类型检查器：构造演算(Calculus of Constructions)

陈鹏

2026 年 1 月 27 日

Contents

Chapter 1

类型系统概述

那么，我们为什么要构建类型系统？答案显然是因为它们非常棒且具有智力上的趣味性。但除此之外，还因为它们确实有用。

让我们从一些背景知识开始；如果你愿意，技术上可以跳过这一节直接看代码，但在深入实现之前，建立一些软性的上下文是有帮助的。

任何关于类型系统的讨论都从λ 演算开始，这是Alonzo Church 在1930 年代开发的一个形式系统，用于表达计算。它是最小的、通用的编程语言。它的语法只包含三个元素：变量、函数抽象（定义匿名函数的一种方式）和函数应用（调用函数）。例如，恒等函数（简单地返回其输入）写作$\lambda x.x$。尽管有这种简单性，任何可计算的问题都可以在λ 演算中表达和解决。

接下来，我们介绍类型系统。类型系统是一组规则，为程序的构造（如变量、表达式和函数）分配一个称为**类型**的属性。主要目的是通过防止没有意义的操作（如将数字除以字符串）来减少错误。验证程序遵守其语言类型规则的过程称为类型检查。

1.1 类型系统

任何类型系统的核心都是一组用于对程序进行逻辑推理的形式规则。这些推理称为**判断**（judgments）。判断是对代码具有某种属性的断言。你将遇到的最常见的判断类型是**类型判断**，它断言给定表达式在特定上下文中具有特定类型。我们使用"转向"符号$\vdash$ 来正式书写。

类型判断的一般形式如下：

$$\Gamma \vdash e : T$$

这个陈述读作："在上下文Γ 中，表达式e 具有类型T。"上下文Γ 本质上是从变量名到其类型的映射。例如，$x : \text{Int}, f : \text{Bool} \to \text{Int}$ 是一个上下文，其中变量x 具有类型Int，f 具有从Bool 到Int 的函数类型。当我们进入程序的更深作用域时，比如在函数体内，我们用新的变量绑定扩展上下文。这通常写作$\Gamma, x : T$，意思是"上下文Γ 扩展了一个新的绑定，说明变量x 具有类型T。"

类型判断使用**推理规则**推导。推理规则说明，如果你能证明一组判断（称为前提），那么你可以得出另一个判断（称为结论）。在处理形式类型系统时，推理规则遵循一致的结构模式，使它们更容易阅读和理解。每个规则的形式为：

$$\frac{\text{前提}}{\text{结论}}(\text{规则名称})$$

这个记号应该读作："如果线上方的所有前提都为真，那么线下方的结论也为真。"前提表示必须满足的条件，而结论表示当这些条件成立时我们可以推导出的内容。规则名称为在讨论和证明中引用规则提供了方便的标签。

如果规则没有前提，它是一个**公理**，一个自明的真理，不需要先前的证明即可成立。例如，一个建立字面量零的类型的公理规则可能如下：

$$\frac{}{\Gamma \vdash 0 : \text{Int}}(\text{T-Zero})$$

这个规则在线上方没有前提，使其成为公理。它简单地说明，在任何上下文Γ中，字面量0 具有类型Int。

这些公理通常处理简单情况，如变量查找或字面值（有时称为**基础类型**）。具有多个前提的规则（用间距或显式合取符号分隔）要求所有条件同时满足才能得出结论。

几乎所有类型系统中的一个基础规则是变量查找规则，它让我们从上下文中找到变量的类型：

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

这个规则不是公理，因为它有一个前提。它读作："如果类型绑定$x : T$ 存在于上下文Γ 中，那么我们可以得出结论，在上下文Γ 中，表达式x 具有类型T。"它正式定义了在当前环境中查找变量类型的操作。

通过定义这些推理规则的集合，我们创建了一个完整的类型系统。每个规则定义如何确定特定类型表达式的类型，如函数调用、字面值或if-then-else 块。例如，函数应用的规则要求作为前提，我们首先证明函数本身具有函数类型$T \to U$，并且其参数具有相应的输入类型T。如果我们能证明这些前提，规则允许我们得出结论，整个函数应用表达式具有输出类型U。通过反复应用这些规则，我们可以构建一个推导树，从关于变量和字面量的公理开始，最终得出关于整个程序类型的单一判断，从而证明它是良类型的。

1.2 判断

一个具有多个前提的常见推理规则是函数应用的规则，它确定函数调用的类型。在λ 演算中，这只是一个表达式$e_1\ e_2$，其中e_1 是函数，e_2 是参数。为了给这个表达式分配类型，我们必须首先确定函数及其参数的类型。这个规则，通常称为"应用"或"函数的消除"（$\to E$），写作：

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

这个规则在线上方有两个前提。它说明，为了得出结论应用$e_1\ e_2$ 具有类型T_2，我们必须首先在同一个上下文Γ 中证明两件事。首先，我们必须证明e_1 具有函数类型，写作$T_1 \to T_2$，这意味着它接受类型T_1 的输入并产生类型T_2 的输出。其次，我们必须证明参数e_2 具有正确的输入类型T_1。如果这两个前提都成立，规则允许我们推导出整个表达式$e_1\ e_2$ 产生函数的输出类型T_2。

现在，让我们看一个例子，它将三个判断链接在一起，对表达式$f\ x$ 进行类型检查。我们将在包含f 和x 绑定的上下文Γ 中工作，具体来说$\Gamma = f : \text{Int} \to \text{Bool}, x : \text{Int}$。我们的目标是证明$\Gamma \vdash f\ x : \text{Bool}$。

我们的推导从两个简单的变量查找开始，这些是我们的公理。这些将形成我们应用规则的前提：

1. **第一个判断（f 的变量查找）**：我们使用变量规则来查找f 的类型。由于$f : \text{Int} \to \text{Bool}$ 在我们的上下文Γ 中，我们可以得出结论：

$$\frac{f : \text{Int} \to \text{Bool} \in \Gamma}{\Gamma \vdash f : \text{Int} \to \text{Bool}}$$

2. **第二个判断（x 的变量查找）**：类似地，我们查找x 的类型。绑定$x : \text{Int}$ 在Γ 中，所以我们可以得出结论：

$$\frac{x : \text{Int} \in \Gamma}{\Gamma \vdash x : \text{Int}}$$

3. **第三个判断（函数应用）**：现在我们有使用函数应用规则的必要前提。我们将e_1 替换为f，e_2 替换为x，T_1 替换为Int，T_2 替换为Bool。由于我们的前两个判断成功证明了前提，我们现在可以形成最终结论：

$$\frac{\Gamma \vdash f : \text{Int} \to \text{Bool} \quad \Gamma \vdash x : \text{Int}}{\Gamma \vdash f\ x : \text{Bool}}$$

将所有这些放在一起，我们可以将完整的推导表示为嵌套推理规则的树，显示最终判断是如何从变量查找的公理构建的：

$$\frac{\dfrac{f{:}\text{Int}\to\text{Bool}\in\Gamma}{\Gamma\vdash f{:}\text{Int}\to\text{Bool}} \quad \dfrac{x{:}\text{Int}\in\Gamma}{\Gamma\vdash x{:}\text{Int}}}{\Gamma \vdash f\ x : \text{Bool}}$$

这种嵌套结构称为**推导树**或**推理树**。树中的每个节点对应于推理规则的应用，叶子是公理（从上下文中的变量查找）。树直观地展示了类型检查器如何从基本事实（上下文中变量的类型）开始，逐步应用规则，得出关于复杂表达式类型的结论。在这个例子中，树的根是判断$\Gamma \vdash f\ x : \text{Bool}$，它的两个子节点是$f$ 和x 的判断，每个都由它们各自的公理证明。这个过程推广到更大的程序，其中推导树增长以反映程序的结构和类型信息的逻辑流。

1.3 项和类型

历史上，许多编程语言在抽象的不同层之间强制执行严格的分离，这个概念称为**分层**（stratification）。在这种模型中，你有一个"项语言"和一个"类型语言"，它们在语法和概念上是不同的。项语言由实际计算值并在运行时执行的表达式组成，如$5 + 2$ 或if x then y else z。另一方面，类型语言由描述项的表达式组成，如Int 或$\text{Bool} \to \text{Bool}$。这两个世界是分开的；类型不能出现在期望项的地方，反之亦然。

这种分层可以进一步扩展。为了给类型语言本身带来秩序，通常引入第三层，称为"种类语言"（kind language）。种类可以被认为是"类型的类型"。例如，像Int 这样的具体类型具有最简单的种类$*$（通常读作"类型"）。但像List 这样的类型构造器本身不是类型；它是接受类型并产生新类型的东西。List 接受Int 产生List Int。因此，它的种类是$* \to *$。这创建了一个严格的层次结构：项由类型分类，类型由种类分类。这种清晰的分离使类型检查更简单、更可预测。

然而，现代强大类型系统设计的一个主要趋势是远离这种严格的分层，而是将项和类型语言统一到一个单一的、一致的语法框架中。在这些统一系统

中，"值"（项）和"描述"（类型）之间的界限开始模糊。语言的语法允许可以在不同"级别"或"宇宙"中解释的表达式。例如，你可能有一个宇宙Type_0，它包含像Bool 这样的简单类型。然后你会有一个更高的宇宙Type_1，其唯一成员是Type_0。这允许你编写操作类型本身的函数，这是依赖类型语言的一个关键特性。更多内容稍后介绍。

为了使这具体化，让我们考虑一个简单的、分层的算项言。我们可以分别定义其项语言（e）和相应的类型语言（τ）。项是我们可以计算的表达式，类型是我们可以分配给它们的静态标签。它们的定义可能如下：

$$\begin{array}{rl} \text{项} & e ::= n \mid e_1 + e_2 \mid \text{iszero}(e) \mid \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\ \text{类型} & \tau ::= \text{Nat} \mid \text{Bool} \end{array}$$

这里，项语言e 定义了自然数（n）、加法、检查零的函数、布尔常量和条件语句。类型语言τ 要简单得多；它只包含类型Nat 和Bool。在这个分层系统中，像$\text{Nat} + 5$ 这样的表达式将是语法错误，因为Nat 属于类型语言，不能用于像加法这样的项级操作。在更现代的统一系统中，这种严格的区分将被放宽。

1.4 类型检查和类型重构

虽然验证程序遵守其类型规则的过程称为类型检查，但一个相关且历史上重要的挑战是**类型重构**，更常称为**类型推断**。类型推断的目标是让编译器自动推导表达式的类型，而无需程序员编写显式类型注释。多年来，开发能够对越来越有表达力的语言执行完整类型推断的算法一直是一个活跃且重要的研究领域。这个承诺很有吸引力：实现静态类型系统的所有安全保证，而无需为每个变量和函数注释的冗长、手动工作。

在最近的几十年中，实现完整类型推断的焦点已经减弱。主要原因是，随着类型系统变得更强大和复杂，完整推断在计算上变得难以处理，或者在许多情况下，从根本上不可判定。现代语言通常包括像高阶多态（将多态函数作为参数传递）、GADT 和各种形式的类型级编程等特性，其中类型本身可能涉及计算。对于这些系统，一个总是能推断任何给定表达式的单一"最佳"类型的通用算法根本不存在。尝试这样做会导致极其复杂的算法，并可能产生对程序员来说巨大且难以理解的推断类型。

因此，许多现代静态类型语言已经收敛到一个实用且优雅的中间立场：**双向类型检查**。不是有一个总是试图推断类型的单一模式，双向检查器在两个不同的模式中操作："检查"模式和"综合"模式。

1. **检查模式**：在这种模式中，算法验证表达式e 符合已知的、期望的类型τ。信息从上下文"向下"流入表达式。我们问问题："我们能证明e 具有类型τ 吗？"

2. **综合模式**：在这种模式中，算法计算或"综合"表达式e 的类型，而无需任何先前的期望。信息从表达式的组件"向上"流动。这里，我们问："e 的类型是什么？"

这种二元性提供了一个强大的框架。语言设计者可以指定哪些语法构造需要注释，哪些不需要。例如，顶级函数的参数可能需要显式注释（将检查器置于函数体的检查模式），但该函数体内局部变量的类型可以推断（综合）。这种方法通过要求程序员仅在可能产生歧义的关键边界提供注释，巧妙地避开了完整推断的困难。它提供了一个"两全其美"的场景：局部推断的便利性与复杂、多态或代码

模糊部分的显式注释的清晰性和强大性，代表了一个理论上的最佳点，平衡了表达性、可用性和可实现性。更多内容稍后介绍。

1.5 前沿

自1970 年代末以来，研究人员注意到计算、形式逻辑和范畴论之间惊人的结构相似性。这种观察，有时称为"计算三位一体"（computational trinitarianism），表明这三个学科从不同角度研究相同的基础数学结构。这些联系是真实且数学上严格的，尽管它们对日常编程的实际影响仍然是一个持续工作的问题，而不是既定事实。

这里的核心洞察是**Curry-Howard 对应**，它建立了类型系统和逻辑证明系统之间的形式对偶性。在这种对应下，类型可以读作逻辑命题，良类型程序构成该命题的构造性证明。这不仅仅是类比：相同的形式结构出现在两个领域中，在一个环境中证明的结果直接转移到另一个环境。

对应自然地扩展到几个方向：

- 逻辑中的**命题**对应于编程中的**类型**，对应于范畴论中的**对象**。
- **证明**对应于该类型的**程序**（或项），对应于对象之间的**态射**。
- **证明简化**对应于**程序求值**，对应于态射的组合。

这在实际中给我们带来了什么？考虑一个检索列表第一个元素的函数。像‘head(List<T>) -> T‘ 这样的简单签名做出了实现无法兑现的承诺：它声称为任何列表产生类型‘T‘ 的值，但空列表没有第一个元素。这个类型是一个错误的命题。

一个更诚实的签名是‘safeHead(List<T>) -> Maybe<T>‘。‘Maybe‘ 类型强制调用者处理缺失的可能性。这是一个适度的例子，但它说明了关键点：类型系统强制执行一个契约，编译器静态检查该契约。类型表达的命题对于任何类型检查的程序实际上都是真的。

依赖类型系统进一步推动了这一点。在像Agda、Coq 或Lean 这样的语言中，你可以定义一个类型‘SortedList<T>‘，其值是列表以及它们已排序的证明。具有签名‘merge(SortedList<T>, SortedList<T>) -> SortedList<T>‘ 的合并函数必须不仅构造合并的列表，还要构造结果已排序的证明。编译器将拒绝任何未能提供此证明的实现。

这确实强大，但它带来的成本很容易被低估。编写证明需要大多数程序员不具备的专业知识。证明负担可能超过实现工作一个数量级。证明助手在自动化方面取得了重大进展，但我们距离深度正确性属性自动产生的世界还很远。支持这些特性的语言仍然是小众的，主要用于关键系统的形式验证和数学研究。

这里有一个诱人的愿景：如果类型是命题，程序是证明，也许我们可以将我们想要的指定为类型，并让计算机自动综合程序。这是从规范进行程序综合的梦想。现实更加温和。当前的综合工具对于小的、高度受限的问题效果很好。从丰富的规范生成正确的实现仍然是一个开放的研究问题，实用工具只处理有限的领域。

随着机器学习系统越来越多地生成代码，类型系统提供了一条通向信任的路径。类型检查器是一个验证器：它只接受满足其规则的程序，无论代码是由人还是模型编写的。更丰富的类型系统提高了可接受性的标准，在部署前捕获更多错误。这是投资于有表达力的类型的合理工程论证，尽管值得注意的是，我们在真

实系统中关心的大多数属性，如性能、针对侧信道攻击的安全性或分布式状态的正确处理，仍然超出了当前类型系统可以表达的范围。

逻辑、计算和范畴论之间的理论联系是美丽的，几十年来一直指导着编程语言的设计。它们表明这些结构中存在某种深刻和统一的东西。但将这些理论转化为工作程序员可以使用的实用工具仍然是持续的工作。原则上可能的东西与在实践中实用的东西之间的差距是真实的，缩小它需要继续研究证明自动化、语言设计和工具。

这些想法值得研究，不是因为它们会立即改变你编写软件的方式，而是因为它们揭示了代码下的数学骨架。理解类型为什么以它们的方式工作，以及它们能够表达什么，使你成为更好的程序和语言设计者。这种理解本身就是回报。

Chapter 2

λ立方

λ立方（Lambda Cube）由Henk Barendregt 在1990 年代初期引入，提供了一个统一的框架来分类和理解编程语言中使用的不同类型系统。它通过考虑三个独立的抽象维度，系统化地展示了八个重要类型系统之间的关系，这些系统从简单类型λ 演算到构造演算（Calculus of Constructions）。

2.1 λ立方的结构

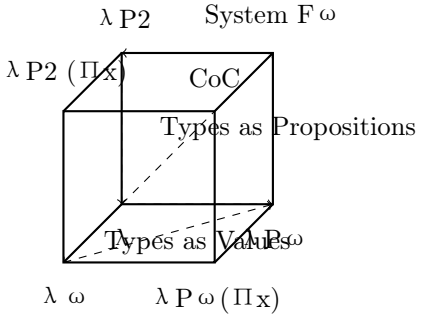
λ立方是一个三维立方体结构，其中每个维度对应于项和类型之间不同类型的依赖关系。这里的"依赖"指的是项或类型能够绑定或依赖于项或类型的能力。

立方体的三个正交轴分别是：

- **→-轴（类型依赖于项）**：启用依赖类型（dependent types），其中类型的结构可以依赖于项的值。例如，‘Vector n‘ 类型，其中长度‘n‘ 是项级值，允许我们精确指定数据结构的属性。
- **↑-轴（项依赖于类型）**：启用多态性（polymorphism），其中项可以抽象类型参数并依赖于类型参数。这允许像$\forall\alpha.\alpha \to \alpha$ 这样的函数，在所有类型上统一工作。
- **↗-轴（类型依赖于类型）**：启用类型运算符（type operators），其中类型可以抽象其他类型并依赖于其他类型。这允许像‘Maybe : * → *‘ 这样的类型构造函数，它们接受类型作为参数并产生新类型。

2.2 八个类型系统

通过组合这三个依赖维度的不同方式，我们得到立方体的八个顶点，每个顶点代表一个不同的类型化系统。



2.2.1 简单类型λ 演算（λ →）

这是立方体的基础系统，仅支持项抽象。函数可以抽象项（$\lambda x : \tau.e$），但类型保持固定和简单。这是最基础的类型系统，对应于简单类型λ 演算。

语法定义：

$$\begin{aligned} \text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \\ \text{项} \quad & e ::= x \mid \lambda x : \tau . e \mid e_1\ e_2 \end{aligned}$$

2.2.2 System F（$\lambda 2$）

通过类型抽象添加多态性，使项能够抽象类型（$\Lambda\alpha.e$）。这引入了参数多态性，其中像恒等函数这样的函数可以在所有类型上统一工作。

语法定义：

$$\begin{aligned} \text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha . \tau \\ \text{项} \quad & e ::= x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid \Lambda \alpha . e \mid e[\tau] \end{aligned}$$

System F 是许多现代函数式编程语言（如Haskell 和ML）中泛型的基础。

2.2.3 System Fω（$\lambda\omega$）

通过添加类型运算符引入高阶类型，允许类型抽象类型（$\lambda\alpha : \kappa.\tau$）。这使我们能够抽象像‘Maybe‘ 和‘List‘ 这样的类型构造函数。

语法定义：

$$\begin{aligned} \text{种类} \quad & \kappa ::= \star \mid \kappa_1 \to \kappa_2 \\ \text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \tau_1\ \tau_2 \mid \lambda \alpha : \kappa . \tau \\ \text{项} \quad & e ::= x \mid \lambda x : \tau . e \mid e_1\ e_2 \end{aligned}$$

2.2.4 Lambda P（λP）

添加依赖类型，其中类型可以依赖于项。这允许像‘Vector n‘ 这样的类型，其中长度‘n‘ 是项级值，能够精确指定数据结构的属性。

语法定义：

$$\begin{aligned} \text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \Pi x : \tau_1 . \tau_2 \\ \text{项} \quad & e ::= x \mid \lambda x : \tau . e \mid e_1\ e_2 \end{aligned}$$

其中$\Pi x : \tau_1.\tau_2$ 是依赖函数类型，当τ_2 不依赖于x 时，它退化为普通函数类型$\tau_1 \to \tau_2$。

2.2.5 System Fω（$\lambda 2\omega$）

将多态性与高阶类型相结合，使项能够抽象类型，类型也能够抽象类型。这是许多现代函数式编程语言的理论基础，为现代函数式编程提供了所需的表达能力。

语法定义：

$$\begin{aligned} \text{种类} \quad & \kappa ::= \star \mid \kappa_1 \to \kappa_2 \\ \text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha : \kappa . \tau \mid \tau_1\ \tau_2 \mid \lambda \alpha : \kappa . \tau \\ \text{项} \quad & e ::= x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid \Lambda \alpha : \kappa . e \mid e[\tau] \end{aligned}$$

2.2.6 System Fω-P（λωP）

将高阶类型与依赖类型相结合，允许依赖于项级值的类型级计算。这个系统结合了类型运算符和依赖类型的能力。

2.2.7 System FP（λ2P）

将多态性与依赖类型相结合，使函数在类型和值上都是参数的，同时允许类型依赖于这些值。这个系统结合了System F 的多态性和Lambda P 的依赖类型。

2.2.8 构造演算（λ2ωP）

最具表达力的系统，结合了所有三种抽象形式：多态性、类型运算符和依赖类型。该系统是像Coq 和Agda 这样的证明助手的基础，并使类型能够表达任意逻辑命题。

构造演算的语法定义：

$$
\begin{aligned}
\text{种类} \quad & \kappa ::= \star \mid \kappa_1 \to \kappa_2 \\
\text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha : \kappa.\tau \mid \Pi x : \tau_1.\tau_2 \mid \tau_1\ \tau_2 \mid \lambda \alpha : \kappa.\tau \\
\text{项} \quad & e ::= x \mid \lambda x : \tau.e \mid e_1\ e_2 \mid \Lambda \alpha : \kappa.e \mid e[\tau]
\end{aligned}
$$

2.3 λ立方的层次结构

λ立方表明这八个系统形成了一个自然的层次结构，每个系统都是其下系统的保守扩展。这种包含关系可以通过立方体的几何结构直观地理解：

- 立方体的底部（$\lambda \to$）是最简单的系统
- 沿着三个轴移动，我们添加新的抽象能力
- 立方体的顶部（$\lambda 2\omega P$，即构造演算）是最具表达力的系统
- 任何两个系统之间的路径表示从一个系统到另一个系统的扩展

这种层次结构的一个重要性质是**保守性**：如果一个项在较简单的系统中是可类型化的，那么它在任何包含该系统的更复杂系统中也是可类型化的，并且类型保持不变。

2.4 命题即类型解释

λ立方中的系统通过**命题即类型**（propositions-as-types）解释与逻辑系统建立联系。在这种对应下：

- 类型对应于逻辑命题
- 项（程序）对应于证明
- 类型检查对应于证明验证
- 程序执行对应于证明简化

这种对应关系，也称为Curry-Howard 对应，为类型系统提供了深刻的逻辑解释：

- $\lambda \to$ 对应于直觉主义命题逻辑
- $\lambda 2$（System F）对应于二阶命题逻辑

- λP 对应于一阶谓词逻辑
- $\lambda 2\omega P$（构造演算）对应于高阶谓词逻辑

2.5 关键性质

λ立方中的所有系统都满足一些重要的元理论性质：

2.5.1 主题归约（Subject Reduction）

如果$\Gamma \vdash M : \sigma$ 且$M \to_\beta N$，那么$\Gamma \vdash N : \sigma$。这意味着β-归约保持类型：如果一个项具有某个类型，那么它的归约结果也具有相同的类型。

2.5.2 强归一化（Strong Normalization）

所有良类型的项都是强归一化的，即不存在无限归约序列。这意味着每个良类型项最终都会归约到一个范式。

2.5.3 类型唯一性（Unicity of Types）

在某些系统中，如果$\Gamma \vdash M : \sigma$ 且$\Gamma \vdash M : \tau$，那么$\sigma = \tau$。这意味着每个项最多只有一个类型（在给定的上下文中）。

2.6 广义类型系统

Berardi（1988）和Barendregt 独立地将λ立方的方法推广到更一般的框架，称为**纯类型系统**（Pure Type Systems，PTS）。纯类型系统通过指定一组**排序**（sorts）和**公理**（axioms）来定义，提供了比λ立方更灵活的类型系统构造方法。

在纯类型系统中，类型系统的特征由三个集合决定：

- $\mathcal{S}$：排序集合（如$\{\star, \Box\}$）
- $\mathcal{A}$：公理集合（如$\star : \Box$）
- $\mathcal{R}$：规则集合（指定哪些依赖关系是允许的）

λ立方中的每个系统都可以表示为特定的纯类型系统，这种统一视角揭示了不同类型系统之间的深层结构相似性。

2.7 实际应用

λ立方中的不同系统在现代编程语言和证明助手中都有实际应用：

- $\lambda \to$：简单类型系统，用于许多教学语言和基础类型检查器
- **System F**（$\lambda 2$）：Haskell、ML 等语言中泛型的基础
- **System Fω**（$\lambda 2\omega$）：支持高阶类型构造，用于更高级的类型级编程
- **Lambda P**（λP）：依赖类型的基础，用于Agda、Idris 等语言
- **构造演算**（$\lambda 2\omega P$）：Coq 证明助手的基础，支持完整的依赖类型和类型级计算

理解λ立方不仅有助于理解这些系统之间的关系，还为设计新的类型系统和理解现有系统的表达能力提供了理论基础。

2.8 总结

λ立方提供了一个优雅的框架来理解类型系统的发展。通过三个简单的维度——类型依赖于项、项依赖于类型、类型依赖于类型——我们可以系统地组织从简单类型系统到完整构造演算的整个类型系统谱系。这种统一视角不仅具有理论美感，还为实际的语言设计和实现提供了指导。

Chapter 3

Algorithm W

Algorithm W 代表了函数式编程语言中类型推断问题最早且优雅（就其时代而言）的解决方案之一。由Robin Milner 在1978 年开发，它为Hindley-Milner 类型系统中推断最一般类型提供了可靠且完整的方法。本节探索我们的Rust 实现，研究数学基础如何转化为可以处理λ 抽象、函数应用、let-多态和复杂统一场景的实用代码。

Algorithm W 的核心洞察在于其通过约束生成和统一进行类型推断的系统化方法。该算法不是试图通过局部分析来确定类型，而是通过生成类型变量、收集约束，然后通过统一解决这些约束来构建全局图景。这种方法确保我们总是找到可能的最一般类型，这是支持函数式语言中多态性的关键属性。

你经常会看到这个算法（或者类型系统，令人困惑的是）被许多名称引用：

- **Hindley-Milner**
- **Hindley-Damas-Milner**
- **Damas-Milner**
- **HM**
- **Algorithm W**

3.1 λ 演算

λ **演算**是计算的最纯粹形式。说这个优雅、微小的系统是几乎所有函数式编程语言的理论基石并不夸张，它的影响在整个计算机科学领域都能感受到。由Alonzo Church 在1930 年代开发，作为研究数学基础的工具，后来被理解为计算的通用模型。在其核心，λ 演算令人震惊地最小化；它的语法只定义了三种表达式，或"项"。

形式上，我们可以将纯λ 演算的语法定义如下：

$$e ::= x \mid \lambda x.e \mid e_1\ e_2$$

让我们分解一下：

1. **变量**（x）：作为值占位符的名称。
2. **抽象**（$\lambda x.e$）：这是一个匿名函数定义。λx 是函数的参数，表达式e 是它的主体。λ 读作"lambda"。
3. **应用**（$e_1\ e_2$）：这是调用函数的行为。表达式e_1 是函数，e_2 是传递给它的参数。

这个形式定义几乎直接映射到像Rust 这样的语言中的数据结构。我们可以使用一个‘enum‘ 来表示这三个核心项：

```
#![allow(unused)]
fn main() {
pub enum Expr {
    Var(String),
```

```
    Abs(String, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
}
}
```

这里，‘Var(name)‘ 对应于x，‘Abs(param, body)‘ 对应于$\lambda x.e$，‘App(function, argument)‘ 对应于$e_1\ e_2$。‘Box‘ 是Rust 的一个细节，允许我们拥有已知大小的递归类型。

λ 演算的力量来自几个基本概念。第一个是**变量绑定**。在像$\lambda x.x+1$ 这样的抽象中，变量x 被称为在lambda 的主体中**绑定**。任何没有被封闭的lambda 绑定的变量是**自由变量**。这种区分对于理解作用域至关重要。这直接导致了α **等价**的概念，它说明绑定变量的名称是无关紧要的。函数$\lambda x.x$ 在语义上与$\lambda y.y$ 相同；两者都是恒等函数。实现必须能够识别这种等价性，以正确处理变量命名。

第二个核心概念是β **归约**，这是λ 演算的计算引擎。它正式定义了函数应用如何工作。当抽象应用于参数时，我们通过在函数主体中用参数替换绑定变量的每个自由出现来归约表达式。例如，将函数$\lambda x.x+1$ 应用于参数5 写作$(\lambda x.x+1)\ 5$。β 归约规则告诉我们在主体$x+1$ 中用5 替换x，产生结果$5+1$。这种替换过程是这个系统中计算的基本机制。实现通常避免直接字符串替换，因为其复杂性和名称冲突的风险，而是使用像de Bruijn 索引这样的技术，用表示它们到绑定器的词法距离的数字替换变量名。

虽然纯λ 演算是图灵完备的，但它也以难以直接用于实际编程而闻名。例如，表示数字3 需要一个复杂的表达式，如$\lambda f.\lambda x.f(f(fx))$。为了使编程更方便，我们几乎总是用额外的表达式类型扩展核心演算。这些可以包括用于局部变量的‘let‘绑定、用于像数字和字符串这样的具体值的字面量，以及像元组这样的数据结构。

```
#![allow(unused)]
fn main() {
#[derive(Debug, Clone, PartialEq)]
pub enum Lit {
    Int(i64),
    Bool(bool),
}
#[derive(Debug, Clone, PartialEq)]
pub enum Type {
    Var(String),
    Arrow(Box<Type>, Box<Type>),
    Int,
    Bool,
    Tuple(Vec<Type>),
}
#[derive(Debug, Clone, PartialEq)]
pub struct Scheme {
    pub vars: Vec<String>, // 量化的类型变量
    pub ty: Type,          // 被量化的类型
}
impl std::fmt::Display for Expr {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Expr::Var(name) => write!(f, "{}", name),
            Expr::Lit(Lit::Int(n)) => write!(f, "{}", n),
            Expr::Lit(Lit::Bool(b)) => write!(f, "{}", b),
            Expr::Abs(param, body) => write!(f, "λ{}.{}", param, body),
            Expr::App(func, arg) => match (func.as_ref(), arg.as_ref()) {
                (Expr::Abs(_, _), _) => write!(f, "({}) {}", func, arg),
                (_, Expr::App(_, _)) => write!(f, "{} ({})", func, arg),
                (_, Expr::Abs(_, _)) => write!(f, "{} ({})", func, arg),
                _ => write!(f, "{} {}", func, arg),
            },
            Expr::Let(var, value, body) => {
                write!(f, "let {} = {} in {}", var, value, body)
            }
            Expr::Tuple(exprs) => {
                write!(f, "(")?;
                for (i, expr) in exprs.iter().enumerate() {
                    if i > 0 {
                        write!(f, ", ")?;
                    }
                    write!(f, "{}", expr)?;
                }
                write!(f, ")")
            }
        }
    }
}
impl std::fmt::Display for Type {
```

```
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Type::Var(name) => write!(f, "{}", name),
            Type::Int => write!(f, "Int"),
            Type::Bool => write!(f, "Bool"),
            Type::Arrow(t1, t2) => {
                // 如果左侧是箭头，添加括号以避免歧义
                match t1.as_ref() {
                    Type::Arrow(_, _) => write!(f, "({}) → {}", t1, t2),
                    _ => write!(f, "{} → {}", t1, t2),
                }
            }
            Type::Tuple(types) => {
                write!(f, "(")?;
                for (i, ty) in types.iter().enumerate() {
                    if i > 0 {
                        write!(f, ", ")?;
                    }
                    write!(f, "{}", ty)?;
                }
                write!(f, ")")
            }
        }
    }
}
impl std::fmt::Display for Scheme {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        if self.vars.is_empty() {
            write!(f, "{}", self.ty)
        } else {
            write!(f, "forall {}. {}", self.vars.join(" "), self.ty)
        }
    }
}
#[derive(Debug, Clone, PartialEq)]
pub enum Expr {
    Var(String),
    App(Box<Expr>, Box<Expr>),
    Abs(String, Box<Expr>),
    Let(String, Box<Expr>, Box<Expr>),
    Lit(Lit),
    Tuple(Vec<Expr>),
}
}
```

这提出了一个关键的设计问题：这些新构造应该作为具有自己语义规则的原始构造"内置"，还是应该用纯演算来定义？这代表了语言设计中的一个基本权衡。例如，‘let‘ 表达式let $x = e_1$ in e_2 可以被视为**语法糖**，直接脱糖为纯λ 演算表达式：$(\lambda x.e_2)\ e_1$。这种方法保持核心语言最小且优雅。另一种方法是使‘let‘ 成为原始构造。这意味着类型检查器和求值器必须有特定的逻辑来处理它。这种"内置"方法通常导致更好的性能和更精确的错误消息，因为编译器对构造的意图有更具体的了解。然而，它增加了核心系统的复杂性。许多语言取得平衡：像数字字面量这样的基础和性能关键特性通常是内置的，而像‘let‘ 绑定这样的高级模式可能被视为语法糖，提供方便的语法，最终转换回lambda、应用和变量的基础概念。

3.2 类型规则

在深入实现细节之前，让我们建立管理Hindley-Milner 类型系统的形式类型规则。我们将引入捕捉类型推断本质的数学符号，但不要担心，一旦你深入细节，每个符号都有精确且直观的含义。

- τ **(Tau)** - 表示单态类型，如Int、Bool 或Int $\to$ Bool。这些是具体的、完全确定的类型。
- α, β, γ **(希腊字母)** - 在推断期间代表未知类型的类型变量。将它们视为在类型级别被解决的未知数。
- Γ **(Gamma)** - 类型环境，它将变量映射到它们的类型。它就像一个字典，记住我们对每个变量类型的了解。
- $\vdash$ **(转向)** - "蕴涵"或"证明"符号。当我们写$\Gamma \vdash e : \tau$ 时，我们说"在环境Γ 中，表达式e 具有类型τ。"
- σ **(Sigma)** - 表示多态类型方案，如$\forall\alpha.\alpha \to \alpha$。这些可以用不同的具体类型实

例化。

- $\forall\alpha$ **(Forall Alpha)** - 对类型变量的全称量化。它意味着"对于任何类型α。"这是我们表达多态性的方式。
- $[\tau/\alpha]\sigma$ - 类型替换，在方案σ 中用类型τ 替换类型变量α 的所有出现。这是我们实例化多态类型的方式。
- S **(替换)** - 从类型变量到类型的映射，表示通过统一找到的解。
- $\mathbf{gen}(\Gamma, \tau)$ - 泛化，通过量化环境中不存在的类型变量，将单态类型转换为多态类型。
- $\mathbf{inst}(\sigma)$ - 实例化，通过用新的类型变量替换量化变量，从多态类型创建新的单态类型。
- $\mathbf{ftv}(\tau)$ - 自由类型变量，出现在类型τ 中的未绑定类型变量的集合。
- $\emptyset$ **(空集)** - 空替换，表示对类型没有更改。
- $[\alpha \mapsto \tau]$ - 将类型变量α 映射到类型τ 的替换。
- $S_1 \circ S_2$ - 替换的组合，先应用S_2，然后应用S_1。
- $\notin$ **(不在)** - 集合成员否定，用于出现检查以防止无限类型。

3.2.1 类型规则

变量规则从环境中查找类型：

$$\frac{x : \sigma \in \Gamma \quad \tau = \text{inst}(\sigma)}{\Gamma \vdash x : \tau}(\text{T-Var})$$

λ 抽象引入新的变量绑定：

$$\frac{\Gamma, x : \alpha \vdash e : \tau \quad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : \alpha \to \tau}(\text{T-Lam})$$

函数应用通过统一组合类型：

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \alpha \text{ fresh} \quad S = \text{unify}(\tau_1, \tau_2 \to \alpha)}{\Gamma \vdash e_1\ e_2 : S(\alpha)}(\text{T-App})$$

Let-多态允许泛化：

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \sigma = \text{gen}(\Gamma, \tau_1) \quad \Gamma, x : \sigma \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}(\text{T-Let})$$

字面量具有相应的基础类型：

$$\frac{}{\Gamma \vdash n : \text{Int}}(\text{T-LitInt})$$

$$\frac{}{\Gamma \vdash b : \text{Bool}}(\text{T-LitBool})$$

这些规则捕捉了Hindley-Milner 类型系统的本质。如果这看起来需要吸收很多内容，不要担心！只需跳到Rust 代码并尝试将公式中的符号追踪到实际行，很

快你就会看到它们如何与代码对应。存在一一映射，虽然符号可能看起来很复杂，但它们实际上是代码中非常直接的表达式，主要只是操作、查找和组合哈希表。

3.3 核心实现

Algorithm W 的核心实现在于‘TypeInference‘ 结构，它维护在整个程序中可靠类型推断所需的状态。

类型变量抽象表示在推断期间将被解决的未知类型。项变量表示出现在表达式中的程序变量。类型环境将项变量映射到它们的类型，而替换将类型变量映射到具体类型。

```
#![allow(unused)]
fn main() {
pub type TyVar = String;
pub type TmVar = String;
pub type Env = BTreeMap<TmVar, Scheme>;  // 现在存储方案，而不是类型
pub type Subst = HashMap<TyVar, Type>;
}
```

这些别名封装了Algorithm W 中的基本数据流。像‘t0‘、‘t1‘ 和‘t2‘ 这样的类型变量作为占位符，随着推断的进行与具体类型统一。项变量表示源程序中的实际标识符。环境现在跟踪多态类型方案而不仅仅是类型，支持适当的let-多态，而替换记录通过统一发现的解。

选择‘String‘ 作为类型和项变量反映了我们实现的简单性。在完整实现中，系统通常使用更复杂的表示，如用于类型变量的de Bruijn 索引或用于性能的内联字符串，但字符串为理解基本算法提供了清晰性。

3.3.0.1 替换和统一

类型替换表示Algorithm W 的核心计算机制。替换将类型变量映射到具体类型，有效地"解决"我们类型推断难题的一部分。

替换的应用必须正确处理类型的递归结构，确保替换通过像箭头和元组这样的复合类型传播。

统一是类型推断的核心，解决类型之间的约束。统一算法产生使两个类型等价的替换：

- **自反性** - 相同类型平凡统一：$\frac{}{\text{unify}(\tau,\tau)=\emptyset}$(U-Refl)
- **变量统一与出现检查**：$\frac{\alpha \notin \text{ftv}(\tau)}{\text{unify}(\alpha,\tau)=[\alpha \mapsto \tau]}$(U-VarL)
- **箭头类型统一**分解为定义域和值域：$\frac{S_1=\text{unify}(\tau_1,\tau_3) \quad S_2=\text{unify}(S_1(\tau_2),S_1(\tau_4))}{\text{unify}(\tau_1 \to \tau_2,\tau_3 \to \tau_4)=S_2 \circ S_1}$(U-Arrow)
- **元组统一**需要逐分量统一：$\frac{S_1=\text{unify}(\tau_1,\tau_3) \quad S_2=\text{unify}(S_1(\tau_2),S_1(\tau_4))}{\text{unify}((\tau_1,\tau_2),(\tau_3,\tau_4))=S_2 \circ S_1}$(U-Tuple)
- **基础类型统一**仅对相同类型成功：$\frac{}{\text{unify}(\text{Int},\text{Int})=\emptyset}$(U-Int) 和$\frac{}{\text{unify}(\text{Bool},\text{Bool})=\emptyset}$(U-Bool)

这些统一规则确保类型约束被系统地解决，同时通过出现检查保持可靠性。

出现检查通过确保类型变量不会出现在它被统一到的类型中来防止无限类型。这个检查对于可靠性至关重要。没有它，我们可能会生成像$t_0 = t_0 \to \text{Int}$ 这样的无限类型，这会破坏我们类型系统的可判定性。

对于像箭头这样的复合类型，统一变得递归。我们必须统一相应的子组件，然后组合产生的替换。这个过程确保复杂类型在允许类型变量的灵活实例化的同时保持它们的结构关系。

3.3.0.2 主要推断算法

核心‘infer‘ 方法实现Algorithm W 本身，分析表达式以确定它们的类型，同时累积必要的替换。我们的实现使用模块化方法，其中每个语法构造都有自己的辅助方法，实现相应的类型规则。

每个辅助方法直接对应于形式类型规则，使理论和实现之间的关系明确。

变量查找 变量查找需要多态类型的实例化。当我们在环境中找到变量时，它可能具有像$\forall\alpha.\alpha \to \alpha$ 这样的多态类型方案。我们通过用新的类型变量替换量化变量来创建新的单态实例。

λ 抽象 λ 抽象引入新的变量绑定。我们为参数分配新的类型变量，扩展环境，并推断主体的类型。在主体推断期间发现的任何约束通过替换传播回来。

函数应用 应用驱动约束生成。我们推断函数和参数的类型，然后将函数类型与从参数类型和新结果类型变量构造的箭头类型统一。

Let-多态 Let 表达式通过泛化实现多态性。在推断绑定表达式的类型后，我们通过量化不受环境约束的类型变量来泛化它。这允许在let 主体中多态使用。

字面量类型 字面量具有已知类型，不需要约束生成。

3.3.0.3 泛化和实例化

泛化和实例化机制处理let-多态，允许在let 表达式中绑定的变量与多个不同类型一起使用。

理解泛化 泛化将具体类型转换为多态类型方案。考虑这个简单例子：

```
let id = \x -> x in (id 42, id true)
```

当我们推断`\x -> x` 的类型时，我们得到类似$t_0 \to t_0$ 的东西，其中t_0 是类型变量。由于t_0 没有出现在环境中的其他地方，我们可以将其泛化为$\forall t_0.t_0 \to t_0$，使`id` 多态。

这允许‘id‘ 在同一表达式中与‘42‘（类型‘Int‘）和‘true‘（类型‘Bool‘）一起使用。没有泛化，第一次使用会将t_0 固定为‘Int‘，使第二次使用失败。

泛化通过检查哪些类型变量不出现在当前环境中自由出现来识别可以多态化的类型变量。如果类型变量不受作用域中其他任何东西的约束，量化它是安全的。

理解实例化 实例化从多态类型创建新的单态版本。当我们使用像恒等函数这样的多态函数时，我们需要为每次使用创建其类型的新副本。

考虑这个表达式：

```
let id = \x -> x in id id
```

这里我们将多态恒等函数应用于自身。第一个‘id‘ 被实例化为$(\alpha \to \alpha) \to (\alpha \to \alpha)$，而第二个‘id‘ 被实例化为$\alpha \to \alpha$。这些不同的实例化允许应用成功进行类型检查。

实例化用新的类型变量替换量化类型变量。这确保多态函数的每次使用都获得自己独立的类型约束，防止不同调用站点之间的干扰。

3.4 示例

最简单的案例涉及字面量和变量，其中类型推断是直接的。整数字面量具有类型‘Int‘，布尔字面量具有类型‘Bool‘，变量接收在类型环境中分配给它们的类型。

让我们检查我们的实现如何使用验证我们Algorithm W 实现的测试套件处理这些基本案例。

```
基本字面量
42
true
false

变量和恒等
\x -> x
\f -> f
\x -> \y -> x
\x -> \y -> y

应用
(\x -> x) 42
(\x -> x) true
(\f -> \x -> f x) (\y -> y) 42

Let 表达式
let x = 42 in x
let f = \x -> x in f
let id = \x -> x in id 42
let id = \x -> x in id true
let f = \x -> x in let g = \y -> y in f (g 42)

元组
(42, true)
(true, false, 42)
(\x -> (x, x)) 42
let pair = \x -> \y -> (x, y) in pair 42 true

高阶函数
\f -> \x -> f x
\f -> \x -> f (f x)
\f -> \g -> \x -> f (g x)
let twice = \f -> \x -> f (f x) in twice
let compose = \f -> \g -> \x -> f (g x) in compose

复杂示例
let K = \x -> \y -> x in K
let S = \f -> \g -> \x -> f x (g x) in S
let Y = \f -> (\x -> f (x x)) (\x -> f (x x)) in Y

多态示例（为我们的基本实现简化）
let id = \x -> x in (id, id)
let const = \x -> \y -> x in const
let flip = \f -> \x -> \y -> f y x in flip

错误案例（应该失败）
\x -> x x
(\x -> x x) (\x -> x x)
```

这些示例展示了类型推断的基础。字面量‘42‘ 立即接收类型‘Int‘，不需要任何复杂的推理。像‘true‘ 和‘false‘ 这样的布尔值类似地接收类型‘Bool‘。类型系统对这些基础案例的处理形成了更复杂推断场景的构建块。

3.4.1 函数类型和应用

函数类型表示λ 演算和函数式编程的核心。当我们遇到λ 抽象时，Algorithm W 为参数分配新的类型变量并推断主体的类型。产生的函数类型通过箭头将参数类型与返回类型连接起来。

函数应用驱动使Algorithm W 强大的约束生成。当将函数应用于参数时，算法生成约束，即函数的类型必须是从参数类型到某个结果类型的箭头。然后统一解决这些约束。

恒等函数`\x -> x`提供了多态类型推断的最简单例子。Algorithm W 为参数`x`分配新的类型变量，然后发现主体只是`x` 本身。产生的类型$t_0 \to t_0$ 捕捉了恒等函数的本质：它接受任何类型并返回相同的类型。

更复杂的函数应用展示了约束如何通过系统传播。当我们将恒等函数应用于整数‘42‘ 时，统一过程发现类型变量t_0 必须是‘Int‘，为整个表达式产生最终类型‘Int‘。

3.4.2 Let-多态

Let 表达式引入了Hindley-Milner 类型系统最强大的特性之一：let-多态。这种机制允许在let 表达式中绑定的变量在不同上下文中与不同类型一起使用，实现灵活的代码重用而不牺牲类型安全。

经典例子涉及在let 表达式中绑定恒等函数，然后与不同类型一起使用它。Algorithm W 通过抽象当前环境中不自由出现的类型变量来泛化绑定表达式的类型。这种泛化允许相同的绑定在每个使用站点用新的类型变量实例化。

考虑表达式`let f = \x -> x in (f 42, f true)`。首先，Algorithm W 推断恒等函数具有类型$t_0 \to t_0$。在泛化期间，由于t_0 不出现在环境中，系统将其视为可以在每次使用时不同实例化的多态类型。

当算法遇到第一次应用‘f 42‘ 时，它创建多态类型的新实例，比如$t_1 \to t_1$，并将其与$\text{Int} \to t_2$（其中t_2 是期望的结果类型）统一。这个统一成功，$t_1 = \text{Int}$ 和$t_2 = \text{Int}$。

对于第二次应用‘f true‘，Algorithm W 创建另一个新实例$t_3 \to t_3$ 并将其与$\text{Bool} \to t_4$ 统一。这成功，$t_3 = \text{Bool}$ 和$t_4 = \text{Bool}$。最终结果是元组类型$(\text{Int}, \text{Bool})$，展示了同一函数如何多态使用。

3.4.3 元组

元组提供了一种组合多个可能不同类型值的方法。我们的Algorithm W 实现通过推断每个分量的类型并将它们组合成元组类型来处理元组。

表达式‘(42, true)‘ 展示了基本元组构造。Algorithm W 推断第一个分量具有类型‘Int‘，第二个具有类型‘Bool‘，产生元组类型‘(Int, Bool)‘。这自然地扩展到嵌套元组，如‘((1, 2), (true, false))‘，它接收类型‘((Int, Int), (Bool, Bool))‘。

元组与多态性有趣地交互。表达式`let f = \x -> (x, x) in f 42` 展示了多态函数如何构造元组。函数`f` 具有类型$t_0 \to (t_0, t_0)$，创建一个元组，其中两个分量具有与输入相同的类型。当应用于`42` 时，这产生类型`(Int, Int)`。

3.4.4 类型错误

Algorithm W 的基于约束的方法使其在检测类型错误和提供有意义的错误消息方面表现出色。当统一失败时，算法可以准确识别类型不兼容的位置和原因。

尝试将非函数值（如‘42 true‘）应用会生成类型错误，因为整数‘42‘ 具有类型‘Int‘，但函数应用需要箭头类型。‘Int‘ 与Bool $\to t_0$ 的统一失败，产生清晰的错误消息。

更微妙的错误来自多态函数的不一致使用。虽然Algorithm W 优雅地处理let-多态，但它正确地拒绝在同一作用域内以不兼容方式使用函数的尝试。

3.4.5 复杂推断场景

真实世界的程序通常涉及函数、应用和数据结构的复杂组合，这些测试了Algorithm W 的全部力量。这些场景展示了算法如何处理复杂的约束传播和替换。

高阶函数提供了特别有趣的例子。表达式`\f -> \x -> f (f x)` 创建一个应用另一个函数两次的函数。Algorithm W 分配新的类型变量并构建捕获所涉及所有类型之间关系的约束。

让我们追踪这个推断过程。外部lambda 为‘f‘ 接收新的参数类型t_0。内部lambda 为‘x‘ 接收类型t_1。应用‘f x‘ 要求‘f‘ 对于某个新类型t_2 具有类型$t_1 \to t_2$。然后外部应用‘f (f x)‘ 要求‘f‘ 对于最终结果类型t_3 也具有类型$t_2 \to t_3$。

统一通过发现$t_2 = t_1$ 和最终类型是$(t_1 \to t_1) \to t_1 \to t_1$ 来解决这些约束。这捕捉了函数组合的本质：给定从某个类型到自身的函数，产生应用它两次的函数。

3.4.6 Y-组合子

Y-组合子代表了Hindley-Milner 类型系统的基本限制。这个著名的不动点组合子无法在我们的系统中被类型化，说明了用简单类型可以表达的内容与需要更高级类型系统特性的内容之间的重要边界。

Y-组合子定义为：

```
\f -> (\x -> f (x x)) (\x -> f (x x))
```

失败发生在尝试类型化自应用‘x x‘ 时。当类型检查器遇到这个表达式时，它必须为变量‘x‘ 分配一个同时是函数类型（要被应用）和参数类型（要传递给自身）的类型。这创建了约束，即某个类型变量t 必须等于$t \to \tau$，对于某个类型τ。出现检查防止这种无限类型，确保类型系统保持可判定和可靠。

这个限制不是错误，而是Hindley-Milner 类型系统的基本特征。Y-组合子需要更高级的类型系统特性，如递归类型或显式不动点运算符。真实的函数式编程语言通常提供像‘let rec‘ 这样的内置递归构造，这些由类型检查器特殊处理，避免了在项级别需要显式不动点组合子。

无法类型化Y-组合子说明了编程语言设计中表达性和可判定性之间的重要权衡。虽然存在可以处理自应用的更强大的类型系统，但它们以增加的复杂性和潜在不可判定的类型检查为代价。Algorithm W 选择可判定性和简单性，使其对真实编程语言实用，同时接受某些表达性限制。

3.4.7 相互递归

我们的Algorithm W 实现有一个重要限制：它不支持相互递归。相互递归定义发生在两个或多个函数相互调用时，创建在类型推断期间需要仔细处理的循环依赖。

考虑这个在我们的实现中会失败的例子：

```
let even = \n -> if n == 0 then true else odd (n - 1) in
let odd = \n -> if n == 0 then false else even (n - 1) in
odd 5
```

问题是当我们遇到‘even‘ 的定义时，函数‘odd‘ 尚未在作用域中，所以在‘even‘ 的主体中对‘odd‘ 的引用失败。标准Algorithm W 顺序处理let 绑定，这使得在没有额外机制的情况下相互递归不可能。

支持相互递归需要更多技术。你通常会通过以下方式之一来实现：

1. **依赖分析**：类型检查器必须分析定义的依赖图，以识别强连通分量（相互递归函数的组）。
2. **同时推断**：相互递归组中的所有函数必须一起类型化，通常通过最初为每个函数分配新的类型变量，然后同时解决所有约束。
3. **泛化延迟**：泛化步骤（引入多态性）必须延迟到所有相互依赖解决之后。

这些扩展显著复杂化了实现，超出了我们简单Algorithm W 演示的范围。

3.4.8 性能（或缺乏性能）

虽然Algorithm W 在理论上优雅，但实际实现必须考虑性能特征。算法的复杂度取决于表达式的大小和所涉及类型的复杂度。大多数真实程序涉及Algorithm W 高效处理的相对简单的类型约束。

我们的实现展示了核心算法，没有生产编译器中的优化。工业级实现通常采用像类型导向编译、约束缓存和增量类型检查这样的技术，以高效处理大型代码库。

Chapter 4

System F

虽然简单的λ 演算与HM 风格的类型方案引入了安全性和有限的多态性，但它也非常严格且表达能力不强。现在我们转向System F，也称为多态λ 演算，它通过引入**参数多态性**打破了这一限制：编写对类型通用的单一代码的能力。这是像Rust、Java 和C++ 这样的语言中泛型的理论基础，它代表了表达能力的巨大飞跃。

为了实现这一点，System F 扩展了项语言和类型语言。对类型语言最重要的添加是全称量词$\forall$ (forall)，它允许我们表达多态函数的类型。

$$\begin{aligned} \text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \text{Int} \mid \text{Bool} \\ \text{表达式} \quad & e ::= x \mid \lambda x : \tau.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \ldots \end{aligned}$$

类型$\forall\alpha.\tau$ 由‘Type::Forall(String, Box<Type>)‘ 表示。项语言现在包括两个专门用于处理多态性的新构造：

1. **类型抽象**（$\Lambda\alpha.e$）：这创建了一个多态函数。大写lambda（Λ）表示我们在抽象一个**类型变量** α，而不是项变量。这个表达式由‘Expr::TAbs(String, Box<Expr>)‘ 表示。
2. **类型应用**（$e[\tau]$）：这通过将多态函数应用于具体类型τ 来特化它。这是我们使用泛型函数的方式。这由‘Expr::TApp(Box<Expr>, Box<Type>)‘ 表示。

与无类型λ 演算的一个关键区别是，我们的项级抽象（λ）现在是显式注释的：$\lambda x : \tau.e$。程序员必须声明函数参数的类型。这是System F 的一个标志；它的力量以完全类型推断为代价，需要这些注释。

System F 力量的典型例子是多态恒等函数‘id‘。我们现在可以编写一个适用于任何类型α 的单一恒等函数。我们使用类型抽象创建它，并给其参数‘x‘ 通用类型α：

$$\text{id} = \Lambda\alpha.\lambda x : \alpha.x$$

这个函数的类型是$\forall\alpha.\alpha \to \alpha$。这个类型说明："对于所有类型$\alpha$，我是一个接受类型$\alpha$ 的参数并返回类型α 的值的函数。"

要使用这个多态函数，我们使用类型应用将其应用于类型。例如，要获得专门用于整数的恒等函数，我们将‘id‘ 应用于类型Int：

$$\text{id}[\text{Int}]$$

这是在类型检查级别发生的计算。这个类型应用的结果是一个新的、特化的表达式，$\lambda x : \text{Int}.x$，它具有相应的特化类型$\text{Int} \to \text{Int}$。然后我们可以将这个特化函数应用于一个值，如$(\lambda x : \text{Int}.x)\ 5$，最终归约为5。类型级应用（$e[\tau]$）和项级

应用（$e_1\ e_2$）的这种分离是System F 的基础。它提供了一种强大且安全的方式来编写泛型代码，但正如我们所看到的，它迫使程序员在注释上更加明确，这是一个权衡，直接激发了更现代语言实现中使用的双向算法。

4.1 抽象语法树

让我们看看类型语言（‘enum Type‘）和项语言（‘enum Expr‘）的完整抽象语法树：

4.1.1 类型级别

```
#![allow(unused)]
fn main() {
#[derive(Debug, Clone, PartialEq)]
pub enum Expr {
    Var(String),                                    // 变量: x
    App(Box<Expr>, Box<Expr>),                      // 应用: e1 e2
    Abs(String, Box<Type>, Box<Expr>),              // Lambda 抽象: λx: T. e
    TApp(Box<Expr>, Box<Type>),                     // 类型应用: e [T]
    TAbs(String, Box<Expr>),                        // 类型抽象: Λα. e
    Ann(Box<Expr>, Box<Type>),                      // 类型注释: e : T
    LitInt(i64),                                    // 整数字面量
    LitBool(bool),                                  // 布尔字面量
    Let(String, Box<Expr>, Box<Expr>),              // Let 绑定: let x = e1 in e2
    IfThenElse(Box<Expr>, Box<Expr>, Box<Expr>), // 条件: if e1 then e2 else e3
    BinOp(BinOp, Box<Expr>, Box<Expr>),             // 二元操作: e1 op e2
}
#[derive(Debug, Clone, PartialEq)]
pub enum BinOp {
    // 算术
    Add, // +
    Sub, // -
    Mul, // *
    Div, // /
    // 布尔
    And, // &&
    Or,  // ||
    // 比较
    Eq, // ==
    Ne, // !=
    Lt, // <
    Le, // <=
    Gt, // >
    Ge, // >=
}
impl std::fmt::Display for Type {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Type::Var(name) => write!(f, "{}", name),
            Type::ETVar(name) => write!(f, "^{}", name),
            Type::Int => write!(f, "Int"),
            Type::Bool => write!(f, "Bool"),
            Type::Arrow(t1, t2) => {
                // 如果左侧是箭头，添加括号以避免歧义
                match t1.as_ref() {
                    Type::Arrow(_, _) => write!(f, "({}) -> {}", t1, t2),
                    _ => write!(f, "{} -> {}", t1, t2),
                }
            }
            Type::Forall(var, ty) => write!(f, "{}. {}", var, ty),
        }
    }
}
#[derive(Debug, Clone, PartialEq)]
pub enum Type {
    Var(String),                // α (普通类型变量)
    ETVar(String),              // ^α (存在类型变量)
    Arrow(Box<Type>, Box<Type>), // A -> B
    Forall(String, Box<Type>),   // α. A
    Int,                        // Int
    Bool,                       // Bool
}
}
```

‘Type‘ 枚举定义了我们语言中所有可能类型的语法。它是我们用来描述表达式的词汇表。

- ‘Type::Var(String)‘：这表示一个简单的类型变量，如α 或β。这些是稍后将指定的类型的占位符，通常用于多态函数。
- ‘Type::ETVar(String)‘：这表示"存在类型变量"，通常写作$^{\alpha}$。这些是类型检查器内部使用的一种特殊变量，特别是在双向算法中。它们充当未知类型的占位符，算法需要在推断期间解决这些类型。

- ‘Type::Arrow(Box<Type>, Box<Type>)‘：这是函数类型，$\tau_1 \to \tau_2$。它表示接受第一个类型的参数并返回第二个类型结果的函数。
- ‘Type::Forall(String, Box<Type>)‘：这是全称量词，$\forall\alpha.\tau$。它是System F 的基石，表示多态类型。它读作："对于所有类型α，以下类型τ 成立。" ‘String‘是被绑定的类型变量α 的名称。
- ‘Type::Int‘ 和‘Type::Bool‘：这些是原始或基础类型。它们是直接内置到语言中的具体类型，分别表示64 位整数和布尔值。

4.1.2 值级别

‘Expr‘ 枚举定义了所有可运行表达式或项的语法。这是执行计算的代码。

- ‘Var(String)‘：项级变量，如‘x‘ 或‘f‘。它引用作用域中的值，如函数参数或‘let‘绑定的变量。
- ‘App(Box<Expr>, Box<Expr>)‘：这是函数应用，$e_1\ e_2$。它表示用参数e_2 调用函数e_1。
- ‘Abs(String, Box<Type>, Box<Expr>)‘：这是一个类型化的lambda 抽象，$\lambda x : \tau.e$。它定义了一个匿名函数。与纯λ 演算不同，参数（‘String‘）必须具有显式类型注释（‘Box<Type>‘）。最后的‘Box<Expr>‘ 是函数的主体。
- ‘TApp(Box<Expr>, Box<Type>)‘：这是类型应用，$e[\tau]$。这是特化多态函数的机制。表达式e 必须具有‘Forall‘ 类型，这个构造将其应用于特定类型τ，有效地"填充"泛型类型参数。
- ‘TAbs(String, Box<Expr>)‘：这是类型抽象，$\Lambda\alpha.e$。这是我们创建多态函数的方式。它引入了一个新的类型变量（‘String‘），可以在表达式主体（‘Box<Expr>‘）中使用。
- ‘Ann(Box<Expr>, Box<Type>)‘：这表示类型注释，$e : T$。这是对类型检查器的显式指令，断言表达式e 应该具有类型T。这在双向系统中对于指导推断过程和解决歧义非常宝贵。
- ‘LitInt(i64)‘ 和‘LitBool(bool)‘：这些是字面值。它们表示具体的、原始的值，这些值"内置"在语言中，对应于基础类型‘Int‘ 和‘Bool‘。它们是最简单的表达式形式，表示常量值。

4.2 类型规则

在深入实现细节之前，让我们建立管理System F 的形式类型规则。系好安全带，因为我们即将进入类型级魔法的有趣魔法世界！我们将引入一些可能一开始看起来很吓人的新符号，但一旦你习惯了它们，它们真的没有那么可怕。

- Γ **(Gamma)** - 类型上下文，就像一个字典，将变量映射到它们的类型，并跟踪我们目前所知道的内容。
- $\vdash$ **(转向)** - "证明"或"蕴涵"符号。当我们写$\Gamma \vdash e \Rightarrow A$ 时，我们说"给定上下文Γ，表达式e 综合类型A。"
- $\Rightarrow$ **(双右箭头)** - 推断模式，我们问"这个表达式有什么类型？"类型检查器为我们找出答案。
- $\Leftarrow$ **(双左箭头)** - 检查模式，我们说"请验证这个表达式具有期望的类型。"我们已经知道我们想要的类型。
- $\forall$ **(Forall)** - 全称量化，意思是"对于任何类型。"当我们看到$\forall\alpha.A$ 时，它意味

着"对于任何类型α，我们有类型A。"

- $\hat{\alpha}$ **(Hat Alpha)** - 存在类型变量，它们就像类型级别的未知数，系统在推断期间解决。将它们视为稍后填充的占位符。
- $\bullet$ **(Bullet)** - 我们推理规则中使用的应用判断符号。当我们写$A \bullet e \Rightarrow B$ 时，我们说"将类型A 应用于表达式e 产生类型B。"
- $<:$ **(子类型)** - 子类型关系，表达一个类型比另一个类型"更具体"。例如，Int $<:$ $\forall\alpha.\alpha$ 意味着Int 是多态类型的子类型。
- $[B/\alpha]A$ - 类型替换，在类型A 中用类型B 替换类型变量α 的所有出现。这是我们实例化多态类型的方式。

现在我们已经装备了这个符号工具包，让我们看看这些部分如何组合以创建System F 类型检查的优雅机制。

4.2.1 基本规则

变量规则从上下文中查找类型：

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}(\text{T-Var})$$

应用检查函数类型是否与参数匹配：

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}(\text{T-App})$$

Lambda 抽象引入新的变量绑定：

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.e \Leftarrow A \to B}(\text{T-Abs})$$

4.2.2 多态规则

全称引入允许我们对类型变量进行泛化：

$$\frac{\Gamma, \alpha \vdash e \Leftarrow A}{\Gamma \vdash e \Leftarrow \forall\alpha.A}(\text{T-ForallI})$$

全称消除实例化多态类型：

$$\frac{\Gamma \vdash e \Rightarrow \forall\alpha.A}{\Gamma \vdash e \Rightarrow [B/\alpha]A}(\text{T-ForallE})$$

类型注释允许从检查模式切换到推断模式：

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}(\text{T-Instr})$$

4.2.3 原始类型规则

整数字面量具有类型Int：

$$\frac{}{\Gamma \vdash n \Rightarrow \text{Int}}(\text{T-LitInt})$$

布尔字面量具有类型Bool：

$$\frac{}{\Gamma \vdash \text{true} \Rightarrow \text{Bool}}(\text{T-LitBool})$$

4.2.4 控制流规则

Let 绑定引入局部变量：

$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma, x : A \vdash e_2 \Rightarrow B}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow B}(\text{T-Let})$$

条件表达式需要Bool 条件和匹配的分支类型：

$$\frac{\Gamma \vdash e_1 \Leftarrow \text{Bool} \quad \Gamma \vdash e_2 \Rightarrow A \quad \Gamma \vdash e_3 \Leftarrow A}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow A}(\text{T-If})$$

4.2.5 二元操作规则

算术操作接受两个整数并返回一个整数：

$$\frac{\Gamma \vdash e_1 \Leftarrow \text{Int} \quad \Gamma \vdash e_2 \Leftarrow \text{Int}}{\Gamma \vdash e_1 \oplus e_2 \Rightarrow \text{Int}}(\text{T-Arith})$$

布尔操作接受两个布尔值并返回一个布尔值：

$$\frac{\Gamma \vdash e_1 \Leftarrow \text{Bool} \quad \Gamma \vdash e_2 \Leftarrow \text{Bool}}{\Gamma \vdash e_1 \wedge e_2 \Rightarrow \text{Bool}}(\text{T-Bool})$$

比较操作接受两个整数并返回一个布尔值：

$$\frac{\Gamma \vdash e_1 \Leftarrow \text{Int} \quad \Gamma \vdash e_2 \Leftarrow \text{Int}}{\Gamma \vdash e_1 < e_2 \Rightarrow \text{Bool}}(\text{T-Cmp})$$

相等操作是多态的，适用于任何类型：

$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1 = e_2 \Rightarrow \text{Bool}}(\text{T-Eq})$$

4.2.6 双向规则

模式切换允许检查推断结果：

$$\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow A}(\text{T-Sub})$$

为未知类型引入存在变量：

$$\frac{\Gamma, \hat{\alpha} \vdash e \Rightarrow A}{\Gamma \vdash e \Rightarrow [\hat{\alpha}/\alpha]A}(\text{T-InstL})$$

4.2.7 应用推断规则

应用推断处理函数类型不是立即可知的复杂情况：

带箭头类型的应用：

$$\frac{\Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash A \to B \bullet e_2 \Rightarrow B}(\text{T-AppArrow})$$

带存在变量的应用：

$$\frac{\Gamma[\hat{\alpha} := \hat{\alpha_1} \to \hat{\alpha_2}], \hat{\alpha_1}, \hat{\alpha_2} \vdash e_2 \Leftarrow \hat{\alpha_1}}{\Gamma \vdash \hat{\alpha} \bullet e_2 \Rightarrow \hat{\alpha_2}}(\text{T-AppEVar})$$

在这些规则中，$\Rightarrow$ 表示**推断**模式（综合类型），而$\Leftarrow$ 表示**检查**模式（验证期望类型）。帽子记号$\hat{\alpha}$ 表示存在类型变量，系统在推断期间解决这些变量。

4.3 核心实现

好的，现在我们超越了1970 年代的简单类型系统，进入了1980 年代的有趣内容！System F 的双向类型检查代表了处理多态类型而不需要在任何地方都要求完整类型注释的方法。双向方法将类型检查分为两个互补的模式：**推断**（从表达式综合类型）和**检查**（验证表达式符合期望类型）。这种划分允许系统优雅地处理类型部分已知或完全未知的情况，使语言更加符合人体工程学，同时保持类型安全。

4.3.0.1 核心数据结构

双向算法维护一个跟踪多种绑定和约束的上下文。我们的上下文系统需要处理不仅仅是项变量及其类型，还需要处理类型变量、存在变量以及它们之间的关系。

```
#![allow(unused)]
fn main() {
use std::collections::HashSet;
use std::fmt;
use crate::ast::{BinOp, Expr, Type};
use crate::errors::{TypeError, TypeResult};
pub type TyVar = String;
pub type TmVar = String;
#[derive(Debug, Clone)]
pub enum Entry {
    VarBnd(TmVar, Type),    // x: A
    TVarBnd(TyVar),         // α
    ETVarBnd(TyVar),        // ^α
    SETVarBnd(TyVar, Type), // ^α = τ
    Mark(TyVar),            // $α
}
#[derive(Debug, Clone)]
pub struct Context(Vec<Entry>);
impl Context {
    pub fn new() -> Self {
        Context(Vec::new())
    }

    pub fn push(&mut self, entry: Entry) {
        self.0.push(entry);
    }

    pub fn find<F>(&self, predicate: F) -> Option<&Entry>
    where
        F: Fn(&Entry) -> bool, {
        self.0.iter().find(|entry| predicate(entry))
```

```
    }

    pub fn break3<F>(&self, predicate: F) -> (Vec<Entry>, Option<Entry>, Vec<Entry>)
    where
        F: Fn(&Entry) -> bool, {
        if let Some(pos) = self.0.iter().position(predicate) {
            let left = self.0[..pos].to_vec();
            let middle = self.0[pos].clone();
            let right = self.0[pos + 1..].to_vec();
            (left, Some(middle), right)
        } else {
            (self.0.clone(), None, Vec::new())
        }
    }

    pub fn from_parts(left: Vec<Entry>, middle: Entry, right: Vec<Entry>) -> Self {
        let mut ctx = left;
        ctx.push(middle);
        ctx.extend(right);
        Context(ctx)
    }
}
pub struct BiDirectional {
    counter: usize,
}
impl BiDirectional {
    pub fn new() -> Self {
        Self { counter: 0 }
    }

    fn fresh_tyvar(&mut self) -> TyVar {
        let var = format!("α{}", self.counter);
        self.counter += 1;
        var
    }
}
}
```

上下文条目表示类型检查器在双向推断过程中需要跟踪的不同类型信息。变量绑定，表示为‘VarBnd(TmVar, Type)‘，将项变量与其类型关联，例如在当前作用域中记录‘x‘ 具有类型‘Int‘。类型变量绑定，表示为‘TVarBnd(TyVar)‘，将类型变量引入作用域，例如出现在全称量化$\forall\alpha.\ldots$ 中的α。

存在类型变量绑定，写作‘ETVarBnd(TyVar)‘，引入存在类型变量，表示需要通过约束求解确定的未知类型。已解决的存在类型变量绑定，表示为‘SETVarBnd(TyVar, Type)‘，记录在推断过程中为存在变量发现的具体解。最后，标记条目，表示为‘Mark(TyVar)‘，标记作用域的开始，以便在退出作用域时正确垃圾回收类型变量。

上下文本身以类似堆栈的结构维护这些条目，其中顺序对于作用域和变量解析至关重要。

4.3.0.2 类型替换和应用

类型替换构成了我们System F 实现的计算核心。当我们实例化多态类型或解决存在变量时，我们需要在整个复杂类型表达式中系统地用具体类型替换类型变量。

替换必须正确处理变量捕获。当替换到‘Forall‘ 类型时，我们必须确保绑定变量不与替换类型冲突。这类似于λ 演算中的α 转换，但在类型级别操作。

上下文应用通过将存在变量解的当前状态应用于类型来扩展替换。这个操作是必不可少的，因为我们的算法逐步构建存在变量的解。当我们了解更多关于未知类型的信息时，我们需要将这些信息传播到我们正在处理的所有类型中。

4.3.0.3 双向算法核心

推断模式从表达式综合类型。我们的实现使用模块化方法，其中主推断函数委托给每个语法形式的专门方法。

每个方法包括注释，将其链接到它实现的形式类型规则，使理论和实现之间的对应关系明确。

检查模式验证表达式符合期望类型。这是算法即使在类型没有完全确定时也能取得进展的地方。

检查模式提供专门处理，利用已知类型信息更有效地指导推断过程。对于lambda 表达式，当检查$\lambda x : \tau_1.e$ 对类型$\tau_1 \to \tau_2$ 时，算法可以立即验证参数类型匹配，然后递归检查主体e 对期望返回类型τ_2。这种直接分解避免了需要综合类型然后验证兼容性。

当检查对全称类型$\forall\alpha.\tau$ 时，算法将类型变量α 引入上下文，然后检查表达式对实例化类型τ。这种方法确保全称量化被正确处理，同时保持可靠类型检查所需的作用域规则。当直接检查策略不适用于当前表达式和期望类型组合时，算法回退到综合加子类型方法，其中它首先为表达式综合类型，然后验证这个综合类型是期望类型的子类型。

4.3.0.4 子类型和实例化

System F 包括一个处理多态类型之间关系的子类型系统。关键洞察是‘α . τ‘比τ 的任何特定实例化更一般。

子类型规则捕捉了管理System F 中类型兼容性的几个基本关系。函数类型表现出经典的反变-协变模式，其中接受更一般参数并返回更具体结果的函数被认为是具有更具体参数要求和更一般返回类型的函数的子类型。这意味着类型$A_1 \to B_1$ 的函数是$A_2 \to B_2$ 的子类型，当$A_2 \leq A_1$（参数反变）和$B_1 \leq B_2$（结果协变）时。

全称量化遵循这样的原则：$\forall\alpha.\tau_1 \leq \tau_2$ 成立，如果当α 用新的存在变量实例化时$\tau_1 \leq \tau_2$，有效地允许多态类型通过它们的实例化相关联。当子类型涉及存在变量时，算法必须解决关于这些变量应该被实例化为什么的约束，以便满足子类型关系，通常导致生成在整个系统中传播的额外约束。

实例化判断处理多态类型检查的核心复杂性。当我们有像$\hat{\alpha} \leq \tau$ 这样的约束（存在变量应该最多与某个类型一样一般）时，我们需要找到适当的实例化。

左实例化（`inst_l`）处理存在变量在约束左侧的情况。这通常意味着我们正在寻找满足约束的最一般类型。

右实例化（`inst_r`）处理对偶情况，其中存在变量在右侧。这通常意味着我们正在寻找满足约束的最具体类型。

实例化算法包括对在约束求解期间出现的几种复杂场景的仔细处理。到达关系发生在两个存在变量相互约束时，要求算法确定哪个变量应该用另一个变量来求解，同时保持适当的排序约束。箭头类型实例化需要将函数类型分解为其组件参数和返回类型，为每个组件创建必须一致求解的单独实例化约束。

实例化和全称量化之间的交互提出了特别的挑战，因为算法必须确保多态类型被正确实例化，同时保持防止类型变量逃逸其预期作用域的作用域规则。这些情况需要约束管理，以确保在整个求解过程中维护所有关系。

4.3.0.5 出现检查

可靠类型推断的一个关键组件是出现检查，它防止在统一期间出现无限类型。当解决像$\hat{\alpha} := \tau$ 这样的约束时，我们必须确保α 不会出现在τ 中，因为这将创建

一个循环类型。

出现检查在InstLSolve 和InstRSolve 实例化情况下应用。没有这个检查，类型系统可能接受会导致无限类型的程序，违反类型检查的可判定性。

4.3.0.6 应用推断

System F 中的函数应用需要仔细处理，因为函数类型可能不是立即可见的。`infer_app` 判断实现了前面定义的T-AppArrow 和T-AppEVar 规则：

实现通过不同的推断规则处理两个核心应用场景。T-AppArrow 规则在函数具有已知箭头类型$A \to B$ 时应用，允许算法检查参数对A 并返回B 作为结果类型。这个直接情况对应于实现中的‘Type::Arrow‘ 模式，表示标准函数应用场景。

T-AppEVar 规则处理更复杂的情况，其中函数类型是存在变量$\hat{\alpha}$。在这种情况下，算法将存在变量实例化为$\hat{\alpha_1} \to \hat{\alpha_2}$，使用新的存在变量，然后检查参数对$\hat{\alpha_1}$ 并返回$\hat{\alpha_2}$ 作为结果类型。这对应于‘Type::ETVar‘ 情况，即使在函数类型最初未知时也能实现类型推断。

当函数具有多态‘Forall‘ 类型时，实例化通过使用SubAllL 规则的子类型机制处理，而不是直接在应用推断中。这种设计选择通过将多态实例化路由到已建立的子类型基础设施来确保可靠性，并遵循标准双向算法设计模式。

4.4 示例

System F 的力量在于它表达在不同类型上工作的多态计算同时保持类型安全的能力。本章探索了一系列示例，展示了我们实现的表达能力，从简单的多态函数到复杂的高阶多态场景。

我们的示例来自实现中的真实测试用例，展示了我们的解析器接受的语法和我们的双向算法推断的类型。这些示例说明了System F 的关键特性，并展示了类型系统如何指导程序构造。

4.4.0.7 基本多态函数

System F 的基本构建块是多态恒等函数。这个简单示例展示了全称量化和类型应用的最基本形式。

让我们检查我们的实现如何处理测试套件中的核心示例：

```
基本字面量
42
true
false

类型注释
\x : Int -> x
\x : Bool -> x
(\x : Int -> x) 42
(\x : Bool -> x) true

使用 forall 的多态恒等
forall a. \x : a -> x

带括号的类型应用
(forall a. \x : a -> x) [Int]
(forall a. \x : a -> x) [Bool]

高阶多态示例
\f : (forall a. a -> a) -> f
(\f : (forall a. a -> a) -> f) (forall a. \x : a -> x)

嵌套类型抽象
forall a. forall b. \x : a -> \y : b -> x
forall a. forall b. \x : a -> \y : b -> y

复杂多态示例
```

```
forall a. \f : (a -> a) -> \x : a -> f x

函数类型
\f : (Int -> Int) -> \x : Int -> f x
(\f : (Int -> Int) -> \x : Int -> f x) (\y : Int -> y) 42

错误案例应由类型检查器处理
(\x : Int -> x) true
```

这些测试用例揭示了我们System F 实现的几个重要方面：

字面量类型和基本函数 最简单的情况涉及字面值和单态函数，其中我们的双向算法产生清晰、精确的类型：

- ‘42 : Int‘ - 整数字面量立即接收基础类型‘Int‘
- ‘true : Bool‘ 和‘false : Bool‘ - 布尔字面量类似地接收它们的基础类型
- `\x : Int -> x : Int -> Int` - 单态lambda 表达式接收具体函数类型

类型推断算法解决所有约束以产生可能的最具体类型。当我们提供显式类型注释时，算法可以在没有歧义的情况下确定完整的函数类型。

类型注释和显式类型化 System F 要求lambda 参数上的显式类型注释，我们的双向算法推断完整的函数类型：

- `\x : Int -> x` 产生类型`Int -> Int`
- `(\x : Int -> x) 42` 应用此函数以产生类型`Int`

这些示例展示了算法如何处理显式注释和类型推断之间的交互。参数类型由注释固定，算法推断返回类型必须匹配参数类型，因为我们返回未更改的参数。

实践中的全称量化 System F 的真正力量随着全称量化而出现。多态恒等函数清楚地展示了这一点：

- `forall a. \x : a -> x : a. a -> a`

这个示例展示了我们实现的几个重要方面：

1. **类型抽象**：‘forall a.‘ 语法将类型变量引入作用域
2. **多态参数**：参数‘x : a‘ 使用量化的类型变量
3. **完整类型推断**：算法正确解决所有类型约束以产生预期的多态类型

类型应用和实例化 类型应用允许我们将多态函数特化为特定类型：

- `(forall a. \x : a -> x) [Int] : Int -> Int`
- `(forall a. \x : a -> x) [Bool] : Bool -> Bool`

两个应用都产生预期的具体函数类型，展示了我们的算法正确实例化多态函数与请求的类型，并将所有类型约束解决为其最终形式。

4.4.0.8 高阶多态

System F 支持高阶多态，其中多态类型可以出现在参数位置。这创建了接受多态参数的函数：

- `\f : (forall a. a -> a) -> f :` $\hat{\alpha}_0$ `->` $\hat{\alpha}_1$

这个函数接受任何类型为‘forall a. a -> a‘ 的参数（多态恒等函数）并返回它不变。高阶性质意味着调用者必须提供一个适用于所有类型的函数，而不仅仅是某个特定类型。

应用示例展示了这一点：

- `(\f : (forall a. a -> a) -> f) (forall a. \x : a -> x) :` $\hat{\alpha}_0$

这里我们将多态恒等函数传递给期望恰好该类型签名的函数。这展示了System F 类型系统的精确性和表达能力。

4.4.0.9 嵌套类型抽象

System F 允许多个类型抽象嵌套，创建在多个类型参数上多态的函数：

- `forall a. forall b. \x : a -> \y : b -> x : a. b. a -> b -> a`
- `forall a. forall b. \x : a -> \y : b -> y : a. b. a -> b -> b`

这些示例创建了：

1. 在两个类型‘a‘ 和‘b‘ 上多态的函数
2. 接受这些相应类型的参数
3. 返回第一个或第二个参数

组合逻辑中的‘K‘ 和‘S‘ 组合子可以自然地以这种风格表达，展示了System F 对抽象计算的表达能力。作为一个有趣的旁注，你可以仅用‘K‘ 和‘S‘ 表达每个其他函数，尽管在实践中这更像是一个理论奇观，因为它难以想象地低效。

4.4.0.10 复杂多态

更多示例展示了System F 如何处理多态和函数应用的复杂组合：

- `forall a. \f : (a -> a) -> \x : a -> f x : a. (a -> a) -> a -> a`

这创建了一个函数，它：

1. 在类型‘a‘ 上多态
2. 接受函数‘f : a -> a‘（‘a‘ 上的自同态）
3. 接受值‘x : a‘
4. 将函数应用于值

这种模式在函数式编程中是基础的，展示了System F 如何自然地表达高阶多态函数。

Chapter 5

System Fω

System Fω 是编程语言理论基础中最重要的类型系统之一，它将参数多态性与高阶类型（higher-kinded types）相结合，实现了强大的抽象机制。该系统为像Haskell 这样的现代函数式编程语言提供了理论基础，并为包括函子、单子和泛型编程在内的高级编程模式提供了所需的表达能力。

要理解System Fω 的重要性，我们必须首先通过称为λ **立方体**的系统分类来审视它在更广泛的类型系统领域中的位置。

5.1 λ 立方体

λ 立方体由Henk Barendregt 引入，通过考虑三个独立的抽象维度，提供了一种系统化的方式来理解不同类型系统之间的关系。立方体的每个维度对应于项和类型之间不同类型的依赖关系，其中"依赖"指的是项或类型绑定项或类型的能力。

λ 立方体的三个正交轴对应于：

$\rightarrow$-轴（类型依赖于项）：启用依赖类型，其中类型的结构可以依赖于项的值。这允许像‘Vector n‘ 这样的类型，其中类型携带关于项级值的信息。

$\uparrow$-轴（项依赖于类型）：启用多态性，其中项可以抽象类型参数并依赖于类型参数。这允许像`identity` : $\forall\alpha.\ \alpha \rightarrow \alpha$ 这样的函数，在所有类型上统一工作。

$\nearrow$-轴（类型依赖于类型）：启用类型运算符，其中类型可以抽象其他类型并依赖于其他类型。这允许像`Maybe` : $* \rightarrow *$ 这样的类型构造函数，它们接受类型作为参数并产生新类型。

立方体的八个顶点来自组合这三个依赖维度的不同方式。每个顶点代表一个不同的类型化系统，通过启用或禁用每种抽象形式获得：

简单类型λ 演算（$\lambda \rightarrow$）：立方体的基础，仅支持项抽象。函数可以抽象项（$\lambda x : \tau.e$），但类型保持固定和简单。

System F（$\lambda 2$）：通过类型抽象添加多态性，使项能够抽象类型（$\Lambda\alpha.e$）。这引入了参数多态性，其中像恒等函数这样的函数可以在所有类型上统一工作。

System Fω（$\lambda\omega$）：通过添加类型运算符引入高阶类型，允许类型抽象类型（$\lambda\alpha : \kappa.\tau$）。这使能够抽象像‘Maybe‘ 和‘List‘ 这样的类型构造函数。

Lambda P（λP）：添加依赖类型，其中类型可以依赖于项。这允许像‘Vector n‘ 这样的类型，其中长度‘n‘ 是项级值，能够精确指定数据结构的属性。

System Fω（$\lambda 2\omega$）：将多态性与高阶类型相结合，使项能够抽象类型，类型也能够抽象类型。这是我们的目标系统，为现代函数式编程提供了所需的表达能力。

System Fω-P（$\lambda\omega P$）：将高阶类型与依赖类型相结合，允许依赖于项级值的类型级计算。

System FP（$\lambda 2P$）：将多态性与依赖类型相结合，使函数在类型和值上都是参数的，同时允许类型依赖于这些值。

构造演算（$\lambda 2\omega P$）：最具表达力的角落，结合了所有三种抽象形式。该系统是像Coq 这样的证明助手的基础，并使类型能够表达任意逻辑命题。

λ 立方体表明这八个系统形成了一个自然的层次结构，每个系统都是其下系统的保守扩展。System Fω 处于一个最佳位置，提供了显著的表达能力，同时保持了可判定的类型检查和相对简单的实现技术。

System F 引入了对类型的全称量化（$\forall\alpha.\tau$），而System Fω 将其扩展到对任意种类（kind）的类型构造函数的量化。这使我们能够编写不仅在像‘Int‘ 或‘Bool‘ 这样的类型上多态的函数，而且在像‘Maybe‘ 或‘List‘ 这样的类型构造函数上多态的函数，甚至是在接受多个类型参数的高阶类型构造函数上多态的函数。

类型系统通过**种类**（kinds）分层，种类对类型进行分类，就像类型对项进行分类一样：

$$\begin{aligned}
\text{种类} \quad & \kappa ::= \star \mid \kappa_1 \to \kappa_2 \\
\text{类型} \quad & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha : \kappa.\tau \mid \tau_1\ \tau_2 \mid \lambda\alpha : \kappa.\tau \\
\text{项} \quad & e ::= x \mid \lambda x : \tau.e \mid e_1\ e_2 \mid \Lambda\alpha : \kappa.e \mid e[\tau]
\end{aligned}$$

这种层次结构自然向上扩展：正如我们需要类型来分类项，需要种类来分类类型，我们原则上可以引入**排序**（sorts）来分类种类。

虽然System Fω 出于实际原因在种类级别停止，但理论框架暗示了一个无限塔：项有类型，类型有种类，种类有排序，排序有超排序，依此类推。这种无限层次结构，在类型理论中称为**累积层次结构**，反映了每个数学对象必须由更高抽象级别的某物分类的基本原则。

5.2 高阶类型

System Fω 的关键创新是种类系统。虽然像‘Int‘ 和‘Bool‘ 这样的普通类型具有种类$*$（读作"星"），但类型构造函数具有函数种类：

- ‘Maybe‘ 具有种类$* \to *$（接受一个类型，返回一个类型）
- ‘Either‘ 具有种类$* \to * \to *$（接受两个类型，返回一个类型）
- 假设的‘StateT‘ 可能具有种类$* \to (* \to *) \to * \to *$

这种分层使能够精确推理类型级函数，同时保持可判定的类型检查。

我们的实现通过一个丰富的表面语言展示了这些概念，该语言编译为核心System Fω 演算。表面语法提供了熟悉的代数数据类型和模式匹配：

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data List a = Nil | Cons a (List a)
```

这些声明创建了适当种类的类型构造函数。‘Maybe‘ 成为一个类型级函数，当应用于像‘Int‘ 这样的类型时，产生具体类型‘Maybe Int‘。

5.3 类型级计算

System Fω 通过类型应用和类型抽象支持类型级计算。虽然我们的实现专注于基础机制，但理论系统允许在类型级别进行任意计算：

- **类型应用**：‘F τ‘ 将类型构造函数‘F‘ 应用于类型‘τ‘
- **类型抽象**：‘λ α : κ . τ‘ 创建类型级函数

这使能够进行类型级编程，尽管我们的实现专注于实际编程语言设计所需的基本特性。

5.4 多态数据结构

全称量化与高阶类型的结合使能够优雅地表达多态数据结构和函数：

```
map :: forall a b. (a -> b) -> List a -> List b
foldRight :: forall a b. (a -> b -> b) -> b -> List a -> b
```

这些函数在所有类型上统一工作，展示了System Fω 中参数多态性的力量。

5.5 实现架构

我们的System Fω 实现由几个关键组件协同工作组成：

- **表面语言**：具有代数数据类型、模式匹配和类型推断的用户友好语法
- **核心语言**：具有种类、类型抽象和应用的显式System Fω 演算
- **精化**：从表面到核心的转换，插入隐式类型参数和抽象
- **类型推断**：基于DK 工作列表方法的双向算法
- **种类推断**：自动推断类型构造函数和类型表达式的种类

实现表明，通过仔细的算法设计和实现技术，类型系统可以变得实用。

我们的实现使用双向类型检查，将问题分为两个互补的模式：

- **综合（⇒）**：给定一个表达式，确定其类型
- **检查（⇐）**：给定一个表达式和期望类型，验证兼容性

这种方法处理高阶多态性和类型应用的复杂性，同时保持可判定性并提供良好的错误消息。

5.6 语言设计

我们的System Fω 实现采用两层架构，将面向用户的语法与类型检查器使用的内部表示分离。这种设计模式在编译器中很常见，使我们能够提供符合人体工程学的编程体验，同时为类型检查算法保持清洁的理论基础。

表面语言提供熟悉的语法，具有代数数据类型、模式匹配和隐式类型推断。**核心语言**提供具有种类、类型抽象和应用的显式System Fω 表示。这些层之间的转换处理插入隐式类型参数和管理System Fω 启用的类型级计算的复杂过程。

5.6.1 表面语言语法

表面语言提供类似Haskell 的语法，程序员可以自然地使用。用户编写程序时不需要显式的类型抽象或应用，编译器在精化到核心语言期间自动插入这些内容。

数据类型声明 代数数据类型形成我们表面语言的基础。这些声明创建类型构造函数和值构造函数：

```
data Bool = True | False;
data Maybe a = Nothing | Just a;
data Either a b = Left a | Right b;
data List a = Nil | Cons a (List a);
```

每个数据声明引入：

- 一个**类型构造函数**（如‘Maybe‘ 或‘List‘），可以应用于类型参数
- **值构造函数**（如‘Nothing‘、‘Just‘、‘Nil‘、‘Cons‘），用于创建类型的值
- 由类型参数的数量和使用确定的隐式**种类信息**

表面语言允许自然的类型参数语法，其中‘data List a‘ 自动暗示‘List‘ 具有种类$* \to *$。

类型签名和方案 类型方案使用熟悉的量词语法提供显式多态类型签名：

```
identity :: a -> a;
const :: a -> b -> a;
map :: (a -> b) -> List a -> List b;
```

表面语言使用隐式量化——类型签名中的任何自由类型变量都会自动全称量化。这提供了像Haskell 这样的语言的便利性，同时保持了System Fω 的理论精确性。

5.6.2 核心语言表示

核心语言提供具有完整类型级计算能力的System Fω 的显式编码。这种表示通过暴露表面语言中的所有隐式操作使类型检查变得易于处理。

核心类型和种类 种类系统分层分类类型：

- **‘Star‘**（$*$）用于像‘Int‘、‘Bool‘、‘List Int‘ 这样的普通类型
- **‘Arrow(k1, k2)‘** 用于像‘Maybe : * -> *‘ 或‘Either : * -> * -> *‘ 这样的类型构造函数

核心类型包括System Fω 的所有表达能力：

- **类型变量**：普通（‘Var‘）和存在（‘ETVar‘）变量用于统一
- **类型构造函数**：内置类型（‘Con‘），如‘Int‘、‘Bool‘
- **函数类型**：显式箭头类型（‘Arrow‘）
- **全称量化**：具有种类注释绑定变量的‘Forall‘ 类型
- **类型应用**：‘App‘ 用于将类型构造函数应用于参数
- **类型抽象**：‘TAbs‘ 用于创建类型级函数

核心表示使表面语言中保持隐式的所有类型级计算变得显式。

5.7 词法分析和解析

我们的System Fω 实现采用精心设计的解析策略，与生产Haskell 实现中发现的有机演化的复杂性形成鲜明对比。虽然Haskell 的语法通过无数的语言扩展和"实用"妥协演化了数十年，导致上下文相关解析、空白敏感性和复杂的布局规则，但我们的实现通过采用具有清晰分隔符的显式语法，故意避开了这些复杂性，实现了独立于上下文或空白变化的可预测结果的直接LALR(1) 解析。

5.7.1 词法分析策略

词法分析阶段将源文本转换为解析器可以确定性处理的标记流。与必须跟踪缩进级别并插入布局标记的Haskell 上下文相关词法分析器不同，我们的词法分析器作为纯有限状态机运行。

词法分析器识别几个类别的标记，这些标记捕获我们表面语言的基本元素。像‘data‘、‘match‘ 和‘forall‘ 这样的关键字建立了声明和表达式的语法框架。标识符通过命名约定区分类型变量、项变量和构造函数名称，词法分析器一致地强制执行这些约定。

运算符得到特殊处理，以处理函数箭头（‘->‘）和类型签名标记（‘::‘），两者对于表达类型和函数签名都至关重要。包括括号、大括号和分号在内的分隔符提供显式结构，消除了基于布局的语法中固有的歧义。

5.8 双向类型检查

类型推断引擎代表了我们System Fω 实现中最重要的组件。围绕DK（Dunfield-Krishnaswami）工作列表算法构建，它处理高阶多态性、存在类型变量和双向类型检查之间的复杂交互，这些使System Fω 既强大又具有挑战性，难以高效实现。

该算法在判断和约束的工作列表上运行，系统地将复杂的类型检查问题简化为更简单的问题，直到所有约束都得到解决。这种方法为System Fω 提供可判定的类型推断，同时保持主类型并支持多态编程模式。

值得注意的是，这种方法与我们之前使用的直接树遍历方法不同。现在我们遍历语法树并发出约束，然后由工作列表算法解决，然后将结果反向传播到原始语法树以给出最终类型。这个系统允许引入更复杂的规则，但它也使整个过程"非局部"，因为我们在查看由许多不同表达式生成的约束，因此错误报告变得更具挑战性，因为将约束问题追溯到其原始语法需要相当多的簿记。

5.8.1 类型规则

现在我们将大胆进入下一代类型检查。我们的类型系统中有一些新符号，但它们大多与以前相同，除了添加了新的种类级操作。

- Γ **(Gamma)** - 存储变量及其类型信息的类型上下文。
- $\vdash$ **(转向)** - "证明"关系，意思是"给定这个上下文，我们可以得出这个判断。"
- $\Rightarrow$ **(双右箭头)** - 综合模式，类型检查器找出表达式具有什么类型。
- $\Leftarrow$ **(双左箭头)** - 检查模式，我们验证表达式具有期望类型。
- $\forall\alpha$ **(Forall Alpha)** - 全称量化，意思是"对于任何类型α。"
- :: **(双冒号)** - "具有种类"关系，告诉我们类型构造函数属于哪个类别。
- $\star$ **(星)** - 像‘Int‘、‘Bool‘ 和‘String‘ 这样的具体类型的种类。

- $\square$ **(方框)** - 种类的种类，种类本身具有的种类。
- $\rightarrow$ **(箭头)** - 项级别的函数类型，或类型级别的种类箭头。
- $[B/\alpha]A$ - 类型替换，在类型A 中用类型B 替换类型变量α 的所有出现。
- $\Lambda\alpha$ **(大Lambda)** - 在类型级别创建多态函数的类型抽象。
- κ **(Kappa)** - 表示类型系统中未知类别的种类变量。
- $\overline{x}$ - 上划线记号，表示元素序列或列表，如多个变量或参数。

在检查DK 工作列表算法实现之前，让我们建立管理System Fω 的形式类型规则。这些规则扩展了System F，添加了高阶类型和更多多态性。

基本表达式规则 变量查找在上下文中查找类型：

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}(\text{T-Var})$$

具有双向检查的函数应用：

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}(\text{T-App})$$

检查模式中的Lambda 抽象：

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.e \Leftarrow A \to B}(\text{T-Lam})$$

类型应用和抽象 类型应用实例化多态表达式：

$$\frac{\Gamma \vdash e \Rightarrow \forall \alpha :: \kappa.A \quad \Gamma \vdash B :: \kappa}{\Gamma \vdash e\ [B] \Rightarrow [B/\alpha]A}(\text{T-TApp})$$

类型抽象引入全称量化：

$$\frac{\Gamma, \alpha :: \kappa \vdash e \Leftarrow A}{\Gamma \vdash \Lambda\alpha :: \kappa.e \Leftarrow \forall \alpha :: \kappa.A}(\text{T-TAbs})$$

高阶类型规则 种类检查确保类型构造函数是良构的：

$$\frac{}{\Gamma \vdash \star :: \square}(\text{K-Type})$$

$$\frac{\Gamma \vdash \kappa_1 :: \square \quad \Gamma \vdash \kappa_2 :: \square}{\Gamma \vdash \kappa_1 \to \kappa_2 :: \square}(\text{K-Arrow})$$

类型构造函数应用：

$$\frac{\Gamma \vdash F :: \kappa_1 \to \kappa_2 \quad \Gamma \vdash A :: \kappa_1}{\Gamma \vdash F\ A :: \kappa_2}(\text{K-App})$$

模式匹配规则 构造函数模式检查数据类型：

$$\frac{\Gamma \vdash e \Rightarrow T \quad \overline{\Gamma \vdash p_i \Leftarrow T \rightsquigarrow \Delta_i \quad \Gamma, \Delta_i \vdash e_i \Leftarrow A}}{\Gamma \vdash \text{match } e \text{ with } \overline{p_i \to e_i} \Rightarrow A}(\text{T-Match})$$

双向控制 从推断到检查的模式切换：

$$\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow A}(\text{T-Sub})$$

存在变量实例化：

$$\frac{\Gamma, \hat{\alpha} :: \kappa \vdash e \Rightarrow A}{\Gamma \vdash e \Rightarrow [\hat{\alpha}/\alpha]A}(\text{T-InstL})$$

在这些规则中，$\Rightarrow$ 表示**推断**模式（综合类型），$\Leftarrow$ 表示**检查**模式（验证期望类型），:: 将类型与其种类关联，$\hat{\alpha}$ 表示在推断期间解决的存在类型变量。

5.8.2 DK 工作列表算法

DK 算法将类型推断表示为约束求解问题。它不是递归遍历表达式并立即做出类型检查决策，而是在工作列表中累积约束并系统地处理它们。

工作列表包含三种基本类型的条目，它们协同工作来管理类型检查过程。类型变量绑定跟踪类型变量及其当前状态，可以是全称（来自显式量词）、存在（表示需要推断的未知类型）或已解决（通过约束求解确定的存在变量）。项变量绑定将程序变量与其推断或声明的类型关联，维护源代码中标识符与其类型信息之间的映射。判断表示待完成的核心类型检查和推断任务，编码像子类型检查、类型综合和类型验证这样的操作。

这种三方结构允许算法在系统地累积正确做出决策所需信息的同时推迟复杂决策。通过将变量管理与实际类型检查工作分离，算法可以处理复杂的依赖关系，并确保所有约束以正确的顺序解决。

5.8.3 双向类型检查

DK 算法采用双向类型检查，将推断分为两个互补的模式，它们协同工作来处理System Fω 的复杂性。

综合模式（$\Rightarrow$）分析表达式以确定其类型，从表达式的结构开始工作以产生类型信息。当遇到变量时，算法在当前类型上下文中查找其类型，根据需要实例化任何多态方案以产生当前使用的具体类型。函数应用要求算法推断函数和参数类型，然后统一它们以确保应用是良类型的。类型应用用具体类型参数实例化多态类型，需要仔细管理类型变量替换和作用域。

检查模式（$\Leftarrow$）验证表达式符合期望类型，从已知类型信息开始向后工作以验证表达式结构。对于lambda 表达式，算法可以立即分解期望函数类型以提取参数和返回类型，然后检查lambda 参数对期望参数类型，并递归检查主体对期望返回类型。当检查多态类型时，算法为量化变量引入新的类型变量，并检查表达式对实例化类型。

当直接检查策略不适用时，算法回退到综合加子类型方法，其中它首先为表达式综合类型，然后验证这个综合类型是期望类型的子类型。检查模式通常比综

合更高效，因为它可以对期望类型结构做出更强的假设，允许算法修剪搜索空间并进行更有针对性的推断。

5.8.4 存在类型变量

存在变量表示使类型推断成为可能的未知类型。它们充当占位符，通过统一和约束传播得到解决。

当算法遇到类型未知的表达式时，它遵循系统的约束生成过程。算法创建新的存在变量来表示未知类型，确保每个未知类型获得一个唯一的占位符，可以在整个求解过程中跟踪。然后这些存在变量通过约束生成与已知类型相关，创建捕获程序结构隐含的类型关系的方程和不等式。

生成的约束被添加到工作列表中以进行系统解决，允许算法推迟复杂的求解决策，直到有足够的信息可用。随着约束得到解决，存在变量接收到具体解，这些解在整个约束系统中传播，可能以级联效应使能够解决额外约束，逐渐确定所有未知类型。

5.8.5 子类型和多态实例化

System Fω 的子类型关系捕获了更多态类型是较少多态类型的子类型的想法。这使能够灵活使用多态函数。

在比较类型进行子类型化之前，算法执行一个称为**zonking**的关键步骤：用其解替换所有已解决的存在变量。这确保子类型比较看到完全解析的类型，而不是过时的变量名称。没有zonking，像Bool $\leq \hat{\alpha}_1$、$\hat{\alpha}_1 \leq \hat{\alpha}_0$、$\hat{\alpha}_0 \leq$ Int 这样的约束链永远不会检测到矛盾Bool $\leq$ Int，因为每次比较都会看到未解决的变量名称而不是其解析类型。

子类型算法处理管理System Fω 中类型兼容性的几个基本关系。自反性确保每个类型被认为是其自身的子类型，为子类型派生提供基本情况。传递性允许子类型关系通过中间类型组合，因此如果$A \leq B$ 和$B \leq C$，则$A \leq C$ 自动成立。

函数类型表现出经典的反变-协变模式，其中函数类型$A_1 \to B_1$ 是$A_2 \to B_2$ 的子类型，如果$A_2 \leq A_1$（参数反变）和$B_1 \leq B_2$（结果协变）。全称量化遵循这样的原则：$\forall\alpha.A \leq B$ 成立，如果当α 用新的存在变量实例化时$A \leq B$。存在实例化要求算法以满足子类型约束的方式解决存在变量，通常导致创建必须解决的额外约束。

5.9 代码生成

本节是可选的，但我们想探索一点如何从高级λ 演算（即System F）一直到可执行机器代码需要几个转换，这些转换保留我们程序的语义，同时使它们适应现代CPU 架构的现实。

我们将构建一个最小化的生成管道，通过三个主要阶段将我们的类型化函数程序转换为命令式机器代码：

1. **类型擦除**：首先我们移除指导类型检查的类型信息
2. **闭包转换**：其次我们使嵌套函数的隐式环境捕获显式
3. **代码生成**：最后我们的代码生成框架（在我们的例子中是Cranelift）从我们的闭包转换代码为目标架构生成优化的机器代码

5.9.1 选择代码生成后端

在深入我们的实现之前，我们应该理解为什么我们选择Cranelift 作为我们的代码生成后端。当实现编译器时，你面临关于如何生成可执行代码的基本选择，每个都有不同的权衡。

定制后端：最教育的方法涉及直接为目标架构发出汇编指令。这给你完全的控制和对机器的深入理解，但需要从头实现寄存器分配、指令选择和优化过程。对于针对多种架构的生产编译器，这很快变得难以处理。

虚拟机方法：许多语言选择编译为自定义虚拟机的字节码。几十年来，已经为函数式语言开发了许多专门的VM。这种方法提供了出色的可移植性并简化了编译器，但由于解释开销而牺牲了性能。

LLVM：生产编译器的主要选择，LLVM 提供工业强度的优化过程并支持几乎所有架构。然而，LLVM 的全面性带来了显著的成本：巨大的二进制大小（数百兆字节）、缓慢的编译时间和需要深入专业知识的复杂C++ API。

Cranelift：最初为WebAssembly 设计，Cranelift 在语言实验方面占据了一个最佳位置。它快速生成高质量代码，具有干净的Rust API，产生小二进制文件，并支持主要架构（x86-64、ARM64）。虽然它无法匹配LLVM 的峰值优化质量，但Cranelift 编译代码的速度快一个数量级。对于教学编译器，迭代速度和代码清晰度比挤出最后10% 的性能更重要，Cranelift 很棒。

5.9.2 类型擦除

System Fω 的类型系统在类型检查期间服务于其目的，确保程序正确性并启用强大的抽象。但正如我们之前讨论的，类型纯粹用于编译时验证，在运行时不携带任何计算内容。类型擦除阶段剥离所有类型信息，只留下基本的计算结构。

擦除的表示捕获了没有类型的计算的本质。变量变成简单的名称，lambda 抽象失去其类型注释，应用仍然作为函数调用的基本操作。整数字面量和二元操作保持不变，因为它们表示实际的运行时计算。关键的是，类型抽象和类型应用在擦除期间完全消失，因为它们仅用于指导类型检查。

5.9.3 闭包转换

函数式语言允许在其他函数内定义函数，创建嵌套作用域，其中内部函数可以引用外部作用域中的变量。这种自然的编程风格对编译到机器代码提出了挑战，在机器代码中，函数通常编译到固定地址，没有对其定义环境的隐式访问。

闭包转换通过将每个函数转换为代码指针和包含捕获值的环境的对来解决这个问题。转换通过以下方式使环境捕获显式：

1. 识别每个函数中的自由变量（使用但未在本地定义的变量）
2. 创建环境结构以保存这些捕获的值
3. 重写函数以接受额外参数（闭包）并从其中提取捕获的值
4. 转换函数调用以传递闭包和常规参数

闭包转换后，每个函数作为单独顶级定义存在，具有用于其闭包环境的显式参数。函数主体可以通过从该环境参数投影来访问捕获的变量。这种扁平结构直接映射到汇编语言的函数模型，其中每个函数具有固定地址和显式参数。

5.9.4 Cranelift 代码生成

闭包转换完成后，程序由一阶函数集合和显式闭包操作组成。Cranelift 编译器框架将此表示转换为目标架构的优化机器代码。

代码生成器维护编译所需的状态，包括累积编译函数的Cranelift 模块、用于构建单个函数的函数构建器上下文，以及我们的函数标识符与Cranelift 函数引用之间的映射。运行时函数结构提供对支持内存分配和其他基本操作的运行时系统原语的访问。

每个闭包转换的函数编译为遵循统一调用约定的Cranelift 函数。函数接收两个参数：包含捕获变量的闭包和函数参数。它们使用相同的标记表示返回单个值。

5.9.5 运行时系统

生成的机器代码不能独立存在。它需要一个运行时系统，提供我们的高级函数语言无法表达的基本服务。此运行时弥合了纯函数代码与底层操作系统之间的差距。

运行时通过遵循我们的调用约定和值表示的函数提供几类基本服务。每个运行时函数都声明给Cranelift，可以直接从生成的代码调用。

运行时支持库服务于几个核心目的：

- **内存分配**：我们的函数语言动态创建闭包和其他数据结构，但没有内存管理的概念。运行时提供`rt_alloc`，一个管理固定1MB 堆的简单bump 分配器。
- **值创建**：`make_int` 函数标记原始整数以供在我们的标记表示中使用，而`make_closure` 分配并初始化具有其代码指针和捕获环境的闭包结构。
- **函数应用**：`apply` 函数实现闭包调用的调用约定。它从闭包中提取代码指针，并使用闭包和参数调用它，处理我们调用约定的低级细节。
- **输入/输出**：纯函数语言没有像打印到控制台这样的效果的内在概念。运行时提供`rt_print_int`，它展开我们的标记整数表示并调用C 库的`printf` 函数来显示值。

5.10 示例

我们的System Fω 实现通过一系列工作示例展示了其能力，这些示例展示了类型系统功能的全部范围。这些示例从基本代数数据类型到高阶多态函数，说明了System Fω 如何在保持类型安全的同时启用高级编程模式。

5.10.1 基本数据类型和模式匹配

我们System Fω 实现的基础在于其对具有全面模式匹配的代数数据类型的支持。这些特性为更多编程模式提供了构建块。

```
-- 具有构造函数的代数数据类型
data Bool = True | False;
data Maybe a = Nothing | Just a;
data Either a b = Left a | Right b;
data List a = Nil | Cons a (List a);

-- 具有模式匹配的函数
not :: Bool -> Bool;
not b = match b {
  True -> False;
  False -> True;
};

isJust :: Maybe a -> Bool;
isJust m = match m {
```

```
  Nothing -> False;
  Just x -> True;
};
```

5.10.2 多态函数

System Fω 的全称量化使能够在所有类型上统一工作的函数，展示了参数多态性的实际应用。

```
id :: a -> a;
id x = x;

const :: a -> b -> a;
const x y = x;

map :: (a -> b) -> List a -> List b;
map f lst = match lst {
  Nil -> Nil;
  Cons x xs -> Cons (f x) (map f xs);
};
```

这些函数展示了：

- **类型变量作用域**：像‘a‘ 和‘b‘ 这样的变量被隐式量化
- **主类型**：实现推断最一般可能的类型
- **多态实例化**：每次使用可以不同地实例化类型

5.10.3 复杂多态编程

更多示例显示System Fω 如何处理多态性、高阶函数和代数数据类型之间的复杂交互。

```
foldRight :: (a -> b -> b) -> b -> List a -> b;
foldRight f acc lst = match lst {
  Nil -> acc;
  Cons x xs -> f x (foldRight f acc xs);
};

filter :: (a -> Bool) -> List a -> List a;
filter pred lst = match lst {
  Nil -> Nil;
  Cons x xs -> match pred x {
    True -> Cons x (filter pred xs);
    False -> filter pred xs;
  };
};
```

这些示例展示了类型推断在复杂场景中的工作方式。当处理像‘map (add 1) someList‘ 这样的表达式时，算法：

1. 为未知类型**生成存在变量**
2. 通过函数应用**传播约束**
3. **统一类型**以发现列表必须具有类型‘List Int‘
4. **报告最终类型**为‘List Int‘

Chapter 6

构造演算

6.1 构造演算

构造演算（简称CoC）代表了λ 立方体的巅峰，占据了立方体中最具表达力的角落，在这里，所有三种抽象维度汇聚一堂。该系统将项、类型和种类统一到一个单一的语法框架中，消除了将计算与逻辑人为分隔的界限，并使类型能够表达带有计算内容的任意数学命题。

在我们探索的先前的系统中，项、类型和种类之间维持严格的层次结构，而CoC 则消除了这些区别。类型成为一等公民，可以被操纵、传递给函数并作为结果返回。这种统一带来了前所未有的表达力，同时通过精心构建的宇宙层次结构维持逻辑一致性。

6.1.1 在λ 立方体中的位置

CoC 位于λ 立方体的顶点$\lambda^2\omega P$，结合了定义立方体维度的所有三种抽象形式：

项依赖于类型（↑-轴）：多态函数，如$\mathsf{id} : \forall A : \mathsf{Type}.A \to A$，其中项抽象于类型参数，实现系统中的所有类型的参数多态。

类型依赖于类型（↗-轴）：类型构造器，如$\mathsf{List} : \mathsf{Type} \to \mathsf{Type}$ 和$\Sigma : (A : \mathsf{Type}) \to (A \to \mathsf{Type}) \to \mathsf{Type}$，其中类型可以抽象于其他类型，实现高阶多态和类型级计算。

类型依赖于项（→-轴）：依赖类型，如$\mathsf{Vec} : \mathsf{Nat} \to \mathsf{Type} \to \mathsf{Type}$，其中类型的结构依赖于项的值，实现数据结构属性和程序不变量的精确规范。

这三个维度的汇聚创造了一个前所未有的表达力系统。与提供多态但在项和类型之间维持清晰分离的System Fω 不同，CoC 允许类型依赖于任意项级计算，同时通过归一化属性维持可判定的类型检查。

6.1.2 Curry-Howard 对应

CoC 实现了计算与逻辑之间深刻的联系，即Curry-Howard 对应。在这种对应中，类型代表逻辑命题，项代表这些命题的构造性证明。这种同构使相同的语法框架能够表达程序和带有证明的数学定理。

命题作为类型：每个逻辑语句对应一个类型。命题"对于所有自然数n，n + 0 = n"成为类型$\forall n : \texttt{Nat.Eq Nat (plus n zero) n}$，其中`Eq` 代表命题等价。

证明作为程序：每个构造性证明对应一个计算命题见证的程序。上述命题的证明成为一个接受自然数并产生添加零保持该数的证据的函数。

证明检查作为类型检查：验证数学证明的正确性归结为检查程序具有预期类型。类型检查器成为证明检查器，确保声称的证明实际建立了其声称的命题。

这种对应将编程转化为定理证明，将定理证明转化为编程，创造了一个统一的框架，其中数学严谨性和计算效率自然共存。

6.1.3 依赖类型：基础

使CoC 表达力成为可能的关键创新是**依赖积类型**，或**Π-类型**。这种构造将熟悉的函数箭头泛化，创建了其结构依赖于抽象值的类型。

6.1.3.1 依赖积（Π-类型）

依赖积类型$\Pi x : A.B$ 代表函数，其中返回类型B 可以依赖于输入值x。当变量x 不出现在B 中时，这归结为简单函数类型$A \to B$。当x 出现在B 中时，我们获得真正的依赖，其中返回类型基于输入值而变化。

```
-- 简单函数类型（非依赖）
add : Nat → Nat → Nat

-- 依赖函数类型
vec : (n : Nat) → Type → Type
create_vec : (n : Nat) → (A : Type) → vec n A

-- 返回类型依赖于输入值 n
lookup : (n : Nat) → (A : Type) → (v : vec n A) → (i : Fin n) → A
```

依赖积使类型级规范能够精确捕捉程序不变量，直接在类型系统中。向量查找函数可以在编译时保证索引落在向量界限内，消除运行时界限检查，同时维持类型安全。

6.1.3.2 依赖和（Σ-类型）

依赖和类型$\Sigma x : A.B$ 代表对，其中第二组件的类型依赖于第一组件的值。这使存在量化和带有精确类型关系的异构数据结构创建成为可能。

```
-- 依赖对：一个数字和该长度向量
sized_vec : Type := Σ n : Nat. vec n Int

-- 创建一个有尺寸向量
example_vec : sized_vec := □3, [1, 2, 3]□

-- 模式匹配以提取组件
process_vec : sized_vec → Int :=
  fun □n, v□ => sum_vector v  -- 类型检查器知道 v 的长度为 n
```

依赖和使存在命题的表达成为可能，其中我们断言具有特定属性的值的存在，而不揭示确切值，同时维持计算使用该值的能力。

6.1.4 宇宙层次结构和逻辑一致性

依赖类型的强大引发了关于自引用和逻辑一致性的基本问题。如果类型可以包含任意值，并且类型本身是值，什么防止了像"所有不包含自身的集合的集合"这样的悖论类型的构造？

CoC 通过**宇宙层次结构**解决这一挑战，该结构按复杂性级别分层类型。每个宇宙包含有界复杂性的类型，宇宙层次结构防止了导致逻辑不一致的自引用类型的构造。

6.1.4.1 宇宙级别

```
-- 宇宙层次结构
Prop : Type          -- 无计算内容的命题
Type : Type 1        -- 小类型（Nat, Bool 等）
Type 1 : Type 2      -- 类型构造器的类型
Type 2 : Type 3      -- 高阶类型构造器
-- ... 无限层次结构
```

Prop：命题宇宙代表逻辑语句，当被证明时，除了其真值之外不携带计算信息。证明无关性意味着同一命题的所有证明被视为相等，实现高效编译，其中证明项可以被擦除。

Type n：宇宙层次结构Type 0，Type 1，Type 2，... 代表在增加抽象级别上的计算类型。像Nat 和Bool 这样的类型居住在Type 0，而像List : Type 0 →Type 0 这样的类型构造器居住在Type 1。

宇宙多态：定义可以多态于宇宙级别，实现跨越整个层次结构的泛型构造。恒等函数可以定义一次，并在任何宇宙级别的类型上工作。

宇宙层次结构为计算类型维持**预测性**，意味着级别n 的类型构造器只能量化严格小于n 的级别的类型。这种限制防止了大消除悖论的构造，同时维持逻辑一致性。

然而，命题宇宙Prop 是**非预测的**，允许量化任意类型，包括命题本身。这种非预测性使强大逻辑原则的表达成为可能，同时通过证明无关性维持一致性。

6.1.5 实现架构

我们的构造演算实现演示了这些理论概念如何转化为实际的类型检查算法和编程语言特性。

```
#![allow(unused)]
fn main() {
use std::fmt;
/// Universe levels for type theory
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Universe {
    Const(u32),                          // Concrete level: 0, 1, 2, ...
    ScopedVar(String, String),           // Scoped universe variable: (scope_name, var_name)
    Meta(u32),                           // Universe metavariable: ?u, ?u, ...
    Add(Box<Universe>, u32),             // Level + n
    Max(Box<Universe>, Box<Universe>),  // max(u, v)
    IMax(Box<Universe>, Box<Universe>), // imax(u, v)
}
/// Universe level constraints for polymorphism
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum UniverseConstraint {
    /// u ≤ v
    LessEq(Universe, Universe),
    /// u = v
    Equal(Universe, Universe),
}
/// Context for universe level variables
#[derive(Debug, Clone, PartialEq)]
pub struct UniverseContext {
    /// Currently bound universe variables
    pub variables: Vec<String>,
    /// Constraints on universe levels
    pub constraints: Vec<UniverseConstraint>,
}
/// Pattern matching arms
#[derive(Debug, Clone, PartialEq)]
pub struct MatchArm {
    pub pattern: Pattern,
    pub body: Term,
}
/// Patterns for match expressions
#[derive(Debug, Clone, PartialEq)]
pub enum Pattern {
    /// Variable pattern: x
    Var(String),
    /// Constructor pattern: Cons x xs
    Constructor(String, Vec<Pattern>),
    /// Wildcard pattern: _
    Wildcard,
}
/// Top-level declarations
#[derive(Debug, Clone, PartialEq)]
pub enum Declaration {
    /// Constant definition: def name {u...} (params) : type := body
    Definition {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        params: Vec<Parameter>,
        ty: Term,
        body: Term,
    },
    /// Axiom: axiom name {u...} : type
    Axiom {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        ty: Term,
    },
    /// Inductive type: inductive Name {u...} (params) : Type := constructors
```

```
    Inductive {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        params: Vec<Parameter>,
        ty: Term,
        constructors: Vec<Constructor>,
    },
    /// Structure (single constructor inductive): structure Name {u...} :=
    /// (fields)
    Structure {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        params: Vec<Parameter>,
        ty: Term,
        fields: Vec<Field>,
    },
}
/// Function parameters
#[derive(Debug, Clone, PartialEq)]
pub struct Parameter {
    pub name: String,
    pub ty: Term,
    pub implicit: bool, // for {x : A} vs (x : A)
}
/// Constructor for inductive types
#[derive(Debug, Clone, PartialEq)]
pub struct Constructor {
    pub name: String,
    pub ty: Term,
}
/// Structure fields
#[derive(Debug, Clone, PartialEq)]
pub struct Field {
    pub name: String,
    pub ty: Term,
}
/// Module containing multiple declarations
#[derive(Debug, Clone, PartialEq)]
pub struct Module {
    pub declarations: Vec<Declaration>,
}
```

项语言将所有语法类别统一到一个单一框架中，其中相同的构造根据上下文服务于多个角色。Lambda 抽象既创建函数又创建证明项，应用既表示函数调用又表示肯定前件，积既表示函数类型又表示全称量化。

与之前一样，我们的实现使用**双向类型检查**，将类型检查分为互补的综合和检查模式。

1. **综合模式：**给定一个项，通过分析项的结构并通过语法树传播类型信息来确定其类型。
2. **检查模式：**给定一个项和预期类型，通过检查定义相等性模的兼容性来验证项是否属于预期类型。

6.1.5.1 基于约束的推断

系统包括用于隐式参数推断和依赖类型统一的先进约束求解。元变量表示未知类型，通过与复杂依赖跟踪的统一来解决。

约束求解器处理高阶统一模式、宇宙级别约束和延迟约束解析，以在保持理论正确性的同时实现依赖类型的实际编程。

6.1.6 理论意义

CoC 不仅仅是System-F 的另一个扩展；它具有深远的理论意义。它本质上展示了计算与数学之间的基本统一。这是理论计算机科学中最深刻的思想之一。

通过展示每个构造性数学证明对应一个程序，每个程序根据逻辑原则进行类型检查，CoC 在形式验证和实际编程之间架起了桥梁。这非常了不起!

该系统还为交互式定理证明提供了基础，其中数学证明通过编程构建并通过类型检查验证。像Coq 和Lean 这样的证明助手建立在CoC 建立的理论基础之上，展示了这种统一的实际价值。

虽然Coq 在历史上很有趣，但我们将主要受到Lean 4 模型的启发，并在我们的玩具实现中采用其类型系统和语法的一个小版本。

6.2 依赖类型

依赖类型是类型理论中最深刻的发展之一，通过允许类型依赖它们所分类的值，从根本上改变了计算与逻辑之间的关系。这种能力将类型从静态标签转变为动态规范，能够直接在类型系统中表达精确的数学属性和程序不变量。

从简单类型到依赖类型的演进，类似于从基本算术到高级数学的演进。正如数学从计数离散对象发展到表达抽象结构之间的关系，类型系统也从分类基本值演变为编码复杂的逻辑命题和计算规范。

6.2.1 简单类型的局限性

传统类型系统，即使是像System Fω 这样的系统，也在项的计算世界和类型的分类世界之间保持着根本的分离。类型作为静态标签，根据值的结构属性对值进行分组，从而实现编译时安全检查和优化机会。

```
-- 简单类型静态分类值
length : List a -> Int
head : List a -> a  -- 部分函数 - 如果列表为空怎么办?
```

这种分离导致了程序员对数据的了解与类型系统所能表达的内容之间的差距。程序员可能知道某个特定列表是非空的，但类型系统无法捕捉这一知识，迫使进行运行时检查，而这些检查理论上可以被消除。

简单类型擅长防止基本错误，如将函数应用于不兼容类型的参数，但它们无法表达值之间的关系或捕捉特定领域的不变量。结果是在类型安全与表达能力之间的紧张关系，而依赖类型通过消除项与类型之间的人为界限来解决这一问题。

6.2.2 依赖类型的革命

依赖类型通过允许类型依赖计算值，消除了计算与分类之间的分离。这种依赖使得类型能够表达关于它们所分类值的任意精确规范，将类型系统转变为能够表达数学定理的规范语言。

6.2.2.1 依赖函数（Π-类型）

依赖积类型$\Pi x : A.B$ 通过允许结果类型B 依赖于输入值x，推广了熟悉的函数箭头$A \to B$。这种依赖使得函数类型不仅指定输入和输出的结构，还指定它们之间的精确关系。

```
-- 非依赖函数类型
length : List A → Nat

-- 依赖函数类型
vec : (n : Nat) → Type → Type
create_vec : (n : Nat) → (A : Type) → vec n A

-- 返回类型依赖于输入值
safe_head : (n : Nat) → (A : Type) → (v : vec (n + 1) A) → A
```

`safe_head` 函数展示了依赖类型的强大功能：通过要求向量长度为$n+1$ 而不是任意自然数，类型系统保证向量非空，从而消除了提取头部元素时发生运行时错误的可能。

6.2.2.2 依赖对（Σ-类型）

依赖和类型$\Sigma x : A.B$ 表示第二组件的类型依赖于第一组件的值。这种结构支持存在量化，并创建保持组件之间精确关系的复杂数据结构。

```
-- 简单对类型
Pair A B : Type := A × B

-- 依赖对类型
DPair A B : Type := Σ x : A. B x

-- 具体示例：带长度的向量
sized_vec : Type := Σ n : Nat. vec n Int

-- 模式匹配保留依赖关系
process_sized : sized_vec → Int
process_sized □n, v□ = sum_vec v  -- 类型检查器知道 v 的长度为 n
```

依赖对能够在类型系统中表达存在性语句。我们不再断言"存在自然数n 使得性质P 成立"，而是构建一个具体的见证，既展示存在性，又提供对见证和性质证明的计算访问。

6.2.3 依赖类型作为规范

依赖类型的真正力量体现在我们认识到它们作为可执行规范时。与用独立规范语言编写的传统规范不同，依赖类型集成在编程语言本身中，使得规范能够被类型系统自动检查。

6.2.3.1 精确的数组边界

传统的数组操作需要运行时边界检查以确保内存安全。依赖类型实现了数组边界的编译时验证，消除了运行时开销和边界违规的可能性。

```
-- 按长度索引的数组类型
Array : Nat → Type → Type

-- 边界安全的数组索引
get : (n : Nat) → (A : Type) → Array n A → (i : Nat) → i < n → A

-- 使用时需要证明索引在范围内
example_access : Array 5 Int → Int
example_access arr = get 5 Int arr 2 (by norm_num)  -- 证明 2 < 5
```

类型系统现在捕捉了数组大小与有效索引之间的关系，将运行时安全属性转变为编译时保证。那些会导致数组边界违规的程序在语法上变得不合法，防止了一整类常见的编程错误。

6.2.3.2 正确性条件

依赖类型可以表达确保算法满足预期属性的正确性条件。排序函数可以被指定为产生既是输入的排列又是已排序的列表。

```
-- 已排序列表的规范
Sorted : List Nat → Prop

-- 排列的规范
Permutation : List A → List A → Prop

-- 正确排序函数的类型
sort : (l : List Nat) → {l' : List Nat // Sorted l' ∧ Permutation l l'}
```

这种规范将"正确排序"的非正式概念转变为类型系统可以验证的精确数学陈述。未能保持所需属性的实现将在编译时被拒绝，提供强大的正确性保证。

6.2.4 实践中的Curry-Howard 对应

依赖类型在实际编程环境中实现了Curry-Howard 对应。对应建立了逻辑命题与类型、数学证明与程序、证明验证与类型检查之间的深刻联系。

6.2.4.1 命题即类型

每个数学命题在依赖类型系统中对应一个类型。例如，"对所有自然数n，n + 0 = n"成为类型$\forall n : \texttt{Nat}, n + 0 = n$，其中等式被表示为一个类型族，仅在其参数相等时被填充。

```
-- 数学命题作为类型
zero_right_identity : n : Nat, n + 0 = n

-- 构造性证明作为程序
zero_right_identity = fun n =>
  Nat.rec
    (Eq.refl 0)                    -- 基情形: 0 + 0 = 0
    (fun k ih => congrArg succ ih) -- 归纳步骤: (k+1) + 0 = k+1
```

证明项通过提供计算见证来展示命题。类型检查器验证该见证是否真正建立了所声称的命题，确保只接受有效证明。

6.2.4.2 程序即证明

反过来，每个构造性证明对应一个计算证明命题证据的程序。复杂的数学定理成为通过计算构建见证的程序。

```
-- 定理: 每个列表都有可决定的相等性测试
list_eq_decidable : (A : Type) → [DecidableEq A] → DecidableEq (List A)
list_eq_decidable A inst =
  -- 为列表构建可决定的相等性过程
  -- 这既是可决定性的证明，也是测试相等性的算法
```

该程序具有双重角色：它提供了可计算执行的算法内容，同时作为建立数学属性的证明。这种二元性消除了规范与实现之间的差距。

6.2.5 依赖类型中的挑战与解决方案

依赖类型的表达能力引入了新的挑战，需要解决方案。计算与规范的整合创造了更简单的类型系统所避免的复杂性。

6.2.5.1 定义相等性

在依赖类型系统中，类型检查需要判断两个类型何时相等。然而，由于类型可能包含计算内容，类型相等性变成了计算等价性，通常是不可判定的。

实际的依赖类型系统通过将**定义相等性**定义为在某些计算规则下的相等性来解决这一挑战。如果两个类型在β-归约、η-扩展和其他定义归约下归约到相同形式，则它们被认为是定义相等。

```
-- 这些类型是定义相等的
Vector (2 + 3) Int  ≡  Vector 5 Int

-- 因为 (2 + 3) 归约到 5
```

这种方法通过将定义相等性限制在归约计算上，保持了可判定性，同时为实际编程模式提供了足够的灵活性。

6.2.5.2 宇宙层次

构造依赖于任意值的类型的能力引发了逻辑一致性问题。如果类型可以包含值，值可以是类型，那么什么阻止了构建自指的悖论类型？

依赖类型系统通过**宇宙分层**保持一致性，即将类型组织成具有严格包含关系的宇宙层次。每个宇宙包含复杂度受限的类型，防止构建会导致逻辑悖论的非预测类型。

数学表述 宇宙层次可以精确表述为一个具有包含关系的无限宇宙序列：

$$\mathcal{U}_0 \subseteq \mathcal{U}_1 \subseteq \mathcal{U}_2 \subseteq \cdots \subseteq \mathcal{U}_\omega$$

每个宇宙$\mathcal{U}_i$ 作为第i 层类型的域，而Type_i 表示$\mathcal{U}_i$ 中类型的类型。基本类型化规则建立了层次：

$$\frac{A : \text{Type}_i}{\text{Type}_i : \text{Type}_{i+1}} \quad \text{(Universe Formation)}$$

$$\frac{A : \text{Type}_i \quad i \leq j}{A : \text{Type}_j} \quad \text{(Cumulativity)}$$

累积性规则允许类型提升到更高宇宙，提供灵活性，同时保持一致性所需的严格分层。

类型形成规则 宇宙层次规定了每个层级如何形成类型。基本类型位于宇宙$\mathcal{U}_0$：

$$\text{Nat} : \text{Type}_0 \qquad \text{Bool} : \text{Type}_0$$

类型构造子必须尊重宇宙层级。对于依赖函数类型，结果宇宙是参数和结果宇宙的最大值：

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_j}{\Gamma \vdash \Pi x : A.B : \text{Type}_{\max(i,j)}} \quad \text{(Pi Formation)}$$

对于归纳类型，宇宙层级由构造子的参数类型决定：

$$\frac{\text{每个构造子}c_k\text{ 的类型为}\Pi\vec{x} : \vec{A}.T \quad \max(\text{levels}(\vec{A})) \leq i}{\text{归纳类型}T : \text{Type}_i} \quad \text{(Inductive Formation)}$$

预测性和一致性 宇宙层次确保**预测性**，即第i 层的类型构造子只能量化第i 层以下的类型。这种限制防止了罗素式的悖论：

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_j \quad j < i}{\Gamma \vdash \Pi x : A.B : \text{Type}_i} \quad \text{(Predicative Pi)}$$

没有这一限制，我们可以构造所有不包含自身的类型的类型，导致矛盾。预测性约束确保类型系统中不可能出现自指。

宇宙多态性 在我们的实现中，我们支持**宇宙多态性**，允许定义抽象过宇宙层级：

$$\mathrm{id} : \Pi u : \mathrm{Level}.\Pi A : \mathrm{Type}_u.A \to A$$

宇宙层级变量支持跨整个宇宙层次的泛型编程：

$$\frac{\Gamma \vdash e : \Pi u : \mathrm{Level}.T \quad \ell : \mathrm{Level}}{\Gamma \vdash e[\ell] : T[u := \ell]} \quad \text{(Universe Application)}$$

宇宙算术 宇宙多态性通常需要对宇宙层级进行算术运算。我们支持以下运算：

- **后继**：$u + 1$ 表示紧接在u 上方的宇宙
- **最大值**：$\max(u, v)$ 表示包含u 和v 的最小宇宙
- **加法**：$u + n$ 表示在u 上方n 层的宇宙

具有以下性质：

$$\max(u, v) = \max(v, u) \quad \text{(Symmetry)}$$

$$\max(u, \max(v, w)) = \max(\max(u, v), w) \quad \text{(Associativity)}$$

$$(u + m) + n = u + (m + n) \quad \text{(Addition Associativity)}$$

一致性与归一化 宇宙层次确保依赖类型系统的几个关键属性：

- **强归一化**：每个类型良好的项具有有限的法式，确保类型检查终止。
- **逻辑一致性**：系统不接受false 的证明，保持其作为逻辑基础的效用。
- **可判定的类型检查**：强归一化与定义相等性的结合使类型检查可判定。

宇宙的正式处理需要仔细关注：

1. **宇宙层级推断**：自动确定多态定义的适当宇宙层级
2. **约束求解**：解决类型检查期间出现的宇宙层级约束系统
3. **累积子类型化**：高效实现类型到更高宇宙的强制转换

在构造演算中的实现 我们采用以下命名约定用于类型宇宙：

- **Prop** - 命题的宇宙，包含可能具有或不具有计算内容的逻辑语句
- **Type** - 类型宇宙（等价于Type 0），包含普通数据类型如自然数和列表
- **Type u** - 位于层级u 的宇宙，其中u 是宇宙层级变量

- **Sort u** - 位于层级u 的通用宇宙，可以表示Prop（当$u = 0$）或Type u（当$u > 0$）

关键原则是位于层级u 的宇宙只能分类严格低于u 的级别的类型。这创建了一个层次结构，防止逻辑悖论，同时实现表达性的类型级编程。

```
-- 基本类型位于 Type (= Type 0)
Nat : Type
Bool : Type
String : Type

-- 类型构造器尊重宇宙层级
List : Type → Type
Option : Type → Type

-- 宇宙层次结构
Type : Type 1
Type 1 : Type 2
Type 2 : Type 3
-- ... 无限层次结构

-- 宇宙多态定义
def id.{u} (A : Type u) (x : A) : A := x

def compose.{u,v,w} (A : Type u) (B : Type v) (C : Type w)
  (g : B → C) (f : A → B) (x : A) : C := g (f x)

-- 命题位于 Prop
inductive Eq.{u} {A : Sort u} (a : A) : A → Prop where
  | refl : Eq a a

-- Sort 统一 Prop 和 Type
def identity.{u} {A : Sort u} (x : A) : A := x
```

6.2.6 类型推断和细化

依赖类型的表达能力为类型推断创造了挑战，因为系统必须经常推断不仅是类型，还有证明项和计算内容。现代依赖类型系统采用**细化**过程，自动插入隐式参数、解析类型类实例并构建证明项。

```
-- 带隐式参数的表面语法
map : {A B : Type} → (A → B) → List A → List B

-- 带显式参数的细化形式
map : (A : Type) → (B : Type) → (A → B) → List A → List B

-- 带自动细化的使用
result = map succ [1, 2, 3]  -- 细化为: map Nat Nat succ [1, 2, 3]
```

细化过程在程序员想要编写的简洁语法和类型检查器需要用于验证的完全显式形式之间架起了桥梁。

6.3 类型系统

我们的构造演算实现围绕一个类型系统展开，该系统将项、类型和种类统一到一个单一的语法框架中。该实现演示了依赖类型、宇宙多态和定义相等性如何协同工作，创建一个实用的依赖类型编程语言。

6.3.1 AST 设计和项语言

我们实现的核心在于统一的项语言，它在单个AST 结构中表示所有语法类别。这种设计反映了CoC 原则，即项、类型和种类都是同一计算宇宙的居民。

```
#![allow(unused)]
fn main() {
use std::fmt;
/// Core terms in the Calculus of Constructions
#[derive(Debug, Clone, PartialEq)]
pub enum Term {
    /// Variables: x, y, z
    Var(String),
    /// Application: f a
    App(Box<Term>, Box<Term>),
    /// Lambda abstraction: λx : A, t (written as fun x : A => t)
    Abs(String, Box<Term>, Box<Term>),
    /// Dependent product: Πx : A, B (written as (x : A) → B or {x : A} → B)
```

```
    Pi(String, Box<Term>, Box<Term>, bool), // bool = implicit
    /// Sort/Type: Sort u, Type, Prop
    Sort(Universe),
    /// Let binding: let x := t in s
    Let(String, Box<Term>, Box<Term>, Box<Term>),
    /// Match expression with patterns
    Match(Box<Term>, Vec<MatchArm>),
    /// Inductive type constructor
    Constructor(String, Vec<Term>),
    /// Local constant (for definitions and axioms)
    Const(String),
    /// Meta-variable for type inference
    Meta(String),
    /// Field projection: term.field
    Proj(Box<Term>, String),
    /// Dependent pair type (Sigma type): Σ(x : A), B
    Sigma(String, Box<Term>, Box<Term>),
    /// Pair constructor: □a, b□
    Pair(Box<Term>, Box<Term>),
    /// First projection: π
    Fst(Box<Term>),
    /// Second projection: π
    Snd(Box<Term>),
}
```

我们的Term 枚举通过使用相同的构造器来表示函数（Abs）、函数类型（Pi）、类型构造器和逻辑命题，展示了统一原则。相同的应用构造器（App）既表示函数应用又表示类型应用，消除了分层类型系统中存在的人为界限。

变量（Var）和常量（Const）构成了我们项语言的基本构建块。变量既表示项级绑定又表示类型级绑定，而常量引用全局上下文中的已定义名称。该实现统一处理这两个类别，使相同的绑定机制在所有抽象级别上工作。

6.3.1.1 函数类型和Lambda 抽象

Pi 构造器表示依赖积类型，推广了简单函数类型和全称量化。当绑定变量出现在体类型中时，我们获得依赖；当它不出现时，Pi 类型归结为简单函数类型。

```
// Pi(variable_name, domain_type, codomain_type, is_implicit)
Pi("x".to_string(), Box::new(nat_type), Box::new(vec_type), false)
```

布尔标志指示参数是否应被视为隐式，使我们的实现能够支持显式和隐式参数传递。Lambda 抽象（Abs）创建Pi 类型的居民，相同的构造器既用于计算函数又用于证明项。

6.3.1.2 模式匹配和归纳消除

```
#![allow(unused)]
fn main() {
use std::fmt;
/// Core terms in the Calculus of Constructions
#[derive(Debug, Clone, PartialEq)]
pub enum Term {
    /// Variables: x, y, z
    Var(String),
    /// Application: f a
    App(Box<Term>, Box<Term>),
    /// Lambda abstraction: λx : A, t (written as fun x : A => t)
    Abs(String, Box<Term>, Box<Term>),
    /// Dependent product: Πx : A, B (written as (x : A) → B or {x : A} → B)
    Pi(String, Box<Term>, Box<Term>, bool), // bool = implicit
    /// Sort/Type: Sort u, Type, Prop
    Sort(Universe),
    /// Let binding: let x := t in s
    Let(String, Box<Term>, Box<Term>, Box<Term>),
    /// Match expression with patterns
    Match(Box<Term>, Vec<MatchArm>),
    /// Inductive type constructor
    Constructor(String, Vec<Term>),
    /// Local constant (for definitions and axioms)
    Const(String),
    /// Meta-variable for type inference
    Meta(String),
    /// Field projection: term.field
    Proj(Box<Term>, String),
    /// Dependent pair type (Sigma type): Σ(x : A), B
    Sigma(String, Box<Term>, Box<Term>),
    /// Pair constructor: □a, b□
    Pair(Box<Term>, Box<Term>),
```

```
    /// First projection: π
    Fst(Box<Term>),
    /// Second projection: π
    Snd(Box<Term>),
}
/// Pattern matching arms
#[derive(Debug, Clone, PartialEq)]
pub struct MatchArm {
    pub pattern: Pattern,
    pub body: Term,
}
```

我们的实现包括通过Match 构造器进行的全面模式匹配，它实现了归纳类型的消除。每个匹配分支指定一个构造器模式和相应的消除项，提供归纳推理所需的计算内容。

模式匹配实现支持嵌套模式和变量绑定，能够在保持通过穷尽性检查的类型安全的同时，解构复杂的归纳数据结构。

6.3.2 宇宙系统实现

该实现包括一个全面的宇宙系统，在防止逻辑悖论的同时实现灵活的类型级计算。我们的宇宙设计既支持具体级别又支持多态宇宙变量。

```
#![allow(unused)]
fn main() {
/// Universe levels for type theory
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Universe {
    Const(u32),                         // Concrete level: 0, 1, 2, ...
    ScopedVar(String, String),          // Scoped universe variable: (scope_name, var_name)
    Meta(u32),                          // Universe metavariable: ?u, ?u, ...
    Add(Box<Universe>, u32),            // Level + n
    Max(Box<Universe>, Box<Universe>),  // max(u, v)
    IMax(Box<Universe>, Box<Universe>), // imax(u, v)
}
/// Universe level constraints for polymorphism
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum UniverseConstraint {
    /// u ⩽ v
    LessEq(Universe, Universe),
    /// u = v
    Equal(Universe, Universe),
}
/// Context for universe level variables
#[derive(Debug, Clone, PartialEq)]
pub struct UniverseContext {
    /// Currently bound universe variables
    pub variables: Vec<String>,
    /// Constraints on universe levels
    pub constraints: Vec<UniverseConstraint>,
}
}
```

6.3.2.1 宇宙级别和算术

宇宙系统支持多种形式的宇宙表达式，既支持具体级别规范又支持多态宇宙计算。由Const(n) 表示的具体级别指定特定的宇宙级别，如Type 0、Type 1 等，形成宇宙层次结构的基础，为特定类型提供确定的居所。通过ScopedVar 的宇宙变量使宇宙多态成为可能，允许定义在宇宙级别上参数化，使同一定义能够在多个宇宙级别同时工作。使用Add 的级别算术运算支持宇宙级别计算，如$u+1$，支持常见模式，其中类型构造器必须位于比其参数高一级的宇宙级别。由Max 和IMax 提供的最大值运算计算选择两个级别中较高的宇宙级别选择，其中IMax 实现用于证明相关上下文的非预测最大值，其中宇宙级别计算必须尊重证明的逻辑结构。

6.3.2.2 宇宙约束求解

```
#![allow(unused)]
fn main() {
/// Universe level constraints for polymorphism
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum UniverseConstraint {
    /// u ⩽ v
```

```
    LessEq(Universe, Universe),
    /// u = v
    Equal(Universe, Universe),
}
}
```

约束系统处理两种基本关系：

相等性约束（`Equal`）要求两个宇宙级别相同，源于依赖上下文中类型相等性要求，其中宇宙级别必须完全匹配。

排序约束（`LessEq`）确保一个宇宙级别小于或等于另一个，维持防止逻辑悖论的预测性要求，通过维持类型宇宙的严格层次结构。

6.3.3 定义相等性和归一化

我们的类型检查器通过处理β-归约、η-扩展和常量定义的定义展开的全面归一化算法实现定义相等性。

```
/// Normalize a term by reducing redexes (beta reduction, eta-conversion,
/// let expansion)
pub fn normalize(&self, term: &Term, ctx: &Context) -> Term {
    match term {
        Term::App(f, arg) => {
            let f_norm = self.normalize(f, ctx);
            let arg_norm = self.normalize(arg, ctx);
            // Beta reduction: (λx.t) s ~> t[s/x]
            if let Term::Abs(x, _ty, body) = &f_norm {
                self.substitute(x, &arg_norm, body)
            } else {
                Term::App(Box::new(f_norm), Box::new(arg_norm))
            }
        }
        Term::Abs(x, ty, body) => {
            let ty_norm = self.normalize(ty, ctx);
            let extended_ctx = ctx.extend(x.clone(), ty_norm.clone());
            let body_norm = self.normalize(body, &extended_ctx);
            // Eta-conversion: λx. f x ≡ f when x is not free in f
            if let Term::App(f, arg) = &body_norm {
                if let Term::Var(arg_var) = arg.as_ref() {
                    if arg_var == x && !Term::occurs_free(x, f) {
                        // Apply eta-conversion: λx. f x ~> f
                        return f.as_ref().clone();
                    }
                }
            }
            Term::Abs(x.clone(), Box::new(ty_norm), Box::new(body_norm))
        }
        Term::Pi(x, ty, body, implicit) => {
            let ty_norm = self.normalize(ty, ctx);
            let extended_ctx = ctx.extend(x.clone(), ty_norm.clone());
            let body_norm = self.normalize(body, &extended_ctx);
            Term::Pi(x.clone(), Box::new(ty_norm), Box::new(body_norm), *implicit)
        }
        Term::Let(x, _ty, val, body) => {
            // Let reduction: let x : T := v in b ~> b[v/x]
            let val_norm = self.normalize(val, ctx);
            self.substitute(x, &val_norm, body)
        }
        Term::Var(x) => {
            // Look up definitions and unfold them
            if let Some(def) = ctx.lookup_axiom(x) {
                def.clone() // Could be further normalized if it's a
                // definition
            } else {
                term.clone()
            }
        }
        _ => term.clone(), // Other terms are already in normal form
    }
}
```

归一化算法实现函数应用的β-归约，处理计算归约和类型级计算。当lambda抽象应用于参数时，实现执行替换，同时仔细管理变量捕获和作用域。

$$(\lambda x : \texttt{Nat}.x + 1)\ 5 \Longrightarrow 5 + 1 \Longrightarrow 6$$

该实现将β-归约扩展到处理依赖类型计算，其中类型级函数可以应用于通过计算产生新类型。

6.3.3.1 Eta 转换

Eta 转换确保函数等于其eta 展开形式，为函数类型提供外延相等性。该实现在校准期间应用η-展开，确保定义相等的项被识别为相同。

$$\lambda x : A.f\ x \equiv f \quad (\text{当} x \notin \text{fv}(f))$$

6.3.3.2 Let 展开和定义展开

该实现通过展开处理`let` 绑定，在校准期间用其定义替换let 绑定变量。这种方法确保局部定义不会干扰定义相等性，同时提供类型检查所需的计算内容。

定义展开使类型检查器能够访问已定义常量的计算内容，使定义相等性跨模块边界工作，并实现强大的抽象机制。

6.3.4 类型检查算法

我们的实现采用双向类型检查，将类型检查问题分为综合和检查模式。这种方法处理依赖类型的复杂性，同时保持可判定性并提供信息丰富的错误消息。

```
/// Infer the type of a term
pub fn infer(&mut self, term: &Term, ctx: &Context) -> TypeResult<Term> {
    self.with_context(term, |checker| checker.infer_impl(term, ctx))
}
```

6.3.4.1 类型综合

综合算法通过分析项的结构并通过语法树传播类型信息来确定项的类型。变量查找通过查阅类型上下文进行操作，该上下文为项变量和类型变量维护绑定，确保每个变量引用对应于当前作用域中的有效绑定。函数应用类型化通过综合函数类型进行，确保它形成Pi 类型，并检查参数类型匹配域规范，仔细处理依赖类型，其中共域可能依赖于特定参数值。Pi 类型形成验证确保域和共域都是类型良好的，共域在扩展了绑定变量的上下文中进行检查，以正确处理域和共域类型之间的依赖关系。

6.3.4.2 类型检查模式

检查算法通过将项结构与类型结构进行比较来验证项是否具有预期类型。这种模式通常为具有复杂依赖类型的项提供更好的错误消息和更高效的检查。Lambda 检查通过验证lambda 抽象是否与Pi 类型匹配进行操作，确保主体在扩展上下文中具有正确类型，其中lambda 参数被正确绑定。当综合和检查产生不同结果时，算法回退到定义相等性检查，将综合和预期类型都归一化为它们的规范形式并比较结果，使类型检查器能够识别定义相等的项，尽管它们具有不同的语法表示。

6.3.4.3 上下文管理

```
use std::collections::HashMap;
use crate::ast::{Term, Universe, UniverseConstraint, UniverseContext};
/// Definition with universe parameters
#[derive(Debug, Clone, PartialEq)]
pub struct Definition {
    /// Universe parameters for the definition
    pub universe_params: Vec<String>,
```

```
    /// The type of the definition
    pub ty: Term,
    /// Name of the definition (for scoping universe parameters)
    pub name: String,
}
impl Context {
    pub fn new() -> Self {
        Context {
            bindings: HashMap::new(),
            universe_vars: HashMap::new(),
            constructors: HashMap::new(),
            axioms: HashMap::new(),
            definitions: HashMap::new(),
            universe_context: UniverseContext::new(),
        }
    }
    /// Extend context with variable binding
    pub fn extend(&self, var: String, ty: Term) -> Self {
        let mut new_ctx = self.clone();
        new_ctx.bindings.insert(var, ty);
        new_ctx
    }
    /// Look up variable type
    pub fn lookup(&self, var: &str) -> Option<&Term> {
        self.bindings.get(var)
    }
    /// Look up constructor type
    pub fn lookup_constructor(&self, name: &str) -> Option<&Term> {
        self.constructors.get(name)
    }
    /// Look up axiom type
    pub fn lookup_axiom(&self, name: &str) -> Option<&Term> {
        self.axioms.get(name)
    }
    /// Add constructor definition
    pub fn add_constructor(&mut self, name: String, ty: Term) {
        self.constructors.insert(name, ty);
    }
    /// Add axiom definition
    pub fn add_axiom(&mut self, name: String, ty: Term) {
        self.axioms.insert(name, ty);
    }
    /// Add definition with universe parameters
    pub fn add_definition(&mut self, name: String, universe_params: Vec<String>, ty: Term) {
        self.definitions.insert(
            name.clone(),
            Definition {
                universe_params,
                ty,
                name: name.clone(),
            },
        );
    }
    /// Look up definition with universe parameters
    pub fn lookup_definition(&self, name: &str) -> Option<&Definition> {
        self.definitions.get(name)
    }
}
/// Type checking context containing variable bindings
#[derive(Debug, Clone, PartialEq)]
pub struct Context {
    /// Variable to type bindings
    bindings: HashMap<String, Term>,
    /// Universe variable constraints
    universe_vars: HashMap<String, Universe>,
    /// Constructor definitions
    constructors: HashMap<String, Term>,
    /// Axiom definitions (without universe params - for backwards compat)
    axioms: HashMap<String, Term>,
    /// Definitions with universe parameters
    definitions: HashMap<String, Definition>,
    /// Universe level context for polymorphism
    universe_context: UniverseContext,
}
```

类型上下文为变量、定义、公理和构造器维护绑定。我们的实现使用一个上下文结构，支持高效查找，同时支持依赖类型所需的复杂作用域规则。变量绑定操作添加带有其类型的新变量绑定，在lambda 抽象和Pi 类型中实现适当的作用域，其中绑定变量可能出现在后续绑定的类型中。定义扩展功能允许添加带有其类型和主体的新常量定义，支持模块化开发和抽象机制，支持大规模程序组织。构造器注册功能注册带有其类型的归纳类型构造器，实现模式匹配和归纳推理，尊重数据类型的结构属性，并在整个消除操作中维持类型安全。

6.3.5 隐式参数和细化

我们的实现包括对由类型检查器自动推断的隐式参数的全面支持。此功能在完全显式的内部表示和更便捷的表面语法之间架起了桥梁。

```
/// Elaborate implicit parameters by trying to instantiate them to match
/// expected type This is a simple version that handles common cases
```

```
/// without full constraint solving
fn elaborate_implicit_parameters(
    &mut self,
    inferred_ty: &Term,
    expected_ty: &Term,
    ctx: &Context,
) -> TypeResult<Term> {
    // If inferred type has implicit parameters, try to instantiate them
    match inferred_ty {
        Term::Pi(param_name, param_ty, body_ty, true) => {
            // This is an implicit parameter - try to infer what it should be
            let implicit_arg = self.infer_implicit_argument(param_ty, expected_ty, ctx)?;
            // Substitute the implicit argument and continue elaboration
            let instantiated_ty = self.substitute(param_name, &implicit_arg, body_ty);
            self.elaborate_implicit_parameters(&instantiated_ty, expected_ty, ctx)
        }
        _ => {
            // No more implicit parameters, return the type as-is
            Ok(inferred_ty.clone())
        }
    }
}
```

6.3.5.1 隐式参数插入

当遇到函数应用时，类型检查器自动插入隐式参数，其中函数类型具有隐式参数。元变量生成创建新的元变量来表示未知的隐式参数，确保每个隐式参数都有一个唯一的占位符，可以通过后续约束求解来解决。约束收集收集将元变量连接到已知类型信息的关系，构建一个方程系统，捕获隐式参数和显式提供的项之间的相互依赖关系。约束求解采用专用约束求解器来确定元变量的具体值，使用统一算法和类型引导搜索来找到满足所有累积约束同时尊重程序类型结构的解决方案。

6.3.5.2 元变量解析

```
#![allow(unused)]
fn main() {
use std::collections::{HashMap, HashSet, VecDeque};
use std::fmt;
use crate::ast::{Term, Universe};
use crate::context::Context;
use crate::errors::TypeError;
/// Meta-variable identifier
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct MetaId(pub u64);
impl fmt::Display for MetaId {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "?m{}", self.0)
    }
}
/// Information about a meta-variable
#[derive(Debug, Clone)]
pub struct MetaInfo {
    pub id: MetaId,
    pub context: Vec<String>,           // Variables in scope when meta was created
    pub expected_type: Option<Term>,    // Expected type of the meta-variable
    pub solution: Option<Term>,         // Solution if solved
    pub dependencies: HashSet<MetaId>,  // Meta-variables this depends on
    pub occurrences: Vec<ConstraintId>, // Constraints this meta appears in
}
}
```

元变量表示必须通过约束求解解决的未知项。我们的实现为每个元变量维护全面的元数据，实现依赖跟踪和解决方案传播。

6.4 归纳类型

归纳类型构成了构造演算中数据结构的基础，提供了一种系统化的方式来定义带有构造器和消除原则的递归类型。我们的实现支持宇宙多态的归纳类型，带有依赖模式匹配，从而实现数据类型抽象，同时通过宇宙层次结构维持逻辑一致性。

归纳类型代表了对纯λ 演算的关键扩展，允许通过基于构造器的规范来定义像自然数、列表和树这样的具体数据结构。该实现提供了归纳声明的句法支持，

以及通过专用类型检查算法的语义处理，这些算法确保构造器一致性和穷尽模式覆盖。

考虑这些直观示例，它们展示了归纳类型的基本概念。一个简单的枚举，如一周的日子，可以表示为带有七个构造器的归纳类型：

```
inductive DayOfWeek : Type with
  | Monday    : DayOfWeek
  | Tuesday   : DayOfWeek
  | Wednesday : DayOfWeek
  | Thursday  : DayOfWeek
  | Friday    : DayOfWeek
  | Saturday  : DayOfWeek
  | Sunday    : DayOfWeek
```

同样，原色形成了另一个自然的归纳类型，带有三个不同的构造器：

```
inductive Color : Type with
  | Red   : Color
  | Green : Color
  | Blue  : Color
```

这些示例说明了归纳类型如何通过构造器声明提供一种系统化的方式来定义有限的独特值集合，每个构造器既作为数据构造器，又作为构造值属于归纳类型的证明。

更复杂的归纳类型可以包含引用正在定义的类型的递归构造器，从而实现像二叉树这样的数据结构：

```
inductive BinaryTree (A : Type) : Type with
  | Leaf : BinaryTree A
  | Node : A -> BinaryTree A -> BinaryTree A -> BinaryTree A
```

这个二叉树定义展示了几个重要概念：该类型由元素类型A参数化，`Leaf` 构造器创建空树，`Node` 构造器接受类型A的值以及两个子树来创建更大的树。`Node` 构造器的递归性质使得能够构建任意深度的树结构，同时通过归纳类型系统维持类型安全。

6.4.1 归纳类型声明

归纳类型声明的核心数据结构捕获了基于构造器的类型定义所需的所有基本组件：

```
#![allow(unused)]
fn main() {
/// Top-level declarations
#[derive(Debug, Clone, PartialEq)]
pub enum Declaration {
    /// Constant definition: def name {u...} (params) : type := body
    Definition {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        params: Vec<Parameter>,
        ty: Term,
        body: Term,
    },
    /// Axiom: axiom name {u...} : type
    Axiom {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        ty: Term,
    },
    /// Inductive type: inductive Name {u...} (params) : Type := constructors
    Inductive {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        params: Vec<Parameter>,
        ty: Term,
        constructors: Vec<Constructor>,
    },
    /// Structure (single constructor inductive): structure Name {u...} :=
    /// (fields)
    Structure {
        name: String,
        universe_params: Vec<String>, // Universe level parameters
        params: Vec<Parameter>,
```

```
        ty: Term,
        fields: Vec<Field>,
    },
}
/// Constructor for inductive types
#[derive(Debug, Clone, PartialEq)]
pub struct Constructor {
    pub name: String,
    pub ty: Term,
}
}
```

归纳声明指定类型名称、可选的宇宙参数用于宇宙多态、类型参数用于泛型类型、结果类型规范，以及构造器定义的集合。这种结构使得从简单枚举到宇宙多态类型族的复杂归纳类型成为可能。

6.4.2 构造器类型特化

当归纳类型包括参数时，构造器类型必须为每个使用上下文适当特化：

```
/// Specialize a constructor type by instantiating type parameters to match
/// expected type
fn specialize_constructor_type(
    &mut self,
    ctor_type: &Term,
    expected_type: &Term,
    _ctx: &Context,
) -> TypeResult<Term> {
    // For a constructor like nil : (A : Type) → List A
    // and expected type List B, we need to instantiate A with B
    let mut current_type = ctor_type.clone();
    let mut substitutions = Vec::new();
    // Collect all type parameters (Pi types where domain is Type)
    while let Term::Pi(param_name, param_ty, body, _implicit) = &current_type {
        if matches!(param_ty.as_ref(), Term::Sort(_)) {
            // This is a type parameter
            // We'll need to figure out what to substitute for it
            substitutions.push(param_name.clone());
            current_type = body.as_ref().clone();
        } else {
            break;
        }
    }
    // Now current_type should be the return type with free variables
    if substitutions.is_empty() {
        // No type parameters, just return the original type
        return Ok(ctor_type.clone());
    }
    // Handle type parameter substitution
    if !substitutions.is_empty() {
        // Extract type arguments from expected_type
        let mut type_args = Vec::new();
        let mut current_expected = expected_type;
        // For types like Pair A B, we need to extract both A and B
        while let Term::App(f, arg) = current_expected {
            type_args.push(arg.as_ref().clone());
            current_expected = f;
        }
        type_args.reverse();
        // Apply substitutions
        let mut specialized = ctor_type.clone();
        for (param, arg) in substitutions.iter().zip(type_args.iter()) {
            // Skip the Pi binding for this parameter and substitute in the body
            if let Term::Pi(p, _, body, _) = &specialized {
                if p == param {
                    specialized = self.substitute(param, arg, body);
                }
            }
        }
        return Ok(specialized);
    }
    // For more complex cases, would need proper unification
    Ok(ctor_type.clone())
}
```

构造器特化处理将泛型构造器类型实例化为特定类型参数的复杂过程。该算法遍历构造器类型签名，并用具体参数替换类型参数，同时保持构造器的结构属性。此过程使像`List A`这样的泛型归纳类型在实例化为特定元素类型时能够正确工作。

特化过程通过正确处理构造器类型中的宇宙参数来维持宇宙多态性。当构造器属于宇宙多态归纳类型时，特化算法确保在整个实例化过程中保持宇宙约束。

6.4.3 模式匹配实现

模式匹配为归纳类型提供消除原则，允许程序通过案例分析来分析归纳值：

```
#![allow(unused)]
fn main() {
/// Patterns for match expressions
#[derive(Debug, Clone, PartialEq)]
pub enum Pattern {
    /// Variable pattern: x
    Var(String),
    /// Constructor pattern: Cons x xs
    Constructor(String, Vec<Pattern>),
    /// Wildcard pattern: _
    Wildcard,
}
/// Pattern matching arms
#[derive(Debug, Clone, PartialEq)]
pub struct MatchArm {
    pub pattern: Pattern,
    pub body: Term,
}
}
```

模式系统支持解构归纳值的构造器模式、将匹配组件绑定到名称的变量模式，以及匹配任何值而不绑定的通配符模式。这种全面的模式语言能够完整分析归纳数据结构。

6.4.4 模式类型检查

模式匹配的类型检查算法确保模式与正在匹配的类型一致，并且所有情况产生兼容的结果类型：

```
/// Check a pattern against a type and return extended context with pattern
/// variables
fn check_pattern(
    &mut self,
    pattern: &Pattern,
    expected_type: &Term,
    ctx: &Context,
) -> TypeResult<Context> {
    match pattern {
        Pattern::Var(x) => {
            // Check if this is actually a constructor with no arguments
            if ctx.lookup_constructor(x).is_some() {
                // It's actually a constructor pattern with no arguments
                return self.check_pattern(
                    &Pattern::Constructor(x.clone(), vec![]),
                    expected_type,
                    ctx,
                );
            }
            // Variable pattern binds the scrutinee to the variable
            Ok(ctx.extend(x.clone(), expected_type.clone()))
        }
        Pattern::Wildcard => {
            // Wildcard matches anything, adds no bindings
            Ok(ctx.clone())
        }
        Pattern::Constructor(ctor_name, ctor_args) => {
            // Look up constructor type - check both constructors and definitions
            let (ctor_type, is_instantiated) =
                if let Some(ty) = ctx.lookup_constructor(ctor_name) {
                    (ty.clone(), false)
                } else if let Some(def) = ctx.lookup_definition(ctor_name) {
                    // Instantiate universe parameters for universe-polymorphic constructors
                    let instantiated = self.instantiate_universe_params(
                        &def.ty,
                        &def.universe_params,
                        &def.name,
                        ctx,
                    )?;
                    (instantiated, true)
                } else {
                    return Err(TypeError::UnknownConstructor {
                        name: ctor_name.clone(),
                    });
                };
            if is_instantiated {
                // For instantiated constructors, use constraint-based approach
                // The constructor type already has metavariables that can unify correctly
                let return_type = extract_constructor_return_type(&ctor_type);
                // Use constraint solving to check the return type matches expected type
                let mut solver = Solver::new(ctx.clone());
                let constraint_id = solver.new_constraint_id();
                let constraint = crate::solver::Constraint::Unify {
                    id: constraint_id,
                    left: return_type.clone(),
                    right: expected_type.clone(),
                    strength: ConstraintStrength::Required,
                };
                solver.add_constraint(constraint);
                let _subst = solver.solve()?; // This will error if unification fails
                // Extract argument types directly from instantiated constructor type
                let arg_types = extract_constructor_arg_types(&ctor_type);
                // Check sub-patterns against argument types
                if arg_types.len() != ctor_args.len() {
                    return Err(TypeError::ConstructorArityMismatch {
```

```
                        name: ctor_name.clone(),
                        expected: arg_types.len(),
                        actual: ctor_args.len(),
                    });
                }
                let mut extended_ctx = ctx.clone();
                for (arg_pattern, arg_type) in ctor_args.iter().zip(arg_types.iter()) {
                    extended_ctx = self.check_pattern(arg_pattern, arg_type, &extended_ctx)?;
                }
                Ok(extended_ctx)
            } else {
                // For regular constructors, use the existing specialization approach
                let specialized_ctor_type =
                    self.specialize_constructor_type(&ctor_type, expected_type, ctx)?;
                let return_type = extract_constructor_return_type(&specialized_ctor_type);
                if !self.definitionally_equal(&return_type, expected_type, ctx)? {
                    return Err(TypeError::TypeMismatch {
                        expected: expected_type.clone(),
                        actual: return_type,
                    });
                }
                // Extract argument types from specialized constructor type and check
                // sub-patterns
                let arg_types = extract_constructor_arg_types(&specialized_ctor_type);
                if ctor_args.len() != arg_types.len() {
                    return Err(TypeError::ConstructorArityMismatch {
                        name: ctor_name.clone(),
                        expected: arg_types.len(),
                        actual: ctor_args.len(),
                    });
                }
                let mut extended_ctx = ctx.clone();
                for (arg_pattern, arg_type) in ctor_args.iter().zip(arg_types.iter()) {
                    extended_ctx = self.check_pattern(arg_pattern, arg_type, &extended_ctx)?;
                }
                Ok(extended_ctx)
            }
        }
    }
}
```

模式类型检查验证构造器模式是否匹配其相应构造器的结构，变量模式是否从匹配上下文接收适当类型，以及通配符模式是否正确使用。该算法维护一个类型上下文，跟踪模式变量的类型，以便在结果表达式中使用。

模式检查器通过确保构造器模式正确实例化宇宙多态构造器来处理宇宙多态性。当模式匹配属于宇宙多态归纳类型的构造器时，检查器验证在整个模式匹配过程中满足宇宙约束。

6.4.5 构造器类型推断

项中的构造器应用需要专门的类型推断来处理构造器类型与其参数之间的交互：

```
/// Infer the type of a term
pub fn infer(&mut self, term: &Term, ctx: &Context) -> TypeResult<Term> {
    self.with_context(term, |checker| checker.infer_impl(term, ctx))
}
```

构造器类型推断从上下文查找构造器类型，用适当的类型参数特化它们，并验证构造器应用于正确类型的参数。该算法处理简单构造器和属于宇宙多态归纳类型的构造器。

6.4.6 基本归纳类型示例

像自然数和布尔值这样的简单归纳类型提供了归纳类型系统在实际中的具体示例：

```
#![allow(unused)]
fn main() {
inductive Bool : Type with
| true : Bool
| false : Bool

inductive Nat : Type with
| zero : Nat
| succ : Nat -> Nat

def is_zero (n : Nat) : Bool :=
  match n with
  case zero => true
  case succ(_) => false
}
```

这些示例展示了带有简单构造器和直接模式匹配的基本归纳类型声明。Bool 类型展示了枚举风格的归纳类型，而Nat 展示了带有构造器参数的递归归纳类型。

6.4.7 高级模式匹配

更多模式匹配示例说明了依赖模式匹配的表达能力：

```
#![allow(unused)]
fn main() {
inductive Bool : Type with
| true : Bool
| false : Bool

inductive Nat : Type with
| zero : Nat
| succ : Nat -> Nat

def predecessor (n : Nat) : Nat :=
  match n with
  case zero => zero
  case succ(m) => m

def not_bool (b : Bool) : Bool :=
  match b with
  case true => false
  case false => true
}
```

高级模式匹配展示了如何通过模式匹配来解构带有参数的构造器，模式变量接收从构造器签名派生的适当类型。predecessor 函数展示了递归构造器模式，而not_bool 函数展示了简单的枚举模式匹配。

6.4.8 依赖归纳类型

该实现支持依赖归纳类型，其中构造器类型可以依赖于项参数，从而实现数据结构：

```
#![allow(unused)]
fn main() {
inductive Nat : Type with
  | zero : Nat
  | succ : Nat -> Nat

inductive Vec (A : Type) : Nat -> Type with
  | nil : Vec A zero
  | cons : (n : Nat) -> A -> Vec A n -> Vec A (succ n)
}
```

依赖归纳类型代表了依赖类型理论中归纳类型的完整力量。这些类型实现了长度索引的向量、良好类型的抽象语法树，以及其他携带关于其内容计算信息的类型的数据结构。

6.5 类型规则

构造演算具有异常丰富的类型系统，将项、类型和种类统一到一个单一的语法类别中。这里呈现的类型规则捕捉了依赖类型、宇宙多态以及使我们的实现既强大又复杂的约束求解的核心本质。

与更简单的类型系统不同，CoC 的类型规则必须同时处理多层抽象。项可以依赖于其他项（函数），类型可以依赖于项（依赖类型），类型可以依赖于类型（多态），甚至种类可以通过宇宙约束依赖于项。

6.5.1 符号表

构造演算使用广泛的形式记号来捕捉项、类型和宇宙之间复杂的关联关系。本符号表提供了理解整个类型规则中使用的符号和概念的参考。

6.5.1.1 核心符号

- Γ (Gamma)：上下文或环境，跟踪哪些变量在作用域内及其类型
- $\vdash$ (turnstile)：判断符号，读作"证明"或"蕴涵"
- Π (Pi)：依赖函数类型（$A \to B$ 的泛化，其中结果类型可以依赖于输入值）
- λ (lambda)：函数抽象（创建函数）
- $\equiv$：定义相等性（根据计算规则"相同"的两个项）
- $\doteq$：统一约束（询问两个项是否可以被等同）
- $\rightsquigarrow$：约束求解（产生约束的解）

6.5.1.2 判断形式

- $\Gamma \vdash t : T$："在上下文Gamma 中，项t 具有类型T"
- $\Gamma \vdash T : \text{Type}_i$："T 是宇宙级别i 中的类型"
- $\Gamma \vdash C$："约束C 在上下文Gamma 中成立"
- $\Gamma \vdash s \equiv t : T$："项s 和t 在类型T 处可转换"

6.5.1.3 类型系统概念

- **宇宙层次结构**：类型居住在宇宙中（`Type_0`、`Type_1` 等）以避免逻辑悖论
- **依赖类型**：可以依赖于值的类型（例如"长度为n 的列表"）
- **元变量**：我们的约束求解器试图求解的未知量（写成?α）
- **替换**：用值替换变量，写成$t[s/x]$（在t 中用s 替换x）
- **正性**：对归纳类型的限制，以确保它们是良基的

6.5.1.4 记号约定

- **上划线**：$\overline{x}$ 表示"一序列x"（如x, x, x, ⋯）
- **括号**：$[s/x]$ 表示用s 替换x 的替换
- **下标**：Type_i 指宇宙级别
- **新鲜变量**：当我们说"fresh(α)"时，表示一个全新的、未使用的变量

6.5.2 核心判断形式

CoC 类型系统使用几种以复杂方式交互的判断形式：

类型判断：$\Gamma \vdash t : T$ 断言项t 在上下文Γ 中具有类型T

宇宙判断：$\Gamma \vdash T : \text{Type}_i$ 断言T 是宇宙i 中的类型

约束判断：$\Gamma \vdash C$ 断言约束C 在上下文Γ 中成立

转换判断：$\Gamma \vdash s \equiv t : T$ 断言s 和t 在类型T 处可转换

6.5.3 变量和常量规则

在上下文中的变量查找：

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

带有显式级别约束的宇宙层次结构：

$$\frac{i < j}{\Gamma \vdash \text{Type}_i : \text{Type}_j} \quad \text{(T-Univ)}$$

带有其类型的原语常量：

$$\frac{}{\Gamma \vdash \text{Nat} : \text{Type}_0} \quad \text{(T-Nat)}$$

$$\frac{}{\Gamma \vdash 0 : \text{Nat}} \quad \text{(T-Zero)}$$

$$\frac{}{\Gamma \vdash \text{succ} : \text{Nat} \to \text{Nat}} \quad \text{(T-Succ)}$$

6.5.4 函数类型和抽象

依赖函数类型（Pi 类型）：

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_j}{\Gamma \vdash \Pi x : A.B : \text{Type}_{\max(i,j)}} \quad \text{(T-Pi)}$$

带有依赖类型的Lambda 抽象：

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A.B : \text{Type}_i}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \quad \text{(T-Lam)}$$

带有替换的函数应用：

$$\frac{\Gamma \vdash f : \Pi x : A.B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[a/x]} \quad \text{(T-App)}$$

6.5.5 归纳类型

带有宇宙约束的归纳类型形成：

$$\frac{\overline{\Gamma \vdash C_i : A_i \to T\ ;\ \text{params}} \quad \Gamma \vdash T : \text{Type}_j}{\Gamma \vdash \text{data}\ T\ \text{where}\ \overline{C_i : A_i} : \text{Type}_j} \quad \text{(T-Data)}$$

带有正性约束的构造器类型：

$$\frac{\Gamma \vdash I : \text{Type}_i \quad \text{Positive}(I, A)}{\Gamma \vdash c : A \to I} \quad \text{(T-Constr)}$$

带有依赖消除的模式匹配：

$$\frac{\begin{array}{c}\Gamma \vdash t : I\ ;\ \overline{p} \\ \Gamma \vdash P : \Pi\overline{x : A}.I\ ;\ \overline{x} \to \mathrm{Type}_k \\ \overline{\Gamma \vdash f_i : \Pi\overline{y : B_i}.P\ ;(c_i\ ;\ \overline{y})}\end{array}}{\Gamma \vdash \mathrm{match}\ t\ \mathrm{return}\ P\ \mathrm{with}\ \overline{c_i\ ;\ \overline{y} \Rightarrow f_i\ ;\ \overline{y} : P}\ ;t} \quad \text{(T-Match)}$$

6.5.6 宇宙多态

类型中的宇宙变量：

$$\frac{\alpha \in \mathrm{UVars}}{\Gamma \vdash \mathrm{Type}_\alpha : \mathrm{Type}_{\alpha+1}} \quad \text{(T-UVar)}$$

宇宙级别约束：

$$\frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash C_1 \wedge C_2} \quad \text{(T-Conj)}$$

$$\frac{i \le j}{\Gamma \vdash i \le j} \quad \text{(T-Leq)}$$

宇宙最大值运算：

$$\frac{\Gamma \vdash i \le k \quad \Gamma \vdash j \le k}{\Gamma \vdash \max(i, j) \le k} \quad \text{(T-Max)}$$

6.5.7 转换和定义相等性

函数应用的Beta 归约：

$$\frac{}{\Gamma \vdash (\lambda x : A.t)\ a \equiv t[a/x] : B[a/x]} \quad \text{(Conv-Beta)}$$

函数类型的Eta 转换：

$$\frac{\Gamma \vdash f : \Pi x : A.B \quad x \notin \mathrm{FV}(f)}{\Gamma \vdash f \equiv \lambda x : A.f\ x : \Pi x : A.B} \quad \text{(Conv-Eta)}$$

模式匹配的Iota 归约：

$$\frac{}{\Gamma \vdash \mathrm{match}\ (c\ ;\ \overline{a})\ \mathrm{return}\ P\ \mathrm{with}\ \overline{c_i\ ;\ \overline{y} \Rightarrow f_i\ ;\ \overline{y} \equiv f\ ;\ \overline{a} : P}\ ;(c\ ;\ \overline{a})} \quad \text{(Conv-Iota)}$$

结构转换的同余规则：

$$\frac{\Gamma \vdash s_1 \equiv t_1 : A \to B \quad \Gamma \vdash s_2 \equiv t_2 : A}{\Gamma \vdash s_1\ s_2 \equiv t_1\ t_2 : B} \quad \text{(Conv-App)}$$

6.5.8 类型转换和子类型

转换允许定义相等的类型可以互换使用：

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A \equiv B : \mathrm{Type}_i}{\Gamma \vdash t : B} \quad \text{(T-Conv)}$$

6.5.9 元变量和约束规则

用于推断的元变量引入：

$$\frac{\mathrm{fresh}(\alpha) \quad \Gamma \vdash T : \mathrm{Type}_i}{\Gamma \vdash ?\alpha : T} \quad \text{(T-Meta)}$$

带有统一的约束求解：

$$\frac{\Gamma \vdash s : T \quad \Gamma \vdash t : T \quad \mathrm{Unify}(s, t, \sigma)}{\Gamma \vdash s \doteq t : T \rightsquigarrow \sigma} \quad \text{(T-Unify)}$$

通过替换的约束传播：

$$\frac{\Gamma \vdash C[\sigma] \quad \mathrm{Dom}(\sigma) \subseteq \mathrm{MetaVars}(C)}{\Gamma \vdash C \rightsquigarrow \sigma} \quad \text{(T-Subst)}$$

这些规则捕捉了依赖类型、宇宙约束和元变量统一之间的相互作用，这使得构造演算既具有表达力又难以实现。关键洞见是类型检查和约束求解必须齐头并进，每个阶段都告知并约束另一个阶段。

6.6 宇宙多态

宇宙多态使定义能够抽象过宇宙级别，创建真正跨整个宇宙层次结构工作的泛型构造。我们的实现包括一个专用的宇宙约束求解器，处理在多态宇宙上下文中出现的复杂算术和约束关系。

6.6.1 宇宙约束求解器

```
#![allow(unused)]
fn main() {
use std::collections::{HashMap, HashSet};
/// 多态宇宙层级的宇宙约束求解器
use crate::ast::{Universe, UniverseConstraint};
impl UniverseSolver {
    pub fn new() -> Self {
        UniverseSolver {
            substitutions: HashMap::new(),
        }
    }

    /// 求解宇宙约束并返回替换
    pub fn solve(&mut self, constraints: &[UniverseConstraint]) -> Result<(), String> {
        // XXX: 简单的约束求解器，在完整系统中会更加复杂
        for constraint in constraints {
            self.solve_constraint(constraint)?;
        }
        Ok(())
    }

    /// 求解单个约束
    fn solve_constraint(&mut self, constraint: &UniverseConstraint) -> Result<(), String> {
        match constraint {
            UniverseConstraint::Equal(u1, u2) => self.unify_universes(u1, u2),
            UniverseConstraint::LessEq(u1, u2) => {
                // XXX: 目前将小于等于视为相等
                self.unify_universes(u1, u2)
            }
        }
    }

    /// 统一两个宇宙层级
    fn unify_universes(&mut self, u1: &Universe, u2: &Universe) -> Result<(), String> {
```

```
        let u1_subst = self.apply_substitution(u1);
        let u2_subst = self.apply_substitution(u2);

        match (&u1_subst, &u2_subst) {
            (Universe::ScopedVar(_, x), u) | (u, Universe::ScopedVar(_, x)) => {
                if let Universe::ScopedVar(_, y) = u {
                    if x == y {
                        return Ok(());
                    }
                }

                // 发生检查
                if self.occurs_check(x, u) {
                    return Err(format!(
                        "宇宙变量 {} 出现在 {}",
                        x,
                        self.universe_to_string(u)
                    ));
                }

                self.substitutions.insert(x.clone(), u.clone());
                Ok(())
            }

            (Universe::Const(n1), Universe::Const(n2)) if n1 == n2 => Ok(()),
            (Universe::Const(_), Universe::Const(_)) => {
                Err("不能统一不同的常量".to_string())
            }

            (Universe::Add(u1, n1), Universe::Add(u2, n2)) if n1 == n2 => {
                self.unify_universes(u1, u2)
            }
            (Universe::Add(_, _), Universe::Add(_, _)) => {
                Err("不能统一不同的加法".to_string())
            }

            (Universe::Max(u1, v1), Universe::Max(u2, v2)) => {
                self.unify_universes(u1, u2)?;
                self.unify_universes(v1, v2)
            }

            (Universe::IMax(u1, v1), Universe::IMax(u2, v2)) => {
                self.unify_universes(u1, u2)?;
                self.unify_universes(v1, v2)
            }

            _ => Err(format!(
                "不能统一宇宙层级 {} 和 {}",
                self.universe_to_string(&u1_subst),
                self.universe_to_string(&u2_subst)
            )),
        }
    }

    /// 对宇宙层级应用替换
    fn apply_substitution(&self, u: &Universe) -> Universe {
        match u {
            Universe::ScopedVar(_scope, x) => {
                if let Some(subst) = self.substitutions.get(x) {
                    self.apply_substitution(subst)
                } else {
                    u.clone()
                }
            }
            Universe::Const(n) => Universe::Const(*n),
            Universe::Add(u, n) => Universe::Add(Box::new(self.apply_substitution(u)), *n),
            Universe::Max(u, v) => Universe::Max(
                Box::new(self.apply_substitution(u)),
                Box::new(self.apply_substitution(v)),
            ),
            Universe::IMax(u, v) => Universe::IMax(
                Box::new(self.apply_substitution(u)),
                Box::new(self.apply_substitution(v)),
            ),
            _ => u.clone(),
        }
    }

    /// 检查宇宙变量是否出现在宇宙层级中
    fn occurs_check(&self, var: &str, u: &Universe) -> bool {
        match u {
            Universe::ScopedVar(_, x) => x == var,
            Universe::Add(u, _) => self.occurs_check(var, u),
            Universe::Max(u, v) | Universe::IMax(u, v) => {
                self.occurs_check(var, u) || self.occurs_check(var, v)
            }
            Universe::Const(_) => false,
            Universe::Meta(_) => false, // 元变量不包含命名变量
        }
    }

    /// 将宇宙转换为字符串表示
    fn universe_to_string(&self, u: &Universe) -> String {
        match u {
            Universe::Const(n) => n.to_string(),
            Universe::ScopedVar(scope, var) => format!("{}::{}", scope, var),
            Universe::Meta(id) => format!("?u{}", id),
            Universe::Add(u, n) => format!("({} + {})", self.universe_to_string(u), n),
            Universe::Max(u, v) => format!(
                "max({}, {})",
                self.universe_to_string(u),
                self.universe_to_string(v)
            ),
            Universe::IMax(u, v) => format!(
```

```
                "imax({}, {})",
                self.universe_to_string(u),
                self.universe_to_string(v)
            ),
        }
    }

    /// 获取宇宙变量的最终替换
    pub fn get_substitution(&self, var: &str) -> Option<Universe> {
        self.substitutions
            .get(var)
            .map(|u| self.apply_substitution(u))
    }

    /// 对宇宙层级应用所有替换
    pub fn substitute_universe(&self, u: &Universe) -> Universe {
        self.apply_substitution(u)
    }

    /// 获取所有替换
    pub fn get_all_substitutions(&self) -> &HashMap<String, Universe> {
        &self.substitutions
    }

    /// 检查宇宙层级约束是否可满足
    pub fn is_satisfiable(&self, constraints: &[UniverseConstraint]) -> bool {
        let mut solver = self.clone();
        solver.solve(constraints).is_ok()
    }

    /// 生成新的宇宙变量
    pub fn fresh_universe_var(&self, base: &str, avoid: &HashSet<String>) -> String {
        let mut counter = 0;
        loop {
            let name = if counter == 0 {
                base.to_string()
            } else {
                format!("{}{}", base, counter)
            };
            if !avoid.contains(&name) && !self.substitutions.contains_key(&name) {
                return name;
            }
            counter += 1;
        }
    }
}
impl Default for UniverseSolver {
    fn default() -> Self {
        Self::new()
    }
}
/// 宇宙层级约束求解器
#[derive(Debug, Clone)]
pub struct UniverseSolver {
    /// 宇宙变量的替换
    substitutions: HashMap<String, Universe>,
}
}
```

UniverseSolver 独立于主约束求解器管理宇宙层级约束，从而为宇宙算术和层级统一提供专门算法。这种分离允许高效处理宇宙多态性，而不使主类型检查算法复杂化。

6.6.1.1 宇宙约束类型

宇宙约束捕获必须为逻辑一致性而保持的宇宙层级之间的关系：

```
#![allow(unused)]
fn main() {
/// Universe level constraints for polymorphism
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum UniverseConstraint {
    /// u ≤ v
    LessEq(Universe, Universe),
    /// u = v
    Equal(Universe, Universe),
}
}
```

相等性约束（Equal）要求两个宇宙级别相同，源于依赖上下文中类型相等性要求，其中宇宙级别必须完全匹配。

排序约束（LessEq）确保一个宇宙级别小于或等于另一个，维持防止逻辑悖论的预测性要求，通过维持类型宇宙的严格层次结构。

6.6.1.2 宇宙统一算法

宇宙统一展示了处理宇宙级别算术的复杂性。该算法处理多种宇宙表达式形式：

1. **变量统一**：当统一宇宙变量与任何表达式时，我们在执行发生检查以防止无限宇宙表达式后创建替换映射。
2. **常量统一**：宇宙常量只能与相同常量统一，确保像`Type 0` 和`Type 1` 这样的具体级别保持不同。
3. **算术表达式**：像$u + 1$ 和$\max(u, v)$ 这样的宇宙表达式需要结构分解，我们递归统一组件宇宙表达式。

6.6.1.3 替换和归一化

```
/// Apply substitutions to a universe level
fn apply_substitution(&self, u: &Universe) -> Universe {
    match u {
        Universe::ScopedVar(_scope, x) => {
            if let Some(subst) = self.substitutions.get(x) {
                self.apply_substitution(subst)
            } else {
                u.clone()
            }
        }
        Universe::Const(n) => Universe::Const(*n),
        Universe::Add(u, n) => Universe::Add(Box::new(self.apply_substitution(u)), *n),
        Universe::Max(u, v) => Universe::Max(
            Box::new(self.apply_substitution(u)),
            Box::new(self.apply_substitution(v)),
        ),
        Universe::IMax(u, v) => Universe::IMax(
            Box::new(self.apply_substitution(u)),
            Box::new(self.apply_substitution(v)),
        ),
        _ => u.clone(),
    }
}
```

宇宙替换应用展示了宇宙表达式的递归性质。当替换到像`Add` 或`Max` 这样的复合表达式时，我们必须递归地将替换应用到所有子组件，同时保持算术结构。

6.6.1.4 宇宙变量的发生检查

```
/// Check if a universe variable occurs in a universe level
fn occurs_check(&self, var: &str, u: &Universe) -> bool {
    match u {
        Universe::ScopedVar(_, x) => x == var,
        Universe::Add(u, _) => self.occurs_check(var, u),
        Universe::Max(u, v) | Universe::IMax(u, v) => {
            self.occurs_check(var, u) || self.occurs_check(var, v)
        }
        Universe::Const(_) => false,
        Universe::Meta(_) => false, // 元变量不包含命名变量
    }
}
```

发生检查通过确保宇宙变量不会出现在其自己的解决方案中来防止无限宇宙表达式。此检查必须遍历宇宙表达式的整个结构，包括算术运算和最大值表达式。

6.6.1.5 宇宙表达式归一化

宇宙求解器包括归一化功能，将宇宙表达式简化为规范形式：

```
match u {
    Universe::Add(base, n) => {
        let base_norm = self.normalize_universe_static(base);
        match base_norm {
```

```
            Universe::Const(m) => Universe::Const(m + n),
            _ => Universe::Add(Box::new(base_norm), *n),
        }
    }
    Universe::Max(u1, u2) => {
        let u1_norm = self.normalize_universe_static(u1);
        let u2_norm = self.normalize_universe_static(u2);
        match (&u1_norm, &u2_norm) {
            (Universe::Const(n1), Universe::Const(n2)) => Universe::Const((*n1).max(*n2)),
            _ => Universe::Max(Box::new(u1_norm), Box::new(u2_norm)),
        }
    }
    _ => u.clone(),
}
```

此归一化过程：

- **算术简化**：在加法表达式中组合常量，如`Const(2) + 3`变为`Const(5)`
- **最大值计算**：评估常量之间的最大值表达式
- **规范形式**：维持归一化表达式，提高统一成功率

6.6.2 宇宙多态定义

宇宙多态使定义能够：

```
def id.{u} (A : Sort u) (x : A) : A := x
```

`.{u}` 语法引入一个宇宙参数，可以在不同级别实例化：

```
-- id 在 Type 0 实例化
id_nat : Nat → Nat := id.{0} Nat

-- id 在 Type 1 实例化
id_type : Type → Type := id.{1} Type
```

6.6.2.1 新鲜变量生成

```
/// Generate fresh universe variable
pub fn fresh_universe_var(&self, base: &str, avoid: &HashSet<String>) -> String {
    let mut counter = 0;
    loop {
        let name = if counter == 0 {
            base.to_string()
        } else {
            format!("{}{}", base, counter)
        };
        if !avoid.contains(&name) && !self.substitutions.contains_key(&name) {
            return name;
        }
        counter += 1;
    }
}
```

新鲜宇宙变量生成确保每个宇宙抽象获得唯一的变量名称，防止复杂多态定义中的冲突。该算法：

1. **基础名称生成**：从描述性基础名称开始
2. **冲突避免**：检查现有变量和替换
3. **计数器扩展**：发生冲突时添加数字后缀
4. **唯一性保证**：确保返回的名称全局唯一

6.6.2.2 替换管理

```
/// Get the final substitution for a universe variable
pub fn get_substitution(&self, var: &str) -> Option<Universe> {
    self.substitutions
        .get(var)
        .map(|u| self.apply_substitution(u))
}
```

```
/// Apply all substitutions to a universe level
pub fn substitute_universe(&self, u: &Universe) -> Universe {
    self.apply_substitution(u)
}
```

求解器提供对已解决的宇宙替换的访问，使主类型检查器能够在整个类型检查过程中应用宇宙级别解决方案。

宇宙替换应用处理宇宙表达式的完全解析，递归应用所有累积替换以产生完全解析的宇宙级别。

6.6.3 与类型检查的集成

宇宙求解器在几个点与主类型检查算法集成：

- **类型形成**：当检查类型是否良构时，宇宙求解器确保满足宇宙级别约束。
- **多态实例化**：当实例化多态定义时，宇宙求解器生成新的宇宙变量并维护它们之间的约束。
- **定义相等性**：当检查具有宇宙多态的类型之间的定义相等性时，宇宙求解器确保保持宇宙关系。

6.6.3.1 约束满足性检查

```
/// Check if universe level constraints are satisfiable
pub fn is_satisfiable(&self, constraints: &[UniverseConstraint]) -> bool {
    let mut solver = self.clone();
    solver.solve(constraints).is_ok()
}
```

满足性检查器使类型检查器能够在提交特定类型赋值之前验证宇宙约束集是否有解决方案。此早期检查防止复杂类型推断场景中的回溯。

6.6.4 宇宙多态示例

我们的实现支持几种形式的宇宙多态定义：

6.6.4.1 多态数据类型

```
structure Pair.{u, v} (A : Sort u) (B : Sort v) : Sort (max u v) :=
  (fst : A)
  (snd : B)
```

`Pair` 类型在两个宇宙级别上多态，结果类型位于参数宇宙级别的最大值。

6.6.4.2 多态函数

```
def const.{u, v} (A : Sort u) (B : Sort v) (x : A) (y : B) : A := x
```

`const` 函数忽略其第二个参数并返回第一个，在两种参数类型的任何宇宙级别上工作。

6.6.4.3 宇宙级别算术

```
def lift.{u} (A : Sort u) : Sort (u + 1) := A
```

`lift` 操作将类型从宇宙u 移动到宇宙$u + 1$，展示了类型表达式中的宇宙级别算术。

6.6.5 失败模式

我们处理几个失败情况，带有自定义错误：

```
/// Convert universe to string representation
fn universe_to_string(&self, u: &Universe) -> String {
    match u {
        Universe::Const(n) => n.to_string(),
        Universe::ScopedVar(scope, var) => format!("{}::{}", scope, var),
        Universe::Meta(id) => format!("?u{}", id),
        Universe::Add(u, n) => format!("({} + {})", self.universe_to_string(u), n),
        Universe::Max(u, v) => format!(
            "max({}, {})",
            self.universe_to_string(u),
            self.universe_to_string(v)
        ),
        Universe::IMax(u, v) => format!(
            "imax({}, {})",
            self.universe_to_string(u),
            self.universe_to_string(v)
        ),
    }
}
```

失败分为三种主要类型：

- **统一失败**：显示无法统一的特定宇宙级别
- **发生检查违规**：识别无限宇宙表达式
- **算术不一致**：指出无效的宇宙级别算术

6.7 约束求解

约束求解器构成了我们构造演算实现的核心，通过系统的约束传播和统一来处理解决未知项、类型和宇宙级别的复杂任务。

这是一个非平凡的软件片段，需要相当多的关注和思考。虽然这是一段困难的代码，但它仍然比Coq 和Lean 中超过10 万行的实现简单得多。这是本项目的激励示例，即拥有一个小的CoC 实现，一个足够有动力（可能还需要咖啡因）的本科生可以在一个下午内读完。

所以拿起一杯浓缩咖啡（或两杯或三杯），系好安全带!

6.7.1 核心数据结构

求解器在几个基本数据结构上运行，这些结构捕获了依赖类型系统中基于约束的类型推断的本质。

6.7.1.1 元变量管理

元变量表示必须通过约束求解解决的未知项。我们的实现为每个元变量维护全面的元数据，实现依赖跟踪和解决方案传播。

```
#![allow(unused)]
fn main() {
use std::collections::{HashMap, HashSet, VecDeque};
use std::fmt;
use crate::ast::{Term, Universe};
use crate::context::Context;
use crate::errors::TypeError;

/// 元变量标识符
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct MetaId(pub u64);

impl fmt::Display for MetaId {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "?m{}", self.0)
    }
}

/// 宇宙元变量标识符
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
```

```
pub struct UniverseMetaId(pub u64);

impl fmt::Display for UniverseMetaId {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "?u{}", self.0)
    }
}

/// 关于元变量的信息
#[derive(Debug, Clone)]
pub struct MetaInfo {
    pub id: MetaId,
    pub context: Vec<String>, // 创建元变量时作用域中的变量
    pub expected_type: Option<Term>, // 元变量的期望类型
    pub solution: Option<Term>, // 如果已解决，则为解决方案
    pub dependencies: HashSet<MetaId>, // 此元变量依赖的其他元变量
    pub occurrences: Vec<ConstraintId>, // 此元变量出现的约束
}
}
```

MetaInfo 结构展示了我们如何跟踪未知项的完整生命周期。context 字段保留了元变量创建点的变量作用域，确保解决方案尊重词法作用域。dependencies 字段跟踪在解决此元变量之前必须解决的其他元变量，从而能够对约束解决进行拓扑排序。

6.7.1.2 约束表示

我们的约束系统支持在依赖类型检查期间出现的多种关系类别：

统一约束（Unify）表示两个项必须相等的核心要求，强度表示求解优先级。这些约束驱动主要的统一算法并处理大多数结构相等性要求。

类型约束（HasType）确保项属于其期望类型，使类型导向的约束生成能够引导求解器找到有意义的解决方案。

宇宙约束处理宇宙级别之间的复杂关系，支持等式（UnifyUniverse）和排序（UniverseLevel）要求，这些要求维持层次结构的一致性。

延迟约束表示无法立即解决的模式，等待特定的元变量被解决后再尝试解决。

```
#![allow(unused)]
fn main() {
/// 约束标识符
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct ConstraintId(pub u64);

/// 约束类型
#[derive(Debug, Clone)]
pub enum Constraint {
    /// 统一: t1 ≡ t2
    Unify {
        id: ConstraintId,
        left: Term,
        right: Term,
        strength: ConstraintStrength,
    },
    /// 类型约束: term : type
    HasType {
        id: ConstraintId,
        term: Term,
        expected_type: Term,
    },
    /// 宇宙统一: u1 ≡ u2
    UnifyUniverse {
        id: ConstraintId,
        left: Universe,
        right: Universe,
        strength: ConstraintStrength,
    },
    /// 宇宙级别约束: u1 ≤ u2
    UniverseLevel {
        id: ConstraintId,
        left: Universe,
        right: Universe,
    },
    /// 延迟约束（用于复杂模式）
    Delayed {
        id: ConstraintId,
        constraint: Box<Constraint>,
        waiting_on: HashSet<MetaId>,
    },
}
```

```
/// 约束强度，用于优先级排序
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum ConstraintStrength {
    Required,  // 必须解决
    Preferred, // 如果可能，应该解决
    Weak,      // 可以推迟
}
}
```

约束强度系统通过三层优先级层次结构实现智能求解顺序，最大化找到解决方案的机会。必需约束表示必须满足的基本关系，以便类型检查成功，并在约束解决期间获得最高优先级。这些约束通常来自显式类型注释或不能妥协的结构要求。

首选约束应该在可能时解决，但如果需要，可以推迟以在更关键的约束上取得进展。这些约束通常表示期望的属性或优化，这些属性或优化提高了解决方案的质量，但对于正确性不是严格必要的。弱约束向求解器提供关于首选解决方案的指导，但如果它们被证明无法解决，则不会阻止进展，允许算法在理想解决方案不可用时找到可行的解决方案。

6.7.2 约束求解器

Solver 代表了我们的约束求解方法的顶峰，将多种算法集成到一个统一的框架中。求解器维护几个关键数据结构：

元变量注册表（metas）跟踪所有未知项及其完整的元数据和依赖关系。

约束管理（constraints、queue）在优先级队列中维护活动约束，该队列能够实现智能求解顺序。

依赖跟踪（dependencies）构建元变量和约束之间关系的图，使能够进行拓扑求解和循环检测。

高级功能（enable_miller_patterns、enable_has_type_solving）可以激活以处理高阶统一模式和复杂的类型约束。

```
#![allow(unused)]
fn main() {
/// 具有依赖跟踪的高级约束求解器
pub struct Solver {
    /// 元变量
    metas: HashMap<MetaId, MetaInfo>,
    /// 活动约束
    constraints: HashMap<ConstraintId, Constraint>,
    /// 约束队列（按优先级排序）
    queue: VecDeque<ConstraintId>,
    /// 已解决的替换
    substitution: Substitution,
    /// 下一个元 ID
    next_meta_id: u64,
    /// 下一个宇宙元 ID
    next_universe_meta_id: u32,
    /// 下一个约束 ID
    next_constraint_id: u64,
    /// 类型检查上下文
    context: Context,
    /// 依赖图：元变量 -> 依赖它的约束
    dependencies: HashMap<MetaId, HashSet<ConstraintId>>,
    /// 无法立即解决的延迟约束
    delayed_constraints: HashSet<ConstraintId>,
    /// 递归深度计数器，防止栈溢出
    recursion_depth: u32,
    /// 启用 Miller 模式求解的标志（高级）
    enable_miller_patterns: bool,
    /// 启用 HasType 约束求解的标志（高级）
    enable_has_type_solving: bool,
}
}
```

6.7.2.1 元变量创建和跟踪

新鲜元变量生成展示了我们如何维护适当的作用域和上下文信息。每个元变量捕获在其创建点作用域中的变量，使能够在解决方案期间进行适当的变量捕获分析。

6.7.2.2 约束管理和依赖

将约束添加到求解器涉及依赖跟踪，该跟踪识别出现在每个约束中的元变量。这种依赖跟踪使求解器能够唤醒等待已解决的元变量的休眠约束，允许先前被阻止的约束变为活动并可能可解决。拓扑排序确保约束按照尊重其依赖关系的顺序解决，通过在处理更复杂的约束之前处理更简单的约束来最大化成功解决的机会。

6.7.3 替换系统

替换表示约束系统的部分解决方案，将元变量映射到其已解决的项。我们的替换系统处理项级别和宇宙级别的替换，并进行适当的归一化。

```
#![allow(unused)]
fn main() {
/// 将元变量映射到项的替换
#[derive(Debug, Clone, Default)]
pub struct Substitution {
    mapping: HashMap<MetaId, Term>,
    universe_mapping: HashMap<u32, Universe>,
}

impl Substitution {
    pub fn new() -> Self {
        Self::default()
    }

    pub fn insert(&mut self, meta: MetaId, term: Term) {
        self.mapping.insert(meta, term);
    }

    pub fn get(&self, meta: &MetaId) -> Option<&Term> {
        self.mapping.get(meta)
    }

    pub fn insert_universe(&mut self, meta_id: u32, universe: Universe) {
        self.universe_mapping.insert(meta_id, universe);
    }

    pub fn get_universe(&self, meta_id: &u32) -> Option<&Universe> {
        self.universe_mapping.get(meta_id)
    }

    /// 将替换应用到项
    pub fn apply(&self, term: &Term) -> Term {
        match term {
            Term::Meta(name) => {
                // 尝试解析元 ID 并查找替换
                if let Some(meta_id) = self.parse_meta_name(name) {
                    if let Some(subst) = self.mapping.get(&meta_id) {
                        // 递归应用替换
                        return self.apply(subst);
                    }
                }
                term.clone()
            }
            Term::App(f, arg) => Term::App(Box::new(self.apply(f)), Box::new(self.apply(arg))),
            Term::Abs(x, ty, body) => Term::Abs(
                x.clone(),
                Box::new(self.apply(ty)),
                Box::new(self.apply(body)),
            ),
            Term::Pi(x, ty, body, implicit) => Term::Pi(
                x.clone(),
                Box::new(self.apply(ty)),
                Box::new(self.apply(body)),
                *implicit,
            ),
            Term::Let(x, ty, val, body) => Term::Let(
                x.clone(),
                Box::new(self.apply(ty)),
                Box::new(self.apply(val)),
                Box::new(self.apply(body)),
            ),
            Term::Sort(u) => Term::Sort(self.apply_universe(u)),
            // 其他情况保持不变
            _ => term.clone(),
        }
    }

    /// 将替换应用到宇宙
    pub fn apply_universe(&self, universe: &Universe) -> Universe {
        let result = match universe {
            Universe::Meta(id) => {
                if let Some(subst) = self.universe_mapping.get(id) {
                    // 递归应用替换
                    self.apply_universe(subst)
                } else {
                    universe.clone()
                }
            }
            Universe::Add(base, n) => Universe::Add(Box::new(self.apply_universe(base)), *n),
```

```
            Universe::Max(u, v) => Universe::Max(
                Box::new(self.apply_universe(u)),
                Box::new(self.apply_universe(v)),
            ),
            Universe::IMax(u, v) => Universe::IMax(
                Box::new(self.apply_universe(u)),
                Box::new(self.apply_universe(v)),
            ),
            // 常量、变量和作用域变量在替换期间保持不变
            Universe::Const(_) | Universe::ScopedVar(_, _) => universe.clone(),
        };
        // 归一化结果以简化像 Const(0) + 1 -> Const(1) 这样的表达式
        self.normalize_universe_static(&result)
    }

    /// 静态归一化
    fn normalize_universe_static(&self, u: &Universe) -> Universe {
        match u {
            Universe::Add(base, n) => {
                let base_norm = self.normalize_universe_static(base);
                match base_norm {
                    Universe::Const(m) => Universe::Const(m + n),
                    _ => Universe::Add(Box::new(base_norm), *n),
                }
            }
            Universe::Max(u1, u2) => {
                let u1_norm = self.normalize_universe_static(u1);
                let u2_norm = self.normalize_universe_static(u2);
                match (&u1_norm, &u2_norm) {
                    (Universe::Const(n1), Universe::Const(n2)) => Universe::Const((*n1).max(*n2)),
                    _ => Universe::Max(Box::new(u1_norm), Box::new(u2_norm)),
                }
            }
            _ => u.clone(),
        }
    }

    fn parse_meta_name(&self, name: &str) -> Option<MetaId> {
        if let Some(stripped) = name.strip_prefix("?m") {
            stripped.parse::<u64>().ok().map(MetaId)
        } else {
            None
        }
    }
}
}
```

替换应用算法展示了依赖类型系统中约束求解的递归性质。当将替换应用到像`Pi`、`Abs` 和`Let` 这样的项时，我们必须仔细处理变量绑定和作用域，以避免捕获问题。

6.7.4 主要约束求解算法

主要求解循环展示了驱动我们的约束求解器通过约束解决和解决方案传播的系统过程的迭代约束传播方法。

```
#![allow(unused)]
fn main() {
/// 主要求解循环
pub fn solve(&mut self) -> Result<Substitution, TypeError> {
    let max_iterations = 1000;
    let mut iterations = 0;

    while !self.queue.is_empty() && iterations < max_iterations {
        iterations += 1;

        // 选择要解决的最佳约束
        if let Some(constraint_id) = self.pick_constraint() {
            let constraint =
                self.constraints
                    .remove(&constraint_id)
                    .ok_or_else(|| TypeError::Internal {
                        message: "约束消失了".to_string(),
                    })?;

            match self.solve_constraint(constraint.clone()) {
                Ok(progress) => {
                    if progress {
                        // 传播解决方案
                        self.propagate_solution()?;

                        // 唤醒可能现在可解决的延迟约束
                        let woken_constraints = self.wake_delayed_constraints()?;

                        // 将唤醒的约束以高优先级添加回队列
                        for woken_id in woken_constraints {
                            self.queue.push_front(woken_id);
                        }
                    }
                }
                Err(e) => {
                    // 如果可能，尝试延迟约束
                    if self.can_delay(&constraint_id) {
                        // 确定要等待的元变量
```

```
                    let waiting_on = match &constraint {
                        Constraint::Unify { left, right, .. } => {
                            let mut metas = self.collect_metas_in_term(left);
                            metas.extend(self.collect_metas_in_term(right));
                            metas
                                .into_iter()
                                .filter(|meta_id| self.is_unsolved_meta(meta_id))
                                .collect()
                        }
                        Constraint::HasType {
                            term,
                            expected_type,
                            ..
                        } => {
                            let mut metas = self.collect_metas_in_term(term);
                            metas.extend(self.collect_metas_in_term(expected_type));
                            metas
                                .into_iter()
                                .filter(|meta_id| self.is_unsolved_meta(meta_id))
                                .collect()
                        }
                        _ => HashSet::new(),
                    };

                    if !waiting_on.is_empty() {
                        // 将约束放回并延迟它
                        self.constraints.insert(constraint_id, constraint);
                        self.delay_constraint(constraint_id, waiting_on)?;
                    } else {
                        // 无法延迟，返回错误
                        return Err(e);
                    }
                } else {
                    return Err(e);
                }
            }
        }
    }

    if iterations >= max_iterations {
        return Err(TypeError::Internal {
            message: "约束求解未收敛".to_string(),
        });
    }

    // 检查未解决的必需约束
    for constraint in self.constraints.values() {
        if let Constraint::Unify {
            strength: ConstraintStrength::Required,
            ..
        } = constraint
        {
            return Err(TypeError::Internal {
                message: "仍有未解决的必需约束".to_string(),
            });
        }
    }

    Ok(self.substitution.clone())
}
}
```

主要求解循环展示了迭代约束传播方法，该方法通过约束解决和解决方案传播的系统过程驱动我们的约束求解器。该算法维护几个关键不变量，确保正确性和终止性。

6.7.4.1 约束选择策略

求解器使用智能约束选择来最大化求解成功：

约束强度提供主要排序标准，必需约束优先于首选约束，首选约束优先于弱约束。这确保在可选优化或指导提示之前解决基本类型检查要求。

每个约束中未知元变量的数量提供次要排序标准，包含较少未知数的约束获得更高优先级，因为它们更可能立即可解决。

约束类型也影响优先级，因为统一约束通常通过实例化出现在多个约束关系中的元变量来解决其他约束。

6.7.4.2 解决方案传播

当约束成功解决时，求解器在整个系统中传播解决方案：

替换应用确保新解决方案应用于系统中的所有剩余约束，可能简化它们或使它们能够解决。此步骤通过用其具体解决方案替换元变量在它们出现的任何地方来转换约束系统。

约束唤醒激活等待已解决的元变量的延迟约束，将先前被阻止的约束带回活动考虑以进行解决。此机制确保约束解决过程可以处理复杂的相互依赖，其中某些约束无法解决，直到其他约束提供必要的信息。

依赖更新修改依赖图以反映新解决的变量，从依赖跟踪中删除已解决的元变量，并更新用于约束选择的拓扑排序。此维护确保求解器的内部数据结构在解决方案累积时保持一致和高效。

6.7.5 依赖类型系统中的统一

依赖类型系统中的统一提出了超越在像System F 这样的简单类型系统中遇到的挑战。项和类型之间的相互依赖意味着统一类型可能需要解决未知项，而统一项可能生成对其类型的约束。

这种相互依赖创建了几个复杂性：

类型-项依赖：当统一$\Pi(x : A).Bx$ 与$\Pi(y : A').B'y$ 时，我们必须统一参数类型A 和A' 以及依赖结果类型Bx 和$B'y$。结果类型的统一取决于参数类型统一的解决方案。

元变量作用域管理：表示未知项的元变量必须尊重依赖类型的变量绑定结构。在特定绑定上下文中创建的元变量不能用引用该上下文之外的变量的项来实例化。

高阶元变量：在依赖类型系统中，元变量可以表示未知函数，导致高阶统一问题，其中我们必须解决未知函数级别的项，而不仅仅是未知基础项。

6.7.5.1 统一算法

核心统一算法处理使两个依赖类型相等的基本任务：

我们的统一算法支持几种模式：

结构统一：当两个项具有相同的头部构造函数时，统一通过递归统一子组件进行。

元变量实例化：当一侧是元变量时，我们创建一个将变量映射到另一项的替换，但需要进行发生检查。

高阶模式：像Miller 模式这样的高级模式使能够进行有限的高阶统一，这些统一仍然是可判定的。

6.7.5.2 元变量解决

元变量解决涉及一个系统的多步骤过程，确保正确性和一致性。解决方案查找首先检查元变量是否已经有来自先前约束解决的解决方案，避免冗余工作并确保正确利用现有解决方案。发生检查确保提议的解决方案不会通过验证元变量不会出现在其自己的解决方案项中来创建无限类型，防止创建会违反类型系统正确性的循环类型定义。

上下文验证验证解决方案尊重变量作用域要求，确保解决方案项不引用在元变量绑定点不在作用域中的变量。此检查对于维护依赖类型系统要求的词法作用

域规则至关重要。最后，解决方案记录将验证的解决方案添加到替换系统并在整个约束系统中传播它，更新所有引用新解决的元变量的约束。

6.7.6 高级约束模式

6.7.6.1 高阶统一和Miller 模式

高阶统一将一阶统一扩展到处理函数级别的未知数，其中元变量可以表示未知函数，而不仅仅是未知项。虽然一阶统一问"什么值使这些项相等？"，但高阶统一问"什么函数使这些应用相等？"

高阶统一的基本挑战在于其不可判定性。与总是以解决方案或失败终止的一阶统一不同，一般高阶统一可以无限期运行而无法得出结论。这种不可判定性源于构造任意复杂函数表达式的能力，这些表达式满足统一约束。

考虑高阶统一问题`?F a b = g (h a) b`。元变量`?F`表示一个未知的两个参数的函数。潜在解决方案包括$\lambda xy.g(hx)y$，但也包括更复杂的形式，如$\lambda xy.g(h(idx))y$，其中`id`是恒等函数。可能解决方案的搜索空间是无限的，使得终止无法保证。

Miller 模式限制通过在对高阶统一问题施加语法限制来解决这种不可判定性，这些限制恢复了可判定性，同时保留了显著的表达能力。Miller 模式具有形式`?M x... x= t`，其中必须满足几个关键条件。所有参数`x... x`必须是不同的绑定变量，确保模式表示适当的功能关系，参数之间没有重复或混淆。

元变量必须仅应用于变量而不是复杂项，保持"变量脊柱"属性，防止当元变量应用于任意表达式时可能出现的潜在解决方案的指数爆炸。项`t`不能包含元变量`?M`本身，防止会导致无限类型的发生检查违规。最后，抽象安全性要求出现在`t`中的所有自由变量必须出现在参数`x... x`中，确保解决方案可以适当地抽象模式变量。

这些限制确保Miller 模式统一问题在存在解决方案时具有唯一的最一般统一器，恢复了这个高阶统一片段的可判定性。

6.7.6.2 延迟约束解决

无法立即解决的复杂约束会被延迟，直到有更多信息可用：

延迟约束系统使求解器能够处理在依赖类型检查场景中出现的复杂模式。

6.7.7 错误处理和诊断

约束求解器提供全面的错误报告，帮助用户理解求解失败：

```
#![allow(unused)]
fn main() {
pub enum ConstraintError {
    UnificationFailure { left: Term, right: Term, reason: String },
    OccursCheck { meta_var: MetaId, term: Term },
    ScopeViolation { meta_var: MetaId, escaped_vars: Vec<String> },
    CircularDependency { cycle: Vec<MetaId> },
    UniverseInconsistency { constraint: UniverseConstraint },
}
}
```

每种错误类型提供关于为什么约束求解失败的特定诊断信息，使用户能够理解和解决潜在问题。统一失败呈现冲突的项以及为什么它们不能相等的详细解释，帮助程序员识别其代码中的类型不匹配和结构不兼容。发生检查违规识别会导致无限类型的提议解决方案的情况，捕获会违反类型系统正确性的递归类型定义。

作用域违规检测逃逸其预期词法作用域的变量，通常发生在元变量解决方案引用在解决方案绑定上下文中不可用的变量时。循环依赖识别无法解决的约束循环，其中约束以阻止任何进展的方式相互依赖，指示约束系统结构中的基本问题。宇宙不一致指示宇宙层次结构的违规，例如试图将大宇宙放在较小的宇宙内，这会损害类型系统的逻辑一致性。

约束求解器代表了构造演算实现中最复杂的组件之一，但它仍然比Coq和Lean 中的工业实现简单得多。通过仔细的算法设计和实现技术，我们实现了一个小而全的约束求解器，具备工业级特性，但代码量仅为Coq/Lean 的1% 以下，真正可读、可维护、可教学。

Bibliography

[1] Pierce, B. C. (2002). *Types and programming languages*. MIT press.

[2] Pierce, B. C. (Ed.). (2024). *Advanced topics in types and programming languages*. MIT press.

[3] Harper, R. (2016). *Practical foundations for programming languages*. Cambridge University Press.

[4] Jones, R., Hosking, A., & Moss, E. (2023). *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC.

[5] Milner, R. (1978). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*.

[6] Pottier, F., & Rémy, D. The Essence of ML Type Inference.

[7] Jones, M. P. (1999). Typing haskell in haskell. *Haskell workshop*, Vol. 7.

[8] Müller, M. (1998, August). Notes on HM (X).

[9] Odersky, M., Sulzmann, M., & Wehr, M. (1999). Type inference with constrained types. *Theory and practice of object systems*, 5(1), 35-55.

[10] Faxén, K. F. (2002). A static semantics for Haskell. *Journal of functional programming*, 12(4-5), 295-357.

[11] Zhao, J., Oliveira, B. C. d. S., & Schrijvers, T. (2019). A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference. *ICFP 2019*.

[12] Dunfield, J., & Krishnaswami, N. R. (2013). Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. *ICFP 2013*.

[13] Zhao, J., & Oliveira, B. C. d. S. (2022). Elementary Type Inference. *ECOOP 2022*.

[14] Bosman, R., Karachalias, G., & Schrijvers, T. (2023). No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System. *ITP 2023*.

[15] Bailey, C. Type Checking in Lean 4. https://ammkrn.github.io/type_checking_in_lean4/

[16] Xue, X., & Oliveira, B. C. d. S. (2024). Contextual Typing. *ICFP 2024*.