arXiv:2602.06715v1 [cs.PL] 6 Feb 2026

实用精化会话类型推断（扩展版）

Toby Ueno[1] 和 Ankush Das[2]

[1] 英国爱丁堡大学
toby.ueno@ed.ac.uk[2] 美国马萨诸塞州波士顿市波士顿大学ankushd@bu.edu

摘要。 会话类型通过在类型中静态捕获进程间的交互协议，在并发消息传递系统中表达并确保安全通信。近期研究为会话类型加入了算术精化，使通信描述能够更加细致，但也增加了程序员的注释负担。为减轻这一负担，我们提出了一种适用于带算术精化的会话类型系统的类型推断算法。我们建立了会话类型子类型关系的理论，其中包括一种算法，并证明该算法相对于基于类型模拟的语义定义是可靠的。我们还提供了一种形式化推断算法，用于生成类型约束和算术约束，随后使用 Z3 SMT 求解器求解这些约束。该算法已基于 Rast 语言实现，并包含 3 项关键优化，使推断切实可行。我们通过在 6 个具有挑战性的基准测试上评估推断引擎的效能，涵盖从一元与二元自然数到线性 λ演算的范畴。我们展示了这些优化在促使 Z3 于合理时间内求解算术约束方面带来的性能优势。

关键词： 会话类型 · 类型推断 · 精化类型

1 引言

二元会话类型 [5, 6, 25–27]为表达并发消息传递系统中的通信协议提供了一种结构化方式。类型被赋予连接各个进程的双向通道，并描述通信的类型和方向。随后，类型检查可确保通道两端的进程遵循通道类型所描述的交互，即遵循类型和方向。因此，标准的类型安全定理保证了会话保真性（保持性），即协议在运行时不会遭到违反；还保证了无死锁性（进展性），即任何进程都不会陷入停滞。然而，即使存在递归，原始会话类型也只能指定非常基本的协议，因此出现了多种扩展，例如上下文无关会话类型 [40, 41], 、标签依赖会话类型 [42],以及一般依赖会话类型 [23, 43–45]。

其中一种扩展是按照 DML [18, 46] 的风格，将会话类型与精化 [13, 14, 16],结合起来，从而对消息传递程序进行轻量级验证。例如，考虑自然数的简单（未精化）会话类型:

$$type\ \mathsf{nat} \triangleq \oplus\{\mathbf{succ} : \mathsf{nat}, \mathbf{zero} : \mathbf{1}\}$$

由此类型支配的通信协议是：发送方可以产生**后继** 标签或**零** 标签，接收方必须据此进行分支处理。然而，除了这种基本行为之外，该类型没有提供任何信息。特别是，该类型没有明确规定会产生多少个**后继** 标签。为了解决这一局限，可以精化该类型:

$$type\ \mathsf{nat}[n] \triangleq \oplus\{\mathbf{succ} : ?\{n > 0\}.\, \mathsf{nat}[n-1], \mathbf{zero} : ?\{n = 0\}.\, \mathbf{1}\}$$

该类型由自然数 n索引，表示发送方产生的**后继**标签数量。发送**后继** 标签时，类型 **?**$\{n\ >0\}$.自然数$[n-1]$ 表示发送方必须给出一个证明 , 证明 $n > 0$; 之后，类型转换为自然数$[n-1]$，这意味着发送方现在将产生 **(**$n-1$**)** 个**后继** 标签。此类精化能够对协议进程进行轻量级验证；例如，可以定义一个加法进程接受两个自然数自然数$[m]$ 和自然数$[n]$, 并产生自然数$[m+n]$。

尽管精化很有用，但也给程序员带来了相当大的语法负担。首先，程序员需要为进程声明（如 add）添加注释，注明与其相连的所有通道的精化类型。其次，为了成功通过类型检查，他们需要在程序中添加显式的断言和假设。最后，如果类型注释有误，编译器通常几乎不会提供反馈。随着程序变得越来越复杂，注释也会随之复杂，进一步加重这项额外负担。

为了解决这种语法开销，本文提出了一种自动类型推断算法，用于精化会话类型。其目标是从进程的定义中推断其类型（可能是递归类型）。然而，这项任务在理论和实现上都面临若干挑战。首先，即使存在简单的线性算术精化，类型检查因而类型推断仍然是不可判定的 [14]。因此，任何实用算法都必须采用近似方法，并依赖适当的启发式方法。其次，会话类型遵循结构和等递归类型规则；类型相等不取决于类型名，而取决于类型结构。此外，程序中没有显式的折叠或展开来处理递归。这种更具表达力的类型特性给推断引擎带来了困难，因为它需要根据通信跟踪类型结构。第三，对一个进程而言，最一般的精化类型往往并非最有用的类型。举例来说，考虑一个产生自然数[0]的进程：它发送一个**零**标签后终止。这个进程最一般的类型是 $\oplus\{$**零**$:\ \mathbf{1}\}$，它是自然数[0], 的一个子类型，但可能不是最有用的类型。另一方面，产生最一般类型的任意超类型也并不理想。举例来说，我们可以推断出许多索引参数的可能取值；例如，加法进程的一种正确类型是接收自然数$[2m]$和自然数$[n]$, 并产生自然数$[2m+n]$。虽然这种类型是正确的，但它并非最精确的

加法进程类型。具体而言，当第一个参数为奇数时，这种类型就不适用。因此，最一般的类型并非总是有用——向用户报告的类型必须足够一般，能够适用于大量程序。

为了推断出更符合实际的进程类型，我们将会话类型中的子类型关系扩展到算术精化。为此，我们采用并推广了 Gay 和 Hole [19] 在其开创性工作中提出的子类型声明式定义。以此定义为基础，我们引入了一组算法式子类型规则，从中生成一组类型约束，据此可以推断出最一般的类型。我们证明该算法是可靠的：如果根据我们的算法，A 是 B 的子类型，那么根据我们的定义，它也必然是其子类型。接下来，我们在推断引擎中引入了一系列启发式方法，在精度与实用性之间谨慎权衡，以获得用户友好的类型。

我们在 Rast [11–13]编程语言之上实现了类型推断引擎。Rast 是一种用于分析会话类型化消息传递程序的并行与顺序复杂度的语言。Rast 支持类型精化和类型检查，但不支持类型推断。我们的实现分两个阶段进行推断：首先推断每个通道的基础会话类型，然后推断类型索引。将这两个阶段分开不仅简化了推断引擎，还能及早拒绝类型错误程序。推断的第一阶段收集子类型约束，并使用标准合一算法求解。第二阶段从子类型关系中提取算术约束，并将其交给 Z3 SMT 求解器 [17]。为提升类型推断的性能，我们在两个阶段中引入了三项关键优化：（*i*）传递性，用于消除进程定义中的中间类型；（*ii*）多项式模板，用于缩小 Z3 搜索算术表达式的空间；以及（*iii*）实数理论，用于更快地找到满足赋值，之后再尽可能将其转换为自然数。

我们分析推断引擎的有效性，以及这些优化在来自 Rast 语言的六个基准测试上所带来的性能提升。这些基准测试包括按其值索引的一元与二元自然数、按其大小索引的列表，以及按项的大小索引的线性 λ-演算表达式。我们推断若干标准进程的类型，例如数上的标准算术运算（如加法和加倍）、标准列表操作（如追加和拆分），以及演算中表达式的求值。实验表明，传递性在第一阶段带来了一个数量级的性能收益。此外，多项式模板和实数理论对于使第二阶段可行都很重要。若没有这两项优化，即使在最简单的示例上 Z3 也会超时。我们的结论是，这三项优化对于使推断可扩展且实用都是必要的。

总而言之，本文做出了以下贡献：

– 精化会话类型子类型关系的声明式定义，以及用于精化会话类型子类型判定的可靠算法（第4节），– 一种类型推断算法，可生成类型约束和算术约束，并附有其可靠性证明（第5节），

– 推断引擎的实现（第 6 节），以及三项优化；– 在六个具有挑战性的基准测试上对算法进行评估（第7节）。

本文的扩展版包含一个附录，其中列出了完整的子类型关系和推断规则，并附有可靠性证明。

2 动机示例

为了突出我们推断引擎面临的主要挑战，我们按照 Rast 语言的语法 [13, 16]介绍一系列动机示例。我们的设定基于二进制 会话类型 [5, 6]，其中包含进程，它们通过双向通道交换消息。进程分为提供方 和客户端，分别构成通道的两个端点：前者向后者提供通道，后者则 consume 它。一般而言，客户端通过生成进程 来创建新通道，从而生成相应的提供方；随后，双方通过这个新创建的通道连接起来。这也确立了进程之间的父子 关系：客户端充当父进程，而新生成的提供方充当子进程。从这个意义上说，一个客户端进程可以有多个子进程，即有多个通道将其连接到提供方；而一个提供方只能有一个客户端，即一个父进程。随着进程数量的增加，这种父子连接会扩展为树形拓扑，但绝不会形成环路，从而避免死锁。

这种对偶性通过推广线性序列式 $A_1, A_2, \ldots, A_n \vdash C$ 并为每个命题分配通道名称来表示，如下所示：

$$(x_1 : A_1), (x_2 : A_2), \ldots, (x_n : A_n) \vdash (z : C)$$

上述判断描述了一个进程：它是通道 $x_1, \ldots, x_n$的客户端，这些通道的类型分别为 $A_1, \ldots, A_n$；同时，它也是类型为 C 的通道 z 的提供方。为了进行通信，提供方和客户端会在该通道上执行相互对偶、相匹配的操作，而这些操作恰好由会话类型所规定。正如通道的含义会因其是被提供还是被消费（即位于序列式的左侧还是右侧）而改变，通道的类型也会随之改变。进程通过通道进行通信时，通道的类型会不断演变，从而描述的不是单次通信，而是一消息会话。

回顾自然数的例子，其基本类型定义如下：

$$\mathit{type}\ \mathsf{nat} = \oplus\{\mathbf{zero} : \mathbf{1},\ \mathbf{succ} : \mathsf{nat}\}$$

$\oplus$ 类型与一组标签 {**零**,**自然数**} 相关联，通道提供方可以从中选择一个发送。消息传输之后的类型由冒号表示：如果提供方发送**零**，续延类型为**1**，表示进程终止并关闭通道；如果提供方发送**后继**，类型则递归回到自然数，这意味着协议会简单地重复。

在 Rast 中，可以通过一条声明 来描述进程类型，并通过一个定义 来描述实现。下面给出名为 two 的进程的声明和定义，该进程表示数字二：

```
decl two : . ⊢ (x: nat)
proc x ← two = x.succ; x.succ; x.zero; close x
```

第一行展示了声明：转向符左侧（⊢）显示进程所使用的通道（及其类型），右侧则显示所提供的通道和类型。 two进程不使用任何通道，因此左侧是一个点（·），并提供类型为 nat的通道 x （记作 $x \leftarrow$ ）。 two 的进程定义（以 proc 开头）表示该进程在通道 x上恰好发送两个**后继** 标签（x.k 表示在通道 x 上发送标签 k），随后发送一个**零** 标签，最终关闭通道 x并终止。没有类型推断时，Rast 仅支持类型检查，这意味着程序员必须同时提供进程声明和定义。本工作的目标是根据定义自动生成声明，采用类似 OCaml 和 SML 等语言的方式。

推断中的结构子类型起初，我们或许会尝试采用一种朴素的类型推断方法，以得出最一般的类型。例如，如果我们在所提供的通道 x 上发送标签 k，那么 x 最一般的类型必定是 $\oplus\{k\colon A_k\}$，其中 A_k 是通信后通道 x 的类型。如果我们将这一方法应用于上面的进程 two，就会得到以下类型：

```
proc x ← two = x.succ;          % x: ⊕{后继: ⊕{后继: ⊕{零: 1}}}
               x.succ;          % x: ⊕{后继: ⊕{零: 1}}
               x.zero;          % x: ⊕{零: 1}
               close x          % x: 1
```

从底向上理解这种类型推断会更容易，这样可以逐步构建推断出的类型。 x 的最后一个操作是关闭，因此此时它的类型必定是 **1**，如右侧注释所示。接下来，我们发送标签**零**，这意味着 x 最一般的类型必定是 $\oplus\{\textbf{零}\colon \mathbf{1}\}$，因为这是允许发送标签**零**的最简单类型。沿着这一思路处理接下来的两个标签，我们得到 x 的类型为 $\oplus\{\textbf{后继}\colon \oplus\{\textbf{后继}\colon \oplus\{\textbf{零}\colon \mathbf{1}\}\}\}$。

然而，这种类型由于多种原因并不令人满意。首先，这种类型不可泛化：one、 two、 three.....的进程类型都将各不相同，进而妨碍代码复用。此外，程序员更倾向于使用自然数作为进程类型。其次，这种类型无法用于像 add 这样更大的进程，该进程需要针对一般的自然数来定义，而不是针对每一个特定的数和类型。其次，尽管生成的类型（或最一般类型）对于类型检查而言是正确的，但在类型错误消息中使用这种类型会不必要地冗长，并且随着程序复杂度的增长，将逐渐变得更加难以理解。

由此可见，子类型关系的概念不可或缺：我们希望有这样一个系统，其中 $\oplus\{\textbf{后继}\colon \oplus\{\textbf{后继}\colon \oplus\{\textbf{零}\colon \mathbf{1}\}\}\}$ 是自然数的子类型。这种子类型化系统还必须体现类型的结构化本质，而非名义本质。这与函数式语言的精化类型系统（如 Liquid Types [37] 和 DML [46] ）形成对比；在这些系统中，基本类型可以通过标准的 Hindley-Milner 推断算法推断出来。正如后文将会看到的

我们的子类型算法中并没有结构化类型系统的基本类型概念。具体而言，这一决定意味着，像 **零** 和**后继** 这样的消息标签并不专属于自然数类型，这与大多数函数式语言中代数数据类型的构造子不同。例如，我们可以如下对多种类型使用相同的构造子：

$$\mathit{type}\ \mathsf{even} = \oplus\{\mathbf{zero} : \mathbf{1},\ \mathbf{succ} : \mathsf{odd}\} \qquad \mathit{type}\ \mathsf{odd} = \oplus\{\mathbf{succ} : \mathsf{even}\}$$

偶数和自然数都是进程 `two` 的有效类型。因此，我们的子类型关系算法必须分析类型的结构（这些结构可能相互递归）。我们的方法是推断程序员在签名中提供的最一般类型。有了子类型关系后，我们可以说，进程提供一个通道，其类型为最一般类型或任何有效的超类型，从而在保证正确性的同时显著提高可读性。

2.1 引入精化

存在算术精化时，子类型关系的概念会变得更加复杂。例如，重新考察自然数类型的精化：

$$\mathsf{type}\ \mathsf{nat}[n] = \oplus\{\mathbf{zero} : ?\{n = 0\}.\mathbf{1},\ \mathbf{succ} : ?\{n > 0\}.\mathsf{nat}[n-1]\}$$

这些精化规定：当 $n = 0$ 时，提供方只允许发送**零**标签；当 $n > 0$ 时，则只允许发送 **后继**。通过指定类型上的精化，我们能够传达关于进程的更多类型级信息：例如，我们可以保证，若提供类型为 $\mathsf{nat}[2]$，则经过修改的 `two` 进程恰好会发送两条**后继**消息。

```
decl two : . ⊢  (x: nat[2])
proc x ← two = x.succ; assert x {2 > 0};
               x.succ; assert x {1 > 0};
               x.zero; assert x {0 = 0}; close x
```

我们描述进程的实现，以展示如何满足证明义务。每次发送**后继** 消息时，我们都需要发送一个证明，证明$n > 0$，这通过断言来实现。因此，进程首先断言，发送**后继** 后，当 `x` 的类型为自然数$[2]$时，满足 $\{2 > 0\}$ 。此后，类型转换为自然数$[1],$，因此进程会再发送一条**后继** 消息，随后断言 $\{1 > 0\}$。最后，类型转换为自然数$[0],$，因此进程发送**零** 标签，并断言 $\{0 = 0\}$ ，然后关闭。

从这个示例可以看出，这些断言给程序员带来了显著的语法开销。为消除这一负担，我们的推断算法还会执行程序重构，自动插入这些断言。我们的推断引擎通过遵循精化后的自然数类型的类型结构来推导这些断言。值得注意的是，即使多个类型（例如偶数、奇数、自然数）使用相同的标签构造子，也能有效地完成这种重构。

精化推断。除了推断自然数等类型名称外，我们还必须推断这些类型上的精化。与 two 进程中的情况不同，这些精化不一定是数字；它们可以是包含自由算术变量的任意表达式。以下这个将两个数字相加的进程，展示的是重构之后、精化推断之前的形式：

```
decl add[m][n]: (x: nat[e0(m,n)]) (y: nat[e1(m,n)]) ⊢  (z: nat[e2(m,n)])
proc z ← add[m][n] x y = case x (
    zero  ⇒  assume x {e0(m,n) = 0}; wait x; z ↔ y
  | succ  ⇒  assume x {e0(m,n) > 0};
        z.succ; assert z {e2(m,n) > 0};
        z ← add[m-1][n] x y )
```

add 进程的情况会分析 x。如果 x 发送 **零** 标签（表示 $e_0(m,n) = 0$），我们只需将通道 y 和 z 对应起来（相当于在函数式环境中返回 y）。另一方面，在 **后继** 分支中，我们假设 x 上的精化大于 0，于是向 z 发送 **后继** 标签，随后断言 $e_2(m,n) > 0$（这是自然数类型所要求的）。最后，我们通过再次调用加法进程进行递归。

假设我们已经确定每个通道的类型都是自然数类型；我们仍须在变量 m、n 中找到表达式 e_0、e_1和 e_2，使所有断言都成立。我们将这一非平凡任务交给 Z3：首先确定候选表达式可能受到的约束，例如，$e_0(m,n) = 0$ 意味着 $e_2(m,n) = e_1(m,n)$，因为在零分支中，通道 y 被转发给了 z。同样，在后继分支中，我们得到 $e_0(m,n) > 0$ 意味着 $e_2(m,n) > 0$。有了这些约束后，我们向 Z3 查询满足精化条件的满足赋值。对于 add，我们预期 $e_0(m,n) = m$、$e_1(m,n) = n$，以及 $e_2(m,n) = m + n$。

自然地，我们最初尝试将 e_i's 视为未解释函数，Z3 能够求解这类函数。我们只需将约束发送给 Z3，并要求它给出一个满足赋值。然而在实践中，这种方法并不可行：求解器要么超时，要么返回无法用我们的精化层表示的非多项式表达式，例如条件分支结构。因此，我们首先用多项式模板替换每个表达式：例如，我们将 $e_0(m,n)$ 转换为 $c_0m+c_1n+c_2$，并只要求 Z3 求解系数c_i's。我们语言中的任何表达式都具有这种形式，因此不会损失一般性；而且我们发现，这项优化显著提高了 Z3 的可靠性和性能。事实上，多项式模板化只是使我们的算法能够处理相当复杂示例的一项优化；其他优化详见第 6 节。

3 精化会话类型背景

我们介绍用于实现类型推断的会话类型语言。首先介绍基本会话类型，它们构成了通信协议的核心；随后介绍精化层，它通过加入发送和接收证明及见证的构造，扩展了核心内容。附录 A 中给出了每种类型构造子的详细示例。

3.1 基本会话类型

我们的基本会话类型系统与直觉线性逻辑 [5] 相对应。这些类型允许交换标签、其他通道以及关闭操作，即终止消息。类型和进程语法定义如下：

$A,\ B ::= \oplus\{\ell\colon A_\ell\}_{\ell\in L} \mid \&\{\ell\colon A_\ell\}_{\ell\in L}$ (内部选择和外部选择)
$\mid A \otimes B \mid A \multimap B$ (张量和线性蕴含)$\mid \mathbf{1} \mid V$ (单位类型和类型变量)$P ::= x.k\,;\ P \mid$ case $x\ (\ell \Rightarrow P_\ell)$（发送和接收标签）$\mid$ send $x\ e\,;\ P \mid y \leftarrow$ recv $x\,;\ P$ (发送和接收通道)$\mid$ 关闭 $x \mid$ 等待 $x;\ P$ (关闭和等待关闭)
$\mid x \leftrightarrow y \mid x \leftarrow f\ y\,;\ P$ (转发和生成进程)

–

结构类型分为成对的类型——也就是说，$\oplus$ 和 & 构成一对，$\otimes$ 和 $\multimap$也构成一对。某个类型的配对类型会以相反的方向展现相同类型的通信。例如，通道x 的提供方，其类型为 $\oplus\{\ell : A_\ell\}_{\ell\in L}$，必须通过表达式$x.k$ 发送标签 $k \in L$ 之一，并继续执行 A_k。反之，如果提供方提供 $x : \&\{\ell : A_\ell\}_{\ell\in L}$，它就会接收 L中的一个标签，并通过表达式 case $x\ (\ell \Rightarrow P_\ell)$ 对其进行分支分析；进程继续执行 P_ℓ，而通道则继续为 A_ℓ。其他类型 $\otimes$ 和 $\multimap$与此类似，但它们发送的是通道而不是标签：类型为 $x\colon A \otimes B$ 的提供方会通过 send $x\ e$发送一个类型为 A 的通道 e。反过来，类型为 $x : A \multimap B$的提供方会接收一个类型为 A 的通道，并通过 $y \leftarrow$ recv x将其绑定到 y。最后，提供方若提供 $(x : \mathbf{1})$，就必须使用 close x 来终止通道 x 和进程；客户端则采用 wait $x\,;\ P$的形式，也就是等待 x 关闭，然后继续执行 P。其余语法部分 $x \leftrightarrow y$ 和 $x \leftarrow f\ y$分别表示转发和生成进程。转发 $x \leftrightarrow y$ 表示将通道x 和 y 视为同一通道，并在二者之间传递所有消息；而生成新进程 f 则类似于其他语言中的函数调用：我们向其提供通道 y，并接收一个通道 x 作为返回值。最后请注意，一个进程提供一个通道，并消耗任意数量的其他通道。类型的角色会根据它是被提供还是被消耗而反转。

3.2 精化层

类型通过以下类型和进程构造进行精化：

$A ::= \cdots \mid V[\vec{e}]$ (索引类型变量)$\mid ?\{\phi\}.A \mid !\{\phi\}.A$ (断言与假设)$\mid \exists n.A \mid \forall n.A$ (量词)$P ::= \cdots \mid$ 断言 $x\ \{\phi\}\,;\ P \mid$ 假设 $x\ \{\phi\}\,;\ P$(断言与假设)$\mid$ 发送 $x\ \{e\}\,;\ P \mid \{n\} \leftarrow$ 接收 $x\,;\ P_n$ (发送和接收见证)

用于验证程序行为的关键类型是 ?，以及它的对偶类型 !。当进程提供 $(x : ?\{\phi\}.A)$ 时，它会通过 assert $x\ \{\phi\}$ 沿 x 发送关于 ϕ 的证明，并且

x 的客户端通过 assume $x\ \{\phi\}$ 接收关于 ϕ 的证明。请注意，运行时不会发送实际的证明对象；我们只会传达存在此类证明这一事实。与之前一样，对偶类型 $(x{:}\,!\{\phi\}.A)$ 会反转通信方向，使 x 的提供方能够接收证明，而客户端能够发送证明。我们还允许通道通过量词类型 $\exists$ 和 $\forall$ 发送和接收见证。$(x:\exists n.\,A)$ 的提供方通过 send $x\ \{e\}$ 发送见证表达式 e；x 的客户端通过 $\{n\} \leftarrow$ recv x 接收 n 的 e 值，并在延续中代入 $[e/n]A$。类型 $\forall n.\,A$ 展现了对偶行为。我们使用这些见证来通信自然数，而这些自然数本身可用于后续的断言和假设。例如，我们可以写成 $\exists k.\,?\{n = 2 * k\}.$自然数$[k]$，表示 n 是偶数，且 k 是它的见证。

算术表达式 e和命题 ϕ 的语言是标准的，具体如下所述。我们用 i表示常数，用 n 表示变量。

$$e ::= i \mid e + e \mid e - e \mid e * e \mid n$$
$$\phi ::= e = e \mid e > e \mid \top \mid \bot \mid \neg\phi \mid \phi \vee \phi$$

虽然原则上，我们的语法允许任意多项式算术表达式，但大多数示例都限制在线性表达式范围内。我们发现，对于任何高次多项式，通过 Z3 进行推断都会变得不可行。

3.3 类型变量和签名

我们使用签名 Σ，其中包含类型定义、进程声明和进程定义，如下所示：

$$\begin{array}{lll}
\Sigma ::= \cdot \mid \Sigma, V[\overline{n} \mid \phi] = A & & \text{(type definition)} \\
\quad \mid \Sigma, \Delta \vdash f[\overline{n}] :: (z : A) & & \text{(process declaration)} \\
\quad \mid \Sigma, x \leftarrow f[\overline{n}]\ \overline{y} = P & & \text{(process definition)} \\
\Delta ::= \cdot \mid \Delta, (x : A) & & \text{(process context)}
\end{array}$$

类型定义 $V[n \mid \phi] = A$ 表示，V 由满足条件 ϕ 的某些精化 n 索引；当我们写作 $V[e]$时，将其解释为 $A[e/n]$。我们对类型采用等递归和结构化的方法，不使用任何显式通信来展开。进程声明$\Delta \vdash f[n] :: (z :\ A)$ 告诉我们，f 提供一个类型为 A的通道 z，并且 f 会在 Δ中消耗所有类型为 A的通道 x。进程 f 也由某些精化 n 索引，这些精化可以在 Δ 和 A的类型中自由使用。对应的定义 $x \leftarrow f[n]\ y = P$同样由 n索引，但它给出的是使用通道 x 和 y的进程表达式 P，而非类型信息。再次强调，n 中的变量可以在定义中自由出现，例如出现在断言和假设中。

4 子类型关系

要定义子类型算法并证明其正确性，我们必须首先从子类型的语义定义入手。该定义采用了 Gay 和 Hole[19]使用类型模拟给出的定义。

类型模拟 我们的类型模拟定义推广了 Das 和 Pfenning [15], 先前的工作。他们为实现类型相等而定义了类型双向模拟。我们依赖于封闭 类型这一概念：若类型不包含自由算术变量，则该类型是封闭的。为处理含有精化变量的类型，我们引入有效性的概念。回顾一下，签名 Σ 收集了所有 形式为 $V[n \mid \phi] = A$ 的类型定义。若隐式约束 ϕ 对签名中所有出现的 $V[e]$ 均成立，则称该签名有效。形式化地说，若$V[e]$ 出现在 Σ中的任何类型定义里，则 $\vDash \phi[e/n]$。此外，我们要求有效签名只包含收缩性 类型定义，不允许出现形式为 $V[n \mid \phi] = V'[e]$ 的定义。我们在附录 B 中给出了有效类型和签名的完整形式化定义。

定义 1. 对于有效签名 Σ，我们定义：若 $V[n \mid \phi] = A \in \Sigma$，则 展开$_\Sigma(V[e]) = A[e/n]$；否则（当 A 不是类型变量时），展开$_\Sigma(A) = A$。

有效签名对于明确定义 unfold$_\Sigma$ 至关重要，因为收缩条件禁止我们展开为另一个类型变量（这可能会无限进行）。

定义 2（类型模拟）。 闭有效类型上的关系 $\mathcal{R}$，若满足 Σ，则称为类型模拟；若对于任意类型 A、B，只要 $(A, B) \in \mathcal{R}$，取 $S =$展开$_\Sigma(A)$ 和 $T =$ 展开$_\Sigma(B)$ 后，下列条件成立：

i) 若 $S = \oplus\{\ell : A_\ell\}_{\ell \in L}$，则 $T = \oplus\{m : B_m\}_{m \in M}$。此外，对于所有 $\ell \in L$，$L \subseteq M$ 且 $(A_\ell, B_\ell) \in \mathcal{R}$。*ii)* 若 $S = \&\{\ell : A_\ell\}_{\ell \in L}$，则 $T = \&\{m : B_m\}_{m \in M}$。此外，对于所有 $m \in M$，$L \supseteq M$ 且 $(A_m, B_m) \in \mathcal{R}$。*iii)* 若 $S = A_1 \otimes A_2$，则 $T = B_1 \otimes B_2$。此外，$(A_1, B_1) \in R$ 且 $(A_2, B_2) \in R$。*iv)* 若 $S = A_1 \multimap A_2$，则 $T = B_1 \multimap B_2$。此外，$(B_1, A_1) \in R$ 且 $(A_2, B_2) \in R$。*v)* 若 $S = \mathbf{1}$，则 $T = \mathbf{1}$。*vi)* 若 $S = ?\{\phi\}.A$，则 $T = ?\{\psi\}.B$。此外，要么 $\vDash \phi$、$\vDash \psi$，且 $(A, B) \in \mathcal{R}$；要么 $\vDash \neg\phi$。*vii)* 若 $S = !\{\phi\}.A$，则 $T = !\{\psi\}.B'$。此外，要么 $\vDash \psi$、$\vDash \phi$，且 $(A, B) \in \mathcal{R}$；要么 $\vDash \neg\psi$。*viii)* 若 $S = \exists m.A$，则 $T = \exists n.B$。此外，$\forall i \in \mathbb{N}$ 我们有 $(A[i/m], B[i/n]) \in \mathcal{R}$。*ix)* 若 $S = \forall m.A$，则 $T = \forall n.B$。此外，$\forall i \in \mathbb{N}$ 我们有 $(A[i/m], B[i/n]) \in \mathcal{R}$。

每种类型构造子在定义 2 中都有一个对应的情形，直观而言，这归结为：对 A 而言，类型对中的第一个类型应模拟第二个类型 B的行为。具体来说，A 允许的通信行为必须是 B允许的通信行为的子集。例如，在 $\oplus$的情形 (*i*) 中，我们要求 A 的标签集是 B的标签集的子集，也就是说，A 的提供方能够发送的每个标签，都能被 B的客户端接收，反之则不然。同样，$S = A_1 \multimap A_2$ 和 $T = B_1 \multimap B_2$ 的情形 (iv) 特别将 (B_1, A_1) 加入 $\mathcal{R}$，因为 $\multimap$（与箭头类型一样）对第一个参数是逆变的。直观而言，如果一个进程提供 $(c: S)$，而我们希望将其类型标注为 $(c : T)$，那么所接收通道的指定类型 B_1 应表现出实际类型 行为的一个子集

。情形 (*i*) –(*v*) 借鉴自 Gay 和 Hole [19]的子类型定义。

值得注意的是新情形 (*vi*) –(*ix*)，因为它们涉及精化。情形 (*vi*) 表明，对于 $?\{\phi\}.\,A$，ϕ 必须要么为真，要么为假。在前一种情况下，ψ 也必须为真，并且 $(A, B) \in \mathcal{R}$。然而，在后一种情况下，该关系平凡成立，因为此类通道的提供方将停止通信，这实际上模拟了任意续延类型。情形 (*vii*) 与之类似，但翻转了约束的方向。最后，情形 (*viii*)（相应地 (*ix*)）表明，若 m 的所有替换都已处于类型模拟之中，则量化类型 $\exists m.\,A$（相应地 $\forall m.\,A$）模拟另一个量化类型。有了类型模拟之后，我们便将子类型关系简单地定义如下：

定义 3. 对于封闭有效类型 A 与 B，我们称 $A <: B$，即 A 是 B 的子类型，如果存在类型模拟 $\mathcal{R}$ 使得 $(A, B) \in \mathcal{R}$。

4.1 算法化子类型

在存在精化的情况下，子类型关系从根本上是不可判定的。这一事实由如下事实所蕴含：即使是更简单的类型相等问题，在存在精化时也是不可判定的，如先前工作 [15]所示。尽管如此，我们提出一种可靠的算法来近似子类型关系，该算法通过一系列推断规则表达，其主要判断的形式为$\mathcal{V}\,;\mathcal{C}\,;\,\Gamma \Vdash A <: B$。这里，$\mathcal{V}$ 和 $\mathcal{C}$ 分别表示自由算术变量的列表以及支配约束，我们同样调用辅助判断 $\mathcal{V}\,;\,\mathcal{C} \vDash \phi$ 来表示语义蕴涵。Γ 是已遇到的形如 $\langle \mathcal{V}\,;\mathcal{C}\,;\,V_1[e_1] <: V_2[e_2]\rangle$的闭包列表，我们捕获它们以用于循环检测。为简洁起见，我们省略了一些标准规则；完整的子类型规则系列见附录 C。

图 1 描述了与算术精化相关的子类型关系的部分规则，其中最有意思的是 $\mathsf{st}_?$ 和 $\mathsf{st}_!$。规则 $\mathsf{st}_?$ 的第一个前提指出，ϕ 必须蕴含 ψ，这体现了定义 2 的直觉：ϕ 和 ψ 要么同时成立，要么 ϕ 为假。请注意，我们只说在某个约束 $\mathcal{C}$下，但定义 2 中并没有这样的约束：这是因为类型模拟仅针对闭类型定义。（我们将在证明中看到如何弥合这一差距。）第二个前提要求续延类型 A 在约束 $\mathcal{C} \wedge \phi$下必须是 B 的子类型。规则 $\mathsf{st}_!$ 与之类似，只是将蕴含方向反转，这与定义 2 相似。规则 $\mathsf{st}_\exists$ 只是在变量上下文 $\mathcal{V}$中引入一个新鲜变量 k，同时还要求将 k 代入每个类型中指定的变量（m 或 n）。规则 $\mathsf{st}_\bot$ 处理约束 $\mathcal{C}$ 矛盾的情况。在这样的约束下，任意类型 A 和 B 都是子类型，因为这种情况在运行时永远不会出现。当类型检查因精化而不可能执行的分支时（例如，在类型 nat $[n+1]$上检查**零** 分支），这条规则会很有用。

循环检测的核心在于 $\mathsf{st}_{\mathsf{expd}}$ 和 $\mathsf{st}_{\mathsf{def}}$ 规则；当我们遇到类型名称 $V_1[e_1]$ 和 $V_2[e_2]$时，每条规则都可能适用。$\mathsf{st}_{\mathsf{expd}}$ 会根据我们的签名 Σ有效地展开这些类型名称，但它也会生成一个闭包

$$\frac{\mathcal{V}\,;\mathcal{C}\vDash\phi\to\psi\qquad\mathcal{V}\,;\mathcal{C}\wedge\phi\,;\Gamma\Vdash A<:B}{\mathcal{V}\,;\mathcal{C}\,;\Gamma\Vdash ?\{\phi\}.A<:?\{\psi\}.B}\ \mathsf{st}_?$$

$$\frac{\mathcal{V}\,;\mathcal{C}\vDash\psi\to\phi\qquad\mathcal{V}\,;\mathcal{C}\wedge\psi\,;\Gamma\Vdash A<:B}{\mathcal{V}\,;\mathcal{C}\,;\Gamma\Vdash !\{\phi\}.A<:!\{\psi\}.B}\ \mathsf{st}_!$$

$$\frac{(k\ \mathsf{fresh})\qquad\mathcal{V},k\,;\mathcal{C}\,;\Gamma\vdash A[k/m]<:B[k/n]}{\mathcal{V}\,;\mathcal{C}\,;\Gamma\vdash\exists m.A<:\exists n.B}\ \mathsf{st}_\exists\qquad\qquad\frac{\mathcal{V}\,;\mathcal{C}\vDash\bot}{\mathcal{V}\,;\mathcal{C}\,;\Gamma\Vdash A<:B}\ \mathsf{st}_\bot$$

$$\frac{V_1[\overline{v_1}|\phi_1]=A\in\Sigma\qquad V_2[\overline{v_2}|\phi_2]=B\in\Sigma\qquad\gamma=\langle\mathcal{V}\,;\mathcal{C}\,;V_1[\overline{e_1}]<:V_2[\overline{e_2}]\rangle}{\mathcal{V}\,;\mathcal{C}\,;\Gamma,\gamma\Vdash A[\overline{e_1}/\overline{v_1}]<:B[\overline{e_2}/\overline{v_2}]}\ \mathsf{st}_{\mathsf{expd}}$$

$$\frac{\langle\mathcal{V}'\,;\mathcal{C}'\,;V_1[\overline{e_1}']<:V_2[\overline{e_2}']\rangle\in\Gamma\qquad\mathcal{V}\,;\mathcal{C}\vDash\exists\mathcal{V}'.\mathcal{C}'\wedge\overline{e_1}'=\overline{e_1}\wedge\overline{e_2}'=\overline{e_2}}{\mathcal{V}\,;\mathcal{C}\,;\Gamma\Vdash V_1[\overline{e_1}]<:V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{def}}$$

图 1：子类型算法的部分规则。

γ 并将其添加到我们的上下文 Γ 中，表示我们已经在特定的 $\mathcal{V}$ 和 $\mathcal{C}$ 下“见过”由这些特定表达式 e_1 和 e_2 精化的 V_1 和 V_2。现在，如果之后在算法中再次遇到相同的 V_1 和 V_2，我们就会立即应用 $\mathsf{st}_{\mathsf{def}}$ 规则。我们不要求新的 $\mathcal{V}'$ 和 $\mathcal{C}'$ 与 γ 中的 $\mathcal{V}$ 和 $\mathcal{C}$ 完全相同——第二个前提（非正式地）断言，我们能够找到它们之间的一个替换；如果这样的替换确实存在，我们便得出算法成功的结论。这种余归纳结构使我们的算法能够处理那些原本可能被无限展开的类型。

$\mathsf{st}^{\mathsf{expd}}$ 要求两种类型同时展开，因此按所写形式，我们无法将名称与结构进行比较。为在实践中规避该问题，我们采用一种称为内部重命名 [10]的预处理步骤。对于我们签名中的每一个类型定义 $V[n]=A$，我们通过将内部类型表达式替换为显式新鲜名来重命名类型表达式 A。例如，我们将列表$_{\mathsf{A}}=\oplus\{\mathbf{nil}:\mathbf{1},\ \mathbf{cons}:\ A\otimes$列表$_{\mathsf{A}}\}$（某种类型（名称） A）的定义改写为如下定义集合：

$$\mathsf{type}\ \mathsf{list}_{\mathsf{A}}=\oplus\{\mathbf{nil}:\mathbf{1},\ \mathbf{cons}:\%0\}\qquad\qquad\mathsf{type}\ \%0=A\otimes\mathsf{list}_{\mathsf{A}}$$

其中 %0 是一个新鲜名。我们在类型表达式内部递归地应用这种重命名，使得类型定义始终具有形式 $A\mathsf{op}\ B$，其中 op 是类型运算符。这样，在任何类型表达式中，类型名称与结构类型总是交替出现。这保证了在应用子类型算法规则时，我们也在名称与名称的比较以及结构类型与结构类型的比较之间交替进行。

这些 $\mathsf{st}^{\mathsf{expd}}$ 和 $\mathsf{st}_{\mathsf{def}}$ 规则还值得注意，因为它们使我们的算法出现了不完备性。遇到两个类型名称时，可以无限地应用 $\mathsf{st}^{\mathsf{expd}}$，从而导致非终止算法。为避免这个问题，我们只展开有限次，然后强制应用 $\mathsf{st}_{\mathsf{def}}$ 规则。但代价是完备性，

因为此时 $\mathsf{st}_{\mathsf{def}}$ 可能无法在 Γ 中找到匹配的闭包。在没有精化的情况下，这是子类型关系中唯一的不完备性来源。

4.2 可靠性

现在，我们证明我们的算法相对于定义 3 是可靠的。证明 $A <: B$ 的子类型关系，实际上归结为构造一个类型模拟 $\mathcal{R}$，使得 $(A,\ B) \in \mathcal{R}$。这里的主要直觉是，可以根据使用我们的算法规则构建的子类型推导来构造这样的类型模拟。我们将在此阐释关键思想；完整的证明概要见附录 D。

定义 4。我们定义 $\forall \mathcal{V}.\mathcal{C} \to A <: B$（读作：对于所有 $\mathcal{V}$，$\mathcal{C}$ 蕴含 A 是 B 的子类型），当且仅当存在一个类型模拟 $\mathcal{R}$，使得对于所有在 $\mathcal{V}$ 上的基底替换 σ，满足 $\vDash \mathcal{C}[\sigma]$ 时，都有 $(A[\sigma],\ B[\sigma]) \in \mathcal{R}$。

该定义将子类型关系的概念从闭类型（例如 $\mathsf{nat}[2]$）扩展到了符号类型（例如 $\mathsf{nat}[n]$）。这是必要的，因为我们的子类型算法作用于符号类型。该定义背后的直觉是，如果所有满足 $\mathcal{C}$的替换 σ 都使得 $A[\sigma]$ 是 $B[\sigma]$ 的子类型，那么 A 就是 B 的子类型。作为示例，我们可以说 $\forall n.\, n > 0 \to \mathsf{nat}[n] <: \mathsf{nat}[n]$，因为 $\mathsf{nat}[1] <: \mathsf{nat}[1]$、$\mathsf{nat}[2] <: \mathsf{nat}[2]$, ，依此类推。

定义 5。给定一个 $\mathcal{D}$ 关于 $\mathcal{V}$；$\mathcal{C}$；$\Gamma \Vdash A <: B$的推导，我们定义闭包集合 $S(\mathcal{D})$，使得对于每个序列式 $\mathcal{V}'$；$\mathcal{C}'$；$\Gamma' \Vdash A' <: B'$，我们都加入$\langle \mathcal{V}'$；$\mathcal{C}'$；$A' <: B' \rangle \in S(\mathcal{D})$。

从推导得到的这组闭包，正是构造类型模拟所需的内容。这里的关键思想是，对于良构推导，所有这些闭包也都必须表示有效的子类型关系。最后，对推导中找到的所有闭包应用基替换，即可构造类型模拟。以下定理刻画了这一思想。

定理 1（可靠性）。若 $\mathcal{V}\,;\mathcal{C}\,;\cdot \Vdash A <: B$，则 $\forall \mathcal{V}.C \to A <: B$。

证明。取上述推导 $\mathcal{D}$，并构造类型模拟

$$\mathcal{R} = \{(A'[\sigma], B'[\sigma]) \mid \langle \mathcal{V}'\,;\mathcal{C}'\,;A' <: B' \rangle \in S(\mathcal{D}) \text{ and } \sigma : \mathcal{V}' \text{ and } \vDash \mathcal{C}'[\sigma]\}$$

我们在附录 D 中证明，$\mathcal{R}$ 确实是一个类型模拟。

5 推断算法

本节介绍我们的约束生成规则，它们构成了类型推断算法的理论基础。这些规则本身与本语言的标准类型检查规则密切对应，后者收录于附录 F。我们的主要判断采用相同的形式 $\mathcal{V}\,;\mathcal{C}\,;\Delta \vdash P :: (z :C)$，表示程序 P 提供类型为 C 的通道 z，并消耗所有通道 $(x : A_x) \in \Delta$。与子类型关系一样，我们也在一个列表的上下文中进行操作，其中包含

自由算术变量 $\mathcal{V}$ a nd，以及支配约束 $\mathcal{C}$。为便于比较，我们在下方列出 $\oplus R$的类型检查规则，并从下往上阅读：

$$\frac{(k \in L) \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : A_k)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash x.k\,;\,P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \text{TC-}\oplus\text{R}$$

由于我们进行的是推断而非类型检查，因此我们修改此规则，以体现我们并未预先获知 x 的类型结构 先验。我们将其视为一个变量，并对其施加一个 类型约束，该约束恰好采用子类型判断 $\mathcal{V}\,;\ \mathcal{C} \Vdash A <: B$ 的形式，针对类型 A 和 B 断言 A 是 B 的子类型。我们用蓝色突出显示引入的中间类型名称和新的算术变量，并假定所有此类名称都是新鲜的。

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash \oplus\{k : B\} <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : B)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash x.k\,;\,P :: (x : A)}\ \oplus\text{R}$$

由此得到的 $\oplus$R 规则同样自底向上阅读，是一条简单的结构规则，可生成单个子类型约束。非正式地说，当我们遇到程序 $\Delta \vdash x.\mathsf{succ}\,;\,P :: (x :\ A)$ 时，我们知道 A 必须是一个 $\oplus$-类型，并且必须包含标签 succ：这两条信息都通过类型约束前提来表达。随后，我们将以新引入的中间类型 B 为依据，继续执行算法。为简洁起见，其余结构规则与此类似，故略去。

上述规则（以及所有结构规则）只会产生类型约束。此外，有些规则会产生另一类约束：算术约束，其形式为语义蕴涵 $\mathcal{V}\,;\ \mathcal{C} \vDash \phi$ 对于一个命题 ϕ，断言在 $\mathcal{V}$和 $\mathcal{C}$的条件下，ϕ 成立。这些规则是为负责交换证明的精化类型构造子生成的。下文将重点介绍几条有代表性的规则。完整列表见附录 E。

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \phi \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash ?\{\phi\}.A' <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : A')}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \mathsf{assert}\ x\ \{\phi\}\,;\,P :: (x : A)}\ ?\text{R}$$

?R 规则值得注意，因为它同时引入算术约束和类型约束。算术约束与对应的类型检查规则相呼应，只是说明 ϕ 必须成立。我们的类型约束与子类型规则相结合，断言 A 必须具有 $?\{\psi\}.A'$ 的结构，其中某个 ψ，但 ψ 不必与 ϕ完全相同：我们需要的是$\mathcal{V}\,;\ \mathcal{C} \vDash \phi \to \psi$，也就是说，$\phi$ 比 ψ更强。

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: ?\{\phi\}.A' \qquad \mathcal{V}\,;\,\mathcal{C} \wedge \phi\,;\,\Delta, (x : A') \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{assume}\ x\ \{\phi\}\,;\,Q :: (z : C)}\ ?\text{L}$$

如果上下文 Δ 中出现了同一类型的通道，我们则改为应用 ?L 规则。在右规则中，我们看到新生成的类型出现在子类型约束的左侧。相比之下，由于类型对偶性，所有左规则中的新类型都位于右侧。当我们提供一个通道 $(z : C)$ 时，我们

希望声明类型比程序所需的类型更宽泛，也就是它的超类型——例如，即使进程只发送零标签，我们也可以声明为 (z : nat)。相反，如果我们 consume 一个通道 (x: A)，就需要 A 比程序所规定的类型更窄，也就是它的子类型；换言之，程序应该能够处理 A 的任何行为。在这种情况下，对于某个 ϕ， $A =?\{\psi\}.A'$，子类型关系要求 ψ 至少与 ϕ 一样强，因为如果 (x : A)“发送”了一个 ψ 的证明，我们便会收到一个 ϕ 的证明。

$$\frac{\mathcal{V}\,;\mathcal{C} \vDash e \geq 0 \qquad \mathcal{V}\,;\mathcal{C} \Vdash \exists n.A' <: A \qquad \mathcal{V}\,;\mathcal{C}\,;\Delta \vdash P :: (x : A')}{\mathcal{V}\,;\mathcal{C} \wedge n = e\,;\Delta \vdash \mathsf{send}\ x\ \{e\}\,;P :: (x : A)}\ \exists\mathrm{R}$$

∃R规则同样包含一个算术约束和一个类型约束。算术约束规定，我们发送的表达式必须是自然数，即 $e \geq 0$ 以维持所有见证均为自然数这一不变量。这个规则尤其值得注意的一点是，由于我们执行的是推断，无法像类型检查时那样进行替换 $A'[e/n]$，因为我们缺少关于 n 可能出现在新鲜名 A'何处的信息。作为一种简洁的解决方案，我们改为将替换存储为支配约束$\mathcal{C}$中的等价关系，使得后续所有延续都能访问 n 的值，而无需我们预先知道它会出现在哪里。

$$\frac{\begin{array}{c}(\overline{y_i' : B_i'})_{i\in I} \vdash f[\overline{n} \mid \phi] = P_f :: (x' : A') \in \Sigma \\ \mathcal{V}\,;\mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash A'[\overline{e}/\overline{n}] <: A \qquad (i \in I) \quad \mathcal{V}\,;\mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash B_i <: B_i'[\overline{e}/\overline{n}] \\ \mathcal{V}\,;\mathcal{C}\,;\Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)\end{array}}{\mathcal{V}\,;\mathcal{C}\,;\Delta, (\overline{y_i : B_i})_{i\in I} \vdash x \leftarrow f\,[\overline{e}]\ \overline{y}\,;Q :: (z : C)}\ \mathsf{def}$$

最后一个例子是生成进程的def规则，值得注意的是它与签名的交互方式：我们对 f 的声明包含 A' 和 B_i' 类型变量，它们既受 f 的主体 P_f 约束，也受此处生成f的操作约束。我们的子类型约束看似简单：对于生成 f时消耗的每个通道（y_i： B_i），我们将其与 f声明中对应的通道（y_i'： B_i'）进行比较，并断言“实际”类型 B_i 是“预期”类型 B_i'的子类型。反过来，我们也会约束 f 的提供类型 A'，使得 A' 是新鲜变量 A的子类型。

在实践中，def规则尤为重要，因为它生成了我们的大多数算术约束，尽管它并未将任何算术约束作为显式前提生成。为求解类型约束，我们应用第4节中介绍的子类型算法，由此生成算术约束。下一节将进一步阐述类型约束与算术约束之间的关系，以及整个约束求解过程。

最后，我们陈述推断算法的可靠性：

定理 2。 设 $z \leftarrow f[\mathcal{V}]\ x_i = P$为进程 f的定义。对于新鲜类型名 A_i 和 C，使得 $(z :\ C)$ 以及 $\Delta = (x_i :\ A_i)_{\forall i}$，并且如果 $\mathcal{V}\,;\ \top\,;\ \Delta \vdash P :: (z :\ C)$，则 P 是一个良类型进程，即通过推断得到的 A_i 和 C 的类型赋值满足类型检查。

6 实现

我们在 Rast 实现 [13], 之上实现了类型推断算法，该实现已经支持会话类型的词法分析、语法分析和类型检查，并支持算术精化。推断引擎由 2,069 行 SML 代码组成，并与 Z3 SMT 求解器 [17] 集成，以对算术约束进行建模。我们的一般方法首先为文件中的每个进程引入占位声明；对于每个定义 `proc x ← f[V] x1 x2 ... xn`，我们引入如下声明

`decl f[V] : (x1 : A1[e1]) (x2 : A2[e2]) ... (xn : An[en]) ⊢ (x : A[e])`

其中，每个类型 A_1、A_2、.....、A_n、A 都是一个新鲜的类型变量，并由进程的自由变量上的算术表达式 $e_i(\mathcal{V})$ 加以细化。我们的目标是找到一个类型赋值，将每个类型变量映射到程序中声明的具体类型名称；同时找到一个表达式赋值，将每个 e_i 映射到一个符号算术表达式。定理 2 指出，这些赋值会得到一个类型正确的程序；我们通过使用推断出的类型运行 Rast 类型检查器来验证这一点。我们将讨论如何分两个阶段进行推断、所面临的实际挑战，以及为应对这些挑战而引入的优化。

两阶段推断 在推断进程的类型时，我们会构造两种不同的约束：类型约束和算术约束。针对进程的约束生成规则会生成这两种约束；而用于求解类型约束的子类型关系算法可能会产生额外的算术约束，这些约束也必须得到满足，推断才能成功。

起初，人们可能会想在统一遍中求解这两类约束。然而，在实际操作中这行不通。要理解原因，首先要注意，由于存在进程派生，算术约束无法急切地求解。考虑进程 `foo`，它两次调用 `two` 进程，并分别将提供的通道赋给 `y` 和 `z`。

```
decl foo: . ⊢ (x: A[e()])
proc x ← foo = y ← two; z ← two; ...
```

在 `foo` 进程中，对 `two` 的两次调用都会生成关于类型 A[$e()$], 的约束，而这些约束又会针对 e 生成多个不同的算术约束。由于 e 在两个约束中必须相同，因此，仅分别证明这两个约束可满足是不够的；否则，求解器可能会错误地为 e 分配不同的值。因此，我们必须先收集所有算术约束，然后将它们作为一个大批次一并求解。

然而，采用这种方法并按照书面规则执行，会导致我们生成的约束数量急剧膨胀，并且不得不进行代价高昂的求解器调用。这种膨胀的主要原因在于存在 thest$_\perp$ 规则，该规则规定，只要 constraint $\mathcal{C}$ 为假，任意类型，即使彼此不匹配，也可以是子类型。因此，每当我们在发生结构不匹配时进行子类型关系判断、可能想要返回假时，

例如 $\mathcal{V}; \mathcal{C} \vDash A <: B$，我们必须改为将约束 $\mathcal{V}; \mathcal{C} \vDash \bot$ 添加到算术约束集。在这种朴素方法中，我们无法排除任何类型赋值，除非进行求解器调用；而由于可能的赋值数量呈指数级增长，这种调用的代价变得高得令人难以承受。

因此，我们选择禁止程序员编写任何死代码；也就是说，不允许出现满足 $\mathcal{V}; \mathcal{C} \vDash \bot$ 的代码。这项限制会带来若干影响，但最关键的是，我们现在无需检查 $\mathcal{C}$ 是否可满足，就能排除结构上的类型赋值。利用这项限制，我们将类型推断拆分为两个阶段：第一阶段仅处理类型结构，此时程序中的所有精化都被移除，并生成一个结构上可行的类型赋值列表；第二阶段针对特定的类型赋值重新检查精化，收集由此产生的算术约束，并将其交给 Z3 SMT 求解器。这种两阶段方法还有一个额外好处，即在算法中清晰划分了可判定与不可判定的片段：具体来说，不含精化的类型推断是可判定的，因此遇到类型不匹配时，我们可以返回明确的错误消息。

6.1 推断的第一阶段

第一阶段收集类型检查和子类型检查过程中生成的约束，去除精化，然后计算类型赋值。由于这一阶段实际上是可判定的，因此我们的算法保证会终止，并找到一个满足条件的类型赋值，或报告类型错误。因此，如果程序因结构化类型（即不考虑精化）而类型错误，就保证能够检测出来。

传递性 如前所述，我们的类型约束生成规则会为遇到的每个进程表达式生成一个新的“中间”类型变量。例如，考虑将给定自然数的值加倍的 double 进程。我们的规则会生成以下约束，行末的注释标明了引出该约束的那一行：

```
decl double: (x: A0) ⊢  (y: A1)
proc y ← double x = case x (            % A0 <: +{zero: I0, succ: I1}
    zero  ⇒  y.zero;                    % +{zero: I2} <: A1
             y ↔ x                      % I0 <: I2
  | succ  ⇒  y.succ;                    % +{succ: I3} <: A1
             y.succ;                    % +{succ: I4} <: I3
             y ← double x )             % I1 <: A0; A1 <: I4
```

我们会生成新的中间类型变量（记作 I_n）作为续延类型的占位符。如果不区分这些中间变量与我们在声明中引入的“签名”变量（例如 A），就会产生大量约束，给推断引擎带来沉重负担。更糟糕的是，当这些类型带有索引精化时，每个这样的中间变量还会产生算术

约束，这会使 Z3 的约束求解变得不可行。这严重限制了推断的可扩展性。

我们通过一个关键观察来解决这一局限：如果 $A <: B$，那么 $\oplus\{k : A\} <: \oplus\{k : B\}$ 也同样如此。为了在上述示例中实际看到这一点，请注意第三个约束： $I_0 <: I_2$ 由此我们推导出 $\oplus\{\text{零} : I_0\} <: \oplus\{\text{零} : I_2\}$。由于我们的第二个约束断言 $\oplus\{\text{零} : I_2\} <: A_1$，我们可传递地得出结论： $\oplus\{\text{零} : I_0\} <: A_1$，从而从我们的约束中消除了 A_2。按照这一进程，我们可以将上述七个约束的集合简化为仅两个： $A_0 <: \oplus\{\text{零} : I_0, \text{后继} : A_0\}$ 以及 $\oplus\{\text{零} : I_0, \text{后继} : \oplus\{\text{后继} : A_1\}\} <: A_1$。这里我们的两个约束对应于 A_0 和 A_1的最一般类型。

这一观察带来了我们推断引擎中最重要的优化，我们称之为传递性。它部分依赖于子类型关系的传递性：如果 $A <: B$ 且 $B <: C$，那么 $A <: C$。在生成类型赋值之前，我们会执行传递性处理，尽可能消除中间变量。这大幅减少了类型约束和算术约束的数量，从而显著加快推断速度；我们将在第7节评估这一效果。不过，正如示例所示，传递性并不能消除所有中间变量——具体来说，在两个通道都已进行过通信的情况下，它们之间的转发会产生形如 $\mathcal{V}$；$\mathcal{C} \vDash I_0 <: I_1$ 的类型约束。在这种情况下，我们选择 I_1，通过传递性执行任何适当的替换，并让 I_0 保持“自由”状态，即不受约束——I_0 只会出现在另一项约束的续延类型中。

类型约束求解 中间变量消除后，我们为每个剩余的类型变量指定一个初始搜索空间，其中包含文件中定义的所有类型名。然后，对于每个类型变量，我们将其替换为其搜索空间中的一个类型，并对任何相关约束运行子类型关系的部分 版本。这个修改后的子类型算法在遇到两个（不相同的）类型名时会立即停止，并返回一个简化后的约束，或直接返回真或假。失败的候选项会从搜索空间中移除，成功的候选项则会保留，但之后也可能因精化而被剪除。

继续前面的例子，假设我们的签名中有以下类型定义。

```
type nat = +{zero: one, succ: nat}
type odd = +{succ: even}
type even = +{zero: one, succ: odd}
```

回顾一下我们经过传递性处理后的约束 $A_0 <: \oplus\{\textbf{零} : I_0, \textbf{后继} : A_0\}$ 和$\oplus\{\textbf{零} : I_0, \textbf{后继} : \oplus\{\textbf{后继} : A_1\}\} <: A_1$。类型约束求解会确定， A_0 可以是其中任意一个，但 A_1 只能是自然数或偶数，因为将奇数代入 A_1 会违反第二个约束中的 st$\oplus$规则。请注意，我们只有在假设不存在死代码时才能在此处排除奇数；否则，可能存在某些精化使行 y.零不可达，从而让我们可以应用 st$_\perp$ 而非 st$\oplus$来满足约束。

程序重构 不依赖精化的第一阶段还允许我们根据每个进程定义所赋予的类型重构 该定义，并补回断言和假设。我们采用强制演算 [16], 尽可能急切地作出假设、仅在绝对必要时才惰性地添加断言的方法，从而确保任何断言都能“看到”所有可能的假设。重构还意味着程序员不再需要手动编写断言或假设——这是一项劳动密集型工作，会大幅增加源代码量，抵消类型推断在实践中的许多潜在好处。我们还会重构不可能的分支，这与我们对程序员不得自行编写不可达代码的限制相一致。具体而言，如果标签集类型中的某个标签未出现在 case 语句中，我们会手动插入该标签，并指出它不可能出现。此方法与两阶段解法结合起来，构成了我们对前述不可能性膨胀问题的解决方案：在类型求解阶段，我们假设 $\mathcal{C}$ 永远不为假，这使我们能够依据结构排除可能的类型赋值。

作为示例的总结，我们任意选取一个有效类型赋值排列 $A_0 =$ 自然数; $A_1 =$偶数。然后，假设自然数采用标准精化，而奇数和偶数也遵循相同规则，我们重构程序精化，以得出第一阶段的结论：

```
decl double[n]: (x: nat[e0(n)]) ⊢  (y: even[e1(n)])
proc y ← double x = case x (
    zero  ⇒  assume x {e0(n) = 0};
        y.zero; assert y {e1(n) = 0};
        y ↔ x
  | succ  ⇒  assume x {e0(n) > 0};
        y.succ; assert y {e1(n) > 0};
        y.succ; assert y {e1(n)-1 > 0};
        y ← double x )
```

6.2 推断的第二阶段

在第二阶段，我们收集算术约束，并尝试寻找一个满足赋值。由于第一阶段已经为我们提供了结构上正确的类型赋值，因此我们通常会跳过与精化无关的结构性构造。不过，对于判断构造转发和生成进程，我们会格外关注，因为它们暗示其操作数之间存在子类型关系。对于这些构造，我们运行完整的子类型算法，并收集由此产生的所有约束。这一生成子阶段会产生一个算术约束列表，并将其发送给 Z3 [17], ；如果这些约束可满足，Z3 会返回一些值，我们可以将其解析为表达式赋值。如果 Z3 未能找到解，即因为不存在解、该问题不可判定，或发生超时，我们就认为不存在解；如果还有下一个类型赋值，则继续处理下一个。请注意，这一阶段实际上试图解决一个不可判定问题，因此我们的算法并不完备。然而，由于我们进行了大量优化并采用了启发式方法，它在实践中表现得非常出色。

多项式模板 我们的实现工作主要集中在让 Z3 配合我们的约束形式，并将其扩展到各种具有挑战性的基准测试。我们将形如 $\mathcal{V}$；$\mathcal{C} \vDash \phi$ 的算术约束转换为逻辑公式 $\forall \mathcal{V}.\mathcal{C} \Longrightarrow \phi$，其中 $\mathcal{C}$ 和 ϕ 包含我们希望求解的表达式变量 e_i（$\mathcal{V}$）。最初，我们的做法是将 e_i 视为未解释函数，并完全依赖 Z3 内置的未解释函数求解器。不幸的是，即使对于最简单的示例，这种方法也被证明是不够的：即便 Z3 没有超时，它也会返回一种分段的函数解释，或者以其他方式返回无法用我们的算术表达式语言表达的结果。

我们的下一项优化基于这样一个观察：我们只允许关于量化变量的精化表达式为多项式。利用这一点，我们将每个表达式表示为次数为 d 的多元多项式，并求解所有引入的系数。例如，若$d = 2$，且表达式 e 中的自由变量为 m, n，我们称 $e(m, n) = c_0 + c_1 m + c_2 n + c_3 m^2 + c_4 mn + c_5 n^2$。通过将此模板提供给求解器，相较于纯粹的未解释函数，我们大幅限制了 Z3 必须考虑的可能解释数量，从而显著减轻求解器的负担。原则上，我们可以通过这种方法支持非线性精化，但在求解非线性约束时，我们经常遇到 Z3 超时，因此实践中选择将次数限制为 $d = 1$。

实数算术 我们的精化只允许自然数，不支持算术表达式中的实数值。不过，与整数逻辑相比，在实数逻辑中求解时，Z3 更少出现停滞。当然，采用这种方法时，我们面临的风险是：在对系数建模时，Z3 可能返回非整数值，而这些系数最终必须是整数。但事实证明，这种情况远没有预期的那么常见。很多时候，即使相同的约束在整数逻辑下会导致超时，Z3 仍会直接返回精确的整数值。如果 Z3 返回浮点数值，我们就退回到整数逻辑。

7 求值

方法论 我们在一系列已知类型正确、且颇具挑战性的基准测试上，评估了推断算法的性能和有效性。实验在一台配备 16GB 内存和 8 核 M1 Pro 中央处理器的 2021年 MacBook Pro 上进行。对于每项基准测试，我们分别报告两个阶段的执行时间：类型约束求解（阶段 1）和算术约束求解（阶段 2）。我们通过以下方式评估三项关键优化的有效性：

– 多项式模板：我们使用两种策略运行推断引擎：**uif**表示未解释函数，而**多项式**表示使用多项式模板。– 实数算术：我们调用 Z3 时也有两种模式：**实数**表示实数算术，而**整数**表示整数。

– 传递性：我们启用或禁用传递性，它会消去中间类型变量，因此无需为这些变量寻找满足条件的类型赋值。

我们的结果汇总于表 1。对于所有实验，每次调用 Z3 的超时设为 10 秒；每次试验（可能包含多次此类调用）的总运行时间上限为 60 秒。所有数值结果均取 10 次试验的平均值，并四舍五入到最接近的百分之一毫秒。

我们的算法可能产生以下四种结果：

– **成功**：推断找到有效的类型和算术赋值，– **超时**：针对某个特定类型赋值的 Z3 调用未能在指定的 10 秒时限内返回，或算法的总执行时间超过 60 秒，– **不可表达**：当 Z3 在**uif** 模式下返回无法用我们的算术语言表示的表达式赋值时，例如表达式包含条件分支结构；或者在实数算术情形下，返回了无法转换为整数的浮点数。

严格来说，还有第四种可能：如果程序的类型不正确，我们的推断算法会返回**不可满足**。不过，在评估中，我们选择只关注类型正确的基准测试。此外，表 1 中省略了那些导致超时的实验。

7.1 结果

对于每个基准测试，我们都会简要说明其内容、展示实验结果，并讨论任何值得注意的发现。

一元自然数 *(nat-u)* 此基准测试主要包含基于类型的精化自然数。

$$type\ \mathsf{nat}[n] = \oplus\{\mathbf{zero} : ?\{n = 0\}.\mathbf{1},\ \mathbf{succ} : ?\{n > 0\}.\mathsf{nat}[n-1]\}$$
$$type\ \mathsf{natpair}[m][n] = \mathsf{nat}[m] \otimes \mathsf{nat}[n] \otimes \mathbf{1}$$

由于线性性限制，复制数字需要 natpair 类型。该模块包含四个主要进程，以及三个辅助和测试进程：**add** 用于将两个数相加，**clone** 用于复制一个数，**consume**用于将一个自然数转换为 **1**，以及**double** 通过 clone 和 add 将一个数加倍。

首先，我们注意到，只有同时使用传递性和多项式模板，推断才能成功。与整数算术相比，实数算术会导致阶段 2 的运行速度变慢，而在阶段 1 中则没有显著差异。正如预期的那样，阶段 2 所需时间比阶段 1 多出 2 到 3 个数量级，因为它需要调用 Z3。每种情况下推断出的类型都符合预期，例如

```
decl add[m][n] : (x : nat[m]) (y : nat[n]) ⊢  (z : nat[m+n])
decl double[n] : (x : nat[n]) ⊢  (y : nat[2*n])
```

基准测试	策略	算术	传递性	结果	阶段 1	阶段 2
nat-u	poly	real	true	成功	0.54	613.64
nat-u	poly	int	true	成功	0.56	171.04
nat-d	poly	real	true	成功	0.52	637.88
nat-d	poly	real	假	成功	17.07	665.02
nat-d	uif	real	true	不可表达	0.47	214.54
nat-d	uif	real	假	不可表达	16.88	249.26
nat-d	uif	int	true	不可表达	0.49	210.77
nat-d	uif	int	假	不可表达	16.87	246.57
bin	poly	real	true	成功	1.26	255.89
list	poly	real	true	成功	0.84	465.14
list	poly	int	true	成功	0.85	228.47
linlam	poly	real	true	成功	1.39	11.00
linlam	poly	int	true	成功	1.39	10.56
linlam	uif	real	true	成功	1.40	9.84
linlam	uif	int	true	成功	1.39	9.63
linlam 大小	poly	real	true	成功	1.75	148.33
linlam 大小	poly	int	true	成功	1.65	1091.58

表 1：求值结果。

直接自然数 （*nat-d*） 自然数的另一种表示方式是直接通过精化来表示，即通过类型 $\mathsf{nat} \;=\; \exists n.\mathbf{1}$。该类型不发送**“n”后继** 消息，而是将单个自然数作为精化生成，然后终止。此基准测试包含与一元自然数完全相同的所有程序，但采用了这种修改后的表示方式。

只有在特定的多项式模板与实数算术组合下，才会出现成功的满足赋值，无论是否使用传递性。传递性可使阶段 1 的速度提升 1–2 个数量级。使用未解释函数时，推断确实能找到解，但严重依赖上述条件分支结构来有效地对约束进行分类分析，而不是寻找线性解。例如，如果我们只将 $\mathsf{add}[m][n]$ 用作 $\mathsf{add}[2][3]$ 和$\mathsf{add}[4][5],$，那么 Z3 输出的不是预期表达式 $m+n$，而是 $\mathsf{if}\;\; m=2$ $\mathsf{then}\; 5\; \mathsf{else}\; 9$。

二进制数 *(bin)* 自然数的一种更高效表示方式是采用二进制形式。类型无需发送 n **后继** 消息，而是可以发送$\lceil$个 $\log_2 n\rceil$ 0 和 1。这个基准测试采用了这种表示方式，其中引入了一种类型 $\mathsf{bin}[n]$，它可以发送标签 $b0$、 $b1$ 或 e，分别表示 0、1 和终止。我们以 小端 格式对它们进行编码，即先发送最低有效位，这使实现更加方便，也让类型更直观。精化对它们的值进行索引。

$$\begin{aligned} \mathit{type}\ \mathsf{bin}[n] = \oplus\{&\mathbf{b0} : ?\{n>0\}.\exists k.\ ?\{n=2k\}.\mathsf{bin}[k],\\ &\mathbf{b1} : ?\{n>0\}.\exists k.\ ?\{n=2k+1\}.\mathsf{bin}[k],\\ &\mathbf{e} : ?\{n=0\}.\mathbf{1}\} \end{aligned}$$

$\mathsf{bin}[n]$ 的类型表示提供方可以发送标签 **b0**、**b1** 或 **e**。在 **b0** 的情况下，提供方会提供一个证明，证明 $n>0$，并生成一个新数 k，使得 $n=2k$，也就是说 n 是偶数。类似地，在 **b1** 的情况下，提供方断言 $n>0$，并生成 k 使得 $n=2k+1$，从而证明其为奇数。

最后，在 **e** 的情况下，提供方证明 $n = 0$ 并终止。对于这个基准测试，我们使用几个辅助进程来实现 **后继** 和**double**。

由于二进制类型的复杂性，这项基准测试对我们的算法来说最具挑战性。只有启用全部三项优化后，推断才能成功，这也证明了这些优化协同使用时的有效性。推断出的类型符合预期，并且通过了类型检查：

```
decl successor[n] : (x : bin[n]) ⊢  (y : bin[n+1])
decl double[n] : (x : bin[n]) ⊢  (y : bin[2*n])
```

列表 （列表） 这是另一种在实践中颇为重要的数据结构，尤其适用于在会话类型的上下文中存储数据。由于目前尚不支持多态会话类型的推断，我们使用由自然数组成的 Lisp 风格列表，并根据列表长度对其进行精化。类型 list$[n]$ 允许在 $n > 0$ 时发送**cons** 标签并转换为 list$[n-1]$，或在 $n = 0$ 时发送**nil** 标签。

$$\begin{aligned} \mathit{type}\ \mathsf{list}[n] = \oplus\{&\mathbf{cons} : ?\{n > 0\}.\,\mathsf{nat} \otimes \mathsf{list}[n-1],\\ &\mathbf{nil} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

为了专注于列表的推断，我们将未精化自然数（nat）用作列表中元素的类型。我们实现了两个重要的列表进程（以及一些辅助进程）：**追加**，用于连接两个列表；以及**拆分**，它通过两个相互递归的进程将列表一分为二，这两个进程分别处理偶数长度和奇数长度列表。我们还加入了 listpair$[m][n]$ 类型，与前述的 natpair 类型类似，用于实现**拆分**。

只有在使用传递性和多项式模板时，我们才能成功找到解。与一元自然数相比，阶段 1 耗时更长，而阶段 2 耗时更短。我们再次观察到，整数算术比实数算术更快，这支持了这样一种假设：整数算术往往能更快地找到解，尽管实数算术在更稳定地识别解方面更有用。正如预期的那样，推断出的类型如下：

```
decl append[m][n] : (x : list[m]) (y : list[n]) ⊢  (z : list[m+n])
decl split_even[n] : (x : list[2*n]) ⊢  (y : listpair[n][n])
decl split_odd[n] : (x : list[2*n+1]) ⊢  (y : listpair[n+1][n])
```

线性 λ演算 *(linlam)* 我们最复杂的基准测试是在会话类型之上实现一个用于表达式的线性 λ 演算。由于其复杂性，我们通过 2 个基准测试来介绍这个示例：第一个不包含任何精化，而第二个则按大小为类型添加索引。表达式使用以下类型表示：

$$\mathit{type}\ \mathsf{exp} = \oplus\{\mathbf{lam} : \mathsf{exp} \multimap \mathsf{exp},\ \mathbf{app} : \mathsf{exp} \otimes \mathsf{exp}\}$$

子类型关系在这里发挥着重要作用，因为写作如下形式的值的类型

$$\mathit{type}\ \mathsf{val} = \oplus\{\mathbf{lam} : \mathsf{exp} \multimap \mathsf{exp}\}$$

是 exp 的子类型，也就是表达式的类型：所有值也都是表达式。我们实现了用于求值和规范化表达式的进程，并在演算中加入了几个简单的程序（id、swap）。

从结果中可以立即看出，传递性在这里产生了显著差异：只有启用传递性的试验能够成功，其余试验则超时。这是因为同时存在 val 和 exp 类型。在先前的示例中，类型赋值可以立即排除搜索空间中的很大一部分，因为只有一种类型在结构上适合某个类型变量，但在这里 exp 和 val 有时可以互换。阶段 1 涉及的工作量要大得多；如果我们放弃传递性，中间类型变量的存在会进一步加剧这一问题。当然，由于缺少精化，Z3 没有需要求解的约束，因此仅与 Z3 相关的**策略** 和 **算术** 优化几乎没有可观察到的效果。由于阶段 2 关注算术精化，而本基准测试中并不存在算术精化，对 Z3 的调用在~10毫秒内完成。

由于子类型关系，我们的算法可以正确推断出多种可能的类型组合。我们的具体实现推断出以下关键声明，并优先选择更窄的类型 val，而非其超类型 exp：

```
decl apply : (e1 : exp) (e2 : exp) ⊢  (e : exp)
decl eval : (e : exp) ⊢  (v : val)
decl norm : (e : exp) ⊢  (n : exp)
```

带大小的线性 λ-演算*(linlam* 大小*)*最后，我们对最复杂的基准测试进行了扩展，加入了表示项大小的精化，如下所示：

$$\begin{aligned}\mathsf{type}\ \mathsf{exp}[n] = \oplus\{&\mathbf{lam} : ?\{n > 0\}.\forall n'.\mathsf{exp}[n'] \multimap \mathsf{exp}[n + n' - 1],\\ &\mathbf{app} : \forall n_1.\forall n_2.\ ?\{n = n_1 + n_2 + 1\}.\mathsf{exp}[n_1] \otimes \mathsf{exp}[n_2]\}\end{aligned}$$

我们将上述子类型 val 调整为 val$[n]$。最后，我们引入一种新类型 boundedVal$[n] = \exists k.\ ?\{k \leq n\}.\mathsf{val}[k]$，作为求值进程的新返回类型：尽管我们无法计算求值后项的确切大小，但可以为其设定一个上界。

与二进制数一样，这个示例体现了我们的优化所能发挥的全部威力。由于这个基准测试扩展了未经精化的线性 λ-演算，关闭传递性总会导致阶段 1 超时，这并不奇怪。由于这个基准测试还需要求解算术约束，我们的多项式模板策略对于避免 Z3 超时至关重要。借助另外两项优化，我们成功找到了整数算术和实数算术的解，但求解整数算术所需的时间要长一个数量级。

我们的实现返回以下关键声明：

```
decl apply[m][n] : (e1 : exp[m]) (e2 : exp[n]) ⊢  (e : exp[m+n+1])
decl eval : (e : exp[n]) ⊢  (v : boundedVal[n])
```

值得注意的是，通过推断出的 **eval** 类型，我们的算法推导出，大小为 n 的表达式求值后得到的值，其大小不超过 n。

8 相关工作

类型精化的推断技术人们提出了许多技术，用于推断函数式程序的精化类型。最早的精化类型推断工作是在 ML [18] 的背景下开展的，其中类型通过程序员指定的有限格进行精化。有限性对于类型推断的可判定性至关重要，类型精化则通过迭代进行，直到达到不动点。我们的类型系统表达能力强得多，支持可能无限的算术精化集合，因此推断不可判定。尽管如此，他们引入了分阶段推断技术：先推断基本类型，再推断类型精化。后来，Liquid Types [37] 也采用了这种方式，将推断简化为对基本类型进行 Hindley-Milner 类型推断 [9]，随后进行液态约束生成和约束求解。在这种设定下，类型推断是可判定的，因为他们采用了一种保守但可判定的子类型关系定义：任意依赖类型之间的子类型关系都被简化为对基本类型进行的一组蕴含检查。相比之下，我们的子类型关系定义是一般性的，因此不可判定。

类型精化的推断也已通过抽象解释 [31] 实现，即将其归约为计算简单的一阶命令式程序的不变量。Fusion 算法 [8]将推断归约为寻找否定范式（NNF）中（嵌套）霍恩子句约束的满足赋值。Hashimoto 和 Unno [24] 将推断归约为类型优化问题，并进一步将其归约为 Horn约束优化问题。该技术可根据用户指定的偏好推断出最优先的精化类型。Pavlinovic 等人 [35]提出了一种更通用的类型系统，该系统在抽象精化域的选择以及控制流信息的上下文敏感性两方面都是参数化的。最近，采用随机化测试 [47]和大语言模型 [38] 的基于学习的技术也被提出用于精化推断。

我们这种精化风格最显著的特点是，会话类型采用结构化类型，而前述系统采用名义类型。对于名义类型而言，基本类型的类型相等和子类型关系取决于名称，因此推断所有中间子表达式的基本类型是可判定的，并且可归约为标准的 Hindley-Milner 推断。另一方面，先前的研究 [14]表明，尽管线性算术以及基本会话类型的类型相等（因而也包括子类型关系）都是可判定的，但将它们结合起来会使类型判定变得不可判定。更具体地说，会话类型中不存在“基本类型”这一概念，因此我们的子类型规则无法拆分为基本类型的子类型规则和精化规则；二者自然地交织在一起。因此，我们的推断第一阶段也不同于这些工作：它消除中间类型，而不是推断它们。与我们的工作更为相关的是近期关于结构化类型精化的研究 [3] ，该研究

基于代数子类型，将名义类型系统和结构化类型系统的特性结合起来。这两项工作的主要区别在于精化层的设计。我们用算术表达式精化类型，而他们使用多态变体，因此支持类型应用和并集。因此，我们依赖求解器进行推断，而他们采用基于约束的类型推断。

会话类型推断 与我们的系统相关的还有一些会话类型推断技术。这些技术源于对 π-演算 [22, 36] 中通信原语的观察，并已在 OCaml [29] 和 Haskell [30] 中实现。会话类型也曾结合控制流信息 [7] 以及在服务与会话演算 [33]中进行推断。Almeida 等人 [2] 提出了一种用于推断 FreeST [1] 的算法；FreeST 是上下文无关会话类型 [41]的一种实现。该技术基于 Quick Look [39] ，以支持多态应用中的类型注释推断。尽管 FreeST 支持非正则协议，但算术精化的通用性更强，能够编码基于大小和值的更强性质，甚至包括非线性性质。因此，其推断算法也大不相同，而且 FreeST 无需求解算术约束。为此，这些工作都不支持像我们的系统那样的 DML 风格 [46]精化。

会话子类型关系会话类型的子类型关系概念源自 Gay 和 Hole 的开创性工作 [19], ，该工作在 π-演算中同时提出了类型模拟的形式化概念，以及针对基本会话类型的实用子类型算法。我们的工作在此基础上，为语言引入了精化。关键在于，他们的算法既可靠又完备，但在存在精化的情况下，不可能有完备的子类型算法；因此，我们的算法是可靠的，但并不完备。会话子类型关系在各种更具体的情境中得到了进一步探索：例如，Horne 和 Padovani [28] 在同构递归和等递归设置下分别研究并比较了子类型关系；另有研究者 [20, 21, 32] 提出了多方会话类型的子类型关系。Mostrous 和 Yoshida [34] 在高阶 π-演算中提出了异步会话类型的子类型关系，将类型模拟的概念扩展为异步类型模拟。Bravetti 等人 [4]证明，存在递归时，异步子类型关系是不可判定的。与这些工作不同，我们首次将会话子类型关系与精化系统结合起来进行研究。我们的精化概念取自 Das 和 Pfenning [16], ，但他们的工作仅限于类型相等，并未提供类型推断算法。

9 结论

本文提出了一种用于带算术精化的结构化会话类型的类型推断算法。我们对子类型关系进行了理论研究，介绍了用于约束生成的形式化推断规则，并在 Rast 语言中实现了该算法。此外，我们详细介绍了可靠求解约束所必需的实用优化，并通过评估该实现处理多个示例时的性能，展示了这些优化的优势。未来，我们希望将该算法扩展到 Rast 语言中的其他构造，即时态类型和资源感知类型 [13]。

数据可用性。 与本文相关的工件可通过 `https://doi.org/10.5281/zenodo.18155032` 公开获取。

参考文献

[1]Almeida, B., Mordido, A., Vasconcelos, V.：FreeST：函数式语言中的上下文无关会话类型。《理论计算机科学电子论文集》**291**，12–23（2019年3月），https://doi.org/10.4204/EPTCS.291.2[2]Almeida, B., Mordido, A., Vasconcelos, V.T.：上下文无关会话类型的局部类型推断。载于：Derakhshan, F., Hoffmann, J.（编）《第16届编程语言方法与并发及通信中心软件国际研讨会论文集》，加拿大汉密尔顿，2025年5月4日，《理论计算机科学电子论文集》第420卷，第1–11页，开放出版协会（2025），https://doi.org/10.4204/EPTCS.420.1[3] Binder, D., Skupin, I., Läwen, D., Ostermann, K.：结构化精化类型。载于：《第7届ACM SIGPLAN类型驱动开发国际研讨会论文集》，第15–27页，TyDe 2022，美国计算机协会，纽约，美国（2022），ISBN 9781450394390，https://doi.org/10.1145/3546196.3550163，统一资源定位符 https://doi.org/10.1145/3546196.3550163[4] Bravetti, M., Carbone, M., Zavattaro, G.：异步会话子类型关系的不可判定性。《信息与计算》**256**，300–320（2017），ISSN 0890-5401，https://doi.org/https://doi.org/10.1016/j.ic.2017.07.010，统一资源定位符 https://www.sciencedirect.com/science/article/pii/S0890540117301190[5] Caires, L., Pfenning, F.：作为直觉主义线性命题的会话类型。载于：Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Gastin, P., Laroussinie, F.（编）《CONCUR 2010——并发理论》，第6269卷，第222–236页，Springer柏林海德堡，柏林，海德堡（2010），ISBN 978-3-642-15374-7 978-3-642-15375-4，https://doi.org/10.1007/978-3-642-15375-4_16，统一资源定位符 http://link.springer.com/10.1007/978-3-642-15375-4_16，丛书名称：《计算机科学讲义》[6] Caires, L., Pfenning, F., Toninho, B.：作为会话类型的线性逻辑命题。《计算机科学中的数学结构》**760**（2014年11月）[7] Collingbourne, P., Kelly, P.H.J.：从控制流推断会话类型。《理论计算机科学电子札记》**238**(6)，15–40（2010年6月），ISSN 1571-0661，https://doi.org/10.1016/j.entcs.2010.06.003，统一资源定位符 https://doi.org/10.1016/j.entcs.2010.06.003[8] Cosman, B., Jhala, R.：局部精化类型检查。《ACM编程语言论文集》**1**(ICFP)（2017年8月），https://doi.org/10.1145/3110270，统一资源定位符 https://doi.org/10.1145/3110270[9]Damas, L., Milner, R.：函数式程序的主类型方案。载于：《第9届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集》，第207–212页，POPL '82，美国计算机协会，纽约，美国（1982），ISBN

0897910656，https://doi.org/10.1145/582153.582176，统一资源定位符 https://doi.org/10.1145/582153.582176[10] Das, A., Deyoung, H., Mordido, A., Pfenning, F.：嵌套会话类型。《ACM编程语言系统汇刊》**44(3)**（2022年7月），ISSN 0164-0925，https://doi.org/10.1145/3539656，统一资源定位符 https://doi.org/10.1145/3539656[11]Das, A., Hoffmann, J., Pfenning, F.：利用时序会话类型进行并行复杂度分析。《ACM编程语言论文集》**2(ICFP)**（2018年7月），https://doi.org/10.1145/3236786，统一资源定位符 https://doi.org/10.1145/3236786[12] Das, A., Hoffmann, J., Pfenning, F.：利用资源感知会话类型进行工作量分析。载于：《第33届ACM/IEEE计算机科学逻辑年会论文集》，第305–314页，LICS '18，美国计算机协会，纽约，美国（2018），ISBN 9781450355834，https://doi.org/10.1145/3209108.3209146，统一资源定位符 https://doi.org/10.1145/3209108.3209146[13] Das, A., Pfenning, F.：Rast：带算术精化的资源感知会话类型（系统描述）。载于：Ariola, Z.M.（编）《第5届计算与演绎形式结构国际会议（FSCD 2020）》，莱布尼茨国际信息学论文集（LIPIcs），第167卷，第33:1–33:17页，施洛斯·达格施图尔–莱布尼茨信息学中心，德国达格施图尔（2020），ISBN 978-3-95977-155-9，ISSN 1868-8969，https://doi.org/10.4230/LIPIcs.FSCD.2020.33，统一资源定位符 https://drops.dagstuhl.de/opus/volltexte/2020/12355[14]Das, A., Pfenning, F.：带算术精化的会话类型，第18页，共512384字节（2020），ISSN 1868-8969，https://doi.org/10.4230/LIPICS.CONCUR.2020.13，统一资源定位符https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2020.13[15] Das, A., Pfenning, F.：带算术精化的会话类型（2020），统一资源定位符 https://arxiv.org/abs/2005.05970[16]Das, A., Pfenning, F.：经验证的线性会话类型并发编程。载于：《第22届声明式编程原理与实践国际研讨会论文集》，第1–15页，ACM，意大利博洛尼亚（2020年9月），ISBN 978-1-4503-8821-4，https://doi.org/10.1145/3414080.3414087，统一资源定位符 https://dl.acm.org/doi/10.1145/3414080.3414087[17] De Moura, L., Bjørner, N.：Z3：一种高效的SMT求解器。载于：《软件理论与实践——第14届系统构造与分析工具和算法国际会议论文集》，第337–340页，TACAS'08/ETAPS'08，Springer-Verlag，柏林，海德堡（2008），ISBN 3540787992[18]Freeman, T., Pfenning, F.：ML的精化类型。《SIGPLAN通知》**26**(6)，268–277（1991年5月），ISSN 0362-1340，https://doi.org/10.1145/113446.113468，统一资源定位符 https://doi.org/10.1145/113446.113468[19]Gay, S., Hole, M.：π演算中的会话类型子类型关系。《信息学报》**42**(2-3)，191–225（2005年11月），ISSN 0001-5903，1432-0525，

https://doi.org/10.1007/s00236-005-0177-z，统一资源定位符 http://link.springer.com/10.1007/s00236-005-0177-z[20] Ghilezan, S., Jakšić, S., Pantović, J., Scalas, A., Yoshida, N.：同步多方会话的精确子类型关系。《逻辑与代数编程方法杂志》**104**，127–173（2019），ISSN 2352-2208，https://doi.org/https://doi.org/10.1016/j.jlamp.2018.12.002，统一资源定位符 https://www.sciencedirect.com/science/article/pii/S2352220817302237[21] Ghilezan, S., Pantović, J., Prokić, I., Scalas, A., Yoshida, N.：异步多方会话的精确子类型关系。ACM 计算逻辑汇刊**24(2)**（2023年11月），ISSN 1529-3785，https://doi.org/10.1145/3568422，统一资源定位符 https://doi.org/10.1145/3568422[22] Graversen, E.F., Harbo, J.B., Hüttel, H., Bjerregaard, M.O., Poulsen, N.S., Wahl, S.：pi演算中的会话类型推断。载于：Hildebrandt, T., Ravara, A., van der Werf, J.M., Weidlich, M.（编），《Web服务、形式化方法与行为类型》，第103–121页，Springer国际出版公司，Cham（2016），ISBN 978-3-319-33612-1[23] Griffith, D., Gunter, E.L.：Liquidpi：可推断的依赖会话类型。载于：Brat, G., Rungta, N., Venet, A.（编），NASA形式化方法，第185–197页，Springer柏林海德堡，柏林、海德堡（2013），ISBN 978-3-642-38088-4[24] Hashimoto, K., Unno, H.：通过Horn约束优化进行细化类型推断。载于：Blazy, S., Jensen, T.（编），静态分析，第199–216页，Springer柏林海德堡，柏林、海德堡（2015），ISBN 978-3-662-48288-9[25] Honda, K.：二元交互的类型。载于：Goos, G., Hartmanis, J., Best, E.（编），CONCUR'93，第715卷，第509–523页，Springer柏林海德堡，柏林、海德堡（1993），ISBN 978-3-540-57208-4 978-3-540-47968-0，https://doi.org/10.1007/3-540-57208-2_35，统一资源定位符 http://link.springer.com/10.1007/3-540-57208-2_35，丛书名称：《计算机科学讲义》[26] Honda, K., Vasconcelos, V.T., Kubo, M.：基于结构化通信的编程的语言原语与类型规则。载于：Hankin, C.（编），编程语言与系统，第122–138页，Springer柏林海德堡，柏林、海德堡（1998），ISBN 978-3-540-69722-0[27] Honda, K., Yoshida, N., Carbone, M.：多方异步会话类型。载于：第35届ACM SIGPLAN-SIGACT编程语言原理年会论文集，第273–284页，POPL '08，ACM，纽约，NY，美国（2008），ISBN 978-1-59593-689-9，https://doi.org/10.1145/1328438.1328472，统一资源定位符 http://doi.acm.org/10.1145/1328438.1328472[28]Horne, R., Padovani, L.：会话类型子类型关系的逻辑刻画。《逻辑与代数编程方法杂志》**141**，100986（2024年），ISSN 2352-2208，https://doi.org/https://doi.org/10.1016/j.jlamp.2024.100986，统一资源定位符 https://www.sciencedirect.com/science/article/pii/S2352220824000403

[29] Imai, K., Lange, J., Neykova, R.：Kmclib：从OCaml程序自动推断和验证会话类型。载于：Fisman, D., Rosu, G.（编），系统构造与分析工具和算法，第379–386页，Springer国际出版公司，Cham（2022），ISBN 978-3-030-99524-9[30] Imai, K., Yuen, S., Agusa, K.：Haskell中的会话类型推断。载于：Honda, K., Mycroft, A.（编），第三届编程语言方法与并发及以通信为中心的软件研讨会论文集，塞浦路斯帕福斯，2010年3月21日，《理论计算机科学电子论文集》，第69卷，第74–91页，开放出版协会（2011），https://doi.org/10.4204/EPTCS.69.6[31] Jhala, R., Majumdar, R., Rybalchenko, A.：Hmc：使用抽象解释器验证函数式程序。载于：Gopalakrishnan,G.,Qadeer, S.（编），计算机辅助验证，第470–485页，Springer柏林海德堡，柏林、海德堡（2011），ISBN 978-3-642-22110-1[32] Li, E., Stutz, F., Wies, T.：判定异步多方会话的子类型关系。载于：Weirich, S.（编），编程语言与系统，第176–205页，施普林格自然瑞士，Cham（2024年）[33] Mezzina, L.G.：如何在服务与会话演算中推断有限会话类型。载于：Lea, D., Zavattaro, G.（编），协调模型与语言，第216–231页，Springer柏林海德堡，柏林、海德堡（2008），ISBN 978-3-540-68265-3[34] Mostrous, D., Yoshida, N.：高阶π演算的会话类型系统与异步子类型关系。《信息与计算》**241**，227–263（2015），ISSN 0890-5401，https://doi.org/https://doi.org/10.1016/j.ic.2015.02.002，统一资源定位符 https://www.sciencedirect.com/science/article/pii/S0890540115000139[35] Pavlinovic, Z., Su, Y., Wies, T.：数据流细化类型推断。《ACM编程语言论文集》**5**(POPL)（2021年1月），https://doi.org/10.1145/3434300，统一资源定位符 https://doi.org/10.1145/3434300[36] Pucella, R., Tov, J.A.：几乎无需类型类的Haskell会话类型。载于：第一届ACM SIGPLAN Haskell研讨会论文集，第25–36页，Haskell '08，美国计算机协会，纽约，NY，美国（2008），ISBN 9781605580647，https://doi.org/10.1145/1411286.1411290，统一资源定位符 https://doi.org/10.1145/1411286.1411290[37]Rondon, P.M., Kawaguci, M., Jhala, R.：Liquid Types。载于：第29届ACM SIGPLAN编程语言设计与实现会议论文集，第159–169页，PLDI '08，美国计算机协会，纽约，NY，美国（2008），ISBN 9781595938602，https://doi.org/10.1145/1375581.1375602，统一资源定位符 https://doi.org/10.1145/1375581.1375602[38]Sakkas, G., Sahu, P., Ong, K., Jhala, R.：神经符号模块化细化类型推断。载于：2025年IEEE/ACM第47届国际软件工程会议（ICSE），第627–627页，IEEE计算机学会，洛斯阿拉米托斯，CA，美国（2025年5月），ISSN 1558-

1225, https://doi.org/10.1109/ICSE55347.2025.00090, URL https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00090[39]Serrano, A., Hage, J., Peyton Jones, S., Vytiniotis, D.：快速了解非谓词性。ACM编程语言论文集 4(ICFP)（2020年8月），https://doi.org/10.1145/3408971, URL https://doi.org/10.1145/3408971[40]Silva, G., Mordido, A., Vasconcelos, V.T.：上下文无关会话类型的子类型关系。载于：Pérez, G.A., Raskin, J.F.（编），第34届并发理论国际会议（CONCUR 2023），莱布尼茨信息学中心信息学论文集（LIPIcs），第279卷，第11:1–11:19页，施洛斯·达格施图尔 – 莱布尼茨信息学中心，德国达格施图尔（2023），ISBN 978-3-95977-299-0，ISSN 1868-8969，https://doi.org/10.4230/LIPIcs.CONCUR.2023.11, URLhttps://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2023.11[41] Thiemann, P., Vasconcelos, V.T.：上下文无关会话类型。载于：第21届ACM SIGPLAN国际函数式编程会议论文集，第462–475页，ICFP 2016，计算机协会，纽约，纽约州，美国（2016），ISBN 9781450342193, https://doi.org/10.1145/2951913.2951926, URL https://doi.org/10.1145/2951913.2951926[42]Thiemann, P., Vasconcelos, V.T.：标签依赖会话类型。ACM编程语言论文集 4(POPL)（2019年12月），https://doi.org/10.1145/3371135, URL https://doi.org/10.1145/3371135[43] Toninho, B., Caires, L., Pfenning, F.：通过直觉主义线性类型论实现依赖会话类型。载于：第13届国际ACM SIGPLAN声明式编程原理与实践研讨会论文集，第161–172页，PPDP '11，计算机协会，纽约，纽约州，美国（2011），ISBN 9781450307765，https://doi.org/10.1145/2003476.2003499, URL https://doi.org/10.1145/2003476.2003499[44] Toninho, B., Caires, L., Pfenning, F.：依赖会话类型十年。载于：第23届声明式编程原理与实践国际研讨会论文集，PPDP '21，计算机协会，纽约，纽约州，美国（2021年），ISBN 9781450386890，https://doi.org/10.1145/3479394.3479398, URL https://doi.org/10.1145/3479394.3479398[45] Wu, H., Xi, H.：依赖会话类型。CoRR **abs/1704.07004**（2017），URL http://arxiv.org/abs/1704.07004 [46] Xi, H., Pfenning, F.：实用编程中的依赖类型。载于：Aiken, A.（编），第26届编程语言原理研讨会会议记录（POPL 1999），第214–227页，ACM出版社，圣安东尼奥，得克萨斯州，美国（1999年1月）[47]Zhu, H., Nori, A.V., Jagannathan, S.：学习精化类型。载于：第20届ACM SIGPLAN国际函数式编程会议论文集，第400–411页，ICFP 2015，计算机协会，纽约，纽约州，美国（2015），ISBN 9781450336697,

https://doi.org/10.1145/2784731.2784766, 统一资源定位符 https://doi.org/10.1145/2784731.2784766

A 会话类型示例

关闭并等待。 进程 example1 具有如下声明:

```
decl example1 : (x : 1) (y : 1) ⊢ (z : 1)
```

根据其类型，我们可以推断它必须等待 x 和 y 关闭，然后关闭它所提供的通道 z。一个合适的定义如下:

```
proc z ← example1 x y = wait x; wait y; close z
```

或者，我们可以先等待 y ，再等待 x。

发送和接收标签。 我们将进程 example1 声明如下:

```
decl negate : (x: ⊕{真 : 1, 假 : 1}) ⊢ (y: ⊕{真 : 1, 假 : 1})
```

该进程会从 x 接收一个标签，可以是真或假，然后根据对 x 发送内容的分情况分析，在 y 上发送真或假。我们将negate 定义为:

```
proc y ← negate x = case x (
  true ⇒ y.false; wait y; close x
  | false ⇒ y.true; wait y; close x
)
```

这些类型并未告知我们 negate 必须发送哪个标签——在给定的定义中，我们发送相反的标签，但也可以始终只发送 **假**；这两个程序可以共用同一个声明。然而，类型系统会强制我们不能在 x 上发送或在 y 上接收标签集 {**真**, **假**} 之外的标签。

发送和接收通道。在我们的语言中，函数复合的等价形式可以写成如下进程 comp:

```
decl comp : (x: A ⊸ B), (y: B ⊸ C) ⊢ (z: A ⊸ C)
proc z ← comp x y = a ← recv z; send x a; send y x;
z ↔ y
```

我们首先从提供给我们的通道 z接收一个类型为 A的通道 a 。然后，我们将 a 发送给 x，这会将 x 的类型更改为 B。现在，我们将 x 发送给 y，以获得 $(y : C)$。剩下的是 $(y : C) \vdash (z : C)$，此时我们在 z 和 y之间转发。

类型名称和递归类型。 假设我们的签名包含类型声明 $\text{nat} = \oplus\{\textbf{零} : \mathbf{1}, \textbf{后继} : \text{nat}\}$ 考虑以下进程:

```
decl consume : (x: 自然数) ⊢ (y: 1)
proc y ← consume x = case x (
zero ⇒ y ↔ x
| succ ⇒ y ← consume x)
```

由于线性性，我们绝不能忽略上下文中的类型；如果想将它们丢弃，就必须将它们转换成一种我们知道如何关闭的类型。进程 consume 正是这样做的：它接收一个自然数，并将其转换为**1**，客户端随后便可以轻松等待它。我们通过首先对输入进行分情况分析来实现这个进程：如果 x发送**零**，我们只需转发；但如果 x发送**后继**，它就会递归回到自然数，因此我们再次生成进程*consume* ，并将 x 作为输入。通过反复重新生成 consume，我们会“consume” x 发送的所有标签，直到它变成**1**。

断言与假设。 现在令我们的自然数类型指向一个新鲜的 Σ 中的声明，其中 $\mathsf{nat}[n] = \oplus\{\textbf{零} : ?\{n = 0\}.\mathbf{1}, \textbf{后继} : ?\{n > 0\}.\mathsf{nat}[n-1]\}$。假设我们有一个 doubleprocess，它将输入值加倍，声明如下：

```
decl double[n] : (x: 自然数[n]) ⊢ (y: 自然
数[2*n])proc y ← double[n] x = case x (
zero ⇒ assume x {n=0};
y.zero; assert y {2*n=0};y ↔ x
| succ ⇒ assume x {n>0};
y.succ; assert y {2*n>0};
y.succ; assert y {(2*n)-1>0};
y ← double[n-1] x)
```

注意进程中的精化参数 n ——生成进程 double时，我们现在用一个表达式代替 n；在此例中，该表达式对应于输入通道的值 x。当我们从 x 接收到**零** 标签时，也会收到一个证明，证明 $n = 0$；我们利用这一事实断言，在 y 上发送**零** 后，$2 * n = 0$ 。而当我们改为在 x 上接收**后继** 时，我们得到 $n > 0$；我们的目标是在 y 上发送 2 条**后继** 消息。发送第一条时，我们利用假设断言 $2 * n > 0$，而根据类型定义， y的类型递归为自然数$[(2 * n) - 1]$。发送第二个**后继** 标签时，我们现在必须断言 $(2 * n) - 1 > 0$；这仍可由我们的断言推出（以及我们只在整数范围内工作的事实）。现在我们有 $(x : \text{自然数}[n-1]) \vdash (y : \text{自然数}[(2 * n) - 2]$，最后，我们通过以精化 $n - 1$ （而非 n）重新生成 double 来完成。

见证。 考虑一种用于二进制数的类型，它发送位 b0 和 b1，并以 e 终止。如果我们想用精化来表达其值，一个自然的想法是：

$$\begin{aligned} \mathit{type}\ \mathsf{bin}[n] = \oplus\{&\mathbf{b0} : ?\{n > 0\}.\mathsf{bin}[k/2], \\ &\mathbf{b1} : ?\{n > 0\}.\mathsf{bin}[(k-1)/2], \\ &\mathbf{e} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

然而，我们的算术语言缺少除法运算符。相反，我们会使用量词来表达同样的想法，如下所示：

$$\begin{aligned} \mathit{type}\ \mathsf{bin}[n] = \oplus\{&\mathbf{b0} : ?\{n > 0\}.\exists k.\ ?\{n = 2k\}.\mathsf{bin}[k], \\ &\mathbf{b1} : ?\{n > 0\}.\exists k.\ ?\{n = 2k+1\}.\mathsf{bin}[k], \\ &\mathbf{e} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

B 有效性

我们引入了三种有效性判断，分别用于签名（$\vdash \Sigma$ valid）、声明（$\vdash_\Sigma \Sigma'$ valid）和类型（$\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A$ valid）。

$$\frac{\vdash_\Sigma \Sigma \text{ valid}}{\vdash \Sigma \text{ valid}}\ \Sigma V \qquad \frac{}{\vdash_\Sigma (\cdot) \text{ valid}}\ \textsc{Nil}V$$

$$\frac{\vdash_\Sigma \Sigma' \text{ valid}}{\vdash_\Sigma \Sigma', \Delta \vdash f[\overline{n}] :: (z : A) \text{ valid}}\ \textsc{ProcDecl}V$$

$$\frac{\vdash_\Sigma \Sigma' \text{ valid}}{\vdash_\Sigma \Sigma', x \leftarrow f[\overline{n}]\ \overline{y} = P \text{ valid}}\ \textsc{ProcDef}V$$

$$\frac{\vdash_\Sigma \Sigma' \text{ valid} \qquad \overline{n}\,;\top \vdash_\Sigma A \text{ valid} \qquad A \neq V'[\overline{e}]}{\vdash_\Sigma \Sigma', V[\overline{n} \mid \phi] = A \text{ valid}}\ \textsc{TpDef}V$$

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma A_\ell \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \oplus\{\ell : A_\ell\}_{\ell \in L} \text{ valid}}\ \oplus V \qquad \frac{(\forall \ell \in L) \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma A_\ell \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \&\{\ell : A_\ell\}_{\ell \in L} \text{ valid}}\ \& V$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \text{ valid} \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma B \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \otimes B \text{ valid}}\ \otimes V$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \text{ valid} \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma B \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \multimap B \text{ valid}}\ \multimap V \qquad \frac{}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \mathbf{1} \text{ valid}}\ \mathbf{1}V$$

$$\frac{\mathcal{V}\,;\mathcal{C} \wedge \phi \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma ?\{\phi\}.A \text{ valid}}\ ?V \qquad \frac{\mathcal{V}\,;\mathcal{C} \wedge \phi \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma !\{\phi\}.A \text{ valid}}\ !V$$

$$\frac{\mathcal{V}, n\,;\mathcal{C} \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \exists n.A \text{ valid}}\ \exists V \qquad \frac{\mathcal{V}, n\,;\mathcal{C} \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \forall n.A \text{ valid}}\ \forall V$$

$$\frac{V[\overline{n} \mid \phi] \in \Sigma \qquad \mathcal{V}\,;\mathcal{C} \vDash \phi[\overline{e}/\overline{n}]}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma V[\overline{e}] \text{ valid}}\ \textsc{def}V$$

C 子类型规则

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_\ell <: B_\ell}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \oplus\{\ell : A_\ell\}_{\ell \in L} <: \oplus\{m : B_m\}_{m \in M}}\ \mathsf{st}_{\oplus}$$

$$\frac{(\forall m \in M) \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_m <: B_m}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \&\{\ell : A_\ell\}_{\ell \in L} <: \&\{m : B_m\}_{m \in M}}\ \mathsf{st}_{\&}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 <: B_1 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_2 <: B_2}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 \otimes A_2 <: B_1 \otimes B_2}\ \mathsf{st}_{\otimes}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash B_1 <: A_1 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_2 <: B_2}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 \multimap A_2 <: B_1 \multimap B_2}\ \mathsf{st}_{\multimap} \qquad \frac{}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \mathbf{1} <: \mathbf{1}}\ \mathsf{st}_{\mathbf{1}}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \to \psi \qquad \mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Gamma \vdash A <: B}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash ?\{\phi\}.A <: ?\{\psi\}.B}\ \mathsf{st}_{?}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \psi \to \phi \qquad \mathcal{V}\ ;\ \mathcal{C} \wedge \psi\ ;\ \Gamma \vdash A <: B}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash !\{\phi\}.A <: !\{\psi\}.B}\ \mathsf{st}_{!}$$

$$\frac{(k \text{ fresh}) \qquad \mathcal{V}, k\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A[k/m] <: B[k/n]}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \exists m.A <: \exists n.B}\ \mathsf{st}_{\exists}$$

$$\frac{\textbf{(}k \text{ 新鲜}\textbf{)} \qquad \mathcal{V},\ \ k\ ;\ \ \mathcal{C}\ ;\ \ \Gamma \vdash A[k/m] <:\ \ B[k/n]}{\mathcal{V}\ ;\ \ \mathcal{C}\ ;\ \ \Gamma \vdash \forall m.A <:\ \ \forall n.B}\ \mathsf{st}_{\forall} \qquad \frac{\mathcal{V}\ ;\ \ \mathcal{C} \vDash \bot}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A <: B}\ \mathsf{st}_{\bot}$$

$$\frac{\mathcal{V};\ \mathcal{C} \vDash e_1 = e_1' \wedge \cdots \wedge e_n = e_n'}{\mathcal{V}\ ;\ \ \mathcal{C}\ ;\ \ \Gamma \vdash V[\overline{e}] <:\ \ V[\overline{e'}]}\ \mathsf{st}_{\mathsf{refl}}$$

$$\frac{\begin{array}{c} V_1[\overline{v_1}|\phi_1] = A \in \Sigma \qquad V_2[\overline{v_2}|\phi_2] = B \in \Sigma \\ \gamma = \langle \mathcal{V}\ ;\ \mathcal{C}\ ;\ V_1[\overline{e_1}] <: V_2[\overline{e_2}] \rangle \\ \mathcal{V}\ ;\ \ \mathcal{C}\ ;\ \ \Gamma,\ \ \gamma \vdash A[\overline{e_1}/\overline{v_1}] <:\ \ B[\overline{e_2}/\overline{v_2}] \end{array}}{\mathcal{V}\ ;\ \ \mathcal{C}\ ;\ \ \Gamma \vdash V_1[\overline{e_1}] <: V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{expd}}$$

$$\frac{\langle \mathcal{V}'\ ;\ \mathcal{C}'\ ;\ V_1[\overline{e_1}'] <: V_2[\overline{e_2}'] \rangle \in \Gamma \qquad \mathcal{V};\ \mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}}{\mathcal{V};\ \mathcal{C};\ \Gamma \vdash V_1[\overline{e_1}] <: V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{def}}$$

D 可靠性

我们首先给出一系列辅助定义，以引出一个关键引理和我们的主要证明。

定义 6. 对于替换 σ，我们用 $\mathcal{V} ; \mathcal{C} \vDash \sigma$ 表示 σ 是 $\mathcal{V}$ 上的基替换，且满足 $\vDash \mathcal{C}[\sigma]$。

定义 7. 给定有效基类型上的关系 $\mathcal{R}$ 以及两个类型 A, B，其中 $\mathcal{V} ; \mathcal{C} \vdash A, B$ 有效，若对于所有满足$\mathcal{V} ; \mathcal{C} \vDash \sigma$的替换 σ，都有 $(A[\sigma], B[\sigma]) \in \mathcal{R}$，则称 $\forall \mathcal{V}.\mathcal{C} \rightarrow A <:_{\mathcal{R}} B$。

若存在满足 $\forall \mathcal{V}.\mathcal{C} \rightarrow A <:_{\mathcal{R}} B$ 的类型模拟 $\mathcal{R}$，则称 $\forall \mathcal{V}.\mathcal{C} \rightarrow A <: B$。

引理 1. 设 $\forall \mathcal{V}'.\mathcal{C}' \rightarrow V_1[e_1'] <:_{\mathcal{R}} V_2[e_2']$，并假设 $\mathcal{V} ; \mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1'} = \overline{e_1} \wedge \overline{e_2'} = \overline{e_2}$。则可得 $\forall \mathcal{V}.\mathcal{C} \rightarrow V_1[e_1] <:_{\mathcal{R}} V_2[\overline{e_2}]$。

证明。根据 $<:_{\mathcal{R}}$ 的定义，只需证明：对于所有满足 $\mathcal{V} ; \mathcal{C} \vDash \sigma$ 的替换 σ，都有 $V_1[e_1[\sigma]] <:_{\mathcal{R}} V_2[e_2[\sigma]]$。

任取这样一个 σ：由于 $\vDash \mathcal{C}[\sigma]$，我们可以将 σ 应用于第二个假设，从而得到 $\exists \mathcal{V}'.\mathcal{C}' \wedge e_1' = e_1[\sigma] \wedge e_2' = \overline{e_2}[\sigma]$。根据定义，存在一个关于 $\mathcal{V}'$ 的某个基替换 σ'，使得 $\vDash \mathcal{C}'[\sigma']$，$e_1'[\sigma'] = e_1[\sigma]$，并且$e_2'[\sigma'] = e_2[\sigma]$。

根据我们的第一个假设，对于任意这样的 σ'，若满足 $\vDash C'[\sigma']$，则可得 $V_1[\overline{e_1'}[\sigma']] <:_{\mathcal{R}} V_2[\overline{e_2'}[\sigma']]$。由于 $e_1'[\sigma'] = e_1[\sigma]$和 $e_2'[\sigma'] = \overline{e_2}[\sigma]$，我们得到 $V_1[e_1[\sigma]] <:_{\mathcal{R}} V_2[e_2[\sigma]]$。

由于这对任意这样的 σ 都成立，因此如上所述，我们已经完成：根据定义，$\forall \mathcal{V}.\mathcal{C} \rightarrow V_1[e_1] <:_{\mathcal{R}} \overline{V_2}[e_2]$。

回顾定理 1 的陈述：若 $\mathcal{V}_0 ; \mathcal{C}_0 ; \cdot \Vdash A_0 <: B_0$，则 $\forall \mathcal{V}_0.\mathcal{C}_0 \rightarrow A_0 <: B_0$。

定理 ***1*** 的证明。由前提可得一个关于 $\mathcal{V}_0, \mathcal{C}_0, \cdot \Vdash A_0 <: B_0$ 的推导 $\mathcal{D}_0$。在封闭有效类型上定义 $\mathcal{R}$ 如下：

$$\mathcal{R} = \{(A[\sigma], B[\sigma]) \mid \langle \mathcal{V} ; \mathcal{C} ; A <: B \rangle \in S(\mathcal{D}_0) \text{ and } \mathcal{V} ; \mathcal{C} \vDash \sigma\}$$

我们将证明 $\mathcal{R}$ 是一个类型模拟。为此，我们考虑任意的$(A[\sigma], B[\sigma]) \in \mathcal{R}$；根据 $\mathcal{R}$的定义，必定存在某个闭包 $\langle \mathcal{V} ; \mathcal{C} ; A <: B \rangle \in S(\mathcal{D}_0)$，以及某个具有 $\mathcal{V}$ 的 σ ； $\mathcal{C} \vDash \sigma$。

首先考虑 $\mathcal{V} ; \mathcal{C} \vDash \bot$的情形。由$\mathsf{st}_{\bot}$ 规则可得$\langle \mathcal{V} ; \mathcal{C} ; A <: B \rangle \in S(\mathcal{D}_0)$。此外，$\forall \mathcal{V}.\mathcal{C} \rightarrow A <: B$ 空洞为真，因此可靠性成立。

若改为 $\mathcal{V} ; \mathcal{C} \nvDash \bot$，则存在某个作用于 $\mathcal{V}$ 的基替换 σ满足 $\mathcal{C}$，即 $\mathcal{V} ; \mathcal{C} \vDash \sigma$。我们对 A 的结构进行分情况分析，并取任意这样的 σ。大多数情形可通过简单的结构分析得出；此处我们收录一个子集以作示范。

若 $A = \oplus\{\ell : A_\ell\}_{\ell \in L}$，则根据规则枚举，必有 $B = \oplus\{m : B_m\}_{m \in M}$。由 $\mathsf{st}_{\oplus}$ 可知，对于所有 $\ell \in L$，我们得到 $\langle \mathcal{V} ; \mathcal{C} ; A_\ell <: B_\ell \rangle \in$

$S(\mathcal{D}_0)$。根据 $\mathcal{R}$ 的定义，我们有 $(A_\ell[\sigma], B_\ell[\sigma]) \in \mathcal{R}$。由于 $A[\sigma] = \oplus\{\ell : A_\ell[\sigma]\}_{\ell \in L}$ 且 $B[\sigma] = \oplus\{m : B_m[\sigma]\}_{m \in M}$，我们得出，对于任意 σ，若 $(A[\sigma], B[\sigma]) \in \mathcal{R}$，则对于所有 $\ell \in L$，都有 $(A_\ell[\sigma], B_\ell[\sigma]) \in \mathcal{R}$。因此，我们满足了类型模拟的条件。

如果 $A = ?\{\phi\}.A'$，那么根据规则枚举，我们必有 $B = ?\{\psi\}.B'$。由 $\mathsf{st}_?$ 可知，我们得到 $\langle \mathcal{V} ; \mathcal{C} \wedge \phi ; A' <: B' \rangle \in S(\mathcal{D}_0)$，以及语义判断 $\mathcal{V} ; \mathcal{C} \vDash \phi \to \psi$。由于我们所考虑的 σ 满足 $\vDash \mathcal{C}[\sigma]$，因此有两个子情形：要么是 $\vDash \phi[\sigma]$，要么是 $\nvDash \phi[\sigma]$。如果是 $\vDash \phi[\sigma]$，则必有 $\vDash (\mathcal{C} \wedge \phi)[\sigma]$，因此根据 $\mathcal{R}$ 的定义，我们得到 $(A'[\sigma], B'[\sigma]) \in \mathcal{R}$。此外，由于 $\vDash \phi[\sigma]$ 且 $\mathcal{V} ; \mathcal{C} \vDash \phi \to \psi$，我们也有 $\vDash \psi[\sigma]$。由于 $A[\sigma] = ?\{\phi[\sigma]\}.A'$ 且 $B[\sigma] = ?\{\psi[\sigma]\}.B'$，我们满足类型模拟的条件 (1)。反之，如果是 $\nvDash \phi[\sigma]$，那么我们显然满足类型模拟的条件 (2)；因此，在这两种子情形下，$\mathcal{R}$ 都是类型模拟。

如果 $A = \exists m.A'$，那么根据规则枚举，$B = \exists n.B'$。由 $\mathsf{st}_\exists$ 可知，我们得到 $\langle \mathcal{V},\ k ;\ \mathcal{C} ;\ A'[k/m] <:\ B'[k/m] \rangle \in S(\mathcal{D}_0)$，其中某个 k 是新鲜的。由于 k 是新鲜的，$k \notin C$，因此对于任意 $i \in \mathbb{N}$，我们都有 $(A'[\sigma, i/k],\ B'[\sigma, i/k]) \in \mathcal{R}$，从而满足 $\mathcal{R}$ 是类型模拟这一条件。

如果 $A = \neg V_1[e_1]$，那么有三种子情形：要么适用 $\mathsf{st}_{\mathsf{refl}}$ 规则，要么适用 $\mathsf{st}_{\mathsf{def}}$ 规则，要么适用 $\mathsf{st}^{\mathsf{expd}}$ 规则；我们按此顺序考虑它们。如果应用 $\mathsf{st}_{\mathsf{def}}$ 规则，根据 $\mathcal{R}$ 的定义，我们不会向 $S(\mathcal{D}_0)$ 添加任何内容，但事实上这样做并无必要。$\mathsf{st}_{\mathsf{def}}$ 规则的第一个前提告诉我们，我们已经见过 $\neg\langle \mathcal{V}' ;\ \mathcal{C}' ;\ V_1[e_1'] <:\ V_2[e_2'] \rangle$，这意味着 $\forall \mathcal{V}'.\neg\mathcal{C}' \to V_1[e_1'] <:_{\mathcal{R}} V_2[e_2']$。根据引理 1 和 $\mathsf{st}_{\mathsf{def}}$ 规则的第二个前提可知，$V_1[\overline{e_1}[\sigma]] <:_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$，因此 $(V_1[e_1[\sigma]],\ V_2[e_2[\sigma]]) \in \mathcal{R}$。

现在我们结束证明。由于我们的推导 $\mathcal{D}_0$ 必须证明 $\mathcal{V}_0, \mathcal{C}_0, \cdot \Vdash A <: B$，根据 S 的定义，我们有 $\langle \mathcal{V}_0 ;\ \mathcal{C}_0 ;\ A_0 <:\ B_0 \rangle \in S(\mathcal{D}_0)$。然后，根据 $\mathcal{R}$ 的定义，对于满足 $\vDash \mathcal{C}_0[\sigma]$ 的任意 $\mathcal{V}_0$ 上的 σ，我们有 $(A_0[\sigma], B_0[\sigma]) \in \mathcal{R}$。接着，根据定义 7，我们说 $\forall \mathcal{V}_0.\mathcal{C}_0 \to A_0 <:_{\mathcal{R}} B_0$；最后，由于 $\mathcal{R}$ 是类型模拟，我们说 $\forall \mathcal{V}_0.\mathcal{C}_0 \to A_0 <: B_0$，证明至此完成。

E 约束生成规则

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash \oplus\{k : B\} <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : B)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash x.k\,;\,P :: (x : A)}\ \oplus\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: \oplus\{\ell : A_\ell\}_{\ell\in L} \qquad (\forall \ell \in L) \quad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A_\ell) \vdash Q_\ell :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell) :: (z : C)}\ \oplus\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash \&\{\ell : A_\ell\}_{\ell\in L} <: A \qquad (\forall \ell \in L) \quad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P_\ell ::, (x : A_\ell)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \mathsf{case}\ x\ (\ell \Rightarrow P_\ell) :: (x : A)}\ \&\,\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: \&\{k : A_k\} \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A_k) \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash x.k\,;\,P :: (z : C)}\ \&\,\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A_1 \otimes A_2 <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : A_2)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (y : A_1) \vdash \mathsf{send}\ x\ y\,;\,P :: (x : A)}\ \otimes\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: A_1 \otimes A_2 \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (y : A_1), (x : A_2) \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash y \leftarrow \mathsf{recv}\ x\,;\,Q :: (z : C)}\ \otimes\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A_1 \multimap A_2 <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (y : A_1) \vdash P :: (x : A_2)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash y \leftarrow \mathsf{recv}\ x\,;\,P :: (x : A)}\ \multimap\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: A_1 \multimap A_2 \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A_2) \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A), (y : A_1) \vdash \mathsf{send}\ x\ y\,;\,Q :: (z : C)}\ \multimap\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash \mathbf{1} <: A}{\mathcal{V}\,;\,\mathcal{C}\,;\,\cdot \vdash \mathsf{close}\ x :: (x : A)}\ \mathbf{1}\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: \mathbf{1} \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : \mathbf{1}) \vdash \mathsf{wait}\ x\,;\,Q :: (z : C)}\ \mathbf{1}\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: B}{\mathcal{V}\,;\,\mathcal{C}\,;\,(y : A) \vdash x \leftrightarrow y :: (x : B)}\ \mathsf{id}$$

$$\frac{\begin{array}{c}(\overline{y_i' : B_i'})_{i\in I} \vdash f[\overline{n} \mid \phi] = P_f :: (x' : A') \in \Sigma \\ \mathcal{V}\,;\,\mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash A'[\overline{e}/\overline{n}] <: A \qquad (i \in I) \quad \mathcal{V}\,;\,\mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash B_i <: B_i'[\overline{e}/\overline{n}] \\ \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)\end{array}}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (\overline{y_i : B_i})_{i\in I} \vdash x \leftarrow f[\overline{e}]\ \overline{y}\,;\,Q :: (z : C)}\ \mathsf{def}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \phi \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash ?\{\phi\}.A' <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : A')}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ P :: (x : A)}\ ?\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: ?\{\phi\}.A' \qquad \mathcal{V}\,;\,\mathcal{C} \wedge \phi\,;\,\Delta, (x : A') \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ ?\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash !\{\phi\}.A' <: A \qquad \mathcal{V}\,;\,\mathcal{C} \wedge \phi\,;\,\Delta \vdash P :: (x : A')}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ P :: (x : A)}\ !\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \phi \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash A <: !\{\phi\}.A' \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A') \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ !\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash e \geq 0 \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash \exists n.A' <: A \qquad \mathcal{V}\,;\,\mathcal{C} \wedge n = e\,;\,\Delta \vdash P :: (x : A')}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \mathsf{send}\ x\ \{e\}\ ;\ P :: (x : A)}\ \exists\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: \exists n.A' \qquad \mathcal{V}, n\,;\,\mathcal{C}\,;\,\Delta, (x : A') \vdash Q_n :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ Q_n :: (z : C)}\ \exists\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C}\,;\,\forall n.A' <: A \qquad \mathcal{V}, n\,;\,\mathcal{C}\,;\,\Delta \vdash P_n :: (x : A')}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ P_n :: (x : A)}\ \forall\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash e \geq 0 \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash A <: \forall n.A' \qquad \mathcal{V}\,;\,\mathcal{C} \wedge n = e\,;\,\Delta, (x : A') \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{send}\ x\ \{e\}\ ;\ Q :: (z : C)}\ \forall\mathrm{L}$$

F 类型检查规则

$$\frac{(k \in L) \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash P :: (x : A_k)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash x.k\ ;\ P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \textsc{tc-}\oplus\mathrm{R}$$

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A_\ell) \vdash P :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell) :: (z : C)}\ \textsc{tc-}\oplus\mathrm{L}$$

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash P :: (x : A_\ell)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell) :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \textsc{tc-}\,\&\,\mathrm{R}$$

$$\frac{(k \in L) \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A_k) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : \&\{\ell : A_\ell\}_{\ell \in L}) \vdash x.k\ ;\ Q :: (z : C)}\ \textsc{tc-}\,\&\,\mathrm{L}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash P :: (x : B)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (e : A) \vdash \mathsf{send}\ x\ e\ ;\ P :: (x : A \otimes B)}\ \textsc{tc-}\otimes\mathrm{R}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : B), (y : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A \otimes B) \vdash y \leftarrow \mathsf{recv}\ x\ ;\ Q :: (z : C)}\ \textsc{tc-}\otimes\mathrm{L}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (y : A) \vdash P :: (x : B)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash y \leftarrow \mathsf{recv}\ x\ ;\ P :: (x : A \multimap B)}\ \textsc{tc-}\multimap\mathrm{R}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : B) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A \multimap B), (e : A) \vdash \mathsf{send}\ x\ e\ ;\ Q :: (z : C)}\ \textsc{tc-}\multimap\mathrm{L}$$

$$\frac{}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \textsc{tc-}\mathbf{1}\mathrm{R}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : \mathbf{1}) \vdash \mathsf{wait}\ x\ ;\ Q :: (z : C)}\ \textsc{tc-}\mathbf{1}\mathrm{L}$$

$$\frac{}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ (y : A) \vdash x \leftrightarrow y :: (x : A)}\ \textsc{tc-id}$$

$$\frac{(\overline{y_i' : B_i})_{i \in I} \vdash f[\overline{n}] = P_f :: (x' : A) \in \Sigma \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, \overline{(y_i : B_i)}_{i \in I} \vdash x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q :: (z : C)}\ \textsc{tc-def}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ P :: (x : ?\{\phi\}.A)}\ \textsc{tc}\text{-}?\mathrm{R}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : ?\{\phi\}.A) \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ \textsc{tc}\text{-}?\mathrm{L}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Delta \vdash P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ P :: (x : !\{\phi\}.A)}\ \textsc{tc}\text{-}!\mathrm{R}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : !\{\phi\}.A) \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ \textsc{tc}\text{-}!\mathrm{L}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vdash e : \mathsf{nat} \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash P :: (x : A[e/n])}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \mathsf{send}\ x\ \{e\}\ ;\ P :: (x : \exists n.A)}\ \textsc{tc}\text{-}\exists\mathrm{R}$$

$$\frac{\mathcal{V}, n\ ;\ \mathcal{C}\ ;\ \Delta, (x : A) \vdash Q_n :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : \forall n.A) \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ Q_n :: (z : C)}\ \textsc{tc}\text{-}\exists\mathrm{L}$$

$$\frac{\mathcal{V}, n\ ;\ \mathcal{C}\ ;\ \Delta \vdash P_n :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ P_n :: (x : \forall n.A)}\ \textsc{tc}\text{-}\forall\mathrm{R}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vdash e : \mathsf{nat} \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A[e/n]) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : \exists n.A) \vdash \mathsf{send}\ x\ \{e\}\ ;\ Q :: (z : C)}\ \textsc{tc}\text{-}\forall\mathrm{L}$$