{exfalso} : \mathbf{0} \to \gamma.$$

可以把这个与前面的 case 对比. exfalso 对应逻辑中的爆炸原理: 从假命题出发可以推出一切命题.

另一方面, (简单类型) 组合子演算也有类似的现象. 这更加明显: S 的类型是 $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$; 而 K 则是 $\alpha \to \beta \to \alpha$. 这刚好是 Hilbert 公理系统里面的两条公理! Curry 在 1934 年初步注意到组合子的这个性质 [25]. 这个想法在后来几十年中逐步

被推广. 人们发现, 用于计算、为数学对象分类的类型, 与用于证明、只有真假的命题, 在许多方面上有着惊人的相似性:

- 组合子对应 Hilbert 命题逻辑演绎系统;
- λ-演算对应自然演绎系统;[1]
- λ-演算可以转换成组合子, 对应演绎定理.[2]

我们或许可以大胆一些, 把这种相似性升格为同一性:

类型是命题.

这就是 Curry–Howard 对应. 它体现了逻辑学和类型论 (进而与计算机科学) 有着深刻的内在联系. 需要注意的是, 之前介绍的简单类型论可以表达高阶逻辑, 但是这个表达能力是额外加入的: 我们添加了命题的类型 o, 并且加入了逻辑连词与公理. 但是在这里简单类型 λ-演算并没有加入任何关于逻辑的内容, 直接通过 Curry–Howard 对应即可表达命题逻辑. 我们下面看看能从这之中挖掘出哪些有价值的数学.

3.2 依值类型

有了命题逻辑, 下一步自然是一阶逻辑, 或者称作谓词逻辑. 这里有形如 $\forall x.P(x)$ 的命题. 直观上看, 它对应 "输入 x, 输出 $P(x)$ 的元素" 的函数构成的类型. 但是这有一点小问题: 对于不同的输入值, 输出的类型是不同的. 换句话说, 类型可以取决于值. 我们称陪域取决于输入值的函数为**依值函数**.

事实上, 这种函数我们在数学中早就见过了. 向量场就是对流形上的每个点 $x \in M$, 给出一个向量 $\vec{v} \in \mathrm{T}_x M$ 的函数. 这里随着输入不同, 输出的向量所处的向量空间也不同. 对于普通的集合也有这样的构造: 给定定义域 B 与一族集合 $F_x, x \in B$, 累乘 $\prod_{x\in B} F_x$ 就可以看作一种依值函数.

对偶地, 命题 $\exists x.P(x)$ 对应的类型就是有序对 (x, p), 其中第二个分量 $p : P(x)$ 类型取决于前一个分量. 它类似于向量丛的全空间 $\mathrm{T}M$. 对于普通的集合, 这对应一族集合 F_x 的不交并 $\coprod_{x\in B} F_x$.

当然, 在通常数学中对依值函数的定义一般采取另一种方法. 考虑全空间 $E = \coprod_{x\in B} F_x$, 我们有一个映射 $\pi : E \to B$ 投影到 B 上. 依值函数集合 $\prod_{x:B} F_x$ 定义为

$$\{f : B \to E \mid \pi \circ f = \mathsf{id}\}.$$

[1] 这是与 Hilbert 系统不同的另一个演绎系统.

[2] 演绎定理说的是, 如果假设 φ 成立, 在 Hilbert 系统中可以证明 ψ, 那么不需要任何假设就能在 Hilbert 系统中证明 $\varphi \Rightarrow \psi$.

这个定义在集合上和上面是等价的, 而在拓扑空间上这就类似纤维丛的截面. 这样定义的优点是只使用了普通的映射, 因此容易推广到拓扑空间、流形等事物.

我们接下来会采用 Π, Σ 来指代这两种类型. 它们的规则和函数、Descartes 乘积非常类似. 如果有变量 $x : A$ 与表达式 $M : B$, 其中 M, B 可以包含 x (注意在普通的函数类型定义中, B 不得包含 x). 那么 $\lambda x.M$ 的类型是 $\prod_{x:A} B$. 反过来, 如果有表达式 $N : A$, $F : \prod_{x:A} B$, 那么 $F(N)$ 的类型就是 $B[x/N]$, 其中 $[x/N]$ 表示把变量 x 代入为 N. 用 $\Gamma \vdash A\,\mathrm{type}$ 表示 "在语境 Γ 中, 表达式 A 是合法的类型" , 那么可以将规则写出来:

$$\frac{\Gamma \vdash A\,\mathrm{type} \quad \Gamma, x{:}A \vdash B\,\mathrm{type}}{\Gamma \vdash \prod_{x:A} B\,\mathrm{type}}$$

注意 B 在语境 $\Gamma, x{:}A$ 中, 因此可以包含变量 x, 以及一切在 Γ 中的变量.

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : \prod_{x:A} B} \qquad \frac{\Gamma \vdash F : \prod_{x:A} B \quad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B[x/N]}$$

这里的 $\lambda x.M$ 实际上也应当标上类型, 即 $\lambda x^A.M$ 或者 $\lambda x{:}A.M$, 但是一般类型明确时都省略. 我们仍然有 β-等价 $(\lambda x.M)(N) = M[x/N]$ 与 η-等价 $\lambda x.M(x) = M$, 其中后者的 M 不含变量 x.

对于 Σ 类型也可以类似操作.

$$\frac{\Gamma \vdash A\,\mathrm{type} \quad \Gamma, x{:}A \vdash B\,\mathrm{type}}{\Gamma \vdash \sum_{x:A} B\,\mathrm{type}}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[x/M]}{\Gamma \vdash (M, N) : \sum_{x:A} B} \qquad \frac{\Gamma \vdash P : \sum_{x:A} B}{\Gamma \vdash \pi_1(P) : A} \qquad \frac{\Gamma \vdash P : \sum_{x:A} B}{\Gamma \vdash \pi_2(P) : B[x/\pi_1(P)]}.$$

对于更详细的介绍, 读者可以参阅 [62] 的第一章.

1967 年, 利用这些想法, 数学家 De Bruijn 设计了一门依值类型论的计算机语言, 称为 Automath. 1976 年, Jutting 在他的学位论文 [47] 中使用 Automath 形式化了 Landau 的《分析学基础》. Automath 没有被大规模使用, 不过它的设计影响到了许多后继的类型论.

F 系统[3] (这严格来说不属于依值类型) 在 1972 年由逻辑学家 Jean-Yves Girard [32] 与 1974 年由计算机科学家 J. C. Reynolds [51] 独立发现. 一边是逻辑学, 一边是计算机科学, 这说明 Curry–Howard 对应不仅仅是一个巧合, 而是切实深刻地影响了两个学科的发展.

F 系统的规则对简单类型 λ-演算做了一点扩展. 它的类型除了函数类型 $\alpha \to \beta$ 之外, 还有形如 $\forall X.\beta$ 的类型, 其中 X 是类型变量, 表达式 β 可以包含 X. 这个类型表示 "不论 X 是什么, 都能充当 β 的元素" 的类型. 具体来说, 如果有一个可以包含 X 的表达式 M, 对于任何类型 α, $M[X/\alpha]$ 都是 $\beta[X/\alpha]$ 类型的元素, 那么 $\Lambda X.M$ 就是 $\forall X.\beta$ 的元素. 而反过来, 如果 f 是 $\Lambda X.M$ 的元素, 那么 $f\alpha$ 就是 $M[X/\alpha]$ 的元素.

举个例子, $\Lambda X.\lambda x.x$ 就是 $\forall X.X \to X$ 的元素.[4] F 系统包含很多奇特的构造. 比如我们可以像无类型 λ-演算中那样编码自然数. 定义类型 $\mathbb{N} = \forall X.(X \to X) \to (X \to X)$, 那么之

[3] 这个名字是因为 Girard 的论文中用 F 缩写法语单词 *fermé* (闭), 阴差阳错成为了现在普遍指代这个系统的名字.

[4] 在一些形式中要求在 λx 上标注出 x 的类型, 比如 $\Lambda X.\lambda x^X.x$. 而在一些形式中甚至不要求写出 ΛX. 这些差别会对语言的性质起到一定的影响, 但是我们这里不过多停留.

前构造的自然数 $\lambda f.\lambda x.f(f(\dots(f(x))))$ 就可以在 F 系统里面写成

$$\Lambda X.\lambda f^{X\to X}.\lambda x^{X}.f(f(\dots(f(x)))).$$

可以同样写出上面介绍过的加法、乘法、幂等等运算. 请读者思考为什么在简单类型 λ-演算中无法做到这一点.

Girard 在他的论文 [32] 中证明了二阶逻辑[5]中可以证明是全函数的递归函数[6]在 F 系统中都能定义. Reynolds 则从计算机科学的视角给出了一个反向的对应, F 系统中能定义的 $\mathbb{N}\to\mathbb{N}$ 的函数都能在二阶逻辑中找到对应的逻辑关系. 这被称为 **Girard–Reynolds 同构**.

F 系统非常强大, 它可以构造很多类型. 譬如 $\mathbf{1}=\forall X.X\to X$, 可以证明它的唯一元素就是 $\Lambda X.\lambda x.x$, 因此可以看作是单元素类型的定义. 而 $\mathbf{2}=\forall X.X\to X\to X$ 则恰好有两个元素 (请读者思考是哪两个元素; 具体的证明则比较复杂, 需要语义工具). 上面我们构造了自然数类型, 这是递归类型的特殊情况. F 系统里面可以直接构造出大量的递归类型 [63]. 它甚至可以构造一个类型 V, 使得 V 和 $(V\to\mathbf{2})\to\mathbf{2}$ 同构. 这在集合的世界中是不可想象的: 一个集合永远比自己的幂集要小, 更不用说自己的幂集的幂集了! Reynolds [50] 在 1984 年证明了简单类型 λ-演算的集合论模型不能扩展到 F 系统. 因此, 2.5 节中讨论的技巧在这里构造语义就非常重要.

1988 年, Coquand 与 Huet 提出了构造演算 (Calculus of Constructions, 缩写为 CoC). 这与上面提到的一些类型系统很快被总结到同一个框架下, 统称为**纯类型系统** [5]. 需要向初学者说明的是, 这一族类型论 (包括其特殊情况, 即 λ-立方) 的定义非常整齐划一, 因此往往吸引人试图将别的类型论都改装到这个框架下. 然而, 这么做往往是违背其数学本质的. 同时, 这个框架也有不少问题, 见 [38] 中的讨论. 本文并不是类型论教程, 因此念其历史意义仍然加以介绍, 但在当代的教科书中则应完全不提到纯类型系统与 λ-立方.

正常的依值函数类型的规则是

$$\frac{x{:}A\vdash B(x)\,\mathrm{type}}{\prod_{x:A}B(x)\,\mathrm{type}}$$

换句话说, 假设含有变量 $x:A$ 的表达式 $B(x)$ 是合法的类型, 那么 $\prod_{x:A}B(x)$ 也是合法的类型. 当然, 如果 B 不含变量 x, 这就退化到普通的函数类型 $A\to B$. 而 F 系统中的 $\forall$ 类型只需要做一点修改:

$$\frac{X{:}\blacksquare\vdash B(X)\,\mathrm{type}}{\forall X.B(X)\,\mathrm{type}}$$

但是这里 ■ 中应该填入什么呢? 注意这里 X 是一个类型, 如 $\forall X.X\to X$. 因此我们需要一个符号表示类型的类型. 即 $X:\mathsf{type}$. 这样, 我们也不需要 $A\,\mathrm{type}$ 这样的声明, 而只需要 $A:\mathsf{type}$ 就可以了. 那么 F 系统中的规则就是

$$\frac{X{:}\mathsf{type}\vdash B(X):\mathsf{type}}{\forall X.B(X):\mathsf{type}}.$$

[5]这里其实移除了排中律, 下一节就会提到相关内容.

[6]有一些递归定义并不能产生处处有定义的函数, 比如 $f(0)=0, f(n)=f(n-2)$, 这在奇数处就没有定义. 我们把处处有定义的函数称为**全函数**.

我们甚至可以与上面依值函数类型的记号类比, 把 $\forall X.B(X)$ 写成 $\prod_{X:\mathsf{type}} B(X)$.

当然, 这立刻就会产生一个问题: type 自己的类型是什么呢? 在 F 系统里面, 由于 $\forall X$ 中的 X 本身不能代入 type, 因此我们可以直接规定 type 没有类型. 当然, 我们也可以引入一个符号 kind, 满足 $\mathsf{type} : \mathsf{kind}$. 这样因为上面的规则中要求 $X : \mathsf{type}$, 所以 X 本身不能取值为 type. 这样我们的改写仍然是等价的.

纯类型系统就是对这个观察的一个推广. 一般来说, 我们有

$$\frac{A : s_1 \quad x{:}A \vdash B(x) : s_2}{\prod_{x:A} B(x) : s_3}.$$

上面的依值函数就是 $(s_1, s_2, s_3) = (\mathsf{type}, \mathsf{type}, \mathsf{type})$ 的情况, 而 F 系统中的 $\forall$ 则可以写成 $(s_1, s_2, s_3) = (\mathsf{kind}, \mathsf{type}, \mathsf{type})$ 的情况. 补上其他的规则, 我们可以总结出一个一般的定义.

定义 3.1. 纯类型系统是一族类型系统, 由三个参量决定: 一个集合 $\mathcal{S}$, 其上有二元关系 $\mathcal{A}$ 与三元关系 $\mathcal{R}$. 类型系统的规则有

$$\frac{}{\Gamma \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A}$$

对于每组 $(s_1, s_2, s_3) \in \mathcal{R}$, 类型系统中都添加两条规则:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B(x) : s_2}{\Gamma \vdash \prod_{x:A} B(x) : s_3}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B(x) : s_2 \quad \Gamma, x{:}A \vdash F(x) : B(x)}{\Gamma \vdash \lambda x.F(x) : \prod_{x:A} B(x)}.$$

如果条目 $x : A$ 在列表 Γ 中出现, 则有

$$\frac{}{\Gamma \vdash x : A}.$$

最后有

$$\frac{\Gamma \vdash F : \prod_{x:A} B(x) \quad \Gamma \vdash M : A}{\Gamma \vdash FM : B(A)}$$

这里严格来说应该写成 $B[x/A]$.

在上面的例子中 $\mathcal{S} = \{\mathsf{type}, \mathsf{kind}\}$, $\mathcal{A} = \{(\mathsf{type}, \mathsf{kind})\}$.

这里有一个非常重要的问题需要注意: 既然类型中可以包含普通的值, 譬如 $P(n)$ 表示 n 维的向量的类型, 那么如果我有一个向量 $v : P(1+1)$, 是否可以说 $v : P(2)$ 呢? 我们当然需要允许这样的推断, 否则依值类型的意义就不大了. 因此我们需要修改上面类型系统的定义, 加入一条规则:

$$\frac{\Gamma \vdash M : A \quad A = B}{\Gamma \vdash M : B}$$

这里的等号与上面说的一样, 实际上是在表达式上定义的一个等价关系, 如 $(\lambda x.x)y$ 与 y 等价. 在定义更加复杂的类型论时, 我们经常会发现这个等价关系附带上类型信息会更加方便.

换句话说, 我们对于每对 Γ, T, 在所有满足 $\Gamma \vdash A : T$ 的表达式 A 构成的集合 $U_{\Gamma,T}$ 上各自定义一个等价关系, 而不是在全集 $\bigcup_{\Gamma,T} U_{\Gamma,T}$ 上定义. 技术上的细节可以参考 [5].

如果我们设 $\mathsf{type} : \mathsf{type}$ 会如何呢? 换句话说, 我们试图让 $(\mathsf{type}, \mathsf{type}) \in \mathcal{A}$. 结果或许并不令人意外: Russell 悖论重新回到了类型论中! 因此我们不能设定存在所有类型的类型, type 自身的类型必须是别的东西. 然而, 即使 $\mathcal{A}$ 中没有这样的循环, 我们仍然可以导出矛盾. 1972 年, Girard 构造出了 U 系统. 这个系统中 $\mathcal{S}$ 有三个元素, $\mathsf{type}, \mathsf{kind}, \mathsf{Kind}$. $\mathcal{A} = \{(\mathsf{type}, \mathsf{kind}), (\mathsf{kind}, \mathsf{Kind})\}$. 也就是说这里只有 $\mathsf{type} : \mathsf{kind} : \mathsf{Kind}$, 没有循环. 然而, 这个系统中仍然能够构造出类似的悖论, 我们称为 Girard 悖论. 因此, 设计纯类型系统时必须小心.

构造演算中, $\mathcal{S} = \{\mathsf{type}, \mathsf{kind}\}$, 并且 $\mathsf{type} : \mathsf{kind}$. 而

$$\mathcal{R} = \{(s_1, s_2, s_2) \mid s_1, s_2 \in \mathcal{S}\}.$$

因此构造演算包含了普通的依值函数类型与 F 系统中的 $\forall$ 类型, 还包含了两种新的依值函数类型. 这个系统已经证明是没有类似的悖论的. 事实上它有很好的典范性. 它包含了 F 系统, 所以自然地携带了 F 系统中强大的编码能力, 如可以直接使用 $\forall$ 类型定义自然数类型, 等等. 因为继续扩展这个系统很容易导致 Girard 悖论出现, 构造演算以及其各种子系统成为了纯类型系统中应用最广泛的系统.

3.3 经典逻辑

前面我们提到简单类型 λ-演算与命题逻辑之间有对应关系, 称作 Curry–Howard 对应. 实际上, 这个对应并不完美, 唯一的差异就在于排中律: 对于任何命题 p, $p \vee \neg p$ 都是真命题. 注意这和矛盾律的区别: 对于任何命题 p, $p \wedge \neg p$ 都是假命题. 当然, 我们仍然可以证明 $\neg\neg(p \vee \neg p)$, 但是我们不能消去双重否定.[7]

为什么简单类型 λ-演算无法推出排中律呢? 我们可以看一个模型. 给定一个拓扑空间 X, 其上的开集代表命题. 全集是真命题, 空集则是假命题. 并集与交集分别表示 $p \vee q, p \wedge q$. p 的否定应当是与 p 矛盾的命题中最弱的, 因此在拓扑上则是与开集 U 不相交的开集中最大的, 换句话说是 $X \setminus U$ 的内部. 类似地, 如果 U, V 分别表示命题 p, q, 那么 $p \to q$ 则是 $V \cup (X \setminus U)$ 的内部. 对于所有的逻辑规则 $\frac{p_1\ p_2 \cdots}{q}$, 可以验证它们对应的开集总是满足 $(U_1 \cap U_2 \cap \cdots) \subseteq V$. 因此如果从逻辑规则中可以推导出排中律, 那么应当有 $U \cup \operatorname{int}(X \setminus U)$ 等于全集. 然而很明显一个开集并上其补集的内部不一定是全集. 从这个模型看来, 排中律不成立有一些几何的内涵. 我们之后会进一步阐述.

我们还有另一种方法. 在简单类型 λ-演算中, 可以证明不含自由变量的类型为 $\alpha + \beta$ 的元素, 必然等价于形如 $\iota_i(M)$ 的表达式. 这也就是说, 在简单类型 λ-演算中, 没有其他条件的情况下证明命题 $p \vee q$, 就必须要确定地构造出具体哪一边是成立的. 而排中律 $p \vee \neg p$ 自

[7] 注意这和 $\neg\neg(\forall p. p \vee \neg p)$ 不同, 我们无法证明这个命题.

然不能直接确定哪一边成立, 因此是无法证明的. 上面这两种方法, 前者是通过观察语义, 而后者是通过直接分析语法.

我们在 3.3.2 节会讨论如何在类型论的 Curry–Howard 对应中加入排中律. 但是我们首先不妨来看看缺少排中律会引发什么变化. 我们将含有排中律的逻辑称作**经典逻辑**.[8]

定理 3.1. *排中律有如下的等价形式.*

- *排中律*: $p \vee \neg p$.
- *双重否定消去*: $\neg\neg p \to p$.
- *逆否命题*: $\neg p \to \neg q$ *等价于* $q \to p$.
- *Peirce 定律*: $((p \to q) \to p) \to p$.
- *Peirce 定律的特殊情况*: $(\neg p \to p) \to p$.

定理 3.2 (Diaconescu)**.** *选择公理可以推出排中律. 事实上排中律可以表述为 "$\{0,1\}$ 满足选择公理".*[9]

一个最经典的使用排中律的证明是这样的.

定理 3.3. *存在无理数 a, b 使得 a^b 是有理数.*

证明. 考虑 $\sqrt{2}^{\sqrt{2}}$. 它要么是有理数要么是无理数. 如果是前者, 则命题成立. 如果是后者, 设 $a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}$, 则

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = 2.$$

因此无论 $\sqrt{2}^{\sqrt{2}}$ 是有理数还是无理数, 都存在 a^b 满足条件. □

注. 一个不使用排中律的方法是考虑 $a = \sqrt{2}, b = \log_2(9)$. 另一个避免排中律的方法是花费功夫证明 $\sqrt{2}^{\sqrt{2}}$ 确实是无理数, 这需要非常复杂的数论证明.

那么正常的数学研究中多大程度上需要排中律呢? 实际上直到 Hilbert 之前的绝大部分数学研究都完全可以绕过排中律. Hilbert 之后的数学也可以发展无需排中律的版本. 现在已经有不少不使用排中律发展数学的努力, 包括代数、几何、分析等等. 在没有排中律时, 会出现很多有趣的现象: 很多数学概念会分裂成多个不等价的概念 (实际上会分裂出无数个不同的概念, 但是正如正常数学中我们一般只会取有意义的概念研究, 在这里我们也只会谈比

[8] 经典逻辑的不少定义中, 排中律是作为其它规则的推论存在的, 正如 Zorn 引理一般是作为选择公理的推论存在的. 因此在这些定义下无法谈论直接 "去掉" 排中律. 因为我们不会涉及定义的细节, 这里不讨论具体如何从形式系统中去掉排中律.

[9] 集合 Y 满足选择公理, 当且仅当对于任何集合 X 与关系 $R \subseteq X \times Y$, 如果对于任何 X 的元素 x 都存在 Y 的元素 y 满足 xRy, 那么存在一个函数 $f : X \to Y$ 使得 $xRf(x)$. 完整的选择公理说的就是所有集合都满足选择公理.

较重要的版本). 譬如域在没有排中律的时候会出现离散域与连续域的变体, 有理数 $\mathbb{Q}$ 是离散域, 而用 Cauchy 列构造的实数 $\mathbb{R}$ 则是连续域. 注意这里完全没有涉及任何拓扑信息! 离散域的定义中, 任何非零元素都可逆. 而连续域 (Heyting field) 中, 如果某个元素不可逆则它为零, 并且如果 $a+b$ 可逆则 a, b 至少有一个可逆. 注意离散域都是平凡的连续域. 在有排中律时, 离散域与连续域都等价于通常的域的定义, 这样这个非常有意思的区分就被抹平了.

另一方面, 事实上去掉排中律完全没有使得能够发展的数学受到限制, 恰好相反, 去掉排中律使得数学变得更丰富了. 这是因为对于任何一个命题 p, 都存在一个命题 p^{N}, 使得这个命题在无排中律下 p^{N} 可证当且仅当 p 在使用排中律时可证, 并且在有排中律时 p^{N} 与 p 等价. 例如 $\forall x.p(x) \vee q(x)$ 会被翻译为 $\neg\neg\forall x.\neg(\neg p(x) \wedge \neg q(x))$. 因此无排中律时的数学严格包含了有排中律时的数学. 换个说法, 排中律使得大量原先不等价的命题变得等价, 所以不加入排中律会让数学更加丰富, 同时不会有任何损失. Hilbert 说过: “将数学家的排中律夺走, 就好比禁止拳击手使用拳头.” 在这个意义下, 这句话自然是不正确的.

仅仅是去掉排中律, 得到的是**中性数学**. 因为排中律仅仅是不可证明或证伪, 中性数学中随时可以重新加入排中律. 反过来说, 这留出了额外的空间, 让我们可以加入与排中律相矛盾的公理. 即加入一些公理使得 $\neg(\forall p.p \vee \neg p)$ 成立. 例如我们可以加入 “所有函数 $\mathbb{R} \to \mathbb{R}$ 都是连续函数”. 这样的公理被称为**反经典**的.

3.3.1 构造主义

前一节中已经提示了, 在没有排中律时, 任何数学对象都必须构造出来才可以认为存在. $p \vee q$ 的证明必须明确给出哪一边成立, 而 $\exists x.p(x)$ 必须具体给出一个 x.[10] 因此, 我们将这种逻辑以及相关的各类数学哲学思想称为**构造主义**. 如果读者对放弃排中律仍然有怀疑, 可以阅读《接受构造主义的五个阶段》[6]. 这篇简短的文章借用了心理学上人们接受悲痛与失去的五个阶段 —— 否认、愤怒、恳求、沮丧、接受 —— 来探讨对构造主义的一些常见误解, 以及构造主义对数学的贡献. 下面的介绍主要参考了 [15, 12].

直觉主义

在 19 世纪, 数学家逐渐开始将许多过去定义模糊的概念严格化. 我们发现可以将实数定义为有理数的 Cauchy 列. Cantor 的工作为无穷数列、无穷集合等等概念夯实了基础, 并且区分了仅仅是稠密的集合 (如有理数) 与真正的实数连续统. Frege 将自然数也使用集合做了定义 —— 自然数 n 就是所有恰好有 n 个元素的集合构成的集合; 自然数集 $\mathbb{N}$ 就是所有有限集在双射下构成的等价类的集合. Kronecker 与 Poincaré 等数学家在 19 世纪末期对此提出了质疑. 他们认为这些使用无穷集合, 并且在许多情况下都是非构造性质的概念是无法作为数学的根基的.

[10] 不过, 需要注意的是它们不需要显式写出这样的构造. 正如一般数学中一样, 我们只需要保证理论上可以给出构造即可, 不需要将构造的每一个细节都写出来才能算作证明.

在世纪之交, Russell 悖论的发现动摇了这套基础. (我们最终因为这个原因抛弃了 Frege 对自然数的定义, 但是 Cauchy 列, Cantor 的连续统等等概念仍然保留了下来.) Hilbert 为了应对这次危机, 提出了一套纲领. Hilbert 认为我们可以信任有限的算数与组合, 希望能给出一套描述无限集合的公理系统, 并且用纯有限的方法证明这套公理系统的自洽性. 这样就可以将无限的概念从数学危机中拯救出来. Hilbert 认为, 我们可以将关于无限的数学化归为对命题机械地按照形式系统的规则操作的过程.

我们现在知道, 以 Gödel 的不完备性定理为代表的对形式系统更进一步的研究粉碎了 Hilbert 原本的愿望, 不过在当时这还并未明晰. Brouwer 继承了 Kronecker 的思想, 对 Hilbert 纲领发起了抨击. 关于 Brouwer 与 Hilbert 的论争的历史, 读者可以参阅 [15]. Brouwer 认为, 数学是人类的精神活动. 数学思想先于语言而存在, 仅仅对命题进行机械操作是不可能完全囊括数学的全部内涵的. 这些数学哲学观点就是**直觉主义**. 遵循这些思想, Brouwer 构建了一套直觉主义数学基础.

直觉主义作为构造主义的一个分支, 拒绝排中律无限制的使用. Brouwer 认为, 只有关于有限个事物的命题, 才能使用排中律 —— 因为对于有限个事物我们可以逐一检查命题是否成立. 我们不能随意地推广到无限的集合. 譬如考虑命题 p: “对于所有偶数 $n \geq 4$, n 可以表示为两个质数的和”. 并没有理由认为 $p \vee \neg p$ 成立, 因为不可能在有限的时间内穷尽所有的偶数. 我们将来可能会给出一个无法被表示为两个质数之和的偶数; 也有可能得到一个算法, 保证任意给定一个偶数都能找出对应的两个质数. 但是在那一刻到来之前, 我们不能从对有限集合的经验出发, 直接认为这两者之一一定成立.

不过, 直觉主义数学并不严格要求一切数学对象都需要被完全确定地构造出来.

> 允许两种创造数学对象的方法: 第一种是无穷序列, 每一个元素都可以从已经得到的数学对象中自由选择 [...]; 第二种是集合, 之前构造出的数学对象中, 满足某个性质的所有对象构成一个集合, 前提是如果一个对象满足该性质, 那么与这个对象按照定义 “相等” 的其他对象也满足该性质. [13]

在其他风格的构造主义中, 对于一个无限数列通常要求存在一个算法能够给定 n, 计算出这个数列的第 n 项. 但是 Brouwer 的直觉主义中允许所谓的*自由选择序列*, 即不需要确定的算法, 可以自由选择. 利用这些自由选择序列, Brouwer 构造出了直觉主义的实数集.

拒绝排中律使得实数集表现得更像延绵连续的整体. 在经典数学中, 实数集是 “脆” 的, 因为可以随时定义不连续的函数:

$$f(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0). \end{cases}$$

经典数学中必须在实数集的基础上定义拓扑, 才能将连续性这样直观的概念恢复出来. 由于没有排中律, 直觉主义的实数并不能证明 $(x < 0) \vee (x = 0) \vee (x > 0)$ 一定成立, 因此无法随意地分割实数. Brouwer 采用了更加适合构造主义的**分离性** (apartness), 即定义一个关系

$x \# y$ 为 “存在 $\epsilon > 0$ 使得 $|x - y| > \epsilon$” . 在经典数学中, 这与 $x \neq y$ 等价, 但是在构造主义数学中这给出了具体的 ϵ 的构造, 因此比 $x \neq y$ 更强. 譬如可以证明任何实数 $x \# 0$ 都有倒数, 但是将条件换为 $x \neq 0$ 就无法证明.

由直觉主义的数学哲学出发, Brouwer 进一步提出了一些公理. 从这些公理出发, 直觉主义脱离了仅仅是移除排中律的中性数学的范畴, 而是与经典逻辑中的结论相矛盾, 对实数连续统建立起了迥然不同的理论体系. 例如著名的一致连续性定理:

定理 3.4 (Brouwer)**.** *给定度量空间 X, Y, 其中 X 完备并且完全有界. 则任何映射 $X \to Y$ 都是一致连续的.*

推论 3.1. *任何函数 $f : [0, 1] \to [0, 1]$ 都是一致连续函数.*

但是需要注意的是, 此时可以认为 Brouwer 的实数已经与经典数学中的实数不是同一个概念了, 因此不能简单的说直觉主义数学与经典数学是 “互斥” 或者 “矛盾” 的. 它们仅仅是在命题的形式上有表面的互斥关系, 而实际情况需要对各自的数学哲学进行更细致的解读.

1930 年, Brouwer 的学生 Arend Heyting 提出了一套形式逻辑系统, 试图刻画直觉主义中使用的逻辑. 这套形式系统大致就是一阶逻辑去掉排中律. 因此, 许多文献中也用**直觉主义逻辑**一词来指代去掉排中律的逻辑. 我们前文中已经提到, 这类逻辑更准确的名字是中性逻辑.

俄罗斯构造主义

在 1940 年代末, 俄罗斯数学家 Andrey A. Markov (Андре́й Андре́евич Ма́рков, 与他的父亲同名; 他的父亲是提出 Markov 链的人) 构建了构造主义逻辑的另一个分支 —— 俄罗斯构造主义, 也称作递归构造主义. 在这个流派中, 一切数学对象都使用适当的递归函数构造.

递归函数的概念在 1933 年由 Gödel 与 Jacques Herbrand 在研究形式系统的性质时提出, 是一类特殊的自然数上的多元函数 $\mathbb{N}^n \to \mathbb{N}$; 在 1936 年 Church 的 (无类型) λ-演算中, 使用 2.3.1 节中提到的编码自然数的方式, 也可以考虑能够在 λ-演算中写出来的自然数上的多元函数; 同年, Alan Turing 试图刻画 “存在有效的方法计算” 的概念时, 提出了 Turing 机. 令人惊讶的是, 这三者给出的是等价的定义. 我们现在将这些定义统称为**可计算函数**或者一**般递归函数**. 如输入自然数 n, 输出第 n 个质数的函数, 就是一个可计算函数.[11]

一般递归函数对某些输入可能会无限地计算下去, 如定义

$$f(n) = \begin{cases} f\left(\left\lceil \frac{n}{2} \right\rceil\right) & (n \geq 1) \\ 0 & (n = 0). \end{cases}$$

[11] 不可计算的函数反而在一般数学中比较少见, 其中一个例子是将所有的命题用自然数做 (合理的) 编号, 对于编号 n 的命题, 如果它在 ZFC 公理系统下可以证明, 那么定义函数 $f(n) = 1$; 否则定义 $f(n) = 0$. 这种命题的编号称作 **Gödel 编码**.

那么计算 $f(1)$ 时就会无限递归下去. 此时我们认为 $f(1)$ 是未定义的. 如果某个函数对一些输入是未定义的, 我们则称其为**部分函数**. 除法就是我们最熟悉的部分函数, 它对除以零的情况没有定义. 对所有输入都有定义的函数称为**全函数**. 在讨论可计算函数时, 除非额外强调, "函数" 一词通常指代部分函数.

递归函数就是那些能通过明确的方法计算的函数, 符合构造主义的大致方向. 需要注意的是, 算法都是有限长的, 因此能够写出的算法只有可数个, 所以 $\mathbb{N} \to \mathbb{N}$ 的可计算函数也只有可数个. 在俄罗斯构造主义中定义的实数, 对应着经典数学中的**可计算实数**. 尽管可计算实数只有可数个, 但是我们遇到的绝大部分实数都是可计算的, 如 $\pi, e, \tan\sqrt{114514}$, 等等. 这使得构造主义中很多看似离奇的命题有了合理的解释: 将实数改为为可计算实数, 函数改为为可计算函数, 以此类推, 这样得到的命题在经典数学中也合理了起来.

Bishop 构造主义

然而, 上面提到的构造主义流派在一段时间内都没有产出其他数学家看来有重大意义的成果. 同时, 有很多类似推论 3.1 的, 看起来完全是稀奇古怪或者自相矛盾的结论. 在当时看来, Hilbert 所评价的 "将数学家的排中律夺走, 就好比禁止拳击手使用拳头" 确实显得有些道理. 在 1967 年, Errett Bishop 出版了一本构造主义分析学的书.

> Bishop 的工作的意义在于证明了原本 Hilbert 与 Brouwer 都同意的一个重要观点是错误的. 两人都认为, 如果真的要发展构造主义数学, 那么就必须 "放弃" 许多现代数学中关键的部分 (例如测度论或者复分析). Bishop 展示了这完全是错误的. [...] 只不过我们需要选择使用一种优雅的方式进行研究, 而不是用 Hilbert 所谓的 "拳击手的拳头". [7]

Bishop 认为, 要构造一个集合, 只需要说明如何构造它的元素, 并且说明构造出的两个元素如何判断相等. 这种定义集合的方式现在被称为 **Bishop 集合**. 在现代类型论[12]的语境下, 这就是一个类型上配备了一个等价关系, 称作**广集** (setoid). 我们可以紧接着将 "广集函数" 定义为尊重这个等价关系的函数. 换句话说, 给定广集 $(A, \sim_A), (B, \sim_B)$, 要构造广集 $A \Rightarrow B$ 的元素, 只需要给出一个对应法则 $f : A \to B$, 使得 $x \sim_A y$ 时 $f(x) \sim_B f(y)$. 给定两个这样的元素 f, g, 我们要说明如何判断它们相等, 也即定义等价关系 $f \sim_{A\Rightarrow B} g$. 为此, 我们规定

$$(f \sim_{A\Rightarrow B} g) \iff \forall x, f(x) \sim_B g(x).$$

读者可以验证这确实是等价关系.

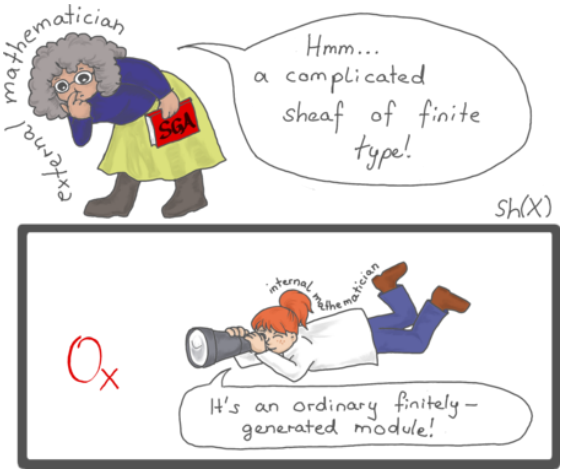
Bishop 用行动展示了数学的许多重要的部分都能在构造主义中发展, 并且只需要对经典数学的理论做较小的修改. 正如前面的引文所说, 我们只需要选取一种优雅的姿态, 就能在构造主义中轻松地完成经典数学中可以做到的事情. 行胜于言, Bishop 极大地推动了构造主义数学的发展.

[12] Bishop 并没有使用类型论, 这是后人的工作.

Martin-Löf 构造主义

1970 年代, Martin-Löf 提出了一种构造主义的类型论, 我们现在称作**构造主义类型论**或者 **Martin-Löf 类型论** (Martin-Löf type theory, 缩写为 MLTT). 我们会留出一章专门讲述, 这里按下不表. 从 Martin-Löf 的理论开始, 构造主义与类型论变得密不可分. 不过需要提醒读者, 类型论在 Russell 起初提出时, 完全是经典逻辑的. 所以说尽管类型论与构造主义有很强的关联, 但是并没有包含关系.

构造主义的几何意义



在当代的几何研究中, 构造主义又被赋予了新的内涵. 一些复杂的几何对象与简单的代数结构在操作上有相似之处, 例如有限型的模层 (sheaf of modules of finite type) 是非常复杂的对象, 而它对应的就是有限生成模.

我们可以使用这种相似的对应关系大大简化一些证明. 但是仔细考察这个对应关系, 会发现排中律没有对应. 几何学家可以选择在经典逻辑中写下繁琐且难以阅读的证明, 或者选择在一种构造主义的语言中写出简洁的证明. 这种语言是**内语言**, 我们会在 5.5 节介绍. 注意这并不是说放弃了排中律. 即使我们认为排中律成立, 这也仅仅是在外部成立, 而在外部进行证明时, 我们就要处理譬如 “有限型的模层” 等概念. 在内语言中, 我们只需要处理有限生成模, 而代价是不能使用排中律. 在内语言中写下的证明, 都可以翻译成外部语言中对复杂几何对象的操作. 有代数几何基础的读者可以参阅 [10]. [13]

3.3.2 经典逻辑的 Curry–Howard 对应

除了研究不含排中律的逻辑之外, 另一个方向自然的问题就是经典逻辑是否有 Curry–Howard 对应. 答案是肯定的. 首先, 我们可以直接将排中律强行加入作为公理, 也就是直接

[13] 插图取自此处, 这里也含有 [10] 的一份副本, 以及许多相关的讲座幻灯片.

加入一个没有定义的常量 lem. 但是这样的缺陷是只有排中律一条规则是公理, 其余的逻辑规则都是自然从类型论中导出的. 同时其余的逻辑规则都有对应类型论中的判值相等. 而由于 lem 未定义, 它没有对应的判值相等规则. 这使得类型论的性质受到一定的破坏.

那么有没有将排中律自然地加入到类型论中的方法呢? 将排中律 (或者其等价形式) 直接对应到类型, 我们得到的就是计算机科学中**计算续体**的概念.

在计算机语言中, 有时候需要将控制权的流向倒转, 方便组织程序. 计算续体用数学语言描述, 就是有一个 "洞" 的表达式, 如 $2 \times (3 + \sin \square)$. 它表达了后继计算的概念. 即将 $\square$ 算出来之后, 后继的计算是取正弦, 加 3 并乘以 2. 在一些计算机语言中, 提供了一个用于获取计算续体的功能, 称作 call/cc. 考虑表达式

$$2 \times [3 + \sin(\mathsf{call/cc}(f))],$$

其中 f 是某个函数. 此时计算机会将 call/cc 的计算续体, 也就是 $u = 2 \times (3 + \sin \square)$ 提取出来. 紧接着, 计算机会求值 $f(u)$. 如果 f 的程序中使用了 u, 即给 u 这个计算续体提供了一个数值 x, 那么程序就会开始计算 $u[x]$, 也就是 $2 \times (3 + \sin x)$. 我们设整个程序最终的类型是 β, 计算续体的洞的类型是 α, 那么计算续体本身的类型就应该是 $\alpha \to \beta$, 因为给它提供一个 $x : \alpha$, 它就会继续计算得出一个 β 类型的值. f 需要接收一个计算续体, 计算出 α 类型的值, 因此 $f : (\alpha \to \beta) \to \alpha$. 最后, call/cc 接收 f, 输出一个 α 类型的值, 所以

$$\mathsf{call/cc} : ((\alpha \to \beta) \to \alpha) \to \alpha.$$

细心的读者可以发现, 这就是前文讲述的 Peirce 定律! 由于 Peirce 定律与排中律等价, 我们就可以得到排中律的 Curry–Howard 对应.

还有另一个直接得到排中律的对应方法. 我们需要 $\mathsf{lem} : \alpha + (\alpha \to \mathbf{0})$, 回忆 $\mathbf{0}$ 是空类型, 对应假命题. 对此, 程序直接给出第二种可能, 即 $\alpha \to \mathbf{0}$. 如果接下来我们始终没有利用这个函数, 则这个函数是什么都不影响结果. 假如在某个时刻我们利用了这个函数, 那么这必然是因为我们提供了一个元素 $x : \alpha$ 作为参数. 此时, 程序跳转到一开始, 并且 "反悔", 给出第一种可能. 因为此时程序已经得到了一个元素 $x : \alpha$, 因此它只需要直接给出这个元素即可.

注意到, 尽管这确实给出了一种类型论的解释, 但是它引入了 "反悔" 的可能, 或者说引入了不确定性. 因此这就破坏了一些好的性质. 我们还有一些其他的引入经典逻辑的方式, 如 $\lambda\mu$-演算, 极化逻辑等等. 它们都是对好性质的权衡: 每一种都会保留一些类型论的好性质, 而破坏另一些性质. 这显然是不可避免的. 停机问题无法判定, 因此无论如何修改类型论, 都不可能得出一个能解决停机问题的程序. 所以要么该类型论中无法证明一个程序要么停机要么不停机, 即限制排中律可使用的范围, 要么该类型论中对此命题的证明无法计算出结果, 即弱化了构造主义数学的可计算性质.

第四章　Martin-Löf 类型论

Martin-Löf 在 70 年代前后提出了多个类似的类型论. 其中最本质的特征在于接下来介绍的相等类型. 因此我们会把具有类似特征的其他类型论也归到此类介绍.

我们首先回顾一下之前遇到的类型. 我们有依值函数的 Π-类型, 还有 Σ-类型表示后者的类型依赖于前者的值有序对的类型. 一般的函数类型 $\to$ 与乘积类型 $\times$ 可以分别表示为它们的特殊情况. 在纯类型系统中, 我们介绍了类型的类型 (如 $\mathsf{type}, \mathsf{kind}$ 等) 对类型系统的影响. 接下来我们会称这些对象为**宇宙**.

4.1　归纳类型

我们可以继续引入类型, 例如自然数. 首先有两种构造自然数的方式: 一种是 $\mathsf{zero} : \mathbb{N}$; 一种是给定一个自然数 $n : \mathbb{N}$, 取它的后继 $\mathsf{succ}(n) : \mathbb{N}$. 我们称 zero 与 succ 为**构造子**. 除了能够构造自然数之外, 我们还需要能够使用自然数.[1] Curry–Howard 对应在这里又发挥了神奇的作用: 用于证明命题的归纳法与用于定义函数的递归定义法, 实际上是同一件事情! 考虑**消去子**

$$\mathsf{elim} : \prod_{P:\mathbb{N}\to\mathsf{type}} P(\mathsf{zero}) \to \left[\prod_{m:\mathbb{N}} P(m) \to P(\mathsf{succ}(m))\right] \to \prod_{n:\mathbb{N}} P(n).$$

回忆之前提到的使用嵌套的函数类型表示多个参数的函数的技巧. 同时这里 Π-类型的优先级最低. 如果从逻辑的视角阅读这个类型, 我们得到的就是

> 对于所有含有变量 $n : \mathbb{N}$ 的命题 $P(n)$, 如果 $P(\mathsf{zero})$ 成立, 并且对于每个 $m : \mathbb{N}$, $P(m)$ 可以推出 $P(\mathsf{succ}(m))$, 那么命题就对所有的 n 成立.

这无疑就是归纳法. 而另一方面, 如果我们代入 $P(n) = \mathbb{N}$, 那么这个类型就变成了

$$\underbrace{\mathbb{N}}_{f(0)} \to [\ \overbrace{\underbrace{\mathbb{N} \to \overbrace{\mathbb{N}}^{f(m)} \to \mathbb{N}}}^{\text{利用 } m \text{ 与 } f(m) \text{ 表达 } f(m+1)}\ ] \to \underbrace{\mathbb{N} \to \mathbb{N}}_{\text{得到 } f}.$$

[1] 构造子就是输出是自然数的函数 (zero 是有零个参数的函数), 而我们需要输入是自然数的函数. 在数学中我们经常遇到的泛性质就是类似的思想.

这就是递归定义. 作为例子, 如果我们令 $f(0)=0$, $f(m+1)=f(m)+2$, 那么 f 就是将输入乘以 2 的函数. 具体写出来就是

$$\mathsf{elim}\, P\, \mathsf{zero}\, \lambda mu.\mathsf{succ}(\mathsf{succ}(u))$$

这里用了类似之前的记号, fmn 表示函数 f 接受两个参数 m 与 n. 读者可以自行和类型比对. 在 [31] 中作者用谈话的形式, 趣味十足地讲解了类似的各种构造, 读者可以参考. 对于一般的 P, 这就是依值类型下的递归. 譬如如果定义了向量空间的概念, 我们可以用消去子构造一个函数, 输入 n, 输出 n 维向量空间中的零向量. 这样就要求 $P(n)$ 是 "n 维向量空间" 这个类型, 因此依赖于 n.

为了递归能够成立, 我们还要引入相关的等式. 我们在之前也已经为函数类型引入了等式. 这实际上是表达式上的等价关系. 如果我们定义了一个函数 $f=\mathsf{elim}\, P\, z\, s$, 那么我们希望 $f(\mathsf{zero})=z$, 并且 $f(\mathsf{succ}(n))=s\, n\, f(n)$. 我们用阿拉伯数字简写自然数, 那么就有

$$f(3)=s\,2\,f(2)=s\,2\,(s\,1\,f(1))=s\,2\,(s\,1\,(s\,0\,z)).$$

这些类型可以推广到一般的**归纳类型**, 以及更强大的一些变体, 如 Martin-Löf 引入了 W-类型的概念. 我们不详细介绍.

4.2 相等类型

数学中最常见的命题就是两个对象的相等. 因此由 Curry–Howard 对应, 我们应该有一个类型反映这一点. 对于两个同类型的元素 $x, y : A$, 我们有**相等类型** $x =_A y$.[2] 如果没有歧义, 可以简写成 $x=y$.

注意这和我们之前使用的等号含义有所区别. 之前使用的等号, 其描述的相等性都是 "不言自明" 的. 如将一个复杂的表达式用一个字母缩写, 此时我们认为这个缩写并不存在于类型论中, 只是为了人们实际书写方便而引入的. 还有如 $(\lambda x.x)y=y$ 的等式; 上面介绍了自然数可以用递归定义加法, 则 $3+5=8$ 也是类似的等式. 这些等式在很多情况下都是有机械的化简或者判定相等的方法的. 然而, 我们引入的相等类型可以描述如 $m+n=_{\mathbb{N}} n+m$ 或者 $\zeta(2)=_{\mathbb{R}} \frac{\pi^2}{6}$ 这样的需要证明的定理. 这些显然没有机械的判定方法, 否则数学家都要失业了! 我们称前者为**判值相等** (judgemental equality), 而后者 (即相等类型) 为**命题相等** (propositional equality). 自然, 判值相等的东西也命题相等.

命题相等还有一个特点, 就是我们可以用它们来构造更复杂的命题, 如 $\neg(a^3+b^3=_{\mathbb{N}} c^3)$ 等等. 判值相等是表达式上的等价关系, 并不是一个类型, 所以不能参与构造更复杂的命题.

为什么我们需要这样辨析相等的概念呢? 其实前一章中已经暗示了这一点. 假如我们有一个 $\mathbb{R}^{1+1}$ 类型中的元素, 那么我需要能将它用于 $\mathbb{R}^2$ 中. 这就需要类型论中有一个可以

[2] 这在文献中有许多不同的记号. 有些文献使用 $\equiv$, 还有一些文献直接使用 $\mathsf{Id}_A(x,y)$.

机械地判断的相等概念, 否则就无法机械地判断某个表达式类型是否正确, 进而在 Curry–Howard 对应下就意味着无法判断某个证明是否真的证明了命题. 一个类型论不一定需要有命题相等, 但是一定有判值相等的概念. 当然这个概念可以是平凡的, 比如只有字面相等的表达式才判值相等; 这就产生了**弱类型论** (weak type theory). 在另一个极端, 我们也可以设定一条规则, 让命题相等的概念和判值相等的概念重合:

$$\frac{M =_A N}{M = N}.$$

这就产生了**外延类型论** (extensional type theory).[3] 这样的代价就是判值相等不再可以机械地判定, 因此在其中写下一个命题的证明, 也没有明确的办法判断这是否是正确的证明. 这三种类型论的联系可以参考 [64].

不过, 我们还得先说明相等类型本身的规则是什么. 首先是构造子

$$\mathsf{refl}_a : a =_A a.$$

这对应同一律: 每个元素都等于其自身. 我们有对应的消去子

$$\mathsf{J} : \prod_{P:\prod_{a,b:A} a=b\to\mathsf{type}} \left[\prod_{c:A} P\,a\,a\,\mathsf{refl}_a\right] \to \prod_{a,b:A}\prod_{p:a=b} P\,a\,b\,p$$

这说的是 “如果 P 对所有的 refl 都成立, 那么它就对所有的 $p : a =_A b$ 成立” . 在实用的类型论中, 通常会允许省略前面两个参数. 因此可以直接写做 $P\,\mathsf{refl}_a$, 可以通过上下文推导出两个省略的参数都是 a. 同时我们为了方便也会将 refl_a 省略为 refl. 类似地, 消去子和构造子的组合有一个等式, 这里不再给出. 读者可以参考 [62]. 用字母 J 是因为 Martin-Löf 在论文 [46] 中的相等类型使用了 I, 而下一个字母就是 J.

神奇的是, 用 J 原理就足够证明等式的各种性质, 比如对称性

$$\prod_{a,b:A} a = b \to b = a$$

与传递性

$$\prod_{a,b,c:A} a = b \to b = c \to a = c.$$

给定任何一个表达式 $P(x)$, 如果有 $a = b$, 那么 $P(a) = P(b)$. 如果类型 $A = B$, 那么其间有一个双射. 这些性质的证明, 读者可以参考 [62, §1.12].

不过仍然有一些问题无法只靠 J 原理解决. 其中最简单的例子是函数的**外延性**, 即如果两个函数逐点相等, 那么它们相等.

$$\mathsf{funext} : \prod_{f,g:A\to B} \left[\prod_{x:A} f(x) = g(x)\right] \to f = g.$$

[3] 相对应的不含这条规则的 Martin-Löf 类型论又称作**内涵类型论** (intensional type theory). 以后谈到 Martin-Löf 类型论, 若不特别说明, 均指内涵类型论.

这在 Martin-Löf 类型论中是无法证明或者证伪的. 在 [11] 中有非常简单易懂的证明. 这可以看作是 Martin-Löf 类型论的一个缺点, 也可以理解为一种优点. 比如将函数看作算法, 那么尽管两个函数逐点相等, 它们的算法不同, 计算需要的时间也不同. 因此函数外延性在研究算法复杂度时就的确是不成立的. 尽管如函数外延性之类的命题无法证明, 我们还是可以将这些命题作为额外的公理加入到理论中.

Martin-Löf 类型论中还有一个重要的缺憾是没有**商类型**. 在数学中, 商去一个等价关系是极其常见的操作, 但是在 Martin-Löf 类型论中很难加入对应的类型, 同时保证类型论仍然具有好的性质. 注意我们可以直接将商类型作为公理加入, 但是其他的类型均不需要公理, 这在审美上有一点缺陷. 同时在证明类型论的性质时, 这种任意加入的公理一般很难处理. 另外, 对于其他的类型, 我们都有对应的判值相等等式, 而公理只能影响命题相等, 因此判值相等的性质就被破坏了. 不过, 使用在 3.3.1 节中提到的广集的概念, 可以一定程度上规避这个问题. 这样做的缺点是每个广集上都自带一个不同的等价关系作为“相等”, 因此需要时刻处理这些额外的信息, 在严格书写证明的时候非常繁琐.

4.3 宇宙

在 Martin-Löf 类型论最早的版本 (于 1971 年提出, 一般被称作 MLTT_{71}) 中, 只有一个宇宙 type, 满足 $\mathsf{type} : \mathsf{type}$. 这个系统中存在 Russell 悖论. 因此后继的几个版本中, Martin-Löf 对此做出了一些修改. MLTT_{73} 中, 引入了一套**宇宙层级**, 即引入可数个宇宙 $\mathsf{type}_i : \mathsf{type}_{i+1}$. 其中 i 不是类型论内部的自然数, 而是我们为了书写方便引入的, 实际上有可数个不同的符号.

宇宙的规则与之前提到的构造演算有一个重要的不同. 构造演算有两层宇宙 $\mathsf{type} : \mathsf{kind}$, 我们写成 $\mathsf{type}_i (i = 1, 2)$. 那么如果 $A : \mathsf{type}_i, B : \mathsf{type}_j$, 函数类型就有 $A \to B : \mathsf{type}_j$. 注意函数类型一定在后者的宇宙之中, 尽管有可能 $i > j$. 这意味着这里的函数类型表现得与集合论中的函数集合相差甚远, 因为考虑函数类型 $\mathsf{type}_1 \to A$, 其中 $A : \mathsf{type}_1$ 是任意一个类型. 那么这个函数类型的定义域中也包含 $\mathsf{type}_1 \to A$, 但是如 ZFC 集合论中函数集合的定义域无论如何也不能包含这个函数集合自身!

在 Martin-Löf 类型论中, 如果 $A : \mathsf{type}_i, B : \mathsf{type}_j$, 函数类型就有 $A \to B : \mathsf{type}_{\max\{i,j\}}$. 这使得 Martin-Löf 类型论成为一种**直谓类型论** (predicative type theory).[4] 这样, 在一般数学中的一些非直谓的定义就会出现一些复杂的变化. 譬如在拓扑的定义中, 我们需要考虑任何一族开集的并. 假设开集的类型是 $\Omega : \mathsf{type}_i$, 那么如何描述一族开集 U_α 呢? 我们需要一个指标 $\alpha : A$, 因此 U 可以看作是 $A \to \Omega$ 的函数. 这便出现了一个问题: 类型 $A : \mathsf{type}_j$ 中, j 的取值应该是什么? 并不是任意选择都合理, 因为我们可能无法把一个集合 S 的内部定义成所有开集 $U \subseteq S$ 的并, 它的宇宙层级不正确.

这些问题在经典数学中其实也有体现, 为了避免 Russell 悖论, 一部分数学家 (包括

[4] 直谓一词有多个相关但不等同的含义, 这里只是一种含义.

Russell 本人) 提出我们应当避免非直谓的操作. 在受到这些限制的情况下发展的数学, 统称为**直谓数学**. 构造主义又给直谓数学增添了新的内容: 有些公理与排中律可以共同推出一些非直谓的结论, 因此经典直谓数学中拒绝这些公理. 在构造主义直谓数学中却可以允许这些公理存在. 读者可以阅读 [30] 进一步了解直谓数学.

构造演算中允许高度的非直谓性. 这导致轻微的拓展就容易出现 Russell 类的悖论. 作为妥协, 在很多变种中, 只允许最底层的宇宙有非直谓性. 换言之, 如果 $A : \mathsf{type}_i$, $B : \mathsf{type}_j$, 那么 $A \to B : \mathsf{type}_{f(i,j)}$, 其中

$$f(i,j) = \begin{cases} 1 & (j = 1) \\ \max\{i, j\} & (j > 1) \end{cases}$$

最底层的宇宙通常被称为命题宇宙, 写作 prop. 构造演算原本不存在集合论的对应, 但是如此修改之后, 可以将 prop 中的类型对应为集合论中的命题. 这样修改之后, 构造演算就能加入类似 Martin-Löf 类型论中的归纳类型与相等类型等等. 不过底层的非直谓性仍然会给类型论设计带来一些困难, 因此为了避免悖论需要注意很多技术细节. 这种类型论称为**归纳构造演算** (Calculus of Inductive Constructions, 缩写为 CIC).

4.4 自洽性

由于对宇宙的直谓性安排, 很容易证明 Martin-Löf 类型论有**自洽性**, 即不存在元素 $u : \mathbf{0}$. 这里 u 不能含有自由变量. 根据 Curry–Howard 对应, 这意味着理论没有矛盾.

我们将每个类型解释为集合. 如 $\mathbf{0}$ 是空集, $\mathbb{N}$ 是自然数集, 等等. type_1 则是集合的集合, 并且它在 Σ-类型, Π-类型, 相等类型等等操作下封闭. 这在集合论中对应一个熟知的构造. 考虑一个基数 $\kappa > \aleph_0$, 使得任何少于 κ 个小于 κ 的基数相加仍然小于 κ, 并且任何小于 κ 的基数幂集仍然小于 κ. 这种基数称为**强不可达基数**. 此时集合论中的 von Neumann 宇宙 V_κ 就包含自然数集合, 并且关于依值函数集等操作封闭. 因此我们可以将 type_1 解释为 V_κ 接下来, 我们设 type_2 解释为包含 type_1, 并且在同样的操作下封闭的集合, 依次类推. 这也就是说要有一列强不可达基数 $\kappa_1 < \kappa_2 < \ldots$, 这样就能每个类型都赋予了一个集合. 并且对于每个类型论的元素 $x : A$, 我们都能在 A 对应的集合中找到一个对应的元素. 而空集没有元素, 因此类型 $\mathbf{0}$ 中没有元素.

然而, 依值类型论中只有自洽性是不够的. 因为有自洽性不代表存在一个机械的方法判断某个证明是否正确. 我们需要更强的典范性. Martin-Löf 类型论的典范性证明, 要么初等而极其繁琐, 要么需要一定的范畴论知识. 我们在之后会介绍一定的范畴论工具. 读者也可以参考 [56, §5.6].

4.5 应用

Martin-Löf 类型论及其变体有非常广泛的应用, 这里拾取一些介绍.

4.5.1 Coq/Rocq

Coq (现已更名为 Rocq) 是一套交互式定理证明软件. 它基于构造归纳演算, 名字也来源于构造演算的缩写 CoC 与其提出者 Coquand 的名字. 它的主要特点是允许一种接近自然语言的证明方法.

我们假设已经有了自然数的加法操作, 满足 $0 + n = n$, $\mathsf{succ}(m) + n = \mathsf{succ}(m + n)$. 回忆这里 succ 指的是自然数的后继. 考虑一个自然语言的证明.

定理 4.1. *对于任何自然数 m, $m + 0 = 0$.*

证明. 只需要对 m 归纳. m 为 0 时由于 $0 + n = n$ 对任何 n 成立, 代入 0 即可. 当 m 为 $\mathsf{succ}(m')$ 时, 根据已知条件有 $\mathsf{succ}(m') + 0 = \mathsf{succ}(m' + 0)$, 再由归纳假设得到 $m' + 0 = m'$, 从而命题成立. □

在 Martin-Löf 类型论中直接写出这个命题的证明是非常长的, 并且写完之后难以阅读. 读者可以尝试一下, 用于检测自己对类型论的知识是否熟悉. 在 Coq 中提供了**证明策略** (tactic) 的语言, 此时这个命题的证明可以写成

```
Proof.
  intros m. induction m as [| m' H ].
  - apply add_zero.
  - rewrite add_succ. rewrite H. reflexivity.
Qed.
```

这与自然语言几乎完全一致. 第一句话用策略 `intros` 引入了变量 m, 第二句话对其进行归纳. 这里 m' 就是在第二种情况下的变量名, H 是归纳假设. 接下来的第一种情况是 $m = 0$, 此时我们使用一条已知的定理 `add_zero`, 即 $0 + n = n$. Coq 自动推断出这里应当取 $n = 0$. 第二种情况是 $m = \mathsf{succ}(m')$, 此时我们有归纳假设 $H : m' + 0 = m'$. 我们的目标是证明 $\mathsf{succ}(m') + 0 = \mathsf{succ}(m')$. 现在我们使用已知的定理 $\mathsf{succ}(m') + n = \mathsf{succ}(m' + n)$ 将等式左侧改写, 得到 $\mathsf{succ}(m' + 0) = \mathsf{succ}(m')$. 接下来再使用归纳假设改写得到 $\mathsf{succ}(m') = \mathsf{succ}(m')$. 而这使用等式的自反性即可.

同时, 注意 Coq 是交互式定理证明软件. 在输入这个证明时, Coq 会时刻告诉我们当前这一步有哪些条件, 目前还有哪些目标. 同时, 我们可以自己编写强大的自动化证明策略, 如 `tauto` 策略可以自动解决命题逻辑的证明 (即不涉及 $\forall, \exists$ 的操作). 不过需要注意的是, Coq 的核心功能是检查人类的证明. 因此它的首要目标是保证检查没有疏漏. 一些其他的软件目标在于自动化证明尽可能多的命题, 它们的目标不同, 因此是无法相互比较的.

在 2005 年, Georges Gonthier 等人在 Coq 中完全形式化了四色定理的证明. 这个定理目前人类所知的证明中涉及到了上千种情况的讨论, 因此人力几乎不可能保证其正确无误. 在 Coq 形式化过程中, 也发现了不少对计算机科学有用的图论技巧. 读者可以阅读发表在《美国数学学会通讯》(*AMS Notices*) 上的文章 [33] 中对证明思路与技巧的概述, 只需要高中数学水平, 不需要任何 Coq 知识.

在计算机领域, Coq 也经常用于验证软件代码是否正确. 2009 年, Xavier Leroy 等人开发了完全经由 Coq 验证的代码编译器 CompCert [42], 证明了这样规模的形式化验证在实践中是可行的.

4.5.2 Lean

Lean 是基于归纳构造演算的通用编程语言与交互式定理证明软件. 它的社区的主要目标是构建一套完整的数学定理库. 为此, 在大部分时候它都使用了排中律作为公理. 同时, 它也直接将商类型作为公理加入类型论中. 尽管前文中说过这都会破坏类型论的性质, 但是由于 Lean 社区的主要目标不同, 这并不非常重要. Lean 社区注重数学家的使用体验, 因此会对许多使用细节针对性优化. 同时, Lean 也有一套强大的证明策略语言.

正如文章开头提到的, Lean 社区在一年半的时间中, 完成了 Fields 奖得主 Scholze 提出的一个挑战, 证明了 Lean 在数学领域的建设是非常成功的. 有本科数学基础的读者如果想要尝试上手一款定理证明软件, 那么笔者推荐使用 Lean. Coq 与 Agda 对有计算机科学背景的读者更加友好.

4.5.3 Agda

Agda 也是基于类似 Martin-Löf 类型系统的交互式定理证明软件. 它尽管也有证明策略语言, 但是主要特征是**类型驱动开发**. 用户在输入证明时, 可以随时留下一个“洞”. 此时 Agda 会提示这个洞需要填入什么类型的表达式, 与当前有哪些可用的条件. 用户可以让 Agda 执行一些操作修改洞附近的代码, 在这个与软件合作的过程中逐步完成整个证明. 譬如证明定理 $\forall n. n + 0 = n$.

```
theorem : (n : Nat) -> n + 0 == n
theorem = ?
```

这里的问号就是待填入的内容. 此时可以使用快捷键命令 Agda 引入变量, 它便会自动将代码修改为

```
theorem : (n : Nat) -> n + 0 == n
theorem n = ?
```

接下来, 我们命令 Agda 对 n 归纳, 代码就会变为

```
theorem : (n : Nat) -> n + 0 == n
theorem zero = ?
theorem (succ x) = ?
```

以此类推. 由于这里类型系统强大的表达能力, 我们可以直接使用类型指导应该填入哪些东西.

Agda 实现了非常多前沿的类型论, 如可以开启立方类型论模式, 就能在立方类型论中进行定理证明. Agda 还支持自定义公理及其重写规则来模拟全新的类型论. 因此, Agda 对类型论研究者非常友好, 便于进行实验.

4.5.4 其他

逻辑框架 (logical framework) 虽然规则上大体相似, 但是与上文描述的 Martin-Löf 类型论的使用方式风格非常不同. 一个具体的软件实现是 Twelf, 主要用于程序验证等.

Nuprl 与 Andromeda 是基于外延类型论的计算机辅助证明软件. 尽管上文中我们说过外延集合论中一个证明是否正确难以机械判定, 但是这些软件中允许用户手动插入额外的说明, 用于帮助软件进行判断. Sterling 与 Favonia 等人在 Nuprl 的思想基础上开发了一种积立方类型论的原型实现 RedPRL.

第五章　范畴语义

本章介绍类型论的范畴语义. 注意我们研究类型论的语义时, 类型论是被研究的数学对象, 而不是正在使用的数学语言. 此时使用的语言可以自由选择, 例如自然语言、集合论甚至类型论本身. 这个外部的语言称作**元语言**, 而被研究的类型论语言就直接称作“语言”. 切忌混淆元语言中的概念与语言内的概念. 另一个容易混淆的事情, 在于语义中的许多概念都是为了与语法中的事物相对应, 因此二者会用相近的名称与记号, 但是它们也不应当混同.

类型论模型的一个极重要的特征就是其 (广义的) 代数性. 何为代数性? 最狭义来说, 一个代数结构就是某集合上配备一些运算, 并且满足形如 $\forall x_1, x_2, \ldots, M = N$ 的公理. 例如群、环、$\Bbbk$-向量空间等等. 代数结构通常满足一系列性质, 例如同态定理, 三条同构定理等. 对代数结构可以推广得到**本质代数结构**或者**广义代数结构**. 而类型论的模型一般都可以写成这种形式, 进而可以运用代数结构的一般结论.

历史上提出了很多描述类型论语义的方案. 为了使叙述更加清晰, 我们先介绍 Awodey 提出的自然模型 [3], 再简单补充一些历史源流. [48] 也有较为详细的讲解.

5.1　范畴与模型

我们先来思考最类型论最直观的模型, 将类型解释为集合. 函数类型对应函数集合, 乘积类型对应乘积集合, 等等. 如果有依值类型 $x{:}A \vdash B(x)$, 那么 $B(x)$ 应当对应“依值集合”, 也就是一族集合 Y_x, 其中 x 取遍 A 所对应的集合. 不过, 在一般情况下, $\Gamma \vdash B$ 依赖于语境 Γ 中的多个类型. 因此我们应该将语境解释为集合, 而类型解释为集合族. 如果 Γ 解释为集合 X, $\Gamma \vdash B$ 解释为集合族 $Y_{x\in X}$, 那么新语境 $\Gamma, y{:}B$ 应该解释为集合族的不交并 $\coprod_{x\in X} Y_x$.

这是定义依值类型论的模型时出现的一般现象. 我们先直观地认为类型应该解释为某物, 而后对于依值类型我们提出对应的“依值某物”, 最后完整的解释是将语境对应为某物, 而类型对应于依值某物.

一般来说, 这里提到的“某物”往往构成一个范畴, 并且许多语义上的操作都对应范畴论中早有研究的概念, 因此便有了范畴语义的研究. 直觉上, 类型之间的箭头应该是函数. 但参考前文所说, 我们应当考虑语境之间对应的箭头. 不难看出, 既然语境是一列类型 $\Gamma = (x_1{:}A_1, \ldots, x_n{:}A_n)$, 那么若 $\Delta = (y_1{:}B_1, \ldots, y_m{:}B_n)$, 则合适的箭头 $\Gamma \to \Delta$ 应当是一列

表达式 $(M_1, \dots, M_m)$, 使得 $\Gamma \vdash M_k : B_k$. 对于 $m = n = 1$ 的情况, 这就与函数是一致的. 不过, 因为有依值类型, B_k 中可能用到了变量 $y_1, \dots, y_{k-1}$. 这时候要将它们对应地替换成 $M_1, \dots, M_{k-1}$. 我们将这列表达式 M_k 记作 σ, 整体写作 $\Gamma \vdash \sigma : \Delta$. 这些在 2.4.2 中已有提及, σ 称作从 Γ 到 Δ 的**代换**.

给定任何语境 Γ, 我们应该有一个集合 $\mathrm{Tp}(\Gamma)$, 表示这语境下合法的类型构成的集合. 如果再有一个代换 $\Delta \vdash \sigma : \Gamma$, 那么就可以得到函数 $\mathrm{Tp}(\Gamma) \to \mathrm{Tp}(\Delta)$. 注意这里方向是相反的. 所有这些函数合在一起就构成了函子 $\mathcal{C}^{\mathrm{op}} \to \mathsf{Set}$, 也就是一个预层. 因此, 对于一般的模型来说, 我们也应当有一个范畴 $\mathcal{C}$ 与其上的预层 Tp.

对于类型的元素来说, 有多种等价的提法. 其中最简单的办法是考虑某个语境中所有类型的元素的不交并 $\mathrm{Tm}(\Gamma)$, 以及映射 $\pi : \mathrm{Tm}(\Gamma) \to \mathrm{Tp}(\Gamma)$ 为每个元素赋予类型. 这就构成了两个预层之间的映射 $\pi : \mathrm{Tm} \to \mathrm{Tp}$. 如果读者不喜欢将不同类型的元素放在同一个集合中, 之后会讨论一些避免这种情况的等价的表述.

给定语境 Γ 与类型 $A \in \mathrm{Tp}(\Gamma)$,[1] 我们应当能构造出语境 $\Gamma, x{:}A$. 因为变量名无关紧要, 我们也写作 (Γ, A). 正如范畴论中的大部分构造一样, 我们应当找出语法中它的泛性质, 并将其作为模型的定义中的一条需要满足的性质.

泛性质无非二者居其一: 描述从该对象出发的箭头有哪些, 或者描述指向该对象的箭头有哪些. (Γ, A) 的泛性质是后者. 要构造 $\Delta \to (\Gamma, A)$ 的代换, 就需要先构造 $\Delta \to \Gamma$ 的代换 σ, 再构造一个元素 $\Delta \vdash t : A[\sigma]$. 换句话说, 这是说我们有一个拉回方

$$\begin{array}{ccc} \hom(\Delta, (\Gamma, A)) & \longrightarrow & \mathrm{Tm}(\Delta) \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \pi} \\ \hom(\Delta, \Gamma) & \xrightarrow[\sigma \mapsto A[\sigma]]{} & \mathrm{Tp}(\Delta) \end{array}$$

这就完全定义了 (Γ, A). 由于对每个 Δ 都有这样的拉回方, 我们可以将其综合为预层上的拉回操作. 这就得到了自然模型的定义.

5.2 类型论的自然模型

定义 5.1. 一个**自然模型**是任意一个有终对象的范畴 $\mathcal{C}$, 配备两个预层与其间的态射 $\pi : \mathrm{Tm} \to \mathrm{Tp}$, 使得对于任何 $\Gamma \in \mathcal{C}$ 与 $A \in \mathrm{Tp}(\Gamma) \cong \hom(よ(\Gamma), \mathrm{Tp})$, 有对象表出以下拉回预层:

$$\begin{array}{ccc} \bullet & \longrightarrow & \mathrm{Tm} \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \pi} \\ よ(\Gamma) & \xrightarrow[A]{} & \mathrm{Tp} \end{array}$$

[1] 需要时刻注意的是, 我们正在使用语法中的现象来启发语义的定义. 因此这里的 “语境” 既可以指语法中的一个具体的语境, 也可以指其模型中的一个对象 $\Gamma \in \mathcal{C}$. 以集合模型为例, 语法是能具体写出表达式的东西, 而语义中这些则对应集合与集合之间的函数. 显然并不是所有集合都能写出具体的表达式, 因此这两者需要区分.

此对象记作 (Γ, A). 满足这种条件的态射也称作**可表态射**. 另外, 将态射 $(\Gamma, A) \to \Gamma$ 记作 p_A, 交换图上沿的态射记作 q_A.

当然, 无需范畴论语言也可以等价地表达这一定义. 但本文不是类型论的教材, 因此就按下不表.

考虑所有合法语境在判值相等关系下构成的范畴 $\mathcal{C}$, 语境之间的态射 $\sigma \in \hom(\Delta, \Gamma)$ 是代换, 并且判值相等的代换视为相同. 这里, 终对象是空语境. 对于每个语境 Γ, 考虑允许包含 Γ 中的变量的合法的类型构成的集合 $\mathrm{Tp}(\Gamma)$, 同样将判值相等的类型视为相同. 这些类型的元素的不交并记作 $\mathrm{Tm}(\Gamma)$. 这样, 如果 Γ 下 A 是合法的类型, 又有代换 $\sigma \in \hom(\Delta, \Gamma)$, 那么就可以将 A 中的变量代换得到 $A[\sigma]$, 是 Δ 下合法的类型. 这就证明了 Tp 是预层. 同理 Tm 也是预层. π 将每个 $a \in \mathrm{Tm}(\Gamma)$ 映射到它所属的类型.

我们来计算定义中提到的拉回. 我们知道预层的拉回是逐点计算的, 因此拉回 F 满足

$$\begin{aligned} F(\Delta) &\cong \hom(\Delta, \Gamma) \times_{\mathrm{Tp}(\Delta)} \mathrm{Tm}(\Delta) \\ &= \{(\sigma, a) \mid \pi(a) = A[\sigma]\} \\ &= \hom(\Delta, (\Gamma, A)). \end{aligned}$$

因此 F 的确可表, 并且其表出对象就对应语境的扩展操作 (Γ, A).

以上就证明了类型论的语法范畴构成自然模型, 记作 $\mathbf{T}$.[2] 以此为模板, 就可以找到类型论语法中许多概念的语义对应. 例如, 类型论中的语境对应自然模型中 $\mathcal{C}$ 的对象, 语境之间的代换对应 $\mathcal{C}$ 中的态射. 语境中的类型对应 $\mathrm{Tp}(\Gamma)$ 的元素. 给定 $A : \mathrm{Tp}(\Gamma)$, 类型论中 A 的元素对应模型中的

$$\{a \mid a \in \mathrm{Tm}(\Gamma), \pi(a) = A\}.$$

由拉回的性质, 可以看出这等价于

$$\{\alpha \mid \alpha \in \hom(\Gamma, (\Gamma, A)), \mathsf{p}_A \circ \alpha = \mathsf{id}_\Gamma\}.$$

对于 $A : \mathrm{Tp}(\Gamma), \sigma : \Delta \to \Gamma$, 由于 Tp 是预层, 我们可以得到 $A[\sigma] : \mathrm{Tp}(\Delta)$. 这是类型论中代换的对应. 而由于下图中右侧的正方形与外侧的正方形均是拉回,

$$\begin{array}{ccccc} よ(\Delta, A[\sigma]) & \dashrightarrow & よ(\Gamma, A) & \longrightarrow & \mathrm{Tm} \\ \downarrow & & \downarrow & & \downarrow \\ よ(\Delta) & \longrightarrow & よ(\Gamma) & \longrightarrow & \mathrm{Tp} \end{array}$$

我们得到虚线的箭头也存在, 使得整体构成交换图并且左侧的正方形是拉回. 因此在 $\mathcal{C}$ 中有这样一个拉回.

$$\begin{array}{ccc} (\Delta, A[\sigma]) & \dashrightarrow & (\Gamma, A) \\ \downarrow & & \downarrow \\ \Delta & \longrightarrow & \Gamma \end{array}$$

[2] 当然, 因为我们还没有引入任何类型, 语法范畴其实是平凡的. 但是上面的证明可以随着加入新的类型而随时拓展.

在一些其他的类型论的模型的定义中, 不涉及预层范畴, 而是直接将这个拉回作为类型论中代换的定义.

如何说明自然模型的确是合适的概念呢? 模型最重要的作用, 是语法中的概念可以解释到所有模型中. 用范畴的语言来说, 对于任何模型 $\mathbf{M}$, 都有唯一的保持结构的态射 $\mathbf{T} \to \mathbf{M}$. 这一点可以靠对语法归纳来证明. 我们之后会定义带有 Π-类型的模型, 带有 Σ-类型的模型, 等等. 它们各自也有对应的性质: 考虑带有 Π-类型的类型论构成的语法模型, 它到任何带有 Π-类型的模型都有唯一的保持结构的态射, 以此类推.

上面还没有定义何为保持结构的态射. 以下为了方便, 对于模型 $\mathbf{M}$, 记它对应的范畴为 $\mathcal{C}_\mathbf{M}$, 配备的预层是 $\mathrm{Tm}_\mathbf{M}, \mathrm{Tp}_\mathbf{M}$, 有可表态射 $\pi_\mathbf{M}$. 假如有函子 $F : \mathcal{C} \to \mathcal{D}$, 那么复合上这个函子, 就给出了预层范畴之间的映射 $F^* : \mathsf{Psh}(\mathcal{D}) \to \mathsf{Psh}(\mathcal{C})$.

定义 5.2. 给定两个自然模型 $\mathbf{M}, \mathbf{N}$, 定义一个**态射** $F : \mathbf{M} \to \mathbf{N}$ 为一个函子 $F : \mathcal{C}_\mathbf{M} \to \mathcal{C}_\mathbf{N}$, 两个态射 $F_{\mathrm{Tp}}, F_{\mathrm{Tm}}$ 使得以下图表交换:

$$\begin{array}{ccc} \mathrm{Tm}_\mathbf{M} & \xrightarrow{F_{\mathrm{Tm}}} & F^*\mathrm{Tm}_\mathbf{N} \\ {\scriptstyle \pi_\mathbf{M}}\downarrow & & \downarrow{\scriptstyle F^*\pi_\mathbf{N}} \\ \mathrm{Tp}_\mathbf{M} & \xrightarrow[F_{\mathrm{Tp}}]{} & F^*\mathrm{Tp}_\mathbf{N} \end{array}$$

此时对于任何 $\Gamma \in \mathcal{C}_\mathbf{M}$ 与 $A \in \mathrm{Tp}_\mathbf{M}(\Gamma)$, 有 $F(\Gamma) \in \mathcal{C}_\mathbf{N}$ 与 $F_{\mathrm{Tp}}(A) \in \mathrm{Tp}_\mathbf{N}(F(\Gamma))$, 并且自然诱导一个态射 $F(\Gamma, A) \to (F(\Gamma), F_{\mathrm{Tp}}(A))$. 我们要求这个态射是同构.

其中, 最后这个条件是在要求自然模型之间的态射保持 $(-,-)$ 这个运算. 如果这不仅是同构, 而是直接严格相等, 就说 F 是自然模型之间的**严格态射**.

5.2.1 自然模型中的类型

自然模型仅仅是搭建起了类型论模型的基本框架. 接下来, 我们要为这个框架加入具体的类型. 先从最简单的单元素类型开始.

我们同样从语法范畴寻找启发. 对于有单元素类型的类型论, 在任何语境 Γ 下都有 $\Gamma \vdash \mathsf{Unit}\,\mathrm{type}$. 因此应当有预层的态射 $1 \to \mathrm{Tp}$, 在 Tp 中指出单位类型. 而单位类型在任何语境下都恰好有一个元素. 因此如果考虑拉回

$$\begin{array}{ccc} \bullet & \longrightarrow & \mathrm{Tm} \\ \downarrow & \lrcorner & \downarrow \\ 1 & \xrightarrow[\mathsf{Unit}]{} & \mathrm{Tp} \end{array}$$

那么其左上角应当也是 1.

定义 5.3. 给定自然模型, 如果有一个映射 $\mathsf{Unit} : 1 \to \mathrm{Tp}$, 使得它与 π 拉回得到的也是终对象, 那么就称 Unit 是该自然模型的**单位类型结构**.

注意单位类型是结构而不是性质, 也就是说同一个自然模型可能有两种不同的单位类型结构, 这是因为两个不同的单位类型可以仅仅是同构而不严格相等.

更复杂的类型也是相似的. 接下来考虑 Π-类型. 它的规则是

$$\frac{\Gamma \vdash A\,\mathrm{type} \quad \Gamma, x{:}A \vdash B\,\mathrm{type}}{\Gamma \vdash \prod_{x:A} B\,\mathrm{type}}$$

因此我们需要考虑一个预层

$$F(\Gamma) = \{(A, B) \mid A \in \mathrm{Tp}(\Gamma), B \in \mathrm{Tp}(\Gamma, A)\}.$$

这个定义的自然性来自 Tp 与 $(-,-)$ 的自然性. 与单位类型类似, 我们需要寻找预层的态射 $\mathsf{Pi} : F \to \mathrm{Tp}$, 指出由 A, B 构成的 Π-类型. Π-类型的元素, 可以通过 $\Gamma, x{:}A \vdash b : B$ 构造. 因此考虑另一个预层

$$G(\Gamma) = \{(A, B, b) \mid b \in \mathrm{Tm}(\Gamma, A), \pi(b) = B\},$$

其中 A, B 取值范围与 F 中相同. 有自然变换 $G \to F$. 至此, 我们应当希望有一个交换方

$$\begin{array}{ccc} G & \longrightarrow & \mathrm{Tm} \\ \downarrow & & \downarrow{\scriptstyle \pi} \\ F & \xrightarrow[\mathsf{Pi}]{} & \mathrm{Tp} \end{array}$$

这就编码了 Π-类型的引入规则, 即 λ 操作. 接下来, 我们要求这个交换方是拉回.

定义 5.4. 给定自然模型, 如果有一个映射 $\mathsf{Pi} : F \to \mathrm{Tp}$, 使得它与 π 拉回得到的是 $G \to F$, 那么就称 Pi 是该自然模型的 **Π-类型结构**.

这就同时得到了 Π-类型的消去规则 (即函数求值), 还有 β 与 η 等式. 更加详细的解释与证明, 可以参考 [3, 定理 8]. Σ-类型也可以类似操作. F 不变, 定义预层

$$H(\Gamma) = \{(A, B, a, b) \mid a, b \in \mathrm{Tm}(\Gamma), \pi(a) = A, \pi(b) = B[a]\}$$

其中 $B[a]$ 是将 a 看作对应的代换 $\Gamma \to (\Gamma, A)$.

定义 5.5. 给定自然模型, 如果有一个映射 $\mathsf{Sigma} : F \to \mathrm{Tp}$, 使得它与 π 拉回得到的是 $H \to F$, 那么就称 Sigma 是该自然模型的 **Σ-类型结构**.

我们证明集合范畴构成含有 Σ-类型与 Π-类型结构的自然模型. 大致上说, Σ-类型结构就是集合族的不交并, 而 Π-类型就是集合族的乘积.

不过, 我们需要避免 Russell 悖论相关的问题, 不能直接取全体集合构成的范畴. 考虑一个强不可达基数 κ, 也就是一个很大的基数, 像如来神掌, 使得用 Σ 和 Π 都逃不出这个基数. 这样集合论宇宙 V_κ 中的集合构成一个范畴 Set_κ, 从我们的目的来看, 就如全体集合的范畴一样的无垠, 在 Σ, Π 操作下都封闭. 但是从集合的视角来看, 它只是一个小范畴, 因此避免了处理真类的麻烦. 如果读者暂时不关心真类的问题, 可以直接取 Set.

对于任何 $\Gamma \in \mathsf{Set}_\kappa$, 取 $\mathrm{Tp}(\Gamma)$ 为映射集 $\Gamma \to \mathsf{Set}_\kappa$. 换句话说, 一个包含 Γ 中变量的类型对应一个集合族. 而一个类型 A 的元素就是这个集合族中的元素族. 换句话说是 $\prod_{x\in\Gamma} A(x)$. 这样, $\mathrm{Tm}(\Gamma)$ 为不交并

$$\coprod_{A\in(\Gamma\to\mathsf{Set}_\kappa)} \prod_{x\in\Gamma} A(x).$$

有显然的态射 $\mathrm{Tm} \to \mathrm{Tp}$. 我们还需要证明此态射可表. 对于任何 $\Gamma \in \mathsf{Set}_\kappa$ 与 $A : \Gamma \to \mathsf{Set}_\kappa$, 计算可表态射定义中的拉回

$$[ɤ(\Gamma) \times_{\mathrm{Tp}} \mathrm{Tm}](\Delta) = \{(\sigma, B, f) \mid A \circ \sigma = B\} = \coprod_{\sigma\in(\Delta\to\Gamma)} \prod_{x\in\Delta} A(\sigma(x))$$

其中 $\sigma \in \hom(\Delta, \Gamma), B \in (\Delta \to \mathsf{Set}_\kappa), f \in \prod_{x\in\Delta} B(x)$. 使用集合乘积的分配律就得到

$$\coprod_{\sigma\in(\Delta\to\Gamma)} \prod_{x\in\Delta} A(\sigma(x)) \cong \Delta \to \coprod_{x\in\Gamma} A(x).$$

因此定义 $(\Gamma, A) = \coprod_{x\in\Gamma} A(x)$, 则上面的拉回预层被 (Γ, A) 表出, 故 $\mathrm{Tm} \to \mathrm{Tp}$ 的确是可表态射.

接下来, 我们证明这个模型上有 Π-类型结构. 考虑上面的预层 $F(\Gamma) = \{(A, B) \mid A \in \mathrm{Tp}(\Gamma), B \in \mathrm{Tp}(\Gamma, A)\}$. 在集合中 $A \in \Gamma \to \mathsf{Set}_\kappa$, $B \in (\Gamma, A) \to \mathsf{Set}_\kappa$ 都是集合族. 我们想要给 (A, B) 赋予它们对应的 Π-类型, 在集合中这就是集合乘积

$$\mathsf{Pi}(A, B) = x \mapsto \prod_{a\in A(x)} B(x, a).$$

注意 $\mathsf{Pi}(A, B) \in \Gamma \to \mathsf{Set}_\kappa$ 也是集合族. 由于 κ 是强不可达基数, 这个集合乘积仍然在 Set_κ 中. 验证这的确给出了 Π-类型结构, 就是简单的集合操作了. 类似地, 我们可以给出 Σ-类型结构

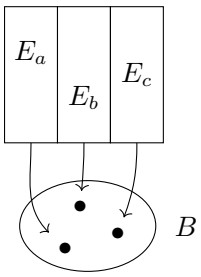
$$\mathsf{Sigma}(A, B) = x \mapsto \coprod_{a\in A(x)} B(x, a).$$

5.3 融贯问题

尽管从类型论的角度, 自然模型的定义的确很自然, 但是我们寻找模型时, 往往找到的东西会相差一个细节. 这个细节就是**融贯问题**. 在数学中, 往往难以直接定义“依值某物”. 例如假设 $b \in B$ 是拓扑空间中的一个点, 我们难以定义一族拓扑空间 $A(b)$ 如何随着 b 的变化而连续变化.

在类型论出现之前, 数学家就早已注意到这个问题. 一般而言, 数学家对此的解决方案是转而考虑如何定义 $E = \sum_{b\in B} A(b)$. 这就是一个普通的对象, 不存在依值的问题. 同时, 它显然应该带有投影函数 $E \to B$. 对 $A(b)$ 的研究都可以转而改为对 $p : E \to B$ 的性质的研

究. 假如我希望描述一族集合, 最直接的方法就是给定一个指标集 B, 然后对每个 $b \in B$, 指定一个集合 E_b. 换句话说就是有 $E_\bullet : B \to \mathsf{Set}$. 但是另一方面, 我也可以将所有的 E_b 聚合起来成为一个大集合 E, 再用一个函数 $p : E \to B$ 指出每个元素所属的指标 b 是哪一个. 这两种描述方式是等价的.



再如, 向量丛理应是依赖于流形上一点 $x \in M$ 的向量空间 $V(x)$, 但我们改将其定义为流形之间的连续映射 $p : E \to M$. 这种技术在代数几何中达到巅峰, 以 Grothendieck 的相对视角 (relative point of view) 为典型. 例如希望表达每个 $A(x)$ 都是紧空间, 则说 p 是紧合映射; 希望表达每个 $A(x)$ 都是仿射空间, 则说 p 是仿射映射.[3] 具体到类型论的语义中, 就是不考虑语境中的类型集合 $\mathrm{Tp}(\Gamma)$, 而是指定一系列特殊箭头 $\Delta \to \Gamma$ 的集合. 这里 Δ 在直观上对应 (Γ, A). 我们称这类映射为**形式丛** (display map).[4]

然而, 想要将这类定义变为类型论中的模型时, 就会遇到严重的问题. 在原本的写法 $A(x)$ 中, 若有函数 $f : Y \to X$, 那么可以直接代入 $A(f(y))$, 就得到依赖 $y \in Y$ 的依值对象. 这样, 先代入 $f : Y \to X$, 再代入 $g : Z \to Y$, 与直接一次性代入 $f \circ g$ 显然按定义是相同的, 都等于 $A(f(g(z)))$. 然而, 如果使用 $p : E \to X$ 的写法, 那么代入操作在转换之后就得到拉回 $p' : E \times_X Y \to Y$. 然而, 两次拉回与一次性拉回基本不会直接相等, 它们仅仅是同构. 这一现象在数学中很常见. 例如集合之间 $A \times (B \times C) \neq (A \times B) \times C$, 因为前者的元素形如 $(a, (b, c))$, 而后者形如 $((a, b), c)$, 显然不相等. 一个简单的想法是我们想办法商去一个等价关系, 使得它们相等. 这也是不可取的. 例如 $A \times B$ 与 $B \times A$ 是同构的, 但是如果我们将它们视作相同, 即令 $(a, b) = (b, a)$, 那么当 $A = B = \mathbb{R}$ 时, 就有 $(1, 2) = (2, 1) \in \mathbb{R}^2$, 故 $1 = 2$, 矛盾.

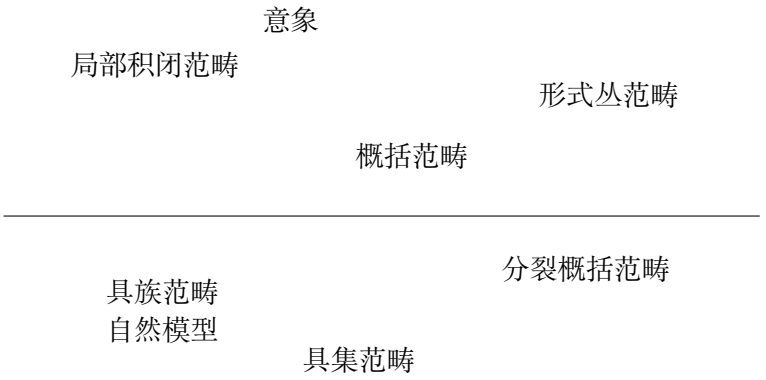
对此, 又能提出补救的办法. 第一个想法是试图证明在类型论中具体需要的相等关系里, 不会涉及上面提到的交换律这样有问题的情况. 再者, 可以给范畴添加一些与已有的对象同构的新对象, 这样不会改变范畴本身, 但是使得上面的商操作得以进行. 这些技巧统称为**融贯定理**.

由于这类问题在技术上比较复杂, 但是在直观上又应当总是成立, 在一般的研究中, 往往不愿在此事上花费过多笔墨. 因此往往有许多不同的办法定义类型论的模型, 有一些与数

[3] 一般来说, 在代数几何中也可以直接考虑前者, 即要求 $p^{-1}\{x\}$ 是仿射空间, 等等. 但是这样往往无法得到正确的定义, 因为这样仅仅要求每个点处 $A(x)$ 都满足条件, 并没有要求不同点之间要求的融洽性.

[4] 这里不直译英文术语, 而是翻译一个不常见的等价说法 (formal bundle) 作为中文术语. 这里的 display 有特殊含义, 类似“依值 (dependent)” 但用法又不全相同. 若读者有更好的中文翻译, 尽可提出.

学中遇到的对象比较接近, 因此便于构造具体的模型, 但是不满足上面提到的严格等式; 另一些与类型论中的语法比较接近, 因此便于证明与类型论的关系, 但是自然的数学对象往往不构成这种模型. 这种情况可以画出图表, 从横线的上方到下方需要解决一个融贯问题:



5.4 范畴语义的历史

- 1984 年, Seely [54] 首次具体写出了用丛表示依值类型, 拉回表示代换的思想. 这隐含了融贯问题, 稍后会介绍.
- 1986 年, Cartmell [16] 提出了具集范畴 (category with attributes) 及有语境性的变体.
- 1993 年, Bart Jacobs [37] 提出了概括范畴, 这统一了之前的诸多概念.
- Curien [23] 与 Hofmann [35] 讨论了融贯问题, 并给出了解决方案.
- 1995 年, Dybjer [26] 为了在类型论内研究类型论, 提出了具族范畴 (category with families) 的概念.
- 2014 年, Clairambault [20] 给出了融贯问题的完整答案, 这在 [24] 中有总结.
- 2015 年, 由 Voevodsky 的相关工作启发了融贯问题的新解决方案 [44], 这强调了宇宙的重要性.
- 2018 年, 具族范畴被 Awodey [3] 和 Fiore 各自独立重新表述为自然模型.
- 2019 年, 上村太一 [60] 提出了通用的框架, 给出了一大类类型论的语法与语义的关系.

5.5 意象与内语言

(less material)
list some concrete biequivalences
谓词, 子对象与代数性的破坏

第六章　同伦类型论

6.1　K 原理的独立性

我们之前已经看到相等类型的归纳子, 也就是 J 原理. 它直觉上大致说的是 “每个 $a = b$ 类型都只有 refl 一个构造器” . 但是想要表达这个直觉, 还有一个看起来很自然的写法:

$$\prod_{a,b:A} \prod_{p,q:a=b} p = q,$$

或者写成

$$\prod_{a:A} \prod_{p:a=a} p = \mathsf{refl},$$

注意这里不能写 $p : a = b$, 因为这样它与 refl 的类型不同, 就不能表达相等了. 当然, 也可以用上面的 Σ-类型补救一下, 写成

$$\prod_{a,b:A} \prod_{p:a=b} (a, b, p) =_{\sum_{a,b:A} a=b} (a, a, \mathsf{refl}).$$

上面的三个命题都是等价的. 我们将它们统称为 **K 原理**[1]. 令人惊讶的是, 很长一段时间中并没有人能从 J 原理推出 K 原理. 在第四章介绍的集合模型中, K 原理显然是成立的, 因此加入 K 公理后类型论仍然无矛盾. 进而无法证伪 K 原理. 那么反过来, 是否能证明 K 呢?

在 Martin-Löf 类型论提出之后, K 原理是否能从 J 原理中推出, 是一个重大的未解问题. 这个问题直到 1998 年才被 Hofmann 等人解决 [36]. 而这一解决方案, 对认识类型论的内禀结构提供了重要的方向 —— 类型可以是群胚!

定义 6.1. 群胚包含以下结构: 给定一些元素 C, 对于每对元素 $x, y \in C$ 都取一个集合 $\hom(x, y)$, 并且有乘法运算 $a * b$ 为二元函数 $\hom(y, z) \times \hom(x, y) \to \hom(x, z)$. 每个元素 $x \in C$ 都有对应的单位元 $\mathsf{id}_x \in \hom(x, x)$, 乘法运算满足单位律与结合律. 每个元素 $a \in \hom(x, y)$ 都有逆元 $a^{-1} \in \hom(y, x)$.

读者可以很快验证, 每个群都是使得 $C = \{\star\}$ 为一元集的群胚. 而另一个极端的例子, 对于任何集合 C, 直接令 $\hom(x, x) = \{\mathsf{id}_x\}$ 为一元集, 其它 $\hom(x, y)$ 均为空集. 这则被称为**离散群胚**.

[1] 这个名字是因为其提出者 Thomas Streicher 顺承 J 原理的名字, 使用了下一个字母 K. [59]

还有一族更加有启发性的例子:

例 6.1. *任给一个拓扑空间 X, 令其点集为 C, $\hom(x,y)$ 为从点 x 到点 y 的道路的集合, 商去同伦关系, 乘法运算为道路拼接操作, 那么它构成一个群胚.*

容易看出这是基本群的推广. 因此我们可以将群胚看成是“允许取多个基点的群”. 另一个角度, 也可以把群胚看作是所有态射均可逆的范畴, 不过需要注意的是, 许多书中群胚的乘法运算记号的左右顺序一般与范畴中相反, 因为范畴中常常把态射理解为类似函数的东西, 因此函数复合 $(f\circ g)(x)=f(g(x))$ 是先作用者在右侧. 但是在群胚中往往将态射看作道路, 因此两条道路 $x \overset{p}{\to} y \overset{q}{\to} z$ 拼接自然是从左到右写为 $p*q$. 当然这仅仅是记号区别.

我们回过头来看类型论, 用 J 原理可以很容易证明, 每个类型上都自带一个群胚结构. 我们有 $\mathsf{refl} : x=x$ 为单位元, 对称性

$$\prod_{x,y:A} x=y\to y=x$$

与传递性

$$\prod_{x,y,z:A} x=y\to y=z\to x=z$$

之前也已经提到过.

因为这些全都只需要 J 原理, 因此我们可以试着构造类型论的一个模型, 使得每个类型被解释为一个群胚. 这之中乘积类型被解释为群胚的乘积, 等等. 最重要的是, 相等类型 $x=y$ 被解释为集合 $\hom(x,y)$ 上的离散群胚. 因为当前的模型中 $\hom(x,y)$ 只是一个集合, 没有其他结构. 最终可以验证, 这个模型满足 J 原理. 具体的理论在 6.3 节会介绍.

构造了这个模型之后, K 原理在这个模型中的解释是什么呢? 我们可以发现它对应“所有群胚都是离散群胚”这个命题. 显然是不成立的. 从而我们构造了 K 原理的反模型. 上面的讨论即证明了如下命题:

定理 6.1. *在只有 J 原理的 Martin-Löf 类型论中, 无法证明或者证伪 K 原理.*

6.2 同伦类型论

在上面的模型中, 每个类型都是群胚. 而由于群胚的 hom 集合只有集合结构, 取相等类型之后我们就得到了一个集合, 也就是离散群胚. 但是仅仅从语法上来看, 相等类型作为群胚也有可能是非平凡的. 换句话说, 尽管我们确定了 K 原理 (解释为“所有类型都是离散群胚”) 是不可证明的, 我们却马上有了高一层的问题, 也就是“所有类型上的相等类型都是离散群胚”. 写成语法, 我们可以对比一下. K 原理是

$$\prod_{a,b:A}\prod_{p,q:a=b} p=q.$$

而这里说的高一层的 K 原理就是

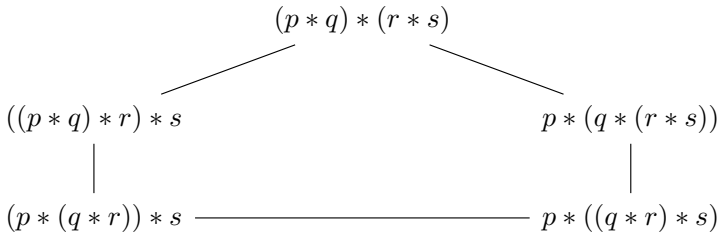
$$\prod_{a,b:A}\prod_{p,q:a=b}\prod_{\alpha,\beta:p=q}\alpha=\beta.$$

很明显这可以一直继续下去. 我们不妨做个编号, K_{-1} 原理就是

$$\prod_{a,b:A}a=b,$$

K_0 原理就是 K 原理, 而 K_1 原理就是上面说的高一层的 K 原理. 以此类推有 K_n 原理.

我们已经知道用群胚模型可以得到 K_0 原理不可证明. 那么以此类推, 是否构造一个 hom 不是集合而是群胚的 "2-群胚", 就能得到 K_1 原理不可证明了呢? 答案是肯定的. 这引出了 n-群胚乃至 ∞-群胚的概念. 具体的定义虽然简单但是细节繁杂, 这里不再给出. 但是相信读者已经有了大致的图像. ∞-群胚是同伦论研究中用到的概念. 事实上, 拓扑空间的道路就可以看作 ∞-群胚: 我们取所有的道路, 不商去同伦. 那么所有道路 (也就是 1-道路) 并不是严格构成群胚, 因为结合律等等式只在道路的同伦意义下成立. 换句话说结合律是道路的道路, 也就是 2-道路. 而进一步, 2-道路之间也有等式. 考虑五条道路之间的 2-道路:



在拓扑空间中可以证明这五条 2-道路复合形成的 2-道路到平凡的 2-道路一定有一条 3-道路, 这些 3-道路之间又会有更高的道路 (本书的封面图就是一些 3-道路之间的 4-道路示意图), 以此类推到任意高的 n-道路. 对于每个 n, n-道路商去 $(n+1)$-道路的同伦, 就得到了 π_n 同伦群. 因此提到 ∞-群胚时, 读者可以大致理解为拓扑空间. 为了强调 ∞-群胚的几何属性 (同时也因为它名字有些长), 提出了一个新的术语, 称为**生象** (anima).

由此, 我们对类型的崭新理解就是

类型是空间.

在 [62] 中对此有详细的论述. 类型不是命题! 我们之前的口号 "类型是命题" 并不是最好的描述, 事实上只有一部分类型应该作为命题理解. 具体来说, 恰好是那些满足 K_{-1} 原理的类型. 这条原理实际上非常符合我们的观念. 譬如考虑偶数集 $\{x\in\mathbb{Z}\mid x\equiv 0\pmod 2\}$. 我们不会认为 x 是偶数的两种不同思路的证明会给出两个不同的偶数. 这两个证明作为数学对象比较的时候永远是相等的. 换句话说, 同一个命题的任何两个证明 (如果存在的话) 一定是相等的. 因此我们可以做一个定义:

定义 6.2. 称一个类型 A 为**命题**, 当且仅当可以证明

$$\prod_{x,y:A} x = y.$$

由此, 我们之前定义的排中律 $\prod_{A:\mathsf{Type}} A + \neg A$ 就不再适合了. 取而代之的是

$$\prod_{A:\mathsf{Type}} \mathsf{isProp}(A) \to (A + \neg A),$$

其中 $\mathsf{isProp}(A)$ 是 "A 为命题" 的缩写. 为了强调其不同, 这可以称作**命题排中律**, 原先的版本则是**类型排中律**.

如果满足 K_0 原理, 这个类型应该称作什么呢? 由上面群胚模型的启发, 这些应该称为**集合**: 每个集合都可以看作一个离散群胚 (或者离散拓扑空间). 我们可以证明自然数类型是一个集合, K_0 原理说的就是 "所有类型都是集合". 进一步, 满足 K_1 原理的被称为群胚; 满足 K_n 的则被称为 n-群胚.

6.2.1 泛等公理

认识到这些性质之后, 自然有了一些将类型论用于研究同伦论的努力. Voevodsky 在 2006 年已经提出了与泛等公理相关的性质, 但是直到 2009 年才意识到这可以直接作为公理加入 Martin-Löf 类型论中.

这条公理来源于 ∞-群胚模型的启发, 但是我们这里只讲述它的形式以及推论.

首先, 我们有某个函数 f 是**等价**的定义. 读者可以参考 [28] 中的定义细节. 对于每个 f, $\mathsf{isEquiv}(f)$ 都是命题. 将类型视作空间时, 这个定义就代表同伦等价. 在集合上, 同伦等价就是双射. 显然我们应当有 $\mathsf{isEquiv}(\mathsf{id})$. 我们定义 $X \simeq Y$ 为 $\sum_{f:X\to Y} \mathsf{isEquiv}(f)$, 即两个类型之间所有的等价构成的类型.

显然有 $(X = Y) \to (X \simeq Y)$. **泛等公理**说的是, 这个函数本身是一个等价. 一个立即的推论就是

$$(X = Y) \simeq (X \simeq Y).$$

直观上来看, 它说的是 "等价的类型都相等". 事实上它有非常多深刻的推论.

推论 6.1 (命题外延)**.** *等价的命题都相等: 如果有两个命题满足 $p \iff q$, 那么 $p = q$.*

推论 6.2 (函数外延)**.** *如果两个函数逐点相等, 那么两个函数相等:*

$$\left[\prod_{x:A} f(x) = g(x)\right] \to f = g.$$

这解决了我们在 Martin-Löf 类型论中留下的问题. 但是我们还有更加奇妙的结论:

推论 6.3. *如果两个群同构, 那么它们相等.*[2]

[2] 更准确的说, 两个群 G, H 之间的不同同构与类型 $G = H$ 中的元素一一对应.

注意我们的泛等公理中完全没有提到群同构, 这是自然导出的定理! 这也非常符合我们的数学实践: 长方形的对称群、模 8 奇数的乘法群, 与 $\mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$ 是完全等同, 随时可以互相替换的. 这也使得有了泛等公理之后, 在类型论里的形式化数学比起集合论等等更加接近我们实际使用的非形式化数学语言. 这也与**数学结构主义**相合. 大致来说, 数学结构主义关心数学对象在其结构中的位置, 而不关心对象内在的属性. 例如 Klein 群并不是四个矩阵, 四个对称变换或者其他什么东西组成的, 而是四个抽象的元素. 除了这些元素上的群结构之外, 它们没有别的属性. [4] 泛等公理可以看成 "一切操作都必须尊重同构".

推论 6.4. *类型排中律不成立.*

这是因为类型排中律 $\prod_{A:\mathsf{Type}} A + \neg A$ 从每个非空类型中选出了一个元素. 这个选择不可能尊重同构关系, 譬如有两个元素的类型 $\mathbf{2}$ 有一个自同构, 交换这两个元素. 那么类型排中律的选择函数在这个同构意义下显然不可能不变, 从而与泛等公理相矛盾.

不过这个性质常常带来一些误解, 即认为泛等数学基础无法与排中律相容. 注意*命题排中律与泛等公理仍然是相容的*, 并且由于我们已经加入了泛等公理, 因此将排中律再作为公理加入类型论也不会破坏更多性质. 同样的, 选择公理也与泛等公理相容, 可以加入同伦类型论中. 不过需要承认, 当前的同伦类型论研究的绝大部分没有假设排中律, 这是应当改变的.

泛等数学基础的出现, 为那些非泛等的类型论的发展也提供了重要的指导. (... 举两个例子?)

6.2.2 高阶归纳类型

我们之前已经见过了许多归纳类型, 如自然数, 二叉树等等. 而在认识到类型的 ∞-群胚解释后, 自然就可以对归纳类型做出推广. 2011 年 Oberwolfach 数学研究所的一次研讨会上, Voevodsky 等人提出了许多重要的研究问题. 高阶归纳类型的想法就是在这里首次出现的.

定义 6.3. **圆**类型有两个构造器, $\mathsf{base} : \mathbb{S}^1$ 与 $\mathsf{loop} : \mathsf{base} = \mathsf{base}$.

这个类型中, loop 就是道路的自由生成元. 这与 CW 复形的圆定义非常相似: 有一个点 base, 将一条线段 loop 的两个端点都粘在这个点上. 利用泛等公理, 我们可以证明各种类似于同伦论中圆的性质, 如它的基本群是 $\mathbb{Z}$, 等等. Guillaume Brunerie [14] 在类型论中证明了 $\pi_4(\mathbb{S}^3) = \mathbb{Z}/2\mathbb{Z}$. 由此看来, 高阶归纳类型真正使得类型论变成了可以研究同伦论的工具. 除了同伦之外, **商类型**也可以作为高阶归纳类型的特殊情况存在. 因此我们完全解决了 Martin-Löf 类型论的诸多不便之处.

我们将这些方向的类型论的各种变体统称为**同伦类型论**. 这个词没有严格定义, 但是一般会包含泛等公理与一定量的高阶归纳类型.

6.2.3 泛等数学基础

在 Oberwolfach 会议之后, 参会者搭建了同伦类型论的网站与博客, 由此进行同伦类型论的推广. 同伦类型论理论上已经可以作为数学基础使用. 正如上面所说, 它与通常的数学语言更加接近. 此时已经有初步的形式化努力.

2012–2013 年是泛等数学基础最重要的一年. 由 Princeton 高等研究院发起了 “泛等数学基础特别年”, 邀请了拓扑学、计算机科学、范畴论、逻辑学等等领域的数学家参加. 其中由 Peter Aczel 提议, 尝试使用同伦类型论书写非形式化的数学. 这次尝试很快获得了初步成功, 最终写出了《同伦类型论: 泛等数学基础》 [62] 这本书. 它从零出发, 整理了泛等数学基础中的结论, 从同伦类型论的角度重新叙述了集合论与范畴论, 并且给出了实数的定义与主要性质的证明. 一方面, 写出实数理论证明了泛等数学基础在实操上可以有效地书写一般数学; 另一方面, 这开创了**综合同伦论** (synthetic homotopy theory) 这门学科. 正如 Euclid 的《几何原本》中并没有将平面定义为 $\mathbb{R}^2$, 而是直接从公理出发推导结论, 综合同伦论直接由类型论的基本规则刻画同伦关系, 而无需依赖拓扑空间的区间 $[0,1]$ 乃至 CW 复形等等. 书中复现了许多传统同伦论中的结论.

阅读这本书不需要类型论基础, 并且内容是开源的. 读者可以自行下载阅读.

6.3 同伦类型论的语义

我们承接 6.1 节, 详细描述 [36] 中的群胚语义. 我们要以群胚 (定义 6.1) 解释类型论中的语境, 那么依照之前的讨论, 需要定义 “依值群胚” 的概念解释语境中的类型. 这就是将集合族的定义推广到 “群胚族”.

定义 6.4. 给定群胚 Γ, **依值群胚** A 包含以下资料:

- 对于 Γ 中的每个元素 x, 都有集合 A_x.
- 对于 Γ 中的道路 $p : \hom(x,y)$ 与 $\alpha \in A_x$, $\beta \in A_y$, 都有集合 $\hom_p(\alpha,\beta)$, 表示 A 中的依值道路.
- 有单位元 $\mathrm{id} \in \hom_{\mathrm{id}}(\alpha,\alpha)$ 与道路拼接 $\hom_p(\beta,\gamma) \times \hom_q(\alpha,\beta) \to \hom_{p*q}(\alpha,\gamma)$, 满足单位律与结合律.

假如 $\Gamma = 1$ 是平凡群胚, 则依值群胚的概念就退化为普通的群胚. 如果有群胚的同态 $\sigma : \Delta \to \Gamma$, 那么给定 Γ 上的依值群胚 A, 就能得到 Δ 上的依值群胚 B. 对 Δ 里的元素 x, 定义 $B_x = A_{\sigma(x)}$, 等等. 这对应于类型论中的代换操作, 因此我们将它记作 $A[\sigma]$ 强调这一联系.

6.4 立方类型论

泛等公理有很多好的结论, 但是它也有一个问题: 它是一条公理. 而前面已经多次强调过, 如果往类型论中强行加入公理, 就会破坏它的性质. 这些不好的性质会对形式化数学带来一些困难 —— 尽管不会造成本质性的阻碍, 但是仍然会增加许多工作量. 例如可以在同伦类型论中定义一个自然数, 它不等价于任何一个具体的自然数, 但是也无法继续化简. (我们仍然可以用相等类型证明它等于某个具体的自然数, 譬如上文提到的 Brunerie 就定义了一个自然数为 $\pi_4(\mathbb{S}^3)$ 的阶数, 并且证明了这个自然数等于 2. 但是在判值相等关系下它并不等价于 2.) 有没有办法设计一个类型论, 使得泛等公理成为一个按照类型规则可以证明的定理呢? 答案是肯定的.

这里, 我们需要设计一套类型规则, 这是类型论的纯语法操作. 但是, 因为语法与语义的研究是一体两面的, 对同伦类型论的语义研究能启发新类型规则的设计. 这正是立方类型论的来源.

在类型论开始使用 ∞-群胚之前, 拓扑学家已经对此有了大量的研究. 拓扑学中已经有利用单形 (simplex)[3] 刻画拓扑空间的传统. 我们可以用许多单形拼接成一个拓扑空间, 也可以直接抽象地考虑单形, 将它们看作类似图论或者组合学的对象处理. 当然, 也有一些使用其他基础形状的研究, 如使用 n 维立方体, 或者使用树形等等. 事实上, Daniel Kan 最早的工作就是使用立方体而不是单形. 然而, 这些形状在后续的研究中发现了它们在一些技术细节上的困难. 因此在拓扑学研究中 ∞-群胚最常用的还是使用单形的定义.

但是, 对于类型论的研究来说, 基于单形的理论有致命的问题: 它大量使用了排中律和选择公理等非构造性的方法. 2015 年, Bezem 与 Coquand [8] 指出当时的理论中非构造性是无法消除的.[4] 进一步的研究发现, 基于立方体的理论的技术困难是可以克服的, 并且我们能够建构一套完全构造性的理论. 在 2013 年, Marc Bezem, Thierry Coquand, Simon Huber 三人提出用立方集合构建同伦类型论的模型. 在 2017 年他们正式给出了相关证明 [9]. 这套模型由他们的姓氏首字母称为 BCH 模型. 根据模型启发了类型论的规则设计, 得到的就是**立方类型论**.

然而, 这个模型虽然支持泛等公理, 但是由于一些技术原因[5]不能支持高阶归纳类型. 在 2017 年, Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg 在论文《立方类型论: 泛等公理的构造性模型》 [2] 中四人提出了一个新的模型. 其中立方体的结构由 de Morgan 代数表示. 根据这个模型 (缩写为 CCHM 模型), 他们构造了一套新的类型论语法. 这套类型论被称为 **de Morgan 立方类型论**. 2019 年, Carlo Angiuli 等六人提出用区间的 Descartes 乘积生成的立方体作为基础, 得到了 ABCFHL 模型与**积立方类型论**.

2021 年, Sterling 等人发展了范畴论工具, 证明了立方类型论的典范性. 在他的学位论

[3] 单形就是三角形、四面体等的 n 维推广.

[4] 后续研究 [34] 中发现可以修改理论去除非构造性. 这与 Bezem 等人的工作并不矛盾, 但是这一点的解释超出了本文的范畴.

[5] 他们选择的立方体只有最基础的结构, 是由一维的线段直接 (在对称张量积下) 生成的, 因此没有对角线.

文 [56] 中, Sterling 进一步描述了这些范畴论工具, 并且指出语法与语义研究之间的关系. 对类型论的范畴语义感兴趣的读者也可以阅读这篇论文.

立方类型论不完全是同伦类型论的子学科. Sterling 等人提出了 XTT 类型论 [58], 属于立方类型论, 但是不支持泛等公理, 属于集合 (即满足 K 原理) 层面的理论.

6.4.1 立方类型论简介

立方类型论中, 引入了一个类型 $\mathbb{I}$ 表示同伦论的区间. 它有两个端点 $0, 1$. 一个变量 $i : \mathbb{I}$ 则表示区间上的一个点. 如果有多个变量 $i, j, k : \mathbb{I}$, 则它们共同表示立方体中的某个坐标. 在同伦类型论中, 相等类型的直观是两点之间道路的空间. 在立方类型论中, 如果有一条道路 $p : x =_A y$, 那么就允许我们取出这条道路上的某一点 $p(i) : A$. 因此, $\lambda i.p(i)$ 就是某个函数 $\mathbb{I} \to A$, 而道路空间就可以看成是端点给定的函数空间.

这样, 我们立刻就可以证明一些定理.

定理 6.2. *给定函数 $f : A \to B$, 如果有两个元素 $x = y$, 那么 $f(x) = f(y)$.*

证明. 我们有 $p : x = y$. 我们需要证明 $f(x) = f(y)$. 由于道路空间可以看成端点固定的函数空间, 我们只需要给出一个函数 $q : \mathbb{I} \to B$ 使得两个端点 $q(0) = f(x), q(1) = f(y)$ 即可. 由于 $p(i)$ 是道路上的一个点, 有 $p(i) : A$. 因此 $f(p(i)) : B$. 我们定义 $q(i) = f(p(i))$, 验证端点 $q(0) = f(p(0)) = f(x), q(1) = f(p(1)) = f(y)$ 满足条件. □

定理 6.3. *如果两个函数逐点相等, 那么它们相等.*

证明. 我们有 $p : \prod_{x:A} f(x) =_B g(x)$. 我们需要证明 $f =_{A \to B} g$. 由于道路空间可以看成端点固定的函数空间, 我们只需要给出一个函数 $q : \mathbb{I} \to (A \to B)$, 使得两个端点 $q(0) = f, q(1) = g$ 即可. 我们定义 $q(i)(x) = p(x)(i)$. 由于 $p(x) : f(x) =_B g(x)$ 是一条道路, 因此 $p(x)(i)$ 就是这条道路上的一个点, 并且其端点符合要求. □

可以看到, 将相等类型使用道路空间的方式定义, 使得证明更加简洁直观. 同时函数外延性在用泛等公理证明时十分复杂, 而使用立方类型论则是最基础的结论.

希望继续了解立方类型论的读者可以参考这篇文章 [21].

第七章　展望

在最近十几年来, 类型论的研究有了大跨步的发展. 在理论方面, 我们对类型论的本质结构有了更加深刻的认识. 在实践层面, 有了许多重要的形式化与非形式化的工作. 不过, 仍然有许多问题有待回答.

利用基于类型论的定理证明检查器形式化了许多著名的成果, 这足够证明类型论形式化数学的潜力. 类型论的核心部分往往非常简洁, 使得证明其性质相对方便, 但是实际使用时会使得写出的证明无比繁琐. 在这个方向, 我们仍然需要进一步改进定理证明的**前端**, 即与使用者直接交互的部分. 例如, 需要让定理证明的体验与数学家的习惯相契合, 能让用户使用各个数学领域常用的符号和缩写, 组织证明时能流畅使用各种常见的证明技巧, 平凡的计算细节可以自动化处理, 等等. 2023 年发表的 Sebastian Ullrich [61] 的学位论文可以参考.

与此相关, 基于神经网络的人工智能在定理证明中也有应用的潜力, 例如在繁杂的定理库中快速筛选出与当前目标相关的部分. 由于已经有成熟的组件检查输出的证明是否正确, 这就避开了最近出现的大语言模型在关于事实准确性方面的短板. 有不少工作将神经网络整合到定理自动证明体系中, 如 LeanDojo [65] 等.

有一类类型论在近年来得到了许多关注, 它们称为**模态类型论**. 这起源于逻辑学中的模态逻辑. 例如可以用模态 $\Diamond p$ 表示 “可能 p” , 进而研究关于可能性与必然性的逻辑. 模态逻辑可用于研究可能性、时态、知识、道德等等在哲学与逻辑学中有意义的话题.[1] 而在数学方面, 许多重要的概念都可以统一到类型论中的模态框架下, 同时一些综合数学系统中也会遇到适合用模态公理化的对象. 模态类型论可能可以将各种优势各异的类型系统统一到同一个类型论中.

在同伦类型论方面, 我们知道简单类型论与积闭范畴之间有等价关系; 而 [20] 中证明了带有外延相等类型的 Martin-Löf 类型论与局部积闭范畴之间有等价关系. 那么内涵 Martin-Löf 类型论对应什么呢? 一个合理的猜想是它与 $(\infty,1)$-局部积闭范畴等价. 但是证明此事困难重重. 第一在于我们尚未了解清楚 Martin-Löf 类型论构成怎样的 ∞-范畴, 或许这需要合适的 ∞-自然模型的概念. 其次是对于二者之间的等价也尚不知道如何构造, 例如从 $(\infty,1)$-局部积闭范畴得到类型论, 目前只知道局部可表范畴的情况. 而再进一步, 同伦类型论对应什么呢? 应当是某种初等 $(\infty,1)$-意象的概念. 但是这连定义都尚未明确, 更不用提证

[1] 例如考虑这个有趣的悖论: 我认识张三, 我不认识面前这个戴面具的人, 因此张三和面具人不相同.

明了.

立方类型论解决了同伦类型论的计算问题. 例如 Brunerie [14] 证明了存在自然数 n 使得 $\pi_4(\mathbb{S}^3) = \mathbb{Z}/n\mathbb{Z}$. 因为立方类型论中任何一个自然数表达式都判值相等于某个具体的自然数 $\mathsf{succ}(\mathsf{succ}(\ldots(\mathsf{zero})))$, 因此理论上我们可以直接在立方类型论中化简, 最后应当得到 $(2, p)$, 其中 p 是同构的证明. 这样就不再需要额外证明这个自然数 $n = 2$. 然而, 因为所需的计算资源与时间过长, 目前还没有电脑成功完成这个计算. 目前有一些旨在改进立方类型论的计算效率的工作, 但是也可能需要设计全新的类型论才能解决这个问题.

关于泛等数学基础, [62] 已经发展了许多半形式化的基于泛等数学基础的理论. 我们需要更多的在泛等基础中半形式化或者非形式化的数学工作. 并且这些工作不应当限制于类型论的话题中, 而是应当更加积极地探索泛等数学基础能为更广泛的领域带来怎样的贡献. 非形式化的数学中, 可以不提及或者模糊处理类型的概念. 然而, 同伦类型论, 包括立方类型论, 是目前泛等数学基础唯一的严格形式化方法. 泛等数学基础是否只能在类型论中实现呢? 这个问题尚待解答.

参考文献

[1] Samson Abramsky and Achim Jung. “Domain theory”. In: *Handbook of logic in computer science (vol. 3) semantic structures*. 1995.

[2] Carlo Angiuli et al. “Syntax and models of Cartesian cubical type theory”. In: *Mathematical Structures in Computer Science* 31 (2021), pp. 424–468.

[3] Steve Awodey. “Natural models of homotopy type theory”. In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 241–286. DOI: 10.1017/S0960129516000268.

[4] Steve Awodey. “Structuralism, Invariance, and Univalence”. In: *Philosophia Mathematica* 22.1 (Oct. 2013), pp. 1–11. URL: https://doi.org/10.1093/philmat/nkt030.

[5] Henk P Barendregt. “Lambda calculi with types”. In: Oxford: Clarendon Press, 1992. URL: http://www.cosc.brocku.ca/~mwinter/Courses/5P05/HBKJ.pdf.

[6] Andrej Bauer. “Five stages of accepting constructive mathematics”. In: *Bulletin of the American Mathematical Society* 54 (2016), pp. 481–498.

[7] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer Berlin, Heidelberg, 1985.

[8] Marc Bezem and Thierry Coquand. “A Kripke model for simplicial sets”. In: *Theor. Comput. Sci.* 574 (2015), pp. 86–91.

[9] Marc Bezem, Thierry Coquand, and Simon Huber. *The univalence axiom in cubical sets*. 2017. URL: https://arxiv.org/abs/1710.10941.

[10] Ingo Blechschmidt. *Using the internal language of toposes in algebraic geometry*. 2021. URL: https://arxiv.org/abs/2111.03685.

[11] Simon Pierre Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. “The next 700 syntactical models of type theory”. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (2017).

[12] Douglas Bridges, Erik Palmgren, and Hajime Ishihara. “Constructive Mathematics”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[13] Luitzen Egbertus Jan Brouwer. *Brouwer's Cambridge lectures on intuitionism*. Ed. by Dirk van Dalen. Cambridge University Press, 1981.

[14] Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory*. 2016. URL: https://arxiv.org/abs/1606.05916.

[15] J. Posy Carl. “Brouwer versus Hilbert: 1907–1928”. In: *Science in Context* 11 (1998), pp. 291–325.

[16] John Cartmell. “Generalised algebraic theories and contextual categories”. In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243. ISSN: 0168-0072. DOI: https://doi.org/10.1016/0168-0072(86)90053-9.

[17] Robert Cartwright, Rebecca Parsons, and Moez AbdelGawad. *Domain Theory: An Introduction*. 2016. URL: https://arxiv.org/abs/1605.05858.

[18] Alonzo Church. “A Set of Postulates for the Foundation of Logic”. In: *Annals of Mathematics* 33 (1932), p. 346.

[19] Alonzo Church. “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58 (1936), p. 345.

[20] Pierre Clairambault and Peter Dybjer. “The biequivalence of locally cartesian closed categories and Martin-Löf type theories”. In: *Mathematical Structures in Computer Science* 24.6 (Apr. 2014).

[21] 1Lab contributors. *1Lab: A formalised, cross-linked reference resource for mathematics done in Homotopy Type Theory*. https://1lab.dev. 2023.

[22] Thierry Coquand. “Type Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[23] Pierre–Louis Curien. “Substitution up to Isomorphism”. In: *Fundamenta Informaticae* 19.1–2 (Sept. 1993), pp. 51–85. ISSN: 0169-2968.

[24] Pierre-Louis Curien, Richard Garner, and Martin Hofmann. “Revisiting the categorical interpretation of dependent type theory”. In: *Theoretical Computer Science* 546 (2014), pp. 99–119. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2014.03.003.

[25] H. B. Curry. “Functionality in Combinatory Logic”. In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590. URL: https://www.pnas.org/doi/abs/10.1073/pnas.20.11.584.

[26] Peter Dybjer. “Internal Type Theory”. In: *Types for Proofs and Programs*. 1995.

[27] Manuel Eberl and Lawrence C. Paulson. “The Prime Number Theorem”. In: *Archive of Formal Proofs* (Sept. 2018). https://isa-afp.org/entries/Prime_Number_Theorem.html, Formal proof development.

[28] Martín Hötzel Escardó. *A self-contained, brief and complete formulation of Voevodsky's Univalence Axiom*. 2018. URL: https://arxiv.org/abs/1803.02294.

[29] William M. Farmer. “The seven virtues of simple type theory”. In: *Journal of Applied Logic* 6.3 (2008), pp. 267–286. URL: https://www.sciencedirect.com/science/article/pii/S157086830700081X.

[30] Solomon Feferman. “Relationships between Constructive, Predicative and Classical Systems of Analysis”. 2000. URL: http://math.stanford.edu/~feferman/papers/relationships.pdf.

[31] Daniel P Friedman and David Thrane Christiansen. *The Little Typer*. The MIT Press. London, England: MIT Press, Sept. 2018.

[32] Jean-Yves Girard. “Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure”. PhD thesis. Université Paris VII, 1972. URL: http://prl.ccs.neu.edu/img/g-thesis-1972.pdf.

[33] Georges Gonthier et al. “Formal proof–the four-color theorem”. In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[34] Simon Henry. *A constructive account of the Kan-Quillen model structure and of Kan's Ex^{∞} functor*. 2019. URL: https://arxiv.org/abs/1905.06160.

[35] Martin Hofmann. “On the interpretation of type theory in locally cartesian closed categories”. In: *Computer Science Logic*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 427–441.

[36] Martin Hofmann and Thomas Streicher. “The groupoid interpretation of type theory”. In: *Twenty Five Years of Constructive Type Theory*. Oxford University Press, Oct. 1998. URL: https://doi.org/10.1093/oso/9780198501275.003.0008.

[37] Bart Jacobs. “Comprehension Categories and the Semantics of Type Dependency”. In: *Theor. Comput. Sci.* 107 (1993), pp. 169–207.

[38] Bart Jacobs. “On cubism”. In: *Journal of Functional Programming* 6.3 (1996), pp. 379–392.

[39] Peter T. Johnstone. *Sketches of an Elephant: A topos theory compendium*. Vol. 1. Oxford University Press, 2003.

[40] F. William Lawvere. “Functorial Semantics of Algebraic Theories”. In: *Proceedings of the National Academy of Sciences of the United States of America* 50 (1963), pp. 869–872.

[41] F. William Lawvere. “Tools for the Advancement of Objective Logic: Closed Categories and Toposes”. In: *The Logical Foundations of Cognition*. Ed. by John Macnamara and Gonzalo E. Reyes. Oxford University Press USA, 1994, pp. 43–56.

[42] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52 (2009), pp. 107–115.

[43] Ralph Loader. “Notes on Simply Typed Lambda Calculus”. In: (1998). URL: http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/ECS-LFCS-98-381.pdf.

[44] Peter Lefanu Lumsdaine and Michael A. Warren. “The Local Universes Model”. In: *ACM Transactions on Computational Logic* 16.3 (July 2015), pp. 1–31.

[45] Saunders Maclane. “Categories for the working mathematician”. In: 1971.

[46] Per Martin-Löf. “Constructive mathematics and computer programming”. In: *Studies in logic and the foundations of mathematics* 104 (1984), pp. 167–184. URL: https://www.cs.tufts.edu/~nr/cs257/archive/per-martin-lof/constructive-math.pdf.

[47] R.P. Nederpelt et al. “Selected papers on Automath”. In: *Studies in logic and the foundations of mathematics* 133 (1994), pp. 299–301.

[48] Clive Newstead. “Algebraic models of dependent type theory”. PhD thesis. Carnegie Mellon University, 2018. URL: https://arxiv.org/abs/2103.06155.

[49] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, Jan. 2002. DOI: 10.1007/3-540-45949-9.

[50] John C. Reynolds. “Polymorphism is not Set-Theoretic”. In: *Semantics of Data Types*. 1984.

[51] John C. Reynolds. “Towards a theory of type structure”. In: *Symposium on Programming*. 1974.

[52] Peter Scholze. *Liquid tensor experiment*. https://xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment/. 2020.

[53] Moses Schönfinkel. "Über die Bausteine der mathematischen Logik". In: *Mathematische Annalen* 92 (1924), pp. 305–316.

[54] Robert A. G. Seely. "Locally cartesian closed categories and type theory". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95 (1984), pp. 33–48.

[55] Peter Smith. *Category Theory I: Notes towards a gentle introduction*. Logic Matters, Cambridge, 2023. URL: https://www.logicmatters.net/resources/pdfs/SmithCat-I.pdf.

[56] Jonathan Sterling. "First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory". Version 1.1, revised May 2022. PhD thesis. Carnegie Mellon University, 2021. DOI: 10.5281/zenodo.6990769.

[57] Jonathan Sterling. *Towards a geometry for syntax*. 2023. arXiv: 2307.09497 [cs.LO].

[58] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "Cubical Syntax for Reflection-Free Extensional Equality". In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 31:1–31:25. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10538.

[59] Thomas Streicher. "Investigations into Intensional Type Theory". Habilitation thesis. Ludwig Maximilian University of Munich, 2019. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

[60] Taichi Uemura. *A General Framework for the Semantics of Type Theory*. 2019. URL: https://arxiv.org/abs/1904.04097.

[61] Sebastian Andreas Ullrich. "An Extensible Theorem Proving Frontend". PhD thesis. Karlsruher Institut für Technologie (KIT), 2023. 243 pp. DOI: 10.5445/IR/1000161074.

[62] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[63] Philip Wadler. "Recursive Types for Free!" Unpublished manuscript. 1990. URL: https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt.

[64] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. *Eliminating Reflection from Type Theory*. https://github.com/TheoWinterhalter/ett-to-itt. 2018.

[65] Kaiyu Yang et al. *LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*. 2023. arXiv: 2306.15626 [cs.LG].