定理证明器的设计与实现

A Design and Implementation of Proof Assistant

KonjacSource

2025.5

重要信息

这篇文章是我的**本科**毕业论文, 受作者水平与时间所限, 其缺少原创性, 缺少对语言性质的分析, 还请读者谅解.

本文虽名为定理证明器的设计与实现, 但其实主要内容仅聚焦在归纳类型的实现上. 本来想添加繁饰的内容, 但精力与时间实在有限, 而且这部分已经了有很好的教程[1]. 与繁饰相比, 归纳类型的实现完全没有中文教程, 所有我觉得这篇文章还是有些意义的.

另外希望读者注意一点, 作者本科是物理专业, 不得不添加了一些关于类型论与编程语言理论的基础信息在第一章, 有经验的读者完全可以从第二章开始阅读. 同样是由于本人的专业问题, 文末装模做样, 画蛇添足地添加了类型论和物理的联系, 也请读者略过.

关于归纳类型的实现, 本文还缺少对极性的检查和终止性检查. 其实这两者十分简单 (相比于模式匹配的其他部分) , 读者可参考[2].

本文对应的代码实现是 ShiTT. 代码姑且可以正常运行, 但还存在一些严重的 bug, 本文完成后我已经意识到问题的所在, 正在计划进行重写. 我会考虑稍后写一篇更贴近实现的笔记.

感谢你的谅解与阅读, 五体投地, 感激涕零 🙏.

摘要

类型论是数学和编程语言理论的共同分支, 它一方面能为编程语言的设计提供指导，另一方面可以代替各类集合论作为数学基础。我们可以籍由类型论的逻辑规则设计编程语言，使得它有能力精确表达和证明数学命题，这避免了自然语言描述数学的歧义性，这类编程语言被称为定理证明器。本文将介绍定理证明器的设计并给出一个具体的实现，探讨类型论在数学研究中的重要性。长期以来，带下标的归纳类型(Indexed Inductive Types)缺少一些指南性的实现教程, 本文是这方面极少的中文资料。

关键词: 定理证明器; 编程语言; 形式化验证; 类型论; 逻辑学

ABSTRACT

Type theory is a shared branch of mathematics and programming language theory.On one hand, it provides guidance for the design of programming languages.On the other, it can serve as a foundation for mathematics in place of various set theories.By designing programming languages based on the logical rules of type theory, we can precisely express and prove mathematical propositions, thus avoiding the ambiguities of natural language descriptions.Such programming languages are called proof assistants.This article introduces the design and implementation of proof assistants and explores the significance of type theory in mathematical research.For a long time, there has been a lack of practical guides for implementing Indexed Inductive Types.This article is one of the few, maybe only, Chinese resources on this topic.

Keywords: proof assistant; programming language; formal verification; type theory; logic

目录

定理证明器是指一类能够机械性地检查数学证明的软件, 对于一些复杂的数学证明, 这种验证是必要的. 本文将说明定理证明器背后的原理与细节, 并完成一个定理证明器的设计与实现.

定理证明器的需求主要来自于两个看起来无关的领域, 纯数学和软件工程. 在前者的视角里, 定理证明器可以看作一个逻辑框架, 我们在这个框架中演绎数学. 而对于后者, 我们先将一段程序的源码看作一个语法结构输入定理证明器, 再在定理证明器中建模对应语言的语义, 最后使用定理证明器对该程序进行分析并验证相关性质, 这种程序验证方法被称为形式化验证, 有很多对安全性要求较高的项目使用了这个方法进行验证.

在数学领域, Lean4[3] 和 Rocq[4] (原 Coq) 已被广泛地用于形式化数学, 例如 Lean4 的数学库 Mathlib[5] 已经基本涵盖了本科数学的所有内容, 并且在前沿数学中发挥了出色的作用. 这已经证明了定理证明器可以在数学研究中发挥重要作用.

1 类型论

我们在设计定理证明器时, 希望追寻一个坚实的逻辑基础, 并在该基础之上对数学命题进行形式化的演绎推理以此保障数学的意义不被自然语言的模糊性所干扰. ZFC 集合论[6] 作为最被数学家所接受的数学基础可以完成这项任务. 但另一方面, Russell 在试图解决他发现的悖论时提出的类型论则是另一个很好的选择(本文将不介绍原始版本的 Russell 类型论[7]).

当前大多数的定理证明器都是基于依值类型论的, 具体原因将在之后说明, 我们在此先对基础的类型论进行一些简要的介绍.

1.1 λ演算

Church 提出的λ演算[8] 可以看作最简单的一个类型论例子, 我们在这里介绍一下它的构造. 首先是它的语法:

$$M, N ::= x \mid \lambda x\ .\ M \mid MN \tag{1.1}$$

这是在说每个λ项(用字母 M, N 表示)可以是三种格式, 要么是一个变量 x, 要么是一个λ表达式 $\lambda x\ .\ M$, 或者是两个λ项的组合.

λ演算的语义的核心是 β 规约, 这条规则可记作:

$$(\lambda x\ .\ M)N \equiv M[N/x] \tag{1.2}$$

其中 $M[N/x]$ 意为将 M 中所有自由出现的变量 x 替换成 N. 我们可以认为每一个λ表达式都对应一个函数, 比如我们可以写这样一个函数 $\lambda x\ .\ x+1$ 用来表示一个把 x 映射到 $x+1$ 的函数. 这样根据 β 归约, 我们可以将 $(\lambda x\ .\ x+1)\ 3$ 规约到 4.

另一方面, λ演算作为一个图灵完备的形式语言有非常强大的计算能力. 我们可以在λ演算中定义自然数和其他常见数据结构以及在这些结构上的各种操作, 简言之λ演算可以看作一个最简单的编程语言.

1. 2 简单类型λ演算

λ演算可以看作只有一个类型的类型论, 所有项都被看作是同一个类型, 这意味着我们可以构造这样一种项 $(\lambda x.\ x) + 1$, 我们在对一个函数做加一的操作, 这当然是没有意义的. 我们可以这样解决这种问题: 给每个项都赋予一个类型, 并只允许类型合适的时候进行函数应用. 这样我们可以自然地提出简单类型λ演算(STLC)[9] , 在 STLC 中, 每个项都有一个类型, 并且项和其中的每一个子项都有对应的语境, 一个语境是从变量到类型的映射, 常记作 Γ. 我们用 $\Gamma \vdash M : A$ 来表示在语境 Γ 下, 项 M 的类型是 A.

该符号的具体定义由一系列规则给出, 我们先来看第一个

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{T-Var} \tag{1. 3}$$

式中横线可看作逻辑蕴含, 该式意义为: 如果在语境 Γ (看作一个映射) 中x 被映射到了类型 A, 那么 $(\Gamma \vdash x : A)$ 成立. 最右边的 "T-Var" 是这个规则的名字.

第二条规则是 "T-App"

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B}\ \text{T-App} \tag{1. 4}$$

这里引入了一个新的类型 $A \to B$, 这表示所有从类型 A 到类型 B 的函数构成的类型. 这条规则很好的捕捉了函数类型的意义, 如果有一个类型为 $A \to B$ 的函数, 且有一个类型为 A 的值, 那将二者应用就能得到一个类型为 B 的值.

最后一个规则指出了我们如何引入函数.

$$\frac{\{(x : A)\} \cup \Gamma \vdash M : B}{\Gamma \vdash (\lambda x : A\ .\ M) : B}\ \text{T-Abs} \tag{1. 5}$$

首先注意到的一点是λ表达式给引入的变量标注了类型. 横线上面的前提是在说: 在一个扩充了的语境下, M 的类型为 B. 这很好理解, 因为λ表达式引入了一个新的变量 x, 并且这个 x 是有可能在 M 中出现的, 所以我们当然希望在检查 M 的类型时能够在一个有 x 的语境中检查. ($\{(x : A)\} \cup \Gamma$ 也常记作 $\Gamma, x : A$.

我们可以通过添加更多规则扩展 STLC 来得到一些更具表现力的类型系统.

比如自然数,

$$\frac{}{\Gamma \vdash \text{zero} : \text{nat}} \quad \frac{}{\Gamma \vdash \text{suc} : \text{nat} \to \text{nat}}$$
$$\frac{}{\Gamma \vdash \text{nat-elim}_A : A \to (\text{nat} \to A \to A) \to \text{nat} \to A} \tag{1. 6}$$

其中 zero 代表数字 0, suc 代表后继函数, 于是我们可以用 suc zero 来代表数字 1, suc (suc nat) 来代表数字 2, 以此类推.

关于 nat-elim 还有一个归约规则需要引入,

$$\begin{aligned} &\text{nat-elim}_A\ a\ f\ \text{zero} \equiv a \\ &\text{nat-elim}_A\ a\ f\ (\text{suc}\ n) \equiv f\ n\ (\text{nat-elim}\ a\ f\ n) \end{aligned} \tag{1. 7}$$

我们称 zero 和 suc 是类型 nat 的构造子, 它们指出了如何构造一个 nat 类型的元素; nat-elim 称为 nat 的消除子, 它指明了我们该如何使用一个 nat 的元素.

我们可以通过 nat-elim 定义全部常用自然数操作, 以加法为例,

$$\text{add} \coloneqq \lambda mn.\ \text{nat-elim}\ n\ (\lambda m' s.\ \text{suc}\ s)\ m \tag{1. 8}$$

再比如这样两个类型, 我们叫他们 $\top$ 和 $\bot$, 他们有如下相关规则

$$\frac{}{\Gamma \vdash \text{tt} : \top} \qquad \frac{\Gamma \vdash b : \bot}{\Gamma \vdash \text{exfalso}\ b : A} \tag{1. 9}$$

$\top$ 有唯一一个元素 tt, 而 $\bot$ 不存在任何元素, 但有一个 exfalso 函数可以从一个 $\bot$ 得到任何类型, 读者可以联想到在集合论中, 空集到任何集合的映射总是存在的.

大多类型都需要这种分别引入构造子和消除子的方式来定义, 这些类型中有一大类被称为归纳类型, 因为消除子的类型规则恰好能对应数学归纳法. 本文的后半部分内容将围绕归纳类型展开.

对于 STLC, 我们可以把类型系统和逻辑系统联系起来.

命题逻辑是这样一种逻辑系统, 它允许用命题之间的运算符来连接命题, 以此构造新的命题, 比如 $A \wedge B$, $A \vee B$ 和 $A \Rightarrow B$, 分别表示“与”, “或” 以及 “蕴含”(“非”可以用 $A \Rightarrow$ False表示, 其中 False 为恒假命题).

我们可以用刚才用过的符号来描述命题逻辑, 考虑这样一个判断 $\gamma \vdash A$, 用来表示在假设集 γ 下, 命题 A 能被证明. 其中 γ 里面有一系列被预先假设的命题, 于是我们有,

$$\frac{A \in \gamma}{\gamma \vdash A} \tag{1. 10}$$

对于蕴含, 我们可以这样定义,

$$\frac{\gamma \cup \{A\} \vdash B}{\gamma \vdash A \Rightarrow B} \quad \frac{\gamma \vdash A \Rightarrow B \quad \gamma \vdash A}{\gamma \vdash B} \tag{1. 11}$$

这也很好理解, $A \Rightarrow B$ 能够成立是因为我们能在假设 A 成立的前提下证明 B. 而第二个规则则是说, 如果我们有 $A \Rightarrow B$ 的证明, 并且有 A 的证明, 那么我们就得到了一个 B 的证明.

这看起来很熟悉, 这三条规则恰好能对应我们刚才 STLC 中的 T-Var, T-Abs 和 T-App. 这个对应关系被称为 Curry-Howard 对应, 它指出了类型系统和逻辑系统的等价性, 我们可以自然地想到: 类型检查器[1]就是定理证明器.

这样 $\bot$ 就可对应恒假命题, 我们说一个语言具有**一致性**, 是指空语境下不存在 $\bot$ 的元素, 对于逻辑系统和定理证明器, 一致性是必要的, 而对于一般编程语言, 一致性无关紧要[2].

STLC 作为一个最简单的类型系统被应用于很多古老的编程语言, 类型检查的实现也相当简单, 基本可以直接把类型规则机械地转换成代码.

1. 3 系统 F

STLC 的表达能力过于羸弱, 不仅作为逻辑系统在数学上毫无用处, 作为编程语言在编程中也处处碰壁. 例如我们想定义一个将两个函数复合的函数 $\lambda f\ .\ \lambda g\ .\ \lambda x\ .\ f(g\ x)$, 这样的函数在 STLC 里没法被很好地类型化, 其原因是缺乏一个允许我们对类型进行泛化的机制, 所以很多编程语言引入了泛型, 这些语言的类型系统就可以看作系统 F[9,10] 的某些限制版本.

系统 F 在 STLC 的基础上添加了一种新的类型 $\forall X.\ T$, 称为全称类型 其中 X 被称作类型变量, 用来表示泛化的类型, T 作为一个类型表达式允许 X 在其中自由出现. 我们有如下规则来构造和消除全称类型的元素,

$$\frac{\Gamma, X : \mathcal{U} \vdash M : T}{\Gamma \vdash \Lambda X.\ M : \forall X.\ T} \quad \frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash f : \forall X\ .\ T}{\Gamma \vdash F\ A : T[A/X]} \tag{1. 12}$$

其中 $\mathcal{U}$ 是所有类型构成的类型[3], 也称作宇宙.

有了这样的结构, 我们可以定义复合函数,

$$\begin{aligned} &\text{comp} : \forall ABC\ .\ (B \to C) \to (A \to B) \to A \to C \\ &\text{comp}\ A\ B\ C\ f\ g\ x \coloneqq f\ (g\ x) \end{aligned} \tag{1. 13}$$

这相当于说,

$$\text{comp} \coloneqq \Lambda ABC.\ \lambda f : B \to C.\ \lambda g : A \to B.\ \lambda x : A.\ f\ (g\ x) \tag{1. 14}$$

[1] 即检查一个项是否能通过上面提出的一系列类型规则被赋予一个类型的机械性算法.
[2] 实际上如果需要某个类型论具有图灵完备性, 那么它就不会是一致的.
[3] 在系统 F 中 $\mathcal{U}$ 只是一个记号, 并不是一个真正的类型.

但是在使用这个函数的时候, 我们会遇到一些不便, 例如我们试图复合两个 suc 得到一个递增 2 的函数,

$$\mathrm{add2} := \mathrm{comp}\ \mathrm{nat}\ \mathrm{nat}\ \mathrm{nat}\ \mathrm{suc}\ \mathrm{suc} \tag{1. 15}$$

这太繁琐了, comp 的前三个参数明明可以通过后两个参数推得, 我们在这里却还要明确的写出. 在编程语言领域, 人们提出一些算法来自动填充必要的类型信息, 这被称为类型推导. 但遗憾的是, 系统 F 的完全类型推导算法是不存在的[11], 即并不存在一个算法能过填充系统 F 中所有省略的类型参数, 用户仍然需要手动添加部分类型信息. 这样我们有两个选择, 削弱系统 F 使得可以进行完整的类型推导, 或者抛弃完整的类型推导只使用一个部分的类型推导, 即尽可能推导类型推导不出时报错给用户. 对于编程语言领域, 第一个选择是可行的, 于是人们提出了 Hindley–Milner 类型系统(HM)[12], 该系统要求每个泛型函数都要有一个名字, 并且泛型函数不能接受其他泛型函数作为参数, HM 在每个泛型函数被使用的时候插入类型变量并试图求解. 这个系统被应用在了许多函数式编程语言中, 包括 Haskell[13] 和 OCaml[14], 在这些语言中, 程序员不需要进行任何类型 标注也能保证程序的类型安全.

但另一方面, 系统 F 的表达能力仍然不足以满足定理证明的需求, 我们不能接受它的进一步削弱, 所以只能实现一个不完全的类型推导, 这样的类型推导过程被称为繁饰(elaboration). 我们将在之后进一步讨论繁饰的具体过程.

作为逻辑系统, 系统 F 在命题逻辑的基础上允许我们对命题进行全称泛化. 仅允许对命题的全称量词显然是不够的, 我们可以设想这样一种类型论, 它允许对任何值进行泛化, 并把类型看作某种特殊的值, 这样我们的新系统就覆盖了原本系统 F 的所有功能. 于是我们可以得到依值类型论.

1. 4 依值类型论

依值类型最重要的特点是Π-类型的引入以及值的判定相等性(这点之后会提及), 关于Π-类型, 我们有,

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma, x : A \vdash B : \mathcal{U}}{\Gamma \vdash (x : A) \to B : \mathcal{U}} \qquad \frac{\Gamma, x : A \vdash M : B[x/y]}{\Gamma \vdash \lambda x\ .\ M : (y : A) \to B}$$
$$\frac{\Gamma \vdash f : (x : A) \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[a/x]} \tag{1. 16}$$

Π-类型也叫依赖函数类型, 有时也记作 $\prod_{x:A} B$, 这是因为它看起来像是一系列类型 $B[x]$ 的积, 它允许参数的值在输出类型中出现, 这是依值一词的意义.

利用Π-类型, 我们可以有如下定义,

$$\begin{aligned} A \to B &:= (\_ : A) \to B \\ \forall X.\ T &:= (X : \mathcal{U}) \to T \end{aligned} \tag{1. 17}$$

所以我们可以看出 STLC 中的函数类型和系统 F 里的全称类型实际上是Π-类型的特殊情形. 这种能够被推广的构造在类型论中很常见.

关于 $\mathcal{U}$ 类型, 在系统 F 中, 它只用来说明一个某个类型表达式是一个合法的类型, 但在依值类型论中它必须被赋予更多意义, 根据Π-类型的规则, 在 $(x : A) \to B$ 中的 $A : \mathcal{U}$, 但是如果我们想让 式 (1. 17) 中对 $\forall$ 的定义成立就必须有 $\mathcal{U} : \mathcal{U}$, 这将给出如下的规则,

$$\Gamma \vdash \mathcal{U} : \mathcal{U} \tag{1. 18}$$

这条规则被称为 Type-in-Type. 如果读者熟悉 Russell 悖论, 可能看到这条规则可能会觉得有问题, 这确实是的, 如果往类型论中加入 Type-in-Type, 确实会导致一个类似 Russell 悖论的悖论[15], 我们在之后会指出补救方法, 现在我们暂且继续使用 Type-in-Type.

现在我们考虑这样的定义,

$$\begin{aligned} &\mathrm{t} : \mathrm{nat} \to \mathcal{U} \\ &\mathrm{t}\ x \coloneqq \mathrm{nat} \\ &\\ &\mathrm{f} : \mathrm{t}\ \mathrm{zero} \to \mathrm{nat} \\ &\mathrm{f}\ x \coloneqq x \end{aligned} \tag{1. 19}$$

我们需要检查 f 定义右手端的 x 的类型是 nat, 但是我们只有 $x : \mathrm{t}\ \mathrm{zero}$, 所以我们需要一个机制来允许把 x 赋型为 nat,

$$\frac{\Gamma \vdash A \equiv B : \mathcal{U} \quad \Gamma \vdash M : A}{\Gamma \vdash M : B} \tag{1. 20}$$

其中 $\Gamma \vdash A \equiv B : \mathcal{U}$ 表示在 Γ 语境下, A 与 B 判定相等. 关于判定相等还有许多规则, 但在大多数类型论中, 判定相等都可被定义为正规形式[4]之间的等价性, 该性质是类型检查算法的基础之一.

现在的类型论已经有了很多描述数学的能力, 包括现在我们可以定义相等性,

$$x =_A y \coloneqq (P : A \to \mathcal{U}) \to P\ x \to P\ y \tag{1. 21}$$

这是在说, x 等于 y 当且仅当对于所有 x 能满足的性质 P, y 也满足. 这被称为莱布尼茨相等性, 此名是因为该定义体现了莱布尼茨提出的不可分者同一性原理.

利用该定义, 我们可以证明等式的三个基本性质: 自反, 对称以及传递,

[4] 正规形式是指一个语境下项能被归约到的最简形式, 这要求语言有正规性, 即正规形式对每个类型合法的项都存在.语义正规性是指类型论的语义中存在这样一种正规形式可以使得这种形式可以表征判定相等性.

$$
\begin{aligned}
&\text{refl} : (A : \mathcal{U})\ (x : A) \to x =_A x \\
&\text{refl}\ A\ x\ P\ p \coloneqq p \\
\\
&\text{symm} : (A : \mathcal{U})\ (x\ y : A) \to x =_A y \to y =_A x \\
&\text{symm}\ A\ x\ y\ e \coloneqq e\ (\lambda i.\ \ i =_A x)\ (\text{refl}\ A\ x) \\
\\
&\text{trans} : (A : \mathcal{U})\ (x\ y\ z : A) \to x =_A y \to y =_A z \to x =_A z \\
&\text{trans}\ A\ x\ y\ z\ e_1\ e_2 \coloneqq e_2\ (\lambda i.\ \ x =_A i)\ e_1
\end{aligned}
\tag{1. 22}
$$

关于 trans 函数, 注意到右式中有 $e_1 : (P : A \to \mathcal{U}) \to P\ y \to P\ z$, 我们只需令 $P = \lambda i\ .\ x =_A i$, 这样便有 $x =_A y \to x =_A z$, 这恰好能将 e_1 转换为目标类型.

我们还可以从逻辑的角度出发, 考虑存在量词在依值类型论中代表什么, 存在量词 $\exists x \in A, B$ 有两个要素, 集合 A 和一个允许 x 自由出现的命题 B, 那考虑类型论中的对应物自然需要这两个要素, 考虑到 Curry-Howard 对应中所说的类型即命题, 我们可以写出,

$$
\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma, (x : A) \vdash B : \mathcal{U}}{\Gamma \vdash (x : A) \times B : \mathcal{U}}
\tag{1. 23}
$$

一个存在量词的命题 $(x : A) \times B$ 该如何证明? 构造性地讲, 我们需要给出一个 $a : A$, 并证明 $B[a/x]$, 这可以导出如下引入规则,

$$
\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : (x : A) \times B}
\tag{1. 24}
$$

引入规则需要 a 和 b 两个要素, 消除规则就可以定义为取出这两个要素的操作,

$$
\frac{\Gamma \vdash e : (x : A) \times B}{\Gamma \vdash \text{fst}\ e : A} \qquad \frac{\Gamma \vdash e : (x : A) \times B}{\Gamma \vdash \text{snd}\ e : B[\text{fst}\ e/x]}
\tag{1. 25}
$$

并且有如下判定等价关系,

$$
\frac{\Gamma \vdash (a, b) : (x : A) \times B}{\Gamma \vdash \text{fst}\ (a, b) \equiv a : A} \qquad \frac{\Gamma \vdash (a, b) : (x : A) \times B}{\Gamma \vdash \text{snd}\ (a, b) \equiv b : B[a/x]}
\tag{1. 26}
$$

可以看到, 存在量词的类型论对应很像编程语言中的序对, 实际上, 存在类型也叫依赖序对或者也叫 Σ-类型, 并记作 $\sum_{x:A} B$, 这是因为它看起来像是一系列类型 $B[x]$ 被 x 索引的和. 当 B 中不含有 x 的时候, 我们可以直接将其简记作 $A \times B$, 这就是普通的序对类型, 读者可以看出该类型在系统 F 中也可以定义.

1. 5 Martin-Löf 类型论 (MLTT)

我们现在考虑以下两个问题, 首先第一点是相等类型能不能定义成某种归纳类型, 既然它是归纳类型, 它的构造子和消除子应该是什么? 第二点是我们上节遗留的 Type-in-Type 问题.

对于第一个问题, 我们可以构造如下类型,

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A}{\Gamma \vdash \mathrm{Id}_A\ x\ y} \qquad \frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash x : A}{\Gamma \vdash \mathrm{refl}_x : \mathrm{Id}_A\ x\ x}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash m : (x : A) \to P\ x\ x\ \mathrm{refl}_x \\ \Gamma \vdash P : (a\ b : A) \to \mathrm{Id}_A\ a\ b \to \mathcal{U} \\ \Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash p : \mathrm{Id}_A\ a\ b\end{array}}{\Gamma \vdash \mathrm{J}_A\ P\ m\ a\ b\ p : P\ a\ b\ p} \tag{1. 27}$$

其中 Id 类型被用来表示相等, 这里 refl 是自反性, 与莱布尼茨相等性不同, Id 类型的出发点是 refl, 即用同一性直接表示相等. 这里 J 是 Id 类型的消除子, 它有一个非常复杂的类型, 我们逐步阐释一下它的意义, 首先参数 $P : (a\ b : A) \to \mathrm{Id}_A\ a\ b \to \mathcal{U}$ 是我们消除的动机, 即: 我们要把一个等式转换成什么类型. m 是说我们需要给出对于每一个 $x : A$, $P\ x\ x\ \mathrm{refl}_x$ 的定义, 然后通过 J, 我们将能把 $P\ x\ x\ \mathrm{refl}_x$ 上的值转换成任何 $P\ a\ b\ p$ 上. 这里的重点是, 当我们有 $p : \mathrm{Id}_A\ a\ b$ 时, 我们就能将 a 和 b 看作同一个值, 并可以把 p 看作 refl.

通过 Id 类型和 J 规则, 我们可以证明刚才说明的等式性质, 在此不多赘述, 之后我们会介绍一个更方便使用的 J 的写法.

现在我们考虑如何避免 Type-in-Type, Martin-Löf[16] 给出的方法就是建立一系列不同的 $\mathcal{U}$ 类型, 并让它们依次归属, 即,

$$\frac{i \in \mathbb{N}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \tag{1. 28}$$

其中 i 是元层面[5]的自然数, 而不是类型论层面的. 然后我们需要小心的修改每一个类型的规则, 使得我们不会构造出从高阶宇宙到低阶宇宙的单射, 最重要的一个修改是关于 Π-类型的[6],

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, (x : A) \vdash B : \mathcal{U}_j}{\Gamma \vdash (x : A) \to B : \mathcal{U}_{\max(i,j)}} \tag{1. 29}$$

MLTT 将作为我们之后叙述的基础.

2 数据类型

很多编程语言都会提供一些自定义类型的工具, 比如 C 的 `struct`, Java 的 `class`, 我们会希望我们的语言也有这种功能, 而不是在元层面添加一些规则来引入新类型[7], 这样能显著提高语言的表达能力.

[5] 元层面是指我们用来描述类型论的语言, 这些大横线定义的规则都是元层面的.
[6] 这里采用的定义使得 MLTT 称为直谓性类型论, 这里我们不深入讨论非直谓性的类型论.
[7] 如果每次都通过添加规则来引入新类型, 我们必须人为地确定每个新类型的性质来保证整个语言的安全性.

2. 1 ADT (Algebraic Data Types)

在函数式编程语言中, 我们会推广 C 风格的 `struct`, 得到所谓的代数数据类型 (Algebraic Data Types), 我们先试着在系统 Fω[8]上增加这套工具, 我们引入一类新的语法表示类型定义,

$$\begin{aligned}
&\text{data_decl} ::= \text{'data'}\ \ \text{data_name}\ \ \ \text{ty_arg_binders}\ \text{':='}\ \text{constr}\\
&\text{constr} ::= \text{'.'} \mid \text{con_name}\ \ \{\text{type}\}\\
&\text{ty_arg_binders} ::= \overline{\text{'('ty_arg_name':'}\ \text{kind}\ \text{')'}}\\
&\text{kind} ::= \mathcal{U} \mid \text{kind} \to \text{kind}
\end{aligned} \tag{2. 1}$$

这里我们使用上划线表示多个语法对象构成的序列. , 如下是一些合乎语法的定义,

$$\begin{aligned}
&\text{data Bool} := \text{true} \mid \text{false}\\
&\text{data Nat} := \text{zero} \mid \text{suc Nat}\\
&\text{data Tup}\ (a\ b : \mathcal{U}) := \text{MkTup}\ a\ b\\
&\text{data Top} := \text{tt}\\
&\text{data Bot} := .\\
&\text{data Option}\ (a : \mathcal{U}) := \text{null} \mid \text{some}\ a\\
&\text{data List}\ (a : \mathcal{U}) := \text{nil} \mid \text{cons}\ a\ (\text{List}\ a)\\
&\text{data BinTree}\ (a : \mathcal{U}) := \text{leaf} \mid \text{branch}\ a\ (\text{BinTree}\ a)\ (\text{BinTree}\ a)
\end{aligned} \tag{2. 2}$$

我们现在直觉上说明一下该语法的语义, 首先对于 Bool 类型, data 是一个关键字用于开始声明类型, Bool 是该类型的名字, 后面的 true | false 代表 Bool 有且只有两个构造子 true 和 false. 对于 Nat 类型, 它有两个构造子 zero 和 suc : Nat → Nat. 这恰好是我们之前手动引入的 nat 类型. Bot 是一个无构造子的类型, 对应我们先前的 ⊥.

Tup, Option, List 以及 BinTree 是含有类型参数的类型, 称为泛型, 我们有, Tup : $\mathcal{U} \to \mathcal{U} \to \mathcal{U}$, Option : $\mathcal{U} \to \mathcal{U}$. Option 代表一个可选值, 要么给出这个值, 要么返回 null[9] ; List 和 BinTree 分别相当于编程语言中常见的单链表和二叉树, 对于列表, 我们经常使用 [] 表示 nil, 用 $x :: l$ 表示 cons $x\ l$.

只有构造子还不够, 我们还需要一个消除子, ADT 的消除子被称为模式匹配,

$$\begin{aligned}
&\text{term} ::= ... \mid \text{'match'}\ \ \text{term}\ \text{'with'}\ \overline{\text{'|'}\ \text{pattern}\ \text{'}\Rightarrow\text{'}\ \text{term}}\\
&\text{pattern} ::= \text{var_name} \mid \text{con_name}\ \overline{\text{pattern}}
\end{aligned} \tag{2. 3}$$

[8] 系统 Fω 是在系统 F 的基础上允许对任意泛型进行全称化, 即允许 $\forall x : K.T$, 其中 K 是 kind.

[9] 当前大多数现代语言(包括 Rust, Kotlin 以及较新版本的 Java)都通过这种方式避免 Tony Hoare 的 "十亿美元错误"[17].

藉由模式匹配, 我们可以将构造子里的值取出来, 再配上递归函数定义, 我们可以实现消除子的所有功能(也可以定义消除子本身).

$$\begin{aligned}\text{add } m\ n \coloneqq \text{match } m \text{ with}\\ \mid \text{zero} \Rightarrow n\\ \mid \text{suc } m' \Rightarrow \text{suc } (\text{add } m'\ n)\end{aligned} \tag{2. 4}$$

我们也经常写作,

$$\begin{aligned}&\text{add} : \text{Nat} \to \text{Nat} \to \text{Nat}\\ &\text{add zero}\quad\ \ n \coloneqq n\\ &\text{add } (\text{suc } m)\ n \coloneqq \text{suc } (\text{add } m\ n)\end{aligned} \tag{2. 5}$$

但是要注意, 不对 ADT 加以限制和允许任意的递归都会破坏语言的一致性, 我们稍后会说明如何限制 ADT 和避免无限递归.

ADT 和模式匹配的一个奇妙特点是它可以随着背景类型论的不同采用不同的形式, 并且能很方便地推广. 例如一个很简单的推广是允许两个 ADT 相互引用, 这被称为互归纳类型, 互归纳类型的消除子的形式十分复杂, 但是如果使用 ADT 版本的定义, 我们完全不需要对模式匹配做任何改动, 只需要允许互递归.

2. 2 GADT (Generalized Algebraic Data Types)

对于一个泛型的 ADT, 比如

$$\text{data F } a \coloneqq \text{C1 } a \mid \text{C2 Bool} \mid \text{C3} \tag{2. 6}$$

会生成如下三个构造子$\text{C1} : \forall a, \text{F } a$, $\text{C2} : \forall a, \text{Bool} \to \text{F } a$, $\text{C3} : \forall a, \text{F } a$. 注意到两点, 泛型上的参数会变成构造子上的类型参数, 并且每个构造子的返回类型都是 F a. GADT 推广了这点, 允许每个构造子的返回的类型参数不同. 这使得我们需要更新一下语法,

$$\begin{aligned}&\text{gadt\_decl} \coloneqq \text{'data'}\ \ \text{data\_name ':'}\ \text{kind}\ \ \text{'where'}\ \{\text{clause}\}\\ &\text{clause} \coloneqq \text{'|'}\ \text{con\_name ':'}\ \text{type}\end{aligned} \tag{2. 7}$$

一个 GADT 的典型应用是用来编码类型安全的表达式, 我们在附录中给出实际可运行的例子.

$$\begin{aligned}&\text{data Expr} &&: \mathcal{U} \to \mathcal{U} \text{ where}\\ &\mid \text{Lit} &&: (t : \mathcal{U}) \to t \to \text{Expr } t\\ &\mid \text{Add} &&: \text{Expr Nat} \to \text{Expr Nat} \to \text{Expr Nat}\\ &\mid \text{IsZero} &&: \text{Expr Nat} \to \text{Expr Bool}\\ &\mid \text{IfThenElse} &&: (t : \mathcal{U}) \to \text{Expr Bool} \to \text{Expr } t \to \text{Expr } t \to \text{Expr } t\end{aligned} \tag{2. 8}$$

这个声明将生成一个类型 $\text{Expr} : \mathcal{U} \to \mathcal{U}$ 和四个构造子, 注意虽然 GADT 允许返回不同的类型参数, 但是最终返回的类型最外层必须是被声明的类型, 此例中为 Expr.

我们可以写出该类型的一个解释器,

$$
\begin{aligned}
&\text{eval} : (t : \mathcal{U}) \to \text{Expr}\ t \to t \\
&\text{eval}\ t\ e = \ \text{match}\ e\ \text{with} \\
&\qquad | \ \text{Lit}\ v \Rightarrow v \\
&\qquad | \ \text{Add}\ a\ b \Rightarrow \text{add}\ (\text{eval}\ \text{Nat}\ a)\ (\text{eval}\ \text{Nat}\ b) \\
&\qquad | \ \text{IsZero}\ n \Rightarrow \\
&\qquad\quad (\text{match}\ \text{eval}\ \text{Nat}\ n\ \text{with} \\
&\qquad\qquad | \ \text{zero} \Rightarrow \text{true} \\
&\qquad\qquad | \ \text{suc}\ \_ \Rightarrow \text{false}) \\
&\qquad | \ \text{IfThenElse}\ b\ l\ r \Rightarrow \\
&\qquad\quad (\text{match}\ \text{eval}\ \text{Bool}\ b\ \text{with} \\
&\qquad\qquad | \ \text{true} \Rightarrow \text{eval}\ t\ l \\
&\qquad\qquad | \ \text{false} \Rightarrow \text{eval}\ t\ r)
\end{aligned}
\tag{2. 9}
$$

这里要注意到在第一个中, 由于 $\text{Lit}\ v : \text{Expr Nat}$, 所以我们应当在这个分支中有 $t : \text{Nat}$, 第二和第三个分支同理, 这使得模式匹配的语义相比先前需要加以修改, 具体的修改将在稍后提及.

2. 3 归纳类型

我们可以定义如下类型,

$$
\begin{aligned}
&\text{data}\ \text{Eq} : (x\ y : \mathcal{U}) \to \mathcal{U}\ \text{where} \\
&\qquad | \ \text{Refl} : (x : \mathcal{U}) \to \text{Eq}\ x\ x
\end{aligned}
\tag{2. 10}
$$

观察构造子 Refl 的类型, 可见, $\text{Eq}\ x\ y$ 元素存在当且仅当 x 就是 y, 这看起来很像 Id 类型, 但是在系统 Fω 的背景下, 这个相等性并没有什么作用. 我们现在试着将其推广到依值类型.

GADT 的类型参数必须是一个类型, 而在依值类型中可以进一步推广为任意的层级的值, 这样就能得到所谓的归纳类型. 而我们可以写出如下定义,

$$
\begin{aligned}
&\text{data}\ \text{Id} : (A : \mathcal{U})(x\ y : A) \to \mathcal{U}\ \text{where} \\
&\qquad | \ \text{refl} : (A : U)(x : A) \to \text{Id}\ A\ x\ x
\end{aligned}
\tag{2. 11}
$$

这是 Eq 类型的直接推广, 该类型几乎等价于[10]我们先前手动引入的 Id 类型.

[10] 不是完全等价是因为我们这里还没有考虑 K 规则.

利用模式匹配, 我们可以更简单地完成一些证明, 例如如果使用 J 来证明相等的对称性, 我们需要,

$$\begin{aligned}&\text{symm} : (A : \mathcal{U})\ (x\ y : \mathcal{U}) \to \text{Id}_A\ x\ y \to \text{Id}_A\ x\ y\\&\text{symm}\ A\ x\ y\ e \coloneqq \text{J}_A\ (\lambda a\ b\ p\ .\ \text{Id}_A\ b\ a)\ (\lambda x\ .\ \text{refl}_x)\ x\ y\ e\end{aligned} \tag{2. 12}$$

而如果使用模式匹配, 我们有,

$$\begin{aligned}&\text{symm} : (A : \mathcal{U})\ (x\ y : \mathcal{U}) \to \text{Id}_A\ x\ y \to \text{Id}_A\ x\ y\\&\text{symm}\ A\ x\ y\ (\text{refl}\ A'x') \coloneqq \text{refl}\ A'x'\end{aligned} \tag{2. 13}$$

简单多了, 这是如何做到的? 我们观察 $\text{refl}\ A'\ x'$ 的类型为 $\text{Id}_{A'}\ x'\ x'$, 而 e 的类型是 $\text{Id}_A\ x\ y$, 比较二者我们能得到 $A' = A, x' = x = y$, 这样右手边所需要的类型 $\text{Id}_A\ x\ y$ 就可以看成 $\text{Id}_{A'}\ x'\ x'$, 所以我们此时只需给出 $\text{refl}\ A'\ x'$ 即可.

归纳类型非常强大, 而且还能继续随着不同的类型论进一步推广, 比如在同伦类型论[18] 中, 我们可以定义所谓的高阶归纳类型, 这允许我们在类型论中定义几何对象.

2. 4 极性

先前我们提到过, 不对 ADT 加以限制和允许任意的递归都会破坏语言的一致性, 我们现在来说明这一点,

考虑如下类型,

$$\begin{aligned}&\text{data Fix}\ (f : \mathcal{U} \to \mathcal{U}) : \mathcal{U}\ \text{where}\\&\quad \mid \text{fix} : f(\text{Fix}\ f) \to \text{Fix}\ f\end{aligned} \tag{2. 14}$$

有了这个类型, 我们可以构造任意递归,

$$\begin{aligned}&\text{any} : (a : \mathcal{U}) \to a\\&\text{any}\ a = \ f\ (\text{fix}\ \ (\lambda x.\ x))\\&\quad \text{where}\ \ f : \text{Fix}\ (\lambda x.\ x \to x) \to a\\&\qquad\quad f\ (\text{fix}\ x) = f\ (x\ (\text{fix}\ (\lambda x.\ x)))\end{aligned} \tag{2. 15}$$

这种谬误的出现使得我们必须对 ADT 的定义加以限制, 这在 Karl Mehltretter[2] 中有详细介绍, 我们在此不多赘述.

3 类型检查与繁饰

我们现在来尝试实现 MLTT 的类型检查, 检查的核心是一个推导类型的函数, 该函数在一个给定的语境中执行, 试图推导输入的表达式的类型. 该函数可能抛出异常 (即返回一个错误状态), 此时表明输入的表达式中存在类型错误.

我们将会使用类 Haskell 的伪代码, 语法和语义很接近我们上面的描述.

3. 1 双向类型检查 (Bidirectional Type Checking)

现在我们介绍一个经典的类型检查算法, 被称为双向类型检查[19], 我们简化一下 MLTT, 暂且不管 Id 类型, 因为之后我们会通过归纳类型实现它. 首先我们考虑定义语法树,

$$\begin{aligned}\texttt{data Term} :=\ & \texttt{Var Name}\\ \mid\ & \texttt{App Term Term}\\ \mid\ & \texttt{Lam Name Term}\\ \mid\ & \texttt{Pi Name Term Term}\\ \mid\ & \texttt{U Nat}\\ \mid\ & \texttt{Let Name Term Term Term}\\ & \quad \{\text{- Let } x\ t\ v\ b \text{ 相当于 let } x : t := v \text{ in } b \text{ -}\}\end{aligned} \tag{3. 1}$$

然后我们需要考虑如何实现判定相等性, 正如先前所述, 我们需要比较正规形式, 所以现在需要一个将表达式计算为正规形式的过程, 我们称为正规化 (normalization).

我们这里介绍一个用于正规化的算法, 该算法利用一个弱求值函数进行规范化操作, 称为 Normalization by Evaluation (NbE), 这可以看作大步求值的实现, 我们定义值在代码中的表示,

$$\begin{aligned}\texttt{data Value} :=\ & \texttt{VRig Name (List Value)}\\ \mid\ & \texttt{VLam Name Closure}\\ \mid\ & \texttt{VPi Name Value Closure}\\ \mid\ & \texttt{VU Nat}\end{aligned} \tag{3. 2}$$

记得我们提到过, 项需要在语境中解释, 所以我们还需要一个类型用来表示语境,

$$\texttt{Env} := \texttt{List (Name} \times \texttt{Value)} \tag{3. 3}$$

列表中每项代表变量名及其对应的值. 为了叙述简单起见, 我们这里采用了具名表示, 但这并非实践上最优的实现方式. 另外 Closure 类型用来表示一个待替换的子项, 例如对于 $\lambda x.\ b$, 我们希望的语义是该项等待接收一个值, 并将该值和 x 绑定, 然后插入到一个语境中并解释 b, 该语境必须是此 λ 项所在的语境, 我们可以认为 b 会捕获该语境, 并称捕获了语境的表达式为闭包(Closure),

$$\texttt{Closure} := \texttt{Env} \times \texttt{Term} \tag{3. 4}$$

现在我们可以来实现求值器了,

$$
\begin{aligned}
&\text{eval} : \text{Env} \to \text{Term} \to \text{Value} \\
&\text{eval}\ env\ (\text{Var}\ x) := \text{lookup}\ x\ env \\
&\text{eval}\ env\ (\text{App}\ a\ b) := \text{app}\ (\text{eval}\ env\ a)\ (\text{eval}\ env\ b) \\
&\text{eval}\ env\ (\text{Lam}\ x\ b) := \text{VLam}\ x\ (env, b) \\
&\text{eval}\ env\ (\text{Pi}\ x\ t\ b) := \text{VPi}\ x\ (\text{eval}\ env\ t)\ (env, b) \\
&\text{eval}\ env\ (\text{U}\ n) := \text{VU}\ n \\
&\text{eval}\ env\ (\text{Let}\ x\ t\ v\ e) := \text{eval}\ ((x, v) :: env)\ e
\end{aligned}
\tag{3. 5}
$$

其中, app 是一个部分函数, 定义如下,

$$
\begin{aligned}
&\text{app} : \text{Term} \to \text{Term} \to \text{Term} \\
&\text{app}\ (\text{VLam}\ x\ (env, b))\ v := \text{eval}\ ((x, v) :: env)\ b \\
&\text{app}\ (\text{VRig}\ f\ sp)\ v := \text{VRig}\ f\ (\text{append}\ sp\ [v])
\end{aligned}
\tag{3. 6}
$$

这个求值算法中比较特别的是 VRig, 它用来表示计算中被临时卡住的项, $\text{VRig}\ x\ [a, b, c]$ 用来表示这种形式的表达式 $x\ a\ b\ c$ 式中 x 为一变量, 这显然是当前语境下的最简形式.

有时我们需要把一个值再转回项, 该操作称为 quote, 其实现是显然的.

我们现在可以实现相等判定[11]了,

$$
\begin{aligned}
&\cdot \vdash \cdot \equiv \cdot : \text{Env} \to \quad \text{Value} \to \text{Value} \quad \to \text{Bool} \\
&\Gamma \vdash (\text{VRig}\ x_1\ sp_1) \equiv (\text{VRig}\ x_2\ sp_2) := x_1 = x_2 \wedge (\Gamma \vdash sp_1 \equiv sp_2) \\
&\Gamma \vdash (\text{VLam}\ x\ a) \quad \equiv (\text{VLam}\ y\ b) \quad := \text{convClosure}\ \Gamma\ x\ a\ y\ b \\
&\Gamma \vdash (\text{VPi}\ x\ s\ a) \quad \equiv (\text{VPi}\ y\ t\ b) \quad := \Gamma \vdash s \equiv t \wedge \text{convClosure}\ \Gamma\ x\ a\ y\ b \\
&\Gamma \vdash (\text{VU}\ n) \qquad \equiv (\text{VU}\ m) \qquad := n = m \\
&\Gamma \vdash v_1 \qquad\qquad \equiv v_2 \qquad\qquad := \text{false}
\end{aligned}
\tag{3. 7}
$$

其中,

$$
\begin{aligned}
&\text{convClosure} : \text{Env} \to \text{Name} \to \text{Closure} \to \text{Name} \to \text{Closure} \to \text{Bool} \\
&\text{convClosure}\ \Gamma\ x\ a\ y\ b := ((x', v) :: \Gamma) \vdash \text{evalClosure}\ x\ v\ a \equiv \text{evalClosure}\ y\ v\ b \\
&\quad \text{其中}\ x' := \text{generateName}\ \Gamma \\
&\qquad v := \text{VRig}\ x'\ []
\end{aligned}
\tag{3. 8}
$$

这里 generateName 是一个生成语境中不包含的名字的一个函数,

$$
\begin{aligned}
&\text{evalClosure} : \text{Name} \to \text{Value} \to \text{Closure} \to \text{Value} \\
&\text{evalClosure}\ x\ v\ (env, b) := \text{eval}\ ((x, v) :: env)\ b
\end{aligned}
\tag{3. 9}
$$

[11]这里省略了单位类型的判等规则, 所以可以不需要类型信息.

现在我们可以正式来看类型检查了, 类型检查所需要的语境需要每个变量的类型信息, 所以我们定义如下的 Env_{Type} 类型, 而由于检查的过程中需要进行求值操作来判定相等性, 该操作需要 Env 类型, 所以我们定义 Context 作为两个语境的组合, 我们的类型检查就可以在 Context 下进行,

$$\begin{aligned} \text{Env}_{\text{Type}} &:= \text{List Name} \times \text{Value} \\ \text{Context} &:= \text{Env} \times \text{Env}_{\text{Type}} \end{aligned} \tag{3. 10}$$

我们接下来使用的类型检查算法被称为双向类型检查, 其形式化描述可见 [19], 这里只提供其算法描述, 算法使用了两个互相递归调用的函数, 分别是 infer 和 check[12]. infer 函数接受语境和一个项, 然后试图推导该项的类型并返回该类型的求值后形式. check 则是在语境下接受一个项和它所属的类型, 并判断该赋型关系是否成立. 计算中的错误以异常形式抛出,

$$\begin{aligned}
&\text{infer} : \text{Context} \to \text{Term} \to \text{Value} \\
&\text{infer } (ty, val)\ (\text{Var } x) := \text{lookup } x\ ty \\
&\text{infer } (ty, val)\ (\text{App } a\ b) := \text{match infer } (ty, val)\ a \text{ with} \\
&\quad \text{VPi } x\ s\ t \Rightarrow \text{if check } (ty, val)\ b\ (\text{eval } val\ s) \\
&\qquad\qquad \text{then evalClosure } x\ (\text{eval } val\ b)\ t \\
&\qquad\qquad \text{else error} \\
&\quad _ \Rightarrow \text{error} \\
&\text{infer } (ty, val)\ (\text{Pi } x\ s\ t) := \\
&\quad \text{let } ty' := (x, \text{eval } val\ s) :: ty \text{ in} \\
&\quad \text{let } val' := (x, \text{VRig } x\ []) :: val \text{ in} \\
&\qquad \max(\text{inferLevel } (ty, val)\ s,\ \ \text{inferLevel } (ty', val')\ t) \\
&\text{infer } (ty, val)\ (\text{U } n) := \text{U } (n + 1) \\
&\text{infer } (ty, val)\ (\text{Let } x\ t\ v\ b) := \\
&\quad \text{let } ty' := (x, \text{eval } val\ t) :: ty \text{ in} \\
&\quad \text{let } val' := (x, \text{eval } val\ v) :: val \text{ in} \\
&\qquad \text{if } (\exists n, \text{inferLevel } (ty, val)\ t = n) \wedge \text{check } (ty, val)\ v\ (\text{eval } val\ t) \\
&\qquad\quad \text{then infer } (ty', val')\ b \\
&\qquad\quad \text{else error} \\
&\text{infer } (ty, val)\ _ := \text{error}
\end{aligned} \tag{3. 11}$$

其中 inferLevel 函数仅是一个辅助函数,

[12] 有的文献中也称为 synthesis 和 inherit.

$$
\begin{aligned}
&\text{inferLevel} : \text{Context} \to \text{Term} \to \text{Nat} \\
&\text{inferLevel } env\ t := \ \text{match infer } env\ t \text{ with} \\
&\qquad\qquad\qquad\qquad\quad \mid \text{VU } n \Rightarrow n \\
&\qquad\qquad\qquad\qquad\quad \mid \_ \Rightarrow \text{error}
\end{aligned}
\tag{3. 12}
$$

然后我们可以定义 check 函数,

$$
\begin{aligned}
&\text{check} : \text{Context} \to \text{Term} \to \text{Value} \to \text{Bool} \\
&\text{check } (ty, val)\ (\text{Lam } x\ b)\ (\text{VPi } y\ s\ t) := \\
&\qquad \text{let } s' := \text{eval } val\ s \text{ in} \\
&\qquad \text{let } v := \text{VRig } (\text{generateName } val)\ [] \text{ in} \\
&\qquad \text{check } ((x, s') :: ty, (x, v) :: val)\ b\ (\text{eval } ((y, s') :: ty, (y, v) :: val)\ t) \\
&\text{check } (ty, val)\ (\text{Let } x\ s\ v\ b)\ t := \\
&\qquad \text{let } ty' := (x, \text{eval } val\ s) :: ty \text{ in} \\
&\qquad \text{let } val' := (x, \text{eval } val\ v) :: val \text{ in} \\
&\qquad \text{check } (ty', val')\ b\ t \\
&\text{check } (ty, val)\ a\ t := val \vdash \text{infer } (ty, val)\ a \equiv t
\end{aligned}
\tag{3. 13}
$$

此算法即为一个完整的双向类型检查算法, 算法中变化的语境是重点内容. 读者可以尝试自行往里面添加其他构造, 包括 Σ-类型, Id-类型等, 只需记住, 构造子在 check 模式下检查, 消除子在 infer 模式下推导.

后文我们将会使用 $\Gamma \vdash t : T$ 来表示在语境 Γ 下 t 的类型为 T, 读者可将此理解成 check

3. 2 加入归纳类型

我们提到过归纳类型是增强语言表现力的强大工具, 现在我们需要考虑如何实现它, 我们有如下几个方案,

- 直接使用 $\mathcal{W}$-类型[16] 提供对归纳类型的支持.
- 允许用户定义归纳类型, 并通过定义生成相应的构造子和消除子.
- 允许用户定义归纳类型, 但不自动生成消除子, 而是通过模式匹配定义相关函数.
- 结合上述两种方式, 将模式匹配定义的函数转换成消除子定义.

我们这里将采用第三种方式; 第一种方案极其不便使用; 第二种由于缺少对模式匹配的支持, 导致在实现各种互归纳类型的时候会十分困难; 第四种虽然很优雅, 但实际上的好处并不是很多, 并且其实现较为复杂. Agda[20] 也使用第三种方式提供对归纳类型的支持. 我们将会设计最简单且自然的规则来提供一个表达能力足够使用的定理证明系统.

我们现在来考察归纳类型和模式匹配的语法以及语义.

定义归纳类型需要如下几个要素:

- 类型的名字.
- 类型的索引.
- 类型的一系列构造子的名字.
- 这些构造子的参数和返回的索引值.

例如如下的类型,

$$\begin{aligned}
&\text{data}\ \overbrace{\text{Vec}}^{\text{类型的名字}}\ : \overbrace{\mathcal{U} \to \text{Nat}}^{\text{类型的索引}} \to \mathcal{U}\ \text{where} \\
&\quad | \ \overbrace{\text{vnil}}^{\text{构造子的名字}}\ : \overbrace{(A : \mathcal{U})}^{\text{构造子的参数}} \to \text{Vec}\ A\ \overbrace{\text{zero}}^{\text{构造子返回的索引值}} \\
&\quad | \ \underbrace{\text{vcons}}_{\text{构造子的名字}}\ : \underbrace{(A : \mathcal{U})(n : \text{Nat}) \to A \to \text{Vec}\ A\ n}_{\text{构造子的参数}} \to \underbrace{\text{Vec}\ A\ (\text{suc}\ n)}_{\text{构造子返回的索引值}}
\end{aligned} \tag{3. 14}$$

不同的构造子返回的索引值可以不同, 但最外层的类型构造(这里是 Vec)一定要是被定义的类型.

我们有了归纳类型的语法, 现在来看它的语义, 我们会希望解释项和函数定义的时候能够查询到已经定义了的类型和它的构造子们, 这使得我们需要添加一个新的语境, 我们记作 Σ, 它是归纳类型定义构成的序列, 并配备如下几个操作,

- $\text{lookup}_\Sigma(\text{data\_name}) \Rightarrow \text{type}$, 在 Σ 中查询名为 data_name 的归纳类型, 并返回它的类型.
- $\text{lookup}'_\Sigma(\text{cons\_name}) \Rightarrow \text{type}$, 在 Σ 中查询名为 cons_name 的构造子, 并返回它的类型.
- $\text{constrof}_\Sigma(\text{data\_name}) \Rightarrow \overline{\text{cons\_name}}$, 在 Σ 中查询名为 data_name 的归纳类型的所有构造子名字构成的序列.

这样原本的类型判定 $\Gamma \vdash t : T$ 就需要改成 $\Sigma; \Gamma \vdash t : T$, 因为 t 中可能含有 Σ 中定义的内容.

于是我们将有如下的两条显然的新规则,

$$\frac{\text{lookup}_\Sigma(x) \Rightarrow T}{\Sigma; \Gamma \vdash x : T} \quad \frac{\text{lookup}'_\Sigma(x) \Rightarrow T}{\Sigma; \Gamma \vdash x : T} \tag{3. 15}$$

3. 3 关于依赖模式匹配

由于在依值类型论中函数的参数可能依赖其他参数, 所以我们在模式匹配时也需要考察不同参数间相互影响的情况, 这里有两种的情形值得注意.

- 对前面的参数模式匹配会导致后面的参数的可能的模式发生变化.
- 对后面的参数模式匹配引起了前面的参数中的变量模式有了确定的值.

关于第一种情形, 一个很好的例子是 Vec 的拼接操作,

$$
\begin{aligned}
&\text{append} : (n\ m : \text{Nat}) \to \text{Vec}\ A\ n \to \text{Vec}\ A\ m \to \text{Vec}\ A\ (\text{add}\ n\ m)\\
&\text{append}\ \text{zero}\ m\ \text{vnil}\ ys \coloneqq ys\\
&\text{append}\ (\text{suc}\ n)\ m\ (\text{vcons}\ \_\ \_\ x\ xs)\ ys \coloneqq \text{vcons}\ \_\ \_\ x\ (\text{append}\ n\ m\ xs\ ys)
\end{aligned}
\tag{3. 16}
$$

我们用下划线表示不需要进行匹配的模式(不绑定变量的变量模式)和显而易见的参数. 注意这里的第一个子句的第一个参数, 它用 zero 模式去匹配参数 n, 这影响了第三个参数的类型 (从 $\text{Vec}\ A\ n$ 变成了 $\text{Vec}\ A\ \ \text{zero}$), 这使得这里不能使用 vcons 而只能使用 vnil; 如果考察第二个子句, 它将 $\text{Vec}\ A\ n$ 变成 $\text{Vec}\ A\ (\text{suc}\ n)$, 同理, 我们这里就不需要考虑 vnil 的情况.

关于第二种情况, 典型的例子是在等式证明中出现的,

$$
\begin{aligned}
&\text{symm}' : (f : A \to B)\ (x : A)\ (y : B) \to \text{Id}\ B\ y\ (f\ x) \to \text{Id}\ B\ (f\ x)\ y\\
&\text{symm}'\ f\ x\ ?_1\ (\text{refl}\ \ ?_2) \coloneqq \text{refl}\ (f\ x)
\end{aligned}
\tag{3. 17}
$$

考虑这里 $?_1$ 和 $?_2$ 该填什么, 按照上一个情形的思路, $?_1$ 处应该填入 $f\ x$, 这样我们就能将第三个参数的类型改写成 $\text{Id}\ B\ (f\ x)\ (f\ x)$, 然后使用 refl 模式正好合适, 但是 $f\ x$ 并不是个模式, 所以我们不能先确定 $y = f\ x$ 再确定第四个参数的类型, 相反, 我们应该直接给出 refl, 并由 refl 确定 $?_1$ 的值, 此时真正决定这个模式匹配的是 refl, 而不是 $?_1$. 我们可以这样写

$$
\text{symm}'\ f\ x\ y\ (\text{refl}\ \ z) \coloneqq \text{refl}\ (f\ x) \tag{3. 18}
$$

这样会在右手边引入如下判定相等关系:

$$
\begin{aligned}
y &= f\ x\\
z &= f\ x
\end{aligned}
\tag{3. 19}
$$

有了以上直觉, 我们来说明如何检查一个由模式匹配, 首先这里需要定义函数的语法,

$$
\begin{aligned}
\text{fun\_decl} &::= \text{id} : \text{term}\\
&\qquad\ \overline{\text{id clause}}\\
\text{clause} &::= \ \overline{\text{pattern}}\ \ ':='\ \text{term}
\end{aligned}
\tag{3. 20}
$$

我们将要定义如下判断,

$$
\text{fun\_decl}\ \textbf{valid} \tag{3. 21}
$$

这将用来表示一个函数声明是合法的. 该定义将由很多部分组成, 我们从检查模式开始,

$$
\text{sig}; \text{ctx} \vdash \textbf{chkPat}\ \overline{\text{pattern}} : \text{telescope} \to \text{term} \Rightarrow \text{ctx}; \text{telescope} \to \text{term} \tag{3. 22}
$$

其中 telescope ::= $\overline{(\text{id} : \text{term})}$, 这表示我们将在 telescope (可以看作参数列表)下检查 $\overline{\text{pattern}}$ 的正确性, 并给出一个新语境和剩下未匹配的参数, 并且该参数中不会含有被匹配掉的变量. 我们将会用 $\bar{t}$ 或者大写希腊字母表示一个 telescope.

要是实现这点需要注意在 telescope 中的 类型可能会依赖于前面变量的匹配值,

$$\frac{\Gamma \vdash \Delta \to T \Downarrow \Delta' \to T'}{\Sigma; \Gamma \vdash \mathbf{chkPat}\ \emptyset : \Delta \to T \Rightarrow \Gamma; \Delta' \to T'}\ \text{PatEmp}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathbf{chkPat}\ p_1 : t_1 \to (t_{\text{rest}} \to T) \Rightarrow \Gamma'; t'_{\text{rest}} \to T' \\ \Sigma; \Gamma' \vdash \mathbf{chkPat}\ p_2 : t'_{\text{rest}} \to T' \Rightarrow \Gamma''; T''\end{array}}{\Sigma; \Gamma \vdash \mathbf{chkPat}\ p_1\ \overline{p_{\text{rest}}} : t_1\ \overline{t_{\text{rest}}} \to T \Rightarrow \Gamma''; T''}\ \text{PatTrans}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash T : \mathcal{U} \\ \Sigma; \Gamma, (x : T), (x' : T = x) \vdash \Delta \to T \Downarrow \Delta' \to T'\end{array}}{\Sigma; \Gamma \vdash \mathbf{chkPat}\ x : (x' : T)\ \Delta \to T' \Rightarrow (x : T), \Gamma; \Delta' \to T'}\ \text{PatVar} \tag{3. 23}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash T \Downarrow D\ \overline{\theta_i} \qquad \text{lookup}_\Sigma(D) \Rightarrow \Theta \to \mathcal{U} \\ \text{lookup}'_\Sigma(c) \Rightarrow \overline{(x_i^c : T_i^c)} \to D\ \overline{\theta'_i} \\ \Sigma; \Gamma \vdash \mathbf{chkPat}\ \overline{p_i} : \overline{(x_i^c : T_i^c)} \to D\ \overline{\theta'_i} \Rightarrow \Gamma'; D\ \overline{\theta''_i} \\ \Sigma; \Gamma' \vdash \mathbf{unify}_{\text{fv}(\overline{\theta_i})} \left(\overline{\theta_i} \sim \overline{\theta''_i} : \Theta\right) \Rightarrow \Gamma'' \\ \Sigma; \Gamma'', (x' : T = c\ \overline{p_i}) \vdash \Delta \to T' \Downarrow \Delta' \to T''\end{array}}{\Sigma; \Gamma \vdash \mathbf{chkPat}\ (c\ \overline{p_i}) : (x' : T)\Delta \to T' \Rightarrow \Gamma''; \Delta' \to T''}\ \text{PatCon}$$

其中的 **unify** 试图匹配两边的表达式, 并在明确无法匹配时返回一个特定状态, 定义如下,

$$\frac{x \in F \quad x \notin \mathrm{fv}(t) \quad \Sigma; \Gamma, (x : T = t) \vdash \Gamma \Downarrow \Gamma'}{\Sigma; \Gamma \vdash \mathbf{unify}_F(x \sim t : T) \Rightarrow \Gamma'} \text{ UnifyVar}$$

$$\frac{x \in F \quad x \notin \mathrm{fv}(t) \quad \Sigma; \Gamma, (x : T = t) \vdash \Gamma \Downarrow \Gamma'}{\Sigma; \Gamma \vdash \mathbf{unify}_F(t \sim x : T) \Rightarrow \Gamma'} \text{ UnifyVar'}$$

$$\frac{\begin{array}{c}\mathrm{lookup}_\Sigma(c) \Rightarrow \Delta \to D\ \overline{\theta} \\ \Sigma; \Gamma \vdash \mathbf{unify}_F(\overline{u} \sim \overline{v} : \Delta) \Rightarrow \Gamma'\end{array}}{\Sigma; \Gamma \vdash \mathbf{unify}_F(c\ \overline{u} \sim c\ \overline{v} : T) \Rightarrow \Gamma'} \text{ UnifyCon}$$

$$\frac{c \neq c'}{\Sigma; \Gamma \vdash \mathbf{unify}_F(c\ \overline{u} \sim c'\ \overline{v} : T) \not\Rightarrow} \text{ UnifyFail} \tag{3. 24}$$

$$\frac{}{\Sigma; \Gamma \vdash \mathbf{unify}_F(\emptyset \sim \emptyset : \emptyset) \Rightarrow \Gamma} \text{ UnifyEmp}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathbf{unify}_F(u \sim v : T) \Rightarrow \Gamma' \\ \Sigma; \Gamma' \vdash u \Downarrow u' \\ \Sigma; \Gamma', (x : T = u') \vdash \Delta \Downarrow \Delta' \\ \Sigma; \Gamma' \vdash \mathbf{unify}_F(\overline{u} \sim \overline{v} : \Delta') \Rightarrow \Gamma''\end{array}}{\Sigma; \Gamma \vdash \mathbf{unify}_F(u\overline{u} \sim v\overline{v} : (x : T)\Delta) \Rightarrow \Gamma''} \text{ UnifyTrans}$$

$$\frac{\Sigma; \Gamma \vdash u \equiv v : T}{\Sigma; \Gamma \vdash \mathbf{unify}_F(u \sim v : T) \Rightarrow \Gamma} \text{ UnifyConv}$$

这套规则可以直接转换成代码, 并且刻意避免了代换操作, 相比于区分 inaccessible patterns 的方案[21] 在实现时会更方便自然, 但性能较为低下, 原因是 **unify** 完成后对语境的更新较为耗时, 但考虑到函数定义皆在顶层, 语境不深, 故可接受.

检查完模式之后我们可以用先前检查的结果来进一步检查整个子式, 对于形如下式的定义

$$\begin{array}{l} f : T \\ f\ \overline{p} \coloneqq rhs \end{array} \tag{3. 25}$$

我们用 $\mathbf{chkCls}\ (f : T; \overline{p} \coloneqq rhs)$ 表示这个子句合法.

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathbf{chkPat}\ \overline{p} : T \Rightarrow \Gamma'; T' \\ \Sigma, (f : T); \Gamma' \vdash rhs : T'\end{array}}{\Sigma; \Gamma \vdash \mathbf{chkCls}\ (f : T; \overline{p} \coloneqq rhs)} \tag{3. 26}$$

继续进行之前我们考虑模式匹配的求值语义, 对于一个包含很多子句的函数定义, 我们采用最先匹配语义 (first match), 即在匹配时执行第一个匹配成功的模式. 我们先定义如何匹配, 记作

$$\Sigma; \Gamma \vdash \mathbf{match}\ (\overline{p}, \overline{t}) \Rightarrow \Gamma' \tag{3. 27}$$

其中 p 为模式, t 为项, 并假设它已经是正规形式, 匹配之后的结果是一个新语境, 其中包含模式中的变量及其被匹配的结果, 由于这套规则只会在求值中使用, 而不在类型检查过程中使用, 所以不需要考察类型信息.

$$\frac{}{\Sigma; \Gamma \vdash \mathbf{match}\ (x, t) \Rightarrow \Gamma, (x = t)}\ \text{MatVar}$$

$$\frac{\Sigma; \Gamma \vdash \mathbf{match}\ (\overline{p}, \overline{t}) \Rightarrow \Gamma'}{\Sigma; \Gamma \vdash \mathbf{match}\ (c\ \overline{p}, c\ \overline{t}) \Rightarrow \Gamma'}\ \text{MatCon}$$

$$\frac{c \neq c'}{\Sigma; \Gamma \vdash \mathbf{match}\ (c\ \overline{p}, c'\ \overline{t}) \nRightarrow}\ \text{MatFail}$$

$$\frac{}{\Sigma; \Gamma \vdash \mathbf{match}\ (c\ \overline{p}, x)\ \mathbf{stuck}_x}\ \text{MatStuck}$$

$$\frac{}{\Sigma; \Gamma \vdash \mathbf{match}\ (\emptyset, \overline{t})\ \ \mathbf{stuck}}\ \text{MatStuckFunc}$$

$$\frac{}{\Sigma; \Gamma \vdash \mathbf{match}\ (\emptyset, \emptyset) \Rightarrow \Gamma}\ \text{MatEmp}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathbf{match}\ (p, t) \Rightarrow \Gamma' \\ \Sigma; \Gamma' \vdash \mathbf{match}\ (\overline{p}, \overline{t}) \Rightarrow \Gamma''\end{array}}{\Sigma; \Gamma \vdash \mathbf{match}\ (p\ \overline{p}, t\ \overline{t}) \Rightarrow \Gamma''}\ \text{MatTrans} \tag{3. 28}$$

$$\frac{\Sigma; \Gamma \vdash \mathbf{match}\ (p, t) \nRightarrow}{\Sigma; \Gamma \vdash \mathbf{match}\ (p\ \overline{p}, t\ \overline{t}) \nRightarrow}\ \text{MatFailTrans1}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathbf{match}\ (p, t) \Rightarrow \Gamma' \\ \Sigma; \Gamma' \vdash \mathbf{match}\ (\overline{p}, \overline{t}) \nRightarrow\end{array}}{\Sigma; \Gamma \vdash \mathbf{match}\ (p\ \overline{p}, t\ \overline{t}) \nRightarrow}\ \text{MatFailTrans2}$$

$$\frac{\Sigma; \Gamma \vdash \mathbf{match}\ (p, t)\ \mathbf{stuck}_x}{\Sigma; \Gamma \vdash \mathbf{match}\ (p\ \overline{p}, t\ \overline{t})\ \mathbf{stuck}_x}\ \text{MatStuckTrans1}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathbf{match}\ (p, t) \Rightarrow \Gamma' \\ \Sigma; \Gamma' \vdash \mathbf{match}\ (\overline{p}, \overline{t})\ \mathbf{stuck}_x\end{array}}{\Sigma; \Gamma \vdash \mathbf{match}\ (p\ \overline{p}, t\ \overline{t})\ \mathbf{stuck}_x}\ \text{MatStuckTrans2}$$

为了执行函数, 我们需要添加如下操作,

- $\text{arity}_\Sigma(f) \Rightarrow n$ 表示函数参数的个数, 只有当函数的参数达到这个数值时模式匹配才会发生.
- $\text{lookup}_\Sigma(f) \Rightarrow \begin{cases} f : T \\ f\ \overline{p} := t \\ \dots \end{cases}$ 在语境中查找函数 f 的定义.
- $\text{signature}\ (f_{\text{decl}}) \Rightarrow T$ 从函数定义中取出函数类型.
- $\text{clauses}\ (f_{\text{decl}}) \Rightarrow \overline{\overline{p} := t}$ 从一个函数定义中取出其各个子句.

$$\frac{\begin{array}{c}\text{lookup}_\Sigma(f) \Rightarrow f_{\text{decl}} \quad \text{clauses}\ (f_{\text{decl}}) \Rightarrow f_{\text{cls}} \quad \overline{t'}\ \overline{t''} = \overline{t} \\ \text{length}(\overline{t}) > \text{arity}_\Sigma(f) \Rightarrow \text{length}(\overline{t'}) = \text{arity}_\Sigma(f) \\ \text{length}(\overline{t}) \leq \text{arity}_\Sigma(f) \Rightarrow \text{length}(\overline{t''}) = 0 \\ \Sigma; \Gamma \vdash \mathbf{evalFun}_f(f_{\text{cls}}, \overline{t'}) \Rightarrow v \quad \Sigma; \Gamma \vdash v\ \overline{t''} \Downarrow v'\end{array}}{\Sigma; \Gamma \vdash f\ \overline{t} \Downarrow v'}\ \text{EvalFun} \tag{3. 29}$$

其中 **evalFun** 定义如下,

$$\frac{\text{length}\ (\overline{t}) < \text{arity}_\Sigma(f)}{\Sigma; \Gamma \vdash \mathbf{evalFun}_f(cls, \overline{t}) \Rightarrow f\ \overline{t}}\ \text{EvalFunWait}$$

$$\frac{\begin{array}{c}\text{length}(\overline{t}) = \text{arity}_\Sigma(f) \\ \Sigma; \Gamma \vdash \mathbf{match}\ (\overline{p}, \overline{t}) \Rightarrow \Gamma' \quad \Sigma; \Gamma' \vdash rhs \Downarrow v\end{array}}{\Sigma; \Gamma \vdash \mathbf{evalFun}_f\left(\begin{Bmatrix}\overline{p} := rhs \\ \overline{c}\end{Bmatrix}, \overline{t}\right) \Rightarrow v}\ \text{EvalFunMatch}$$

$$\frac{\begin{array}{c}\text{length}(\overline{t}) = \text{arity}_\Sigma(f) \quad \Sigma; \Gamma \vdash \mathbf{match}(\overline{p}, \overline{t}) \nRightarrow \\ \Sigma; \Gamma \vdash \mathbf{evalFun}_f(\overline{c}, \overline{t}) \Rightarrow v\end{array}}{\Sigma; \Gamma \vdash \mathbf{evalFun}_f\left(\begin{Bmatrix}\overline{p} := rhs \\ \overline{c}\end{Bmatrix}, \overline{t}\right) \Rightarrow v}\ \text{EvalFunNext} \tag{3. 30}$$

$$\frac{\text{length}(\overline{t}) = \text{arity}_\Sigma(f) \quad \Sigma; \Gamma \vdash \mathbf{match}(\overline{p}, \overline{t})\ \mathbf{stuck}_x}{\Sigma; \Gamma \vdash \mathbf{evalFun}_f\left(\begin{Bmatrix}\overline{p} := rhs \\ \overline{c}\end{Bmatrix}, \overline{t}\right) \Rightarrow f\ \overline{t}}\ \text{EvalFunStuck}$$

这套规则体现了最先匹配语义, 当前面的子句无法匹配的时候才会尝试后面的子句.

现在我们来考虑模式的**完全性检查** (Coverage Check), 检查的思路是为模式生成一组相应类型的变量, 再尝试用这些变量去匹配模式, 若成功, 则说明没有遗漏的模式, 否则继续将这些变量精化为构造器模式, 并重复这个过程, 直到所有的项都成功匹配或者找出遗漏的模式.

举例如下, 考虑加法函数的两组模式,

$$\text{add} : \quad \text{Nat} \quad \to \quad \text{Nat} \quad \to \quad \text{Nat}$$

$$
\begin{aligned}
&\text{add} \quad \text{zero} \quad\ \ n \quad\ \ := n \\
&\text{add} \quad (\text{suc}\ m) \quad n \quad\ \ := \text{suc}\ \ (\text{add}\ m\ n)
\end{aligned}
$$

为了检查这组模式是否完整, 我们生成两个变量 m_1 和 n_1, 并用这两个变量去匹配这些模式, 根据先前关于 **match** 的规则, 我们得到这组模式匹配的结果是 $\mathbf{stuck}_{m_1}$, 即变量 m_1 导致匹配卡住了, 所以我们尝试分解 m_1, 根据 m_1 的类型 Nat, 它有两个构造器, zero 和 suc, 于是我们的问题分成了两个, 分别是用 zero 代替 m_1 去完成刚才的匹配和用 suc m_2 代替刚才的 m_1 去完成匹配. 两个子问题都能成功完成匹配, 于是我们认为这组模式是完全的, 如若这组模式中缺少 suc, 那么必然会发生一个非 **stuck** 的失败匹配, 此时我们认为模式是不完全的. 另外, 这个过程一定是终止的, 原因有两点, 首先是模式的长度有限且一个归纳类型的构造子个数也有限. 注意这个思路并不保证这组模式一定能表示为一个分支树[22].

首先我们需要一个用来分解变量, 找出哪些模式可用的机制, 在依赖模式匹配下, 这个过程会涉及 unification. 我们定义

$$
\Sigma;\Gamma \vdash \mathbf{cases}_P(x) \Rightarrow \overline{t;\Gamma'} \tag{3. 31}
$$

来完成这一操作, 其中 t 表示 x 可能被分解成的一个构造器, Γ' 表示对应分解操作完成后的新语境, 由于可能存在多个构造器, 所以这里也要考虑多个新语境. 其中 P 是这组模式里所有的变量模式构成的集合, 这是为了避免在生成新变量的时候和已有变量重名.

$$
\frac{\begin{array}{c}
\Sigma;\Gamma \vdash x : D\ \overline{\theta} \\
\text{constrof}_\Sigma(D) \Rightarrow \overline{c_i} \quad \text{lookup}_\Sigma(D) \Rightarrow \Theta \to \mathcal{U} \\
\forall \Gamma_i', \begin{cases}
\Sigma;\emptyset \vdash c_i : \Delta \to D\ \overline{\theta'} \\
\Delta_{\text{var}} \cap \Gamma_{\text{var}} = \emptyset \wedge \Delta_{\text{var}} \cap P = \emptyset \\
\Sigma;\Gamma\Delta \vdash \mathbf{unify}_{\text{fv}(\overline{\theta})}\left(\overline{\theta} \sim \overline{\theta'} : \Theta\right) \Rightarrow \Gamma_i' \\
\Sigma;\Gamma_i' \vdash T \Downarrow T'
\end{cases}
\end{array}}{\Sigma;\Gamma \vdash \mathbf{cases}_P(x) \Rightarrow \overline{c_i \Delta_{\text{var}};\Gamma_i', (x : T' = c_i \Delta_{\text{var}})}} \tag{3. 32}
$$

其中 Δ_{var} 表示语境中的变量, 这可以通过对 $\Delta \to D\ \overline{\theta'}$ 进行重命名得到.

我们用 $\Sigma;\Gamma \vdash \mathbf{cover}\ \left(\overline{p} : \Theta \mid \overline{p'}\right)$ 表示一组模式 $\overline{p}$ 可以匹配 $\overline{p'}$,

$$
\frac{\Sigma;\Gamma \vdash \mathbf{matchAll}\left(\overline{\overline{p}} \mid \overline{p'}\right) \Rightarrow \Gamma'}{\Sigma;\Gamma \vdash \mathbf{cover}\ \left(\overline{\overline{p}} : \Theta \mid \overline{p'}\right)}\ \text{CoverPass}
$$

$$
\frac{\begin{array}{c}
\Sigma;\Gamma \vdash \mathbf{matchAll}\left(\overline{\overline{p}} \mid \overline{p'}\right)\ \overline{\mathbf{stuck}_x} \\
\Sigma;\Gamma \vdash \mathbf{cases}_{\text{fv}\left(\overline{p'}\right)}(x) \Rightarrow \overline{c_i\theta;\Gamma_i} \\
\forall \Gamma_i, \Sigma;\Gamma_i \vdash \mathbf{cover}\ \left(\overline{\overline{p}} : \Theta \mid \overline{p'}\left[c_i\overline{\theta}/x\right]\right)
\end{array}}{\Sigma;\Gamma \vdash \mathbf{cover}\ \left(\overline{\overline{p}} : \Theta \mid \overline{p'}\right)}\ \text{CoverSplit} \tag{3. 33}
$$

其中 **matchAll** 是对一组模式进行匹配, 并返回第一个成功匹配的结果, 其定义是显然的, 这里不做赘述.

现在我们终于可以定义对整个函数的检查,

$$\frac{\begin{array}{c}\mathrm{clauses}(f_{\mathrm{decl}}) \Rightarrow \overline{cls} \\ \forall cls, \Sigma; \Gamma \vdash \mathbf{chkCls}\ (f : \Theta \to T; cls) \\ \Sigma; \Gamma \vdash \mathbf{cover}\ \left(\overline{\overline{p}} : \Theta \mid \Theta_{\mathrm{var}}\right)\end{array}}{\begin{cases} f : \Theta \to T \\ f\ \overline{p} := t \quad \mathbf{valid} \\ \cdots \end{cases}} \tag{3. 34}$$

$$\frac{f_{\mathrm{decl}}\ \mathbf{valid}}{\Sigma \vdash \mathbf{chkFun}\ f_{\mathrm{decl}} \Rightarrow \Sigma, f_{\mathrm{decl}}}$$

到目前为止, 我们所描述的系统已经是一个合格的编程语言了, 有一个可靠的类型系统, 但是作为定理证明器, 我们追求一致性, 所以还需要终止性检查. Karl Mehltretter[2] 对此有详细的描述, 并能支持互递归的函数. 其思路是用一个矩阵表示两个函数间的互递归关系, 然后通过矩阵乘法最终寻找到函数到自身的调用关系. 本文在此处不深入介绍.

3. 4 繁饰

繁饰是另一个大话题, 受写作时间限制, 本文不会深入解释, 关于此话题, András Kovács 的示例项目[1] 是一个极好实现示例.

3. 5 实现

对于本章描述的系统, 作者在此处提供了一个完整的实现[13], 该实现是一个基于 Haskell 的依值类型语言, 其实现了完整的双向类型检查算法, 模式匹配, 模式完全性检查与终止性检查, 并支持本文叙述至此的所有特性. 我们会在附录中给出基于此语言的编程和定理证明的例子.

4 扩展

我们现在来讨论在这个系统之外能提供什么样的扩展, 以及定理证明器未来可能的发展方向.

4. 1 证明策略

实用定理证明器大多会提供一个所谓的证明策略(Tactic)系统, 其本质上是一个元编程机制, 允许用户通过命令式的方式来指导生成具体的证明项. Rocq 和 Lean[23] 是两个知名的例子, 合适的证明策略能大幅缩短证明长度, 提高证明效率.

若要实现一个生成具体证明项的证明策略系统颇为复杂. 另一个实现证明策略系统的简单方式, 是仅验证证明策略的正确性而不生成具体的证明项, 这使得证明策

[13] https://github.com/KonjacSource/ShiTT

略系统成为一个完全独立的新语言, 为了实现这一点, 我们需要为定理证明器添加一个 Prop 宇宙, 满足 $(A : \mathrm{Prop})(x\ y : A) \to x = y$, 并只允许证明策略系统在 Prop 宇宙中的类型上运作, 这样就能保证即使不生成证明项也无关紧要.

4. 2 同伦类型论

同伦类型论将类型类比作几何化的空间, 元素对应空间中的点, 而两个元素之间的相等性被看作空间中的一条定端道路, 并允许用户通过泛等公理[18] 来构造类型间的道路, 这使得 K 公理不再成立. K 公理是说: 相等性的证明是唯一的. 在同伦类型论下, 这意味这道路只有一条, 显然是不能成立的.

但我们原本定义的模式匹配过程则是可以证明 K 公理的, 所以如果我们需要考虑对模式匹配做一些改变, 使得它能排除一些可能会导致 K 的模式, Jesper Cockx 对此有详细的描述[24] , 基本思路是识别对同一个变量的多次 unify, 这样就能避免把两条同端的道路缩为一条.

同伦类型论还提供了一个对归纳类型的扩展, 被称为高阶归纳类型, 一般的归纳类型只允许定义构造子, 但高阶归纳类型允许定义构造子之间的道路, 这使得同伦类型论有能力定义具体的几何对象. 例如如下定义就给出了一个圆周,

$$\begin{aligned} &\mathrm{data}\ S^1 : \mathcal{U}\ \mathrm{where} \\ &\quad \mid \mathrm{base} : S^1 \\ &\quad \mid \mathrm{loop} : \mathrm{base} = \mathrm{base} \end{aligned} \tag{4. 1}$$

其中 loop 定义了一条非平凡道路连接 base 到它自身, 这便定义了一个环. 对 S^1 的模式匹配必须保证在 loop 的两端函数给出的值必须和 base 处给出的值判定相等. 这要求我们为模式匹配添加一轮新的检查, 称为边界检查.

4. 3 更多归纳类型

正如先前所述, 归纳类型仍然有很大的推广空间, 除了同伦类型论中的高阶归纳类型, 另一个知名的例子是归纳-归纳类型, 这是说两个类型 $A : \mathcal{U}$ 和 $B : A \to \mathcal{U}$, 它们的构造器可以互相引用对方的类型, Forsberg[25] 对此有详细的描述.

在本文叙述的框架下, 归纳-归纳类型十分易于实现, 只需在检查类型 B 的类型之前先将 A 的类型插入到语境, 再将 A 和 B 的类型都插入语境, 然后检查 A 和 B 的定义. 其余部分, 包括模式匹配, 都不需要任何修改. 注意归纳-归纳类型和更简单一些的互递归类型必须在有互递归函数的系统中才能正常使用.

Thorsten Altenkirch 和 Ambrus Kaposi[26] 展示了一个归纳-归纳类型(以及商归纳类型, 这是高阶归纳类型的一个例子)的极佳示例.

5 对于物理学的意义

5. 1 公理化物理

Hilbert 在 1900 年提出过二十三个最重要的数学问题, 其中的第六个问题是关于物理学的公理化, 这是指以严格的数学语言来描述物理学的基本定律, 当前除了量子场论[14], 大多物理分支已有良好的公理化. 最简单的一个例子是拉格朗日力学, 它可以被完全地描述为配备了一个切丛上的标量场的微分流形. 这意味着一个拉格朗日力学系统可以被定义为一个代数结构, 这当然可以用类型论的语言来描述, 进而也可以使用定理证明器来验证, 这方面的尝试比较少, 但随着 Lean4 证明器的完善, 这方面的工作也在逐渐增多. 当前已有人做了一些尝试, 比如对量子场论中的 Wick 定理[27] 和物理化学的形式化验证[28,29].

5. 2 量子编程语言

量子计算机上的程序在资源控制上和经典计算机有很大的不同, 在量子计算机上, 受限于不可克隆原理[30], 程序无法复制特定的量子比特, 这恰好对应了线性类型论[31] 的理念: 参数必须被恰好使用一次, 这样就可以使用线性类型来指导量子编程语言的设计, 在这方面 Proto-Quipper-D[32] 实现了线性依值类型, 以此安全地编写量子计算机程序.

[14] 主要原因是路径积分的形式在数学上并不严格.

参考文献

[1] ANDRÁS KOVÁCS. elaboration-zoo[CP/OL]. https://github. com/ AndrasKovacs/elaboration-zoo.

[2] MEHLTRETTER K. Termination Checking for a Dependently Typed Language[D/OL]. 2007. https://www. cse. chalmers. se/~abela/mehltret-da. pdf.

[3] LEAN FRO. Lean4 Prover[EB/OL]. https://lean-lang. org/.

[4] THE COQ DEVELOPMENT TEAM. Rocq Prover[EB/OL]. https://rocq-prover. org/.

[5] MATHLIB COMMUNITY. Mathlib[CP/OL]. https://github. com/leanprover-community/mathlib4.

[6] 李文威. 代数学方法 卷一: 基础架构[M]. 高等教育出版社, 2019.

[7] RUSSELL B. Mathematical Logic as Based on the Theory of Types[J]. American Journal of Mathematics, 1908.

[8] J. ROGER HINDLEY, JONATHAN P. SELDIN. Lambda-Calculus and Combinators–An Introduction[M/OL]. Cambridge University Press, 2008. DOI:10. 1017/CBO9780511809835.

[9] PIERCE B C. Types and Programming Languages[M]. The MIT Press.

[10] REYNOLDS J. Towards a theory of type structure[A]. 1974.

[11] WELLS J. Typability and type checking in System F are equivalent and undecidable[J/OL]. Annals of Pure and Applied Logic, 1999. https://www. sciencedirect. com/science/article/pii/S0168007298000475.

[12] HINDLEY J R. The Principal Type-Scheme of an Object in Combinatory Logic[A/ OL]. (1969). DOI:10. 2307/1995158.

[13] MARLOW S. Haskell 2010 Language Report[A/OL]. https://www. haskell. org/.

[14] LEROY X, DOLIGEZ D, FRISCH A, et al. [EB/OL]. https://ocaml. org/manual/5. 3/index. html.

[15] GIRARD J Y. InterprÈtation fonctionnelle et Èlimination des coupures de l'arithmÈtique d'ordre supÈrieur[A/OL]. (1972). https://www. cs. cmu. edu/~kw/ scans/girard72thesis. pdf.

[16] PER MARTIN-LÖF. Constructive Mathematics and Computer Programming[A/ OL]. Elsevier, 1982. DOI:10. 1016/S0049-237X(09)70189-2.

[17] HOARE T. Null References: The Billion Dollar Mistake[EB/OL]. InfoQ. com, 2009. https://www. infoq. com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

[18] THE UNIVALENT FOUNDATIONS PROGRAM. Homotopy Type Theory: Univalent Foundations of Mathematics[M/OL]. 2013. https:// homotopytypetheory. org/book.

[19] DUNFIELD J, KRISHNASWAM N. Bidirectional Typing[A/OL]. (2019). https://doi. org/10. 48550/arXiv. 1908. 05839.

[20] THE AGDA COMMUNITY. Agda2[CP/OL]. https://github. com/agda/agda.

[21] NORELL U. Towards a practical programming language based on dependent type theory[D/OL]. https://www. cse. chalmers. se/~ulfn/papers/thesis. pdf.

[22] COCKX J, ABEL A. Elaborating dependent (co)pattern matching[A/OL]. (2018). DOI:10. 1145/3236770.

[23] MOURA L de, AVIGAD J, KONG S, et al. Elaboration in Dependent Type Theory[A/OL]. (2015). https://arxiv. org/pdf/1505. 04324.

[24] COCKX J, DEVRIESE D, PIESSENS F. Pattern matching without K[A/OL]. (2014). DOI:10. 1145/2628136. 2628139.

[25] FORSBERG F N. Inductive-inductive definitions[D/OL]. 2013. http://cs. swan. ac. uk/~csfnf/thesis/thesis. pdf.

[26] ALTENKIRCH T, KAPOSI A. Type theory in type theory using quotient inductive types[A/OL]. (2016). https://doi. org/10. 1145/2837614. 2837638.

[27] SCHWARTZ M. Quantum Field Theory and the Standard Model[M]. Cambridge University Press, 2014.

[28] P. BOBBIN M, SHARLIN S, FEYZISHENDI P, et al. Formalizing Chemical Physics using the Lean Theorem Prover[A/OL]. (2023). https://arxiv. org/ abs/2210. 12150v5.

[29] TOOBY-SMITH J. Digitalizing Wick's theorem[A/OL]. (2025). https://arxiv. org/abs/2505. 07939.

[30] WOOTTERS W K, ZUREK W H. A single quantum cannot be cloned[J/OL]. Nature, 1982. DOI:10. 1038/299802a0.

[31] GIRARD J Y. Linear Logic[A/OL]. (1987). DOI:10. 1016/0304-3975(87)90045-4.

[32] FU P, KISHIDA K, SELINGER P. Linear Dependent Type Theory for Quantum Programming Languages[A/OL]. (2022). DOI:10. 46298/lmcs-18(3:28)2022.