STLC 开始的类型论入门

∞-type Café 暑期学校讲座

Mar 06, 2024

Alias Qli
alias@qliphoth.tech

目录

全文使用香蕉空间风格的术语翻译和书写规范. 例如非中文的人名统一保留源语言中的拼写, 简短的字母和符号也不翻译. 标点符号全部使用半角. 作者水平有限, 讲义也是在仓促之间写就, 若读者发现本文有不连贯的地方, 遗漏或者错误, 请联系作者.

1. 前置知识

高中数学水平.

1.1. 类型论, 与一个类型论

就像集合论有两重含义一样 — 狭义的集合论可以指朴素集合论, ZFC 集合论, 质料集合论等, 而广义的集合论指研究全部这些 (狭义) 集合论的学科, 或者前面这些 (狭义) 集合论的统称 — 类型论也有两重含义: 狭义上, 类型论可以指今天要讲的 STLC, 或者 MLTT, HoTT 等, 而广义上则是这

些类型论的统称.[1]本课程的主要内容, 就是借介绍 STLC 这一**个类型论**, 使读者熟悉**类型论**中的某些重要概念.

1.2. 元变量

我们在讨论一个名叫 STLC 的**形式语言**[2]. 这门语言中的东西, 比如 $\lambda x.\lambda y.x : \mathrm{Ans} \to \mathrm{Ans} \to \mathrm{Ans}$, 都是以**字符串**的形式表示的. 根据我们的需求, 我们给这些字符串的特定部分以特定的名字, 比如将 x 称为变量, $\lambda x.\lambda y.x$ 称为项, 将 $\mathrm{Ans} \to \mathrm{Ans} \to \mathrm{Ans}$ 称为类型, 或者自己构造一个字符串 $\emptyset, x : \mathrm{Ans}, y : \mathrm{Ans}$ 称为语境. 它们都是指某些具有特定特征的字符串.

另一方面, 我们是在使用**自然语言**对 STLC 进行讨论, 这时候相对于 STLC 而言自然语言就称为**元语言**. 我们希望这个元语言中有一些变量, 它们能够取值于 STLC 中有特定特征的字符串. 例如, 说 A 是取值于 STLC 的类型的元变量, 就是说 A 取值于那些被称为类型的字符串, A 因此可以用来表示那个它所取值的字符串中记载的那个类型. 类似的, x 可以取值于那些被称为变量的字符串.

本文中常常将元变量当作 STLC 中对应的部分来使用, 不作明显的区分. 只是读者需要注意, 当本文中提到 "某类型 A" 等时, 含义是 "某个类型, 我们用元变量 A 表示它", 而不是 STLC 中有一个名为 A 的类型, 这点请务必区分清楚.

1.3. 符号约定

- 使用 A, B 等表示 STLC 的类型. 换句话说, A, B 是取值于 STLC 的类型的元变量, 下同.
- 使用 x, y, z 等表示 STLC 的变量. 这时候元语言和 STLC 使用了相同的变量记号, 但不至于造成麻烦.
- 使用 大写字母 M, N 等或小写字母 $a, b, c, ..., f, g$ 等表示 STLC 的项.
- 使用 Γ 等表示 STLC 的语境.
- 使用 $=$ 表示相等, $\equiv$ 表示判值相等 (judgemental equality), $:=$ 表示 "定义为".

2. λ 演算的动机

让我们暂且忘记集合论中对于函数的实现, 回到高中, 看看我们是怎么构造和使用函数的.

当我们构造函数时, 我们给出它的解析式:

$$f(x) := M$$

其中, M 是一个可以使用自变量 x 的项. 换句话说, $f(x)$ 创造了一个可以自由使用自变量 x 的语境. 只有在这个语境内, 变量 x 才是可以引用的.

当我们应用这个函数, 比如求值 $f(3)$ 时, 我们把 $x = 3$ 代入 M, 就得到 $f(3)$ 的值.

上面的描述虽然粗浅, 但是很好地描述了我们是如何如何**构造**和**使用**一个函数的, 而这正是类型论中一个类型最根本的两项特征. 不过, 在类型论中, 我们将函数表示为 λ 项. λ 演算使用语法 $\lambda x.M$ 表示 "参数为 x, 函数体为 M 的函数". 可以认为, 一个 λ 项就是一个没有名字的函数 (匿名函数). 有一些计算机方向的读者可能会对 "匿名函数" 这个名字比较熟悉: 事实上编程语言中的匿名函数就来源于 λ 演算.

[1]类型论分为 Curry 风格和 Church 风格两大类, 后者我们不涉及. 这里 (以及整个暑校中) , "类型论" 都只会指前者.

[2]计算机方向来的读者, 可以类比为编程语言.

3. 简单类型 λ 演算

简单类型 λ 演算 (Simply Typed Lambda Calculus, 后文常以 STLC 代称) 是指一种具有简单类型系统的类型论. [3]下面依次介绍 STLC 中的各个部分.

3.1. 相继式演算

一个类型论中, 决定哪些类型合法、一个项又具有什么类型的规则称为**类型规则 (typing rules)**, 它们常常以相继式演算 (sequent calculus) 的形式给出. 学习类型论, 一定要熟悉这种记号.

一般的, 一条规则具有形式

$$\frac{\text{Premise}_1 \quad \text{Premise}_2 \quad \dots \quad \text{Premise}_n}{\text{Conclusion}}\,(\text{Name})$$

记号的主体是一个分数线, 分子位置的是**条件 (premise)**, 分母位置的是**结论 (conclusion)**, 右侧是规则的名字, 没有实际的含义. 条件和结论同属**判断 (judgement)**. 判断是一系列关于这个类型论中的对象 (类型, 语境, 项) 的命题. 这样一条规则表示如果作为分子的所有条件成立, 那么作为分母的结论也成立.

3.2. 类型

类型论最大的特点就是每个项都有其**类型 (type)**. 我们引入一个新的判断 A type 表示 A 是一个合法的类型. 对于 STLC, 首先要取定一个基本类型 (base type) 的集合 $\mathcal{B}$, 然后才能递归地定义其的类型, 规则共 2 条:

$$\frac{A \in \mathcal{B}}{A \text{ type}}\ \text{Base}$$

虽然这里的规则很简单, 但本着面向初学者的精神, 我们在这一章还是尽量对每一条规则进行解释. Base 规则表示 "如果 A 是一个基本类型, 那么 A 是 STLC 的类型".

$$\frac{A \text{ type} \quad B \text{ type}}{A \to B \text{ type}}\ \text{Func}$$

Func 规则表示 "如果 A, B 是 STLC 的类型, 那么 $A \to B$ 是 STLC 的类型".

类型 $A \to B$ 表示类型 A 到类型 B 的函数, 其中 $\to$ 表示 "映射到". 为了方便, 我们规定 $\to$ 是右结合的, 这就是说 $A \to B \to C$ 是 $A \to (B \to C)$ 的简写.

读者可以发现, 实际上 STLC 的类型相当简单, 就算不用相继式演算, 也可以将 STLC 的类型定义为

定义 3.2.1:
- 如果 $A \in \mathcal{B}$, 那么 A 是 STLC 的类型.
- 如果 A, B 是 STLC 的类型, 那么 $A \to B$ 是 STLC 的类型.

[3]读者不要将 STLC 和**简单类型论** (Simple Type Theory, STT) 混淆, 这是两个不同的东西. STT 是 Church 风格的类型论.

为什么这么简单的类型也要采用相继式演算的特殊写法呢? 有两个原因. 其一是与下文中语境和项的定义保持一致, 以示类型论中定义的一致性; 其二是在有的类型系统中, 定义类型比这复杂得多, 此时使用相继式演算的记号是相当方便的.

3.3. 语境

语境 (context)可能是类型论中最重要的概念. 就像第 2 章中提到的在 M 中可以使用自变量 x, 语境记录的正是 "有哪些变量可用" 的信息. 为了形式化地定义语境, 我们引入新的判断 $\Gamma\ \mathrm{ctx}$ 表示 Γ 是一个合法的语境. 语境的规则有 2 条.

$$\frac{}{\emptyset\ \mathrm{ctx}}$$

这是我们第一次接触到没有条件的规则. 这条规则表示, 在任何情况下, $\emptyset$ 都是一个合法的语境. 我们把 $\emptyset$ 称为空语境.

$$\frac{\Gamma\ \mathrm{ctx}}{\Gamma, x : A\ \mathrm{ctx}}$$

这条规则表示如果 Γ 是一个语境, 那么 $\Gamma, x : A$ 是一个语境. $\Gamma, x : A$ 表示在 Γ 的基础上加入了一个新的变量 x, 并且具有类型 A.

读者可以想见, 外观上, 语境是一个 "变量 : 类型" 二元组组成的表头在右侧的列表. 我们定义一个谓词 $x : A \in \Gamma$ 表示 $x : A$ 这个二元组在 Γ 这个列表中存在.

练习 3.3.1: 使用相继式演算的语法形式化地定义这个谓词. 提示: 对语境的结构进行归纳.

非空语境具有 $\emptyset, x : A, y : B, \ldots$ 的形式, 为了方便, 在书写非空语境时, 常常省略 $\emptyset$, 记为 $x : A, y : B, \ldots$.

3.4. 项

一个类型论中的**项 (term)**是这个类型论中最常被讨论的部分. 在类型论中, 我们要时刻记得当讨论一个项时, 不能脱离其类型和所在的语境, 并引入一个新的判断 $\Gamma \vdash M : A$ 表示**在语境 Γ 下** M 是一个合法的项, 并且具有类型 A.

对于 STLC, 我们首先要给定一个常量的集合 $\mathcal{C}$, 以及一个映射 $\mathrm{toType} : \mathcal{C} \to \mathcal{B}$ 为 $\mathcal{C}$ 中的每一个常量指定一个它具有的基本类型, 然后递归地定义 STLC 的项, 规则共 4 条:

$$\frac{c \in \mathcal{C}}{\Gamma \vdash c : \mathrm{toType}(c)}\ \mathrm{Const}$$

这是常量的类型规则. 只要 c 是常量, 那么在任意语境中, c 都具有对应的基本类型.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \mathrm{Var}$$

这是变量的类型规则, 捕捉了我们之前关于 "语境记录了有哪些变量可用" 的直觉: 只要语境中存在变量 $x : A$, 那么变量 x 就可以 "使用", $x : A$ 就是这个语境下的项.

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda(x : A).M : A \to B}\ \mathrm{Lam}$$

项 $\lambda(x:A).M$ 表示 x 为类型 A 的自变量, M 为函数体的函数. 相比之前两个这条规则就稍微复杂了. 由于 M 是函数体, 可以额外使用函数的自变量, 因此如果 $\lambda x.M$ 在语境 Γ 中, M 就应当在语境 $\Gamma, x:A$ 中, 这样才能额外使用变量 x. 这条类型规则完整地解读为: 如果 $M:B$ 是语境 $\Gamma, x:A$ 的项, 那么 $\lambda(x:A).M:A\to B$ 是语境 Γ 中的项.

当 $\lambda(x:A)$ 中的类型 A 显然 (或者在其他版本的类型论中) 时, 常常可以省略.

$$\frac{\Gamma\vdash M:A\to B\quad \Gamma\vdash N:A}{\Gamma\vdash MN:B}\ \text{App}$$

这是函数应用的类型规则, 比较直接: 如果在语境 Γ 下, M 具有类型 $A\to B$, N 具有类型 A, 那么 MN 具有类型 B.

项 MN 表示函数 M 应用于参数 N (的结果). 与数学的函数不同, 类型论表示函数调用不用括号. 为了方便, 我们规定函数调用是左结合的, 即 MNR 是 $(MN)R$ 的简写. [4]

当语境比较明确, 或者不需要给出时也可以省略. 有时会省略一条规则中相同的语境.

例子 3.4.1: 规则 Lam 可以省略为

$$\frac{x:A\vdash M:B}{\lambda(x:A).M:A\to B}$$

当我们讨论 "存在一个语境 Γ, 这个语境下有项 $\Gamma\vdash M:A$", 如果讨论的重点是项, 也常常省略语境, 写作 $M:A$. 然而, 读者应始终记得**讨论一个项时, 不能脱离其类型和所在的语境**. 注意, 虽然 $M:A$ 和集合论中 $x\in A$ 类似, 但类型论中并没有与 $x\notin A$ 对应的记号: 类型论中无法讨论 "某个项不具有某类型".

顺道一提, 相继式演算中的的结论也可以当作进一步的条件:

例子 3.4.2: 空语境下 $\lambda f.\lambda x.f\ x:(A\to B)\to A\to B$ 的完整类型检查过程为

$$\dfrac{\dfrac{\dfrac{\dfrac{f:A\to B\in f:A\to B,x:A}{f:A\to B,x:A\vdash f:A\to B}\ \text{Var}\quad\dfrac{x:A\in f:A\to B,x:A}{f:A\to B,x:A\vdash x:A}\ \text{Var}}{f:A\to B,x:A\vdash f\ x:B}\ \text{App}}{f:A\to B\vdash\lambda x.f\ x:A\to B}\ \text{Lam}}{\emptyset\vdash\lambda f.\lambda x.f\ x:(A\to B)\to A\to B}\ \text{Lam}$$

通过上面的例子, 读者可能发现了, $\lambda f.\lambda x.f\ x$ 是一个二元函数. 实际上, 这就是 λ 演算构造多元函数的方法: 构造一个返回函数的函数, 返回的函数再接受剩余的参数. 规定 $\to$ 右结合以及函数应用左结合都是为了方便这一点. 为了进一步支持这个语法, 常常还把 $\lambda x.\lambda y....$ 简写为 $\lambda x\ y.....$

[4] 虽然本课程不会使用, 但有些时候在类型论的高端应用中, 仍然采用数学的函数写法, 这时候 $f(x,y)$ 表示 $f\ x\ y$. 注意, 之后会提到积类型的语法也是 (x,y), 因此在这种场合中读者需要自己根据上下文分辨对应的含义.

练习 3.4.1: 写出空语境下 $\lambda f\ g\ x.f(g\ x) : (B \to C) \to (A \to B) \to A \to C$ 的完整类型检查过程.

注意, 这里定义的 STLC 其实不是一个, 而是一族类型论: 每一个 $\mathcal{B}$, $\mathcal{C}$ 和 toType 的取法, 都给出了一个 STLC. 只是一族 STLC 的性质大致都相同, 因此我们只研究其中有代表性的一个即可. 这里我们取

$$\mathcal{B} = \{\top, \bot, \mathrm{Ans}\}$$
$$\mathcal{C} = \{\mathrm{tt}, \mathrm{yes}, \mathrm{no}\}$$
$$\mathrm{toType}(c) = \begin{cases} \top & (c = \mathrm{tt}) \\ \mathrm{Ans} & (c \in \{\mathrm{yes}, \mathrm{no}\}) \end{cases}$$

3.5. 替换

接下来我们介绍替换的概念, 这和语境一起通行于全部类型论, 可谓是类型论中最基本的概念.

替换捕捉的是 "代入" 的概念, 然而在定义替换之前我们先看一个问题. 按照第 2 章中建立的直觉, 请读者回答: 我们能将 $\lambda x.y$ 中的 y 替换为 x 吗? 答案显然是不能, 这破坏了 $\lambda x.y$ 原本的含义. 因此我们必须区分绑定变量和自由变量.

3.5.1. 出现

除了形如 $\lambda x.$ 中的 x 之外, 对变量的使用都称为**出现 (occurence)**. 其中, 如果变量 x 的某个出现在 $\lambda x.M$ 的某个 M 中, 就称这个 x 的出现为绑定出现 (bound occurence), 对应的 λx 为这个出现的绑定处 (binding site, 也称 binder); 否则为自由出现 (free occurence). (注意, 这不是真的自由了, 它们仍然会出现在语境中.) 直接说可能比较抽象, 不如直接举例: 下式中, 黑色的为自由出现, 彩色的为绑定出现, 颜色标明了绑定出现和绑定处的对应关系.

$$\lambda x.x\ (\lambda y.\lambda x.x\ y)\ z\ (x\ y)$$

注意, 绑定出现和自由出现是一对相对的概念, 可以相互转化. 如果只讨论上式框中的部分, 那么第一个 x 的出现就成了自由出现; 而如果在上式外面放上一个 λz 的绑定处, 则 z 又成了绑定出现.

$$\lambda x.\ \boxed{x\ (\lambda y.\lambda x.x\ y)\ z}\ (x\ y)$$

如果两个出现具有相同的名字, 并且都是自由变量或都被同一个绑定处绑定, 就称它们为同一个变量的出现. 实际上经常也将出现称为变量, 这应当不会造成什么混淆.

3.5.2. α 转换

正如 (以正确的方式) 给函数的参数改名不会出现什么问题, 同时给一个绑定处和被这个绑定处绑定的所有出现重命名也一样, 这称为**α 转换 (α conversion)**. α 转换的正式定义如下:

定义 3.5.2.1: 记将项 M 中所有 x 的自由出现重命名为 y 后的项为 N, 则 $\lambda x.M$ 和 $\lambda y.N$ 等价, 只要这个重命名没有将自由出现变成绑定出现.

对于一个（或有限任意多的）项, 总能经过有限次的 α 转换, 使其中不同的变量两两不同名. 这将为我们进行替换等提供莫大的方便. 日后当我们处理项时, 总认为它们中不同的变量是两两不同名的. 相对应的, 如果两个项在 α 转换下等价, 我们就认为它们是同一个项, 对它们不作区分.

有了 α 转换的铺垫, 我们立刻就可以定义**替换 (substitution)**:

定义 3.5.2.2: 记$M[x \mapsto N]$为将项 M 中全部 x 的自由出现替换为 N 得到的项. [5]

由于 M 和 N 中不同的变量两两不同名, 因此替换并不会使得 N 中原本的自由变量被绑定.

这里必须说明, α 转换并不是 λ 演算必须的规则, 而只是我们采用了字符串作为 λ 项的形式而产生的繁琐的技术细节. 我们完全可以使用语法树或者范畴中的对象表示 λ 项, 也的确存在这样的表示 (de Bruijn index), 其中的一个项对应我们的表示下的一个 α 等价类 (注意 α 转换确定了一个项上的等价关系). 离开了这一节, 我们立刻停止谈论出现和 α 转换的概念, 只使用替换的记号.

3.6. 归约

归约 (reduction) 主要描述了一个类型论的项怎么化简, 或者说计算. 对于 STLC 而言, 有 β 和 η 两条归约规则.

在给出归约的具体规则之前, 我们先思考一下归约规则的形式是什么. 在我们将 $(a+b+c)(a+b-c)$ 化简为 $(a+b)^2-c^2$ 时, 凭借的是 $(x+y)(x-y)=x^2-y^2$ 这个等式, 而归约规则在这里起的就是等式的作用. 同时, 由于等式没有方向, 因此 "最简形式" 是需要规定的.

接下来我们给出 β 和 η 两条归约的规则 (省略语境):

$$\frac{x : A \vdash M : B \quad N : A}{(\lambda x.M)N \equiv M[x \mapsto N] : B}\ \beta \text{ rule}$$

β 规则描述了函数的计算规则: 将实际参数 "代入" 函数中, 或者正式地说, 用实际的参数替换掉函数体中的自变量. 正向使用这个等式称为 β 归约.

$$\frac{f : A \to B}{f \equiv \lambda x.f\ x : A \to B}\ \eta \text{ rule}$$

η 规则描述了函数的外延性[6]: f 和 $\lambda x.f\ x$ 在任意相同的参数处取到相同的值, 因此也因当相等. 正向使用这个等式称为 η 展开 (η expansion), 反向使用称为 η 归约.

由于上两条规则的等号两侧具有相同的类型, 且共用相同的语境, 因此 STLC 中归约尊重类型和语境. 类型论的这种性质称为 subject reduction.

如果对一个 STLC 项不能再做任何的 β 或 η 归约, 且每个函数都是完全应用的 (即被应用到了最大数量的参数), 我们就说这个项是 $\beta\eta$-既约的, 简称既约的 (normalized), 或称其是**既约形式 (normal form)**. 空语境中的既约形式称为**闭既约形式 (canonical form)**.

[5] 替换的记号有很多且不统一, 除了 $M[x \mapsto N]$, 笔者见过的还有 $M[x := N]$, $M[N/x]$, $M[x/N]$, 后两种是刚好相反的, 请读者多加注意. 相较而言, $M[x \mapsto N]$ 应该是最不至于引起混淆的记号了.

[6] α 和 β 规则是 λ 演算最开始提出的时候就有的规则, 这两个名字没有实际含义, 而 η 规则是后来加上的, 起这个名字是考虑到了外延性 (extensionality) 的首字母 e. 类似的, 还有一个表示定义展开的 δ 归约, 命名取了定义 (definition) 的首字母 d.

例子 3.6.1: 将 $(\lambda x.\lambda y.x)y$ 归约到既约形式. (由于 subject reduction, 在下面的归约过程中, 我们省略类型和语境.)

$$\begin{aligned}
&(\lambda(f : \text{Ans} \to \text{Ans}).f\ \text{yes})(\lambda(x : \text{Ans}).x) \\
\equiv\ &(f\ \text{yes})[f \mapsto \lambda(x : \text{Ans}).x] && (\beta\ \text{reduction}) \\
\equiv\ &(\lambda(x : \text{Ans}).x)\ \text{yes} && (\text{substitution}) \\
\equiv\ &x[x \mapsto \text{yes}] && (\beta\ \text{reduction}) \\
\equiv\ &\text{yes} && (\text{substitution})
\end{aligned}$$

关于 STLC 的归约有两条重要的定理, 我们都仅作了解:

定理 3.6.1: 任何 STLC 项都可以通过有限步归约达到既约形式.

如果一个类型论的归约具有这种性质, 我们就称它是**停机的 (terminating)**. 停机性满足了数学上对于函数的要求: 经过有限步的计算 (归约) 之后总能得到结果 (既约形式). 与此同时, 停机性违背了计算机科学对一门编程语言的期待: 只要归约是停机的, 这个类型论 (作为编程语言时) 就不能图灵完备 (因为图灵完备的程序可以显然的不停机). 由于类型论的数学属性, 我们基本上都会要求一个类型论的归约是停机的.

进一步的, 我们还有 Church–Rosser 定理:

定理 3.6.2: 一个 STLC 项有唯一的既约形式, 与归约顺序无关.

类型论的这个性质称为 Church-Rosser 性, 或者**合流性 (confluence)**. 合流性说明了 (作为数学函数的) 计算结果与计算过程无关. 这两条性质结合起来, 就表明一个类型论作为计算系统是合格的.

练习 3.6.1: 将以下 STLC 项归约到既约形式:
1. $(\lambda x.\lambda y.x)\ y$
2. $\lambda(f : A \to B \to C).f$

4. STLC 的模型

模型 (model), 也叫做**语义 (semantics)** (与之相对, 这时类型论就叫做**语法 (syntax)**), 是我们研究一个类型论的性质最重要的工具. 为了研究不同的性质, 可以为一个类型论构造许多不同的模型.

通俗的说, 为类型论构造一个模型, 就是将类型论中的每个对象 (项, 类型, 语境等) 映射 (解释) 到模型中的一个对象, 使得原本的类型论中存在的每个构造 (类型规则), 在模型中也都对应地存在; 原本的类型论中成立的每个基本性质 (β, η 规则), 对模型也都成立 (即模型保持类型论的基本性质). 而由于复杂的性质都由这些基本性质推出, 因此模型就保持类型论的全部性质, 这叫做一致

性 (soundness). 因此, **如果一个性质对某个类型论成立, 那么它对这个类型论的每个模型都成立; 反之, 如果一个性质对某个模型不成立, 那么它就对这个类型论不成立**.

注意, "命题不成立" 定义为 "如果命题成立, 那么存在矛盾". 它不等同于 "命题的否定成立", 后者能推出前者.

在这里我们不形式化地给出模型的定义, 但我们给出 STLC 的一个模型 [1]. 这个模型称为 **STLC 的集合模型**.

定理 4.1 (STLC 的集合模型): STLC 在集合论中存在一个模型.

证明: 按照惯例, 我们用同一个记号 $[\![-]\!]$ 表示模型中的三个映射.

首先, 我们将类型解释为集合:

$$\begin{aligned}
[\![\bot]\!] &= \emptyset \\
[\![\top]\!] &= \{*\} \\
[\![\mathrm{Ans}]\!] &= \{0, 1\} \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!]
\end{aligned}$$

然后是语境. 这里我们把长度为 n 的语境解释为了 $n+1$ 元笛卡儿积:

$$\begin{aligned}
[\![\emptyset]\!] &= \{*\} \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!] \times [\![A]\!]
\end{aligned}$$

最后, 我们将项 $\Gamma \vdash M : A$ 解释为 $[\![\Gamma]\!] \to [\![A]\!]$ 的函数:

$$\begin{aligned}
[\![\Gamma \vdash \mathrm{tt} : \top]\!] &= - \mapsto * \\
[\![\Gamma \vdash \mathrm{yes} : \mathrm{Ans}]\!] &= - \mapsto 1 \\
[\![\Gamma \vdash \mathrm{no} : \mathrm{Ans}]\!] &= - \mapsto 0 \\
[\![\Gamma \vdash x : A]\!] &= \pi_{i+1} \quad (x : A\text{是}\Gamma\text{中的第}i\text{项})
\end{aligned}$$

这里 π_{i+1} 表示从笛卡尔积中取出第 $i+1$ 项的投影函数.

我们特别讨论函数应用和构造的情况, 它们写在一行里并不容易理解. 首先考虑

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \mathrm{App}$$

记 $m = [\![\Gamma \vdash M : A \to B]\!] \in [\![\Gamma]\!] \to ([\![A]\!] \to [\![B]\!]), n = [\![\Gamma \vdash N : A]\!] \in [\![\Gamma]\!] \to [\![A]\!]$. 我们构造 $[\![\Gamma \vdash MN : B]\!] = \gamma \mapsto m(\gamma)(n(\gamma)) \in [\![\Gamma]\!] \to [\![B]\!]$.

然后是

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \mathrm{Lam}$$

记 $m = [\![\Gamma, x : A \vdash M : B]\!] \in [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$, 则可以将 $[\![\Gamma \vdash \lambda x.M : A \to B]\!]$ 构造为 $\gamma \mapsto (x \mapsto m(\gamma, x)) \in [\![\Gamma]\!] \to ([\![A]\!] \to [\![B]\!])$.

最后我们验证 α 转换, β 和 η 的规则是否成立. 由于上述构造不涉及变量的名称, α 转换显然成立. 感兴趣的读者可以自己尝试证明 β 规则成立, 这里仅为感兴趣的读者提供 η 规则成立的证明.

首先我们记 $f = [\![\Gamma \vdash f : A \to B]\!] \in [\![\Gamma]\!] \to ([\![A]\!] \to [\![B]\!])$. 根据上面的构造, 我们有

$$\begin{aligned}[\![\Gamma, x : A \vdash x : A]\!] &= (\gamma, x) \mapsto x \\ [\![\Gamma, x : A \vdash f\ x \ \ ]\!] &= (\gamma, x) \mapsto f(\gamma)(x)\end{aligned}$$

因此由 (集合论) 函数的外延相等,

$$\begin{aligned}&[\![\Gamma \vdash \lambda x.f\ x : A \to B]\!] \\ =\ &\gamma \mapsto (x \mapsto f(\gamma)(x)) \\ =\ &\gamma \mapsto f(\gamma) \\ =\ &f \\ =\ &[\![\Gamma \vdash f : A \to B]\!]\end{aligned}$$

□

另外我们还有

定理 4.2 (模型的一致性): 如果 $\Gamma \vdash M \equiv N : A$, 那么 $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!]$.

证明: 因为模型保持 α, β, η 规则给出的相等, 通过对项的结构进行归纳, 可以证明模型保持项上的相等关系. 这条定理适用于任何 STLC 的模型. □

于是我们立刻就有一些简单的推论:

推论 4.1: 空语境下不存在类型为 $\bot$ 的项.

证明: 这样的项在集合模型中是 $\{*\} \to \emptyset$ 的函数. □

推论 4.2: 任何语境下都有 $\frac{\text{yes}}{=} = \text{no} : \text{Ans}$.

证明: 我们构造另一个集合模型, 唯一的区别是 $[\![\bot]\!] = \{*\}$. 假设存在 Γ 使 $\text{yes} \equiv \text{no}$, 那么 $[\![\Gamma \vdash \text{yes}]\!] = [\![\Gamma \vdash \text{no}]\!]$, 然而前者是值为 1 的常函数, 后者是值为 0 的常函数. 而 $[\![\Gamma]\!]$ 显然不为空, 因此这两个常函数不相等, 矛盾. □

实际上, 模型的能力远不止于此. 通过构造更复杂的模型, 可以证明类型论中更复杂的性质, 如 (开/闭) 典范性等, 这常常要借助范畴论的工具. 事实上, "STLC 的模型" 这一概念也可以通过范畴论的语言简洁地形式化定义[2]:

定义 4.1: 由图表 $\bot, 1 \to \top, 1 \rightrightarrows \text{Ans}$ 自由生成的积闭范畴称为 STLC 的**语法范畴**. 一个从这个范畴出发的保积闭函子称为 STLC 的一个**语义**.

事实上, 上面定义的集合模型就是一个从语法范畴出发到集合范畴 Set 的函子.

相较而言, 因为集合模型不仅是将 STLC 的项解释到了函数, 而且实际上被解释到的是可计算函数 (我们不展开这个概念. 大致的说, 就是可以在图灵机上表示/编程语言中写出的函数.), 它的一个更大的作用是解释运行 STLC. 下面是一个 STLC 的解释器, 几乎完全照搬了集合模型的构造, 只用 14 行就完成了.

```
function interprete(context, term, gamma) {
  if (term.type === 'const') {
    return term.value;
  } else if (term.type === 'var') {
    return gamma[context.indexOf(term.name)];
  } else if (term.type === 'lam') {
    return x => interprete([term.var].concat(context), term.body, [x].concat(gamma));
  } else if (term.type === 'app') {
    var func = interprete(context, term.f, gamma);
    var arg = interprete(context, term.x, gamma);
    return func(arg);
  }
}
```

5. 扩展 STLC

实际上到了这里, STLC 的表达能力还是很弱的. 这一章里我们要对 STLC 做一些简单的扩展, 扩充它的表达能力.

5.1. Bool 类型

可能有的读者会疑惑为什么不将 Ans 类型称为 Bool 类型. 实际上, Ans 类型和 Bool 类型并不一样, STLC 内无法使用一个 Ans 类型的项, 从 Ans 类型出发的函数一定是平凡的. 比如说, 无法对每个类型 A 都定义函数 $\mathrm{choose} : \mathrm{Ans} \to A \to A \to A$, 使得 $\mathrm{choose\ yes}\ a\ b \equiv a$, $\mathrm{choose\ no}\ a\ b \equiv b$.

练习 5.1.1: 构造一个 $[\![\mathrm{yes}]\!] = [\![\mathrm{no}]\!]$ 的模型, 证明上述命题. 注意, 这和推论 4.2 不矛盾. 想一想为什么.

Ans 类型缺少的是**消去规则 (elimination rule)**. 消去规则描述了我们如何使用这个类型的值, 与之相对的, 说明如何构造一个类型的项的规则称为**引入规则 (introduction rule)**. 这里我们就以简单的 Bool 类型 (记为 $\mathbb{B}$) 为例子, 它的引入规则和 Ans 一样 (这里全部省略语境, 表示 "在任意语境中"):

$$\frac{}{\mathrm{true} : \mathbb{B}} \qquad \frac{}{\mathrm{false} : \mathbb{B}}$$

消去规则类似于我们常见的 C 语言三元表达式[7]:

$$\frac{b : \mathbb{B} \quad c_t : C \quad c_f : C}{\mathrm{elim}_{\mathbb{B}}(b, c_t, c_f) : C}$$

然而我们还缺少两种规则, 第一种描述了 $\mathrm{elim}_{\mathbb{B}}(b, c_t, c_f)$ 怎么归约, 这称为**计算规则 (computation rule)**:

[7]这里打括号, 表示 $\mathrm{elim}_{\mathbb{B}}$ 不是一个具有函数类型的项; 只有 $\mathrm{elim}_{\mathbb{B}}(b, c_t, c_f)$ 才是一个完整的项.

$$\frac{c_t : C \quad c_f : C}{\mathrm{elim}_{\mathbb{B}}(\mathrm{true}, c_t, c_f) \equiv c_t : C}$$

$$\frac{c_t : C \quad c_f : C}{\mathrm{elim}_{\mathbb{B}}(\mathrm{false}, c_t, c_f) \equiv c_f : C}$$

或许读者会认为规则到这里已经足够了, 但其实我们还缺少一条**唯一性规则 (uniqueness rule)**:

$$\frac{b : \mathbb{B} \quad c : \mathbb{B} \to C}{\mathrm{elim}_{\mathbb{B}}(b, c\ \mathrm{true}, c\ \mathrm{false}) \equiv c\, b : C}$$

这 4 种规则完整地定义了一个类型.

练习 5.1.2: 构造满足 $[\![\mathrm{true}]\!] = [\![\mathrm{false}]\!]$ 的所有模型.

或许读者会难以理解唯一性规则的作用, 没关系, 我们继续看几个例子.

5.2. 积类型

积类型 (product type) 在编程中经常称为元组类型, 在数学中的类似物 (以及在集合模型中) 是笛卡尔积. 它引入了一个新的类型, 因此我们还要首先定义积类型的**形成规则 (formation rule)**[8]:

$$\frac{A\ \mathrm{type} \quad B\ \mathrm{type}}{A \times B\ \mathrm{type}}$$

之后我们给出积类型的项的的定义. 首先是引入规则, 描述了如何用两个项构造一个元组:

$$\frac{a : A \quad b : B}{(a, b) : A \times B}$$

然后是消去规则, 描述了如何将元组中的项取出来:

$$\frac{p : A \times B}{\mathrm{fst}(p) : A} \qquad \frac{p : A \times B}{\mathrm{snd}(p) : B}$$

计算规则保证了消去规则取到的是正确的项:

$$\frac{a : A \quad b : B}{\mathrm{fst}((a, b)) \equiv a : A} \qquad \frac{a : A \quad b : B}{\mathrm{snd}((a, b)) \equiv b : B}$$

那唯一性规则保证了什么? 正如消去规则是引入规则的对偶, 唯一性规则也是计算规则的对偶. 因此, 这里唯一性规则应当保证引入规则构造的是正确的项:

$$\frac{p : A \times B}{p \equiv (\mathrm{fst}(p), \mathrm{snd}(p)) : A \times B}$$

更一般的说, 计算规则保证的是消去规则和引入规则相抵消, 而唯一性规则保证的是引入规则和消去规则项抵消. 有了完整定义的这 4 种规则, 一个类型才是性质良好的.

其实, 受限于类型论的表达能力, 这里的许多规则都不是完全体, 在 MLTT 之类的类型论中读者才能看到这些规则的完全版本.

[8] 类似 Bool 类型的简单类型同样有形成规则 $\frac{}{\mathbb{B}\ \mathrm{type}}$, 只是形式简单, 我们省略了.

5.3. 余积类型

余积类型 (coproduct type) 也称为和类型 (sum type), 大致类似于编程中的具名联合 (tagged union), 是积类型的对偶.

$$\frac{A \text{ type} \quad B \text{ type}}{A + B \text{ type}}$$

只要有 A 或 B 类型的一个项, 就能构造一个 $A + B$ 类型的项. 注意这和积类型的消去规则类似的形式:

$$\frac{a : A}{\text{inl}(a) : A + B} \quad \frac{b : B}{\text{inr}(b) : A + B}$$

相对的, 必须同时指明如何使用 A 和 B 类型的项, 才能使用一个 $A + B$ 类型的项.

$$\frac{s : A + B \quad c_l : A \to C \quad c_r : B \to C}{\text{elim}_{A+B}(s, c_l, c_r) : C}$$

这里的对偶并不显然, 但我们可以把积类型的引入规则改写成类似的形式:

$$\frac{c : C \quad c_l : C \to A \quad c_r : C \to B}{(c_l\ c, c_r\ c) : A \times B}$$

练习 5.3.1: 给出余积类型的计算规则. 唯一性规则比较复杂, 不作要求.

5.4. 自然数类型

我们把自然数类型记作 $\mathbb{N}$. 参照皮亚诺公理, 我们对自然数采取递归的定义: 一个自然数要么是 0, 要么是某个自然数的后继 (加一). 引入规则描述了这一点.

$$\frac{}{\text{zero} : \mathbb{N}} \quad \frac{n : \mathbb{N}}{\text{suc}(n) : \mathbb{N}}$$

消去规则表达了我们能在 $\mathbb{N}$ 上进行的递归, 为了描述这一点, 计算规则也是递归的:

$$\frac{n : \mathbb{N} \quad c_0 : C \quad c_s : \mathbb{N} \to C \to C}{\text{elim}_{\mathbb{N}}(n, c_0, c_s) : C}$$

$$\frac{c_0 : C \quad c_s : \mathbb{N} \to C \to C}{\text{elim}_{\mathbb{N}}(\text{zero}, c_0, c_s) \equiv c_0 : C} \quad \frac{n : \mathbb{N} \quad c_0 : C \quad c_s : \mathbb{N} \to C \to C}{\text{elim}_{\mathbb{N}}(\text{suc}(n), c_0, c_s) \equiv c_s\ n\ \text{elim}_{\mathbb{N}}(n, c_0, c_s) : C}$$

不正式的说,

$$\text{elim}_{\mathbb{N}}(n, c_0, c_s) \equiv \underbrace{c_s(n-1)(c_s(n-2)...(c_s\ \text{zero}\ c_0))}_{n \text{ times}}$$

唯一性规则相较而言比较平凡[9]:

$$\frac{n : \mathbb{N} \quad c : \mathbb{N} \to C}{c\ n \equiv \text{elim}(n, c\ \text{zero}, \lambda m.\underline{\lambda\_}.c\ \text{suc}(m)) : C}$$

[9] 下划线是一个习惯记法, 表示不会使用由 λ 引入的这个变量.

有了递归, 我们就可以在自然数上定义一些运算.

例子 5.4.1 (自然数加法): 自然数上的加法定义为

$$\begin{aligned}&\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\&\mathrm{add}\ m\ n \coloneqq \mathrm{elim}_{\mathbb{N}}(m, n, \lambda\_.\lambda n.\ \mathrm{suc}(n))\end{aligned}$$

容易知道这相当于

$$\mathrm{add}\ m\ n \coloneqq \underbrace{\mathrm{suc}(\mathrm{suc}(...(\mathrm{suc}(n))))}_{m \text{ times}}$$

练习 5.4.1: 定义自然数上的乘法.

5.5. 模式匹配

在许多函数式语言 (基于 STLC 或它的扩展的编程语言) 中, 实际使用的不是消去规则, 而是**模式匹配 (pattern matching)**. 既然我们已经由计算规则和唯一性规则知道了消去规则和引入规则之间互相抵消, 我们就可以对引入规则中的每种情况 (称为**构造器 (cosntructor)**) 进行归纳, 这等价于我们应用了消去规则.

我们使用类似 Haskell 的语法 case ... of 来示范一下如何将几个消去规则翻译到模式匹配:

$$\begin{aligned}&\mathrm{elim}_{\mathbb{B}}(b, c_t, c_f) && \mathrm{case}\ b\ \mathrm{of}\\&&&\quad \mathrm{true} \to c_t\\&&&\quad \mathrm{false} \to c_f\\&f\ \mathrm{fst}(p)\ \mathrm{snd}(p) && \mathrm{case}\ p\ \mathrm{of}\\&&&\quad (a, b) \to f\ a\ b\\&\mathrm{elim}_{A+B}(s, c_l, c_r) && \mathrm{case}\ s\ \mathrm{of}\\&&&\quad \mathrm{inl}(a) \to c_l\ a\\&&&\quad \mathrm{inr}(b) \to c_r\ b\end{aligned}$$

自然数的模式匹配稍稍不一样. 自然数的结构是递归的, 它的模式匹配也必然是递归的. 这里我们采用递归函数的语法, 在函数的参数处进行模式匹配:

$$\begin{aligned}&f \coloneqq \lambda n.\mathrm{elim}_{\mathrm{Nat}}(n, c_0, c_s) && f\ \mathrm{zero} \quad \coloneqq c_0\\&&& f\ \mathrm{suc}(m) \coloneqq c_s\ m\ (f\ m)\end{aligned}$$

例子 5.5.1: 使用模式匹配的语法, 例子 5.4.1 中的自然数加法可以写成

$$\begin{aligned}&\mathrm{add}\ \mathrm{zero} \quad n \coloneqq n\\&\mathrm{add}\ \mathrm{suc}(m)\ n \coloneqq \mathrm{suc}(\mathrm{add}\ m\ n)\end{aligned}$$

练习 5.5.1: 将练习 5.4.1 中定义的自然数乘法改写成模式匹配的形式.

在定义 $f\ \mathrm{suc}(m)$ 的时候我们递归地使用了 $f\ m$. 递归在编程上不成问题, 图灵完备的编程语言允许任意的递归; 但正是因此在数学中不能有任意的递归, 因为对于自变量的每一个可能的值, 都要有一个唯一的函数值与之对应, 而类似 $f(x) \coloneqq f(x)$ 的递归显然不满足这样的要求. 这里我们的要求其实就是之前提到的停机性: 我们只能使用那些保证归约过程停机的递归.

不幸的是, 判断任意递归函数是否停机是不可能的. 然而, 我们可以找到递归函数停机的一个充分不必要条件, 对这个条件进行判断, 这称为停机性检查 (termination check). 这在证明助手中常常是必要的. 停机性检查的具体实现十分复杂, 其核心思想是检查函数是否有一个参数在每个递归调用中都是 "减小" (比如, 失去了一个构造器) 的.

5.6. 为什么类型论都选择了 λ 演算

与积类型一样, 函数类型同样应当有上述的 4 种规则: 他们是什么?

其实这些规则都已经藏身于 λ 演算当中. 引入规则毫无疑问是 Lam:

$$\frac{x : A \vdash M : B}{\lambda x.M : A \to B}$$

而消去规则, 与之相对, 应该描述一个函数类型的项如何被使用, 因此这是 App:

$$\frac{M : A \to B \quad N : A}{MN : B}$$

计算规则应该描述如何归约, 因此这是 β, 同时 β 也描述了消去规则如何抵消引入规则:

$$\frac{x : A \vdash M : B \quad N : A}{(\lambda x.M)N \equiv M[x \mapsto N] : B}$$

与之相对的, 是引入规则如何抵消消去规则, η:

$$\frac{f : A \to B}{\lambda x.f\ x \equiv f : A \to B}$$

受函数类型的影响, 在其它的类型上, 计算规则同样常常被称为 β 规则, 唯一性规则称为 η 规则.

由此可见, λ 演算中绝大多数的规则都是在定义函数类型. 事实上, 这正是函数类型最精简的定义. 因此无论是简单的类型论如 STLC, 还是复杂的类型论如 MLTT, HoTT 等, 都不约而同的选择了 λ 演算作为了函数的定义.

5.7. 特殊的类型

实际上, 不是每个 (性质好的) 类型都有这 4 种规则, 而不具有全部规则的类型往往有特殊的性质. 比较常见的例子是 $\top$ 和 $\bot$, 它们是对偶类型.

$\top$ 只有引入规则和计算规则:

$$\frac{}{\mathrm{tt} : \top}$$

$$\frac{e : \top}{e \equiv \mathrm{tt} : \top}$$

这两条规则说明了 $\top$ 类型只有唯一的项 tt, 没有承载任何信息, 因此任何的消去规则都是没有意义的.

$\bot$ 只有消去规则和唯一性规则. 没有引入规则意味着这个类型的项不能被构造 (见推论 4.1), 因此也无需计算规则说明如何计算. 而正因 $\bot$ 类型的项不存在, 给出一个 $e : \bot$, 就可以拿到具有任意性质的任意项 — 反正 $\bot$ 类型的项不存在. 这称为谎言爆炸原理 (principle of explosion).

$$\frac{e : \bot}{\mathrm{elim}_\bot(e) : C}$$

$$\frac{e : \bot \quad c : C}{\mathrm{elim}_\bot(e) \equiv c : C}$$

$\top$ 是 $A \times B$ 类型的单位元, 换句话说, $A \cong A \times \top \cong \top \times A$; 对偶的, $\bot$ 也是 $A + B$ 的单位元. 读者可以尝试证明.

5.8. Curry–Howard 同构

在这一节中, 我们首先简单的介绍一下直觉主义自然演绎. "直觉主义" 表示这个系统不包含排中律; 其余的部分相当于一个命题逻辑系统.

记 Γ 为一个命题的有限集合, 称为假设 (或公理). 定义仅在下列情况中, 命题 P 可由 Γ 推出, 记为 $\Gamma \vdash P$:

- P 是 Γ 中的一个命题. 这称为引入假设 (或使用公理).
- P 具有 $A \to B$ 的形式, 并且 $\Gamma, A \vdash B$. 这称为演绎定理 (deduction theorem).
- 存在 A 使得有 $\Gamma \vdash A \to P$ 和 $\Gamma \vdash A$. 这称为肯定前件 (*modus ponens*).

使用相继式演算的记号, 就是

$$\frac{P \in \Gamma}{\Gamma \vdash P}\ \mathrm{Ax} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to I \qquad \frac{\Gamma \vdash A \to P \quad \Gamma \vdash A}{\Gamma \vdash P} \to E$$

读者或许已经发现, 这三条规则与 Var, Lam 和 App 十分相像. 而这其实说明了一个事实: 我们是可以把 STLC 当作一个逻辑系统来使用的. STLC 中的函数类型对应于直觉主义自然演绎中的蕴含, 类型对应于其中的命题, 而语境对应于其中的假设. 这种对应关系称为 **Curry–Howard 同构** (简称 CH 同构). 而由于这种同构过于普遍, 我们现在已经认为类型论就是一种逻辑系统, 即使没有一个现成的逻辑系统与之对应.

那么项对应于什么呢? 答案是证明. 项的结构对应了证明的结构: 变量代表了证明的这一步引入了假设; λ 表示这一步应用了演绎定理; 函数应用则表示应用了肯定前件. 由此可知, 如果在一个语境下一个类型有项, 那么这个类型对应的命题在语境对应的假设下为真. 我们将空语境下有项的类型称为**居留 (inhabited)** 的, 对应直觉主义命题逻辑中的重言式.

例子 5.8.1: 证明 $(P \to Q) \to (Q \to R) \to P \to R$ (按照我们的惯例, 规定 $\to$ 右结合).

$\text{proof} : (P \to Q) \to (Q \to R) \to P \to R$

$\text{proof} := \lambda f\ g\ x.g\ (f\ x)$

逻辑对应: 由演绎定理, 证明 $(P \to Q) \to (Q \to R) \to P \to R$, 等价于在假设 $P \to Q$, $Q \to R$, P 下证明命题 R. 对 $P \to Q$ 和 P 应用肯定前件, 得到 Q. Q 再和 $Q \to R$ 应用肯定前件, 得到 R.

一个更常见的逻辑系统是直觉主义 Hilbert 演绎系统, 它和上述的直觉主义自然演绎等价. 相较于上述的这个逻辑系统, 它没有 $\to I$, 但是多了两个公理模式, 具体而言就是 $\Gamma \vdash P$ 如果 P 是以下两种公理模式之一的一个实例:

- $A \to B \to A$
- $(A \to B \to C) \to (A \to B) \to A \to C$

而在 STLC 中, 也可以写出如下两个函数:

$K : A \to B \to A$
$K := \lambda x\ y.x$

$S : (A \to B \to C) \to (A \to B) \to A \to C$

$S := \lambda x\ y\ z.x\ z\ (y\ z)$

刚好与这 2 条公理模式对应. 而对应的, 将 S, K 当作常量并移除 Lam 规则的类型论叫组合子演算, 它的表达能力同样和 STLC 等价.

类似的, STLC 的扩展也能在逻辑系统中找到对应:

- $\top$ 对应真命题, tt 是它的证明.
- $\bot$ 对应假命题, $\mathrm{elim}_\bot$ 是谎言爆炸原理.
- $A \times B$ 对应 $A \wedge B$: 如果命题 A 有证明 a, 命题 B 有证明 b, 那么 $A \wedge B$ 有证明 (a, b).
- $A + B$ 对应 $A \vee B$: 如果命题 A 有证明 a, 那么 $A \vee B$ 有证明 $\mathrm{inl}(a)$; 如果命题 B 有证明 b, 那么 $A \vee B$ 有证明 $\mathrm{inr}(b)$.

这里没有列出否定, 因为 "P 的否定" 就是 "P 能推出假".

例子 5.8.2: 证明 $A \wedge B \to (A \to C) \vee (B \to C) \to C$.

$\text{proof} : A \times B \to (A \to C) + (B \to C) \to C$

$\text{proof}\ (a, b)\ \mathrm{inl}(f) := f\ a$

$\text{proof}\ (a, b)\ \mathrm{inr}(g) := g\ b$

逻辑对应: 由演绎定理, 知这是在假设 $A \wedge B$, $(A \to C) \vee (B \to C)$ 下证明命题 C. 由于 $A \wedge B$ 为真, A 和 B 都为真. 如果 $(A \to C) \vee (B \to C)$ 中
- $A \to C$ 为真: 对 $A \to C$ 和 A 应用肯定前件, 得到 C 为真.
- $B \to C$ 为真: 对 $B \to C$ 和 B 应用肯定前件, 得到 C 为真.

问题 5.8.1: 类型 $A \to B \to A + B$ 有两个项: $\lambda a\ b.\ \mathrm{inl}(a)$ 和 $\lambda a\ b.\ \mathrm{inr}(b)$. 它们在逻辑中的对应是什么?

或许有读者会疑问, 这不等于在说命题 $A \to B \to A \vee B$ 有两个证明吗? 实际上的确差不多如此: 这是在说 $A \to B \to A \vee B$ 有两种证明方式. 同理, $\mathbb{B}$ 对应了一个 "有两种证明方式的命题".

那难道 $\mathbb{N}$ 对应一个有无数种证明方式的命题吗? 这显然不大有意义. 事实上, 这暴露了我们的 CH 同构中的某些瑕疵: 并不是每个类型都对应一个有意义的命题. 这个问题我们留待本暑校中讲到更复杂的类型论时解决.

5.8.1. 直觉主义

直觉主义数学一般指构造性数学, 它们只承认构造性的证明, 认为 "命题为真" 等价于 "证明存在", 这和用类型论做证明时的姿态是相似的: 证明一个命题就是要构造对应类型的一个项, 无怪乎各式类型论都对应于各式直觉主义的逻辑系统. 因此直觉主义数学拒斥非构造性的公理, 即在证明中使用选择公理和排中律 (后者是前者的推论; 而在类型论中, 排中律又叫选择公理). 因此, 一个直觉主义的证明比经典的证明具有更强的含义. 这里简单讲一讲拒绝排中律的理由.

澄清 5.8.1.1: 这里排中律不单指 $A \vee \neg A$, 而且泛指那些逻辑系统中能够推出排中律的公理等, 例如 Hilbert 第三公理 $(\neg P \to \neg Q) \to P \to Q$, 双重否定消去律 $\neg\neg A \to A$ 以及 Peirce 律 $((P \to Q) \to P) \to P$ 等.

在经典数学中, 承认 $A \vee \neg A$ 只是相当于承认 A 和 $\neg A$ 中有一个为真, 这没有大的问题. 而在直觉主义数学中, 承认 $A \vee \neg A$ 就相当于给出了 $A \vee \neg A$ 的一个构造性的证明; 进一步的, 这就是在给出 A 和 $\neg A$ 之一的证明, 而这相当于 "任何命题和它的否定之中总有一个可证"! 这显然是不可接受的.

对逻辑不了解的读者需要注意, 直觉主义逻辑只是不承认 $A \vee \neg A$ ($A \vee \neg A$ 无法证明), 而不是 $A \vee \neg A$ 为假 ($\neg(A \vee \neg A)$ 有证明). 这是之前提到的命题不成立和命题的否定成立的区别. 事实上, 对于某些 (很多) 具体的 A, 排中律仍然是成立的, 只是没有 $\forall A.A \vee \neg A$.

但是, 缺少了排中律会影响我们做数学吗? 答案是令人惊奇的: 没有显著的影响.

定理 5.8.1.1 (Godel 嵌入): 命题 P 在 Peano 算术中可证, 当且仅当 $\neg\neg P$ 在 Heyting 算术中可证.

Heyting 算术是 Peano 算术的无排中律版本. 类似的关系在其它的逻辑系统中也成立. 因此在直觉主义逻辑中如果我们只考虑形如 $\neg\neg P$ 的命题, 就能获得原本经典逻辑的使用体验. 在这个基础上, 讨论 $\neg\neg P$ 与 P 的区别 (例如, 对于哪些 P, 只有 $\neg\neg P$ 可证明, 哪些二者都可证明?), 是直觉主义逻辑相较于经典逻辑更为丰富的地方[2]. 因此, 放弃了排中律反而丰富了数学.

参考文献

[1] Qi, Xuanrui, "Type Theory and the Logic of Toposes", 2021, [Online]. Available: https://www.xuanruiqi.com/assets/masters_thesis.pdf

[2] Trebor, *A Brief History of Type Theory*. [Online]. Available: https://github.com/Trebor-Huang/history