引言

简单类型 λ 演算（Simply Typed Lambda Calculus，STLC）由阿隆佐·丘奇 [1] 引入，经哈斯凯尔·柯里 [2] 代数化研究，是有类型函数式编程的理论基础。其核心算法是**主类型推导**：给定项 t，计算最一般的类型判断 $\Gamma \vdash t : \tau$，使得所有其他类型判断都是它的特例。罗杰·欣德利 [3] 证明了 STLC 的主类型始终存在；罗宾·米尔纳 [4] 与达马斯-米尔纳 [5] 将该结论推广至多态 λ-演算。

罗宾逊合一算法 [6] 是大多数类型推导过程的计算核心。其终止性证明需要用到自由类型变量集合的势作为良基的度量，同时还需要一个压缩引理。

本文面向熟悉 Lean 基础、对类型论有一定了解的本科生，希望能对读者有所裨益。

本文内容一览：

1. STLC 定义及类型检查器；
2. 主类型推导算法的 Lean 4 实现；
3. 利用 Lean 依赖类型系统对罗宾逊合一算法终止性的证明；
4. 类型推导算法的正确性证明。

导读

2.1 什么是 λ 演算?

λ 演算是由阿隆佐·丘奇在 20 世纪 30 年代提出的形式系统，用于研究函数定义、函数应用和递归。它构成了函数式编程语言的理论基础。最基本的 λ 项由变量、抽象（函数定义）和应用（函数调用）组成：

```
1  t ::= x | λx.t | t t
```

例如，恒等函数写作 `λx.x`，它接受一个参数 `x` 并直接返回 `x`。

2.2 为什么要引入类型?

无类型 λ 演算虽然表达能力强，但允许一些“病态”的项，例如 `(λx.x x)(λx.x x)`，它会导致无限循环（Ω 组合子）。为了排除这类无意义的程序，人们引入了类型系统。简单类型 λ 演算（Simply Typed Lambda Calculus, STLC）是最基础的类型系统，它为每个项赋予一个类型，类型由基本类型和函数类型构成：

```
1  τ ::= α | τ → τ
```

类型规则确保函数应用时参数类型匹配。例如，恒等函数 `λx.x` 的类型可以是 `α → α`，其中 `α` 是一个类型变量，表示“任意类型”。

2.3 什么是主类型推导?

给定一个项，我们可能想知道它是否能被赋予一个类型，如果能，最一般的类型是什么？这就是**主类型推导**问题。主类型（principal type）是指：项的所有可能类型都是该类型的某个替换实例。例如，恒等函数的主类型是 `α → α`，任何其他类型如 `(β → β) → (β → β)` 都可以通过将 `α` 替换为 `β → β` 得到。

主类型推导算法通常分为两步：

1. 为项生成一组类型约束（例如，应用时要求左项类型为函数，且右项类型与函数参数一致）。
2. 通过合一算法求解这些约束，得到最一般的替换。

2.4 一个简单的例子

考虑项 `λf.λx.f x`（即函数应用组合子）。我们手动推导其主类型：

- 假设 `f` 的类型为 `α`，`x` 的类型为 `β`。

- 应用 `f x` 要求 `f` 是函数类型，设 `α = γ → δ`，且 `x` 的类型 `β` 必须等于 `γ`。
- 于是 `f x` 的类型为 `δ`。
- 抽象 `x`：`λx.f x` 的类型为 `β → δ`，但已知 `β = γ`，所以为 `γ → δ`。
- 再抽象 `f`：`λf.λx.f x` 的类型为 `(γ → δ) → γ → δ`。

用类型变量替换 `γ`，`δ` 为任意类型，得到最一般的类型 `(α → β) → α → β`。这就是该组合子的主类型。

2.5 本文的价值

虽然 STLC 的主类型推导早已被形式化验证[7]（例如在 Coq 中），但本文使用 **Lean 4**——一个年轻但强大的定理证明器——重新实现了整个算法并证明了其正确性。

通过阅读本文，读者将能够：

- 理解 STLC 的语法、类型规则和类型检查器；
- 掌握罗宾逊合一算法的实现及其终止性证明；
- 见证主类型推导算法的完整正确性证明。

尽管 Coq 等工具早已有类似工作，但 Lean 4 的现代设计、简洁的语法和强大的社区支持，使得这一经典结果以新的形式呈现，为类型理论和定理证明的学习者提供了宝贵的资源。

背景与基本定义

3.1 简单柯里类型与 λ 演算

定义 3.1.1：**简单柯里类型**（CurryType）定义如下：

$$\tau, \sigma ::= \varphi_p \mid \tau \to \sigma \tag{1}$$

其中 p 为类型变量的编号（本文使用字符表示），$\to$ 为右结合的函数类型构造符。

```lean
inductive CurryType : Type
| phi   : Char → CurryType
| arrow : CurryType → CurryType → CurryType
```

定义 3.1.2：**λ 表达式**（Term）定义如下：

$$t, s ::= x \mid \lambda x.t \mid t\ s \tag{2}$$

其中 x 为变量的编号（本文使用字符表示）。

```lean
inductive Term : Type
| var : Char → Term
| abs : Char → Term → Term
| app : Term → Term → Term
```

定义 3.1.3：**类型上下文**（TypeCtx）是由偶对 $(x, \tau) \in \mathrm{Char} \times \mathrm{CurryType}$ 组成的有限列表，加上一个计数器（用于生成新的类型变量）。记 $\Gamma(x)$ 为 Γ 中 x 对应的第一个类型（若存在）。空上下文为 $\Gamma_\emptyset = ([], \mathrm{A})$。 上下文扩展记作 $\Gamma + (x : \tau)$。

```lean
structure TypeCtx where
  env   : List (Char × CurryType)
  label : Char
```

`next ctx` 返回一个新的类型变量并更新计数器；`add c ty ctx` 向列表插入 (c, ty)。

定义 3.1.4：类型 τ 的**自由类型变量集合**（`vars`）定义为：

$$\begin{aligned} \mathrm{vars}(\varphi_p) &= \{\!|p|\!\} \\ \mathrm{vars}(\tau \to \sigma) &= \mathrm{vars}(\tau) \cup \mathrm{vars}(\sigma) \end{aligned} \tag{3}$$

定义 3.1.5: **类型赋值**（类型判断）$\Gamma \vdash t : \tau$ 由三条标准规则定义:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\Gamma + (x : \tau_1) \vdash m : \tau_2}{\Gamma \vdash \lambda x.m : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash m : \tau_1 \to \tau_2 \quad \Gamma \vdash n : \tau_1}{\Gamma \vdash m\ n : \tau_2} \tag{4}$$

3.2 类型替换

定义 3.2.1: **分配律**: 令$S : \text{CurryType} \to \text{CurryType}$, 若$S(\tau \to \sigma) = S(\tau) \to S(\sigma)$, 则称$S$满足分配律。

定义 3.2.2: **类型替换**是满足分配律的函数 $S : \text{CurryType} \to \text{CurryType}$。定义对上下文的逐点应用: $S(\Gamma) = \{\!\{(x, S(\tau)) \mid (x, \tau) \in \Gamma\}\!\}$, 在代码中通过 `applySubst` 实现。

定义 3.2.3: 对类型变量 α 和类型 σ, **替换** $\tau[\alpha := \sigma]$ 将 τ 中 α 的所有自由出现替换为 σ:

$$\begin{aligned} \varphi_\alpha[\alpha := \sigma] &= \sigma \\ \varphi_\beta[\alpha := \sigma] &= \varphi_\beta \quad (\beta \neq \alpha) \\ \tau \to \tau'[\alpha := \sigma] &= \tau[\alpha := \sigma] \to \tau'[\alpha := \sigma] \end{aligned} \tag{5}$$

实现

4.1 `check` 函数

类型检查函数 `check` 验证项是否符合类型赋值规则。它要求上下文已包含项中所有自由变量的类型绑定, `check` 的作用是描述何为正确的类型推导结果。

定义 4.1.1:

$$\begin{aligned}
\mathrm{check}(\Gamma, x) &= \Gamma(x) \\
\mathrm{check}(\Gamma, \lambda x.m) &= \begin{cases} \mathrm{none} & \text{if } \mathrm{check}(\Gamma, m) = \mathrm{none} \text{ or } \Gamma(x) = \mathrm{none} \\ \mathrm{some}(\tau' \to \tau) & \text{if } \mathrm{check}(\Gamma, m) = \mathrm{some}(\tau) \text{ and } \Gamma(x) = \mathrm{some}(\tau') \end{cases} \\
\mathrm{check}(\Gamma, m\ n) &= \begin{cases} \mathrm{some}(b) & \text{if } \mathrm{check}(\Gamma, m) = \mathrm{some}(a \to b) \text{ and } \mathrm{check}(\Gamma, n) = \mathrm{some}(a) \\ \mathrm{none} & \text{otherwise} \end{cases}
\end{aligned} \tag{6}$$

```lean
def check (ctx : TypeCtx) (t : Term) : Option CurryType :=
  match t with
  | .var c ⇒ ctx.lookup c
  | .abs x m ⇒
    match check ctx m with
    | none ⇒ none
    | some ty ⇒ match ctx.lookup x with
      | none    ⇒ none
      | some ty' ⇒ some (CurryType.arrow ty' ty)
  | .app m n ⇒
    match check ctx m, check ctx n with
    | some (.arrow a b), some ty_n ⇒ if a = ty_n then some b else none
    | _, _ ⇒ none
```

容易观察到实现与 定义 3.1.5 一一对应。

4.2 `Subst` 类型

`Subst left right` 是一个依赖类型结构，将替换函数 `apply` 与三个性质打包在一起。这三个性质是终止性证明的关键：

$$\begin{aligned}
&P_1 : \forall \tau.\ \mathrm{vars}(\mathrm{apply}(\tau)) \subseteq \mathrm{vars}(\mathrm{left}) \cup \mathrm{vars}(\mathrm{right}) \cup \mathrm{vars}(\tau) \\
&P_2 : \forall \tau. |\mathrm{vars}(\mathrm{apply}(\tau))| = |\mathrm{vars}(\mathrm{left}) \cup \mathrm{vars}(\mathrm{right}) \cup \mathrm{vars}(\tau)| \Rightarrow \mathrm{apply} = \mathrm{id} \\
&P_3 : \forall a, b.\ \mathrm{apply}(a \to b) = \mathrm{apply}(a) \to \mathrm{apply}(b)
\end{aligned} \tag{7}$$

P_1 限制了像的变量域；P_2 声称：若像的变量集合势与原集合相等，则该替换必为恒等替换（这是终止性的关键约束）；P_3 是分配律。`Subst` 有四种构造子：`id`（恒等）、`replace`/`replacer`（单变量替换）、`comp`（复合）。

> 何为依赖类型?
>
> 注意到 `Subst left right` 中的 `left` 不是一个类型，而是一个值。这意味着 `Subst` 的类型依赖具体值，这被称为依赖类型。

4.3 合一算法

定义 4.3.1：合一算法（`unify`）接受两个类型并返回一个可选的合一替换：

$$
\begin{aligned}
&\mathrm{unify}(\varphi_p, \varphi_p) = \mathrm{some}(\mathrm{id}) \\
&\mathrm{unify}(\varphi_p, \varphi_q) = \mathrm{some}\big(\mathrm{replace}(p, \varphi_q)\big) \quad \text{if } p \neq q \\
&\mathrm{unify}(\varphi_p, \tau) = \mathrm{some}(\mathrm{replace}(p, \tau)) \text{ if } \neg\ \mathrm{occ}(p, \tau) \\
&\mathrm{unify}(\varphi_p, \tau) = \mathrm{none} \text{ if } \mathrm{occ}(p, \tau) \\
&\mathrm{unify}(\tau, \varphi_p) = \mathrm{some}(\mathrm{replacer}(p, \tau)) \text{ if } \neg\ \mathrm{occ}(p, \tau) \\
&\mathrm{unify}(\tau \to \tau', \sigma \to \sigma') = \\
&\begin{cases}
\mathrm{none} & \text{if } \mathrm{unify}(\tau, \sigma) = \mathrm{none} \\
\mathrm{none} & \text{if } \mathrm{unify}(\tau, \sigma) = \mathrm{some}(S_1) \text{ and } \mathrm{unify}(S_1(\tau'), S_1(\sigma')) = \mathrm{none} \\
\mathrm{some}(\mathrm{comp}(S_1, S_2)) & \text{otherwise}
\end{cases}
\end{aligned}
\tag{8}
$$

4.3.1 合一算法的终止性

> 如何证明某递归算法终止?
>
> 一般来说，需要为输入定义一个度量，证明每次递归输入都会变小，最终达到基础情况。例如：阶乘每次会将输入减一，列表映射每次会将输入列表长度减一，等等。

如何证明合一算法的终止性？ 合一算法（`unify`）的终止性并不直观，因为它的递归调用可能作用于经过替换后的类型，而不再是原始项的子结构。例如，在箭头情形中，我们需要先合一参数类型得到替换 S_1，然后用S_1 作用于返回类型再合一。第二次调用的参数可能比原始输入更大（在变量数量上），这使得简单的“结构大小”度量失效。

4.3.1.1 度量的选择

我们采用字典序度量

$$
\mu(\tau_1, \tau_2) = \mathrm{vars}(\tau_1) \cup \mathrm{vars}(\tau_2)|, \mathrm{size}(\tau_1) + \mathrm{size}(\tau_2)) \tag{9}
$$

其中第一分量是自由类型变量集合的势，第二分量是类型的结构大小（即语法树的节点数）。字典序保证：若第一分量严格减小，则整体度量减小；若第一分量不变，则依赖第二分量减小。

4.3.1.2 终止性论证的要点

- **第一次调用** `unify(τ, σ)`：参数是原始类型的子项，结构大小严格减小，因此度量第二分量减小。无论第一分量如何，字典序整体下降。

- **第二次调用** `unify(S₁(τ'), S₁(σ'))`：由 P_1 可知，变量集合不会扩大；即 $|\text{vars}(S_1(\tau')) \cup \text{vars}(S_1(\sigma'))| < |\text{vars}(\tau') \cup \text{vars}(\sigma') \cup \text{vars}(\text{left}) \cup \text{vars}(\text{right})|$)，而后者正是第一次调用前变量集合的势。因此第一分量**不增**。
  - 若第一分量严格减小，则度量下降。
  - 若第一分量不变，则由 P_2 推出$S_1 = \text{id}$)。此时第二次调用的参数简化为原始的τ',σ')，它们是原始箭头类型的子项，结构大小严格减小，因此第二分量下降，度量下降。

4.3.1.3 在 Lean 中的实现

`Subst` 的每个构造子（如 `id`、`replace`、`comp`）都携带证明，确保它们满足上述三条性质。这样，当我们从 `unify` 返回一个替换时，它的类型本身就“记住”了这些性质，后续证明可以直接调用。例如，在第二次递归调用前，我们利用P_1 和P_2 分析S_1)，从而获得度量的递减保证。 这种将性质内嵌于类型的方式，使得终止性证明变得模块化且易于机器检查，也体现了依赖类型在形式化验证中的强大能力。

4.4 上下文合一

上下文合一将两个类型上下文中共有变量的类型两两合一。首先有递归辅助函数:

定义 4.4.1:

```lean
def unifyCtx' (commonVars : List (Char × CurryType × CurryType))
    : Option (List ExSubst) :=
    match commonVars with
    | [] ⇒ some []
    | (_, a, b) :: rest ⇒
      match unify a b with
      | none    ⇒ none
      | some s1 ⇒
        -- 将 s1 作用于所有剩余对，再递归处理
        -- 因为后续合一必须在前一个替换的基础上进行，否则可能引入不一致。
        let rest' := rest.map (fun (y, ty1, ty2) ⇒
                       (y, s1.apply ty1, s1.apply ty2))
        match unifyCtx' rest' with
        | none       ⇒ none
        | some substs ⇒ some (ExSubst.from s1 :: substs)
```

其中 `ExSubst` 是对 `Subst` 的类型擦除包装，使得不同索引类型的替换可存入同质列表。

`unifyCtx` 基于 `unifyCtx'` 构建：收集所有共有变量三元组$(x, \Gamma_1(x), \Gamma_2(x))$，调用 `unifyCtx'`，然后用 `foldl` 从左到右复合所有替换：

```
def unifyCtx (ctx1 ctx2 : TypeCtx) : Option (CtxSubst ctx1 ctx2) :=
  let commonVars := ctx1.env.filterMap (fun (x, a) ⇒
    (ctx2.env.lookup x).map (fun b ⇒ (x, a, b)))
  match unifyCtx' commonVars with
  | none ⇒ none
  | some exList ⇒
    let apply t := exList.foldl (fun acc ex ⇒ ex.cast acc) t
    some (CtxSubst.from commonVars apply …  ctx1 ctx2)
```

Lean

4.5 主类型推导算法 `pp'`

定义 4.5.1：辅助函数 $\mathrm{pp}' : \mathrm{TypeCtx} \times \mathrm{Term} \to \mathrm{Option}(\mathrm{PrincipalPair})$ 按项的结构分情形定义。

变量情形。 设 $(\alpha, \Gamma') = \mathrm{next}(\Gamma)$：

$$\mathrm{pp}'(\Gamma, x) = \begin{cases} \mathrm{some}(\Gamma, \tau) & \text{if } \Gamma(x) = \mathrm{some}(\tau) \\ \mathrm{some}(\Gamma' + (x : \alpha), \alpha) & \text{if } \Gamma(x) = \mathrm{none} \end{cases} \tag{10}$$

抽象情形。 设 $(\alpha, \Gamma'') = \mathrm{next}(\Gamma')$：

$$\mathrm{pp}'(\Gamma, \lambda x.m) = \begin{cases} \mathrm{none} & \text{if } \mathrm{pp}'(\Gamma, m) = \mathrm{none} \\ \mathrm{some}(\Gamma', \tau' \to \tau) & \text{if } \Gamma'(x) = \mathrm{some}(\tau') \\ \mathrm{some}(\Gamma'' + (x : \alpha), \alpha \to \tau) & \text{if } \Gamma'(x) = \mathrm{none} \end{cases} \tag{11}$$

应用情形。 设 $(\Gamma_m, \pi_1) = \mathrm{pp}'(\Gamma, m)$，$(\Gamma_n, \pi_2) = \mathrm{pp}'(\Gamma_m, n)$，$(\alpha, \Gamma_3) = \mathrm{next}(\Gamma_n)$，$S_1 = \mathrm{unify}(\pi_1, \pi_2 \to \alpha)$，$S_2 = \mathrm{unifyCtx}(S_1(\Gamma_m), S_1(\Gamma_3))$，则：

$$\mathrm{pp}'(\Gamma, m\ n) = \mathrm{some}(S_2(S_1(\Gamma_3)), S_2(S_1(\alpha))) \tag{12}$$

顶层函数$\mathrm{pp}(t) := \mathrm{pp}'(\Gamma_\emptyset, t)$。

结构性引理

本章收录支撑最终结果（主类型推导算法的正确性）的基础引理。

5.1 自由变量相关引理

引理 5.1.1 (类型的大小为正)：$\forall \tau, \mathrm{sz}(\tau) > 0$。

引理 5.1.2 (自由变量的包含性)：$\forall \tau, \tau', \mathrm{vars}(\tau) \subseteq \mathrm{vars}(\tau \to \tau')$。

引理 5.1.3 (替换后的自由变量范围)：若 $\neg\, \mathrm{occ}(\alpha, \sigma)$，则 $\forall \tau, \mathrm{vars}(\tau[\alpha := \sigma]) \subseteq \{\!|\alpha|\!\} \cup \mathrm{vars}(\sigma) \cup \mathrm{vars}(\tau)$。证明对 τ 作结构归纳。

引理 5.1.4 (非自由变量出现于自由变量集)：$c \in \mathrm{vars}(\tau) \Rightarrow \mathrm{occursIn}(c, \tau) = \mathrm{true}$。

引理 5.1.5 (替换的完全性)：若 $\neg\, \mathrm{occ}(\alpha, \sigma)$，则 $\forall \tau, \alpha \notin \mathrm{vars}(\tau[\alpha := \sigma])$。自由变量检查确保 α 不会通过 σ 被重新引入。

引理 5.1.6 (替换必然引入新类型变量)：若 $\neg\, \mathrm{occ}(\alpha, \sigma)$, 则 $\forall \tau, |\mathrm{vars}(\tau[\alpha := \sigma])| \neq |\{\!|\alpha|\!\} \cup \mathrm{vars}(\sigma) \cup \mathrm{vars}(\tau)|$。

证：设 $L = \mathrm{vars}(\tau[\alpha := \sigma])$，$R = \{\!|\alpha|\!\} \cup \mathrm{vars}(\sigma) \cup \mathrm{vars}(\tau)$。假设 $|L| = |R|$。由 引理 5.1.3 得 $L \subseteq R$；由 引理 5.1.5 得 $\alpha \notin L$；而 $\alpha \in R$，故 $L \subsetneq R$。那么，$|R| \leq |L|$ 与 $L \subseteq R$ 合并推得 $L = R$，矛盾。 ■

5.2 上下文操作的基本性质

引理 5.2.1 (新变量的唯一性)：对 `ctx` 中任何现有变量 $x \neq \alpha$，向上下文中添加绑定 (α, τ) 不影响对 x 的查找：

$$\mathrm{ctx}(x) = \mathrm{some}(\tau) \leftrightarrow (\mathrm{add}(y, \mathrm{next}(\mathrm{ctx}).\mathrm{fst}, \mathrm{next}(\mathrm{ctx}).\mathrm{snd}))(x) = \mathrm{some}(\tau). \quad (13)$$

`next ctx` 始终产生一个相对于 `ctx` 的新变量 α

5.3 类型检查的结构性质

引理 5.3.1 (检查不关心计数器)：`check` 只依赖上下文的 `env` 字段，与 `label` 无关：若 $\Gamma.\text{env} = \Gamma'.\text{env}$，则 $\text{check}(\Gamma, t) = \text{check}(\Gamma', t)$。

引理 5.3.2 (添加新变量不影响检查)：若 $\Gamma'(x) = \text{none}$，$\text{ctx} = \Gamma' + (x : \alpha)$，$m \neq \text{var } x$，且 $\text{check}(\Gamma', m) = \text{some}(\pi)$，则 $\text{check}(\text{ctx}, m) = \text{some}(\pi)$。

证：对 m 作结构归纳。

$\boldsymbol{m = \text{var } c,\ c \neq x}$：新上下文的 `env` 为 $(x, \alpha) :: \Gamma'.\text{env}$；由 `LawfulBEq`，查找 c 时跳过头部，回退到 $\Gamma'.\text{env}$，结果不变。

$\boldsymbol{m = \lambda y.m'}$：归纳假设适用于 m'（若 $m' = \text{var } x$ 则 $\text{check}(\Gamma', m') = \text{none}$，与前提矛盾）。查找 y 时，因 $x \neq y$（否则 $\Gamma'(x) = \text{some}(\ldots)$，矛盾），结果同样不变。

$\boldsymbol{m = m_1\ m_2}$：对 m_1, m_2 分别应用归纳假设。 ■

引理 5.3.3 (列表查找与映射交换)：若 $\text{lookup}(x, \text{xys}) = \text{some}(y)$，则 $\text{lookup}(x, \text{xys.map}(S)) = \text{some}(S(y))$。

引理 5.3.4 (替换的点应用可组合)：$\text{applySubst}(f, \text{applySubst}(g, \Gamma)) = \text{applySubst}(f \circ g, \Gamma)$。

定理 5.3.5 (替换保持检查结果)：若 $\text{check}(\Gamma, t) = \text{some}(\tau)$ 且 S 在箭头类型上满足分配律，则 $\text{check}(\text{applySubst}(S, \Gamma), t) = \text{some}(S(\tau))$。

证：对 t 作结构归纳。

$\boldsymbol{t = \text{var } c}$：由引理 5.3.3 作用于假设 $\Gamma(c) = \tau$ 即得。

$t = \lambda x.m$：设 $\mathrm{check}(\Gamma, m) = \mathrm{some}(\tau_m)$，$\Gamma(x) = \mathrm{some}(\tau')$。归纳假设给出 $\mathrm{check}(S(\Gamma), m) = \mathrm{some}(S(\tau_m))$；由 引理 5.3.3 得 $S(\Gamma)(x) = \mathrm{some}(S(\tau'))$。利用 S 的分配律 P_3 得 $\mathrm{some}(S(\tau') \to S(\tau_m)) = \mathrm{some}(S(\tau' \to \tau_m))$。

$t = m\ n$：设 $\mathrm{check}(\Gamma, m) = \mathrm{some}(a \to b)$，$\mathrm{check}(\Gamma, n) = \mathrm{some}(a)$。归纳假设给出 $\mathrm{check}(S(\Gamma), m) = \mathrm{some}(S(a) \to S(b))$（利用 P_3）和 $\mathrm{check}(S(\Gamma), n) = \mathrm{some}(S(a))$。由 `LawfulBEq` 和 `beq_iff_eq` 得分支返回 $\mathrm{some}(S(b)) = \mathrm{some}(S(\tau))$。 ■

5.4 辅助定义

定义 5.4.1：$\mathrm{distributes}(f) := \forall a, b,\quad f(a \to b) = f(a) \to f(b)$. 表示函数 f 在箭头类型上满足分配律。定理 5.3.5 等需要此条件来在类型替换下推进 `check`。

定义 5.4.2：$\mathrm{lookup\_inv}(f, \Gamma, \Gamma') := \forall x, \tau,\quad \Gamma(x) = \mathrm{some}(\tau) \Rightarrow \Gamma'(x) = \mathrm{some}(f(\tau))$. f 是从 Γ 到 Γ' 的**查找保持性**：若 Γ 将 τ 赋给 x，则 Γ' 将 $f(\tau)$ 赋给 x。

定义 5.4.3：$\mathrm{check\_inv}(f, \Gamma, \Gamma') := \forall t, \tau,\quad \mathrm{check}(\Gamma, t) = \mathrm{some}(\tau) \Rightarrow \mathrm{check}(\Gamma', t) = \mathrm{some}(f(\tau))$. f 是从 Γ 到 Γ' 的**类型检查保持性**。

定义 5.4.4：

$$\begin{aligned} \mathrm{equiv\_lookup\_on}(f, g, \Gamma) &:= \forall x, \tau,\quad \Gamma(x) = \mathrm{some}(\tau) \Rightarrow g(\tau) = f(\tau). \\ \mathrm{equiv\_check\_on}(f, g, \Gamma) &:= \forall t, \tau,\quad \mathrm{check}(\Gamma, t) = \mathrm{some}(\tau) \Rightarrow g(\tau) = f(\tau). \end{aligned} \tag{14}$$

两个替换 f, g 在 Γ 赋值（或 Γ 下可推导）的所有类型上相容。

定义 5.4.5：谓词 $\mathrm{occurs}(x, t)$ 表示变量 x 在项 t 中出现：

$$\begin{aligned} \mathrm{occurs}(x, \mathrm{var}\ y) &:= (x = y) \\ \mathrm{occurs}(x, \lambda y.m) &:= (x = y) \vee \mathrm{occurs}(x, m) \\ \mathrm{occurs}(x, m\ n) &:= \mathrm{occurs}(x, m) \vee \mathrm{occurs}(x, n). \end{aligned} \tag{15}$$

5.5 通用引理

引理 5.5.1 (替换后上下文查找)：若 $\Gamma(x) = \mathrm{some}(\tau)$，则 $(\mathrm{applySubst}(f, \Gamma))(x) = \mathrm{some}(f(\tau))$。（引理 5.3.3 的推论。）

引理 5.5.2 (左折叠保持相等)：若 $f(a) = f(b)$，则对任意替换列表 es：

$$(\text{es.foldl}(\lambda\ \text{acc}, \text{ex}.\ \text{ex} \circ \text{acc}, f))(a) = (\text{es.foldl}(\lambda\ \text{acc}, \text{ex}.\ \text{ex} \circ \text{acc}, f))(b). \quad (16)$$

即=可以穿透$(\text{es.foldl}(\lambda\ \text{acc}, \text{ex}.\ \text{ex} \circ \text{acc}, f))$。通过 `congrArg` 易证。

引理 5.5.3 (类型检查依赖于出现变量)：若 $\forall x, \text{occurs}(x, t) \Rightarrow \Gamma_1(x) = \Gamma_2(x)$，则 $\text{check}(\Gamma_1, t) = \text{check}(\Gamma_2, t)$。

类型检查仅依赖于在项中语法出现的变量的类型赋值。证明对 t 作结构归纳，每步只需比较 t 的子项中出现的变量。

引理 5.5.4 (检查成功则变量绑定)：若 $\text{check}(\Gamma, t) = \text{some}(\tau)$，则对 t 中出现的每个变量 x，有 $\exists \tau', \Gamma(x) = \text{some}(\tau')$。类型检查成功蕴含所有出现的变量均在上下文中绑定。

5.6 上下文合一正确性

引理 5.6.1 (上下文合一正确性)：若 $\text{unifyCtx'}(l) = \text{some}(\text{exs})$，$(x, a, b) \in l$，设 $F = \text{exs.foldl}(\lambda\ \text{acc}, \text{ex}.\ \text{ex} \circ \text{acc}, \text{id})$，则 $F(a) = F(b)$。

`unifyCtx'` 的复合替换确实合一了输入列表中的每一对。证明对列表 l 作归纳，利用 定理 6.1 和 引理 5.5.2。

定理 5.6.2 (公共变量类型一致)：若 $\text{unifyCtx}(\Gamma_1, \Gamma_2) = \text{some}(s)$，$\Gamma_1(x) = \text{some}(\tau_1)$，$\Gamma_2(x) = \text{some}(\tau_2)$，则 $s(\text{apply})(\tau_1) = s(\text{apply})(\tau_2)$。

`unifyCtx` 的输出替换将两个上下文公共变量的类型赋值为相同的像。该定理是应用情形证明的关键：它联系了 `unifyCtx` 的操作行为与类型检查的语义要求。

5.7 `pp'` 的性质

以下三个引理描述了 `pp'` 的输出上下文与输入上下文之间的关系。

引理 5.7.1 (输出上下文包含输入像)：若 $\mathrm{pp'}(\Gamma, m) = \mathrm{some}(\langle \Gamma', \tau \rangle)$，则存在替换 f 使得 $(\mathrm{applySubst}(f, \Gamma)).\mathrm{env} \subseteq \Gamma'.\mathrm{env}$。

输出上下文包含（在某个替换 f 的像下）输入上下文的所有绑定。证明对项作结构归纳，在每种情形下显式构造 f。

引理 5.7.2 (推理查找保持唯一性)：若 $\mathrm{pp'}(\Gamma, m) = \mathrm{some}(\langle \Gamma', \ldots \rangle)$，则存在替换 f 使得：

- $\mathrm{lookup\_inv}(f, \Gamma, \Gamma')$（查找保持性），
- $\mathrm{distributes}(f)$，
- 对任何其他满足 $\mathrm{lookup\_inv}(g, \Gamma, \Gamma')$ 的 g，均有 $\mathrm{equiv\_lookup\_on}(f, g, \Gamma)$。

此唯一性条件将推导的操作行为与语义正确性联系起来：输入上下文在输出上下文中有一个典范的、唯一的（在 Γ 赋值的域上）替换像。

引理 5.7.3 (推理类型检查保持唯一性)：若 $\mathrm{pp'}(\Gamma, m) = \mathrm{some}(\langle \Gamma', \ldots \rangle)$，则存在替换 f 使得：

- $\mathrm{check\_inv}(f, \Gamma, \Gamma')$，
- $\mathrm{distributes}(f)$，
- 对任何其他在 Γ 下可推导类型上满足 $\mathrm{check\_inv}(g, \Gamma, \Gamma')$ 的 g，均有 $\mathrm{equiv\_check\_on}(f, g, \Gamma)$

合一的可靠性

定理 6.1 (合一算法的正确性)：若 $\mathrm{unify}(\tau_1, \tau_2) = \mathrm{some}(S)$，则 $S(\tau_1) = S(\tau_2)$。

证：利用 Lean 4 的函数式归纳策略 `unify.induct` [8]，对 `unify` 的每个分支生成一个证明目标。

$\boldsymbol{\tau_1 = \tau_2 = \varphi_p}$：$S = \mathrm{id}$，显然。

$\boldsymbol{\tau_1 = \varphi_p,\ \tau_2 = \varphi_q,\ p \neq q}$：$S = \mathrm{replace}(p, \varphi_q)$。由 `replaceVar` 定义，$S(\varphi_p) = \varphi_q = S(\varphi_q)$。

$\boldsymbol{\tau_1 = \varphi_p,\ \tau_2 = \tau,\ \neg\, \mathrm{occ}(p, \tau)}$：$S = \mathrm{replace}(p, \tau)$。$S(\varphi_p) = \tau$；由辅助引理（对 τ 归纳，利用 $\neg\, \mathrm{occ}(p, \tau)$）得 $S(\tau) = \tau$。

自由变量检查失败或 `unify` 返回 `none`：无 S 须考虑。

$\boldsymbol{\tau_1 = a \to b,\ \tau_2 = c \to d,\ S = \mathrm{comp}(S_1, S_2)}$：由归纳假设，$S_1(a) = S_1(c)$ 且 $S_2(S_1(b)) = S_2(S_1(d))$。利用 S_1, S_2 的分配律 P_3：

$$\begin{aligned} S(a \to b) &= S_2(S_1(a) \to S_1(b)) = S_2(S_1(a)) \to S_2(S_1(b)) \\ S(c \to d) &= S_2(S_1(c) \to S_1(d)) = S_2(S_1(c)) \to S_2(S_1(d)) \end{aligned} \tag{17}$$

由上述两个归纳假设可知两侧相等。

对称情形 $\boldsymbol{\tau_1 = \tau,\ \tau_2 = \varphi_p}$：`replacer` 与 `replace` 使用相同的底层函数，论证对称。 ■

推导的正确性

引理 7.1（主类型推导算法的正确性）：若 $\mathrm{pp}'(\Gamma, t) = \mathrm{some}(p)$，则 $\mathrm{check}(p.\mathrm{ctx}, t) = \mathrm{some}(p.\mathrm{ty})$。

证：

变量情形与抽象情形。

- $\boldsymbol{t = \mathrm{var}\ c,\ \Gamma(c) = \mathrm{some}(\tau)}$：`pp'` 返回$(\Gamma, \tau)$。`check(Γ, var c) = Γ(c) = some τ`。
- $\boldsymbol{t = \mathrm{var}\ c,\ \Gamma(c) = \mathrm{none}}$：设$(\alpha, \Gamma') = \mathrm{next}(\Gamma)$。`pp'` 返回$(\Gamma' + (c : \alpha), \alpha)$。查找$c$ 在 $((c, \alpha) :: \mathrm{env})$中立即成功。

- $t = \lambda x.m,\ \Gamma'(x) = \mathbf{some}(\tau')$：归纳假设给出 `check(Γ', m) = some τ`；`pp'` 返回$(\Gamma', \tau' \to \tau)$，`check` 直接返回 `some (τ' → τ)`。
- $t = \lambda x.m,\ \Gamma'(x) = \mathbf{none}$：`pp'` 分配$\alpha = \text{next}(\Gamma')$，返回$(\Gamma'' + (x : \alpha), \alpha \to \tau)$。归纳假设加 `CheckInvUnderAdd`（唯一性条件$m \neq \text{var } x$由“若 $m = \text{var } x$ 则 `check(Γ', var x) = none`，与归纳假设矛盾”推得）将结论提升至扩展上下文。

应用情形。设 $t = m\ n$，算法输出：$(\Gamma_m, \tau_1) = \text{pp'}(\Gamma, m)$；$(\Gamma_n, \tau_2) = \text{pp'}(\Gamma_m, n)$；$\alpha = \text{next}(\Gamma_n).\text{fst}$；$S_1 = \text{unify}(\tau_1, \tau_2 \to \alpha)$；$S_2 = \text{unifyCtx}(S_1(\Gamma_m), S_1(\Gamma_n))$；$p.\text{ctx} = S_2(S_1(\Gamma_n))$，$p.\text{ty} = S_2(S_1(\alpha))$。

对 $\text{check}(p.\text{ctx}, m)$ 和 $\text{check}(p.\text{ctx}, n)$ 分情形。成功分支需证 `check p.ctx (app m n) = some p.ty`，论证分为以下关键环节：

环节一（上下文一致性）：对 m 中出现的每个变量 x，用引理 5.7.2（对 m）取得替换 g 使得 $\text{lookup\_inv}(g, \Gamma_m, \Gamma_n)$，再由 定理 5.6.2 得 S_2 等同 $S_1(\Gamma_m)(x)$ 与 $S_1(\Gamma_n)(x)$ 在 S_2 作用下的像；从而 $p.\text{ctx}(x) = (S_2 \circ S_1)(\Gamma_m)(x)$。由 引理 5.5.3，`check(p.ctx, m) = check(applySubst (S₂∘S₁) Γₘ, m)`。

环节二（归纳假设在替换上下文中的应用）：由对 m 的归纳假设和定理 5.3.5，$\text{check}(\text{applySubst}(S_2 \circ S_1, \Gamma_m), m) = \text{some}((S_2 \circ S_1)(\tau_1))$。

环节三（箭头类型）：由 定理 6.1，$S_1(\tau_1) = S_1(\tau_2 \to \alpha) = S_1(\tau_2) \to S_1(\alpha)$。再由 S_2 的分配律，$(S_2 \circ S_1)(\tau_1) = (S_2 \circ S_1)(\tau_2) \to (S_2 \circ S_1)(\alpha) = (S_2 \circ S_1)(\tau_2) \to p.\text{ty}$。

环节四（参数类型匹配）：对 n 类似地由环节一的对称版本和对 n 的归纳假设加 定理 5.3.5 得 `check(p.ctx, n) = some ((S₂∘S₁)(τ₂))`。箭头类型的定义域与参数类型吻合，`check(p.ctx, app m n)` 返回 `some p.ty`。

`check` 返回 `none` 的各分支通过 引理 5.7.2 + 引理 5.5.3 的反证法处理。 ■

定理 7.2 (`InferenceSound`)：若 $\text{pp}(t) = \text{some}(p)$，则 $\text{check}(p.\text{ctx}, t) = \text{some}(p.\text{ty})$。

证：$\text{pp}(t) = \text{pp'}(\Gamma_\emptyset, t)$。直接应用引理 7.1。 ■

推论 7.1：若 $\mathrm{pp}(t) = \mathrm{some}(p)$，则 $p.\mathrm{ctx} \vdash t : p.\mathrm{ty}$（按定义 2.5 的类型赋值规则）。

结论

本文给出了简单柯里类型 λ 式主类型推导算法的完整 Lean 4 形式化证明，包括：

1. STLC 语法、类型赋值规则与类型检查函数；
2. 罗宾逊合一算法与其终止性证明；
3. 主类型推导算法的完整正确性证明。

未来工作方向包括：将框架推广至带名字和递归的λ式（LCNR）或米尔纳的 ML 类型系统，证明主类型算法的完备性等。

AI 辅助声明

本文形式化的开发借助了语言大模型 DeepSeek [9] 的辅助。所有 AI 生成的证明草稿均经过仔细审查、修正与简化。

参考文献

[1] A. Church,《A Formulation of the Simple Theory of Types》, Journal of Symbolic Logic, 卷 5, 期 2, 页 56～68, 1940.

[2] H. B. Curry,《Functionality in Combinatory Logic》, Proceedings of the National Academy of Sciences, 卷 20, 期 11, 页 584～590, 1934.

[3] J. R. Hindley,《The Principal Type-Scheme of an Object in Combinatory Logic》, Transactions of the American Mathematical Society, 卷 146, 页 29～60, 1969.

[4] R. Milner,《A Theory of Type Polymorphism in Programming》, Journal of Computer and System Sciences, 卷 17, 期 3, 页 348～375, 1978.

[5] L. Damas 和 R. Milner,《Principal Type-Schemes for Functional Programs》, 收入 Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL), 1982, 页 207～212.

[6] J. A. Robinson,《A Machine-Oriented Logic Based on the Resolution Principle》, Journal of the ACM, 卷 12, 期 1, 页 23～41, 1965.

[7] C. Dubois 和 V. Ménissier-Morain,《Certification of a Type Inference Tool for ML: Damas–Milner within Coq》, J. Autom. Reasoning, 卷 23, 页 319～346, 1999, doi: 10.1023/A:1006285817788.

[8] L. de Moura 和 S. Ullrich,《The Lean 4 Theorem Prover and Programming Language》, 收入 Proceedings of the 28th International Conference on Automated Deduction (CADE), 2021, 页 625～635.

[9] DeepSeek-AI,《DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning》. [在线]. 载于: https://arxiv.org/abs/2501.12948