类型论中的等式类型

1 前置知识与术语

这篇笔记会介绍类型论中的等式类型的不同口味。等式类型指的是形如 `t = u` 的类型。这个类型表达了“`t` 和 `u` 这两个表达式是相等的”这一命题，一个 `p : t = u` 的值就是这个命题的一个证明。

在一般的类型系统中，像 `t` 和 `u` 这样的表达式不能出现在类型中。所以，接下来讨论的类型系统都是类型可以依赖于值的、支持 dependent type 的类型系统。Coq、Agda 等定理证明器就往往使用这样的类型系统。

在 dependent type 中，类型之间的相等关系会更加复杂。例如，$(\lambda x.x)A$ 和 A 应当被视为同一类型。所以，在 dependent type 中，会有一个特殊的等价关系 $t \equiv u$，成为 definitional equality，用于表示两个表达式是相等的。Definitional equality 是类型系统规则的一部分、往往可以由类型检查器自动判定。

下面的 2〜5 节是较为常识性内容，已经有所了解的读者可以放心跳过。9〜11 节的内容我本人并不是特别熟悉，所以可能有一些错误或细节/术语上的不准确，请懂的读者见谅，不懂的读者看看就好。

2 为什么我们需要等式类型

既然 dependent type 中已经有了 definitional equality 这一等价关系，为什么还需要引入一个特殊的等式类型呢？这是因为，definitional equality 是类型系统自身的、元语言中的构造，它无法在该类型系统的程序中操作。例如，我们无法将 definitional equality 用作某一命题的假设。所以，我们需要将 definitional equality 用某种形式加入到语言内部，让它成为语言中的一个普通类型，从而利用它表达各种命题。

3 最直接的做法：extensional equality

由于等式类型是作为“definitional equality”在语言内部的对应存在的，我们可以直接将这一性质写成类型规则，得到如下的定义：

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t = u : A \text{ type}}$$

$$\frac{\Gamma \vdash t \equiv u : A}{\Gamma \vdash \texttt{refl}\ t : t = u : A}\ \text{(eq-intro)} \qquad \frac{\Gamma \vdash p : t = u : A}{\Gamma \vdash t \equiv u : A}\ \text{(eq-elim)}$$

从 (eq-intro) 与 (eq-elim) 两条规则可以看出，这里定义的等式类型 $t = u : A$ 就是给类型系统自身的 definitional equality 裹了一层语言内部的类型而已。然而，这个简单的定义非常强大，因为它可以复用 definitional equality 自身的规则。例如，我们可以证明 *functional extensionality*，即两个函数相等，当且仅当对每种输入它们的结果都相等：

$$\cfrac{\cfrac{\cfrac{\cfrac{H : (\Pi_{x:A} f\ x = g\ x : B[x]), x : A \vdash H\ x : f\ x = g\ x : B[x]}{H : (\Pi_{x:A} f\ x = g\ x : B[x]), x : A \vdash f\ x \equiv g\ x : B[x]}\ \text{(eq-elim)}}{H : (\Pi_{x:A} f\ x = g\ x : B[x]) \vdash \lambda x.f\ x \equiv \lambda x.g\ x : \Pi_{x:A} B[x]}\ (\eta)}{H : (\Pi_{x:A} f\ x = g\ x : B[x]) \vdash f \equiv g : \Pi_{x:A} B[x]}\ \text{(eq-intro)}}{H : (\Pi_{x:A} f\ x = g\ x : B[x]) \vdash \texttt{refl}\ f : f = g : \Pi_{x:A} B[x]}\ \text{(eq-intro)}$$

由于上述等式类型 $t = u : A$ 只是 definitional equality 的一层包装，它的证明/值 $p : t = u : A$ 的内容对类型检查和计算语义都没有影响。因此，我们还可以加入一条名为**等式证明的唯一性**（Uniqueness of Identity Proof，UIP）的规则：

$$\frac{\Gamma \vdash p : t = u : A \qquad \Gamma \vdash q : t = u : A}{\Gamma \vdash p \equiv q : (t = u : A)}$$

如此定义出来的等式类型一般称为 *extensional equality*。尽管 extensional equality 的定义非常简单、表达能力也很强，实际上它很少被直接使用。在 dependent type 中，definitional equality 的一个非常重要的性质是它是类型系统的一部分，可以由类型系统自动判定。然而，由于上述 (eq-elim) 规则，在有 extensional equality 的类型系统中，判定 definitional equality $t \equiv u : A$ 就需要判断“是否存在一个类型为 $t = u : A$ 的值”。判断一个类型是否存在合法的值往往是非常困难/不可能的。在 extensional equality 上也是如此。可以证明，有 (eq-elim)（常称为 *equality reflection*）的情况下，definitional equality 是不可判定的 [1]。

4 最常见的做法：intensional equality

由于 equality reflection 的不可判定性，extensional equality 难以实现到语言当中。Equality reflection 的一大问题是，它把等式的证明 $p : t = u : A$ 丢掉了：它的结论中没有出现这个证明。因此，为了判定 definitional equality，类型检查器就必须猜出这个证明——这是非常困难的。所以，我们可以尝试给等式类型设

计它自己的 eliminator，从而把证明以标注的形式留在表达式中。Definitional equality 与类型系统最重要的互动是如下的规则：

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash t : B} \text{ (conv)}$$

如果我们把这条规则复刻到等式类型上，并设计一个显式的 eliminator，就会得到：

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash p : A = B : \texttt{Set}}{\Gamma \vdash \texttt{coe}(p, t) : B} \text{ (coe)}$$

但是，只加入 `coe` 会产生诸多问题。首先，由于 `coe` 无法消除，它会一直累积在表达式里，使得表达式越来越复杂、臃肿。因此，我们需要给 `coe` 设计一条计算规则：

$$\texttt{coe}(\texttt{refl}\ A, t) \equiv t$$

这条规则说明，当类型 A 和 B 真的满足 definitional equality 的时候（$\texttt{refl}\ A : A = B$），我们可以复用 definitional equality 的 (conv) 规则，直接返回 t。然而，加入了计算规则依然是不够的。尽管我们复用了 definitional equality 的 (conv) 规则，definitional equality 的其他规则、用于从小的等式构造大的等式的规则无法被等式类型复用。我们还希望加入等式的**替换原则**：

$$\frac{\begin{array}{ll}\Gamma \vdash t : A & \Gamma \vdash u : A \\ \Gamma \vdash f : A \to B & \Gamma \vdash p : t = u : A\end{array}}{\texttt{subst}(f, p) : f\ t = f\ u : B}$$

对于替换原则，同样可以加入如下的计算规则：

$$\texttt{subst}(f, \texttt{refl}\ t) = \texttt{refl}\ (f\ t)$$

事实上，`coe` 和 `subst` 可以被组合成**一个** eliminator。这个 eliminator 由于历史原因叫作叫作 J：

$$\frac{\Gamma, x : A, y : A, z : x = y : A \vdash C \text{ type} \qquad \Gamma \vdash p : t = u : A \qquad \Gamma \vdash r : C\ t\ t\ \texttt{refl}}{\Gamma \vdash J(x.y.z.C, t, u, p, r) : C\ t\ u\ p} \ (J)$$

$$J(x.y.z.C, t, t, \texttt{refl}\ t, r) \equiv r$$

初看之下，J 的签名有些复杂。首先，不妨假设 C 不依赖于 $z : t = u : A$，那么 J 的签名就变成了：

$$(C : A \to A \to \texttt{Set}) \to (x : A) \to (y : A) \to (x = y : A) \to C\ x\ x \to C\ x\ y$$

这其实就是 subst 和 coe 的复合：我们先对 $C\ x$ 应用 subst，再进行 coe。我们希望等式类型具有的各种规则，同样可以用 J 导出：

$$\frac{\begin{array}{ll}\Gamma \vdash t : A & \Gamma \vdash u : A \\ \Gamma \vdash f : A \to B & \Gamma \vdash p : t = u : A\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{subst}(f, p) \equiv J(x.y.z.(f\ x = f\ y : B), t, u, p, \mathtt{refl}\ (f\ t)) \\ : f\ t = f\ u : B\end{array}}$$

$$\frac{\Gamma \vdash p : t = u : A}{\begin{array}{c}\Gamma \vdash \mathtt{inv}(p) \equiv J(x.y.z.(y = x : A), t, u, p, \mathtt{refl}\ t) \\ : u = t : A\end{array}}$$

$$\frac{\Gamma \vdash p : t_1 = t_2 : A \qquad \Gamma \vdash q : t_2 = t_3 : A}{\begin{array}{c}\Gamma \vdash \mathtt{comp}(p, q) \equiv J(x.y.z.(t_1 = y : A), t_2, t_3, q, p) \\ : t_1 = t_3 : A\end{array}}$$

所以，以 J 为 eliminator 的等式类型、*intensional equality*，能够表达许多我们希望等式类型具有的性质。此外，如果把 refl 看作构造器，那么 J 的计算规则就像普通的 inductive type 一样，非常容易实现。因此，intensional equality 是实际的 dependent type 语言中最常见的等式类型。Coq、Agda 等的等式类型都是 intensional equality。

5 intensional equality 的问题

尽管 intensional equality 是最常见的等式类型，它无法表达 extensional equality 可以表达的一些重要性质。其中，最重要的两条是 *functional extensionality* 和 *UIP*。在 J 中，等式 $p : t = u : A$ 和 C 活在同一个 context 中，所以 p 无法依赖一些 C 内部的 bound variable，因此 J 无法表达 functional extensionality。UIP 同样无法从 J 推出。因为可以证明 intensional equality 和一些 UIP 不成立的模型是兼容的 [1]。因此，尽管 intensional equality 应用广泛，这更多的是一种向类型检查的实现妥协的结果。研究者也一直在尝试找到能够进行类型检查、但更加强大的等式类型。

既然 intensional equality 中没有 functional extensionality 和 UIP，那么我们能否以公理的形式加入它们呢？加入这些公理不会破坏类型系统的一致性，但会破坏语言的计算性质。Intensional equality 中，等式的证明是有计算上的含义的。J 只有在遇到 refl 时才能触发计算规则。所以，向基于 intensional equality 的系统中加入等式相关的公理会破坏语言的计算性质。即使在没有任何自由变量的场合，计算也可能因为公理而停滞。

除此之外，intensional equality 中的 J 计算规则只在等式是 refl 时成立。这意味着在 open context 下，J 的使用可能会累积在表达式中，使得表达式变得复杂。

6 挽救 extensional equality 的尝试

尽管 extensional equality 会导致类型检查不可判定，但它可以表达 intensional equality 无法表达的许多重要原则。所以，对 extensional equality 做出一些改造或限制，从而恢复类型检查的可能性是一个非常诱人的想法。

实现 extensional equality 的最著名的项目是 NuPRL 项目 [2]。NuPRL 认为，“可判定的类型检查”本身并不是一个类型系统的必要属性。即使类型检查可能不停机，只要当它停机时能给出正确的结果而且类型系统本身是一致的，那么它给出的检查就依然是可信的。因此，NuPRL 直接实现了 extensional equality，并使用内建或用户定义的 tactic 来尝试猜出等式的证明。然而，这会导致类型检查的行为较为混沌、难以预测。

另外一个想法是：extensional equality 难以处理的原因是类型检查需要猜等式的证明。那么，我们为什么不要求用户以类型标注的形式自行提供等式的证明呢？如此一来，类型系统就不需要自己猜证明了。不含 equality reflection 的 definitional equality，例如一般的 $\beta\eta$ equality，依然可以由类型系统自动判定。也就是说，我们在语言中保留 J、不猜等式的证明。但进行计算时，为了使 J 成为“标注”，我们需要把 J 擦除。事实上，这样的类型系统已经被提出过了 [3]。其中等式类型的构造器是（原文是彻底 untyped 的，但同样需要擦除）：

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad |t| \equiv_{\beta\eta} |u|}{\Gamma \vdash \mathtt{join}(t, u) : t = u : A}$$

等式类型的消去依然是通过 J。但与 intensional equality 不同的是，在判断两个表达式是否满足 definitional equality 时，会通过 $|\_|$ 这一构造把所有对 J 的使用擦除。

与 intensional equality 不同，上述系统中等式类型的证明本身没有计算含义：J 会在判定 definitional equality 时被擦除，它不需要检查等式的证明是否为 `refl`。所以，我们可以向系统中加入等式相关的公理，例如 functional extensionality 和 UIP。这些公理不会破坏类型系统的计算性质。

上述系统能够解决 extensional equality 的类型检查需要猜证明的问题。但很遗憾，它依然是 undecidable 的。因为 extensional equality 的不良性质除了需要猜证明外还有一点：当上下文中有错误的假设时，类型安全会被破坏。例如，上述系统中可以写出如下的表达式：

$$\overline{A : \mathtt{Set}, p : A = A \to A \vdash \Omega \equiv \lambda(x : A).\mathtt{coe}(p, x)\ x : A \to A}$$

（`coe` 在前面出现过，可以用 J 定义出）通过再次使用 p，可以使 $\Omega\ \Omega$ 通过类型检查。然而，如果 $\Omega\ \Omega$ 出现在类型里，类型检查需要判定它是否等于另一个表达式时，对 $\Omega\ \Omega$ 进行求值就会触发无限循环。所以，在错误的假设下，上述系统中可以写出不停机，或形如 1 2 这样类型错误的表达式。因此，上述系统没有 decidable 的类型检查，而且在 open context 下的性质不佳。

7 向 intensional equality 中加入 UIP

下面，让我们聚焦于 UIP。Extensional equality 中有/可以加入 UIP，而 intensional equality 则不行。这是因为 intensional equality 中等式的证明有计算含义。前面提到的、加入等式证明标注的 extensional equality 系统可以支持 UIP。然而，由于计算是所有 J 被**无条件**地擦除了，类型系统的性质被破坏了。所以，如果我们想要 UIP，又不想要破坏类型系统的性质的话：

- 在计算 J 时，不能依赖于等式证明的值
- 不能无条件地擦除 J，只应该在“安全”时擦除 J

所以，问题就在于，什么时候擦除/计算 J 才是“安全”的，以及如何能够在不检查等式的证明的情况下进行“安全”与否的判定。

我们知道，当等式是 `refl`，也就是等式两边的表达式满足 definitional equality 时，擦除 J 是安全的。那么，怎样才能在不检查等式的证明的情况下，判定一个等式是否是 `refl` 呢？答案是：不看证明，直接检查等式两边的表达式是否相等 [4]。根据这一思路，我们可以写下如下的计算规则：

$$J(x.y.z.C, t, t, p, r) \equiv r$$

注意到，和 intensional equality 中原版的计算规则相比，等式证明 p 不再需要是 `refl`，计算规则的成立与否是通过直接比较等式两端是否是同一表达式 t 来决定的。这会使类型检查中的计算稍复杂一些，但不会破坏类型检查的 decidability [5]。由于一切等式的证明不再有计算意义，我们可以直接向类型系统中加入 UIP：

$$\frac{\Gamma \vdash p : t = u : A \qquad \Gamma \vdash q : t = u : A}{\Gamma \vdash p \equiv q : (t = u : A)}$$

有上述计算规则和 UIP 的等式类型一般被称为 *definitionally proof irrelevant equality*。这一名称来自于上述系统中 UIP 是以 definitional equality，而非等式类型的形式成立的。

那么，这一解决方案中能否以公理的形式加入 functional extensionality 呢？由于等式的证明不再有计算意义，加入等式相关的公理不再会破坏类型系统的计算性质。然而，将 functional extensionality 以公理形式加入可能会以不同的形式破坏计算性质。考虑一个 indexed inductive type $T : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to \texttt{Set}$，即使我们通过 functional extensionality 证明了 $(+) = \texttt{flip}\ (+) : (\mathbb{N} \to \mathbb{N} \to \mathbb{N})$，$T\ (+)$ 和 $T\ (\texttt{flip}\ (+))$ 作为类型依然不满足 definitional equality。所以，哪怕在一个没有自由变量的环境里，在这个两个类型间使用 J 进行转换依然无法计算。

8 更精细的计算行为：observational equality

通过调整 J 的计算规则，我们向 intensional equality 中加入了 UIP。然而，functional extensionality 依然无法在不破坏计算性质的情况下加入。所以，我

们希望能沿着同样的思路，给 J 更多的计算行为。

之前的系统中，J 只有在等式两边完全相等时才会计算。但是，假如我们知道等式两边的顶层结构是相等的，即使无法完全消除 J，也依然可以将计算向前推进。例如，假设我们有 $p : A \times B = A' \times B'$，尽管等式两边不完全相等，我们依然可以通过：

$$\mathtt{coe}(p, t) = (\mathtt{coe}(p_1, \pi_1 t), \mathtt{coe}(p_2, \pi_2 t))$$

来推进计算。其中 π_1 和 π_2 取出一个 pair 中的两个分量，$p_1 : A = A'$，$p_2 : B = B'$。这种通过等式两侧的表达式的结构逐步推进计算的等式类型称为 *observational equality* [6, 7, 8]。

为了使得计算能够推进下去，我们需要把大的等式证明拆成小的等式证明的能力。例如，为了计算沿着一个等式 $p : A \to B = A' \to B'$ 的转换，我们需要得到一个 $A = A'$ 和 $B = B'$ 的证明。因此，我们需要为每种类型构造加入对应的等式拆解构造：

$$\frac{\Gamma \vdash p : \Pi_{x_1 : A_1} B_1[x_1] = \Pi_{x_2 : A_2} B_2[x_2]}{\Gamma \vdash \pi_{\mathrm{arg}} p : A_1 = A_2}$$

$$\frac{\Gamma \vdash p : \Pi_{x_1 : A_1} B_1[x_1] = \Pi_{x_2 : A_2} B_2[x_2]}{\Gamma \vdash \pi_{\mathrm{ret}} p : \Pi_{x_1 : A_1} \Pi_{x_2 : A_2} (x_1 : A_1 = x_2 : A_2) \to B_1[x_1] = B_2[x_2]}$$

注意到，在 π_{ret} 中，出现了形如 $x_1 : A_1 = x_2 : A_2$ 的等式：等式的类型两边不一样了。这是因为，普通的、等式两边类型相同的等式类型 $t = u : A$ 不能满足上述拆解构造的需求。因为在 $\Pi_{x_1 : A_1} B_1[x_1] = \Pi_{x_2 : A_2} B_2[x_2]$ 中，A_1 和 A_2 不一定是同一类型。所以我们必须带着这两个不同的类型去考虑 $B_1[x_1]$ 和 $B_2[x_2]$ 之间的相等关系。这类等式两边类型可能不同的等式类型称为 *heterogeneous equality* [9]，而两边类型相等的等式类型称为 *homogeneous equality*。

需要注意的是，采用 heterogeneous equality 并不意味着等式可以无视类型了。它只是把证明两侧类型相等的职责延后了。那些为真的等式（也就是在没有任何假设的 closed context 下成立的等式）的两侧的表达式和类型依然应当是相等的。

和 definitionally proof irrelevant equality 一样，observational equality 中等式类型的证明没有计算意义，计算通过检查等式两边的表达式自身来完成。所以，UIP 同样可以放心加入。但与 definitionally proof irrelevant equality 不同，得益于更精细的计算规则，observational equality 中可以加入 functional extensionality 等公理，而不破坏计算性质。考虑上一节中的例子，我们有 $T : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to \mathtt{Set}$，并证明了 $(+) = \mathtt{flip}\ (+) : (\mathbb{N} \to \mathbb{N} \to \mathbb{N})$。由此，我们可以证明类型上的等式 $p : T\ (+) = T\ (\mathtt{flip}\ (+))$。接下来，当我们尝试沿着这一等式计算 $\mathtt{coe}(p, t)$，其中 $t : T\ (+)$ 时，尽管等式两边不完全相等（根据 definitional equality），它们的顶层类型构造器都是 T，所以 coe 会根据 T 的定义对 t 进行归纳。这样一来，只要 t 本身是由 T 的构造器构建出的值，计算就能推进下去。

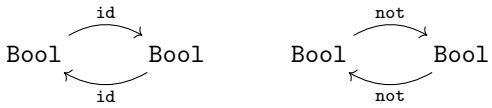
由于 observational equality 中加入等式相关的公理不会破坏计算性质，我们可以向其中加入 functional extensionality，以及 definitional 版本的 UIP 等等。在 observational equality 相关的工作中，往往会充分利用这一优势，将等式的引入按照各个类型的结构分别定义。例如，两个元组相等就是各个分量相等、两个函数相等就是对每个可能输入都给出相同结果，等等。事实上，observational equality 的名字就来源于此：等式的证明方式通过各个类型可以如何被使用/观测决定，对于元组就是取分量、对于函数就是提供输入。

9 放弃 UIP：同伦类型论

Intensional equality 本身无法证明 UIP，这对于许多应用来说可能是一个缺点。然而，这也意味着 intensional equality 可以兼容一些 UIP 不成立的系统。这其中最著名的一个例子就是同伦类型论(Homotopy Type Theory，HoTT)[10]。

同伦类型论的核心是一条名为 *Univalence* 的公理，它声称同构的类型是相等的。例如，我们可以在同伦类型论中证明 $A \times B = B \times A$。Univalence 将“数据的具体表示方式不重要，只要结构同构性质就一样”这一点用公理的形式表达，并允许类型系统的使用者利用这一公理来简化证明。

同伦类型论中的等号类型支持 J 及其计算规则。尽管这篇笔记前面的内容几乎都在讨论 intensional equality 与 J 能力的不足，比起其他不是等式的类型，J 使用起来依然是非常方便灵活的。尽管支持 J，同伦类型论中 UIP 是**不成立**的。考虑布尔类型 `Bool`，`Bool` 和它自己有两种同构方式：



$$\begin{aligned} &\texttt{id} \quad b \quad &&= b \\ &\texttt{not} \quad \texttt{true} \quad &&= \texttt{false} \\ &\texttt{not} \quad \texttt{false} \quad &&= \texttt{true} \end{aligned}$$

由这两种等价方式可以通过 Univalence 生成两个 `Bool` = `Bool` 的等式证明 $p_{\texttt{id}}$ 和 $p_{\texttt{not}}$。然而，沿着这两个等式证明计算 `coe` 会得出不同的行为，因为我们希望沿着一个由 Univalence 得到的等式进行 `coe` 会得到用于生成这个证明的同构关系自身。所以：

$$\begin{aligned} \texttt{coe}(p_{\texttt{id}}, \texttt{true}) &\equiv \texttt{id true} \equiv \texttt{true} \\ \texttt{coe}(p_{\texttt{not}}, \texttt{true}) &\equiv \texttt{not true} \equiv \texttt{false} \end{aligned}$$

如果 $p_{\texttt{id}} = p_{\texttt{not}}$，那么我们就能得到 `true` = `false`! 所以，在同伦类型论中，UIP 不仅不成立，而且是不一致的。

尽管同伦类型论不支持 UIP，但它能够支持 functional extensionality。事实上，functional extensionality 可以通过 Univalence 直接证明 [11]。

10 能计算的 HoTT：立方类型论

虽然同伦类型论支持 Univalence 这一非常灵活的公理，目前它的语义缺乏优秀的计算性质，有一些等式相关的操作无法被完全消除。为此，受到同伦类型论的一个数学模型的启发，一些叫作立方类型论 (Cubical Type Theory) 的类型系统被设计了出来 [12, 13, 14]。同伦类型论对于等式应该如何形成没有做太多描述，只是要求 Univalence 这一公理应当成立，以及等式类型应当满足 intensional equality 中的 J 及其计算性质。而立方类型论则改变了这一状况，它使用一种名为区间（interval）的抽象来表达所有等式。

区间是立方类型论中内建的一个类型 $\mathbb{I}$，它有两个值 $\mathtt{L}, \mathtt{R} : \mathbb{I}$。然而，区间**不是**布尔类型，这体现在区间的值不能被模式匹配：即使拿到了一个区间值 $i : \mathbb{I}$，也无法判断 i 是 $\mathtt{L}$ 还是 $\mathtt{R}$。这正是区间这一名字的由来：区间上的一个点除了其两个端点 $\mathtt{L}$ 和 $\mathtt{R}$，还可能是中间的某个点，所以不能把区间上的一个点当成左右端点中的一个来处理。

那么，区间如何能够用来描述等式呢？假如我们有一个类型为区间的变量 $i : \mathbb{I}$。我们不能对 i 模式匹配，所以只剩下两种使用 i 的方法：

- 完全不使用 i，此时无论 i 是什么值结果都一样
- 将 i 传递给另一个以 $\mathbb{I}$ 为参数的函数

假设我们有一个函数 $f \equiv \lambda i. \ldots : I \to A$，那么这一函数就像等式一样有两个端点 $f\ \mathtt{L}$ 和 $f\ \mathtt{R}$。此外，f 对参数 i 的使用方式同样和等式高度相似：参数 i 未被使用时，f 的两个端点本来就是相等的，这对应 `refl`。当参数 i 被传递给另一个以 $\mathbb{I}$ 为参数的函数时，就相当于利用了另外一个等式来构建一个更大的等式。无法对 i 模式匹配的特点导致我们无法构造出“坏”的等式。所以，在立方类型论中，我们可以直接将 A 上的等式定义为 $I \to A$ 的函数。

利用区间来定义等式能提供非常高的灵活性。例如，“各分量相等时元组相等”可以表达为：

$$p_1 : I \to A, p_2 : I \to B \vdash \lambda(i : \mathbb{I}).(p_1\ i, p_2\ i)$$

Functional extensionality 也可以用区间轻松地定义出：

$$h : A \to (I \to B) \vdash \lambda(i : \mathbb{I}).\lambda(x : A).h\ x\ i$$

读者可以自行验证把 $I \to A$ 解释为等式时，这两个证明对应的就是我们想要的等式。

那么，如何利用一个由区间构造的等式进行计算呢？这里的计算规则与 observational equality 非常相似：根据等式两端的结构进行逐步的计算。只不过，observational equality 的计算靠的是等式的端点，而立方类型论的证明是通过检查等式的证明自身来完成的。例如，在类型 $A \times B$ 上的 coercion 可以如此计算：

$$\mathsf{coe}((\lambda(i : \mathbb{I}).A \times B), t) \equiv (\mathsf{coe}((\lambda i.A), \pi_1 t), \mathsf{coe}((\lambda i.B), \pi_2 t))$$

然而，形如 $\texttt{coe}((\lambda(i:\mathbb{I}).\lambda(j:\mathbb{I}).\ldots),\ldots)$ 的，等式自身上的 coercion 应该如何计算呢？这一次，无法根据类型的结构写出直白的计算规则了。为了解决这一问题，需要引入一个名为 comp（composition）的新操作。这一操作不止要处理一维的区间 $\mathbb{I}\to A$，还要处理更高维的“立方体” $\mathbb{I}^n\to A$。这正是立方类型论名字的由来。此外，向立方类型论中加入 Univalence 公理同样需要引入额外的构造。由于我自己也不懂、复杂度较高，此处不进行讨论。

目前，立方类型论有两个变种。一种叫作 Cartesian Cubical Type Theory [14]，它有较为复杂的 comp 等操作。另一种叫作 De Morgan Cubical Type Theory [13]，它的操作较为简单，但给 interval 加上了更丰富的结构（de Morgan algebra）。此外，在 Arend [15] 语言中实现了一个名为 HoTT with an interval 的类型系统 [16]。它虽然并不属于立方类型论，但同样使用了区间来构建等式，并支持 Univalence。Arend 的类型系统比立方类型论要简单许多，但它没有好的计算性质（虽然比 HoTT 要好）。

11 使用区间，但回到 UIP：XTT

立方类型论中“区间”的概念用于构建等式非常方便：包括 functional extensionality 在内的诸多性质都能简单地证明。然而，完整的立方类型论缺乏 UIP、支持 Univalence，并为此增加了许多复杂度。那么，是否能利用区间这一抽象来表达等式，但与此同时支持 UIP 呢？沿着这一思路得到的类型系统就是 XTT [17, 18]。

XTT 中所有等式类型都使用区间 $\mathbb{I}$ 来构造，类型 A 上的等式类型就是 $\mathbb{I}\to A$ 的函数。因此，functional extensionality 等原则都在 XTT 中成立。为了支持 UIP，XTT 引入了一条名为 boundary seperation 的原则：给定两个函数/等式 $p,q:I\to A$，如果 $p\,\mathtt{L}\equiv q\,\mathtt{L}$ 且 $p\,\mathtt{R}\equiv q\,\mathtt{R}$，那么 $p\equiv q$。这说明任意两个端点相同的等式都是相等的，正是 UIP 在区间下的表述。

XTT 中的 coercion 的计算和 observational equality/立方类型论中相同，是根据类型的结构逐步完成的。然而，和 observational equality 不同，XTT 中的 coercion 的计算是根据等式的**证明**——一个 $\mathbb{I}\to A$ 的函数，而非等式的两个端点来进行的。所以，XTT 中等式的证明是有计算含义的。为了处理等式类型自身上的 coercion，依然需要引入一个 comp 操作。不过，由于 boundary separation/UIP 的存在，这一操作比立方类型论中的对应版本简单。

XTT 中还有一个额外的复杂度来源。和 observational equality 一样，XTT 需要把大的等式拆解成小的等式的能力。大部分情况下，这一点可以直接通过对区间的操作完成。然而，有一个例外：函数的参数类型。假如我们有 $p:A\to B=A'\to B'$，很难通过正常手段得到一个 $A=A'$ 的证明。在 observational equality 中可以通过加入新的公理/内建构造解决这一需求，但在 XTT 中，由于等式的证明有计算意义，加公理是不可行的。为此，XTT 引入了一个对类型进行模式匹配的构造 typecase。函数的参数类型可以通过 $\texttt{typecase}(T)\{\Pi_{x:A}B[x]\Rightarrow A\mid \_\Rightarrow\ldots\}$ 取出。然而，typecase 使得函数可以对任意类型进行模式匹配与分类讨论，这在增加灵活性的同时，会使得语言失去 parametricity、行为变得更加混乱。所以，这也可能成为 XTT 的一个缺点。

12 总结

类型系统/等式类型	functional extensionality	UIP	计算性质	类型检查
intensional equality	无	无	好	可判定
intensional equality + 公理	有	有	差	可判定
extensional equality	有	有	好	不可判定且非常困难
extensional equality + 等式标注	有	有	好	不可判定
definitionally proof irrelevant equality	无	有	好	可判定
observational equality	有	有	好	可判定
同伦类型论	有	不成立	差	可判定
立方类型论	有	不成立	好	可判定
Arend/HoTT with an interval	有	不成立	差	可判定
XTT	有	有	好	可判定

References

[1] M. Hofmann, "Extensional concepts in intensional type theory," 1995.

[2] "PRL Project Home." `https://www.nuprl.org`.

[3] V. Sjöberg and A. Stump, "Equality, quasi-implicit products, and large eliminations," *Electronic Proceedings in Theoretical Computer Science*, vol. 45, pp. 90–100, jan 2011.

[4] T. Altenkirch, "Extensional equality in intensional type theory," in *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pp. 412–420, 1999.

[5] A. Abel, "Extensional normalization in the logical framework with proof irrelevant equality," 2009.

[6] T. Altenkirch and C. McBride, "Towards observational type theory," 2006.

[7] T. Altenkirch, C. McBride, and W. Swierstra, "Observational equality, now!," in *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, (New York, NY, USA), p. 57–68, Association for Computing Machinery, 2007.

[8] L. Pujet and N. Tabareau, "Observational equality: Now for good," *Proc. ACM Program. Lang.*, vol. 6, jan 2022.

[9] C. McBride, "Dependently typed functional programs and their proofs," 2000.

[10] The Univalent Foundations Program, "Homotopy type theory: Univalent foundations of mathematics," tech. rep., Institute for Advanced Study, 2013.

[11] "Univalence implies function extensionality." `https://planetmath.org/49univalenceimpliesfunctionextensionality`.

[12] T. Coquand, M. Bezem, and S. Huber, "Computational content of the axiom of univalence," 2013.

[13] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: a constructive interpretation of the univalence axiom," 2013.

[14] C. Angiuli, G. Brunerie, T. Coquand, Kuen-Bang Hou (Favonia), R. Harper, and D. R. Licata, "Cartesian cubical type theory," 2013.

[15] "Arend Theorem Prover." `https://arend-lang.github.io`.

[16] V. Isaev, "Models of homotopy type theory with an interval type," 2020.

[17] J. Sterling, C. Angiuli, and D. Gratzer, "Cubical syntax for reflection-free extensional equality," 2019.

[18] J. Sterling, C. Angiuli, and D. Gratzer, "A cubical language for bishop sets," 2020.