一个依值类型 Lambda 演算的教学实现

Andres Löh
原作者
乌特勒支大学

Conor McBride
原作者
斯特拉斯克莱德大学

Wouter Swierstra
原作者
诺丁汉大学

Chuigda Whitegive
翻译
第七通用设计局

Gemini
类型论支持、校对
Google Deepmind

Claude
校对
Anthropic

译者前言

本文是对文章 A tutorial implementation of a dependently typed lambda calculus 的中文翻译，部分字句有所改动。排版经过调整，使图表总是位于直接描述该图表的文本下方，以便阅读。术语 *(terminology)* 在正文中第一次出现的地方以仿宋体（中文）或 *Italic (English)* 呈现，如果某个术语难以辨认，则总是会以这种形式呈现。本文附有其它语言的实现。

摘要

本文介绍了依值类型演算核心的类型规则，并提供了一个简洁的 Haskell 实现。本文着重阐述了从简单类型 Lambda 演算过渡到依值类型 Lambda 演算所需的变更。此外，本文还描述了如何扩展核心语言的数据类型，并给出了几个小型示例程序。本文附带一个可执行解释器和示例代码，方便读者立即体验本文所描述的系统。

1. 介绍

许多函数式程序员都对使用依值类型 *(dependent type)* 编程犹豫不决。他们常说：依值类型会使类型检查变得不可判定，类型检查器总是陷入死循环，以及依值类型非常、非常难。

然而，同一批程序员却非常热衷于使用各种复杂的类型系统扩展进行编程。比如说，现在的 Haskell 实现支持广义代数数据类型，带函数依赖的多参数类型类，关联类型和类型家族，非直谓 *(impredicative)* 高阶类型，凡此种种。程序员们似乎对依值类型唯恐避之不及。

函数式社区对依值类型普遍缺乏理解，这是阻碍依值类型进一步普及的主要障碍之一。尽管目前已经有了不少基于依值类型的优秀实验工具和编程语言，理解这些工具到底是如何工作的却是一件难事。很大一部分关于依值类型的文献都是由类型学家撰写、供其他类型学家阅读的，这些文献对函数式程序员来说并不友好。本文致力于弥补这一现状。

本文从简单类型 *Lambda* 演算 *(simply-typed lambda calculus)*（第二节）开始，给出了抽象语法、求值和类型检查的数学规范和 Haskell 实现。以简单类型 Lambda 演算为起点，本文进一步地研究最小化的依值类型 *Lambda* 演算 *(dependently typed lambda calculus)*（第三节）。

受 Pierce 逐步发展类型系统[21]的启发，本文重点阐述了向依值类型 Lambda 演算过渡所需的规范和实现方面的变更。令人惊讶的是，所需的变更并没有想象中那么多。我们希望通过尽可能明确地阐述这些变更，能让已经熟悉简单类型 Lambda 演算的读者尽可能顺利地过渡到依值类型。

尽管本文中没有发明新的类型系统，我们相信本文可以作为 Haskell 中依值类型系统实现的入门教程。实现一个类型系统是学习其中所有微妙细节的最佳途径之一。尽管我们不打算全面探讨所有实现类型化 Lambda 演算的方法，但我们会尽量明确阐述我们的设计决策，仔细提供其他选择，并概述更广泛的设计空间。

只有将数据类型添加到这个基础演算中，才能真正发挥依值类型的全部威力。因此，我们在第 4 节中演示了如何使用自然数和向量 *(vector)*扩展我们的语言。使用这些新增的数据类型，我们编写了经典的向量追加操作，以演示如何在我们的核心演算中进行编程。而利用本节解释的原理，可以向演算中加入更多数据类型。

最后，我们简化了系统实验流程：本文源代码包含一个小型解释器，用于解释我们描述的类型系统和求值规则。由于使用了与本文相同的源代码，该解释器能够保证严格遵循我们描述的实现，并且文档齐全。因此，它为进一步学习和实验提供了一个宝贵的平台。

本文并非对依值类型编程的入门介绍，也未讲解如何实现完整的依值类型编程语言。然而，我们希望本文能够消除函数式程序员对依值类型的诸多误解，并鼓励读者进一步探索这一令人兴奋的研究领域。

2. 简单类型 Lambda 演算

在探索依值类型的过程中，我们希望从熟悉的领域入手。因此，在本节中，我们将讨论简单类型 Lambda 演算，简称 $\lambda_{\to}$。从某种意义上来说，$\lambda_{\to}$是最小的静态类型函数式语言。每个词项 *(term)*[1] 都显式地标注了类型，不需要类型推理。相比于作为 ML 或 Haskell 等支持多态类型和类型构造子 *(type constructor)* 的语言的基础的那些类型化 *Lambda* 演算 *(typed lambda calculi)*，$\lambda_{\to}$的结构要简单得多。在$\lambda_{\to}$中只有基础类型，且函数类型不能是多态的 *(polymorphic)*。如果不增加额外的规则，$\lambda_{\to}$是强正规化 *(strongly normalizing)* 的：对于任何词项，无论求值策略如何，求值总是能停机。

2.1. 抽象语法

$\lambda_{\to}$的类型语言仅由两种结构组成。存在一组基本类型 α；复合类型 $\tau \to \tau'$ 则对应着接受一个 τ，返回一个 τ' 的函数：

$$\begin{array}{lll} \tau ::= \alpha & & \text{基本类型} \\ \quad \mid \tau \to \tau' & & \text{函数类型} \end{array}$$

$\lambda_{\to}$共有四种词项：带显式类型注解 *(type annotation)* 的词项、变量 *(variable)*、应用 *(application)* 和 *Lambda* 抽象 *(lambda abstraction)*：

$$\begin{array}{lll} e ::= e :: \tau & & \text{带类型注解的词项}^{2} \\ \quad \mid x & & \text{变量} \\ \quad \mid e\, e' & & \text{应用} \\ \quad \mid \lambda x \to e & & \text{Lambda 抽象} \end{array}$$

词项可以被求值为值 *(value)*。一个值要么是一个中性项 *(neutral term)* ——一个被应用了零个或多个值的变量，要么是一个 Lambda 抽象：

$$\begin{array}{lll} v ::= n & & \text{中性项} \\ \quad \mid \lambda x \to v & & \text{Lambda 抽象} \\ n ::= x & & \text{变量} \\ \quad \mid n\, v & & \text{应用} \end{array}$$

[1]译注：词项这一术语翻译取自练琪灏的依值类型入门讲义：https://m.or.gd/notes/mltt-20230703.pdf。

[2]类型学家使用符号 ":" 或者 "$\in$" 表示类型归属关系。然而，在 Haskell 中，符号 "`:`" 被用作列表的 `cons` 运算符，因此 Haskell 的设计者为类型注解选择了非标准的"::"。本文将尽可能地遵循 Haskell 语法，以缩小本文所涉及的不同语言之间的语法差距。

2.2. 求值

$\lambda_{\to}$的求值规则（大步）如图 1 所示。记号 $e \Downarrow v$ 表示将 e 完全求值的结果是 v。因为$\lambda_{\to}$是强正规化的语言，求值策略自然无关紧要。简单起见，我们尽可能地把所有东西求值到底——包括 Lambda 内部的东西。类型注解在求值阶段会被忽略。求值变量会得到变量自身。

唯一值得关注的情况是应用：在对应用求值时，需要依据应用左边的子项的求值结果作出不同的行动：当左子项的求值结果是中性项时，求值无法继续进行，此时我们把两个子项的求值结果组合成一个新的中性项；当左子项的求值结果是 Lambda 抽象时，我们进行β-归约 *(β-reduce)*，这一过程可能产生新的可归约项 *(redex, reducible-expression,* 直译为可归约表达式*)*，因此我们还要对 β-归约的结果继续求值。

$$\frac{e \Downarrow v}{e :: \tau \Downarrow v} \quad \text{类型注解会被忽略}[3]$$

$$\frac{}{x \Downarrow x} \quad \text{求值变量会得到变量自身}$$

$$\frac{e \Downarrow v}{\lambda x \to e \Downarrow \lambda x \to v} \quad \text{求值 Lambda 抽象的内部}$$

$$\frac{e \Downarrow \lambda x \to v \quad v[x \mapsto e'] \Downarrow v'}{e\, e' \Downarrow v'} \quad \text{对应用求值：左边的子项是 Lambda 抽象}$$

$$\frac{e \Downarrow n \quad e' \Downarrow v'}{e\, e' \Downarrow n\, v'} \quad \text{对应用求值：左边的子项是中性项}$$

图 1　$\lambda_{\to}$的求值规则

作为例子，以下是$\lambda_{\to}$中的一些词项和它们的求值结果。我们用 `id` 代指词项 $\lambda x \to x$，用 `const` 代指词项 $\lambda x\, y \to x$ ——这种写法是 $\lambda x \to \lambda y \to x$ 的语法糖。

$$(\texttt{id} :: \alpha \to \alpha)\, y \Downarrow y$$

$$(\texttt{const} :: (\beta \to \beta) \to \alpha \to \beta \to \beta)\ \texttt{id}\ y \Downarrow \texttt{id}$$

2.3. 类型系统

类型规则 *(type rule)*大都形如 $\Gamma \vdash e :: \tau$，这表示在上下文 Γ 中，词项 e 的类型是 τ。类型上下文中列出了有效的基本类型，并将标识符与类型信息相关联。我们用 $\alpha :: *$ 表示 α 是一个基本类型，用 $x :: \tau$ 表示词项 x 的类型是 τ。

$$\begin{aligned} \Gamma ::= {} & \varepsilon && \text{空的类型上下文} \\ \mid {} & \Gamma, \alpha :: * && \text{添加一个类型标识符} \\ \mid {} & \Gamma, x :: \tau && \text{添加一个词项标识符} \end{aligned}$$

词项和类型中的每个自由变量 *(free variable)* 都必须在类型上下文中出现。例如，如果我们要声明 `const` 具有类型 $(\beta \to \beta) \to \alpha \to \beta \to \beta$，则类型上下文中至少应该包含：

$$\alpha :: *,\ \beta :: *,\ \texttt{const} :: (\beta \to \beta) \to \alpha \to \beta \to \beta$$

注意 α 和 β 是在它们被 `const` 的类型使用前引入的。

[3]译注：这些说明性文本是译者加的。

基于上述规则，我们在图 2 中给出了类型上下文及其有效性的定义。

$$\frac{}{\mathrm{valid}(\varepsilon)} \qquad \frac{\mathrm{valid}(\Gamma)}{\mathrm{valid}(\Gamma, \alpha :: *)} \qquad \frac{\mathrm{valid}(\Gamma) \quad \Gamma \vdash \tau :: *}{\mathrm{valid}(\Gamma, x :: \tau)}$$

$$\frac{\Gamma(\alpha) = *}{\Gamma \vdash \alpha :: *}\ [\mathrm{TVAR}] \qquad \frac{\Gamma \vdash \tau :: * \quad \Gamma \vdash \tau' :: *}{\Gamma \vdash \tau \to \tau' :: *}\ [\mathrm{FUN}]$$

图 2　$\lambda_{\to}$中的类型上下文和良构 *(well-formed)* 的类型

图 2 中的最后两条规则解释了一个类型在什么情况下是良构的——当类型中的所有自由变量都在上下文中的时候。在类型良构性 *(well-formedness)* 规则以及后续的类型规则中，我们隐含地假设所有类型上下文都有效。

注意$\lambda_{\to}$不是多态的：每个类型标识符都代表一个具体的类型，不能被实例化。

最终，我们可以给出如图 3 所示的类型规则。我们不推断被 Lambda 绑定的变量的类型。也就是说，总体上而言，我们只进行类型检查。不过我们可以很容易地判定带类型注解的词项、变量和应用的类型。因此，当一个类型是类型检查算法的输入时，我们给类型规则标上 $::_{\downarrow}$；当类型是类型检查算法的输出时，我们给类型规则标上 $::_{\uparrow}$。目前这些记号只是为了提供一些直觉，但在实现中，两种规则之间的差异会变得明显。

$$\frac{\Gamma \vdash \tau :: * \quad \Gamma \vdash e ::_{\downarrow} \tau}{\Gamma \vdash (e :: \tau) ::_{\uparrow} \tau}\ [\mathrm{ANN}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x ::_{\downarrow} \tau}\ [\mathrm{VAR}]$$

$$\frac{\Gamma \vdash e ::_{\uparrow} \tau \to \tau' \quad \Gamma \vdash e' ::_{\downarrow} \tau}{\Gamma \vdash e\, e' ::_{\downarrow} \tau'}\ [\mathrm{APP}]$$

$$\frac{\Gamma \vdash e ::_{\uparrow} \tau}{\Gamma \vdash e ::_{\downarrow} \tau}\ [\mathrm{CHK}] \qquad \frac{\Gamma, x :: \tau \vdash e ::_{\downarrow} \tau'}{\Gamma \vdash \lambda x \to e ::_{\downarrow} \tau \to \tau'}\ [\mathrm{LAM}]$$

图 3　$\lambda_{\to}$的类型规则

我们首先来看可推断项 *(inferable term)* [ANN]，我们将带类型注解的词项与其类型注解进行比对，然后返回该类型。变量的类型可以在环境中查找 [VAR]。对于应用 [APP]，我们首先处理函数 e——它必须具有函数类型，然后我们将参数 e'与函数的输入类型进行比较，并将函数的返回类型[4]作为结果类型返回。

最后两条规则用于类型检查。如果我们能推断出一个词项的类型，且该类型与给定的类型一致，那么该词项也能通过给定类型的检查 [CHK]。Lambda 抽象只能被检查为函数类型 [LAM]。我们在扩展后的上下文中检查 Lambda 抽象的函数体 *(body)*。

请注意，这些规则几乎完全是语法制导 *(syntax-directed)* 的：尽管连接可检查项和可推断项的规则 [CHK] 似乎可以匹配任何词项，然而规则中并没有用于推断 Lambda 抽象的类型的规则，也没有显式地检查类型注解、变量或应用的规则。因此，这些规则可以被很容易地转换为语法制导的算法。

以下是根据上述规则推出的两个示例 `id` 和 `const` 的类型判断 *(type judgement)*：

$$\alpha :: *,\ y :: \alpha \qquad \vdash (\mathtt{id} :: \alpha \to \alpha)\, y :: \alpha$$

$$\alpha :: *,\ y :: \alpha,\ \beta :: * \vdash (\mathtt{const} :: (\beta \to \beta) \to \alpha \to \beta \to \beta)\ \mathtt{id}\ y :: \beta \to \beta$$

[4]译注：“输入类型”原作“定义域 *(domain)*”，“返回类型”原作“值域 *(range)*”。“值域”这一术语并不严谨，既可指陪域/到达域 *(codomain)*，亦可指像/像集 *(image/image set)*。且在类型论语境下，将集合 *(Set)* 与类型 *(Type)* 混为一谈，则容易引出许多的危险。简单起见，译者扬弃了这两个术语，统一使用“输入类型”和“返回类型”。至于相关理论知识，本文限于篇幅不再赘述，感兴趣的读者可自行探索。

2.4. 实现

现在，我们给出一个$\lambda_{\to}$的 Haskell 实现。我们提供了一个用于对类型良定的 *(well-typed)* 词项求值的求值器，以及一个用于对$\lambda_{\to}$词项进行类型检查的函数。该实现与我们刚才介绍的形式化描述非常吻合。

规则的实现方式有着相当的自由度。我们选择的实现方式既能让我们紧密地遵循类型系统，又能将技术上的心智负担降到最低，让我们能够专注于算法的本质。接下来，我们会简要讨论我们的设计决策，并给出一些替代方案。需要指出的是，对于依值类型的实现来说，任何一种选择都不是必然的。

2.4.1. 变量和值的表示

[5]有很多不同的方法可以被用于表示约束变量 *(bound variable)*，每种方法都各有优劣。为了最大限度地发挥优势，我们在实现的不同地方选择不同的表示方法。

我们使用德布鲁因索引 *(de Bruijn indices)* 表示局部约束变量：变量由数字而非字符串或字母表示，数字的含义是变量出现的位置和引入它的约束符 *(binder)*[6] 之间隔了多少层约束符。例如，使用德布鲁因索引，`id` 可以写成 $\lambda \to 0$，而 `const` 可以写成 $\lambda \to \lambda 1$。这一表示法的优势在于不需要重命名变量——词项之间的 α-等价性 *(α-equivalence)*[7] 可以简化为语法相等性 *(syntactic equality)*。

德布鲁因索引的缺点则是处理带有自由变量的词项相当麻烦。我们可以用未被 Lambda 约束的索引来表示自由变量，但这些索引是相对的 *(relative)* ——当我们遍历每个词项、进入 Lambda 表达式内部时，这些索引必须相应地更新。

因此，我们使用绝对 *(absolute)* 指涉——也就是名称 *(name)* ——来表示自由变量。这种为局部变量使用数字、为相对于当前词项的全局变量使用变量名的做法称为局部无名 *(locally nameless)* 表示法 [23, 13]。

最后，我们使用高阶抽象语法 *(high-order abstract syntax)* 表示值：使用 Haskell 函数来表示函数值。这样做的优势在于我们可以利用 Haskell 的函数应用，不必自己实现替换，也不用担心名称捕获 *(name capture)*。这种方法的一个小缺点是 Haskell 函数既不能显示，也不能进行相等性 *(equality)* 检查。幸运的是，通过将值引用 *(quote)* 回具体的表示形式，这一缺陷可以被轻易缓解。在我们定义完求值器和类型检查器之后，我们会进一步讨论引用的具体机制。

2.4.2. 分离可推断项和可检查项

正如我们在图 3 的$\lambda_{\to}$的类型规则中所揭示的，我们区分了可以读出类型的项（称为可推断项）和需要进行类型检查的项（可检查项 *checkable term*）。可检查项和可推断项之间的这种语法区分至少可以追溯到 Pierce 和 Turner 的研究 [22]。

另一种做法是，我们可以要求抽象语法中的每个 Lambda 绑定变量都必须显式标注类型——这样我们就只拥有可推断项了。然而，能够为任意词项标注类型是非常有用的。既然有了通用的标注机制，就不再需要强制要求每个 Lambda 绑定变量都带类型注解了。事实上，允许未标注的 Lambda 在不增加额外成本的情况下提供了很大的便利：形如 $e\,(\lambda x \to e')$ 的函数应用可以在没有类型标注的情况下被处理，因为 x 的类型是由 e 的类型决定的。

---

[5]译注：本小节标题原为 “Representing bound variables”, 直译为“约束变量的表示”。然实则本小节讨论内容不仅限于约束变量的表示，故改。

[6]译注：在这里，约束符指的就是 λ。之后会加入 $\forall/\Pi$。

[7]译注：“α-equivalence” 原作 α-“equality”。

2.4.3. 抽象语法

我们为可推断项 (`Term↑`) 、可检查项 (`Term↓`) 和名称 (`Name`) 引入以下数据类型:

```
data Term↑
  = Ann    Term↓ Type
  | Bound  Int
  | Free   Name
  | App    Term↑ Term↓
  deriving (Show, Eq)

data Term↓
  = Inf  Term↑
  | Lam  Term↓
  deriving (Show, Eq)

data Name
  = Global  String
  | Local   Int
  | Quote   Int
  deriving (Show, Eq)
```

带类型注解的词项以 `Ann` 表示。如前所述，我们用整数表示约束变量 (`Bound`) ，用名称表示自由变量 (`Free`) 。那些通常指涉全局实体的名称使用字符串（`Global`)。当类型检查算法[8]进入一层约束符[9] 时，我们需要将约束符引入的约束变量临时转换成一个自由变量，我们用 `Local` 表示这类变量。进行引用时，我们使用 `Quote` 构造子。构造子 `App`[10] 表示应用。

可推断项通过构造子 `Inf` 嵌入到可检查项中。Lambda 抽象（因为我们用了德布鲁因索引，所以不会引入显式的变量）使用 `Lam` 表示。

类型只有两种：类型标识符 (`TFree`) 和函数箭头 (`Fun`) 。我们为类型标识符复用 `Name` 数据类型。在$\lambda_{\to}$中，类型层面上不会有约束变量，所以不需要 `TBound` 构造子。

```
data Type
  = TFree  Name
  | Fun    Type Type
  deriving (Show, Eq)
```

值包括 Lambda 抽象 (`VLam`) 和中性项 (`VNeutral`)。

```
data Value
   = VLam      (Value -> Value)
   | VNeutral  Neutral
```

正如我们在讨论高阶抽象语法时所描述的，我们使用类型为 `(Value -> Value)` 的 Haskell 的函数表示函数值。例如，词项 `const` 求值后就会得到值 `VLam (\x -> VLam (\y -> x))`。

表示中性项的数据类型和形式化的抽象语法完全吻合。一个中性项要么是一个自由变量 (`NFree`)，要么是中性项对值的应用 (`NApp`)。

---

[8]译注：原文无此“类型检查”，依后文补。

[9]译注：原文为“passing a binder”。因为 pass 既可以表示“通过”又可以表示“传递”，同时又少了递归的意思，因此此处意译为“进入”。

[10]译注：原文用符号是中缀运算符“`:@:`”，译者依个人好恶改。

```haskell
data Neutral
  = NFree  Name
  | NApp   Neutral Value
```

我们引入一个函数 `vfree`，用于创建对应于自由变量的值：

```haskell
vfree :: Name -> Value
vfree n = VNeutral (NFree n)
```

2.4.4. 求值

用于求值的代码在图 4 中给出。函数 `eval↑` 和 `eval↓` 实现了可推断项和可检查项的大步求值规则。将代码与图 1 对比，足见这一实现直截了当。

替换是通过传递一个包含值的环境 `Env` 来实现的。因为约束变量是用数字表示的，`Env` 可以被简单地实现成一个列表 `[Value]`，其中第 i 个元素对应着变量 `Bound i`。当进入一层约束符时，我们向环境中（列表的头部）添加一个新元素；当遇到变量 `Bound` 时，我们使用 Haskell 的 `!!` 运算符从列表中拿出正确的元素。

而对于 Lambda 抽象（`Lam`），我们引入一个 Haskell 函数，该函数首先将约束变量 x 添加到环境中，然后对函数体求值。

```haskell
type Env = [Value]

eval↑ :: Term↑ -> Env -> Value
eval↑ (Ann   e _ ) d = eval↓ e d
eval↑ (Free  x   ) d = vfree x
eval↑ (Bound i   ) d = d !! i
eval↑ (App   e e') d = vapp (eval↑ e d) (eval↓ e' d)

vapp :: Value -> Value -> Value
vapp (VLam f    ) v = f v
vapp (VNeutral n) v = VNeutral (NApp n v)

eval↓ :: Term↓ -> Env -> Value
eval↓ (Inf i) d = eval↑ i d
eval↓ (Lam e) d = VLam (\x -> eval↓ e (x : d))
```

图 4　$\lambda_{\to}$求值器的实现

2.4.5. 上下文

在着手实现类型检查之前，我们定义类型上下文。上下文以（反向的）列表形式实现，将名称与 $*$ (`HasKind Star`) 或者一个类型（`HasType τ`）关联起来。上下文的扩展自然是通过列表的 `cons` 操作 (`:`) 实现的；在上下文中查找名称则是使用 Haskell 的标准列表函数 `lookup` 进行的。

```haskell
data Kind = Star
  deriving (Show)

data Info
  = HasKind  Kind
  | HasType  Type
  deriving (Show)

type Context = [(Name, Info)]
```

2.4.6. 类型检查

现在我们实现图 3 中所描述规则，代码如图 5 所示。类型检查算法可能失败，而为了优雅地处理错误，该算法使用 `Result` 单子 *(monad)* 返回结果。简单起见，在本演示中，我们使用标准的错误单子：

```
type Result α = Either String α
```

并使用函数 `throwError :: String -> Result α` 报告错误。

用于可推断项的函数 `type↑` 返回一个类型，而用于可检查项的函数 `type↓` 接受一个类型作为输入，并返回 `()`。类型的良构性由函数 `kind↓` 检验。函数定义中的每个分支都直接对应一条规则。

```
kind↓ :: Context -> Type     -> Kind -> Result ()
kind↓ Γ (TFree x) Star
  = case lookup x Γ of
      Just (HasKind Star) -> pure ()
      Nothing             -> throwError "unknown Identifier"
kind↓ Γ (Fun κ κ') Star
  = do kind↓ Γ κ  Star
       kind↓ Γ κ' Star

type↑0 :: Context -> Term↑ -> Result Type
type↑0 = type↑ 0

type↑ :: Int -> Context -> Term↑ -> Result Type
type↑ i Γ (Ann e τ)
  = do kind↓ Γ τ Star
       type↓ i Γ e τ
       pure τ
type↑ i Γ (Free x)
  = case lookup x Γ of
      Just (HasType τ) -> pure τ
      Nothing          -> throwError "unknown identifier"
type↑ i Γ (App e e')
  = do σ <- type↑ i Γ e
       case σ of
         Fun τ τ' -> do type↓ i Γ e' τ
                        pure τ'
         _        -> throwError "illegal application"

type↓ :: Int -> Context -> Term↓ -> Type -> Result ()
type↓ i Γ (Inf e) τ
  = do τ' <- type↑ i Γ e
       unless (τ == τ') (throwError "type mismatch")
type↓ i Γ (Lam e) (Fun τ τ')
  = type↓ (i + 1) ((Local i, HasType τ) : Γ)
          (subst↓ 0 (Free (Local i)) e) τ'
type↓ i Γ _ _
  = throwError "type mismatch"
```

图 5　$\lambda_{\to}$类型检查器的实现[11]

[11]译注：`pure` 原作 `return`，依现代 Haskell 实践改。

类型检查函数接受一个整数，该整数表示我们进入了多少层约束符。初次调用时，这个参数应该为 0，我们为此提供了包装函数 type↑0。我们用这个整数模拟处理约束变量时的类型规则：在 Lambda 抽象的类型规则 [LAM] 中，在检查函数体时，我们将约束变量添加到上下文中——而我们的实现正是这么做的。计数器 i 表示我们进入了多少层约束符，因此 Local i 总是一个可用的新名称。我们先将 Local i 添加到上下文 Γ 中，将其与约束变量关联起来，然后对函数体作类型检查。因为我们把一个约束变量变成了一个自由变量，所以我们要对函数体进行相应的替换。类型检查器不会遇到约束变量，因此函数 type↑ 没有用于处理 Bound 的分支。

请注意，在检查可推断项时，类型的等价性检查是通过对数据类型 Type 进行简单的语法相等性判断来实现的。我们的类型检查器不执行合一 *(unification)* 。

用于替换的代码如图 6 所示，替换算法同样包含两个函数：用于可推断项的函数 subst↑ 和用于可检查项的函数 subst↓。整数参数表示要替换哪个变量。在 Bound 的分支中，我们要检查遇到的变量是不是需要替换的变量。在 Lam 的分支中，我们要增加 i 的值，因为在 Lambda 抽象的函数体中，待替换变量是由更高的编号指涉的。

```
subst↑ :: Int -> Term↑ -> Term↑ -> Term↑
subst↑ i r (Ann   e τ ) = Ann (subst↓ i r e) τ
subst↑ i r (Bound j   ) = if i == j then r else Bound j
subst↑ i r (Free  y   ) = Free y
subst↑ i r (App   e e') = App (subst↑ i r e) (subst↓ i r e')

subst↓ :: Int -> Term↑ -> Term↓ -> Term↓
subst↓ i r (Inf e) = Inf (subst↑ i r e)
subst↓ i r (Lam e) = Lam (subst↓ (i + 1) r e)
```

图 6　$\lambda_{\to}$替换算法的实现

2.4.7. 引用

我们的简单类型 Lambda 演算求值器就快完成了。目前还剩一个小问题：求值器返回的是 Value，而当下我们无法打印 Value 类型的值，也不能对 Value 作相等性判断。这是因为 Value 类型的 VLam 构造子接受的是一个 Haskell 函数，我们不能像对其他类型那样简单地为其派生 Show 和 Eq[12]。因此，如果我们要重新得到一个值的内部结构，我们就需要 quote 函数。代码在图 7 中给出。

```
quote0 :: Value -> Term↓
quote0 = quote 0

quote :: Int -> Value -> Term↓
quote i (VLam     f) = Lam (quote (i + 1) (f (vfree (Quote i))))
quote i (VNeutral n) = Inf (neutralQuote i n)

neutralQuote :: Int -> Neutral -> Term↑
neutralQuote i (NFree x  ) = boundfree i x
neutralQuote i (NApp  n v) = App (neutralQuote i n) (quote i v)
```

图 7　$\lambda_{\to}$中的引用

函数 quote 接受一个整数参数，该参数用来记录我们已经进入了多少层约束符。初次调用时，这个参数总是为 0，我们为此提供了包装函数 quote0。

[12]译注：此处译文结构较原文有较大变化。

如果传给 quote 的值是一个 Lambda 抽象 (VLam)，我们生成一个新变量 Quote i，把它传给 Haskell 函数 f，然后对函数 f 返回的值递归调用 quote (i + 1)[13]。这里，我们使用接受一个 Int 的构造子 Quote 来确保新创建的名称不会与值中的其他名称冲突。

如果值是一个中性项（VNeutral，也就是将自由变量应用于零个或多个[14]其他值），则使用 neutralQuote 函数处理[15]。boundfree 函数被用于检查出现在中性项[16]头部的变量是一个自由变量，还是一个 Quote ——也就是约束变量：

```
boundfree :: Int -> Name -> Term↑
boundfree i (Quote k) = Bound (i - k - 1)
boundfree i x         = Free x
```

用例子来理解函数是如何被引用的再好不过了。与词项 const 对应的值是 VLam (\x -> VLam (\y -> x))。对其应用 quote0 可得：

```
  quote 0 (VLam (\x -> VLam (\y -> x)))
= Lam (quote 1 (VLam (\y -> vfree (Quote 0))))
= Lam (Lam (quote 2 (vfree (Quote 0))))
= Lam (Lam (neutralQuote 2 (NFree (Quote 0))))
= Lam (Lam (Bound 1))
```

当 quote 进入一层约束符时，我们为约束变量引入一个临时的名称。为确保在引用过程中该名称不会和其他名称发生冲突，我们只使用 Quote 构造子。如果约束变量在函数体中出现过，那么我们迟早会抵达这些出现的地方。此时即可根据引入和观测到 Quote 构造子之间经过的约束符数量，生成德布鲁因索引。

2.4.8. 例子

略。请自行参照原文。

[13]译注：原文为 “The value resulting from the function application is then quoted at level i + 1.” 译者不想翻译这个 “level”，因为这个词的这种用法只出现了一次，且在后文中还有别的意思。同时，尽管译者在翻译所有 “refer” 的地方都用了“指涉”，“被引用”听起来仍然很别扭。故利用 Haskell 的柯里化特性将其意译。

[14]译注：原文无此“零个或多个”，依前文补。

[15]译注：原文为 “If the value is a neutral term (hence an application of a free variable to other values), the function neutralQuote is used to quote the arguments.” 此处原文显然逻辑不通（我们对中性项的参数调用的是 quote 而非 neutralQuote），故译者完全重写了这一句。

[16]译注：“中性项”原作“应用”。

3. 依值类型

在本节中，我们将修改简单类型 Lambda 演算的类型系统，使其成为依值类型 Lambda 演算，简称λ_{Π}。在本节的开头，我们会先讨论这些更改中的两个核心思想。接着，我们会给出抽象语法、求值和类型规则，并将这些规则中与$\lambda_{\rightarrow}$不同的部分用高亮标出。最后，我们在本节中讨论如何调整实现。

3.0.1. 依值函数空间

在 Haskell 这样的语言中，我们可以定义多态函数，例如恒等函数：

```
id :: ∀α. α -> α
id = \x -> x
```

利用多态，我们可以避免为不同类型的数据——例如整数和布尔值——反复编写同样的函数。若将多态解释为一种类型抽象，则可使其变成一种显式 *(explicit)* 行为。这样一来，恒等函数就接受两个参数：一个类型 α，和一个类型为 α 的值。而在调用这个恒等函数时，必须显式地用一个类型将其实例化 *(instantiate)*：

```
id :: ∀α. α -> α
id = \(α :: *) (x :: α) -> x

id Bool True :: Bool
id Int 3 :: Int
```

因此，多态允许类型对类型进行抽象。为什么我们还要做与此不同的事情呢？考虑以下数据类型：

```
data Vec0 α = Vec0
data Vec1 α = Vec1 α
data Vec2 α = Vec2 α α
data Vec3 α = Vec3 α α α
```

显然，这些类型遵循着某种模式。我们希望能有一个单一的类型族 *(family)*，并按元素数量进行索引：

$$\forall \alpha :: * . \forall n :: \texttt{Nat} . \texttt{Vec}\ \alpha\ n$$

但在 Haskell 里我们不能简单地这么做。问题在于，这个类型 `Vec` 是对值 n 抽象的。

依值函数空间 “∀” 扩展了通常的函数空间 “→”，它允许函数返回值的类型依赖于输入值[17]。Haskell 中的参数化多态 *(parametric polymorphism)* 可以看作是依值函数的一种特例[18]，这也是我们使用符号 “∀” 的动机[19]。但与参数化多态不同的是，依值函数空间不止能对类型进行抽象。上面的 `Vec` 类型是一个有效的依值类型。

值得注意的是，依值函数空间是通常函数空间的泛化。例如，我们可以为应用于上述 `Vec` 类型的恒等函数 `id` 添加这样的类型注解：

$$\forall \alpha :: * . \forall n :: \texttt{Nat} . \forall v :: \texttt{Vec}\ \alpha\ n . \texttt{Vec}\ \alpha\ n$$

[17]译注：原文为 “The dependent function space ‘∀’ generalizes the usual function space ‘→’ by allowing the range to depend on the domain”，译者依个人理解进行了意译。例如，依值类型的函数 `zeros` :: $\forall \alpha :: * . \forall n ::$ `Nat` . `Vec` $\alpha\ n$ 返回的类型 `Vec` $\alpha\ n$ 显然依赖于参数 n 的值。

[18]译注：因为类型也可以被视作一种值。下一小节“一切皆项”将详细展开。

[19]类型学家称依值函数类型为 Π-类型，并且会这么写：$\Pi \alpha : * . \Pi n :$ `Nat` . `Vec` $\alpha\ n$。这也是为什么我们将依值类型 Lambda 演算称为λ_{Π}。

注意类型 v 并没有在返回类型中出现：而这就是非依值函数空间，这对 Haskell 程序员来说已经很熟悉了。与其在这种地方引入像 v 这样不必要的变量，我们不妨为非依值的情况使用常规的函数箭头。因此上面的类型注解也可以写成：

$$\forall \alpha :: * \,.\, \forall n :: \texttt{Nat} \,.\, \texttt{Vec}\ \alpha\ n \to \texttt{Vec}\ \alpha\ n$$

在 Haskell 中，程序员可以一定程度上“模拟”出依值类型空间 [11]，例如在类型层面上定义自然数（也就是定义 `Zero` 和 `Succ` 这样的数据类型）。然而，类型层面和值层面的自然数之间，终究是云泥之异，难越鸿沟，程序员不得不在两个层面之间重复大量的概念。尽管利用高级的类型类编程技巧可以将值层面的东西提升到类型层面，但要用这些类型作计算需要许多额外的努力。而使用依值类型，我们可以直接用值将类型参数化，并且仍然使用常规的求值规则——我们很快会看到的。

3.0.2. 一切皆词项

允许值在类型中自由出现打破了词项、类型和种类 *(kind)* 的分界。不同的层级之间不再有语法上的差异，因为一切皆词项。在 Haskell 中，符号 “`::`” 将实体与不同的语法层级关联起来：在 `0 :: Nat` 中，`0` 在语法上是一个值，而 `Nat` 是一个类型；在 `Nat :: *` 中，`Nat` 在语法上是一个类型，而 `*` 是一个种类。而现在，`*`、`Nat` 和 `0` 都是词项。`0 :: Nat` 和 `Nat :: *` 依然成立，但符号 “`::`” 现在是将词项与词项关联起来。我们还是称 $\rho :: *$ 中的 ρ 为类型，也仍然会称呼 $*$ 为种类，但所有这些实体现在都处于同一个语法层级了。这样一来，所有语言结构就在任何地方都可用了。特别地，我们现在可以在类型和种类的层面进行抽象和应用了。

现在我们已经熟悉了依值类型系统的核心概念。接下来，我们将讨论实现λ_{Π}要对$\lambda_{\to}$作什么样的修改。

3.1. 抽象语法

我们不再需要区分词项、类型和种类了，所有层级中的所有结构现在都被整合到了词项语言中：

$$\begin{array}{rll}
e, \rho, \kappa ::= & e :: \rho & \text{带类型注解的词项} \\
\mid & * & \text{类型之类型} \\
\mid & \forall x :: \rho \,.\, \rho' & \text{依值函数空间} \\
\mid & x & \text{变量} \\
\mid & e\ e' & \text{应用} \\
\mid & \lambda x \to e & \text{Lambda 抽象}
\end{array}$$

抽象语法中与 2.1 节有所变动之处以高亮显示。

现在我们也使用符号 ρ 和 κ 指涉作为类型和种类的词项。

原本位于类型和种类的语法规则中的构造现在也被导入了进来。种类 $*$ 现在是一个词项。依值函数空间涵盖了原本的箭头种类和箭头类型。类型变量和词项变量现在也已合二为一。

3.2. 求值

新增的求值规则如图 8 所示。除了与新添加的两个构造相关的规则外，所有规则都和$\lambda_{\to}$别无二致。你也许会感到惊讶，求值竟然扩展到了类型！但这正是我们想要的：依值类型的威力正源于将值和类型混合的能力，这样一来我们就能在类型层级上定义函数、进行计算。

$$\frac{}{* \Downarrow *} \qquad \text{对 } * \text{ 求值会得到它自身}$$

$$\frac{\rho \Downarrow \tau \quad \rho' \Downarrow \tau'}{\forall x :: \rho \,.\, \rho' \Downarrow \forall x :: \tau \,.\, \tau'} \qquad \text{递归求值依值函数空间}$$

图 8　λ_{Π}新增的求值规则

相对而言，我们新加入的构造在计算过程中就不那么有趣了：对 $*$ 求值会得到它自身；在依值函数空间中，我们递归地对输入类型和返回类型求值。我们必须相应地扩展值的抽象语法：

$$\begin{array}{lll} v, \tau ::= n & & \text{中性项} \\ \quad \mid * & & \text{类型之类型} \\ \quad \mid \forall x :: \tau \,.\, \tau' & & \text{依值函数空间} \\ \quad \mid \lambda x \to v & & \text{Lambda 抽象} \end{array}$$

现在我们使用符号 τ 指涉作为类型的值。

3.3. 类型系统

在接触类型规则之前，我们得先回顾一下上下文。因为现在一切皆词项，上下文的抽象语法和有效性规则（图 9，与图 2 对比）自然也相应地简化了。

$$\begin{array}{lll} \Gamma ::= \varepsilon & & \text{空上下文} \\ \quad \mid \Gamma, x :: \tau & & \text{添加一个变量} \end{array}$$

$$\frac{}{\mathrm{valid}(\varepsilon)} \qquad \frac{\mathrm{valid}(\Gamma) \quad \Gamma \vdash \tau ::_{\downarrow} *}{\mathrm{valid}(\Gamma, x :: \tau)}$$

图 9　λ_{Π}的类型上下文及其有效性

上下文中现在只有一种形式的条目，也就是说我们总是假设变量有其类型。注意我们在上下文中存储的都是求值后的类型。非空上下文有效性规则中的前提条件 $\Gamma \vdash \tau ::_{\downarrow} *$ 指涉的不再是一个特殊的类型良构性的判断，而是我们即将定义的类型规则——我们不再需要为类型设置专门的良构性规则了。这一前提条件尤其确保了 τ 中不含有未知的变量。

类型规则在图 10 中给出。类型规则现在与上下文、词项和值相关——所有类型都被尽可能早地求值。和之前一样，我们高亮了规则中和图 3 不同的部分。我们仍然为推断和检查使用不同的符号：当一个类型是输出时，我们用 $::_{\uparrow}$；当一个类型是输入时，我们用 $::_{\downarrow}$。新的构造 $*$ 和 $\forall$ 属于我们能推断其类型的那一类。和之前讨论$\lambda_{\to}$的时候一样，我们假定类型规则中出现的所有上下文都是有效的。

$$\frac{\Gamma \vdash \rho ::_\downarrow * \quad \rho \Downarrow \tau \quad \Gamma \vdash e ::_\downarrow \tau}{\Gamma \vdash (e :: \rho) ::_\uparrow \tau} \text{ [ANN]}$$

$$\frac{}{\Gamma \vdash * ::_\uparrow *} \text{ [STAR]} \qquad \frac{\Gamma \vdash \rho ::_\downarrow * \quad \rho \Downarrow \tau \quad \Gamma, x :: \tau \vdash \rho' ::_\downarrow *}{\Gamma \vdash \forall x :: \rho \,.\, \rho' ::_\uparrow *} \text{ [PI]}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x ::_\downarrow \tau} \text{ [VAR]} \qquad \frac{\Gamma \vdash e ::_\uparrow \forall x :: \tau \,.\, \tau' \quad \Gamma \vdash e' ::_\downarrow \tau \quad \tau'[x \mapsto e'] \Downarrow \tau''}{\Gamma \vdash e\, e' ::_\downarrow \tau''} \text{ [APP]}$$

$$\frac{\Gamma \vdash e ::_\uparrow \tau}{\Gamma \vdash e ::_\downarrow \tau} \text{ [CHK]} \qquad \frac{\Gamma, x :: \tau \vdash e ::_\downarrow \tau'}{\Gamma \vdash \lambda x \to e ::_\downarrow \forall x :: \tau \,.\, \tau'} \text{ [LAM]}$$

图 10　λ_Π的类型规则

带类型注解的词项的规则 [ANN] 有两项变动：注解 ρ 的种类检查规则不再指涉到类型良构性规则，而是指涉到类型检查规则；同时因为注解 ρ 现在未必是一个值，我们需要先对它求值，然后返回求值结果。

种类 $*$ 自身的类型就是 $*$ [STAR]。尽管这种选择存在一些理论上的反对意见（见第五节），我们相信就本文而言，相对于实现的简单性，这些反对意见无足轻重。

依值函数空间的规则 [PI] 和图 2 中箭头类型的良构性规则 [FUN] 有相似之处。依值函数的输入类型 ρ 和返回类型 ρ' 都必须归属于种类 $*$。与之前的规则 [FUN] 有所不同的是，ρ' 中可以包含[20] x，因此我们在检查 ρ' 的时候要将 $x :: \tau$（其中 τ 是对 ρ[21] 求值的结果）加入上下文 Γ。

在函数应用 [APP] 中，现在函数必须具有依值函数类型 $\forall x :: \tau \,.\, \tau'$。与普通函数类型不同的是，$\tau'$ 中可以包含 x。因此在应用的结果类型中，我们需要将 τ' 中出现的形参 x 替换成实参 e'。

对可推断项作检查的规则 [CHK] 还是跟以前一样：给定一个词项 e 和一个类型 τ，我们首先推断出 e 的类型，接着检查推断出的类型和期望的类型 τ 是否相等。然而，我们现在处理的是已求值的类型，因此这种相等性要比类型词项的语法相等性要强得多：不然要是 `Vec` $\alpha\, 2$ 和 `Vec` $\alpha\, (1 + 1)$ 表示的不是同一个类型，那就太不幸了。而我们的系统能识别出它们相等，因为两个类型的求值结果都是 `Vec` $\alpha\, 2$。

许多支持依值类型的类型系统都有一条这样的规则：

$$\frac{\Gamma \vdash e :: \rho \quad \rho =_\beta \rho'}{\Gamma \vdash e :: \rho'}$$

然而，这条被称作转换规则 *(conversion rule)* 的规则显然是非语法制导的，而区分可推断项和可检查项使得我们只在处理有显式类型注解的词项 [ANN] 时才需要应用这条转换规则。

最后一条规则 [LAM] 是用来检查 Lambda 抽象的。与之前不同的是，Lambda 抽象的类型现在是一个依值函数类型了，约束变量 x 不仅可能出现在函数体 e 中，还可能出现在返回类型 τ' 中[22]。因此在对函数体 e 作类型检查和对返回类型 τ' 作种类检查时，都要使用扩展过的上下文 $\Gamma, x :: \tau$。

总结下来，我们所作的所有修改都是围绕着我们在第 3 节开头引入的两个核心概念进行的：函数空间被泛化为依值函数空间；类型和种类也是词项。

[20]译注：“包含”原作“指涉”。
[21]译注：“ρ” 原作 “τ'”，但规则 [PI] 中并无 τ'，故依实际情况改写为 ρ。
[22]译注：后半句“还可能出现在返回类型 τ' 中”为译者补文。

3.4. 实现

我们给出的λ_Π的类型规则仍然是语法制导的、算法性的，所以$\lambda_\to$实现的总体结构可以被λ_Π复用。在接下来的部分里，我们会遍览实现的各个方面，但只讨论需要修改的部分。

3.4.1. 抽象语法

现在我们不再需要 `Type` 和 `Kind` 了。我们为 `Term` 添加了两个构造子，并将构造子 `Ann` 中出现的 `Type` 替换成了 `Term↓`：

```
data Term↑
  = Ann    Term↓ Term↓
  | Star
  | Pi     Term↓ Term↓
  | Bound  Int
  | Free   Name
  | App    Term↑ Term↓
```

我们也需要扩展用来表示值的类型：

```
data Value
  = VLam     (Value -> Value)
  | VStar
  | VPi      Value (Value -> Value)
  | VNeutral Neutral
```

和之前一样，我们用高阶抽象语法表示值，也就是用 Haskell 函数来表示约束结构[23]。我们用 `VPi` 表示新的约束结构 $\forall/\Pi$。在依值函数空间中，由 $\forall x : A \,.\, B$ 引入的约束变量 x 不会出现在变量自身的类型 A 中，但函数的返回类型 B 中却有可能包含 x[24]。因此，输入类型可以简单地用一个 `Value` 表示，而返回类型则要用 Haskell 函数 `Value -> Value` 表示[25]。

3.4.2. 求值

要适配求值器，我们只需为函数 `eval↑` 添加两个用于处理 `Star` 和 `Pi` 的新分支，如图 11 所示（$\lambda_\to$的求值器见图 4）。`Star` 的求值非常简单。对于 `Pi`，我们求值其输入类型和返回类型，且在求值返回类型时，我们需要将约束变量 x 添加到上下文中。

```
eval↑ Star       d = VStar
eval↑ (Pi τ τ') d = VPi (eval↓ τ d) (\x -> eval↓ τ' (x : d))

subst↑ i r (Ann e↓ τ) = Ann (subst↓ i r e↓) (subst↓ i r τ)
subst↑ i r Star       = Star
subst↑ i r (Pi τ τ')  = Pi (subst↓ i r τ) (subst↓ (i + 1) r τ')

quote i VStar     = Inf Star
quote i (VPi v f) = Inf (Pi (quote i       v)
                            (quote (i + 1) (f (vfree (Quote i)))))
```

图 11　λ_Π对求值、替换和引用的扩展

[23]译注：也就是 λ 和 $\forall/\Pi$。

[24]译注：此句为译者改写。原文为 “In the dependent function space, a variable is bound that is visible in the range, but not in the domain.”

[25]译注：不妨把 $\Pi x : A \,.\, B$ 看作 $\Pi x : A \,.\, B(x)$，即类型 B 是关于值 x 的函数。

3.4.3. 上下文

上下文将变量映射到其类型，而类型现在也在词项层级上。我们存储的是类型求值后的形式，因此我们如是定义类型和上下文：

```
type Type    = Value
type Context = [(Name, Type)]
```

3.4.4. 类型检查

现在我们逐一分析图 12 中的分支，你可以将其与图 5 对比。

```
type↑ :: Int -> Context -> Term↑ -> Result Type
type↑ i Γ (Ann e ρ)
  = do type↓ i Γ ρ VStar
       let τ = eval↓ ρ []
       type↓ i Γ e τ
       pure τ
type↑ i Γ Star
  = pure VStar
type↑ i Γ (Pi ρ ρ')
  = do type↓ i Γ ρ VStar
       let τ = eval↓ ρ []
       type↓ (i + 1) ((Local i, τ) : Γ)
             (subst↓ 0 (Free (Local i)) ρ') VStar
       pure VStar
type↑ i Γ (Free x)
  = case lookup x Γ of
      Just τ  -> pure τ
      Nothing -> throwError "unknown identifier"
type↑ i Γ (App e e')
  = do σ <- type↑ i Γ e
       case σ of
         VPi τ τ' -> do type↓ i Γ e' τ
                        pure (τ' (eval↓ e' []))
         _        -> throwError "illegal application"

type↓ :: Int -> Context -> Term↓ -> Type -> Result ()
type↓ i Γ (Inf e) v
  = do v' <- type↑ i Γ e
       unless (quote0 v == quote0 v') (throwError "type mismatch")
type↓ i Γ (Lam e) (VPi τ τ')
  = type↓ (i + 1) ((Local i, τ) : Γ)
          (subst↓ 0 (Free (Local i)) e) (τ' (vfree (Local i)))
type↓ i Γ _ _
  = throwError "type mismatch"
```

图 12　λ_Π类型检查器的实现

对于注解项 $e :: \rho$，我们首先用类型检查函数 `type↓` 检查类型注解 ρ 具有种类 $*$。接着我们对 ρ 求值，用求值结果 τ 对 e 作类型检查，如果类型检查成功，整个表达式的类型就是 τ[26]。注意我们假设 `type↑` 所处理的类型中没有非约束变量，因此我们总是传给 `eval↓` 一个空环境。

`Star` 求值后的类型是 `VStar`。

[26]译注：“τ” 原作 “v”，然此处显然为作者笔误。

对于依值函数类型 $\forall x :: \rho \,.\rho'$，我们首先对输入类型 ρ 作种类检查，然后将其求值为 τ[27]。接着，我们将值 τ 加入上下文，对返回类型 ρ' 作种类检查——这里的思路跟$\lambda_{\rightarrow}$和λ_{Π}中对 `Lam` 作类型检查的规则有异曲同工之妙。

对于函数应用 $e\,e'$，类型推断函数 `type↑` 现在会给出一个值。该值的形式必须是 `VPi τ τ'` ——也就是依值函数类型。在图 10 相应的类型规则中，返回类型 τ' 中的约束变量要用 e' 替换。而在实现中，`τ'` 是一个函数，替换是通过将 `τ'` 应用于（求值后的）`e'` 实现的。

在处理 `Inf` 时，我们必须对给定的 `v` 和推断出的 `v'` 作类型相等性判断。与类型规则不同的是，在 Haskell 中我们不能直接比较两个 `Value`。因此我们要用 `quote` 将这两个值引用回词项，然后再对词项作语法相等性判断。

对于 Lambda 抽象 $\lambda x \rightarrow e$，现在我们要求其具有依值函数类型 `VPi τ τ'`。和$\lambda_{\rightarrow}$一样，我们要在对函数体 e 作类型检查时将约束变量 x（类型为 τ）加入上下文；但与$\lambda_{\rightarrow}$不同的是，现在除了要用 `subst↓` 替换函数体 e 中的 x 之外，还要将 `τ'` 应用于 `(Local i)` 来替换 τ' 中的 x。

为此，我们还要扩展替换函数，使其能够遍历注解项中的类型，并能够处理新结构 `Star` 和 `Pi`，如图 11 所示。对于 `Star`，没有需要替换的东西。而对于 `Pi`，我们要在对返回类型作替换时增加计数器的值，因为我们进入了一层约束符。

3.4.5. 引用

现在只要再扩展引用函数 `quote`，我们的λ_{Π}实现就大功告成了。引用操作在λ_{Π}中比在$\lambda_{\rightarrow}$中更加重要，因为如我们所见，现在类型检查的过程中要作相等性检查，而相等性检查需要用到 `quote`。我们还是只需要处理新结构 `VStar` 和 `VPi`，如图 11 所示。

引用 `VStar` 会得到 `Star`。而因为依值函数类型是一个约束结构，引用 `VPi τ τ'` 的过程就类似于引用 `VLam`：在引用返回类型 `τ'` 时，我们将代表着函数返回类型的 Haskell 函数 `τ'` 应用于 `Quote i`，然后对结果应用 `quote (i + 1)`[28]。

3.5. 依值类型今何在?

现在我们已经实现了依值类型系统，但不幸的是，我们还没见到任何例子。

和之前一样，我们为λ_{Π}的类型检查器写了一个小的解释器，我们可以用它定义实体、执行类型检查。例如，我们可以这样定义和检查多态恒等函数（类型参数是显式的）：

$$
\begin{aligned}
&\rangle\!\rangle\ \textbf{let}\ id = (\lambda\alpha x \rightarrow x) :: \forall(\alpha :: *)\,.\,\alpha \rightarrow \alpha \\
&id :: \forall(x :: *)\,(y :: x)\,.\,x \\
&\rangle\!\rangle\ \textbf{assume}\ (\mathrm{Bool} :: *)\,(\mathit{False} :: \mathrm{Bool}) \\
&\rangle\!\rangle\ id\ \mathrm{Bool} \\
&\lambda x \rightarrow x :: \forall x :: \mathrm{Bool}\,.\,\mathrm{Bool} \\
&\rangle\!\rangle\ id\ \mathrm{Bool}\ \mathit{False} \\
&\mathit{False} :: \mathrm{Bool}
\end{aligned}
$$

相比于简单类型，我们能做的事情更多，但其中并没有非用依值类型不可的。不幸地，尽管我们已经有了依值类型的框架，如果我们不为我们的语言添加一些特定的数据类型的话，我们就写不出什么有意思的程序。

[27]译注：原文为 “For a dependent function type, we first kind-check the domain τ. Then the domain is evaluated to v.” 此处显然为作者笔误。

[28]译注：原文为 “To quote the range, we increment the counter i, and apply the Haskell function representing the range to `Quote i`.”

4. λ_Π 之后

在 Haskell 中，数据类型是通过特殊的 `data` 声明引入的：

```
data Nat = Zero | Succ Nat
```

这一声明引入了新类型 `Nat`，以及它的两个构造子 `Zero` 和 `Succ`。而在本节中，我们将探讨如何用数据类型——例如自然数——来扩展我们的语言。

显然，我们需要添加类型 `Nat` 以及它的两个构造子。但我们要如何定义诸如加法的用来操作数字的函数呢？在 Haskell 中，我们可以让函数对参数作模式匹配，并递归调用自身来处理更小的数：

```
plus :: Nat -> Nat -> Nat
plus Zero     n = n
plus (Succ k) n = Succ (plus k n)
```

然而，我们的演算里既没有模式匹配，函数也不能递归。这下可怎么定义 `plus` 呢？

在 Haskell 中，我们可以用 `fold` [16] 为数据类型定义递归函数。所以，相比于引入会带来一堆问题的模式匹配和递归，不如用 `fold` 来为自然数定义函数。不过在依值类型的世界，我们可以定义一个更通用的 `fold`，我们称之为*消去子 (eliminator)*。

自然数的 `fold` 函数具有如下类型：

$$\texttt{foldNat} :: \forall \alpha :: * . \alpha \to (\alpha \to \alpha) \to \texttt{Nat} \to \alpha$$

这应该很熟悉。不过，在依值类型环境中，类型 α 无需在自然数的各个构造子之间保持一致[29]，因此我们使用 $m :: \texttt{Nat} \to *$ 而非 $\alpha :: *$。如此，我们可以给出 `natElim` 的类型[30]：

$$\begin{array}{lll}
\texttt{natElim} :: & \forall m :: \texttt{Nat} \to * & \text{动机} \\
& \bullet \ m\ \texttt{Zero} & \text{基准情况} \\
& \to (\forall l :: \texttt{Nat} . m\, l \to m\, (\texttt{Succ}\ l)) & \text{归纳情况} \\
& \to \forall k :: \texttt{Nat} & \text{要消去的数字} \\
& \bullet \ m\, k & \text{返回类型}
\end{array}$$

消去子的第一个参数有时也被称为*动机 (motive)* [10]，它描述了我们尽除凡数的缘由[31]。第二个参数对应*基准情况 (base case)*，它的类型是我们把自然数 `Zero` 传给 m 得到的。第三个参数对应*归纳情况 (inductive case)*，它的类型是我们把自然数 `Succ` l 传给 m 得到的。在归纳情况中，我们必须描述如何从自然数 l 和类型 $m\, l$ 构造出一个类型为 $m\, (\texttt{Succ}\ l)$ 的东西。最后一个参数就是我们要消去的数字。`natElim` 的返回类型就是 $m\, k$。

借助上述讨论中获得的启发，我们可以为自然数给出如图 13 和 14 所示的求值规则和类型规则。注意我们不把 `natElim` 当作一个函数，而是当作一个词项形成符 *(term former)*：它不能像函数那样部分应用，必须参数饱和才能构成有效的词项。求值规则所说明的行为和 `fold` 如出一辙。对自然数 k 求值可能会得到一个中性项，我们需要一条规则来处理这种情况：当遇到这种情况时，消去子的求值会卡住，而求值结果中会包含（无法被 β-规约的）消去子 `natElim` 的应用。

[29]译注：例如，普通的 `fold` 所接受的函数必须为 `Nat` 的两个构造子 `Zero` 和 `Succ` 返回同一个类型的值；消去子 `natElim` 所接受的函数则不然。

[30]译注：译者改写了公式，使 `natElim` 的每个输入类型和返回类型都独占一行，并在每个类型旁都加了注解。

[31]译注：就是说，“你想把自然数变成什么别的类型？”

$$\frac{}{\texttt{Nat} \Downarrow \texttt{Nat}} \qquad \frac{}{\texttt{Zero} \Downarrow \texttt{Zero}} \qquad \frac{k \Downarrow l}{\texttt{Succ}\ k \Downarrow \texttt{Succ}\ l}$$

$$\frac{k \Downarrow \texttt{Zero} \quad mz \Downarrow v}{\texttt{natElim}\ m\ mz\ ms\ k \Downarrow v} \qquad \frac{k \Downarrow \texttt{Succ}\ l \quad ms\ l\ (\texttt{natElim}\ m\ mz\ ms\ l) \Downarrow v}{\texttt{natElim}\ m\ mz\ ms\ k \Downarrow v}$$

$$\frac{k \Downarrow n}{\texttt{natElim}\ m\ mz\ ms\ k \Downarrow \texttt{natElim}\ m\ mz\ ms\ n}$$

图 13　自然数的求值

$$\frac{}{\Gamma \vdash \texttt{Nat} ::\uparrow *} \qquad \frac{}{\Gamma \vdash \texttt{Zero} ::\uparrow \texttt{Nat}} \qquad \frac{\Gamma \vdash k ::\downarrow \texttt{Nat}}{\Gamma \vdash \texttt{Succ}\ k ::\uparrow \texttt{Nat}}$$

$$\frac{\begin{array}{c}\Gamma \vdash m ::\downarrow \texttt{Nat} \to * \\ m\ \texttt{Zero} \Downarrow \tau \quad \Gamma \vdash mz ::\downarrow \tau \\ \forall l :: \texttt{Nat}\,.\, m\ k \to m\ (\texttt{Succ}\ l) \Downarrow \tau' \quad \Gamma \vdash ms ::\downarrow \tau' \\ \Gamma \vdash k ::\downarrow \texttt{Nat}\end{array}}{\Gamma \vdash \texttt{natElim}\ m\ mz\ ms\ k ::\uparrow m\ k}$$

图 14　自然数的类型规则

4.1. 自然数的实现

总结一下，向我们的语言中添加自然数需要添加三样东西：类型 `Nat`，构造子 `Zero` 和 `Succ`，以及消去子 `natElim`。我们将扩展抽象语法，并为求值和类型检查函数添加新的分支。这些新的分支不需要修改现有代码，所以我们只关注新代码片段。

4.1.1. 抽象语法

我们如是扩展抽象语法：

```
data Term↑ = ...
  | Nat
  | Zero
  | Succ Term↓
  | NatElim Term↓ Term↓ Term↓ Term↓
```

我们向 `Term↑` 添加了四个构造子：数据类型 `Nat`，数据构造子 `Zero` 和 `Succ`，以及消去子 `NatElim`。构造子 `NatElim` 是完全应用的：它不接受更多的参数。

4.1.2. 求值

之前，值只有两种，Lambda 抽象和“卡住”的应用。而现在，我们需要扩展用于表示值的数据类型来适配自然数的构造子：

```
data Value = ...
  | VNat
  | VZero
  | VSucc Value
```

引入消去子会让求值变得复杂。当要消去的自然数无法求值成构造子时，消去子的求值就会卡住。相应地，我们要为这种情况扩展用于表示中性项的数据类型：

```
data Neutral = ...
  | NNatElim Value Value Value Neutral
```

图 15 中求值的实现严格遵循了图 13 中的规则。

```
eval↑ Nat       d = VNat
eval↑ Zero      d = VZero
eval↑ (Succ k) d = VSucc (eval↓ k d)
eval↑ (NatElim m mz ms k) d
  = let mzVal = eval↓ mz d
        msVal = eval↓ ms d
        rec kVal = case kVal of
          VZero      -> mzVal
          VSucc l    -> vapp (vapp msVal l) (rec l)
          VNeutral k -> VNeutral (NNatElim (eval↓ m d) mzVal msVal k)
          _          -> error "internal: eval natElim"
    in rec (eval↓ k d)
```

图 15　为自然数扩展求值器

其中，消去子是唯一值得注意的情况。本质上，消去子会被求值为一个 Haskell 函数 `rec`，其行为合乎预期：若待消去的数字求值为 `VZero`，则求值基本情况 `mz`；若数字求值为 `VSucc` l，则将 `ms` 应用于前驱 l 和对消去子的递归调用 `rec` l；最后，若数字求值为中性项，则整个 `natElim` 的求值结果亦为中性项。如果待消去的值既不是自然数也不是中性项，这在类型检查阶段就会导致类型错误。因此，最后的兜底分支永远都不会被执行。

4.1.3. 类型

图 16 包含了用于处理自然数的类型检查器的实现。

```
type↑ i Γ Nat  = pure VStar
type↑ i Γ Zero = pure VNat
type↑ i Γ (Succ k) =
  do type↓ i Γ k VNat
     pure VNat
type↑ i Γ (NatElim m mz ms k) =
  do type↓ i Γ m (VPi VNat (const VStar))
     let mVal = eval↓ m []
     type↓ i Γ mz (vapp mVal VZero)
     type↓ i Γ ms (VPi VNat
                       (\l -> VPi (vapp mVal l)
                                  (\_ -> vapp mVal (VSucc l))))
     type↓ i Γ k VNat
     let kVal = eval↓ k []
     pure (vapp mVal kVal)
```

图 16　为自然数扩展类型检查器

对构造子 `Zero` 和 `Succ` 的类型检查过程相当直截了当。但对消去子 `natElim` 的类型检查就有些复杂了。回想一下，消去子的类型是：

$$
\begin{aligned}
\texttt{natElim} :: \ & \forall m :: \texttt{Nat} \to * && \text{动机} \\
& \bullet \ m\ \texttt{Zero} && \text{基准情况} \\
& \to (\forall l :: \texttt{Nat} \,.\, m\, l \to m\,(\texttt{Succ}\, l)) && \text{归纳情况} \\
& \to \forall k :: \texttt{Nat} && \text{要消去的数字} \\
& \bullet \ m\, k && \text{返回类型}
\end{aligned}
$$

首先我们检查并求值动机 `m`。拿到 `m` 的值之后，我们就可以检查基准情况和归纳情况了。用于处理 `Zero` 的函数 `mz` 应具有类型 `m Zero`，而用于处理任意自然数后继的函数 `ms` 应具有类型 $\forall l :: \texttt{Nat} \,.\, m\, l \to m\,(\texttt{Succ}\ l)$。抛开手动输入这个复杂类型的细枝末节，思路本身并不复杂[32]。最后，我们对 `k` 作检查，确保它确实是一个自然数。整个 `natElim` 的返回类型就是将动机 `m` 的值应用于待消去自然数 `k` 的值得到的结果。

4.1.4. 其他函数

要完成自然数的定义，我们必须相应地扩展用于替换和引用的辅助函数。不过这些新代码相当简单直接，因为我们没有引入新的约束结构。

4.1.5. 加法

有了手头的这些东西之后，我们终于可以在我们的解释器里定义加法了：

$$\begin{aligned}
&\gg \textbf{let}\ \mathit{plus} = \texttt{natElim}\ \ (\lambda_ \to \texttt{Nat} \to \texttt{Nat}) \\
&\qquad\qquad\qquad\qquad\quad (\lambda n \to n) \\
&\qquad\qquad\qquad\qquad\quad (\lambda k\ \mathit{rec}\ n \to \texttt{Succ}\ (\mathit{rec}\ n)) \\
&\mathit{plus} :: \forall (x :: \texttt{Nat})\,(y :: \texttt{Nat})\,.\,\texttt{Nat}
\end{aligned}$$

我们通过对加法的第一个参数进行消去来定义函数 *plus*。我们为基准情况和归纳情况定义的函数的类型都是 $\texttt{Nat} \to \texttt{Nat}$，我们依此设定我们的动机为 $\lambda\_ \to \texttt{Nat} \to \texttt{Nat}$。在基准情况中，我们需要将 0 加到参数 `n` 上——也就是直接返回 `n`。在归纳情况下，传入的参数分别是前驱 `k`、递归调用 `rec`（对应于操作 *plus* `k`）和数字 `n`，而我们要把数字 `n` 和 `Succ k` 相加。我们调用 `rec` 将 `n` 与 `k` 相加，然后在结果上再套一层 `Succ`。

在定义了 *plus* 之后，我们就可以在解释器里求值简单的加法了：

$$\begin{aligned}
&\gg \mathit{plus}\ 40\ 2 \\
&42 :: \texttt{Nat}
\end{aligned}$$

4.2. 实现向量

只有自然数还是算不得亦可赛艇：我们在 Haskell 里也能轻松写出这样的数据类型。所以作为真正用到依值类型的例子，我们接下来会展示如何实现向量。

和自然数一样，我们要为向量定义三个组件：向量类型、构造子和消去子。我们之前已经给出过向量的类型，它有一个类型和一个自然数作为参数：

$$\forall \alpha :: * \,.\, \forall k :: \texttt{Nat} \,.\, \texttt{Vec}\ \alpha\ k :: *$$

向量的构造子和 Haskell 的列表 `List` 很像，唯一的区别在于向量的构造子会将长度信息记录在构造出的类型中：

$$\begin{aligned}
&\texttt{Nil}\ \ :: \forall \alpha :: * \,.\, \texttt{Vec}\ \alpha\ \texttt{Zero} \\
&\texttt{Cons} :: \forall \alpha :: * \,.\, \forall k :: \texttt{Nat} \,.\, \alpha \to \texttt{Vec}\ \alpha\ k \to \texttt{Vec}\ \alpha\ (\texttt{Succ}\ k)
\end{aligned}$$

[32]译注：原文为 “Despite the appearent complication resulting from having to hand code complex types, type checking these branches is exactly what would happen when type checking a fold over natural numbers in Haskell.” 译者依个人好恶进行了改写。

向量的消去子本质上和列表的 `foldr` 相同，但它的类型更加通用[33]（因而也更复杂）：

$$
\begin{array}{lll}
\texttt{vecElim} :: & \forall \alpha :: * & \text{向量的元素类型} \\
& \quad \bullet \ \forall m :: (\forall k :: \texttt{Nat} \,.\, \texttt{Vec}\ \alpha\ k \to *) & \text{动机} \\
& \quad \bullet \ m\ \texttt{Zero}\ (\texttt{Nil}\ \alpha) & \text{基准情况} \\
& \quad \to (\forall l :: \texttt{Nat} \,.\, \forall x :: \alpha \,.\, \forall xs :: \texttt{Vec}\ \alpha\ l. & \\
& \qquad m\ l\ xs \to m\ (\texttt{Succ}\ l)\ (\texttt{Cons}\ \alpha\ l\ x\ xs)) & \text{归纳情况} \\
& \quad \to \forall k :: \texttt{Nat} \,.\, \forall xs :: \texttt{Vec}\ \alpha\ k & \text{要消去的向量} \\
& \quad \bullet \ m\ k\ xs & \text{返回类型}
\end{array}
$$

整个消去子的类型是对向量的元素类型 α 泛化[34]的。紧随其后的参数便是动机，和自然数的消去子 `natElim` 一样，`vecElim` 的动机本质上是一个接受向量 ($\texttt{Vec}\ \alpha\ k$)、返回类型（种类 $*$）的函数，而因为向量类型有其长度 k 作为参数，所以动机还需要一个 `Nat` 型的参数 ($\forall k :: \texttt{Nat}$)[35]。接下来的两个参数对应于 `Vec` 的两个构造子。构造子 `Nil` 表示空向量，因此对应于基准情况的参数的类型就是 $m\ \texttt{Zero}\ (\texttt{Nil}\ \alpha)$。用于处理归纳情况——也就是构造子 `Cons`——的参数，则接受一个数字 l、一个类型为 α 的元素 x、一个长度为 l 的向量 xs，以及递归应用消去子的结果，其类型为 $m\ l\ xs$。它需要将这些元素结合起来，构造出符合所需类型 $m\ (\texttt{Succ}\ l)\ (\texttt{Cons}\ \alpha\ l\ x\ xs)$ 的一个词项，该类型对应于长度为 $\texttt{Succ}\ l$、内容为 $x : xs$ 的向量[36]。在向 `vecElim` 提供了这些参数之后，我们就得到了一个能消去任意长度向量的函数。

这个消去子的类型看起来相当复杂，不过如果我们把它和列表的 `foldr` 函数对比：

$$\texttt{foldr} :: \forall \alpha :: * \,.\, \forall m :: * \,.\, m \to (\alpha \to m \to m) \to [\alpha] \to m$$

我们可以看到，它们的结构是一样的。`vecElim` 的签名看似盘根错节，其实这种复杂性很大程度上只是因为动机以向量为参数，而向量又以自然数为参数。

事实上，不是所有的参数都是必须的——有些参数能从其他参数中推断出来。这种推断能大幅削减语法噪音，用消去子编写程序也就不再寸步难行。所以λ_Π看似繁文缛节，实则是设计使然——我们有意地把它设计成了一种显式的、低级的语言。

4.2.1. 抽象语法

和自然数一样，我们要扩展抽象语法。我们向 `Term↑` 中加入向量的类型、构造子和消去子：

```
data Term↑ = ...
  | Vec Term↓ Term↓
  | Nil Term↓
  | Cons Term↓ Term↓ Term↓ Term↓
  | VecElim Term↓ Term↓ Term↓ Term↓ Term↓ Term↓
```

注意构造子 `Nil` 也有一个参数，这是因为两个构造子在元素类型上都是多态的，它们都需要元素类型作为参数。对于向量和许多其他数据类型而言，构造子的归类有一定的灵活性：我们可以略去 `Nil` 和 `Cons` 的类型参数以及 `Cons` 的长度参数，并将构造子作为可检查项 `Term↓` 而不是可推断项 `Term↑`。这样我们就可以在应用这两个构造子时少传些参数，代价则是我们必须显式注明向量表达式的类型。

---

[33]译注：原文为 “but its type is a great deal more specific”，但依前文，更具体的显然是 `foldr`，故改。

[34]译注：“泛化”原作“量化” (quantified)。

[35]译注：原文为 “As was the case for natural numbers, the motive is a type (kind $*$) parameterized by a vector. As vectors are themselves parameterized by their length, the motive expects an additional argument of type `Nat`.”

[36]译注：原文为 “It combines those elements to form the required type, for the vector of length `Succ` l where x has been added to xs.”

我们还需要扩展用于表示值和中性项的数据类型：

```
data Value = ...
  | VNil Value
  | VCons Value Value Value Value
  | VVec Value Value

data Neutral = ...
  | NVecElim Value Value Value Value Value Value
```

4.2.2. 求值