

依赖于类型的类型

读书笔记

许博

本章将介绍另一个扩充 $\lambda \rightarrow$ 的系统 λ_{ω} ，而非在 λ_2 上继续扩充的系统。

1 疑问

在 4.7 变换规则中：

(conv) 如果 $B =_{\beta} B'$ ，则 $\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$ 。

其中 $B' : s$ ，但是 λ_{ω} 中并没有对类 (kind) 的抽象，如 $\lambda \kappa : \square. M$ ，也没有相对应用，换句话说，当 $s \equiv \square$ 时，所有的 B 与 B' 应当是恒等的（不包括 β -等价），这样理解是否正确？

2 类型构造子

上一章中引入了一般性的 (generalised) 项，通过从类型变量抽象项。比如作用于确定类型 σ 的恒等函数 $\lambda x : \sigma. x$ 可以被泛化成项 $\lambda \alpha : *. \lambda x : \alpha. x$ ，多态的 (polymorphic) 恒等函数，抽象于 α 。

通过类似的方式也可以构造一般性的类型。比如形如 $\beta \rightarrow \beta, \gamma \rightarrow \gamma, \dots$ ，等具有结构 $\diamond \rightarrow \diamond$ 的类型，其中箭头的左边和右边是一样的类型。

为了处理这种情况，引入一个包含了这种结构的本质 (essence) 的一般性的表达式： $\lambda \alpha : *. \alpha \rightarrow \alpha$ 。这个表达式本身并不是一个类型，而是将类型当作值的一个函数。称之为类型构造子 (type constructor)。只有当喂给它类型（比如 β, γ ）时我们可以得到类型：

$$(\lambda\alpha : *. \alpha \rightarrow \alpha)\beta \rightarrow_{\beta} \beta \rightarrow \beta, (\lambda\alpha : *. \alpha \rightarrow \alpha)\gamma \rightarrow_{\beta} \gamma \rightarrow \gamma.$$

我们由类型 α 抽象出类型 $\alpha \rightarrow \alpha$ ，来获得类型构造子 $\lambda\alpha : *. \alpha \rightarrow \alpha$ 。类似的，还可以构造出更复杂的类型构造子，比如 $\lambda\alpha : *. \lambda\beta : *. \alpha \rightarrow \beta$ 。

而一个显然的问题是：类型构造子的类型是什么？我们可以将 $\lambda\alpha : *. \alpha \rightarrow \alpha$ 看作是一个将类型 α 映射到类型 $\alpha \rightarrow \alpha$ 的函数，因为 $\alpha : *$ 且 $\alpha \rightarrow \alpha : *$ ，我们可以得到： $\lambda\alpha : *. \alpha \rightarrow \alpha : * \rightarrow *$ 。

因此在 $*$ 之后，需要一个新的“超级类型 (super-type)”，即 $* \rightarrow *$ 。

类似的，我们可以得到： $\lambda\alpha : *. \lambda\beta : *. \alpha \rightarrow \beta : * \rightarrow (* \rightarrow *)$ 。

需要注意的是，在上一章中，提到了 $*$ 是所有 ($\mathbb{T}2$) 类型的类型，而 $*, * \rightarrow *, \dots$ 等类型不属于 $\mathbb{T}2$ ，因此它们的类型也不是 $*$ 。以及在 $\lambda\alpha : *. \alpha \rightarrow \alpha$ 中， $\alpha \rightarrow \alpha$ 的类型是 $*$ 而不是 $* \rightarrow *$ ，是因为 $\alpha \rightarrow \alpha$ 是一个接收类型为 α 的输入值，返回类型为 α 的输出值的函数的类型，而 $* \rightarrow *$ 是一个接收类型，返回类型的函数的类型。

上述扩展称作依赖于类型的类型 (types depending on types)，扩展后的系统记为 $\lambda\omega$ 。

所有的超级类型，单独的 $*$ 以及箭头分割的若干 $*$ 符号，称为类 (kind)，所有类的集合 \mathbb{K} 的抽象定义为：

$$\mathbb{K} = * | (\mathbb{K} \rightarrow \mathbb{K}).$$

而所有类的类型我们使用符号 \square 表示，有且仅有一个的超级超级类型 (super-super-type)。也即有 $* : \square, * \rightarrow : \square, \dots$ 。

如果 κ 是一个类，则对于每个类型是 κ 的 M (也即 $M : \kappa$)， M 被称作是一个类型构造子，简称为构造子。而之前的类型，比如 α 或者 $\alpha \rightarrow \alpha$ 也都是构造子，尽管它们什么也没有构造。

我们使用术语 (term) 真构造子 (proper constructor) 表示不是类型的构造子 (即类型不是 $*$ 的构造子)。因此构造子的集合被分为了 (旧的) 类型和真构造子。

最后，使用类别 (sort) 或符号 s 表示 $*$ 或 \square (我认为 s 其实是代表任意类型的类型或任意构造子类型的类型)：

定义 2.1 (构造子，真构造子，类别)

(1) 如果 $\kappa : \square$ 且 $M : \kappa$ ，则 M 是一个构造子，如果 $\kappa \neq *$ ，则 M 是一个真构造子。

(2) 类别 (s) 的集合为 $\{*, \square\}$ 。

随着 \square 的引入，我们的语法中有四个层级 (level)：

定义 2.2 (层级, *level*)

- 第 1 层: 项 (*terms*);
- 第 2 层: 构造子 (包括类型和真构造子);
- 第 3 层: 类 (*kinds*);
- 第 4 层: 超级超级类型 \square 。

关于这里的真构造子和类型在同一层, 我的理解是, 因为真构造子其实就是依赖于类型的类型, 正如之前依赖于项的项和项是一个层级的, 依赖于类型的类型故也处在类型所在的层级。

对于语句 $A : B$, 可以得出 B 所处的层级一定比 A 高一级, 比如当 A 是一个项时, B 是一个类型, 或者 A 是一个类型时, $B \equiv *$ 。

3 λ_{ω} 中的类别规则和变量规则, sort-rule and var-rule in λ_{ω}

本章中描述的系统叫做 λ_{ω} , 它是 $\lambda \rightarrow$ 的另一个扩展:

- $\lambda_2 = \lambda \rightarrow$ 加 依赖于类型的项,
- $\lambda_{\omega} = \lambda \rightarrow$ 加 依赖于类型的类型。

给出 λ_{ω} 的具体推导规则。

首先形式化 $*$ 的类型是 \square , 这个规则称为类别规则 (sort-rule):

定义 3.1 (类别规则, *sort-rule*)

(类别, *sort*) $\emptyset \vdash * : \square$

为了确定给定的上下文中所有的声明都是可推导的, 在 λ_2 和 λ_{ω} 中使用变量规则 ((var)-rule) 推导。但是在 λ_{ω} 中, 我们用有一点不同的方式: 巧妙地将上下文声明的可推导性与构造合适的上下文相结合。

原因是 λ_{ω} 中的类型更为复杂, 所以必须保证类型的定义是良构的 (well-formed)。在 $\lambda \rightarrow$ 中, 合法类型的集合已经预先给出, 所以没有问题, 而在 λ_2 中, 必须确定一个 (合适的) λ_2 -上下文, 这个上下文也提供了在其中使用到的类型需要的条件 (requirements), 见定义 3.4.4(3), ρ 中出现的所有自由类型变量需要在上下文中声明, 此时才可以推定 ρ 的良构与否。因此, 与 $\lambda \rightarrow$ 不同, λ_2 中出现在一个推定 (judgement) 中的类型的合法性不再能通过引用外部的集合来判定, 但是应该依赖于包括其上下文的自身推定的一个检查。

在现在的系统中，类型需要的条件更加严格：出现在一个推定中的类型的合法性只取决于（follow）我们是否可以形式化地推导出它。

这个方式是：如果类型 A 已经是合法的，我们只用一个声明 $x : A$ 扩展一个上下文。并且一个语句中的合法类型位于第 2 层或第 3 层，也即是一个类型（因为构造子不是类型，不能出现在 $:$ 的右边）或是一个类。可以通过一个规则来表示：

定义 3.2 (变量规则, *var-rule*)

(变量, *var*) 如果 $x \notin \Gamma$, 则 $\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$ 。

需要注意的是，在这条规则中可以看到， x 的类型不能是 \square ，是因为目前为止，该系统不允许应用时右边是 \square 类型的类 (kind)，因此也不存在类型是 \square 的绑定变量。

s 表示 $*$ 或 \square ，因此 A 是一个类型或一个类 (kind)， x 因此也表示一个项变量或是一个类型变量。这个规则允许我们以一个声明 $x : A$ 扩展上下文 Γ ，并且在扩展的上下文中推导出与声明一样的语句。

$x \notin \Gamma$ 保证了变量 x 未在 Γ 中出现，因此声明在一个上下文中的所有变量都是不同的，避免了变量名相同（类型不同）时造成的混淆。

使用 (sort) 和 (var) 规则的一个例子，观察它们如何工作：

$$\begin{array}{l} (1) \emptyset \vdash * : \square \\ \frac{(1) \emptyset \vdash * : \square}{(2) \alpha : * \vdash \alpha : *} (var) \\ \frac{(2) \alpha : * \vdash \alpha : *}{(3) \alpha : *, x : \alpha \vdash x : \alpha} (var) \end{array}$$

第 (1) 行由 (sort) 规则推导而出，(1)-(2) 以及 (2)-(3) 都应用了 (var) 规则。同时可以清楚地发现，新的 (var) 规则没有 $\lambda \rightarrow$ 中的 (var) 规则通用，因为当前的 (var) 规则只允许推导出上下文中新添加的最后的声明 $x : A$ ，而 $\lambda \rightarrow$ 中，任意出现在 Γ 的声明 $x : \sigma$ 都是可推导的。

如在 $\lambda \rightarrow$ 中一样，我们也希望在 λ_{ω} 中可以推导 $\alpha : *, x : \alpha \vdash \alpha : *$, $\alpha : *, \beta : * \vdash \alpha : *$ 以及 $\alpha : *, \beta : * \vdash \beta : *$ 。但目前尚不可行，因为缺少 (var) 规则需要的 **premiss**，需要注意的是，尽管 $\alpha : *, \beta : * \vdash \beta : *$ 中 $\beta : *$ 是上下文中最后一个声明，但我们不能得到 **premiss**:

$$\alpha : * \vdash * : \square.$$

因为前边所提到的 (sort) 规则，只给出了空上下文时的规则。

为了解决这个问题，引入了所谓的弱化规则 (weakening rule)。

4 λ_{ω} 中的弱化规则 (weakening rule)

弱化规则允许我们通过添加新的声明来弱化一个推定 (judgement) 的上下文。

定义 4.1 (弱化规则, *weakening rule*)

$$(weak) \text{ 如果 } x \notin \Gamma, \text{ 则 } \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}.$$

若上下文 Γ 已经可以推导出 $A : B$, 在 Γ 的尾部添加了一个任意的声明 (弱化) 后仍然可以推导出 $A : B$ 。

需要注意的是第二个 **premiss**, 添加的声明的类型需要是可推导出的类型或类 (kind), 与 (var) 规则类似。

在添加了弱化规则以后, 前边所提到的稀疏引理 (Thinning Lemma) 在 λ_{ω} 中依然成立, 即子上下文可以推定出来的语句在扩展后的上下文中依然可以推定, 尽管弱化规则只允许在尾部添加声明。

现在便可以推导 $\alpha : *, x : \alpha \vdash \alpha : *$, 推导树:

$$\frac{\frac{(1) \emptyset \vdash * : \square}{(2) \alpha : * \vdash \alpha : *} (var) \quad \frac{(1) \emptyset \vdash * : \square}{(2) \alpha : * \vdash \alpha : *} (var)}{(3) \alpha : *, x : \alpha \vdash \alpha : *} (weak)$$

5 λ_{ω} 中的形成规则 (formation rule)

在 λ_2 中, 有一个叫做 (form) 的形成规则, 用于在一个上下文中类型化语句的构造, 这个规则基于 λ_2 -类型的集合 \mathbb{T}_2 , 正如之前提到的, λ_{ω} 中的类型更为复杂, 因此, 引入了一个“真正”的推导规则, 包含 **premisses** 和 **conclusion**, 用于类型 (以及类 (kind)) 的构造:

定义 5.1 (形成规则, *formation rule*)

$$(form) \frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A \rightarrow B : s}$$

需要注意的是, 规则中出现的三个 s 是相同的, 同时表示 $*$ 或 \square 。

在引入新的 (form) 规则之前, 由 (var) 规则与 (weak) 规则只能推导出 $* : \square, \alpha : *, x : \alpha$ 而不能推出箭头类型 $* \rightarrow * : \square$ 以及 $\alpha \rightarrow \beta : *$ 等。引入

新的形成规则可以覆盖我们需要的所有类型和类 (kind)。

$$\text{例如: } \frac{\emptyset \vdash * : \square \quad \emptyset \vdash * : \square}{\emptyset \vdash * \rightarrow * : \square} (form)$$

6 λ_{ω} 中的应用与抽象规则

λ_{ω} 中的这两个规则与之前略有不同, 首先, 用于类型的元变量 (meta-variable) 的名字不同, 因为 λ_{ω} 中的类型更为通用, 其次, 需要保证类型是良构的 (即可以由上下文推导出):

定义 6.1 (应用规则)

$$(appl) \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

定义 6.2 (抽象规则)

$$(abst) \frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash A \rightarrow B : s}{\Gamma \vdash \lambda x : A. M : A \rightarrow B}$$

需要注意的是, 因为 $s \in \{*, \square\}$, 所以这些规则 (包括之前定义的) 都同时具有两种作用, 比如当 $s \equiv *$ 时, $A \rightarrow B$ 是一个第二层的类型, 如 $(\alpha \rightarrow \beta) \rightarrow \gamma$, 当 $s \equiv \square$ 时, $A \rightarrow B$ 就是一个第三层的类型 (或者说类, kind), 如 $(* \rightarrow *) \rightarrow *$ 。

7 简化 (shortened) 推导

为了保证推导系统的严谨性以及完整性所定义的这些规则, 导致在推导的过程中有许多很无趣以及显而易见的部分, 比如在推导 $\alpha : *, \beta : * \vdash \alpha \rightarrow \beta : *$ 时, 需要下面这些推定:

$$\begin{aligned} & \emptyset \vdash * : \square \text{ (sort),} \\ & \alpha : * \vdash \alpha : * \text{ (var),} \\ & \alpha : * \vdash * : \square \text{ (weak),} \\ & \alpha : *, \beta : * \vdash \alpha : * \text{ (weak),} \\ & \alpha : *, \beta : * \vdash \beta : * \text{ (var).} \end{aligned}$$

而上述推定包括 $\alpha : *, \beta : * \vdash \alpha \rightarrow \beta : *$ 都是非常显而易见的。这里不是很有趣的步骤出现在 (尤其是) 下面三个情况下:

- (i) 当使用规则 (sort),(var) 和 (weak) 时,
- (ii) 当使用规则 (form) 时, 以及
- (iii) 当确定 (abst) 规则的第二个 **premiss** 的合法性时。

为了将注意力放在真正有趣的步骤上, 我们将允许跳过如上的所有推定, 或者只确定某个类型的良构与否。因此 $\alpha : *, \beta : * \vdash \alpha \rightarrow \beta : *$ 现在可以直接使用。

8 变换 (conversion) 规则

变换规则的定义如下:

定义 8.1 (变换规则, *conversion rule*)

$$(conv) \text{ 如果 } B =_{\beta} B', \text{ 则 } \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}.$$

需要注意的是, 因为 B 是作为推定 $\Gamma \vdash A : B$ 中的一个类型, 所以 B 已经是良构的。为了保证 B' 也是良构的, 添加了第二个 **premiss**: $\Gamma \vdash B' : s$, 保证了 B' 也是良构的类型或类 (kind)。

需要注意的是, β -规约不保证类型匹配, 换言之, $B =_{\beta} B'$ 无法保证 B 是良构时 B' 是良构的, 比如 $\beta \rightarrow \gamma =_{\beta} (\lambda \alpha : *. \beta \rightarrow \gamma)M$, 右边在进行 β -规约时, 并不检查 M 的类型, 而只是进行符号的替换, 当 M 的类型不是 $*$ 时, 右边即不是良构的。

在拥有了变换规则之后, 我们可以进行如下推导, 其中令 $\Gamma \equiv \beta : *, x :$
 $(\lambda \alpha : *. \alpha \rightarrow \alpha)\beta :$

$$\frac{\Gamma \vdash x : (\lambda \alpha : *. \alpha \rightarrow \alpha)\beta \quad \Gamma \vdash \beta \rightarrow \beta : *}{\Gamma \vdash x : \beta \rightarrow \beta} (conv)$$

需要注意的是, 推定中对象 (subject), 即 $A : B$ 中的 A 进行规约后的类型不变, 且仍可被推导出, 而这一定理可以由之前的规则推出, 不再赘述。

至此, 所有的 $\lambda\omega$ -规则如下:

(sort)	$\emptyset \vdash * : \square$
(var)	如果 $x \notin \Gamma$, 则 $\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$
(weak)	如果 $x \notin \Gamma$, 则 $\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$
(form)	$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A \rightarrow B : s}$
(appl)	$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$
(abst)	$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash A \rightarrow B : s}{\Gamma \vdash \lambda x : A. M : A \rightarrow B}$
(conv)	如果 $B =_{\beta} B'$, 则 $\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$

9 λ_{ω} 的性质

λ_{ω} 满足之前几章所提到的大部分性质，但是类型唯一性引理需要进行调整：

引理 9.1 (类型唯一性引理)

如果 $\Gamma \vdash A : B_1$ 且 $\Gamma \vdash A : B_2$, 则 $B_1 =_{\beta} B_2$ 。