FREEST 编程语言的种类推断

Bernardo Almeida AndreiaMordidoVasco T.Vasconcelos里斯本大学理学院，LASIGE，葡萄牙{bpdalmeida,afmordido,vmvasconcelos}@ciencias.ulisboa.pt

我们提出了一种适用于 FREEST 编程语言的种类推断算法。该算法的输入是将部分或全部种类注释替换为种类变量的 FREEST 源代码。算法会为所有种类变量推断出具体的种类。我们先将 FREEST 测试套件中所有类型变量的种类注释替换为新鲜种类变量，再运行该算法，并得出结论：算法能够正确推断所有种类。我们还发现，程序员在 20% 的情况下并未选择最一般的种类，这并不令人意外。

1 引言

软件系统通常需要处理文件和通信通道等资源。这类资源的正确使用通常遵循某种协议，用以描述有效的交互模式。例如，文件应先打开，最终再关闭；关闭后绝不能再执行读取或写入操作。通信通道的情况也类似：通道打开后，双方交换消息，通道最终可能关闭；关闭后则不应再交换消息。会话类型 [6, 7, 15] 能够表达复杂的协议（例如文件和通道的协议），并确保程序遵守这些协议。

FREEST [1, 2, 3] 是一种基于 System F 的并发函数式编程语言，进程通过受上下文无关会话类型 [16] 约束的异构类型通道进行通信。上下文无关会话类型能够描述各种协议，例如算术表达式的序列化。下面来看算术表达式的数据类型。

```
1 数据 表达式 = 字面量 整数 | 加法 表达式 表达式 | 乘法 表达式 表达式
```

一个表达式 可以是一个包含整数的字面量（字面量 整数）、两个子表达式之和（加法 表达式 表达式），或两个子表达式之积（乘法 表达式 表达式）。要序列化类型为表达式 的值，我们可以使用如下会话类型。

```
2 类型 表达式通道 = ⊕{字面量通道 : !整数 , 加法通道 : 表达式通道 ; 表达式通道 , 乘法通道 : 表达式通道 ; 表达式通道}
```

缩写表达式通道 定义了从写入方视角看通道的类型。类型为表达式通道 的通道提供一组选项：字面量通道、加法通道和乘法通道。如果选择第一个选项，就必须发送一个整数（!整数）；而选择其他选项时，则需要发送两个（子）表达式。

现在，假设 序列化 是一个将 Exp序列化到通道 表达式通道上的函数。

```
3 序列 化：Exp → 表达式通道；a → a
```

该函数需要一个通道，其初始部分的类型为 表达式通道，随后其行为如 a：序列化 因此是关于 a 的多态函数。它会消耗通道的前端部分（类型为 表达式通道），并返回未使用的通道部分（类型为 a）。

IlariaCastellani 和 Alceste Scalas（编）：第 14 届面向并发与通信中心软件的编程语言方法研讨会（PLACES2023）。
EPTCS 378，2023 年，第 1–13 页，doi:10.4204/EPTCS.378.1

© B. Almeida、A. Mordido 和 V.T. Vasconcelos 本作品采用知识共享署名许可协议授权。

尽管上述代码看起来很简单，但在当前版本的 FREEST 中并无效。实际代码需要添加更多注解，以便区分函数式类型与会话类型，以及线性类型与不受限类型。这种区分通过根据种类对类型进行分类来实现。

在 FreeST 中，种类由一个重数和一个基本种类组成。重数控制一个值可以被使用的次数：恰好一次（线性，1）或零次或多次（不受限，$*$）。基本种类用于区分函数类型（T）和会话类型（S）。FreeST 需要种类的原因在于多态性。如果 $!\mathbf{Int};?\mathbf{Int}$ 显然是会话类型，而 $\mathbf{Int} \to \mathbf{Bool}$ 是函数类型，那么多态变量 a 则不然。它是会话类型还是函数类型？答案取决于 a 的基本种类：如果是 S，那么它就是会话类型；如果是 T，那么它就是函数类型。因此，种类对于判断诸如 $a;!\mathbf{Int}$ 这样的类型是否良构是必要的。

第 1 行定义的数据类型目前以带注释的形式写作如下。

```
4 data Exp:*T = Lit Int | Plus Exp Exp | Times Exp Exp
```

种类注释 $*T$ 表明该数据类型是函数式的。至于重数，我们选择不受限制的用法，因此可以根据需要任意多次使用。尽管如此，也可以将 Exp 声明为种类1T，在这种情况下，serialise 必须变为线性函数（类型为 Exp $\to$ ExpC;a $1 \to a$）。

Ex展开缩写并在第 2 行为数据类型添加注释后，我们得到以下类型 e.

```
5类型 ExpC:1S =递 归 a:1S. ⊕{LitC:!Int,PlusC:a;a,TimesC:a;a }
```

ExpC 定义了一个递归类型；当其主体（外部选择 $\oplus$）的种类是递归变量种类的子种类时，该类型便是良构的。在此例中，递归变量 ExpC被标注为1S，因为其主体本身就是一个线性会话。

最后，函数 serialise 目前写作如下。

```
6序列化: ∀a :1 S . Exp →表达式通道; a →a
```

多态变量 a 代表续延通道；它必须是线性会话。将a 标注为无限制会话 $*$S意味着它只能实例化为 Skip，这是唯一的无限制会话类型。

即使在 FREEST 语言的底层理论中种类不可或缺，它们也会使代码显得繁杂。第 1–3 行的代码更容易理解，编写起来也更快；程序员无需纠结于每种种类的细微差别。请注意，一旦推断出种类，$\forall$ 的前束出现就可以省略。本文提出的算法会为所有类型变量标注其种类，将第 1–3 行的代码转换为第 4–6 行的代码。

与 FREEST 关系更密切的工作包括 Quill [9], Affe [13], Alms [17], F$^\circ$ [8]、FuSe$^{\{\}}$ [11]和 Linear Haskell [4]。这些语言都具有次结构类型系统，用于处理线性类型、函数式类型和仿射类型（Affe 中的情况）。

Quill [9] 是一种具有线性类型且语法类似于 Haskell 的语言。Quill 的独特设计将线性类型与函数类型结合起来。与 FREEST 不同，Quill 不使用种类机制来区分线性类型和函数类型，而是使用类型谓词（或限定类型）来推理线性性。此外，Quill 不支持子种类。Quill 还拥有一个已被证明可靠且完备的类型推断算法。Affe [13] 是一种类 ML 语言，支持线性、仿射和不受限类型。与 Quill 类似，Affe 使用种类和受约束类型来区分线性类型与仿射类型。Affe 支持子种类，并具备完整的主类型推断能力。与 Affe 类似，Alms [17] 也是一种类 ML 语言，但它基于 System $F^{\omega}_{<:}$，即带子类型的高阶多态 λ演算。Alms 支持仿射类型和不受限类型。它具有丰富的种类系统，包括依赖种类、并集和交集。此外，Alms 支持 ML 模块，允许将不受限类型公开为仿射类型，从而为库程序员提供灵活性

m ::= $*$ | $\mathbf{1}$ | φ 重数υ::= S | T 预种类κ::= mυ | χ 种类$\sharp$::= ! | ? 极性$\star$::= $\oplus$ | & 视图$(\!|\cdot|\!)$ ::= $\{\cdot\}$ | $\langle\cdot\rangle$ 记录 T ::=Skip |End | $\sharp$T | $\star(\!|\ell\colon \mathrm{T}_\ell|\!)_{\ell\in L}$ |T ; T |$()_m$类型|T m$\to$T | $(\!|\ell\colon \mathrm{T}_\ell|\!)_{\ell\in L}$ | $\forall \mathrm{a}^\kappa.\mathrm{T}$ | $\mu \mathrm{a}^\kappa.\mathrm{T}$ |a e ::=$()_m$ |x | $\lambda_m \mathrm{x}:\mathrm{T}\,.\,\mathrm{e}$ | $\Lambda \mathrm{a}^\kappa.\mathrm{v}$ |e e | $\{\ell{=}\mathrm{e}_\ell\}_{\ell\in L}$ |let $\{\ell{=}\mathrm{x}_\ell\}_{\ell\in L}$ =e in e 表达式| ℓ e |let $()_m$ =e in e |case e of $\{\ell{\to}\mathrm{x}_\ell\}_{\ell\in L}$ |e[T] |match e with $\{\ell{\to}\mathrm{x}_\ell\}_{\ell\in L}$

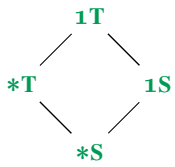
图 1：支持种类推断的种类和类型语法

并且配备了局部类型推断。F° [8] 是 System F 的扩展，使用种类来区分线性类型和不受限类型。与 Affe 和 Alms 类似，它支持子种类。与 FreeST 类似，但不同于 Affe，F° 不支持对种类进行量化。在上下文无关会话类型方面，与 FreeST 最接近的工作是 FuSe$^{\{\}}$ [11]。Padovani 提出了一种上下文无关会话类型的替代表述，其中代码和类型通过额外注解相互对应，而我们决定在 FreeST 中避免这种做法。Linear Haskell [4] 是一项将线性类型引入 Haskell 的提案。在 Linear Haskell 中，函数 $T \to U$ 和 T U 描述函数参数的使用方式。后一种形式受线性逻辑 [5], 启发，表示参数 T 恰好使用一次。在 FreeST 中，带注解的箭头 $T \ *{\to}\ U$ 或 $T \ \mathbf{1}{\to} U$ 描述函数的使用方式（无界使用或恰好使用一次）。FREEST 的种类判定系统区分会话类型和函数类型，也根据类型的使用方式将其归类为线性或不受限。其他系统会分别考虑这些概念（或只考虑其中一个）。我们的推断算法背后的思想与 Quill 和 Affe 类似，但具体细节大不相同，因为我们不使用类型限定符来推理线性性。

2 种类、类型和表达式的语法

本节简要介绍种类、类型和表达式的概念；有关详情，请参阅先前的研究 [1]。FreeST 使用一组基础类型变量（记作 a、b、c）和另一组基础标签（记作 k、 ℓ）。为进行种类推断，我们还使用重数变量（记作 φ）和种类变量（记作 χ）。种类、类型和表达式的语法见图 1。

重数 用于指明一个值可以使用的次数。重数可以是不受限的（$*$），表示可以使用任意次数；也可以是线性的（$\mathbf{1}$），表示恰好使用一次；还可以是重数变量（φ）。种类判定系统依赖于两种基本种类：S 表示会话类型，T 表示任意类型。种类可以是基本种类与重数的组合，也可以是种类变量 χ。由于非线性类型的值可以使用零次或多次，而线性类型的值必须恰好使用一次，因此显然，在需要线性值的地方可以使用不受限的值



。类似地，对基本种类的解释应当使会话类型（$*_S$, 1_S）可以替代任意类型（1_T）。非变量的子种类关系（记作$\kappa <: \kappa$）构成一个格，如图所示。

会话类型包括表示不进行通信的 Skip、表示已准备好关闭的通道的 End、输出（! T）和输入（? T）消息、内部选择（& $\{\ell: T_\ell\}_{\ell\in L}$）和外部选择（$\oplus\{\ell: T_\ell\}_{\ell\in L}$），以及顺序组合（T ;U）。函数类型由线性和不受限单位类型（$()_1$ 和 $()_*$）、线性和不受限函数（T $\to_1$ U 和 T $\to_*$ U）、记录 $\{\ell: T_\ell\}_{\ell\in L}$、变体 $\langle\ell: T_\ell\rangle_{\ell\in L}$ 以及全称类型 $\forall a^\kappa.T$ 构成。递归类型 $\mu a^\kappa.T$ 根据 κ，可以是会话类型，也可以是函数类型。类型变量 a 在递归类型中可指递归变量，在全称类型中可指多态变量。在函数体中捕获自由线性变量的函数本身必须是线性的。

表达式包括变量 x、项抽象 $\lambda_m x : T . e$ 和应用 e e、类型抽象 $\Lambda a^\kappa.v$ 和应用 e [T],记录 $\{\ell=e_\ell\}_{\ell\in L}$ 以及记录消除 let $\{\ell=x_\ell\}_{\ell\in L}$ = e in e、单位 $()_m$ 以及单位消除 let $()_m$ = e in e、变体中的注入 ℓ e 和变体消除 case e of $\{\ell\to x_\ell\}_{\ell\in L}$。通道操作的表达式包括通道创建 new T，以及对选择进行分支的 match e with $\{\ell\to x_\ell\}_{\ell\in L}$。其余通道操作——即 new、send、receive 和 select ℓ—— 都被视为常量（预定义变量）。

鉴于我们的目标是推断种类注释，读者或许会疑惑，为什么我们允许在源语言中使用种类注释，也就是在多态类型 $\forall a^\kappa.T$、递归类型 $\mu a^\kappa.T$以及类型抽象$\Lambda a^\kappa.v$中。程序员可以根据需要在源代码中提供种类注释。这些注释会传递给算法。对于省略的注释，则会在相应位置生成一个新鲜种类变量 χ。

3 种类推断

我们采用的种类推断方法遵循成熟的两步流程：第一步收集约束，第二步求解约束。约束生成步骤会生成两种形式的约束： $\kappa <: \kappa$ 和 $\varphi = \bigsqcup_{\ell\in L}\mathrm{mult}(\kappa_\ell)$。第一种形式表示子种类约束，第二种形式表示重数变量与给定重数集合的最小上界之间的等式。为提高可读性，我们用简写符号 $\varphi = \mathrm{mult}(\kappa)$ 表示 $\varphi = \bigsqcup\mathrm{mult}(\kappa)$，并用 $\bigsqcup$ 作为二元集合的中缀表示法。

从类型生成约束种类和重数约束由判断式表示 $\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}$。该判断式指出，类型$T$在种类上下文 κ 下具有种类 Δ（从类型变量到种类的映射），并生成约束集 $\mathscr{C}$。为明确区分输入和输出，我们使用下标**in**表示参数，使用下标**out** 表示结果。

我们解释图 2 中的约束生成规则的核心子集（完整集合见图 4）。规则 CG-Var 从种类上下文中读取类型变量 a （递归变量或多态变量）的种类，不生成额外的限制。规则 CG-Rec 处理递归类型，而递归类型可以是会话类型或函数式类型。分析类型 T时，递归变量的种类会被复制到种类上下文中。为确保递归类型体的种类 是递归变量种类 的子种类，会生成约束 $\kappa' <: \kappa$ 。规则 CG-Arrow 处理函数 $T\, m\to U$。它递归地将算法应用于 T 和 U，并将种类 m_T 赋予函数类型，其中m 来自箭头注释。规则 CG-Rcd 为记录中的所有元素构建种类和约束。它生成一个新的、未使用过的重数变量 φ。结果是种类 φ_T，约束集由所有 $\ell \in$ L 对应的 $\mathscr{C}_\ell$ 的并集，以及一个将变量 φ与 κ_ℓ的重数的最小上界关联起来的新约束组成。为确保 φ获得预期的重数，所有元素

$\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}}$

$$\text{CG-Var}\ \frac{}{\Delta, a : \kappa \vdash a : \kappa \Rightarrow \varnothing} \qquad \text{CG-Rec}\ \frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C}}{\Delta \vdash \mu\, a^{\kappa}.\, T : \kappa' \Rightarrow \mathscr{C} \cup \{\kappa' <: \kappa\}} \qquad \text{CG-Arrow}\ \frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U : \kappa_2 \Rightarrow \mathscr{C}_2}{\Delta \vdash T\ m\!\rightarrow U : m\mathbf{T} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2}$$

$$\text{CG-Rcd}\ \frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \varphi \text{ fresh} \qquad (\forall \ell \in L)}{\Delta \vdash \{\ell : T_\ell\}_{\ell \in L} : \varphi\mathbf{T} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell), \kappa_\ell <: \varphi\mathbf{T}\}} \qquad \text{CG-TAbs}\ \frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C} \qquad \varphi \text{ fresh}}{\Delta \vdash \forall a^{\kappa}.\, T : \varphi\mathbf{T} \Rightarrow \mathscr{C} \cup \{\varphi = \mathtt{mult}(\kappa')\}}$$

图 2：从类型生成约束

都必须是记录种类的子种类，即 $\varphi\mathbf{T}$。因此，如果记录中至少有一个条目是线性的，那么 φ也会被约束为线性。规则 CG-TAbs 在检查函数体 T时，将多态变量的种类添加到类型上下文中。随后，它将种类 $\varphi\mathbf{T}$赋予传入类型 $\forall a^{\kappa}.\, T$，其中新的重数变量 φ 表示类型 T的种类的重数。

类型运算符 `mult`只有在分析表达式后才能完全确定。在此阶段，它只能部分确定。应用于形如 $m\upsilon$的种类时，运算符 `mult` 会重写为重数 m，即 $\mathtt{mult}(m\upsilon) = m$。

例如，我们来考虑一个提取对中第一个元素的函数。

$$\mathtt{fst} : \forall a^{\chi_a}.\forall b^{\chi_b}.\{\mathsf{fst} : a, \mathsf{snd} : b\} *\!\rightarrow a$$

应用图 2 中的规则，得到约束集 $\{\varphi_1 = \mathrm{mult}(\varphi_2\mathbf{T})\ \varphi_2 = \mathrm{mult}(*\mathbf{T})\ \varphi_3 = \mathrm{mult}(\chi_a) \sqcup \mathrm{mult}(\chi_b)\}$。求解该约束集可得 $\{\varphi_1 = *, \varphi_2 = *, \varphi_3 = \mathrm{mult}(\chi_a) \sqcup \mathrm{mult}(\chi_b)\}$。我们假设种类变量 χ_a 和 χ_b 均为 $\mathbf{1}$T（种类格中的最大值），以消除其不确定性。于是，解为 $\{\varphi_1 = *, \varphi_2 = *, \varphi_3 = \mathbf{1}, \chi_a = \mathbf{1}\mathrm{T}\ \chi_b = \mathbf{1}\mathrm{T}\}$。

我们认为，为 χ_a 和 χ_b 指定 $\mathbf{1}\mathrm{T}$（最大种类）是首选方案，因为它在所有方案中限制最少。如果选择另一种种类，例如 $*\mathbf{T}$，那么将无法对线性值（种类为 $\mathbf{1}\mathrm{T}$ 的类型的值）调用函数 fst。这样一来，我们就会不合意地排除一些行为完全良好的程序。

但是，$\mathbf{1}\mathrm{T}$ 是变量 χ_a 和 χ_b 的最佳种类吗？答案取决于 fst 的定义 .

$$\mathtt{fst} = \Lambda a^{\chi_a}.\Lambda b^{\chi_b}.\lambda_{*}\, p : \{\mathsf{fst} : a, \mathsf{snd} : b\}.\, \mathsf{let}\ \{\mathsf{fst} = x, \mathsf{snd} = y\} = p\ \mathsf{in}\ x$$

考察表达式 $\mathsf{let}\ \{\mathsf{fst} = x, \mathsf{snd} = y\} = p\ \mathsf{in}\ x$ 可以发现，对的第二个元素 y 被丢弃了。因此，χ_b 必须是不受限的。如果选择 $\chi_b = \mathbf{1}\mathrm{T}$，那么 FREEST 在对该函数进行类型检查时会报告线性性违反。换句话说，约束 $\chi_b <: *\mathbf{T}$ 必须添加到约束集中，但仅检查 fst 的类型并不能提供足够的信息来生成这样的约束。在下文中，我们将介绍一些规则，通过检查表达式中的变量使用来生成诸如 $\chi_b <: *\mathbf{T}$ 这样的约束。

从表达式生成约束 表达式的约束由以下判断式推导得出$\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}$。该判断式表明，在种类上下文 Δ 和类型上下文 Γ下，表达式 e 的类型为 T。它会生成一个约束集 $\mathscr{C}$和一个使用上下文 Σ。类型上下文将项变量 x映射到类型 T；使用上下文则将项变量 x映射到其类型的种类 κ。使用

$$\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}}$$

$$\textsc{Inf-Var}\quad \frac{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma, x : T \vdash x : T \Rightarrow \mathscr{C} \mid \{x : \kappa\}}$$

$$\textsc{Inf-Abs}\quad \frac{\Delta \vdash T_1 : \kappa \Rightarrow \mathscr{C}_1 \quad \Delta \mid \Gamma, x : T_1 \vdash e : T_2 \Rightarrow \mathscr{C}_2 \mid \Sigma \quad \mathscr{C}_3 = \texttt{if isAbs } e \texttt{ then } \{\kappa <: m\mathbf{T}\} \texttt{ else } \varnothing}{\Delta \mid \Gamma \vdash \lambda_m x : T_1 . e : T_1 \; m\!\rightarrow T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Weaken}(\Sigma, x, \kappa) \mid \Sigma \setminus x}$$

$$\textsc{Inf-App}\quad \frac{\Delta \mid \Gamma \vdash e_1 : T_1 \; m\!\rightarrow T_2 \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : T_1 \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T_1 \; m\!\rightarrow T_2 : \kappa \Rightarrow \mathscr{C}_3}{\Delta \mid \Gamma \vdash e_1 \; e_2 : T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Merge}(\Sigma_1, \Sigma_2) \mid \Sigma_1 \cup \Sigma_2}$$

$$\textsc{Inf-RcdElim}\quad \frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 : \{\ell : T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma, (x_\ell : T_\ell)_{\ell \in L} \vdash e_2 : T \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T : \kappa \Rightarrow \mathscr{C}_3 \\ \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathscr{C}_\ell \cup \texttt{Merge}(\Sigma_1, \Sigma_2) \cup \texttt{Weaken}(\Sigma_2, x_\ell, \kappa_\ell) \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma \vdash \text{let } \{\ell = x_\ell\}_{\ell \in L} = e_1 \text{ in } e_2 : T \Rightarrow \mathscr{C} \mid (\Sigma_1 \cup \Sigma_2) \setminus \{x_\ell\}_{\ell \in L}}$$

图 3：从表达式生成约束

上下文有助于推理变量的使用情况：如果变量恰好使用一次，它可以是线性的；否则，它必须是不受限的。接下来，我们定义函数 弱化和 合并。前者检查表达式中是否使用了变量。如果变量未被使用，则返回带有约束 $\kappa <: *\mathbf{T}$的集合。后者检查变量是否被使用多次：如果变量出现在多个使用上下文中，它也必须是不受限的。

$$\texttt{Weaken}(\Sigma, x, \kappa) = \begin{cases} \varnothing & \text{if } x \in \Sigma \\ \{\kappa <: *\mathbf{T}\} & \text{otherwise} \end{cases} \qquad \texttt{Merge}(\Sigma_1, \Sigma_2) = \{\kappa <: *\mathbf{T} \mid x : \kappa \in \Sigma_1 \cap \Sigma_2\}$$

现在我们可以解释图 3 中表达式的规则了（完整规则集见图 5）。规则 Inf-Var 用于在给定的类型上下文中为变量指定类型。该规则要求类型上下文 Γ包含一项 $x : T$。与类型 T 相关的约束收集在 $\mathscr{C}$中。为反映 x 的使用情况，该规则返回单元素映射 $x : \kappa$，其中 κ 是 T 的种类。规则 Inf-Abs 处理抽象 $\lambda_m x : T_1 . e$。它递归调用关于 T 1 和 e 的判断，以收集约束集 $\mathscr{C}_1$、$\mathscr{C}_2$以及使用上下文 Σ。该规则使用一个新的谓词 isAbs e，当 e 是抽象时该谓词成立。接着，如果 e 是闭包，T_1 的种类必须是 mT 的子种类，其中 m 是该抽象的重数。这一限制确保不受限抽象不会闭合捕获线性值。结果是类型 $T_1\ m\!\rightarrow T_2$，以及由 $\mathscr{C}_1$、$\mathscr{C}_2$、$\mathscr{C}_3$ 和弱化的结果合并而成的约束集。弱化函数会检查变量在其作用域结束时是否未被使用。在此情形下，λ 抽象引入项变量 x，因此在作用域结束时，我们必须检查它的使用情况。规则 Inf-App 指出，若 e_1 的类型为 $T_1\ m\!\rightarrow T_2$，且 e_2 的类型为 T_1,，那么表达式 $e_1\ e_2$ 的类型为 T_2。约束 $\mathscr{C}$和使用上下文 Σ 通过合并对 e_1, e_2 和 T 进行种类推断的结果来计算。最终约束集是 Σ_1、Σ_2、Σ_3以及合并函数的结果的并集；该函数规定，同时出现在 Σ_1 和 Σ_2 中的任何变量都必须是不受限的。最终的使用上下文为$\Sigma_1 \cup \Sigma_2$。规则 Inf-RcdElim 综合了此前讨论的所有概念：它求值表达式 e_1 和 e_2,，收集 $\mathscr{C}_1$、$\mathscr{C}_2$ 以及 Σ_1、Σ_2。结果是 e_2,的类型，以及约束集 $\mathscr{C}$，它是以下各项的并集：

$\mathscr{C}_1$、$\mathscr{C}_2$、$\mathscr{C}_3$、结果是 合并 对 Σ_1 和 Σ_2，并对 弱化 应用于 Σ_2 以检查所有 $x_\ell : \kappa_\ell$ 是否为未使用变量。最终的使用上下文是 Σ_1 与 Σ_2 的组合，并移除所有 x_ℓ 对应的项。

分析函数 fst 的类型所产生的约束时，我们直观地得出结论：对中的第二个元素必须是不受限的，因为它会被丢弃。应用图 3 中的规则后，得到约束集 $\{\chi_b <: *\mathrm{T}\ \chi_a <: \varphi_1\mathrm{T}, \chi_b <: \varphi_1\mathrm{T}, \chi_a <: \varphi_0\mathrm{T}, \chi_b <: \varphi_0\mathrm{T}, \varphi_0 = \mathrm{mult}(\chi_a) \sqcup \mathrm{mult}(\chi_b)\ \varphi_1 = \mathrm{mult}(\chi_a) \sqcup \mathrm{mult}(\chi_b)\}$。该约束集的一个解是 $\{\varphi_0 = 1, \varphi_1 = 1, \chi_a = 1\mathrm{T}\ \chi_b = *\mathrm{T}\}$。正如我们所预测的，种类变量 χ_b 被设为 $*\mathrm{T}$。约束集是通过合并对所有子表达式应用判断式后生成的约束集，以及函数 Merge 和 Weaken 的结果计算得出的。首先，我们考察 Merge 函数：它将上下文 $\{\mathrm{p} : \kappa_p\}$ 和$\{\mathrm{x} : \chi_a\}$ 作为输入，并计算两个上下文的交集，为交集中的每个元素添加一个约束 $\kappa <: *\mathrm{T}$。此过程确保两个上下文中都使用的任何变量均为不受限的。Weaken 函数用于检查新引入的变量是否最终被丢弃。在我们的例子中，Weaken 将 $\mathrm{x} : \chi_a$和 $\mathrm{y} : \chi_b$与使用上下文 $\{\mathrm{p}: \kappa_p, \mathrm{x} : \chi_a\}$进行比较。对于 $\mathrm{y} : \chi_b$，Weaken 函数的处理如下：由于上下文中不存在 y，因此会添加一个新约束$\{\chi_b <: *\mathrm{T}\}$。另一方面，由于 x 已经在上下文中，因此不会创建约束。

约束求解 现在我们描述一种用于求解约束集的算法。

1. 将所有种类变量 χ初始化为种类格的最大值 $1\mathrm{T}$。同样，将所有重数变量 φ初始化为重数的最大值 1。将它们存储在 σ 中。
2. 遍历集合中的每个约束：(a)如果约束的形式为$\chi <: \kappa$，则将 σ 中 χ 的条目更新为 κ 和 $\sigma(\chi)$的最大下界。例如，如果$\sigma = [\chi \mapsto 1\mathrm{T}]$，并且我们正在分析约束$\chi <: *\mathrm{T}$，那么必须将 σ 中 χ 的值更新为 $1\mathrm{T} \sqcap *\mathrm{T} = *\mathrm{T}$。此步骤之后，我们将得到 $\sigma = [\chi \mapsto *\mathrm{T}]$。(b)如果约束的形式为 $\kappa <: \chi$，则检查 κ 以及 σ 中 χ 的种类是否处于子种类关系中；如果不是，则失败。例如，如果$\sigma = [\chi \mapsto 1\mathrm{T}]$，并且我们正在分析约束 $*\mathrm{T} <: \chi$，那么我们发现它处于子种类关系中，因为 $*\mathrm{T} <: 1\mathrm{T}$。对于 $\sigma = [\chi \mapsto 1\mathrm{T}]$，则会发生失败。(c)如果约束的形式为 $\kappa_1 <: \kappa_2$，且其中两个元素都不是种类变量，则检查 $\kappa_1 <: \kappa_2$ 是否处于子种类关系中；如果不是，则失败。如果没有失败，则从约束集中移除约束 $\kappa_1 <: \kappa_2$。(d) 如果约束是重数约束 $\varphi = \bigsqcup_{\ell \in L} \mathrm{mult}(\kappa_\ell)$，则计算这些重数的最小上界。如果任何 κ_ℓ 是种类变量（χ）或带有重数变量的基种类（$\varphi\mathrm{T}$），我们就从 σ 获取其种类（回想一下，根据步骤 1，所有变量都在 σ 中）。如果由此获得的种类比 σ 中 φ 的种类限制更多（例如，$*$ 相对于 $\sigma(\varphi) = 1$），则将其存储在 σ 中。如果 $\varphi = *$，则从集合中移除该约束。
3. 重复步骤 2 中的过程，直到不再需要进行更新。
4. 如果所有约束都已满足，则返回解 σ。否则，约束集不可满足。

I对于函数 fst，图 3 中各规则收集到的约束如下 .

$$\chi_1 <: \varphi_0\mathrm{T}, \chi_0 <: \varphi_0\mathrm{T}, \chi_1 <: \varphi_1\mathrm{T}, \chi_0 <: \varphi_1\mathrm{T}, \chi_1 <: *\mathrm{T},$$
$$\varphi_0 = \mathtt{mult}(\chi_0) \sqcup \mathtt{mult}(\chi_1), \varphi_1 = \mathtt{mult}(\chi_0) \sqcup \mathtt{mult}(\chi_1)$$

注释类别	源代码中的注解数量 源代码	更一般的注释数量 生成的注解
数据类型	129	0
类型缩写	206	7
全称类型	282	94
显式递归类型	23	10
类型抽象	30	25
总计	670	136

表1：注解分布

我们从 $\sigma = [\chi_0 \mapsto \mathbf{1}\mathrm{T}\ \chi_1 \mapsto \mathbf{1}\mathrm{T}\ \varphi_0 \mapsto \mathbf{1}, \varphi_1 \mapsto \mathbf{1}]$ 开始。接着选取约束 $\chi_1 <: \varphi_0\mathrm{T}$，并使用条目 2(a)。于是有 $\chi_1 <: \mathbf{1}\mathrm{T}$，因为 $\sigma(\varphi_0) = \mathbf{1}$。鉴于 $\sigma(\chi_0)$ 等于 $\mathbf{1}\mathrm{T}$，且子种类关系具有自反性，$\sigma(\varphi_0)$ 仍为 $\mathbf{1}\mathrm{T}$。处理第二个约束 $\chi_1 <: \varphi_0\mathrm{T}$ 的过程类似。我们分析约束 $\chi_1 <: \mathbf{1}\mathrm{T}$，因为 $\sigma(\varphi_0) = \mathbf{1}$。同样在这种情况下，条目 2(a) 不会改变 σ。接下来的两个约束 $\chi_1 <: \varphi_1\mathrm{T}$ 和 $\chi_0 <: \varphi_1\mathrm{T}$ 也通过条目 2(a) 处理。同样，σ 不需要更新。现在我们选取约束 $\chi_1 <: *\mathrm{T}$。根据条目 2(a)，算法计算 $*\mathrm{T}$ 和 $\mathbf{1}\mathrm{T}$ 的最大下界，结果为 $*\mathrm{T}$，因此相应地更新 σ。对于最后两个约束，我们使用条目 2(d)。我们从 σ 中读取 χ_0 和 χ_1 的值，并计算 $\mathrm{mult}(\mathbf{1}\mathrm{T})$ 和 $\mathrm{mult}(*\mathrm{T})$ 的最小上界，结果为 $\mathbf{1}$。χ_0 和 χ_1 的两个条目已经都是 $\mathbf{1}$，因此不更新 σ。由于我们已分析所有约束，且在本轮算法迭代中更新了 σ，因此尚未达到不动点，所以我们再次遍历每个约束。这一次没有进行任何更新，因此算法以 $\sigma = [\chi_0 \mapsto \mathbf{1}\mathrm{T}\ \chi_1 \mapsto *\mathrm{T}\ \varphi_0 \mapsto \mathbf{1}, \varphi_0 \mapsto \mathbf{1}]$ 终止。

该算法会反复更新种类变量和重数变量的值，直到无法再进行更新，也就是说，直到达到不动点。由于种类格是有限的，任何更新序列最终都必然收敛到一个不动点。基于同样的原因，每个约束只能被更新有限次。因此，该算法会在有限次迭代后终止。

约束生成算法的运行时间与输入表达式的大小呈线性关系；约束满足算法的运行时间呈二次关系。在最坏情况下，约束的数量等于表达式的大小。每个约束最多只能更新 σ 两次（当找到限制性更强的解时）。最坏情况发生在每次迭代中都由不同的约束执行更新时，这会迫使算法在每次迭代中分析所有约束。一种合理的优化是在完成项目 2(a) 和 2(b) 后，也将约束从约束集中移除，因为可以得出结论：它们无法将 σ 更新为限制性更强的解。由于更新只能执行常数次，算法的运行时间便与输入表达式的大小呈线性关系。

评估 我们实现了该算法，并将其集成到 FREEST 解释器中。随后，我们进行了一项评估，以检验该算法应用于 FREEST 源代码时的行为。评估过程中，我们将 FREEST 测试套件和标准库中的 232 个有效程序（共 9131 行代码）里的全部 670 个种类注释替换为新鲜种类变量，运行该算法，并检查算法能否推断出原有的注释。

种类注释分布在数据类型、类型缩写、全称类型、递归类型和类型抽象中。注释分布如表1所示。递归类型和类型抽象中的注释数量较少，这是因为它们通常以隐式方式引入：要么通过类型缩写（如第2行代码所示），要么由编译器细化引入类型

抽象 Λ, $^{\kappa}$.用于 与其签名一同出现的函数。

我们得出结论，算法正确地推断出了所有注解，并发现 670 个注解中有 136 个（即 20%）过于具体，可以放宽为更一般的种类。算法发现的更一般注解数量最多的是全称类型。我们认为这归因于程序员的保守倾向：如果我们在开发 Church 编码（大量使用多态），又何必要求线性类型变量呢？算法并未改进数据类型的种类：程序中通常以不受限的方式使用数据类型。此外，在测试套件中，它们通常作为具有不受限闭包的函数的第一个参数（用于模式匹配）出现，因此不可能是线性的。

例如　　对于算法建议使用更一般种类的示例，可以考虑函数组合。

```
dot : ∀ a:*T b:*T c:*T . (b → c) → (a → b) → a → c
dot f g x = f (g x)
```

如果我们只向dot提供不受限参数，那么多态变量a、b 和c 取种类 *T就没有理由不可以。不过，这样一来，我们就排除了将dot 应用于线性参数的程序。请看下面的程序。

```
dot : (b → c) → (a → b) → a → c
dot f g x = f (g x)

g:?在 t ; End →在 t
gc =令 (x,c) =接收 c封装于 c;x

main:输入 t
main =
  let (w,r) = new () in
  fork (\_ 1→ let w = send 5 w in close w);
  dot id g r
```

由于我们将多态变量 a 实例化为线性会话类型 ?Int;End，因此这个程序会被判定为不可类型化。由于没有理由认为 a、b 和c 应当是不受限的，算法会为这三个多态变量赋予种类 1T 。

4 未来工作

未来工作有几个方向。最紧迫的是证明该算法相对于类型系统的正确性。之后，借助种类推断，我们可以考虑引入第三种基种类，用于那些最终必须关闭（达到类型 End）的会话类型。在这种情况下，我们会要求函数 new 的参数种类属于新引入的种类。我们还计划研究对种类或重数进行量化的可能性，以提高编程的灵活性。

致谢 我们感谢匿名评审者提出的详细意见，这些意见极大地帮助我们改进了论文。本研究由 FCT 通过 SafeSessions 项目（编号 PTDC/CCI-COM/6453/2020）以及 LASIGE研究单位（编号 UIDB/00408/2020 和 UIDP/00408/2020）资助。

参考文献

[1] Bernardo Almeida、Andreia Mordido、PeterThiemann 和Vasco T.Vasconcelos（2022）：带有上下文无关会话类型的多态λ演算。《信息与计算》289（Part），第104948页，doi:10.1016/j.ic.2022. 104948。[2] Bernardo Almeida、Andreia Mordido 和 VascoT.Vasconcelos（2019）：FreeST，一种带有上下文无关会话类型的并发编程语言。https://freest-lang.github.io。最后访问于 2023 年。[3] Bernardo Almeida、Andreia Mordido 和 Vasco T. Vasconcelos（2019）：FreeST：函数式语言中的上下文无关会话类型。载于：PLACES，EPTCS291，第12–23 页，doi:10.4204/EPTCS.291.2。[4] Jean-Philippe Bernardy、Mathieu Boespflug、Ryan R. Newton、Simon Peyton Jones 和 ArnaudSpiwack（2018）：Linear Haskell：高阶多态语言中的实用线性性。ACM《编程语言论文集》2（POPL），第5:1–5:29 页，doi:10.1145/3158093。[5]Jean-YvesGirard（1987）：线性逻辑。《理论计算机科学》50，第 1–102 页，doi:10.1016/ 0304-3975(87)90045-4。[6] Kohei Honda（1993）：二元交互类型。载于：CONCUR，LNCS715，施普林格，第 509–523页，doi:10.1007/3-540-57208-2_35。[7] Kohei Honda、Vasco Thudichum Vasconcelos 和MakotoKubo（1998）：基于结构化通信的编程的语言原语和类型纪律。载于：ESOP，LNCS1381，施普林格，第122–138 页，doi:10.1007/BFb0053567。[8] Karl Mazurak、Jianzhou Zhao 和SteveZdancewic（2010）：System F 中的轻量级线性类型。载于Andrew Kennedy 和Nick Benton 编：TLDI 2010 论文集：2010年ACM SIGPLAN 国际类型、语言设计与实现研讨会，西班牙马德里，2010年 1月 23 日，ACM，第 77–88页，doi:10.1145/1708016.1708027。[9] J.GarrettMorris（2016）：兼得两者之长：毫不妥协的线性函数式编程。载于 Jacques Garrigue、Gabriele Keller 和EijiroSumii 编：《第 21届 ACM SIGPLAN函数式编程国际会议论文集》，ICFP2016，日本奈良，2016 年9 月 18–22 日，ACM，第 448–461页，doi:10.1145/2951913.2951925。[10] Martin Odersky、ChristophZenger和Matthias Zenger（2001）：彩色局部类型推断。载于 Chris Hankin和Dave Schmidt 编：《POPL 2001 会议记录：第28 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会》，英国伦敦，2001年 1月 17–19 日，ACM，第 41–53 页，doi:10. 1145/360204.360207。[11]Luca Padovani（2019）：上下文无关会话类型推断。《ACM 编程语言与系统汇刊》41(2)，第9:1–9:37 页，doi:10.1145/3229062。[12] BenjaminC.Pierce 和 David N. Turner（2000）：局部类型推断。《ACM编程语言与系统汇刊》22(1)，第 1–44 页，doi:10.1145/345099.345100。[13]Gabriel Radanne、Hannes Saffrich 和PeterThiemann（2020）：友善地弯曲以解放我们。ACM《编程语言论文集》4（ICFP），第 103:1–103:29 页，doi:10.1145/3408985。[14] John C. Reynolds（1974）：走向类型结构理论。载于 Bernard J. Robinet编：《编程研讨会论文集》，巴黎，法国，1974年 4月 9–11 日，《计算机科学讲义》19，施普林格，第 408–423页，doi:10.1007/3-540-06859-7_148。[15]Kaku Takeuchi、KoheiHonda 和 Makoto Kubo（1994）：一种基于交互的语言及其类型系统。载于：PARLE，LNCS817，施普林格，第 398–413页，doi:10.1007/3-540-58184-7_118。[16] PeterThiemann和VascoT. Vasconcelos（2016）：上下文无关会话类型。载于 Jacques Garrigue、Gabriele Keller 和 EijiroSumii 编：《第 21届 ACM SIGPLAN函数式编程国际会议论文集》，ICFP2016，日本奈良，2016 年9 月 18–22 日，ACM，第 462–475 页，doi:10.1145/2951913.2951926。

[17] Jesse A. Tov & Riccardo Pucella (2011)：实用仿射类型。载于 Thomas Ball & Mooly Sagiv 编：第 38 届 ACM SIGPLAN-SIGACT编程语言原理研讨会论文集，POPL2011，美国德克萨斯州奥斯汀，2011 年 1 月 26–28 日，ACM，第 447–458 页，doi:10.1145/1926385.1926436。[18] J. B. Wells (1994)：二阶 λ 演算中的可定型性与类型检查等价且不可判定。载于：LICS，IEEE Computer Society，第 176–185 页，doi:10.1109/LICS.1994.316068。[19] Andrew K. Wright (1995)：简单命令式多态。LISPSymb.Comput.8(4)，第 343–355 页。

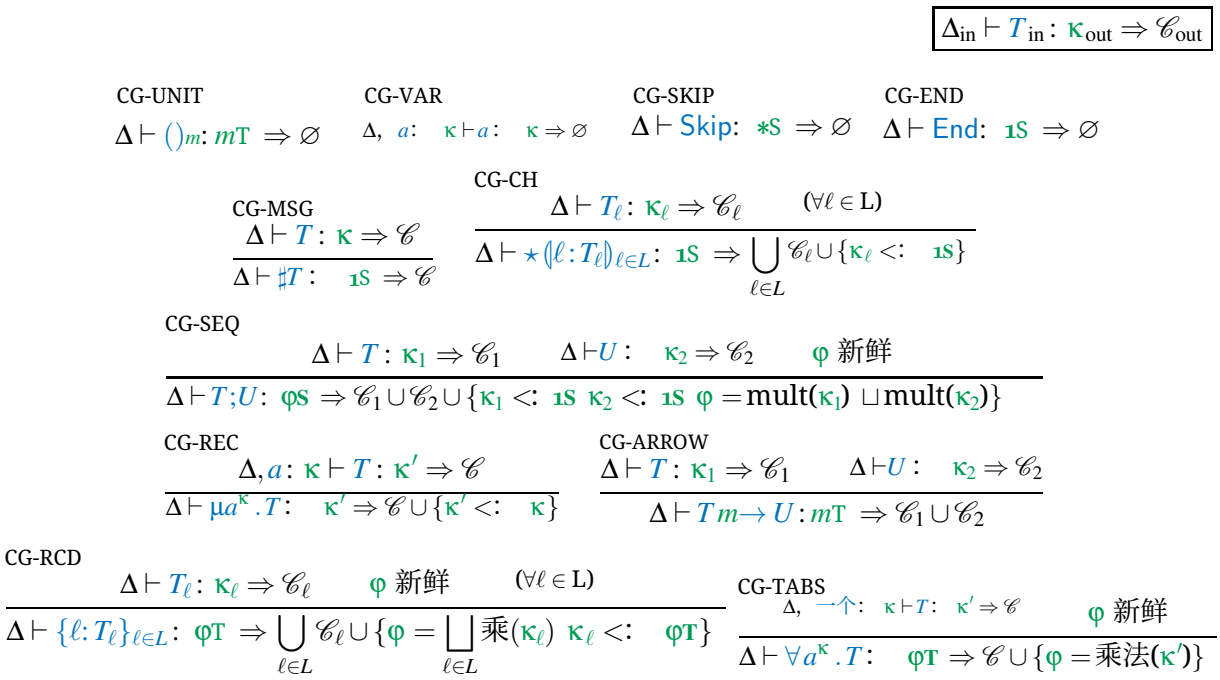


图 4：从类型生成约束（完整规则集）

$$\boxed{\Delta_{in} \mid \Gamma_{in} \vdash e_{in} : T_{out} \Rightarrow \mathscr{C}_{out} \mid \Sigma_{out}}$$

$$\textsc{Inf-Const}\ \frac{}{\Delta \vdash \texttt{typeof}(c) : \kappa \Rightarrow \mathscr{C}} \qquad \textsc{Inf-Var}\ \frac{}{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}$$

$$\frac{}{\Delta \mid \Gamma \vdash c : \texttt{typeof}(c) \Rightarrow \mathscr{C} \mid \varnothing} \qquad \frac{}{\Delta \mid \Gamma, x : T \vdash x : T \Rightarrow \mathscr{C} \mid \{x : \kappa\}}$$

INF-ABS

$$\frac{\Delta \vdash T_1 : \quad \kappa \Rightarrow \mathscr{C}_1 \qquad \Delta \mid \Gamma, x : \quad T_1 \vdash e : \quad T_2 \Rightarrow \mathscr{C}_2 \mid \Sigma \qquad \mathscr{C}_3 = \text{如果 } \texttt{isAbs}\, e \text{ 则 } \{\kappa <: m\mathbf{T}\} \text{否则 } \varnothing}{\Delta \mid \Gamma \vdash \lambda_m x : T_1 . e : T_1 \ m\!\rightarrow T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Weaken}(\Sigma, x, \kappa) \mid \Sigma \setminus \{x : \kappa\}}$$

$$\textsc{Inf-App}\ \frac{\Delta \mid \Gamma \vdash e_1 : T_1 \ m\!\rightarrow T_2 \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : T_1 \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T_1 \ m\!\rightarrow T_2 : \kappa \Rightarrow \mathscr{C}_3}{\Delta \mid \Gamma \vdash e_1\ e_2 : T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Merge}(\Sigma_1, \Sigma_2) \mid \Sigma_1 \cup \Sigma_2}$$

$$\textsc{Inf-TAbs}\ \frac{\Delta, a : \kappa \mid \Gamma \vdash v : T \Rightarrow \mathscr{C}_1 \mid \Sigma \qquad \Delta \vdash T : \kappa' \Rightarrow \mathscr{C}_2}{\Delta \mid \Gamma \vdash \Lambda a^{\kappa} . v : \forall a^{\kappa} . T \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \mid \Sigma}$$

$$\textsc{Inf-TApp}\ \frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \mid \Gamma \vdash e : \forall a^{\kappa_2} . U \Rightarrow \mathscr{C}_2 \mid \Sigma}{\Delta \mid \Gamma \vdash e\,[T] : U[T/a] \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \mid \Sigma}$$

INF-RCDELIM

$$\frac{\Delta \mid \Gamma \vdash e_1 : \quad \{\ell : \quad T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma, \quad (x_\ell : \quad T_\ell)\ _{\ell \in L} \vdash e_2 : \quad T \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T : \kappa \Rightarrow \mathscr{C}_3 \quad (\forall \ell \in L)}{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \text{合并}(\Sigma_1, \ \Sigma_2) \cup \text{弱化}(\Sigma_2, \ x_\ell, \ \kappa_\ell)}$$

$$\Delta \mid \Gamma \vdash \text{令 } \{\ell = x_\ell\}_{\ell \in L} = e_1 \text{ 于 } e_2 : \quad T \ \Rightarrow \mathscr{C} \mid (\Sigma_1 \cup \Sigma_2) \setminus \{x_\ell : \quad \kappa_\ell\}_{\ell \in L}$$

INF-RCD

$$\frac{\Delta \mid \Gamma \vdash e_\ell : \quad T_\ell \Rightarrow \mathscr{C}_\ell \mid \Sigma_\ell \qquad \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}'_\ell \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \{\ell = v_\ell\}_{\ell \in L} : \quad \{\ell : \quad T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_\ell \cup \mathscr{C}'_\ell \cup \text{合并}(\Sigma_\ell) \mid \bigcup_{\ell \in L} \Sigma_\ell}$$

INF-VARIANT

$$\frac{\Delta \mid \Gamma \vdash e : \quad T_k \Rightarrow \mathscr{C}_1 \mid \Sigma \qquad \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad k \in L \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash k\ e : \langle \ell : T_\ell \rangle_{\ell \in L} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_\ell \mid \Sigma}$$

无穷情况

$$\frac{\Delta \mid \Gamma \vdash e : \quad \langle \ell : \quad T_\ell \rangle_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_\ell : \quad T_\ell \ m\!\rightarrow T \ \Rightarrow \mathscr{C}_\ell \mid \Sigma_\ell \qquad \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}'_\ell \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \text{case } e \text{ of } \{\ell \rightarrow x_\ell\}_{\ell \in L} : T \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_\ell \cup \mathscr{C}'_\ell \mid \Sigma_1 \cup \Sigma_\ell}$$

INF-SEL

$$\frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \Delta \mid \Gamma \vdash e_\ell : T_\ell \ m\!\rightarrow T \Rightarrow \mathscr{C}'_\ell \mid \Sigma_\ell \qquad k \in L \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \text{select } k : \oplus \langle \ell : T_\ell \rangle_{\ell \in L} \ m\!\rightarrow T_k \Rightarrow \mathscr{C}_\ell \cup \mathscr{C}'_\ell \mid \bigcup_{\ell \in L} \Sigma_\ell}$$

INF-NEW

$$\frac{\varnothing \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma \vdash \text{new} T : \quad \{\text{fst} : \quad T, \quad \text{snd} : \quad T\} \Rightarrow \mathscr{C} \mid /0}$$

图5：由表达式生成约束（完整规则集）