STLC 开始的类型论入门

∞-type Café 暑期学校讲座

Jun 18, 2023

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

[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.

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 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 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 M, N 表示 STLC 的项.
- 使用 Γ 表示 STLC 的语境.

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.

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

[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 Chruch 风格的类型论.

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

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.

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 Γ ctx 表示 Γ 是一个合法的语境. 语境的规则有 2 条.

$$\frac{}{\emptyset \text{ ctx}}$$

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.

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

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”二元组组成的反向列表. 我们定义一个谓词 $x : A \in \Gamma$ 表示 $x : A$ 这个二元组在 Γ 这个反向列表中存在.

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.

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:

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

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.

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

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.

[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.

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 $\lambda f.\, \lambda x.\, f\ x : (A \to B) \to A \to B$ 的完整类型推导过程为

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

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 $\lambda x\ y.\, ...$.

练习 3.4.1: 写出空语境下 $\lambda f\ g\ x.\, f(g\ x) : (B \to C) \to (A \to B) \to A \to C$ 的完整类型推导过程.

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

$$B = \{\top, \bot, \text{Ans}\}$$

$$\mathcal{C} = \{\text{tt}, \text{yes}, \text{no}\}$$

$$\text{toType}(\text{tt}) = \top$$

$$\text{toType}(a) = \text{Ans} \quad (a \in \{\text{yes}, \text{no}\})$$

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

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. 规约

规约 (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 两条规约规则.

[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.

在给出规约的具体规则之前，我们先思考一下规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, 而规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 两条规约的规则 (省略语境):

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

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 规约.

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

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 规约.

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 中规约尊重类型和语境. 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 和 η 规约, 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)**.

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

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

关于 STLC 的规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规约达到既约形式.

如果一个类型论的规约具有这种性质, 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 (规约) 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: 只要规约是停机的, 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规约是停机的.

进一步的, 我们还有 Chruch–Rosser 定理:

[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 规约, 命名取了定义 (definition) 的首字母 D.

定理 3.6.2：一个 STLC 项有唯一的既约形式, 与规约顺序无关.

类型论的这个性质称为 Church-Rosser 性, 或者**融贯性 (confluence)**. 融贯性说明了 (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 项规约到既约形式:
1. $(\lambda x.\, \lambda y.\, x)y$
2. $\lambda(f : A \to B).\, \lambda(x : A).\, f\ x$

3.7. 模型

模型 (model), 也叫做**语义 (sematics)** (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成立的每个基本性质 (类型规则, β η 规则等), 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**.

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**.

定义 3.7.1：**(STLC 的集合模型)** 我们用记号 $[\![-]\!]$ 表示模型中的映射.

首先, 我们将类型解释为集合:

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

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:

$$[\![\emptyset]\!] = \{\emptyset\}$$
$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

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

$$[\![\Gamma \vdash \mathrm{tt} : \top]\!] = - \mapsto *$$
$$[\![\Gamma \vdash \mathrm{yes} : \mathrm{Ans}]\!] = - \mapsto 1$$
$$[\![\Gamma \vdash \mathrm{no} : \mathrm{Ans}]\!] = - \mapsto 0$$

$$[\![\Gamma, x : A]\!] = \pi_{i+1} \quad (x : A\text{是}\Gamma\text{中的第}i\text{项})$$

我们特别讨论函数应用和构造的情况, 因为它们的构造形式化地写出来似乎并不容易理解. 首先考虑

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

这时候 $[\![\Gamma \vdash M : A \to B]\!]$ 是 $[\![\Gamma]\!] \to ([\![A]\!] \to [\![B]\!])$ 的函数, $[\![\Gamma \vdash N : A]\!]$ 是 $[\![\Gamma]\!] \to [\![A]\!]$ 的函数. 我们需要的 $[\![\Gamma \vdash MN : B]\!]$ 应该是一个 $[\![\Gamma]\!] \to [\![B]\!]$ 的函数, 这个构造应该是显然的.

然后是

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

这时 $[\![\Gamma, x : A \vdash M : B]\!]$ 是 $[\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$ 的函数, 我们需要的 $[\![\Gamma \vdash \lambda x.\, M : A \to B]\!]$ 则应该是 $[\![\Gamma]\!] \to ([\![A]\!] \to [\![B]\!])$, 从前者得到后者的构造同样显然.

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 (部分被省略) 的构造, $[\![\Gamma, x : A \vdash x : A]\!] = (\gamma, x) \mapsto x$, $[\![\Gamma, x : A \vdash f\ x]\!] = (\gamma, x) \mapsto f(\gamma)(x)$, 因此由 (集合论) 函数的外延相等, $[\![\Gamma \vdash \lambda x.\, f\ x : A \to B]\!] = \gamma \mapsto (x \mapsto f(\gamma)(x)) = \gamma \mapsto f(\gamma) = f = [\![\Gamma \vdash f : A \to B]\!]$.

另外我们还有

定理 3.7.1: **(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 3.7.1: 任何语境下都不存在类型为 $\perp$ 的项.

证明: 这样的项在集合模型中是值域为 $\emptyset$ 的函数.

推论 3.7.2: 任何语境下都有 $\mathrm{yes} \not\equiv \mathrm{no} : \mathrm{Ans}$.

证明: 假设存在 Γ 使 $\mathrm{yes} \equiv \mathrm{no}$, 那么 $[\![\Gamma \vdash \mathrm{yes}]\!] = [\![\Gamma \vdash \mathrm{no}]\!]$, 然而前者是值为 1 的常函数, 后者是值为 0 的常函数, 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 3.7.2**: 由图表 $\bot, 1 \to \top, 1 \rightrightarrows \mathrm{Ans}$ 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.

4. 扩展 STLC

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