arXiv:2603.23652v1 [cs.LO] 24 Mar 2026

基于 Agda 对带类型 Top 的系统 I 进行形式化

Agustín Séttimo* Cristian Sottile† Cecilia Manzino‡

摘要

系统 I 是一种近期引入的、带有对的简单类型 λ 演算，其中同构类型被视为相等。本文提出了带有类型 Top 的系统 I 的一个变体，并在 Agda 中对该演算进行了完整的形式化，其中包括进展性和强正规化的证明。

关键词： λ 演算、类型同构、Agda、强正规化

1 引言

类型同构的研究是一个新兴的研究领域，有着多种应用。从编程语言的角度来看，如果我们将同构类型视为相同，就可以把类型相同、语法不同但语义等价的程序视为相同。这使一种新颖的编程方式成为可能；例如，由于类型$A \to B \to C$ 和 $B \to A \to C$ 同构，函数可以按任意顺序接收其参数。另一方面，从证明系统的角度来看，既然类型可以被视为命题，程序则对应于证明，那么将同构命题的不同证明视为同一个证明，就意味着我们实现了某种形式的证明无关性。关于此类应用的深入讨论，请参阅 Díaz-Caro 和 Dowek 的奠基性论文 λ演算模同构的引言 [5]。

这些模同构系统大多以简单类型λ演算为框架，并对其进行扩展，使同构类型被视为相同。其中最早的系统，也是与本文最相关的系统，是系统 I [5]：一种带有对、模同构的简单类型λ演算。该演算的早期版本已在 Haskell 中实现了解释器 [7], ，此外还提出了带有 η展开规则的变体 [6], 、同构更少且没有类型系统的变体 [2], ，以及具有多态性的变体 [12] 。

本工作的主要贡献如下：

1. 一种系统 I 的变体，将类型 Top 作为基类型，并作了相应调整，使其更适合形式化。要在模同构意义下向演算添加新的类型构造子，就必须扩展类型上的等价关系，纳入涉及新构造子的同构，以及同构类型之间的项等价关系。选择这些项同构时，也以实现可正规化的演算为目标。

*计算机科学系，罗萨里奥国立大学，阿根廷罗萨里奥。agustinsettimo.ips@gmail.com

†科学与技术系，基尔梅斯国立大学，阿根廷布宜诺斯艾利斯省贝尔纳尔。计算机科学研究所（ICC），CONICET / 布宜诺斯艾利斯大学，阿根廷布宜诺斯艾利斯。csottile@dc.uba.ar

‡计算机科学系，罗萨里奥国立大学，阿根廷罗萨里奥。ceciliam@fceia.unr.edu.ar

2. 在 Agda 中对该演算进行形式化，定义其语法、语义和类型规则。我们在表示带类型项和符号记法时所采用的方法，与 [14] 中的方法类似。3. 在 Agda 中证明进展性和强正规化。在我们的设定中，强正规化意味着从一个带类型项出发的每个归约序列都是有限的。

完整的 Agda 代码可在 GitHub 仓库 https://github.com/AgusSett/ thesis 中获取。

本文结构如下。第 2 节概述若干模同构演算。第 3 节为系统 I 扩展了类型 Top。第 4 节介绍扩展演算的 Agda 形式化，并证明进展性。第 5 节证明强正规化性质。第 6 节实现求值函数。第 7 节总结全文。第 8 节介绍可能的未来工作。

2 模同构的 STLC

本节介绍系统 I [5], ，它是我们随后扩展并形式化的基础系统。类型 A 和 B 同构（记作 $\equiv$），当且仅当存在两个函数f：$A \to B$ 和 g：$B \to A$，使得 $f \circ g = id_B$ 和 $g \circ f = id_A$。

Di Cosmo [4]刻画了这些系统中出现的所有类型同构，并将其归类为公理集合。例如，对应于简单类型λ演算的集合由称为交换的同构（$A \to (B \to C) \equiv B \to (A \to C)$）构成。在带笛卡尔积的简单类型λ演算（记作 $\times$）中，所有同构都可以由以下四个同构和同余规则推导得出：

$$A \times B \equiv B \times A \qquad (\textsc{comm})$$
$$A \times (B \times C) \equiv (A \times B) \times C \qquad (\textsc{asso})$$
$$A \to (B \times C) \equiv (A \to B) \times (A \to C) \qquad (\textsc{dist})$$
$$(A \times B) \to C \equiv A \to B \to C \qquad (\textsc{curry})$$

表 1：系统 I 中的类型等价

我们注意到，未包含交换同构，因为它可以通过柯里化和交换律构造。

系统 I 的类型语法与传统 λ 演算相同，而其项语法仅在投影上有所不同：投影由类型而非位置参数化：

$$A \quad := \quad \tau \mid A \to A \mid A \times A$$
$$r \quad := \quad x \mid \lambda x^A.r \mid rr \mid \langle r, r \rangle \mid \pi_A(r)$$

相较于带类型 λ 演算，这种演算的类型系统扩展了一条类型规则，用以断言：如果项 t 具有类型 A 和 $A \equiv B$，那么 t 也具有类型 B:

$$\frac{\Gamma \vdash r : A \qquad A \equiv B}{\Gamma \vdash r : B} (\equiv)$$

$$\langle r, s\rangle \rightleftarrows \langle s, r\rangle \qquad (\textsc{comm})$$
$$\langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle \qquad (\textsc{asso})$$
$$\lambda x^A.\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s\rangle \qquad (\textsc{dist}_\lambda)$$
$$\langle r, s\rangle t \rightleftarrows \langle rt, st\rangle \qquad (\textsc{dist}_{\text{app}})$$
$$r\langle s, t\rangle \rightleftarrows rst \qquad (\textsc{curry})$$

$$\frac{r \rightleftarrows s}{\lambda x^A.r \rightleftarrows \lambda x^A.s} \quad \frac{r \rightleftarrows s}{rt \rightleftarrows st} \quad \frac{r \rightleftarrows s}{tr \rightleftarrows ts}$$
$$\frac{r \rightleftarrows s}{\langle t, r\rangle \rightleftarrows \langle t, s\rangle} \quad \frac{r \rightleftarrows s}{\langle r, t\rangle \rightleftarrows \langle s, t\rangle} \quad \frac{r \rightleftarrows s}{\pi_A(r) \rightleftarrows \pi_A(s)}$$

表 2：系统 I 中项的等价

这条规则与刻画该演算的四个同构共同导出了一些项之间的等价关系。例如，由于类型 $A \times B$ 和 $B \times A$ 等价，项 $\langle r,\ s\rangle$和 $\langle s,\ r\rangle$（其中 $r : A$ 且 $s : B$）必定表示同一个项。因此，有必要将项层面同构所带来的结果纳入归约关系。为此，定义项之间的等价关系，记作 $\rightleftarrows$，并列于表 2。关系 $\rightleftarrows^*$ 是其自反传递闭包。

系统 I 的操作语义由两个关系定义：项之间的等价关系，以及一种类似于经典 β-归约的归约关系。这里我们将说明，为什么不能直接将 β-归约与表 2 中的规则结合使用。

通常的投影规则通过元素的位置访问一个对，但 ($\textsc{comm}$) 同构允许改变对中元素的顺序，因此可以投影出两个元素中的任意一个：

$\pi_1\langle r, s\rangle \hookrightarrow r$
$\pi_1\langle r, s\rangle \rightleftarrows \pi_1\langle s, r\rangle \hookrightarrow s$

这会导致类型保持出现问题，也会引入非确定性。解决方法是通过对中元素的类型来访问该元素，因此定义了一条新规则：若 $\Gamma \vdash r : A$，则 $\pi^A\langle r,\ s\rangle \hookrightarrow r$。这条规则解决了类型保持问题，但演算中仍存在非确定性。不过，即使两个项的类型相同，也可以编码出确定性投影。因此，系统 I 的非确定性被视为一种特性，而非问题。

另一个与经典 β-归约及项的等式规则发生冲突的情况，是由同构 $A \to B \to C \equiv B \to A \to C$ 导致的；它允许我们改变向函数传递参数的顺序。

为解决这种情况下的类型保持问题，β-归约作了修改，要求参数与形参具有相同的类型：若 $\Gamma \vdash s : A$，则 $(\lambda x^A.r)s \hookrightarrow r[s/x]$。

最后，归约关系 $\rightsquigarrow$ 定义为模 $\rightleftarrows$ 的关系 $\hookrightarrow$（即 $\rightsquigarrow := \rightleftarrows^* \circ \hookrightarrow \circ \rightleftarrows^*$），而 $\rightsquigarrow^*$ 是其自反传递闭包。

例如，考虑此演算中一个有类型的项的以下归约：

$(\lambda x^A.\lambda y^B.r)z^B w^A \rightleftarrows^* (\lambda x^A.\lambda y^B.r)w^A z^B \hookrightarrow_\beta$
$(\lambda y^B.r[w^A/x^A])z^B \hookrightarrow_\beta r[w^A/x^A, z^B/y^B]$

3 添加 Top 并转向显式性

本节将类型 Top 引入系统 I，并对其进行调整，使其更适合形式化：通过为等价关系指定方向，从而得到重写关系。我们还对同构类型的相应项所选取的隐式见证作了一些修改，以确保进展性并保持强正规化。

类型语法通过将 $\top$ 用作基类型进行修改：

$$A \quad := \quad \top \mid A \to A \mid A \times A$$

允许导出此演算中任意同构的类型同构集合如下：在系统 I 的同构基础上，添加与 $\top$ 相关的三个同构。

$$A \times B \equiv B \times A \qquad (\textsc{comm})$$
$$A \times (B \times C) \equiv (A \times B) \times C \qquad (\textsc{asso})$$
$$A \to (B \times C) \equiv (A \to B) \times (A \to C) \qquad (\textsc{dist})$$
$$(A \times B) \to C \equiv A \to B \to C \qquad (\textsc{curry})$$
$$A \times \top \equiv A \qquad (\textsc{id-}\times)$$
$$\top \to A \equiv A \qquad (\textsc{id-}\to)$$
$$A \to \top \equiv \top \qquad (\textsc{abs})$$

由于形式化过程中的调整需要在构造项时对每个应用的同构进行语法跟踪，我们还明确加入以下类型同构关系的相容规则：

$$\frac{}{A \equiv A}(\textsc{refl}) \quad \frac{A \equiv B}{B \equiv A}(\textsc{sym}) \quad \frac{A \equiv B \quad B \equiv C}{A \equiv C}(\textsc{trans})$$

$$\frac{A \equiv B}{A \times C \equiv B \times C}(\textsc{cong}\times_1) \quad \frac{A \equiv B}{C \times A \equiv C \times B}(\textsc{cong}\times_2)$$

$$\frac{A \equiv B}{A \to C \equiv B \to C}(\textsc{cong}\to_1) \quad \frac{A \equiv B}{C \to A \equiv C \to B}(\textsc{cong}\to_2)$$

在逻辑中，类型 $\top$ 对应于命题 True。同构 (id-$\times$) 对应于 True 相对于合取的中性性质，而 (id-$\to$) 和 (abs) 则对应于 True 相对于蕴含的中性和吸收性质。

项的语法扩展了两个构造子：类型 $\top$的值 $\star$，以及一个用于显式见证同构应用的语法构造子。这意味着，为了应用转换规则（$\equiv$），即 在 $A \equiv B$时将类型 B 赋予类型为 A 的项 t，需要提供一个确立类型等价的同构见证。见证基于等价关系，因此遵循以下语法：

$$\begin{aligned}\rho \quad := \quad & \textsc{comm} \mid \textsc{asso} \mid \textsc{dist} \mid \textsc{curry} \mid \textsc{id-}\times \mid \textsc{id-}\to \mid \textsc{abs} \\ & \mid \textsc{cong}\times_1\ \rho \mid \textsc{cong}\times_2\ \rho \mid \textsc{cong}\to_1\ \rho \mid \textsc{cong}\to_2\ \rho \\ & \mid \textsc{sym}\ \rho \mid \textsc{trans}\ \rho\ \rho\end{aligned}$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ (ax) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B}\ (\to_i)$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash ts : B}\ (\to_e)$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B}{\Gamma \vdash \langle t, s\rangle : A \times B}\ (\times_i) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_A(t) : A}\ (\times_e)$$

$$\frac{}{\Gamma \vdash \star : \top}\ (\top_i) \qquad \frac{A \equiv_\rho B \qquad \Gamma \vdash t : A}{\Gamma \vdash [\rho]{\equiv} t : B}\ (\equiv)$$

图 1：类型规则

项的语法如下：

$$t \quad := \quad \star \mid x \mid \lambda x^A.t \mid tt \mid \langle t, t\rangle \mid \pi_A(t) \mid [\rho]{\equiv} t$$

其中 $x \in \mathcal{V}$ 是无限变量的集合。

该类型系统比系统 I 多一条规则：它为项 $\star$ 赋予类型 $\top$，并修改了类型同构的规则（$\equiv$），以便对构造子 $[\rho]{\equiv} t$ 进行类型判断。该演算的类型规则见图 1。

请注意，在（$\equiv$）中，为了关联 A 和 B 而应用的等价规则，我们在 $\equiv$ 中将其标记为下标 ρ。一个重要的观察是，由于同构 $\top \to \top \equiv \top$，项 $\Omega = (\lambda x^\top.xx)\ (\lambda x^\top.xx)$ 在该演算中具有类型。在图 2 中，我们展示了它的一种可能的类型推导。

$\boldsymbol{\delta} =$

$$\dfrac{\dfrac{\top \equiv_{\text{SYM ABS}} \top \to \top \qquad \dfrac{}{x : \top \vdash x : \top}\ (ax)}{x : \top \vdash [_{\text{SYM ABS}}]{\equiv} x : \top \to \top}\ (\equiv) \qquad \dfrac{}{x : \top \vdash x : \top}\ (ax)}{\dfrac{x : \top \vdash xx : \top}{\vdash \lambda x.xx : \top \to \top}\ (\to_i)}\ (\to_e)$$

$$\dfrac{\boldsymbol{\delta} \qquad \dfrac{\top \to \top \equiv_{\text{SYM}} \top \qquad \boldsymbol{\delta}}{\vdash [_{\text{SYM}}]{\equiv} \lambda x.xx : \top}\ (\equiv)}{\vdash (\lambda x.xx)(\lambda x.xx) : \top}\ (\to_e)$$

图 2：Ω 的类型推导

自然会产生这样一个问题：在存在此类项的情况下，该演算是否仍能保持强正规化。我们采用的方法是，在应用于项的类型同构中携带见证，从而保持终止性。也可以采用其他方法。关键的观察是，我们处理的是 δ 的一个特例，它接受一个 $\top$。例如，在不带见证的变体中，可以添加一条归约规则 $(\lambda x^A.t)\ s \hookrightarrow t[\star/x]$，条件是 $A \equiv \top$。在第 6 节中，我们将以求值函数为例，展示 Ω 在我们的系统中如何归约。

下一步是根据类型同构来定义项之间的关系。为简便起见，我们将同构类型中彼此对应的项称为“项同构”。该定义

包含两项重要的变化。第一项是外延性的，即 包括由涉及 $\top$ 的同构关联起来的项。第二项是结构性的，与我们采用的显式化有关，其目的是使演算更贴近形式化；具体做法是复制规则并为其指定方向，以便分解同构见证。这对项同构施加了一项限制：在系统 I 中，项等价可以任意应用；而在这里，项同构只能按照项中构造子 $[]\equiv$所附带的标签来应用。请注意，尽管我们使用的是重写关系而非等价关系，为保持一致，仍保留双箭头。

表 3 概述了选定的项同构，其中省略了它们精确的对称形式，以及
con与系统 I 共用的合流规则（表 2）。

选择这些项同构时，主要遵循两个目标：保持强正规化，并确保进展性。根据类型构造子的引入和消去可能的组合，选择等价关系有多种方式；例如，$\pi_B(rs) \rightleftarrows (\pi_{A\to B}r)s$ 涉及两个类型构造子的消去，而$\langle \lambda x.r,\ \lambda x.s\rangle \rightleftarrows \lambda x.\langle r,\ s\rangle$ 则涉及它们的引入。在这种情况下，我们采用了全引入策略。涉及消去的等价关系与 η-展开结合使用，并纳入组合规则，以确保没有项会陷入停滞。此外，由于柯里同构现在基于引入而非消去，因此必须通过特殊规则 T-SUBST 单独处理参数的重新排列。

因此，我们纳入的等价并非总是类型同构的直接推论。表格中的前五项是与$\to$ 和 $\times$相关的类型同构的推论，即 交换律、结合律、分配律和柯里化。接下来的五项是与 $\top$相关的新加入类型同构的推论，即 id-$\times$、id-$\to$和 abs。最后五项则是为了避免项卡住而必需的。

为进一步说明这种选择，请考虑 η-展开和分裂规则（$r \rightleftarrows \langle \pi_A r,\ \pi_B r\rangle$其中 $r : A \times B$），它们以某种形式是避免某些项卡住所必需的。但是，如果直接纳入它们，规范化性质就会丧失，因为例如，η-展开可以无限次应用。我们采用的解决方案与 [7], 中使用的方案类似：我们定义新的构造子来嵌入这些规则，例如SPLIT-ASSO和$\text{SPLIT-DIST}_\lambda$ ，以此嵌入分裂规则：

$\langle r, s\rangle \rightleftarrows_{\text{SPLIT}} \langle r, \langle \pi_B s, \pi_C s\rangle\rangle \rightleftarrows_{\text{ASSO}_i} \langle\langle r, \pi_B s\rangle\ \pi_C s\rangle$
$\lambda x^A.r \rightleftarrows_{\text{SPLIT}} \lambda x^A.\langle \pi_B r, \pi_C r\rangle \rightleftarrows_{\text{DIST}_\lambda} \langle \lambda x^A.\pi_B r, \lambda x^A.\pi_C r\rangle$
第一条规则用于尝试将结合律应用于一个对，其中第二分量项的形式不是一个对。第二条规则则例如用于尝试将投影应用于一个抽象，而其主体的形式不是一个对。同样，规则η-DISTapp 和 η-CURRY嵌入了 η-扩展。

该演算的操作语义定义为 $\rightsquigarrow := \rightleftarrows \cup \hookrightarrow$。我们将其自反传递闭包记作 $\rightsquigarrow^*$。如本节开头所述，$\rightsquigarrow$ 的定义并不完全与系统 I 中相同，也就是说，$\hookrightarrow$ 模 $\rightleftarrows$。相反，我们将其定义为 $\hookrightarrow$与关系 $\rightleftarrows$的扩展，因此，同构类型对应的项现在构成一个归约步骤。

4 形式化

本节介绍使用 Agda 对一种扩展了类型同构的简单类型λ演算进行形式化，该演算包含对和 Top。

本工作中展示的大部分代码都基于 [14]。改编主要包括添加类型同构和演算的语义，并扩展 β-归约，使其将项等价（即 $\rightleftarrows$）也纳入可能的归约步骤。

$$[\textsc{comm}] \equiv \langle t, s\rangle \rightleftarrows \langle s, t\rangle \qquad (\textsc{comm}_i)$$
$$[\textsc{asso}] \equiv \langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle \qquad (\textsc{asso}_i)$$
$$[\textsc{dist}] \equiv \lambda x^A.\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s\rangle \qquad (\textsc{dist}_\lambda)$$
$$[\textsc{curry}] \equiv \lambda x^A.\lambda y^B.t \rightleftarrows \lambda z^{A\times B}.t[\pi_A z/x, \pi_B z/y] \qquad (\textsc{curry})$$
$$[\textsc{sym curry}] \equiv \lambda x^{A\times B}.t \rightleftarrows \lambda y^A.\lambda z^B.t[\langle y, z\rangle/x] \qquad (\textsc{uncurry})$$

$$\text{If } \Gamma \vdash t : \top,\ [\textsc{id-}\times] \equiv \langle s, t\rangle \rightleftarrows s \qquad (\textsc{id}_{\times_e})$$
$$[\textsc{sym id-}\times] \equiv t \rightleftarrows \langle t, \star\rangle \qquad (\textsc{id}_{\times_i})$$
$$\text{If } \Gamma \vdash t : \top,\ [\textsc{sym abs}] \equiv t \rightleftarrows \lambda x^A.t \qquad (\textsc{abs}_\lambda)$$
$$\text{If } \Gamma \vdash t : A \to \top,\ [\textsc{abs}] \equiv t \rightleftarrows \star \qquad (\textsc{abs}_\top)$$
$$\text{If } \Gamma \vdash t : \top \to A,\ [\textsc{id-}\to] \equiv t \rightleftarrows t\star \qquad (\textsc{id}_{\to_e})$$
$$[\textsc{sym id-}\to] \equiv t \rightleftarrows \lambda x^\top.t \qquad (\textsc{id}_{\to_i})$$

$$[\textsc{asso}] \equiv \langle t, s\rangle \rightleftarrows \langle\langle t, \pi_B(s)\rangle, \pi_C(s)\rangle \qquad (\textsc{split-asso})$$
$$\text{If } \Gamma \vdash t : B \times C,\ [\textsc{dist}] \equiv \lambda x^A.t \rightleftarrows \langle \lambda x^A.\pi_B(t), \lambda x^A.\pi_C(t)\rangle \qquad (\textsc{split-dist}_\lambda)$$
$$\text{若 } \Gamma \vdash t : A \to B, \Gamma \vdash s : A \to C,$$
$$[\textsc{sym dist}] \equiv \langle t, s\rangle \rightleftarrows \lambda x^A.\langle t\, x, s\, x\rangle \qquad (\eta\text{-}\textsc{dist}_{\text{app}})$$
$$\text{若 } \Gamma \vdash t : B \to C,$$
$$[\textsc{sym curry}] \equiv \lambda x^A.t \rightleftarrows \lambda z^{A\times B}.t[\pi_A(z)/x]\pi_B(z) \qquad (\eta\text{-}\textsc{curry})$$
$$\text{若 } A \equiv_p B \text{ 和 } y \notin FV(t),$$
$$[\textsc{cong-}\to_1\ \rho] \equiv \lambda x^A.t \rightleftarrows \lambda x^B.t[[\textsc{sym}\ \rho] \equiv x^B/x^A] \qquad (\textsc{t-subst})$$

表 3：系统 I 中 Top 上同构类型的对应项。

形式化带类型的 λ-演算有两种方法：使用外在类型化项，即分别定义项和类型（因此项可能有类型，也可能没有）；或者使用内在类型化项，即先定义类型，再构造具有给定类型的项。表示变量名也有两种方式：使用更易阅读的具名变量，或使用能使形式化更紧凑的德布鲁因索引。本工作选择了内在表述和德布鲁因索引。

4.1 类型、上下文与变量

类型由类型 Top、函数类型和对组成：

```
数据类型: 集合 其中⊤    :
类型_⇒_: 类型 →类型 →类
型_×_ : 类型 →类型 → 类型
```

由于我们使用自然数来表示变量，因此类型上下文被形式化为类型列表。与经典列表不同，上下文是从右向左读取的。

```
数据上下文: 集合其中∅: 上下文_,
_: 上下文 → 类型 → 上下文
```

内在类型化变量用德布鲁因索引表示。每个变量都由其类型以及该变量在其中具有类型的上下文索引。

```
data _∋_ : Context → Type → Set where
  Z : ∀ {Γ A} → Γ , A ∋ A

  S_ : ∀ {Γ A B} → Γ ∋ B  → Γ , A ∋ B
```

那么， $\Gamma \ni A$ 是类型为 A 且位于上下文 Γ中的变量类型。

例如，以下类型为 $\top$和 $\top \Rightarrow \top$的变量可在上下文 $\emptyset$,$\top \Rightarrow \top$, $\top$中进行类型判断，而且也表示了该类型的证明。

```
_ : ∅ , ⊤ ⇒ ⊤ , ⊤ ∋ ⊤
_ = Z

_ : ∅ , ⊤ ⇒ ⊤ , ⊤ ∋ ⊤ ⇒ ⊤
_ = S Z
```

从示例中可以看出，证明某个变量在某个上下文中具有某种类型，就是一个德布鲁因索引。

4.2 项

这里我们介绍该演算的类型规则。由于我们使用项的内在类型化表示，因此该数据类型的每个构造子都编码了一条类型规则，并表示该演算的一个项，构造子 `[_]≡_` 除外。

考虑到这个例外， $\Gamma \vdash A$ 是上下文Γ中类型为 A 的项的类型，而该类型的每个项都是它的证明，因此项实际上就是类型推导。构造子 `[_]≡_` 不表示项，但会用于某些项的类型推导。

```
数据 _⊢_:  上下文 → 类型 → 集合 其中

  `_ : ∀ {Γ A}       - (ax)
    → Γ ∋ A
    → Γ ⊢ A

  ⋆ : ∀ {Γ} → Γ ⊢ ⊤  - (⊤ᵢ)

  [_]≡_ : ∀ {Γ A B} - (≡)
    → A ≡ B
    → Γ ⊢ A
    → Γ ⊢ B

  ƛ_ : ∀ {Γ A B}     - (⇒ᵢ)
    → Γ , A ⊢ B
    → Γ ⊢ A ⇒ B
```

```
_·_ : ∀ {Γ A B}    - (⇒ₑ)
  → Γ ⊢ A ⇒ B
  → Γ ⊢ A
  → Γ ⊢ B

⟨_,_⟩ : ∀ {Γ A B} - (×ᵢ)
  → Γ ⊢ A
  → Γ ⊢ B
  → Γ ⊢ A × B

π : ∀ {Γ A B}      - (×ₑ)
  → (C : Type)
  → {proof : (C ≅ A) ⊎ (C ≅ B)}
  → Γ ⊢ A × B
  → Γ ⊢ C
```

请注意，构造子 π 接受类型C作为参数。该类型是此函数用于执行投影的类型；此外，它还接受一个隐式参数，作为证明C等于类型A 或类型B。在此参数的类型中，我们使用命题等式，记作 $\cong$ ，本文中均如此。

以下是一些项的示例。每个示例都是一个证明，证明该项在给定的上下文中具有相应的类型。

```
T₁ : ∅ , ⊤ ⊢ ⊤ ⇒ ⊤
T₁ = ƛ ‘ Z

T₂ : ∅ , ⊤ ⊢ (⊤ ⇒ ⊤) × ⊤
T₂ = ⟨ T₁ , ⋆ ⟩

T₃ : ∅ , ⊤ ⊢ ⊤
T₃ = (π fun {inj₁ refl} T₂) · ‘ Z
```

4.3 替换与归约

使用 de Bruijn 表示，替换就是从自然数到项的映射，因此可以将其解释为项的无限序列。这些序列可以通过一些运算符 [1] 构造：

- id：恒等代换：$\{i \mapsto i\}$
- $\uparrow$：移位算子：$\{i \mapsto i+1\}$
- $a \bullet s$：项 a 与替换 s 的连接：$\{0 \mapsto a, i+1 \mapsto s(i)\}$
- $\circ$：替换复合

利用这些操作，我们可以对替换应用于项进行归纳定义，记作 $\langle\langle s \rangle\rangle t$。于是， β-归约可以定义如下：

$$(\lambda t) r \hookrightarrow_\beta \langle\langle r \bullet id \rangle\rangle t$$

为了实现替换，我们采用 Altenkirch 和 Reus 提出的方法，该方法由 McBride 在 [10] 中形式化。替换通过重命名来实现。重命名是从一个上下文中的变量到另一个上下文中的变量的函数，并且保持类型判断。在 Agda 中，重命名和替换的类型如下：

```
重命名: 上下文 → 上下文 → Set 重命名
Γ Δ = ∀{A} → Γ ∋ A → Δ ∋ A

替换: 上下文 → 上下文 →Set subst
Γ Δ = ∀{A} → Γ ∋ A → Δ ⊢ A
```

我们将对项中的变量应用重命名的函数称为 rename : ∀ {Γ Δ} : 重命名 Γ Δ →(∀ {A } → Γ ⊢ A → Δ ⊢ A)，而替换和简单替换分别由函数 ⟪_⟫ : ∀ {Γ Δ} → subst Γ Δ (∀ {C} → Γ ⊢ C → Δ ⊢ C) 和_[_] : ∀{Γ A B } → Γ, B ⊢ A → Γ ⊢ B → Γ ⊢ A 实现。由于这些函数的实现相当标准，我们在此省略其定义。

接下来，我们给出如下数据类型定义的归约关系：

```
data _↪_ : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  β-λ : ∀ {t : Γ , A ⊢ B} {s : Γ ⊢ A}
  → (λ t) · s ↪ t [ s ]

  β-π₁ : ∀ {r : Γ ⊢ A} {s : Γ ⊢ B}
  → π A {inj₁ refl} ⟨ r , s ⟩ ↪ r

  β-π₂ : ∀ {r : Γ ⊢ A} {s : Γ ⊢ B}
  → π B {inj₂ refl} ⟨ r , s ⟩ ↪ s
```

在此定义中，我们省略了构造子 ξ-·1、 ξ-·2、 ξ- ⟨,⟩ 1、 ξ- ⟨,⟩ 2、 ξ-π、 ξ-≡ 和 ζ，它们表示同余规则。构造子 β-λ 对应于 β 归约，而构造子 β-π1 和 β-π2 对应于投影的应用。

由于关系 _↪_由两个相同类型的项索引，因此无需证明它保持类型。这是使用内在类型化项的优势之一。

4.4 类型同构

此形式化中包含的类型同构对应于第 2 节给出的公理集合、表示 ≡对称性的同构，以及一些表示同余规则的同构。

```
数据 _≡_ : 类型 →类型 →集合，其中 comm: ∀ {A B } →
A × B ≡ B × A 结合律: ∀{A B C } →A ×(B × C ) ≡
(A × B ) ×C 分配律: ∀{A B C } →(A ⇒ B ) ×(A ⇒ C )
≡A ⇒B ×C 柯里化: ∀{A B C } →A ⇒B ⇒C ≡(A
× B ) ⇒C id-× : ∀ {A } → A × ⊤ ≡ A id-⇒ : ∀ {A }
→ ⊤ ⇒ A ≡ A abs: ∀ {A } → A ⇒ ⊤ ≡ ⊤sym: ∀ {A
B } → A ≡ B → B ≡ A cong⇒₁ : ∀ {A B C } → A ≡
B → A ⇒ C ≡ B ⇒ C
```

$cong{\Rightarrow}_2 : \forall \{A\ B\ C\} \to A \equiv B \to C \Rightarrow A \equiv C \Rightarrow$
$B\ cong{\times}_1 : \forall \{A\ B\ C\} \to A \equiv B \to A \times C \equiv B \times C$
$cong{\times}_2 : \forall \{A\ B\ C\} \to A \equiv B \to C \times A \equiv C \times B$

我们没有将对应于≡的传递性的同构纳入形式化中，因为它可以使用与类型规则 `[_]≡_` 对应的构造子(≡)以及 ≡的自反性导出，因为它们不会使形式化具有更强的表达能力。

4.5 项的等价

下面给出项之间的等价关系（对应于同构类型 ⇄）的形式化：

```
data _⇄_ : (Γ ⊢ A) → (Γ ⊢ A) → Set where comm : ∀
{r : Γ ⊢ A} → {s : Γ ⊢ B }→ [ comm ]≡ 〈 r , s 〉 ⇄
〈 s , r 〉 交换律 : ∀{r : Γ ⊢ A } → {s : Γ ⊢ B } → {t :
Γ ⊢ C }→ [ 结合律 ]≡ 〈 r , 〈 s , t 〉 〉 ⇄ 〈 〈 r , s 〉
, t 〉 dist-λ : ∀{r : Γ , C ⊢ A } → {s : Γ , C ⊢ B }→ [
dist ]≡ 〈 λ r , λ s 〉 ⇄ λ 〈 r , s 〉 柯里化 : ∀ {r : Γ ,
A , B ⊢ C }→ [ 柯里化 ]≡ (λ λ r ) ⇄ λ subst σ- 柯里化
r uncurry : ∀ {r : Γ , A × B ⊢ C }→ [ sym 柯里化 ]≡
(λ r ) ⇄ λ λ subst σ-uncurry r split-结合律 : ∀{r : Γ ⊢
A } → {s : Γ ⊢ B × C }→ [ 结合律 ]≡ 〈 r , s 〉 ⇄ 〈
〈 r , π B {inj₁ refl} s 〉 , π C {inj₂ refl} s 〉 split-dist-
λ : ∀{r : Γ , C ⊢ A } → {s : Γ , C ⊢ B }→ [ sym dist ]≡
(λ 〈 r , s 〉 ) ⇄ 〈 λ r , λ s 〉 η-dist-app : ∀{r : Γ ⊢
C ⇒ A } → {s : Γ ⊢ C ⇒ B }→ [ dist ]≡ 〈 r , s 〉 ⇄λ
〈 重命名 S_ r · ‘ Z , 重命名 S_ s · ‘ Z 〉 η-柯里化 : ∀ {r : Γ ,
A ⊢ B ⇒ C }→ [ ]≡ λ ⇄ 柯里化 ( r ) subst -
柯里化(重命名 S_ r · ‘ Z) id-× : ∀ {r : Γ ⊢ A } → {t :
Γ ⊢ ⊤}→ [ id-× ]≡ 〈 r , t 〉 ⇄ r abs-e : ∀ {r : Γ ⊢ A
⇒ ⊤} → [ abs ]≡ r ⇄ ⋆abs-i : ∀ {t : Γ ⊢ ⊤}→ [ sy
m abs ]≡ t ⇄ λ 重命名 (S_ {A = A }) t
```

```
id⇒i : ∀ {r : Γ ⊢ ⊤ ⇒ A} → [ id-⇒ ]≡ r ⇄ r · ⋆

id⇒e : ∀ {r : Γ ⊢ A}
  → [ sym id-⇒ ]≡ r ⇄ ƛ rename S_ r
```

wh这里的函数 σ-柯里化 和 σ-uncurry 是类型 Subst (Γ , A , B) (Γ , A × B) 以及 Subst (Γ, A × B) (Γ, A, B) 分别。

在此定义中，为简洁起见，我们省略了对应于表 3 中给出的同余规则的构造子。我们还省略了一些带有前缀 sym 的构造子；这些构造子都可以由基本等价与sym 构造子组合得到。不过，为了展示强正规化，我们在形式化中仍将它们包含在内。

我们注意到，所有构造子左侧都有一条规则（≡），而该规则在右侧消失。例如，comm 构造子会从项中移除规则 [comm]≡ 。对于此处未展示的同余规则情形，所应用的类型同构将被简化。例如，构造子 cong⇒$_2$ 会将同构 [cong⇒$_2$ iso]≡ 简化为 [iso]≡。因此，项的等价只能用于消去或简化项中的构造子（≡）。

我们注意到，项之间的等价关系 ⇄ 和归约关系↪只能关联类型相同的项。这些关系的主体归约性质是类型内在表示的结果。

接下来，我们在 Agda 中定义归约关系 ⇝ ，它同样满足主体归约性质，如下所示：

```
_⇝_ : ∀ {Γ A} → (t t' : Γ ⊢ A) → Set
t ⇝ t' = t ↪ t' ⊎ t ⇄ t'
```

这个定义背后的思路是，构造子 ↪ 消去应用，而 ⇄ 消去同构。

4.6 进展性

本节将证明该演算的进展性，即任何类型正确的闭项要么是一个值，要么可以归约为另一个项。

第一步是定义值的概念。在此演算中，值包括：抽象、项 ⊤，以及值对。

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where
  V-ƛ : ∀ {Γ A B} {t : Γ , A ⊢ B} → Value (ƛ t)
  V-⋆ : ∀ {Γ} → Value (⋆ {Γ})

  V-⟨_,_⟩ : ∀ {Γ A B} {r : Γ ⊢ A} {s : Γ ⊢ B}
    → Value r  → Value s  → Value ⟨ r , s ⟩
```

在下面给出的进展性定义中，我们采用值调用归约顺序作为归约策略。在抽象下进行归约时，可能会遇到自由变量。因此，要考虑的归约最终状态是范式，而不是值。接着，我们证明开项的一个进展性版本。为了完成对闭项进展性的证明，我们证明每个处于范式的闭项都是一个值。

我们首先定义语法，以刻画该语言中范式项的语法 e:

$$\textbf{范式} := \langle \textbf{范式}, \textbf{范式} \rangle \mid \lambda x.\textbf{范式} \mid \star \mid \textbf{中性项}\ \textbf{中性项} :$$
$$= var \mid \textbf{中性项} \cdot \textbf{范式} \mid \pi\ \textbf{中性项} \mid [iso] \equiv \textbf{中性项}$$

其中 var 表示变量，而语法类别 **neu** 用于刻画中性形式，即无法约化且不是值的项。在形式化中，符号 $\Downarrow$ 用于表示中性形式， $\Uparrow$ 用于表示范式。

```
data ⇓ : ∀ {Γ A} → Γ ⊢ A → Set
data ⇑ : ∀ {Γ A} → Γ ⊢ A → Set

data ⇓ where
  `_ : ∀ {Γ A} (x : Γ ∋ A) → ⇓ (` x)

  _·_ : ∀ {Γ A B} {r : Γ ⊢ A ⇒ B} {s : Γ ⊢ A}
    → ⇓ r → ⇑ s  → ⇓ (r · s)

  π : ∀ {Γ A B C p} {t : Γ ⊢ A × B}
    → ⇓ t  → ⇓ (π C {p} t)

  [_]≡_ : ∀ {Γ A B} {t : Γ ⊢ A}
    → (iso : A ≡ B)  → ⇓ t  → ⇓ ([ iso ]≡ t)


数据,⇑其中
  ^_ : ∀ {Γ A} {t : Γ ⊢ A}  → ⇓ t  → ⇑ t

  N-ƛ : ∀ {Γ A B} {t : Γ , A ⊢ B}  → ⇑ (ƛ t)

  N-⟨_,_⟩ : ∀ {Γ A B} {r : Γ ⊢ A} {s : Γ ⊢ B}
    → ⇑ r  → ⇑ s  → ⇑ ⟨ r , s ⟩

  N-⋆ : ∀ {Γ}  → ⇑ (⋆ {Γ})
```

现在我们定义一个关系，用来描述项 t 满足进展性的情形，即：它可以变换为一个等价项，可以进行一个归约步骤，或者它已经处于范式：

```
数据 Progress {ΓA } (t :  Γ ⊢ A ) : Set 其中 step⇄ :  ∀
{t' :  Γ ⊢ A } → t  ⇄ t' → Progress t step↪ :  ∀ {t' :
 Γ ⊢ A } → t  ↪ t' → Progress t done :  ⇑ t →Progress t
```

现在我们可以证明进展性。变量情形和单位情形都非常简单，因为这两个项都处于范式：

```
进展性 :  ∀{ΓA} →(t :  Γ ⊢ A ) →进展性 t进展性
(` x) =完成 (^ (` x))进展性 ⋆      = 完成 N-⋆
```

如果项是 λ抽象，那么我们就在绑定符下调用进展函数；如果这返回一个归约步骤，则使用 ζ （抽象的同余规则）扩展该步骤。否则，该抽象处于范式：

progress (λ t) 与 progress t 的情况如下..... | step$\rightleftarrows$ t$\rightleftarrows$t' = step$\rightleftarrows$ (ζ t$\rightleftarrows$t') ..... | step$\hookrightarrow$ t $\hookrightarrow$t' = step$\hookrightarrow$ (ζ t$\hookrightarrow$t') ..... |done $\Uparrow$t = done N-λ

对于对，证明方法类似。如果项是投影或应用，我们就递归地对每个子项应用进展性；如果结果属于归约或等价的情况，则使用同余规则 ζ；如果属于 done 的情况，那么我们得到一个范式，或者可以应用β-归约。

这项工作中最有趣的情况是，项为 [同构]≡ t。在这种情况下，我们递归地对其子项 t 应用进展性；如果结果属于归约或等价的情况，则使用同余规则 ξ-≡；如果属于 done 的情况，我们就对每种范式以及每种情况下适用的所有同构进行分类讨论，以继续证明。这里仅展示几个情况，其余情况类似。

进展性 ([同构]≡ t), 进展性 t ... | 步骤$\rightleftarrows$ t$\rightleftarrows$t' = 步骤$\rightleftarrows$ (ξ-≡ t$\rightleftarrows$t') ... | 步骤$\hookrightarrow$ t$\hookrightarrow$t' = 步骤$\hookrightarrow$ (ξ-≡ t$\hookrightarrow$t') 进展性 ([交换律]≡ _) |完成 N-〈 $\Uparrow$r , $\Uparrow$s 〉 =步骤$\rightleftarrows$ 交换律 进展性 ([结合律]≡ _) |完成 N-〈 $\Uparrow$r , $\Uparrow$s 〉 =步骤$\rightleftarrows$ (结合律分裂) 进展性 ([分配律]≡ _) |完成 N-〈 N-λ , N-λ 〉 =步骤$\rightleftarrows$ 分配律-λ进展性 ([分配律]≡ _) |完成 N-〈 N-λ , $\Uparrow$s 〉 =步骤$\rightleftarrows$ (分配律-$\lambda\eta_r$) 进展性 ([分配律]≡ _) |完成 N-〈 $\Uparrow$r , N-λ 〉 =步骤$\rightleftarrows$ (分配律-$\lambda\eta_l$) 进展性 ([分配律]≡ _) |完成 N-〈 $\Uparrow$r , $\Uparrow$s 〉 =步骤$\rightleftarrows$ (分配律-$\lambda\eta_{lr}$) 进展性 ([柯里化]≡ (λ λ _)) | 完成 N-λ =步骤$\rightleftarrows$ 柯里化 进展性 ([柯里化]≡ _)| 完成 N-λ =步骤$\rightleftarrows$ (柯里化-η) 进展性 ([id-×]≡ _) |完成 N-〈 $\Uparrow$r , $\Uparrow$s 〉 =步骤$\rightleftarrows$ id-×进展性 ([id-⇒]≡ _) | 完成 $\Uparrow$t =步骤$\rightleftarrows$ id-⇒进展性 ([abs]≡ _) | 完成 $\Uparrow$t = 步骤$\rightleftarrows$ abs 进展性 ([同构]≡ _) | 完成 (^ $\Downarrow$t) = 完成 (^ [同构]≡ $\Downarrow$t)

进展性的定义展示了每个项同构如何对应于必须消除一个规则的情形（≡），以便继续进行归约。

最后，我们通过证明任何处于范式中的闭项都是值来完成进展性的证明；这里仅展示该函数的类型：

```
closed⇑→Value : ∀ {A} {t : ∅ ⊢ A} → ⇑ t → Value t
```

当项是 Top 值、抽象或对时，这个函数的定义非常简单；当项是中性形式时，我们使用函数 ⊥-elim，并证明闭项不可能是中性形式。这个证明可以构造出来，因为中性项必定包含一个自由变量，这是因为变量是其定义中唯一非递归的构造子。

5 强正规化

In t本节证明关系 $\rightsquigarrow$ 的强正规化性质，即每个类型化的项都不可能无限归约，这意味着每个归约序列最终都会以一个值结束。

证明强正规化最常用的技术是可归约性，这一方法由 Tait [13] 提出，并由 Girard [8, 9] 推广。在这项工作中，我们采用 Schäfer 的方法 [11], ；他对可归约性的经典表述进行了修改，由此得到一种更适合形式化的技术，并用它在 Coq 中证明了 System F 的强正规化。我们的形式化基于 András Kovács[1]将 Schäfer 的思想改编到 Agda、用于简单类型λ演算的工作。

我们首先给出强正规化项的构造性定义，该定义最初由 Altenkirch 提出。

关于归约 $\rightsquigarrow$ 的强正规化项集合，通过以下规则归纳定义：

$$\frac{\forall t.s \rightsquigarrow t \implies t \in SN}{s \in SN}$$

这意味着，如果对于每个满足 $s \rightsquigarrow t$ 的项 t ， t 都是强正规化的，那么 s 也是强正规化的。

在 Agda 中，我们可以使用以下归纳数据类型定义集合 SN：

```
data SN {Γ A} (t : Γ ⊢ A) : Set where
  sn : (∀ {t'} → t ⇝ t' → SN t') → SN t
```

我们的目标是定义一个具有以下类型的函数：

```
strong-norm : ∀ {Γ A} (t : Γ ⊢ A) → SN t
```

我们给出该函数的定义，它通过在类型 SN中加入一个关于项的谓词来推广正规化证明。在证明中，我们使用这个谓词，在构造引入时补充关于项的信息，然后在消去的各个情形中使用这些信息。

```
data SN* {Γ A} (P : Γ ⊢ A → Set) (t : Γ ⊢ A) : Set where
  sn* : P t → (∀ {t'} → t ⇝ t' → SN* P t') → SN* P t
```

很容易看出，如果一个项满足 SN*，那么它也满足 强正规化。

要开始证明，我们需要给出一些定义。首先，我们定义“项的解释”，它将用于添加一些额外的假设，使我们能够继续

[1]https://github.com/AndrasKovacs/misc-stuff/blob/master/agda /STLCStrongNorm/StrongNorm2.agda

在消去情形下进行证明。一个对的解释是对中每个元素正规化结果的积，抽象（λ t）的解释是项$\rho(t)[u]$,的正规化结果，其中 u是任意一个可正规化项，而 ρ是任意一个重命名函数；其他项则解释为模块 Data.Unit 的单位类型。

```
⟦_⟧ : ∀ {Γ A} → Γ ⊢ A → Set
⟦ (λ t) ⟧     = ∀ {Δ}{ρ : Rename _ Δ}{u} →
SN* ⟦_⟧ u → SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ t)
⟦ ⟨ a , b ⟩ ⟧ = SN* ⟦_⟧ a ⊗ SN* ⟦_⟧ b
⟦ t ⟧         = Top
```

其中 $\otimes$ 是模块 Data.Product 中运算符 × 的重命名，而 Top 是 $\top$（来自 Data.Unit）的重命名。通过定义一个谓词，可以将项的解释扩展到替换上。该谓词说明替换 σ 在上下文 Γ中是适当的，记作 $\Gamma \vDash\sigma$，当且仅当将 σ 应用于任意变量所得到的所有项都是强正规化的。

```
_⊨_ : ∀{Δ} → (Γ : Context) → (σ : Subst Γ Δ) → Set
Γ ⊨ σ = ∀{A} (v : Γ ∋ A) → SN* ⟦_⟧ (σ {A} v)
```

特别地，恒等代换 ids 是一个适当的替换：

```
⊨ids : ∀{Γ} → Γ ⊨ ids
⊨ids _ =  SN*-rename S_ (σ v)
```

正规化证明的复杂之处在于一个名为充分性的基本定理的证明。该定理指出，对于任意项 t 和适当的替换 σ，SN* ⟦_⟧ (⟪ σ ⟫ t)成立。该定理的证明篇幅很长，还需要证明一些额外的引理。下一小节将介绍证明中一些最重要的情形；这里仅给出该定理的类型：

```
adequacy : ∀ {Γ Δ A} {σ : Subst Γ Δ} → (t : Γ ⊢ A) →
Γ ⊨ σ → SN* ⟦_⟧ (⟪ σ ⟫ t)
```

最后，通过将充分性 应用于恒等代换，证明强正规化性质：

```
强正规化 :  ∀{Γ A}(项 :  Γ ⊢ A) →强正规化 项强
正规化项 =transport强正规化sub-id (SN*-SN(充分
性项 ⊨恒等代换))
```

其中引理sub-id 消除了对项应用恒等代换，而函数 transport 是 subst 的重命名，定义于模块 Relation.Binary.PropositionalEquality 中。

5.1 充分性函数

我们通过对项 t 进行归纳来证明充分性定理。在证明的每种情况下，我们都会使用辅助引理。为简化说明，我们按情况给出证明。

变量情况

当项t 是一个变量时，我们必须证明，应用于该变量的替换 σ 满足 SN* ⟦_⟧。由于我们已假设 σ 是一个适当的替换，只需将 σ 应用于该变量即可：

```
充分性(‘ v) ⊨σ = ⊨σ v
```

Top 情形

这是最简单的情况，因为 $\star$没有可能的归约步骤:

```
adequacy ⋆ _ = sn* tt (λ ())
```

Pair 情形

当项 t 是一个对时，我们定义了以下引理，以便在对 SN* ⟦_⟧ t 的子项满足 SN* ⟦_⟧ 时，得出 t。

```
lemma-⟨,⟩ : ∀ {Γ A B} → {a : Γ ⊢ A} {b : Γ ⊢ B} →
  SN* ⟦_⟧ a → SN* ⟦_⟧ b → SN* ⟦_⟧ (⟨ a , b ⟩)
```

此引理的证明通过对 SN* ⟦_⟧ t 所采取的归约步骤进行分类讨论来完成，该步骤可以是关系 $\hookrightarrow$ 和 $\rightleftarrows$ 的左相容性或右相容性；在每种情况下，我们都使用关于子项的归纳假设。

然后，我们如下完成此情形下的 充分性 证明:

```
adequacy ⟨ a , b ⟩ ⊨σ =
  lemma-⟨,⟩ (adequacy a ⊨σ) (adequacy b ⊨σ)
```

Application 情形

对于 t 是应用的情形，我们需要一个类似的引理:

```
lemma-· : ∀ {Γ A B} → {a : Γ ⊢ A ⇒ B} {b : Γ ⊢ A} →
  SN* ⟦_⟧ a → SN* ⟦_⟧ b → SN* ⟦_⟧ (a · b)
```

在这个引理中，我们的证明与前一个引理类似，不同之处在于，此时该步骤也可以是一次 β-归约。通过将应用中左侧项（必须是一个抽象）的解释应用于右侧项的归纳假设，即可解决这种情形。此情形的充分性引理证明如下:

```
adequacy (a · b) ⊨σ =
  lemma-· (adequacy a ⊨σ) (adequacy b ⊨σ)
```

Projection 情形

当 t 是投影时，需要以下引理，它断言SN* ⟦_⟧t成立，只要被投影的项满足 SN* ⟦_⟧ :

```
lemma-π : ∀ {Γ A B C p} → {a : Γ ⊢ A × B} →
  SN* ⟦_⟧ a → SN* ⟦_⟧ (π C {p} a)
```

这个引理的证明与 lemma-· 的证明类似，因为当这一步是归约（β-π_1 或 β-π_2）时，该引理通过子项的解释来证明；在此例中，子项的解释是一个形如 $\langle a , b \rangle$ 的对，其形式为 SN* ⟦_⟧ a $\otimes$ SN* ⟦_⟧ b。此情形的充分性引理证明如下:

```
adequacy (π _ x) ⊨σ = lemma-π (adequacy x ⊨σ)
```

抽象情形

当 t 是一个抽象时，情况比前几种更困难。与前几种情况一样，我们需要根据 (SN* ⟦_⟧ (λ t) 来证明 SN* ⟦_⟧ t 的解释，给定 (λ t) 的解释：

```
lemma-λ : ∀ {Γ A B} → {t : Γ , B ⊢ A} →
  ⟦ λ t ⟧ → SN* ⟦_⟧ t → SN* ⟦_⟧ (λ t)
```

这个证明需要用到若干引理，例如，必须证明对于任意项 u和 v以及替换 σ:

1. if $u \hookrightarrow v$ then, $\langle\!\langle\ \sigma\ \rangle\!\rangle\ u \hookrightarrow \langle\!\langle\ \sigma\ \rangle\!\rangle\ v$
2. if $u \rightleftarrows v$ then, $\langle\!\langle\ \sigma\ \rangle\!\rangle\ u \rightleftarrows \langle\!\langle\ \sigma\ \rangle\!\rangle\ v$
3. 若 u 可正规化，且 σ 适当，则 $u \bullet \sigma$ 适当。
4. 若 $u \hookrightarrow v$ 则 SN* ⟦_⟧ $u \hookrightarrow$ SN* ⟦_⟧ v5. 若$u \rightleftarrows v$ 则 SN* ⟦_⟧ $u \rightleftarrows$SN* ⟦_⟧ v

此外，我们还需要证明，适当替换与重命名复合后，仍然是适当替换：

```
⊨rename : ∀{Γ Δ Δ₁} {σ : Subst Γ Δ} →
  Γ ⊨ σ → (ρ : Rename Δ Δ₁) → Γ ⊨ (⟪ ids ∘ ρ ⟫ ∘ σ)
```

并且，适当替换的扩展也仍然适当：

```
⊨exts : ∀{Γ Δ A} {σ : Subst Γ Δ} →
  Γ ⊨ σ → (Γ , A) ⊨ (exts σ)
```

现在，可以如下定义抽象的充分性情形：

```
adequacy {σ = σ} (λ t) ⊨σ =
  lemma-λ
    (λ { {ρ = ρ}{u = u} SNu →
      transport (SN* ⟦_⟧)
        (subst-split {t = t})
        (adequacy t (⊨ SNu • (⊨rename ⊨σ ρ)))})
    (adequacy t (⊨exts ⊨σ))
```

函数subst-split 用于将替换 σ 和$u \bullet$ (ids $\circ \rho$) 合并为单个替换。

同构情形

最后，t 为同构构造子的情形最为困难，因为我们必须处理每一种项的等价性的情形。与其他情形一样，我们有一个主要引理：

```
lemma-≡ : ∀ {Γ A B iso} → {t : Γ ⊢ A} →
  SN* ⟦_⟧ t → SN* {A = B} ⟦_⟧ ([ iso ]≡ t)
```

这个函数是通过以下辅助引理定义的：

```
aux : ∀ {Γ A iso t'} → {t : Γ ⊢ A} →
  SN* ⟦_⟧ t → ([ iso ]≡ t) ⇝ t' → SN* ⟦_⟧ t'
```

该引理通过对第二个参数（归约步骤）进行分类讨论来定义。这个引理的目标是从 `SN* ⟦_⟧ t'` 推出 `SN* ⟦_⟧ t`，其中 t 和 t' 是由相应同构关联的项。在这个证明中，我们使用了上文针对不同项构造子定义的引理。

由于篇幅较长且复杂，我们不在此给出该定理的证明，而是简要概述证明中的一些关键要点及所用技巧：

1. 当同构为 $\langle \lambda x^A.r, \lambda x^A.s\rangle \rightleftarrows \lambda x^A.\langle r, s\rangle$ 时，难点在于如何实例化左侧抽象的解释，以构造右侧抽象的解释。为此，我们使用一个引理，该引理由 `SN* ⟦_⟧ t` 推出`SN* ⟦_⟧ (ƛ t)`。该引理的思路是将项中索引为零的第一个变量替换掉，从而得到完全相同的项。

2. 在涉及柯里同构的情形中，必须将多个替换合并为一个替换，而这个替换可以通过抽象的解释得到。

最后，我们指出，项的解释使我们能够处理消去情形（投影和应用）；但另一方面，引入情形（积和抽象）则更为复杂，因为正是在这些情形中构造了此类解释。

6 求值

证明强规范化性质后，由于我们已经证明求值函数会终止，因此可以在 Agda 中定义求值函数。

首先，我们将关系 `_⇝*_` 定义为 `_⇝_`的传递自反闭包：

```
data _⇝*_ {Γ A} : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  _∎ : (t : Γ ⊢ A) → t ⇝* t
  _⇄⟨_⟩_ : (t : Γ ⊢ A) {r t' : Γ ⊢ A} →
    t ⇄ r → r ⇝* t' → t ⇝* t'
  _↪⟨_⟩_ : (t : Γ ⊢ A) {r t' : Γ ⊢ A} →
    t ↪ r → r ⇝* t' → t ⇝* t'
```

给定一个封闭类型化项 t，求值函数的返回类型将是从 t 到某个值 t'的归约序列，其定义如下：

```
data Steps {A} : ∅ ⊢ A → Set where
  steps : {t t' : ∅ ⊢ A} →
    t ⇝* t' → Value t' → Steps t
```

Then, 求值函数是根据这个 f 定义的 函数：

```
求值′ : ∀{A} →(t : ∅ ⊢ A) →强正规化 t →步骤 t 求值′ t _
with 进展性 t 求值′ t
_ | done ⇑t = 步骤 (t) (closed⇑→值 ⇑t)
```

```
eval′ t (sn f)
  | step⇄ {r} t⇄r with eval′ r (f (inj₂ t⇄r))
```

```
...| steps r⇝t' fin=steps (t ⇄⟨ t⇄r ⟩ r⇝t') fin
eval´ t (sn f)
   | step↪ {r} t↪r with eval´ r (f (inj₁ t↪r))

...| steps r⇝t' fin=steps (t ↪⟨ t↪r ⟩ r⇝t') fin
```

此函数将进展性应用于类型化项。有三种可能：

1. 如果项 t 处于范式，则归约序列是平凡的：$t \rightsquigarrow^* t$，并且 t 也是一个值。2. 如果 $t \rightleftarrows r$，我们递归调用 求值，并传入 r 具有强正规化性的证明。结果是在递归结果的基础上添加步骤 $t \rightleftarrows r$ 所得到的归约序列。3. 如果 $t \hookrightarrow r$，则以类似的方式处理。

请注意，为了通过 Agda 的终止性检查器，必须提供证据来证明 t 具有强正规化性。在每个递归步骤中，都会从该参数中移除一个构造子 sn，使其在结构上变得更小。

最后，我们利用强规范化定理定义 求值：

```
求值 :  ∀ {A} → (t : ∅ ⊢ A) → Steps t求
值 t = 求值´ t (强规范化 t)
```

示例 考虑项 Ω 的求值，它可以在 Agda 中定义如下 s:

```
Ω : ∅ ⊢ ⋆
Ω = (ƛ [ sym abs ]≡ (` Z) · ` Z) ·
      ([ abs ]≡ (ƛ [ sym abs ]≡ (` Z) · ` Z))
```

此定义对应于图 2 中给出的类型推导的形式化。该项的求值如下：

```
Ω⇝⋆ : Ω ⇝* ⋆
Ω⇝⋆ =
      Ω
        ⇄⟨ ξ-·₁ (ζ (ξ-·₁ sym-abs)) ⟩
      (ƛ (ƛ ` (S Z)) · ` Z) ·
      ([ abs ]≡ (ƛ ([ sym abs ]≡ (` Z)) · ` Z))
      - (λx. (λy. x) · x) · (λx. x · x)
        ↪⟨ ξ-·₁ (ζ β-ƛ) ⟩
      (ƛ ` Z) · ([ abs ]≡ (ƛ ([ sym abs ]≡ (` Z)) · ` Z))
      - (λx. x) · (λx. x · x)
        ⇄⟨ ξ-·₂ (ξ-≡ (ζ (ξ-·₁ sym-abs))) ⟩
      (ƛ ` Z) · ([ abs ]≡ (ƛ (ƛ ` (S Z)) · ` Z))
      - (λx. x) · (λx. (λy. x) · x)
        ↪⟨ ξ-·₂ (ξ-≡ (ζ β-ƛ)) ⟩
      (ƛ ` Z) · ([ abs ]≡ (ƛ ` Z))
      - (λx. x) · (λx. x)
        ⇄⟨ ξ-·₂ abs ⟩
```

```
    (λ ‘ Z) · ⋆
  – (λx. x) ·  ⋆
    ↪ ⟨ β-λ ⟩
  ⋆
```

```
_: 求值 Ω ≅ 步 Ω⇝⋆ V-⋆
_ = refl
```

我们可以看到，要应用 β-归约，左侧的项必须是一个抽象。因此，要归约形如 `([ iso ]≡ r) · s` 的项，必须通过应用项等价，从左侧移除构造子 `[_]≡_` 。

另一方面，由等价关系 `sym-abs` 和 `sym-id-⇒` 生成的项会添加一个新的抽象，该抽象不会捕获其主体中的任何变量。尽管这看似没有必要，但它实际上对于实现变量的自应用至关重要。

7 结论

在这项工作中，我们在 Agda 中形式化了加入类型 Top 的系统 I，并证明了强正规化定理和进展性定理。

强正规化定理之所以重要，主要有两个原因。从形式化的角度看，构造性证明使我们能够通过对强正规化证据进行归纳，为该演算定义一个全评估函数。从 Curry–Howard 的角度看，将该语言视为一种逻辑时，这一定理确保了其一致性。

这项工作给出的形式化有一些有趣的特点。其中之一是内在类型化项的表示：演算中的每个项都直接实现了一条类型规则。由于这种编码，对这些带类型项进行的函数操作能够保持类型，而项同构的应用则由语法结构决定。因此，关系 $\rightleftarrows$会消除项的类型同构，正如关系 $\hookrightarrow$ 会消除应用和投影一样。这一点对于确保归约序列有限至关重要，因为它能防止某些项同构（例如 $\langle r、\ s\rangle \rightleftarrows \langle s、\ r\rangle$）被无限次地应用。

另一个有趣之处在于，强正规化的证明采用了 Schäfer 的技巧。该技巧以 Tait 和 Girard 的可归约性为框架，同时进行了若干改动，旨在使形式化过程更加顺畅。在这一方法中，我们选择了 Kovács 的变体，因为它更适用于 Agda 的构造性环境。值得注意的是，据我们所知，这是首个对模同构的 λ 演算的正规化证明进行形式化的版本。

在这类演算中，由于项之间存在等价关系，正规化证明中必须考虑的归约情形数量非常庞大，因此，在 Agda 所提供的严格环境下开展工作很有帮助。

8 未来工作

未来，我们希望将这种形式化方法改编到其他模同构的系统中，例如多态系统 I [12], 即 模同构的系统 F，以及分配律 λ-演算 [2], 。后者是系统 I 的无类型变体，其中只考虑分配规则。

沿着类似的方向，我们希望探究已形式化的演算与系统 I_{η} [6] 之间的关系；系统 I 加入了外延性。他们的方法与我们不同：他们将 η-扩展规则引入为归约关系，而我们则将 η-扩展与一些项同构规则结合起来。

此外，定义一种不在项中包含句法见证的形式化变体也会很有趣。这将产生与其类型推导不相对应的项，因此需要某种形式的类型同构推断，以确定可以应用于该项的项同构。

参考文献

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 显式替换。《函数式编程杂志》, 1(4):375–416,1991.

[2]Beniamino Accattoli 和 Alejandro Díaz-Caro。函数式珍珠：分配律 λ- 演算。载于 Keisuke Nakano 和 Konstantinos Sagonas 编，函数式与逻辑编程——第*15*届国际研讨会，*FLOPS 2020*，日本秋田，*2020*年*9*月*14*日至*16*日，论文集，收入《计算机科学讲义》第12073卷，第33–49页。Springer，2020年。[3] T. Altenkirch 和 B. Reus。使用广义归纳类型表示 λ 项的单子表示。载于第*13*届国际研讨会暨 *EACSL* 第*8*届计算机科学逻辑年会论文集，CSL '99，第453–468页，德国柏林、海德堡，1999年。Springer-Verlag。[4] Roberto Di Cosmo。类型同构：从 λ-演算到信息检索与语言设计。《理论计算机科学进展》。Birkhauser，瑞士，1995年。[5] Alejandro Díaz-Caro 和 Gilles Dowek。在识别同构命题的逻辑中进行证明规范化。*LIPIcs*（*FSCD'19*），131:14:1–14:23，2019年。[6] Alejandro Díaz-Caro 和 Gilles Dowek。模同构命题逻辑中的外延证明。《理论计算机科学》，977:114172，2023年。[7] Alejandro Díaz-Caro 和 Pablo E. Martínez López。视同等式的同构：通过实现 λ^{+} 来投影函数并增强部分应用。$ACM\ IFL$，2015(9):1–11，2015年。[8] Jean-Yves Girard。Une extension de L'interpretation de gödel a L'analyse, et son application a L'elimination des coupures dans L'analyse et la theorie des types。载于 J.E. Fenstad 编，第二届斯堪的纳维亚逻辑研讨会论文集，收入《逻辑与数学基础研究》第63卷，第63–92页。Elsevier，1971年。[9] Jean-Yves Girard。高阶算术的函数解释与切消去。博士论文，巴黎第七大学，1972年。[10] Conor McBride。保类型重命名与替换。2005年。[11] Steven Schäfer。构造类型论中的形式系统工程。博士论文，萨尔兰大学，德国萨尔布吕肯，2019年。

[12] Cristian Sottile、Alejandro Díaz-Caro 和 Pablo E. Martínez López。多态系统 I。载于 Olaf Chitil 编，*IFL 2020*：第*32*届函数式语言实现与应用研讨会，线上活动／英国坎特伯雷，*2020*年*9*月*2*日至*4*日，第127–137页。ACM，2020年。[13] William W. Tait。有限类型泛函的内涵解释 I。《符号逻辑杂志》，32(2):198–212，1967年。[14] Philip Wadler、Wen Kokke 和 Jeremy G. Siek。*Agda* 中的程序语言基础。2022年8月。