JFP，?? 页，2023年。© 剑桥大学出版社2023年
doi:10.1017/xxxxx

arXiv:2303.04678v1 [cs.PL] 8 Mar 2023

爱丽丝还是鲍勃？编排中的进程多态

EVA GRAVERSEN
南丹麦大学数学与计算机科学系 电子邮箱：efgraversen@imada.sdu.dk

ANDREW K. HIRSCH
*Dep*计算机科学与工程系，纽约州立大学布法罗分校 电子邮箱：akhirsch@buffalo.e *du*

FABRIZIO MONTESI
南丹麦大学数学与计算机科学系，电子邮件：fmontesi@imada.sdu.dk

摘要

我们介绍 PolyChorλ，这是一种用于高阶函数式编排式编程的语言。高阶函数式编排式编程是一种新兴范式，程序员借此编写通信进程系统所需的协作行为，然后将其编译为各个进程的分布式实现；这一转换称为端点投影。与其前身 Chorλ不同，PolyChorλ同时具有受 System F_ω启发的类型多态和进程多态。也就是说，PolyChorλ是首个使程序员能够编写通用编排并在运行时确定参与者的（高阶）函数式编舞语言。这种新颖的特性组合还使 PolyChorλ进程能够通信分布式值，从而带来一种新颖直观的委托编写方式。尽管 PolyChorλ的一些函数式特性使编排语义与其端点投影所得并发系统之间的对应关系弱于某些其他编舞语言，但我们仍然获得了编舞式编程的标志性成果：投影后的程序从设计上就是无死锁的。

关键词：编排、并发、 λ-演算、类型系统、多态

1 引言

并发程序涉及相互交互的进程。通常，程序员会为每个进程编写一个程-序，然后将这些程序并行组合起来。这些程序包含发送 和 接收 表达式，用于在进程之间传输数据。因此，很容易写出会死锁的代码，或因发送和接收模式不匹配而陷入停滞。 会话类型 (Honda，1993；Honda 等人，1998) 可用于描述程序中的发送和接收模式，为旨在防止通信不匹配和死锁的静态分析提供基础（Scalas 和 Yoshida，2019；Caires 和 Pfenning，2010；Wadler，2012；DeYounget al.，2012；Honda 等人，2016；Dardha 等人，2012）。使用会话类型，程序员可以确保系统中的通信遵循兼容的发送/接收模式。

或者，程序员可以采用全局视角（例如参见 Dalla Preda 等人，2017；Giallorenzo 等人，2021；Carbone 和 Montesi，2013；Cruz-Filipe 等人，2021；Hirsch 和 Garg，2022；Richter 等人，2022；Jongmans 和 van den Bos，2022；L´opez 等人，2016）。采用全局视角的程序员编写一个程序，然后将其编译为每个进程对应的程序。 编舞式编程 (Montesi，2013) 是一种全局编程范式，其理论基础得到了尤为充分的探索（Cruz-Filipe 和 Montesi，2020；Montesi，2023）。在编排中，发送 和 接收 命令被放在一起编写，通常采用受安全协议启发的表示法（Needham 和 Schroeder，1978年）。这有两个关键优势。首先，它使程序员能够表达进程之间所需的通信流程，而无需手动编写发送和接收操作。其次，它确保不存在可能导致死锁的不匹配；这一性质后来被称为设计上无死锁 (Carbone 和 Montesi，2013)。

要了解其威力，可以看看这个（臭）名昭著的书商示例——这是编舞式编程和会话类型文献中反复出现的示例（Carbone 和 Montesi，2013年；Honda 等人，2016；Montesi，2023年）。买家想从卖方那里买一本书。为此，买家将书名——例如《不可儿戏》——发送给卖方，卖方随后将价格发回来。买家随后可以将价格与自己的预算进行比较，并根据比较结果告知卖方如果价格在预算范围内，自己就想买这本书；否则则告知卖方自己不想买。我们可以用以下编排来描述这一过程：

$$\begin{aligned}&\text{let x} = \text{通信}_{\text{Buyer},\text{Seller}}(\text{“《不可儿戏》”} @ \text{买家}) \text{ in let y} = \text{通信}\\&_{\text{Seller},\text{Buyer}}(\text{价格查询 x}) \text{ in if y} \lessdot \text{预算 then 选择}_{\text{Buyer},\text{Seller}} \text{Buy}\\&(() @ \text{卖方}) \text{ else 选择}_{\text{Buyer},\text{Seller}} \text{Quit } (() @ \text{卖方})\end{aligned}$$

在这个编排中，与所有编舞程序一样，计算发生在通过消息传递进行通信的多个进程之间。值位于各个进程中；例如，在编排的第一行，书的书名最初位于买家处。函数 $\text{com}_{P,Q}$ 将一个值从进程 P 通信给进程 Q。它接收 P 处的一个局部值，并返回 Q 处的一个局部值。[1] 因此，x 表示卖方进程处的字符串“《不可儿戏》”，而 y 表示买家进程处的价格。最后，我们在本地检查书的价格是否在买家的预算之内。无论结果如何，我们都使用函数 select 从买家向卖方发送一个标签，以表示买家选择继续购买还是不购买。无论结果如何，该编排都会在卖方处返回哑值 ()。

虽然早期大多数关于编排的工作都集中于上述示例中的简单低阶命令式编程，但近期研究已经展示了如何开发高阶编舞式编程语言。这些语言允许程序员使用高阶编程的常见抽象来编写无死锁代码，

[1] 当我们在第 3 节对语言进行形式化时，我们要求为 com_P 添加类型注解 Q，但在引言中，为了使示例更清晰，我们没有添加类型注解。

例如对象（Giallorenzo 等人，2020年）和高阶函数（Hirsch 和 Garg，2022年；Cruz-Filipe 等人，2021年）。

例如，上面的程序将书名和书的价格写死在代码中。然而，我们可能希望在买家想买任何一本书时都能使用这段代码，并让买家 使用任意本地函数来决定是否按某个价格购买这本书。

```
λ书名 : 字符串 @ 买家.λ按价格购买? : 整数
@ 买家 → 布尔值 @ 买家. let x = 通信Buyer,
Seller 书名 in let y = 通信Seller,Buyer (价格查询
x) in if 按价格购买? y then 选择Buyer,Seller
Buy (() @ 卖方) else 选择Buyer,Seller Quit (()
@ 卖方)
```

使用高阶编舞语言的程序员，和使用任何高阶编程语言的程序员一样，可以编写一次程序，并在大量场景中使用。例如，提供不同的书名 和 按价格购买?值后，同一段代码就能用于购买多种不同的书名，而且买家 可以用自己想要的任何方法来判断是否愿意按该价格购买这本书。

尽管从一阶编程迈向高阶编程意义重大，但此前关于高阶编舞式编程理论基础的研究仍未涉及其他形式的抽象（Hirsch 和 Garg，2022年；Cruz-Filipe 等人，2021年）。具体而言，这些研究不支持多态，即程序可以同时抽象数据类型和数据，从而适用于更多场景；也不支持委托，即一个进程可以请求另一个进程代替自己执行操作。

这些抽象形式都相当常见：委托是并发演算中的一项重要操作，而多态对现代编程至关重要。然而，在编舞式编程中，另一种抽象形式也顺理成章：对进程进行抽象。当前的高阶编舞语言要求代码提及具体的进程名称。但我们往往希望编写更通用的代码，使同一段代码能在许多进程上运行。例如，上述程序允许买家 决定是否从卖方 处购买书籍，并使用任意本地函数按价格购买?。更自然的做法是将卖方 视为售书服务 ，不同的客户端都可以通过相同的方式与之交互并购买书籍。

本文探讨了三种抽象形式。首先，我们证明，对进程进行抽象是一种多态，我们称之为进程多态。其次，我们为 Chorλ——一种简单类型函数式编舞语言——加入多态（包括进程多态），并将这一新语言称为 PolyChorλ。第三，我们加入了通信分布式值（如函数）的能力。令人惊讶的是，这一扩展还使我们能够编写包含委托的代码，从而提供了一种简洁的语言，用于研究这三种抽象形式。

让我们看看扩展语言中的书商服务：

ΛB :: Proc.λ书名 : 字符串 @ B.λ按价格购买? : 整数 @ B $\rightarrow_{\varnothing}$ 布尔值 @ B. let x = 通信$_{B,\text{Seller}}$ 书名 in let y = 通信$_{\text{Seller},B}$ (价格查询 x) in if 按价格购买? y then 选择$_{B,\text{Seller}}$ Buy (() @ 卖方) else 选择$_{B,\text{Seller}}$ Quit (() @ 卖方)

此程序允许一个名为 B 的进程与 卖方 联系，以购买一本书。 B 随后提供一个字符串 书名 和一个决策函数 按价格购买?。这样，我们就不必再为每个可能想从 卖方购买书籍的进程单独编写函数。

虽然这项扩展看似简单，却带来了一些独特的理论挑战。首先，编舞语言的目标是将全局程序编译为每个进程各自的局部程序。然而，由于 B 并不代表任何特定进程，因此尚不清楚如何编译上述多态代码。我们通过一个简单的原则解决这个问题：每个进程都知道自己的身份。基于这一原则，我们可以将代码编译为每个进程中的一个条件表达式：如果进程扮演 B 的角色，就运行一个分支；否则就运行另一个分支。

值得注意的是，每个进程都会动态地选择要运行代码的哪种解释。这种灵活性很重要，因为我们可能希望允许不同进程动态地占据B的位置。例如，我们可以设想这样一种情形： 买家$_1$ 和 买家$_2$ 携手购买一本特别昂贵的书：他们或许会比较各自的银行账户余额，由钱更多的一方买下这本书，供双方共享。在我们的系统中，可以用以下代码实现这一点，其中卖家服务是上文函数的名称：

λ书名：字符串 @ 买家$_1$.令 x =通信$_{\text{Buyer}_1}$,Buyer2 银行账户余额$_1$，若 x < 银行账户余额$_2$则选择$_{\text{Buyer}_2}$,Buyer1 我选择$_{\text{Buyer}_2}$,Seller 我 seller service Buyer$_2$ (通信$_{\text{Buyer}_1}$,Buyer2 书名) (λz. z < 银行账户余额$_2$) 否则选择$_{\text{Buyer}_2}$,Buyer1 你选择$_{\text{Buyer}_2}$, Seller 他们 (seller service Buyer$_1$ 书名 (λz. z < x))

在我们扩展语言的操作语义中也会遇到一个相关的挑战。像 PolyChorλ这样的语言通常具有操作语义，通过允许乱序执行，使其与编译后代码的语义相匹配：不同进程中的可归约表达式可以按任意顺序归约。不过，处理进程多态时必须谨慎，因为可能无法确定两个可归约表达式属于同一个进程还是不同的进程。

除类型多态和进程多态外，PolyChorλ还是首个允许通信分布式值的编排语言：这些值并非完全位于发送方。这些值包括由分布式函数描述的完整编排，可用于建模委托。为了说明进程多态和分布式值通信如何实现委托，请考虑以下场景：当买家询问

一本书时，卖方首先检查该书是否有库存。如果有，销售将照常继续。否则，卖方将委托给第二个卖方，后者可能会将该书卖给买家。我们可以如下定义这段代码：

```
ΛB :: Proc。
 λ书名 : 字符串 @ B。
  λ按价格购买? : 整数 @ B →∅ 布尔值 @ B。
   设 x =通信B,卖方 书名
   在 if 找到(价格查询x )
     然后选择 卖方,B 继续
          选择Seller,卖方 2 断开连接
           令 y = 通信卖方,B (价格(价格查询 x) )
           若 按价格购买?y
             则 选择B,Seller 购买 (() @B)
             否则 选择B, Seller 退出 (() @B)
     否则 选择Seller,B 委托
          选择Seller,卖方 2 委托
                                                    ⎛                                          ⎞
           令 F =通信卖方,卖方2                      ⎜ λ书名2 : 字符串 @ 卖方.                  ⎟
                                                    ⎜   如果 找到(价格查询2 书名2)            ⎟
                                                    ⎜   那么选择 卖方,B 继续                  ⎟
                                                    ⎜         令 y′ =价格(价格查询2 书名2)     ⎟
                                                    ⎜         在令 y = 通信卖方,B y′           ⎟
                                                    ⎜           在如果 按价格购买? y           ⎟
                                                    ⎜             则选择B,卖方 购买 (() @B)    ⎟
                                                    ⎜             else 选择B,卖方 退出 (() @B) ⎟
                                                    ⎜   else 选择Seller, B 退出 (() @B)        ⎟
                                                    ⎝                                          ⎠
           in let 书名2 =通信Seller,卖方 2 x
              在F 书名2
```

论文结构。我们首先在第 2 节考察 PolyChor 的系统模型λ。随后，我们将介绍以下贡献：

- 在第 3 节，我们详细介绍 PolyChorλ语言。该语言同时包含类型多态和进程多态。我们为 PolyChorλ 开发了类型系统、种类系统和操作语义。
- 在第 4 节，我们介绍用于描述分布式实现的本地网络语言。我们还详细说明如何通过端点投影获得这一实现；该投影将 PolyChorλ程序编译为每个进程各自的程序。
- 在第 5 节中，我们将介绍本文的主定理，即相对于我们的操作语义而言，端点投影的正确性。与以往关于编排的研究相比，由于进程多态具有动态特性，这需要进行大量推理。

最后，我们将在第 6 节讨论相关工作，并在第 7节总结。

2 系统模型

我们首先讨论 PolyChorλ程序运行方式所依赖的假设。这些假设尽可能宽松，使 PolyChorλ能够在许多不同的场景中运行。具体而言，我们假设存在一组固定的进程，它们可以通过消息进行通信。每个进程都可以用一种多态λ演算来描述，该演算类似于 System Fω，但增加了通信原语。

2.1 进程

我们假设存在一个固定集合 **N** ，其中包含进程名称 P、Q、Alice等。这些进程可以表示分布式系统中的节点、系统进程、线程等。进程多态使我们能够使用类型变量来引用进程，而这些变量可能会进入或离开作用域。尽管如此，实际运行的进程集合仍保持不变。

我们假设每个进程都知道自己的身份。因此，每个进程都可以根据自身身份选择要运行的代码。在许多实际场景中，这一假设是合理的；例如，分布式系统中的节点通常都知道自己的身份。这种能力对于我们实现进程多态的策略至关重要。

2.2 通信

我们假设进程通过同步消息传递进行通信。因此，如果P 向Q发送消息，那么P 在Q 接收到消息之前不会继续执行。此外，我们假设消息传递是即时且可靠的，因此消息不会丢失。

进程可以接收两种消息：局部程序的值（如下文所述），以及描述计算过程中所作选择的标签。这些消息用于确保不同进程彼此步调一致。

2.3 局部程序

我们假设进程运行一种局部语言，该语言将在第4节中介绍。这是一种扩展了通信功能的函数式语言，与 GV（Gay 和 Vasconcelos，2010；Wadler，2012年）类似。然而，与 GV 及其衍生语言不同，我们的局部语言包含类型多态和进程多态。

端点投影将 PolyChorλ转换为这种“网络进程”语言。因此，我们进一步扩展了 GV，加入端点投影机制所需的特性。例如，我们提供了一个 AmI 表达式形式，使进程能够根据自身身份选择要运行的代码。尽管进行了这些扩展，对于熟悉多态 λ-演算的读者而言，这种语言仍会让人感到熟悉。

变量	x, y .. .		
类型变量	X,Y . . .		
整数	n		
标签	ℓ		
进程名称	P		
进程名称集合	ρ	$\in$	$2^{\text{Type 值}}$
种类	K	::=	$* \mid K_1 \Rightarrow K_2 \mid \text{Proc} \mid K \setminus \rho$
类型	τ	::=	$\nu \mid \tau_1\ \tau_2 \mid \tau_1 \to_\rho \tau_2$
		|	$\tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \forall X :: K.\, \tau \mid \lambda X :: K.\, \tau$
类型值	ν	::=	$X \mid () @ \nu \mid \text{整数} @ \nu \mid \nu_1 \to_\rho \nu_2 \mid P$
		|	$\nu_1 + \nu_2 \mid \nu_1 \times \nu_2 \mid \forall X :: K.\, \nu \mid \lambda X :: K.\, \nu$
表达式	M、N ...	::=	$x \mid () @ \nu \mid n @ \nu \mid \lambda\, x : \tau.\, M \mid \Lambda X :: K.\, M$
		|	$M\,N \mid M\ \tau \mid \text{左注入}_\tau M \mid \text{右注入}_\tau M$
		|	$\text{case}\, M\, \text{of 左注入}\, x \Rightarrow N_1; \text{右注入}\, y \Rightarrow N_2$
		|	$(M,N) \mid \text{第一}\, M \mid \text{第二}\, M$
		|	$\text{com}^\tau_{\nu_1,\nu_2} \mid \text{选择}_{\nu_1,\ \nu_2}\, \ell M \mid \text{f}$
值	V	::=	$x \mid () @ \nu \mid n @ \nu \mid \lambda\, x : \tau.\, M \mid \Lambda x :: K.\, M$
		|	$\text{左注入}_\tau V \mid \text{右注入}_\tau V \mid (V_1,V_2)$
		|	$\text{com}^\tau_{\nu_1,\nu_2}$

图 1. PolyChorλ语法

3 多态 Chorλ语言

现在，我们转向第一个主要贡献：设计多态的、编舞式 λ-演算 PolyChorλ。该演算扩展了 Cruz-Filipe 等人（2021年）提出的编舞式 λ-演算 Chorλ，加入了类型多态，更重要的是，加入了进程多态。我们首先介绍 PolyChorλ与基础演算 Chorλ共有的特性，然后再介绍新增特性。PolyChorλ的语法见图 1。

继承自 Chor 的语法λ。 由于编舞程序描述的是整个通信进程网络的行为，我们需要推理各项位于何处。换言之，我们需要知道哪些进程存储了某一项所表示的数据。基本类型的项（例如整数）恰好由一个进程存储。在我们的类型系统中，这通过将基本类型与一个进程名称匹配来表示。例如，由进程 Alice 存储的整数表示为类型 Int @Alice。此类型的值也标明存储它的进程，因此值 5@ Alice (读作“位于 Alice 处的整数 5”）的类型为 Int @ Alice。在图 1中，唯一的基本类型是 () @ P 和 Int @P，不过很容易为语言扩展其他基本类型，例如引言中使用的类型 String @P 或 Bool @ P 。我们将在示例中继续自由使用其他基本类型。

虽然基本类型只位于一个进程上，但更复杂类型的数据可能涉及多个进程。例如，项 (5 @ Alice，42 @ Bob) 同时涉及由 Alice 和 Bob存储的数据。这一点仍记录在类型中：上述项的类型为Int @ Alice × Int @ Bob。除了基本类型和积类型之外，PolyChorλ还具有和类型（写作 $\tau_1 + \tau_2$），以及相应的标准引入形式和消去形式。

函数的处理方式更为特殊：虽然我们具有标准的 λ和应用形式，但也允许函数彼此以相互递归的方式定义。为此，每个 PolyChorλ编排都关联着一个列表 D，其中包含函数到函数变量 f 的绑定，而函数变量本身也是表达式。随后，在执行过程中，可以根据该列表将函数变量实例化为其定义。正如我们将在第 3.3 节中看到的，PolyChorλ项会在一个上下文中求值，该上下文将每个函数变量与一个项相关联。请注意，虽然在原始 Chorλ中，类型也以类似方式相互递归，但在 PolyChorλ中我们不支持递归类型。要了解原因，请注意，我们在语法上将许多类型限制为类型值。这样便无需推理由任意项表示的进程——例如，我们不能向“进程” (λX :: Proc. X) P 发送数据，但可以写作 (ΛY :: Proc. $\mathrm{com}^{\tau}_{Q,Y}$) ((λX :: Proc. X) P)；根据我们的按值调用语义，这会强制类型先归约为 P，然后再实例化 Y。正如我们将在第4节中看到的，允许任意类型之间进行通信会使端点投影变得困难。然而，由于递归类型不一定能归约为类型值，因此它们无法用于类型系统的许多部分。

函数类型也比 λ演算中的常见构造更加具体：它们写作 $\tau_1 \to_\rho \tau_2$。这里，ρ 是进程集合，用来描述除τ_1 和 τ_2之外，哪些进程可以参与函数体（注意，ρ可以包含类型变量）。因此，如果Alice 想直接（不经中介）向Bob 发送一个整数，那么她应该使用类型为整数@Alice $\to_\varnothing$ 整数@ Bob的函数。不过，如果她愿意使用进程 Proxy 作为中介，那么她应该使用类型为整数@Alice $\to_{\{\mathrm{Proxy}\}}$ 整数@ Bob的函数。

为了允许值在进程之间传递，我们提供了原始通信函数 $\mathrm{com}^{\tau}_{P,Q}$。该函数接收一个位于 P 处、类型为 τ的值，并返回其在 Q 处对应的类型。正如引言中所述，大多数编舞语言都提供一种通信项，其模型源自密码协议中的“Alice 与 Bob”记法。例如，Alice -> Bob : 5 可以表示 Alice 向 Bob 发送 5。只需应用函数 $\mathrm{com}^{\tau}_{\mathrm{Alice},\mathrm{Bob}}$，即可轻松实现这一点。例如，项 $\mathrm{com}^{\mathrm{Int@Alice}}_{\mathrm{Alice}\ \mathrm{Bob}}$, (5 @ Alice) 表示 Alice 向 Bob 发送一条包含 5 的消息：其求值结果为 5 @ Bob，类型为 整数 @ Bob。

最后，考虑以下情况，其中 M 的类型为 整数 @ Alice ＋整数 @ Alice：

$$\begin{array}{l}\mathsf{case}\ M\ \mathsf{of}\\ \quad \mathsf{inl}\ \ x \Rightarrow 3\ @\ \mathrm{Bob};\\ \quad \mathsf{inr}\ \ y \Rightarrow 4\ @\ \mathrm{Bob}\end{array}$$

显然，Bob 需要知道选择了哪个分支，因为他需要在每个分支中存储不同的返回值。然而，只有Alice 知道 M 求值结果是右注入$_{\mathrm{Int@Alice}}V$ 还是左注入$_{\mathrm{Int@Alice}}V$（此处左注入和右注入用于表示一个值是和类型的右侧或左侧部分，并标注和类型另一部分的类型，以确保类型主性）。因此，这种编排无法对应任何网络程序。借用编舞语言文献中的术语，我们可以说，这种编排不可实现，因为选择知识不足（Castagna 等人，2012年；Montesi，2023年）。

为了支持进程的行为会根据其他进程的数据而有所不同的程序，例如Bob 的行为会根据Alice的数据而变化，我们提供选择项。这些项允许一个进程告知另一个进程选择了哪个分支，从而避免知识“凭空出现”。例如，我们可以将上面的程序扩展为：

$$\begin{array}{l}\mathsf{case}\ M\ \mathsf{of}\\ \quad \mathsf{inl}\ \ x \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Left}\ (3\ @\ \mathsf{Bob});\\ \quad \mathsf{inr}\ \ y \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Right}\ (4\ @\ \mathsf{Bob})\end{array}$$

这表示与上文相同的程序，只不过Alice 告知 Bob 选择了左分支还是右分支。与前一个版本的示例不同，它确实表示一个（无死锁的）网络程序。一般而言，我们允许通过 选择项发送任意标签，因此可以选择具有语义意义的标签。

虽然 通信 和 选择 都会在两个进程之间传递信息，但它们传递的信息不同。通信 会将一个值（例如整数或函数）从发送方传给接收方。选择 则使用标签告知接收方发送方所做的选择。一些编舞语言将两者结合起来，因此标签和值会同时传递；但与大多数编舞语言一样，PolyChorλ将两者分开处理。

相较于 Chor 的语法扩展λ。 为了在 PolyChorλ中实现（类型和进程）多态，我们添加了若干基于 System Fω（Girard，1972）的特性。具体而言，我们添加了种类和全称类型 $\forall X :: \mathsf{K}.\ \tau$ ，以及类型抽象和应用。我们从 System Fω继承了种类 $*$，它是类型的种类。此外，我们还继承了种类 $\mathsf{K}_1 \Rightarrow \mathsf{K}_2$ ，它表示从类型到类型的函数。

此外，我们继承了来自 System F 的类型级函数 $\lambda X :: \mathsf{K}.\ \tau$ ω。这些函数表示类型构造子的定义。此外，我们还有类型级函数应用 $\tau_1\ \tau_2$。由于类型包含计算，我们还定义了类型值，它们就是不含应用的类型。

请注意，基本类型 () @ ν和 Int @ ν与局部值一样，在语法上受到限制，其子项只能是类型值。这样，我们便可以使用类型变量动态地计算值的位置，但不能使用任意项，否则在投影时将难以判断值的位置。因此，我们可以写作 (λX :: Proc. Int @ X) (Y P)，以动态地计算整数的位置（在实例化 X 之前，Y P 必须先规约为类型值），但不能直接写作 Int @ (Y P)。这样，我们的投影演算就能在（运行时）实例化 X 时判断它是否被实例化为 P。若要检查 Y 是否被实例化为输出 P 的函数类型，则会更加复杂。

除了 System F 的种类 $*$ 和 $\mathsf{K}_1 \Rightarrow \mathsf{K}_2$ 之外ω，我们还有进程名称的种类 Proc 。因此，进程名称是类型，但不能用于为任何项指定类型。此外，我们还有种类 $\mathsf{K} \setminus \rho$，它表示种类为 K 且不提及集合 ρ中任何进程的类型。由于我们会根据类型所包含的进程来限制可通信的类型，正如我们很快将看到的，这种受限种类可用于定义包含通信的多态函数。例如，项

$$\Lambda X :: \mathsf{Proc}.\ \Lambda Y :: \mathsf{Proc} \setminus \{X\}.\ \mathsf{com}_{X,Y}^{\mathsf{Int}\ @\ X}\ (5\ @\ X)$$

定义了一个函数：给定互不相同的进程 X 和 Y 后，该函数会使 X 向 Y 发送 5。

在本节剩余部分，我们将探讨 PolyChorλ 的语义。首先，我们考察其静态语义，包括类型判断和种类判断两种形式。其次，我们描述其操作语义。在全文中，我们将继续基于 PolyChorλ 的并发解释给出直观说明，不过，本文给出的语义并不直接对应于该解释。

3.1 类型判断

现在我们转向 PolyChorλ 的类型系统。和之前一样，我们的类型系统建立在 Chorλ 的类型系统之上。这里，我们重点讨论本研究中新引入的规则。因此，我们将重点讨论与多态相关的规则，以及因多态而必须修改的规则。

PolyChor 的类型判断式λ 具有如下形式： $\Theta; \Gamma \vdash M : \tau$，其中 Θ 是进程名称的集合——包括 N 中的名称，或种类为 Proc的类型变量——它们被用于 M 中，或用于 M 的类型中。 类型环境 Γ 是一个列表，将变量和函数名与其类型相关联，并将类型变量和进程名称与其种类相关联。我们有时将 $\Theta; \Gamma$ 这一对称为类型上下文。

我们类型系统的选定规则见图 2。完整的规则集见附录 1。同样，许多规则直接继承自 Chorλ（Cruz- Filipe 等，2021年）；因此，我们重点介绍因新增内容而发生变化的规则。这些规则中有许多（甚至大多数）受 System Fω的启发。不过，加入进程的种类和 Without 种类——即形如 $K \setminus \rho$的种类——也带来了一些变化。

规则 [Tunit] 和 [Tint] 为基本类型的值赋予类型。这里，我们必须确保项的位置是一个进程。直观而言，我们希望该位置具有种类 Proc。不过，它可能带有限制，也就是说，可能具有 $\text{Proc} \setminus \rho$的形式。在这种情况下，我们的子种类系统（详情见第 3.2 节）仍允许我们应用该规则。

我们分别通过 [Tapp] 和 [Tabs] 规则来表示函数应用和抽象。应用规则 [Tapp] 基本上是标准的，唯一的补充是在函数类型上添加一个集合，如前文所述。另一方面，抽象规则 [Tabs], 则更加复杂。首先，它确保参数类型 τ_1 的种类为 $*$。然后，它确保修饰箭头的集合中的每个元素都是进程名称，也就是说，其种类为 Proc。最后，它检查在扩展环境中，函数体具有输出类型 τ_2。通常，这个扩展环境会为参数指定类型。不过，它会将可用的进程名称限制为集合 ρ中的进程名称，以及类型 τ_1 和 τ_2 中提及的进程名称。

类型 τ提及进程有两种方式：可以直接命名该进程，也可以通过类型变量命名。因此，在规则 [Tabs] 中，我们允许τ_1 和 τ_2 的自由变量保留在进程上下文中，并使用（标准的）自由类型变量函数来计算它们，其中 $\forall X :: K.M$ 和 $\lambda X :: K.M$ 都绑定 X。不过，我们还必须

$$[\textsc{Tunit}]\ \frac{\Theta;\Gamma\vdash \nu :: \mathsf{Proc}}{\Theta;\Gamma\vdash () \mathbin{@} \nu : () \mathbin{@} \nu} \qquad [\textsc{Tint}]\ \frac{\Theta;\Gamma\vdash \nu :: \mathsf{Proc}}{\Theta;\Gamma\vdash n \mathbin{@} \nu : \mathsf{Int} \mathbin{@} \nu}$$

$$[\textsc{Tapp}]\ \frac{\Theta;\Gamma\vdash N : \tau_1 \to_\rho \tau_2 \qquad \Theta;\Gamma\vdash M : \tau_1}{\Theta;\Gamma\vdash N\ M : \tau_2}$$

$$[\textsc{Tabs}]\ \frac{\Theta;\Gamma\vdash \tau_1 :: * \qquad \Theta;\Gamma'\vdash \nu :: \mathsf{Proc} \text{ for all } \nu\in\rho \qquad \Theta\cap(\rho\cup \mathsf{ip}(\tau_1)\cup\mathsf{ip}(\tau_2)\cup \mathrm{ftv}(\tau_1)\cup\mathrm{ftv}(\tau_2)); \Gamma, x:\tau_1\vdash M:\tau_2}{\Theta;\Gamma\vdash \lambda\, x:\tau_1.\, M : \tau_1 \to_\rho \tau_2}$$

$$[\textsc{Tsel}]\ \frac{\Theta;\Gamma\vdash \nu_1 :: \mathsf{Proc} \qquad \Theta;\Gamma\vdash \nu_2 :: \mathsf{Proc} \qquad \Theta;\Gamma\vdash M : \tau}{\Theta;\Gamma\vdash \mathsf{select}_{\nu_1,\nu_2}\ \ell\ M : \tau}$$

$$[\textsc{Tcom}]\ \frac{\Theta;\Gamma\vdash \tau :: \mathsf{Proc} \Rightarrow * \qquad \Theta;\Gamma\vdash \nu_1 :: \mathsf{Proc} \setminus (\mathsf{mp}(\tau)\cup\mathrm{ftv}(\tau)) \qquad \Theta;\Gamma\vdash \nu_2 :: \mathsf{Proc} \setminus (\mathsf{mp}(\tau)\cup\mathrm{ftv}(\tau))}{\Theta;\Gamma\vdash \mathsf{com}^{\tau}_{\nu_1,\nu_2} : (\tau\ \nu_1 \to_\varnothing \tau\ \nu_2)}$$

$$[\textsc{TappT}]\ \frac{\Theta;\Gamma\vdash M : \forall X :: \mathsf{K}.\, \tau_1 \qquad \Theta;\Gamma\vdash \tau_2 :: \mathsf{K}}{\Theta;\Gamma\vdash M\ \tau_2 : \tau[X\mapsto\tau_2]}$$

$$[\textsc{TabsT}]\ \frac{\begin{array}{c}\Theta';\Gamma', X :: \mathsf{K}\vdash M : \tau \\ \text{if } \exists \mathsf{K}', \rho.\, \mathsf{K} = \mathsf{K}'\setminus\rho \text{ then } \Gamma' = (\Gamma + X)\ \&\ \rho\setminus X \text{ else } \Gamma' = \Gamma + X \\ \text{if } \mathsf{K} = \mathsf{Proc} \text{ or } \exists \rho.\, \mathsf{K} = \mathsf{Proc}\setminus\rho \text{ then } \Theta' = \Theta\cup\{X\} \text{ else } \Theta' = \Theta\end{array}}{\Theta;\Gamma\vdash \Lambda X :: \mathsf{K}.\, M : \forall X :: \mathsf{K}.\, \tau}$$

$$[\textsc{Teq}]\ \frac{\Theta;\Gamma\vdash M : \tau_1 \qquad \tau_1 \equiv \tau_2 \qquad \Theta;\Gamma\vdash \tau_2 :: *}{\Theta;\Gamma\vdash M : \tau_2}$$

图 2. 类型规则（选定）

确定类型中涉及的进程，记作 $\mathsf{ip}(\tau)$，并按如下方式计算：

$$\mathsf{ip}(X) = \varnothing \qquad \mathsf{ip}(\mathsf{P}) = \mathsf{P} \qquad \mathsf{ip}(() \mathbin{@} \nu) = \mathsf{ip}(\mathsf{Int} \mathbin{@} \nu) = \mathsf{ip}(\nu)$$

$$\mathsf{ip}(\forall X :: \mathsf{K}\setminus\rho.\, \tau) = \mathsf{ip}(\lambda X :: \mathsf{K}\setminus\rho.\, \tau) = \mathsf{ip}(\tau)\cup(\mathbf{N}\setminus\rho)$$

$$\mathsf{ip}(\forall X :: \mathsf{K}.\, \tau) = \mathsf{ip}(\lambda X :: \mathsf{K}.\, \tau) = \mathbf{N} \text{ if } \nexists \mathsf{K}', \rho.\, \mathsf{K} = \mathsf{K}'\setminus\rho$$

其他类型的涉及进程以同态方式定义。

通信原语 选择 和 通信 分别通过 [Tsel] 和 [Tcom],进行类型判断。项 选择$_{\nu_1,\ \nu_2}\ \ell$ M 的行为与 M 相同，其中进程 ν_1 会通知进程 ν_2 已选择 ℓ 分支，正如我们之前所见。因此，如果 M 具有类型τ，则整个项也具有该类型。此外， ν_1 和 ν_2 必须是进程。

规则 [Tcom] 类型 通信 项。为了便于阅读，到目前为止，我们一直简化通信 $^{\tau}_{\mathsf{P}}$中使用的类型Q。我们一直用 τ表示输入类型，但事实证明，要

类型 $通信^{\tau}_{P,\ Q}$ 若要正确表达，我们需要稍微增加一些复杂性。直观而言，项 $通信^{\tau}_{\nu_1,\ \nu_2}\ M$ 表示 ν_1 将 M 在 ν_1 上的各部分通信到 ν_2。因此，我们要求 τ是一个需要进程的类型变换器。此外，ν_1 和 ν_2 不能在 τ 中提及；否则，在上面的例子中，M 在 ν_1 上的类型并非每一部分都会转移到 ν_2。为此，我们使用以下关于被提及进程的概念：

$$\mathrm{mp}(X) = \varnothing \qquad \mathrm{mp}(\mathsf{P}) = \mathsf{P} \qquad \mathrm{mp}(() \ @\ \nu) = \mathrm{mp}(\mathsf{Int} \ @\ \nu) = \mathrm{mp}(\nu)$$

$$\mathrm{mp}(\forall X :: \mathsf{K} \setminus \rho.\, \tau) = \mathrm{mp}(\lambda X :: \mathsf{K} \setminus \rho.\, \tau) = \mathrm{mp}(\tau) \cup \rho$$

$$\mathrm{mp}(\forall X :: \mathsf{K}.\, \tau) = \mathrm{mp}(\lambda X :: \mathsf{K}.\, \tau) = \mathrm{mp}(\tau) \text{ if } \nexists \mathsf{K}', \rho.\, \mathsf{K} = \mathsf{K}' \setminus \rho$$

再次说明，其他类型也以同态方式定义。涉及的进程与提及的进程之间的区别很微妙。若没有多态，它们就是相同的；但在处理带有限制的多态时，两者则恰好相反：涉及的进程包括限制中未出现的每个进程（变量可能被实例化为涉及这些进程的内容，因此这些进程可能会参与其中），而提及的名称则包括限制中提到的进程。提及的名称仅用于类型判断通信。如果我们有这样一个类型级函数，τ，以及两个类型值 ν_1 和 ν_2，它们都不是、将来也不会被实例化为 τ中提及的任何内容，那么我们可以将 $通信^{\tau}_{\nu_1,\nu_2}$类型判断为从 $\tau\ \nu_1$ 到 $\tau\ \nu_2$的函数。由于这是直接通信，不需要中介，因此我们可以将这条箭头与空集 $\varnothing$关联起来。

值得一提的是，通信规则启发我们使用 System Fω，而不是缺少类型级计算能力的纯 System F。在 Chorλ和其他早期的编舞语言中，通信的值必须是发送方本地的。在 PolyChorλ中，这意味着通信的类型不能包含类型变量，也不能包含发送方以外的进程。由于我们引入了将通信用作委托手段的思想，因此放宽了这一限制。这意味着 PolyChorλ程序可以通信更大型的编排，其类型可能涉及其他进程，更重要的是，还可能涉及其他类型变量。我们可以在第 1 节末尾的委托示例中看到这一点，其中有如下通信：$通信_{\mathsf{Seller},\mathsf{Seller}_2}$。加入所需的类型注解（我们在引言中省略了它），便得到 $通信^{\lambda X :: \mathsf{Proc}.\ \mathsf{String}@X \to_{\varnothing} ()@B\ 卖方,卖方_2}$。请注意，这仍然留下了一个自由类型变量 B，它表示 卖方要告知 $卖方_2$ 与之交互的未知进程！由于我们不能禁止通信类型中出现自由类型变量，因此必须创建一个能够处理它们的类型系统，而这需要类型级计算。

要了解这为何引导我们采用类型级计算，不妨考虑另一种方案。在 Chorλ及其他编舞式作品中，我们会使用过程替换而非通信，得到一种通信类型。此时，带注释的程序将是$通信^{\mathsf{String}@\mathsf{Seller} \to_{\varnothing} ()@B\ 卖方,卖方_2}$。将其应用于类型适当的程序后，结果的类型将是

$$(字符串\ @\ 卖方\ \to_{\varnothing}\ ()\ @B)[卖方\ \mapsto 卖方_2] = 字符串\ @\ 卖方_2 \to_{\varnothing} ()\ @\ B$$

请注意，由于 B 是一个类型变量，因此替换时忽略了它。如果之后将 B 替换为包含 Seller的内容，那么我们就丢失了所需的信息，无法将其更改为 Seller_2。因此，我们需要某种机制来延迟这种替换；

我们没有采用显式替换之类的机制，而是转而使用 System Fω这一标准工具。这似乎更优雅，也不那么临时拼凑；此外，它还加入了一些现实世界中的 PolyChorλ实现无论如何都会需要的功能。

现在回到图 2中的类型规则，接下来介绍 [TappT] 和 [TabsT]规则，它们用于全称量化。 [TappT] 规则完全是标准的，而 [TabsT] 规则则是我们最复杂的规则。在这条规则中，根据 M的种类，我们为其类型上下文提供了 4 种不同的定义。按照惯例，我们会检查函数体在参数 X 具有种类 K时，是否具有正确的类型。不过，首先，如果 X 是一个进程，那么我们需要将 Θ 和 X加入其中。此外，我们还必须进一步调整上下文，确保那些种类受 X 限制的类型，与 X的种类限制相对应。

首先，新的类型变量 X 可能会遮蔽先前定义的X。因此，我们需要从上下文中已有的所有限制里移除 X 。我们使用以下操作来实现这一点 $\mathsf{K} + \nu$:

$$(\mathsf{K} \setminus \rho) + \nu = (\mathsf{K} + \nu) \setminus (\rho \setminus \{\nu\})$$

我们在其他种类上同态地定义 $+$，并像往常一样将其扩展到上下文 l:

$$\Gamma + \nu = \{x : \tau \mid x : \tau \in \Gamma\} \cup \{X : \mathsf{K} + \nu \mid X : \mathsf{K} \in \Gamma\}$$

此外，如果 X 本身具有 Without 种类——也就是说，X 的种类表明它不可能是 ρ 中的任何进程——那么我们就需要对 X 对 ρ 中的每种类型对称地添加限制。否则，我们就无法在不能使用 X 的任何地方使用 ρ 中的角色，尽管我们知道X 不会被实例化为这些进程。我们通过操作 $\Gamma \,\&\, \rho \setminus X$ 来实现这一点，定义如下:

$$\begin{aligned}\Gamma \,\&\, \rho \setminus X = \; & \{x : \tau \mid x : \tau \in \Gamma\} \cup \{\tau :: \mathsf{K} \mid \tau :: \mathsf{K} \in \Gamma \text{ and } \tau \notin \rho\} \\ & \cup \{\tau :: \mathsf{K} \setminus (\rho_2 \cup \{X\}) \mid \tau :: \mathsf{K} \setminus \rho_2 \in \Gamma \text{ and } \tau \in \rho\} \\ & \cup \{\tau :: \mathsf{K} \setminus \{X\} \mid \tau :: \mathsf{K} \in \Gamma, \mathsf{K} \neq \mathsf{K}_2 \setminus \rho_2, \text{ and } \tau \in \rho\}\end{aligned}$$

有了这些操作，我们现在就能完整理解 [TabsT]。当 K 实际上是 Without 种类时，我们必须同时处理遮蔽和对称限制。但是，当它不是 Without 种类时，我们只需处理遮蔽。

类型系统最后加入的规则是 [Teq]。这是 System F 中另一个标准规则ω；它告诉我们可以在类型中进行计算。更具体地说，它告诉我们

我们可以使用以下等价关系，将一种类型替换为等价的类型：

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to_\rho \tau_2 \equiv \tau_1' \to_\rho \tau_2'} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 + \tau_2 \equiv \tau_1' + \tau_2'} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'}$$

$$\frac{\tau \equiv \tau'}{\lambda X :: \mathsf{K}.\, \tau \equiv \lambda X :: \mathsf{K}.\, \tau'} \qquad \frac{}{(\lambda X :: \mathsf{K}.\, \tau_1)\, \tau_2 \equiv \tau_1[X \mapsto \tau_2]} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\, \tau_2 \equiv \tau_1'\, \tau_2'}$$

$$\frac{\tau \equiv \tau'}{\forall X :: \mathsf{K}.\, \tau \equiv \forall X :: \mathsf{K}.\, \tau'}$$

虽然 [Teq] 是我们添加到类型系统中的最后一条规则，但它并不是图2中的最后一条规则。我们还添加了一种额外的判断，形式为 $\Theta; \Gamma \vdash \mathrm{D}$，其中 $\Theta; \Gamma$ 与之前一样是类型上下文，而 D 是函数变量的定义集——即 $\mathrm{D} = \{\mathrm{f}_1 = \mathrm{M}1, \ldots \mathrm{f}_n = \mathrm{M}_n\}$。我们用 D(f) 表示 D 中与 f 关联的项。此判断唯一的规则是 [Tdefs], ，它指出，若 D 中的每个变量都关联有类型 τ （位于 Γ 中），且 D 中 f 的函数体在上下文 /0; Γ 下可被赋予类型 τ ，则该定义集是良构的。我们要求 f 的函数体能够在角色集为空的情况下通过类型判断，因为它们是全局预定义函数，因此不应局限于任何单个进程。

$$[\textsc{Tdefs}]\ \frac{\forall f \in \mathsf{domain}(D).\ f : \tau \in \Gamma \wedge \emptyset; \Gamma \vdash D(f) : \tau}{\Theta; \Gamma \vdash D}$$

3.2 种类判断

我们最后讨论 PolyChor 的静态语义λ，具体来看我们的种类系统。我们的种类系统只使用一种判断， $\Theta; \Gamma \vdash \tau :: \mathsf{K}$，它表示在类型上下文 $\Theta; \Gamma$中，类型 τ具有种类 K。我们的种类系统规则见图3。这些规则大多直接继承自 System Fω。不过，我们必须考虑 Proc 和 Without 种类。

例如，规则 [Kunit] 和 [Kint] 用于检查表示存储数据的进程的类型确实具有种类 Proc。类似地， [Kfun] 确保所有可能中间项集合中的类型都是进程。类型变量的规则 [Kvar],确保如果类型变量X 被赋予种类 Proc，那么 X 也必须属于 Θ。

我们的种类系统与 System F 的种类系统之间最大的区别之一ω，然而，是规则 [Ksub] ，它表明我们的系统具有子种类关系。子种类关系规则源自 Without 种类上的子集序。子种类关系规则如下

$$[\textsc{Kvar}]\ \frac{X :: \mathsf{K} \in \Gamma \qquad \text{if } \mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\} \text{ then } X \in \Theta}{\Theta; \Gamma \vdash X :: \mathsf{K}}$$

$$[\textsc{Krole}]\ \frac{\mathsf{P} :: \mathsf{K} \in \Gamma \qquad \mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\} \qquad \mathsf{P} \in \Theta \qquad \text{if } \mathsf{K} = \mathsf{Proc} \setminus \rho \text{ then } \mathsf{P} \notin \rho}{\Theta; \Gamma \vdash \mathsf{P} :: \mathsf{K}}$$

$$[\textsc{Kunit}]\ \frac{\Theta; \Gamma \vdash \tau :: \mathsf{Proc} \setminus \rho}{\Theta; \Gamma \vdash () \ @\ \tau :: * \setminus \rho} \qquad [\textsc{Kint}]\ \frac{\Theta; \Gamma \vdash \tau :: \mathsf{Proc} \setminus \rho}{\Theta; \Gamma \vdash \mathsf{Int} \ @\ \tau :: * \setminus \rho}$$

$$[\textsc{Kfun}]\ \frac{\Theta; \Gamma \vdash \tau_1 :: * \setminus \rho_2 \qquad \Theta; \Gamma \vdash \tau_2 :: * \setminus \rho_2 \qquad \forall \nu \in \rho_1.\ \Theta; \Gamma \vdash \nu :: \mathsf{Proc} \setminus \rho_2}{\Theta; \Gamma \vdash \tau_1 \to_{\rho_1} \tau_2 :: * \setminus \rho_2}$$

$$[\textsc{Kabs}]\ \frac{\Theta'; \Gamma, X :: \mathsf{K}_1 \vdash \tau :: \mathsf{K}_2}{\Theta; \Gamma \vdash \lambda X :: \mathsf{K}_1.\ \tau :: \mathsf{K}_1 \Rightarrow \mathsf{K}_2} \qquad [\textsc{Kall}]\ \frac{\Theta'; \Gamma, X :: \mathsf{K} \vdash \tau :: * \setminus \rho}{\Theta; \Gamma \vdash \forall X :: \mathsf{K}.\ \tau :: * \setminus \rho}$$

$$[\textsc{Karr}]\ \frac{\Theta; \Gamma \vdash \tau :: (\mathsf{K}_1 \setminus \rho) \Rightarrow (\mathsf{K}_2 \setminus \rho)}{\Theta; \Gamma \vdash \tau :: (\mathsf{K}_1 \Rightarrow \mathsf{K}_2) \setminus \rho} \qquad [\textsc{Ksub}]\ \frac{\Theta; \Gamma \vdash \tau :: \mathsf{K}_1 \qquad \mathsf{K}_1 <: \mathsf{K}_2}{\Theta; \Gamma \vdash \tau :: \mathsf{K}_2}$$

$$[\textsc{Ksum}]\ \frac{\Theta; \Gamma \vdash \tau_1 :: * \setminus \rho \qquad \Theta; \Gamma \vdash \tau_2 :: * \setminus \rho}{\Theta; \Gamma \vdash \tau_1 + \tau_2 :: * \setminus \rho}$$

$$[\textsc{Kprod}]\ \frac{\Theta; \Gamma \vdash \tau_1 :: * \setminus \rho \qquad \Theta; \Gamma \vdash \tau_2 :: * \setminus \rho}{\Theta; \Gamma \vdash \tau_1 \times \tau_2 :: * \setminus \rho}$$

图3. 种类判定规则

所示:

$$[\textsc{SKRefl}]\ \frac{}{\mathsf{K} <: \mathsf{K}} \qquad [\textsc{SKTrans}]\ \frac{\mathsf{K}_1 <: \mathsf{K}_2 \qquad \mathsf{K}_2 <: \mathsf{K}_3}{\mathsf{K}_1 <: \mathsf{K}_3}$$

$$[\textsc{SKArr}]\ \frac{\mathsf{K}_1 <: \mathsf{K}'_1 \qquad \mathsf{K}_2 <: \mathsf{K}'_2}{\mathsf{K}_1 \Rightarrow \mathsf{K}_2 <: \mathsf{K}'_1 \Rightarrow \mathsf{K}'_2} \qquad [\textsc{SKEmpty}]\ \frac{}{\mathsf{K} <: \mathsf{K} \setminus \varnothing}$$

$$[\textsc{SKWithoutL}]\ \frac{\mathsf{K}_1 <: \mathsf{K}_2}{\mathsf{K}_1 \setminus \rho <: \mathsf{K}_2} \qquad [\textsc{SKWithoutUnion}]\ \frac{\mathsf{K}_1 <: \mathsf{K}_2}{\mathsf{K}_1 \setminus (\rho_1 \cup \rho_2) <: \mathsf{K}_2 \setminus \rho_1}$$

引理1. 设 τ 是一个类型。若存在一个类型上下文 Θ; Γ 使得 Θ; $\Gamma \vdash \tau :: K$ 则存在唯一的类型值 ν 使得 $\tau \equiv \nu$。

证明 其存在性 ν 可由对 Θ 的归纳法得到; $\Gamma \vdash \tau :: \mathsf{K}$ 而其唯一性可由对 $\tau \equiv \nu$ 的归纳法得到。
■

$$[\text{应用标签}]\ \frac{\tau \equiv \nu}{(\Lambda X :: K.\, M)\ \tau \to_D M[X \mapsto \nu]} \qquad [\text{MTAPP1}]\ \frac{M_1 \to_D M_2}{M_1\ \tau \to_D M_2\ \tau}$$

$$[\text{定义}]\ \frac{}{f \to_D D(f)}$$

$$[\text{选择}]\ \frac{}{\text{选择}_P\ ,^{\text{问}}\ \ell M \to_D M} \qquad [\text{通信}]\ \frac{}{\text{通信}^{\tau}_{P,Q}\ V \to_D V[P \mapsto Q]}$$

图 4. PolyChorλ（选定规则）的语义

引理 2 (类型限制)。 设 τ是一个类型。若存在一个类型上下文 Θ； Γ 使得Θ； $\Gamma \vdash \tau :: K \setminus \rho$ 则 (进程标识集合 $(\tau) \cup ftv(\tau)) \cap \rho = /0$

证明由种类判断规则直接得出。 ■

定理 1 （可定种类类型）。 设 M 是一个编排，且 τ 是一个类型，使得 Θ； $\Gamma \vdash M : \tau$。则 Θ； $\Gamma \vdash \tau :: *$。

证明 由 Θ； $\Gamma \vdash M : \tau$的推导归纳法及种类判断规则直接得出。 ■

我们还发现，类型与其等价类型值具有相同的种类。由于 β- 展开，可定种类类型可能等价于不可定种类类型，但不可能等价于不可定种类的类型值。

定理 2 (种类保持)。 设 τ是一个类型。若存在一个类型上下文 Θ； Γ，使得 Θ； $\Gamma \vdash \tau :: K$，那么对于任意类型值 ν，只要 $\tau \equiv \nu$，就有 Θ； $\Gamma \vdash \nu :: K$。

证明 由种类判断规则和类型等价规则可得。我们利用这样一个事实：如果使用规则 $(\lambda X :: K.\, \tau_1)\ \tau_2 \equiv \tau_1[X \mapsto \tau_2]$，通过额外的应用构造一个无法进行种类判断的 $\tau_2 \equiv \tau_1$，那么在得到基本类型之前，也必须使用同一规则移除这个新的类型应用。 ■

例 1. 我们回到第 1 节中的委托示例，并尝试对其进行类型判断。由于 B在一个值的类型中自由出现，而 F是在 卖方 与 卖方_2之间通信的，因此 B 实际上必须具有受限种类 Proc $\setminus \{\text{卖方}, \text{卖方}_2\}$。因此，该编排获得类型

$$\forall B :: \text{Proc} \setminus \{\text{卖方}, \text{卖方}_2\}.\text{字符串}\ @\ B \to^{\{\text{Seller},\text{Seller}_2\}} ((\text{整数}\ @\ B \to_{\varnothing} \text{布尔值}\ @\ B) \to^{\{\text{Seller},\text{Seller}_2\}} ()\ @\ B)$$

这种类型展示了编排的输入、输出以及涉及的角色。

3.3 操作语义

最后，我们考虑 PolyChorλ 的操作语义。这些语义基本上是针对类型化 λ演算的标准按值调用归约语义。不过，归约语义

还必须携带一组D函数定义。只有少数规则比较特殊或需要修改；这些规则见图4。其余规则见附录2。

规则 [AppTAbs] 和 [MTApp1] 源自 System Fω。规则 [AppTAbs] 与普通的 CBV β 归约类似，但它说明了如何归约一个应用于类型值的类型抽象；不过需要注意，如果没有类型值，我们必须先使用类型等价来得到一个，然后才能进行归约。规则 [MTApp1] 说明了可以归约应用于任意参数的类型函数。

规则 [Def] 允许我们通过查找集合 D 中的定义来归约函数名。

最后，我们有通信规则。规则 [Sel] 说明选择操作是一个空操作，正如我们前面所述。虽然这看起来可能多余，但正如我们将在下一节看到的，此类项对于投影至关重要。更重要的是，[Com] 规则说明了如何在编舞层表示通信：通过角色替换。这也有助于解释 [Tcom] 中的一些限制。由于我们将 V 中所有提及 P 的地方都替换为 Q，因此不能允许在 V 的类型变换器中其他地方提及 P。否则，可能会出现一些在通信过程中不应被替换的 P，但我们的模型并未对此进行建模。与对 com$^{\tau}_{\mathsf{P},\mathsf{Q}}$ V 进行类型判断时不同，在执行通信时，我们知道（因为我们只考虑没有自由变量的编排）τor V 中的任何类型变量都已经实例化，因此无需考虑如何替换那些之后可能会被实例化为 P 或 Q 的变量。

得知我们的语义只是按值调用归约语义，尤其是对熟悉编排的读者而言，可能会令人惊讶。毕竟，编排本应表示并发程序，因此任何时刻都应该存在多个可归约表达式。事实上，早期关于编排编程的研究（例如 Hirsch 和 Garg，2022年；Cruz-Filipe 和 Montesi，2020年；Carbone 和 Montesi，2013年）提出了一种包含乱序执行的语义，使编排的操作语义与并发解释中的所有可能归约相匹配。相反，我们采用这种更简单、不包含乱序执行的语义。作为交换，第5 节中的结果将会较弱：我们只保证，编排能够归约得到的任何值，并发解释也能够归约得到。

要了解我们为何选择得到这个较弱的结果，请看下面的编排：

$$f\,((\mathsf{com}^{\lambda X::\mathsf{Proc}.\,\mathsf{Int}@X}_{\mathsf{Q}_1,\mathsf{Q}_2}\ (3\ @\ \mathsf{Q}_1)), (4\ @\ \mathsf{P}))$$

这里有一个函数f 需要用一个分布式对实例化。P 已经准备好将参数中属于自己的部分传入f 并开始计算结果，而Q_1 和Q_2 仍在计算参数中属于它们的部分。我们可以用两种方式并发地解释 PolyChorλ：一种是在所有进程都进入函数时进行同步；另一种则是允许 提前进入函数。我们选择第二种更实用的方式。不过，这意味着若不禁止分布式值，或不允许以某种方式分多个步骤调用单个值，就无法将至少一种求值顺序反映到编排的语义中。这一认识促使我们采用了上文所述的较弱保证。

与标准的按值调用 λ-演算一样，我们能够证明，相对于我们的操作语义，我们的类型系统是可靠的，具体由以下两个定理表述：

定理 3 (类型保持)。 设 M 是一个编排，D 是一个函数映射，包含 M 中的每个函数。若存在一个类型上下文 $\Theta;\Gamma$，使得 $\Theta;\Gamma\vdash M:\tau$，并且 $\Theta;\Gamma\vdash D$，则对于任意满足 M $\Theta;\Gamma\vdash M':\tau$的 M'，都有 $\to_D M'$。

证明 由类型判断规则和语义规则以及定理 2直接得出。 ■

定理 4 （进展性）。 设 M 是一个封闭编排，D 是一个函数映射，包含 M 中的每个函数。若存在一个类型上下文 $\Theta;\Gamma$ 使得 $\Theta;\Gamma\vdash M:\tau$并且$\Theta;\Gamma\vdash D$，则 M $=$V，或者存在 M' 使得 M $\to_D M$。

证明 由类型判断规则和语义规则直接得出。 ■

4 端点投影

现在我们将讨论任何编舞式编程语言最重要的结果： 端点投影。端点投影为我们的语言 PolyChor 提供了一种并发解释λ，即将其转换为程序的并行组合，每个进程对应一个程序。不过，要定义端点投影，我们必须先定义进程语言，我们称之为局部语言。局部语言的语法见图 5。图中还列出了局部转移标签和网络转换标签的语法；我们将在介绍网络的操作语义时对二者进行说明。

与 PolyChor 一样λ，我们的局部语言继承了 System F 的许多结构ω。具体来说，我们有积类型、和类型、函数、全称量化和λ类型，以及与之对应的项。事实上，有些类型看起来更像标准的 System Fω，而非 PolyChorλ：函数类型不需要指定可以参与函数的进程集合，基本类型也不再需要位置。

不过，并非所有内容都那么熟悉；我们引入了新的项和新的类型。send_ν 和 recv_ν 这两个项分别允许项发送和接收值。我们还将 select 项拆分为两个项：提供项 $\&_\nu\ \{\ell_1:\mathrm{L}1,\ldots,\ell_n:\mathrm{L}_n\}$，它允许 ν 选择该项将如何演化。我们使用形式为 $\oplus_\nu\ \ell\ \mathrm{L}$ 的选择项来表示这类选择。该项告知由 ν 表示的进程，它应当规约为标记为 ℓ的子项，随后自身规约为项 L。尽管这些对于 System Fω 这类多态语言而言并不寻常，但它们在π演算等进程语言中却很常见。我们还加入了未定义类型和项，分别记作 $\bot$ 和 $\bot$。它们表示定义不当的项；我们用它们表示某个进程 P 上不存在、但为了结构上完整而需要写入 P 的程序中的数据。

我们还引入了一项较为少见的特性：进程的显式替换。项 $\mathsf{sub}[\nu_1\mapsto\nu_2]$ 是一个函数，应用时会将其参数中由 ν_1表示的角色替换为由 ν_2 表示的角色。借助这个函数，我们可以表示第三方视角下的通信：角色只需更改，无需任何额外机制。例如，假设 Alice 想告诉 Bob 向 Cathy发送一个整数。她可以通过向 Bob 发送函数 $通信^{\lambda X::\mathsf{Proc}.\ \mathsf{Int}@X,\ \mathsf{Cathy}}_{\mathsf{Alice}}$ 来实现。在 PolyChorλ 中，

变量	$x, y \ldots$		
类型变量	$X, Y \ldots$		
进程名称	P		
局部转移标签	μ	$::=$	$\tau \mid P \mid 发送_P\, V\, V' \mid 接收_P\, V'\, V$
		$\mid$	$\oplus_P\, \ell \mid \&_P\, \ell$
网络转移标签	μ	$::=$	$\tau_{\mathscr{P}}$
进程标签	$\mathscr{P}$	$::=$	$P \mid P, Q$
类型	τ	$::=$	$\nu \mid \tau_1\, \tau_2 \mid \mathrm{AmI}\ \nu?\ \tau_1 \,\&\, \tau_2 \mid \tau_1 \to \tau_2$
		$\mid$	$\tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \forall \mathrm{X}.\, \tau \mid \lambda \mathrm{X}.\, \tau$
类型值	ν	$::=$	$X \mid () \mid 整数 \mid \nu_1 \to \nu_2 \mid P \mid \bot$
		$\mid$	$\nu_1 + \nu_2 \mid \nu_1 \times \nu_2 \mid \forall \mathrm{X}.\, \nu \mid \lambda \mathrm{X}.\, \nu$
表达式	$\mathrm{M}, \mathrm{N} \ldots$	$::=$	$x \mid () \mid n \mid \lambda\, x : \tau.\, M \mid \Lambda X.\, M$
		$\mid$	$M_1\, M_2 \mid M\ \tau \mid 左注入_\tau\, M \mid 右注入_\tau M$
		$\mid$	$\mathsf{case}\, M\, \mathsf{of}\, 左注入\, x \Rightarrow M_1; 右注入\, y \Rightarrow M_2$
		$\mid$	$(M_1, M_2) \mid 第一项\, M \mid 第二项\, M$
		$\mid$	$发送_\nu \mid \mathrm{recv}_\nu$
		$\mid$	$\&_\nu\, \{\ell_1 : \mathrm{M}_1, \ldots, \ell_n : \mathrm{M}_n\} \mid \oplus_\nu\, \ell\, \mathrm{M}$
		$\mid$	$子[\nu_1 \mapsto \nu_2] \mid f \mid \mathrm{AmI}\ \nu?\ M_1 \,\&\, M_2$
值	V	$::=$	$x \mid () \mid n \mid \bot \mid \lambda\, x : \tau.\, M \mid \Lambda X.\, M$
		$\mid$	$左注入_\tau\, V \mid 右注入_\tau V \mid (V_1, V_2)$
		$\mid$	$发送_\nu \mid 接收_\nu \mid 子[\nu_1 \mapsto \nu_2]$

图 5. 局部语言语法

这对应于以下编排

$$\mathsf{com}^{\lambda X :: \mathsf{Proc}.\, \mathsf{Int}@X \to_{\varnothing} \mathsf{Int}@\mathsf{Cathy}}_{\mathsf{Alice}, \mathsf{Bob}} \left(\mathsf{com}^{\lambda X :: \mathsf{Proc}.\, \mathsf{Int}@X}_{\mathsf{Alice}, \mathsf{Cathy}} \right)$$

为了投影这个编排，即使上面的通信函数没有应用于任何参数，我们也需要能够对其进行投影。这时就要用到显式替换：我们将通信函数投影为 $\mathsf{sub}[\mathsf{Alice} \mapsto \mathsf{Bob}]$。

最后，我们介绍一项独特特性：AmI 项及其对应的类型。它们表示进程能够知道自身身份，并基于这一认知采取行动。进程多态要求在进程 P 处实例化一个进程变量时，必须附带一个条件表达式，用于判断该变量实例化为 P，还是实例化为另一个可能与 P 交互的进程。具体而言，若由 ν 所表示的进程运行，项 $\mathrm{AmI}\ \nu?\ \mathrm{M}_1 \,\&\, \mathrm{M}_2$ 会规约为 M_1；否则则规约为 M_2。由于 M_1 和 M_2 可能具有不同类型，我们提供了形如 $\mathrm{AmI}\ \nu?\ \tau_1 \,\&\, \tau_2$的类型，它们表示类型 τ_1（对由 ν 所表示的进程上的项进行类型判断时）或类型 τ_2（否则）。这些项构成了 PolyChorλ端点投影的基础：每个绑定进程的 Λ 项都会被转换为包含一个 AmI 项。例如，考虑对以下编排进行投影

$$\Lambda X :: \mathsf{Proc}.\, \mathsf{com}^{\lambda X' :: \mathsf{Proc}.\, \mathsf{Int}@X'}_{\mathsf{Q}, X}\ 4 @ \mathsf{Q}$$

到某个进程 P。根据应用此函数时传入的参数，P 的行为会截然不同：如果将其应用于P 自身，它应当从Q接收某些内容。然而，如果将其应用于任何其他项，它就不应执行任何操作。因此，我们将上述编排投影为以下针对P的程序：

$$\Lambda X.\, \mathsf{AmI}\ X\ ?\, \mathsf{recv}_{\mathsf{Q}}\ \bot\ \&\ \bot$$

请注意，AmI 构造对于一般情况下的进程多态是必不可少的，除非进程变量无法实例化为其所在的进程。它以及多个进程抽象所导致的组合爆炸，并非由编舞语言造成；相反，编舞语言将其隐藏起来，使程序员无需分别显式描述 AmI 的双方。

请注意，我们没有针对局部程序的种类系统。事实上，我们根本不检查局部程序的类型。不过，由于类型具有计算内容，我们也需要对其进行投影。为了保留这些计算内容，我们再次使用一种类型等价，它对应于 β，η-等价。不过，为了适应AmI 类型，我们必须用一个进程为这种等价建立索引。于是，关于AmI 类型，我们有两条规则：

$$[\textsc{IAm}]\ \frac{}{\mathsf{AmI}\ \mathsf{P}\ ?\, \tau_1\ \&\ \tau_2 \equiv_{\mathsf{P}} \tau_1} \qquad [\textsc{IAmNot}]\ \frac{\mathsf{P} \neq \mathsf{Q}}{\mathsf{AmI}\ \mathsf{Q}\ ?\, \tau_1\ \&\ \tau_2 \equiv_{\mathsf{P}} \tau_2}$$

现在我们已经了解了在每个进程上运行的程序语法，可以来看看整个网络：

定义 1。网络 $\mathscr{N}$是从一组进程到局部程序的有限映射。我们通常用 $\mathsf{P}_1[L_1] \mid \cdots \mid \mathsf{P}_n[L_n]$ 表示这样一个网络，其中进程 P_i 的行为是 L_i。

两个网络 $\mathscr{N}$和 $\mathscr{N}'$ 的并行组合（其定义域不相交），$\mathscr{N} \mid \mathscr{N}'$，只需将每个进程的行为指定为定义该进程的网络中的行为。任何网络都等价于定义域为单元素的网络的并行组合，这一点已由上述语法体现。

现在我们来考察局部程序和网络的操作语义。这些语义通过标记迁移系统给出；两类标签的语法见图 5。两种操作语义的选定规则见图 6。完整规则见附录 3。与前文一样，迁移由函数定义集合 d 索引。函数变量通过查找其在 d中的定义来归约。由于该迁移不涉及通信，因此标记为空迁移，τ。

或许令人意外的是，函数的未定义参数并不会立即导致应用未定义。要理解其中原因，可以考虑形如（λx : 整数 @ P. M）的编排，其中某个进程 Q ≠ P 同时参与了 M 和 N。我们将其投影为 Q 上形如（λx : ⊥. $[\![\mathrm{M}]\!]_{\mathrm{Q}}$） $[\![\mathrm{N}]\!]_{\mathrm{Q}}$的应用。注意，由于我们知道 N 的类型为 Int @ P，因此投影 $[\![\mathrm{N}]\!]_{\mathrm{Q}}$ 的类型为 ⊥，最终会求值为 ⊥。因此，如果（λx : ⊥. $[\![\mathrm{M}]\!]_{\mathrm{Q}}$） ⊥ 立即求值为 ⊥，进程 Q 就无法参与 M，而它们必须参与其中！所以，我们允许其求值为 $[\![\mathrm{M}]\!]_{\mathrm{Q}}$。然而，当函数本身也未定义时，我们将其求值为带有空标签 τ的 ⊥，正如规则 [NBot] 和 [NBott]中所示

局部语言：

$$[\textsc{NDef}]\ f \xrightarrow{\tau}_d d(f)$$

$$[\textsc{NBot}]\ \bot\ \bot \xrightarrow{\tau}_d \bot \qquad [\textsc{NBott}]\ \bot\ \bot \xrightarrow{\tau}_d \bot$$

$$[\textsc{NSub}]\ \mathsf{sub}[\mathsf{P} \mapsto \mathsf{Q}]\ V \xrightarrow{\tau}_d V[\mathsf{P} \mapsto \mathsf{Q}]$$

$$[\textsc{NAmIR}]\ \mathsf{AmI}\ \mathsf{P}\ ?\ L_1\ \&\ L_2 \xrightarrow{\mathsf{P}}_d L_1 \qquad [\textsc{NAmIL}]\ \frac{\mathsf{Q} \neq \mathsf{P}}{\mathsf{AmI}\ \mathsf{P}\ ?\ L_1\ \&\ L_2 \xrightarrow{\mathsf{Q}}_d L_2}$$

$$[\textsc{NCho}]\ \oplus_{\mathsf{P}}\ \ell\ L \xrightarrow{\oplus_{\mathsf{P}}\ \ell}_d L \qquad [\textsc{Noff}]\ \&_{\mathsf{P}}\ \{\ell_1 : L_1, \ldots, \ell_n : L_n\} \xrightarrow{\&_{\mathsf{P}}\ \ell_i}_d L_i$$

$$[\textsc{NSend}]\ \mathsf{send}_{\mathsf{P}}\ V_1 \xrightarrow{\mathsf{send}_{\mathsf{P}}\ V_1\ V_2}_d V_2 \qquad [\textsc{NRecv}]\ \mathsf{recv}_{\mathsf{P}}\ V_1 \xrightarrow{\mathsf{recv}_{\mathsf{P}}\ V_2\ V_1}_d V_2$$

网络：

$$[\textsc{NCom}]\ \frac{L_1 \xrightarrow{\mathsf{send}_{\mathsf{P}}\ V\ (V'[\mathsf{Q}\mapsto\mathsf{P}])}_d L_1' \qquad L_2 \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ (V[\mathsf{Q}\mapsto\mathsf{P}])\ V'}_d L_2'}{\mathsf{Q}[L_1] \mid \mathsf{P}[L_2] \xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_d \mathsf{Q}[L_1'] \mid \mathsf{P}[L_2']}$$

$$[\textsc{NSel}]\ \frac{L_1 \xrightarrow{\oplus_{\mathsf{P}}\ \ell}_d L_1' \qquad L_2 \xrightarrow{\&_{\mathsf{Q}}\ \ell}_d L_2'}{\mathsf{Q}[L_1] \mid \mathsf{P}[L_2] \xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_d \mathsf{Q}[L_1'] \mid \mathsf{P}[L_2']}$$

$$[\textsc{NProAm}]\ \frac{L \xrightarrow{\mathsf{P}}_d L'}{\mathsf{P}[L] \xrightarrow{\tau_{\mathsf{P}}}_d \mathsf{P}[L']} \qquad [\textsc{NPro}]\ \frac{L \xrightarrow{\tau}_d L'}{\mathsf{P}[L] \xrightarrow{\tau_{\mathsf{P}}}_d \mathsf{P}[L']}$$

$$[\textsc{NPar}]\ \frac{\mathscr{N}_1 \xrightarrow{\tau_{\mathscr{P}}}_d \mathscr{N}_2}{\mathscr{N}_1 \mid \mathscr{N}' \xrightarrow{\tau_{\mathscr{P}}}_d \mathscr{N}_2 \mid \mathscr{N}'}$$

图 6. 网络语义（选定规则）

如前所述，显式替换 $\mathsf{sub}[\mathsf{P} \mapsto \mathsf{Q}]$ 是函数，应用时会在其所作用的值中执行所请求的替换。这由规则 [NSub]实现。

AmI 项通过规则 [NAmIR] 和 [NAmIL]赋予含义。规则 [NAmIR] 指出，项 $\mathsf{AmI}\ \mathsf{P}\ ?\ \mathrm{L}_1\ \&\ \mathrm{L}_2$ 可以求值为 L_1，其标签为 P；而规则 [NAmIL] 指出，它也可以规约为 L_2，其标签为 Q，其中 $\mathrm{Q} \neq \mathrm{P}$。我们稍后会看到，在网络语义中，我们只允许带有执行迁移的进程所对应标签的迁移。

选择项和提供项按照规则 [NCho] 和 [Noff]进行归约。第一条规则 [Ncho], 说明选择项会直接归约为其延续，并带有一个转换标签，指明所作出的选择。第二条规则 [Noff], 说明提供项可以归约为

任意延续，并带有一个转换标签，指明它所归约到的延续的标签。稍后我们将看到，网络的语义只允许提供项归约为与匹配的选择项所选延续。

最后，发送和接收项通过以下方式赋予含义 [NSend] 和 [NRecv], 。不过，这些规则的行为与预期略有不同：它们并非像普通发送和接收那样运作，而更像是交换信息。

在普通发送中，发送方发送后不会获得任何信息——或许该项会带有一个延续，但它与发送操作无关。此外，接收方不会提供任何信息，而只是接收发送方的信息。不过，在发送带有进程多态的编排时，发送方可能需要参与延续，具体取决于多态函数的应用方式。例如，考虑以下编排，其中P 向Q发送一个多态函数，而生成的多态函数应用于P：

$$(\mathsf{com}_{P,Q}^{\lambda Y::\mathsf{Proc}.\,\forall X::\mathsf{Proc}.\,\mathsf{Int}@X}\ (\Lambda X :: \mathsf{Proc}.\,\mathsf{com}_{P,X}^{\lambda Y'::\mathsf{Proc}.\,\mathsf{Int}@Y'}\ (5 \mathbin{@} P)))\ P$$

上述通信所产生的多态函数如下：

$$\Lambda X :: \mathsf{Proc}.\ \left(\mathsf{com}_{Q,X}^{\lambda Y'::\mathsf{Proc}.\,\mathsf{Int}@Y'}\ (5 \mathbin{@} Q)\right)$$

将其应用于P 会得到一个程序，其中P 从Q接收。由于P 需要参与该程序，P 在将多态函数发送给Q之后，必须仍有一个程序可继续执行。

虽然这解释了为何 send 项不能简单地例如返回单位，但这并未解释为何 send 和 recv 项会 交换 结果。为此，考虑将一个项从进程 P 发送给另一个进程 Q 时会发生什么。根据我们的类型系统，我们知道Q 未在被发送项的类型中提及，并且我们知道，发送后所有对 P 的提及都会变为对 Q 的提及。因此，发送后，P 所持有的该项版本应当是未参与该项的进程所看到的视图。这恰好就是发送Q 所持有的该项版本在 发送之前的状态。因此，send 和 recv 的交换式行为会产生正确的行为。

既然我们已经讨论了局部程序的语义，接下来讨论网络语义。网络语义中的每个迁移都带有一个由参与该归约的进程索引的静默标签：$\tau_{\mathscr{P}}$，其中 $\mathscr{P}$ 由一个进程名称（该进程上的局部动作）或两个进程名称（涉及这两个进程的交互）构成。我们将 $\mathscr{P}$ 视为一个集合，从而隐式允许交换。

例如，规则 [通信] 描述了通信。在这里，一个局部项必须与一个 发送 标签归约，而另一个则与一个 接收 标签归约。这些标签必须匹配，也就是说，接收 方接收的值必须是发送方发送的值——不过接收方和发送方的位置互换，反之亦然。于是，局部项与相应进程关联的网络 Q 和 P可以通过标签 $\tau_{Q,\ P}$归约。类似地，规则 [选择] 会归约相匹配的选择项和选择项，产生标签 τ_{Q}

$_{,P}$.

虽然 [通信] 和 [选择] 描述了通信，但其余规则描述了网络中单个进程的项如何随时间演变。尤其有趣的是[NProam], ，它指出，一个 AmI 项只能按照与之关联的进程进行归约

。这里可以看到，生成的标签是 τ_{P}，这表示此归约步骤仅涉及P。

规则 [NPro] 告诉我们如何提升标签为空的步骤 τ。这类步骤不对网络作出任何假设，因此这类项可以与任意进程 P关联。当网络中发生这样的规约时，我们将产生的迁移标记为 τ_{P}。

最后，规则 [NPar] 说明，如果网络的一部分可以通过标签 τ进行规约，那么整个网络也可以通过同一标签进行规约。这使得其他假设网络规模最小的规则能够应用于更大的网络。今后，我们将使用 $\rightarrow^*$和 $\rightarrow^+$ 分别表示任意标签的动作序列和至少包含一个动作的序列。

4.1 投影

现在我们可以定义编排的端点投影（EPP）。它描述单个进程对编排的视角；编排的并发解释则通过将投影并行组合到每个进程上来给出。对特定进程 P 的端点投影定义为关于类型推导 $\Theta;\Gamma \vdash M : \tau$的递归函数。不过，为了便于阅读，我们将其写成关于项 M的递归函数，并使用记号 $\mathsf{typeof}(N)$ 指代隐式类型推导中赋予任意项 N 的类型。类似地，我们使用 $\mathsf{kindof}(\tau)$ 指代隐式类型推导中类型 τ 的种类。我们用 $[\![M]\!]_{\mathsf{P}}$ 表示项 M （隐式地附有关于 M的类型推导，证明其具有某种类型）到进程 P的投影。

直观而言，投影会将编排项转换为对应的本地行为。例如，通信动作会投影为发送（对于发送方）、接收（对于接收方）、替换（对于所通信值类型中的其他进程），或者空进程（对于其余进程）。然而，case 语句的情况则更为复杂。例如，考虑以下编排：它对一个和类型进行匹配，该类型要么是在 Alice 上的整数，要么是在 Alice上的空值。如果是整数，那么 Bob 会从 Alice 接收该整数并将其返回。否则，Bob会返回默认值 42。 Alice 会告知 Bob 她选择了哪个分支，方法是使用选择 项。

$$\left(\begin{array}{l}\lambda\, z : (\mathsf{Int} \mathbin{@} \mathsf{Alice}) + (() \mathbin{@} \mathsf{Alice}).\\ \quad \mathsf{case}\ z\ \mathsf{of}\\ \quad\quad \mathsf{inl}\ \ x \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Just}\ (\mathsf{com}^{\lambda X.\, \mathsf{Int}@X}_{\mathsf{Alice},\mathsf{Bob}}\ x);\\ \quad\quad \mathsf{inr}\ \ y \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Nothing}\ (42 \mathbin{@} \mathsf{Bob})\end{array}\right)\ \mathsf{inl}_{()@\mathsf{Alice}}\ (3 \mathbin{@} \mathsf{Alice})$$

设想从 Bob 的视角来投影这段程序。他没有和类型中的任何信息，因此无法参与选择要执行 case 表达式的哪个分支。相反，他必须等待 Alice 告诉他自己处于哪个分支。如果我们天真地只翻译 case 表达式的第一个分支，Bob 就会等待 Alice 向他发送标签 Just ，然后再等待 Alice 向他发送一个整数。类似地，在第二个分支中，Bob 会等待 Alice 向他发送标签 Nothing ，之后返回默认值 42。我们需要以某种方式将这些操作合并起来，使得 Bob 等待任一标签，然后执行相应的操作。

我们通过合并Bob在各个分支中的局部程序来实现这一点（Carbone 等人，2012年；Cruz-Filipe 和 Montesi，2020年；Honda 等人，2016）。合并是一种偏函数，它组合两个兼容的局部程序，并合并选择语句。换句话说，合并的关键性质是：

$$\&_{\mathsf{P}}\ \{\ell_i : B_i\}_{i\in I} \sqcup \&_{\mathsf{P}}\ \{\ell_j : B'_j\}_{j\in J} = \&_{\mathsf{P}}\ \left(\{\ell_k : B_k \sqcup B'_k\}_{k\in I\cap J} \cup \{\ell_i : B_i\}_{i\in I\setminus J} \cup \{\ell_j : B'_j\}_{j\in J\setminus I}\right)$$

合并在其他项上以同态方式定义，但对于不兼容的项则未定义。因此，例如，左注入$_\tau$ $M \sqcup$左注入$_\tau$ $N =$左注入$_\tau$ $(M \sqcup N)$是有定义的，但左注入$_{\tau_1}$ $M \sqcup$右注入$_{\tau_2}$ N则未定义。

然后，我们可以使用这一点，将上述编排投影到 Bob，得到：

$$(\lambda\, z : \bot.\&_{\mathsf{Alice}}\ \{\mathsf{Just} : (\mathsf{recv}_{\mathsf{Alice}}\ \bot, \mathsf{Nothing} : 42\})\ \bot$$

其中，$\bot$ 表示由 Alice 执行的编排的一部分。

定义 2。 编排的端点投影M 对于进程 P 由图7、8 和9中的规则定义。

因此，要从编排投影出一个网络，我们分别对每个进程的编排进行投影，并将结果并行组合：$[\![M]\!] = \prod_{\mathsf{P}\in \mathsf{ip}(M)} \mathsf{P}\ [[\![M]\!]_{\mathsf{P}}]$。

直观地说，将一个编排投影到未参与其中的进程，会返回一个$\bot$。然而，更复杂的编排可能涉及其类型中未显示的进程。这解释了投影应用时的第一个子句：即使 P 没有出现在M的类型中，它也可能参与M内部的交互。类似的观察也适用于 case 的投影，其中同样使用了合并。

选择和通信遵循上文给出的直觉，不过有一个有趣的细节：自选择会被忽略，而自通信会投影为身份函数。这与许多其他编排演算不同，在那些演算中不允许自通信——我们不希望在 PolyChorλ 中施加这一限制，因为我们采用了进程多态，因此不希望不必要地限制编排可以用哪些进程实例化。

对于进程抽象 ΛX :: Proc. M，任何进程 P 都必须准备两种行为：一种用于 X 以 P 自身实例化时，另一种用于 X 以其他进程实例化时。为此，我们使用 AmI 项，使 P 能够利用自己对其身份的了解来选择执行哪种行为。（当 X 的种类受到限制时，这一点同样成立，只要其基础种类是 Proc；不过，如果 P 被排除在 X 的类型之外，且 P 不参与 M，那么我们只需投影为 $\bot$。）然而，若 K 不是进程种类，则类型抽象 ΛX :: K. M 不使用 AmI 项，因为 P 无法实例化 X。

一般而言，投影一个类型会在该类型未使用的任何进程处得到 $\bot$。我们利用种类上的限制，在确定不需要时避免投影类型变量和类型抽象，并将所有进程名称投影为其自身；除此之外，类型构造的投影与相应进程项的投影类似。

最后，为了执行投影后的编排，我们需要将编排函数的定义集投影为局部函数的定义集。由于这些函数都以每个参与进程为参数，因此只需投影这些定义即可

$$[\![x]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\mathsf{typeof}(x)]\!]_{\mathsf{P}} = \bot \\ x & \text{otherwise} \end{cases} \qquad [\![f]\!]_{\mathsf{P}} = f$$

$$[\![() \mathbin{@} \nu]\!]_{\mathsf{P}} = \begin{cases} () & \text{if } [\![\nu]\!]_{\mathsf{P}} = \mathsf{P} \\ \bot & \text{otherwise} \end{cases} \qquad [\![n \mathbin{@} \nu]\!]_{\mathsf{P}} = \begin{cases} n & \text{if } [\![\nu]\!]_{\mathsf{P}} = \mathsf{P} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\lambda\, x : \tau.\, M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ & \text{and } [\![\tau]\!]_{\mathsf{P}} = \bot \\ \lambda\, x : [\![\tau]\!]_{\mathsf{P}}.\ [\![M]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![M\ N]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = [\![N]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}}\ [\![N]\!]_{\mathsf{P}} & \text{if } \mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(M)) \\ & \quad \text{or } [\![M]\!]_{\mathsf{P}} \neq \bot \neq [\![N]\!]_{\mathsf{P}} \\ [\![M]\!]_{\mathsf{P}} & \text{if } [\![N]\!]_{\mathsf{P}} = \bot \\ [\![N]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\Lambda X :: \mathsf{K}.\, M]\!]_{\mathsf{P}} = \begin{cases} \Lambda X.\, \mathsf{Aml}\, X & \text{if } \mathsf{K} \in \{\mathsf{Proc},\ \mathsf{Proc} \setminus \rho\} \\ \qquad ?\, [\![M[X \mapsto \mathsf{P}]]\!]_{\mathsf{P}} & \\ \qquad \&\ \ [\![M]\!]_{\mathsf{P}} & \\ \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ & \quad \text{and } \mathsf{K} = \mathsf{K}' \setminus \{\mathsf{P}\} \cup \rho \\ \Lambda X.\ [\![M]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![M\ \tau]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = [\![\tau]\!]_{\mathsf{P}} = \bot \\ & \text{if } [\![\tau]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}} & \quad \text{and } \mathsf{kindof}(\tau) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho) \\ [\![\tau]\!]_{\mathsf{P}} & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}}\ [\![\tau]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{inl}_{\tau}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \text{ and} \\ & \quad \mathsf{kindof}(\tau) = \mathsf{K} \setminus \rho \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{inl}_{[\![\tau]\!]_{\mathsf{P}}}\ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{inr}_{\tau}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \text{ and} \\ & \quad \mathsf{kindof}(\tau) = \mathsf{K} \setminus \rho \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{inr}_{[\![\tau]\!]_{\mathsf{P}}}\ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

图7. PolyChor 程序的投影　λ

$$[\![\mathsf{case}\ M\ \mathsf{of\ inl}\ x \Rightarrow N_1;\ \mathsf{inr}\ y \Rightarrow N_2]\!]_{\mathsf{P}} =$$
$$\begin{cases} \mathsf{case}\ [\![M]\!]_{\mathsf{P}}\ \mathsf{of\ inl}\ x \Rightarrow [\![N_1]\!]_{\mathsf{P}};\ \mathsf{inr}\ y \Rightarrow [\![N_2]\!]_{\mathsf{P}} & \text{if } P \in \mathsf{ip}(\mathsf{typeof}(M)) \\ \bot & \text{if } [\![M]\!]_{\mathsf{P}} = [\![N_1]\!]_{\mathsf{P}} = [\![N_2]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}} & \text{if } [\![N_1]\!]_{\mathsf{P}} = [\![N_2]\!]_{\mathsf{P}} = \bot \\ [\![N_1]\!]_{\mathsf{P}} \sqcup [\![N_2]\!]_{\mathsf{P}} & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ (\lambda\, z : \bot.\ ([\![N_1]\!]_{\mathsf{P}} \sqcup [\![N_2]\!]_{\mathsf{P}}))\ [\![M]\!]_{\mathsf{P}}\ \textit{(z fresh)} & \text{otherwise} \end{cases}$$

$$[\![(M_1, M_2)]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M_1]\!]_{\mathsf{P}} = [\![M_2]\!]_{\mathsf{P}} = \bot \\ ([\![M_1]\!]_{\mathsf{P}}, [\![M_2]\!]_{\mathsf{P}}) & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fst}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{fst}\ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{snd}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{snd}\ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{select}_{\mathsf{Q}_1,\mathsf{Q}_2}\ \ell\ M]\!]_{\mathsf{P}} =$$
$$\begin{cases} \oplus_{\mathsf{Q}'}\ \ell\ [\![M]\!]_{\mathsf{P}} & \text{if } \mathsf{P} = \mathsf{Q}_1 \neq \mathsf{Q}_2 \\ \&_S\ \{\ell : [\![M]\!]_{\mathsf{P}}\} & \text{if } \mathsf{P} = \mathsf{Q}_2 \neq \mathsf{Q}_1 \\ [\![M]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$\left[\!\left[\mathsf{com}^{\tau}_{\mathsf{Q}_1,\mathsf{Q}_2}\right]\!\right]_{\mathsf{P}} =$$
$$\begin{cases} \lambda\, x : \tau\ \mathsf{P}.\, x & \text{if } \mathsf{P} = \mathsf{Q}_1 = \mathsf{Q}_2 \\ \mathsf{send}_{\mathsf{Q}_2} & \text{if } \mathsf{P} = \mathsf{Q}_1 \neq \mathsf{Q}_2 \\ \mathsf{recv}_{\mathsf{Q}_1} & \text{if } \mathsf{P} = \mathsf{Q}_2 \neq \mathsf{Q}_1 \\ \mathsf{sub}[\mathsf{Q}_1 \mapsto \mathsf{Q}_2] & \text{if } [\![\tau]\!]_{\mathsf{P}} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

图 8. PolyChorλ程序的投影（续）

到任意选定的进程名称上。

$$[\![D]\!] = \{f \mapsto [\![D(f)]\!]_{\mathsf{P}} \mid f \in \mathsf{domain}(D)\}\}$$

请注意，函数名始终会被投影到所有地方。这意味着，如果某个函数应用于任一进程中的某个值时不会终止，那么将其应用于编排中的该值以及每个其他进程中的该值时，也都会发散。

示例 2。现在我们将展示如何投影第 1 节中的书商服务示例。与该示例一样，我们将 let x $=$ M in M′ 用作语法糖，表示 λx : τ. M′ M（其中 τ为某个值）；并将 if $\mathrm{M_1}$ then $\mathrm{M_2}$ else $\mathrm{M_3}$ 用作语法糖，表示 case $\mathrm{M_1}$ of inl x $\Rightarrow$ $\mathrm{M_2}$; inr x $\Rightarrow$$\mathrm{M_3}$（其中 x $\notin$(fv($\mathrm{M_2}$) $\cup$fv($\mathrm{M_3}$))）。我们针对卖方进行投影，得到以下进程：

Λ B. AmI B ? λ书名.λ 按价格购买?.
let x = (λy. y) 书名 in let y = (λz. z)
(价格查询 x) in if 按价格购买? y
then () else () & λ书名.λ 按价格购买?.
let x = 接收 $_B$ ⊥in let y = 发送 $_B$ (价
格查询 x) in &$_B$ {Buy : (), Quit : ()}

这里可以看到，如果买家 B 最终就是卖方本身，那么所有通信都会变成恒等函数，卖方也不会将自己的选择告知自己。否则，我们会得到一个函数；该函数在用一个买家实例化后，会接收两个 ⊥，表示 B 中已存在的值。随后，它会等待 B 发送一个书名，返回该书名对应的价格，并等待 B 决定是否购买。函数以两个 ⊥ 为参数看起来或许有些奇怪，但这个例子实际上说明了为什么我们不一定能从进程中移除最前面的两个 λ，否则会导致死锁。注意，let y =send$_B$ (price lookup x)in &$_B$ {Buy : (), Quit : ()} 是 (λy. &$_B$ {Buy : (), Quit : ()}) (send$_B$ (price lookup x)) 的语法糖。这里需要在 y 上保留抽象，即使它是在 ⊥ 后才被实例化的；此时卖方已将 price lookup x 的结果发送给 B。反之，如果我们只有 (&$_B$ {Buy : (), Quit : ()}) (send$_B$ (price lookup x))，那么应用的第一部分就不是一个值，并会等待 B 在 Buy 和 Quit 之间作出选择；而 B 在 y 上有一个抽象，因此会将应用的第一部分视为一个必须等待实例化的函数。于是，B 预期会接收 price lookup x 的结果，系统便会陷入死锁。这就是为什么我们绝不希望将一个值投影到非值项上，并且需要保留所有用于守护编排中涉及卖方部分的抽象。

5 端点投影的正确性

现在我们证明，编排及其投影的执行之间存在密切对应关系。直观地说，这种对应关系表明，当且仅当编排能够执行某个动作时，其投影也能够执行相同的动作，并且二者都迁移到处于相同关系中的新项。然而，这并不完全正确：如果编排M 根据规则 [CaseL], 进行规约，那么结果中的分支数会少于通过在C的投影中执行相应规约而得到的网络中的分支数。

为了刻画这一点，我们重新采用分支关系（Montesi，2023年；Cruz-Filipe 和 Montesi，2020年），其定义为 $M \sqsupseteq N$ 当且仅当 $M \sqcup N = M$。直观地说，如果 $M \sqsupseteq N$，那么 M 提供

$$[\![X]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } \mathsf{kindof}(X) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho) \text{ for } \mathsf{K} \neq \mathsf{Proc} \\ X & \text{otherwise} \end{cases} \qquad [\![\mathsf{Q}]\!]_{\mathsf{P}} = \mathsf{Q}$$

$$[\![() \ @ \ \mathsf{Q}]\!]_{\mathsf{P}} = \begin{cases} () & \text{if } \mathsf{P} = \mathsf{Q} \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathsf{Int} \ @ \ \mathsf{Q}]\!]_{\mathsf{P}} = \begin{cases} \mathsf{Int} & \text{if } \mathsf{P} = \mathsf{Q} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\tau_1 \times \tau_2]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} \times [\![\tau_2]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\tau_1 + \tau_2]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} + [\![\tau_2]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\tau_1 \to_{\rho} \tau_2]\!]_{\mathsf{P}} = \begin{cases} [\![\tau_1]\!]_{\mathsf{P}} \to [\![\tau_2]\!]_{\mathsf{P}} & \text{if } \mathsf{P} \in \rho \text{ or } [\![\tau_1]\!]_{\mathsf{P}} \neq \bot \neq [\![\tau_2]\!]_{\mathsf{P}} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\forall X :: \mathsf{K}. \, \tau]\!]_{\mathsf{P}} =$$

$$\begin{cases} \bot & \text{if } [\![\tau]\!]_{\mathsf{P}} = \bot \text{ and } \mathsf{K} = \mathsf{K}' \setminus (\{\mathsf{P}\} \cup \rho) \\ \forall X. \, \mathsf{Aml} \ X \ ? \ [\![\tau[X \mapsto \mathsf{P}]]\!]_{\mathsf{P}} \ \& \ [\![\tau]\!]_{\mathsf{P}} & \text{if } \mathsf{K} \in \{\mathsf{Proc}, \ \mathsf{Proc} \setminus \rho\} \\ \forall X. \, [\![\tau]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\tau_1 \ \tau_2]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} & \text{if } [\![\tau_2]\!]_{\mathsf{P}} = \bot \text{ and } \mathsf{kindof}(\tau_2) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho) \\ [\![\tau_2]\!]_{\mathsf{P}} & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} \ [\![\tau_2]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\lambda X :: \mathsf{K}. \, \tau]\!]_{\mathsf{P}} =$$

$$\begin{cases} \bot & \text{if } [\![\tau]\!]_{\mathsf{P}} = \bot \text{ and } \mathsf{K} = \mathsf{K}' \setminus (\{\mathsf{P}\} \cup \rho) \\ \lambda X. \, \mathsf{Aml} \ X \ ? \ [\![\tau[X \mapsto \mathsf{P}]]\!]_{\mathsf{P}} \ \& \ [\![\tau]\!]_{\mathsf{P}} & \text{if } \mathsf{K} \in \{\mathsf{Proc}, \ \mathsf{Proc} \setminus \rho\} \\ \lambda X. \, [\![\tau]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

图 9. PolyChorλ类型的投影

与 N相同甚至更多的行为。这一概念可通过如下定义扩展到网络：$\mathscr{N} \sqsupseteq \mathscr{N}'$ 这意味着，对于任意角色 P，$\mathscr{N}(\mathsf{P}) \sqsupseteq \mathscr{N}'(\mathsf{P})$。

借助这一点，我们可以证明，编排的端点投影能够完成编排所做的一切（完备性），并且只会做原编排所做的事情（可靠性）。对于传统的命令式编舞语言，这种对应关系表现为编排中的一个动作对应投影网络中的一个动作。而我们允许编排在一个动作中操作分布式值，例如 $\lambda\, x{:}\ \text{整数} \ @ \ \mathsf{Bob} \times \text{整数} \ @ \ \mathsf{Alice}。\ M\,(3 \ @ \ \mathsf{Bob}, \ 3 \ @ \ \mathsf{Alice})$ 其中 Bob 和 Alice 各自独立地取出这对值的第一部分。

定理 5 (完备性)。 给定一个封闭编排 M，若 $M \to_D M'$， $\Theta;\ \Gamma \vdash D,\ \Theta;\ \Gamma \vdash M : \tau$，且 $[\![M]\!]$ 有定义，则存在网络 $\mathscr{N}$ 和 M''，使得：$[\![M]\!] \to^+_{[\![}\ D]\!]\ \mathscr{N}$ 且 $\mathscr{N} \sqsupseteq [\![M']\!]$。

在前文中，我们用 L 表示局部表达式，用 U 表示局部值，以便让证明更易读，因为我们将在不同层次之间频繁切换。

在证明完备性之前，我们需要一些引理。首先，我们证明编排值总是投影为局部值。

引理 3。 对于任意编排值 V 和进程 P，若 $\Theta;\ \Gamma \vdash V : \tau$ 则 $[\![V]\!]_P$ 要么是一个值，要么未定义。

证明：由投影规则可直接得出。 ■

接下来，我们证明类型值也满足相同性质。

引理 4。 给定一个类型值 ν_1，若 $\Theta;\ \Gamma \vdash \nu_1 :: K$ 则对于任意进程 P 属于 $ip(\nu_1)$，$[\![\nu_2]\!]_P = \nu_2$。

证明：由投影规则直接可得。 ■

接着我们证明，类型值在未涉及的进程处被投影为 $\bot$。

引理 5。 给定一个类型值 $\nu \neq P$，对于任意进程 $Q \notin ip(\nu)$， $[\![\nu]\!]_Q = \bot$。

证明：对 ν 进行归纳法即可直接得出。 ■

类似地，我们证明编舞值在未涉及其类型的进程处投影为 $\bot$。

引理 6。 给定一个值 V，若 $\Theta; \Gamma \vdash V : \tau$，则对于任意进程 $P \notin ip(\tau)$，我们有 $[\![V]\!]_P = \bot$，或者 $[\![V]\!]_P$ 未定义。

证明 由引理 3 和 5 以及投影规则可得。 ■

最后，我们证明等价类型会被投影为等价的局部类型。

引理 7. 给定一个闭类型 τ，若 $\tau \equiv \tau'$ 且 $\Theta;\ \Gamma \vdash \tau :: K$，则对于任意进程 P，$[\![\tau]\!]_P \equiv_P [\![\tau']\!]_P$。

证明 我们对 $\tau \equiv \tau'$ 进行结构归纳来证明这一点。除一种情况外，其余情况均可通过简单归纳法得出。

唯一有趣的情况是，如果 $T = \lambda X.\ K\tau_1\ \nu$ 且 $\tau' = \tau_1[X := \nu]$。那么：(1) 如果 $K = K' \setminus (\{R\} \cup \rho)$ 且 $[\![T[1]]\!]_P = \bot$，则有 $[\![T]\!]_P = [\![T']\!]_P = \bot$。(2) 如果 $K \in \{\text{Proc},\ \text{Proc} \setminus \rho\}$ 那么 $[\![T]\!]_P = (\forall X.\ \text{AmI}\ X\ ?\ [\![\tau_1[X := P]]\!]_P\ \&\ [\![\tau_1]\!]_P)\ [\![\nu]\!]_P$ 且 $[\![\tau']\!]_P = [\![\tau_1[X := \nu]]\!]_P$。由于 T 是一个闭类型且 $\Theta\Gamma \vdash \nu :: K$，$\nu$ 必定是一个进程。如果 $\nu = P$，那么显然 $(\forall X.\ \text{AmI}\ X\ ?\ [\![\tau_1[X := P]]\!]_P\ \&\ [\![\tau_1]\!]_P)\ [\![\nu]\!]_P \equiv_P [\![\tau_1[X := \nu]]\!]_P$。如果 $\nu \neq P$，那么 $(\forall X.\ \text{AmI}\ X\ ?$

$[\![\tau_1[X:=P]]\!]_P$ & $[\![\tau_1]\!]_P$) $[\![v]\!]_P \equiv_P [\![\tau_1]\!]_P[X:=[\![v]\!]_P]$，但由于 v 是一个进程 Q，$[\![v]\!]_P = Q$ 且 $[\![X]\!]_P = X$，因此得到 $[\![\tau]\!]_P \equiv_P [\![\tau']\!]_P$。而 (3) 否则我们有$[\![\tau]\!]_= \lambda X.\ [\![\tau_1]\!]_P\ [\![v]\!]_P$ 且 $[\![\tau']\!]_P = [\![\tau_1]\!]_P[X:=[\![v]\!]_P]$。因此得到 $[\![\tau]\!]_P \equiv_P [\![\tau']\!]_P$。 ■

现在我们可以证明完备性了。

证明 [定理 5] 我们通过对 $M \to_D M'$进行结构归纳来证明。

- 假设 $M = \lambda x:\tau.N\ V$ 且 $M' = N[x:=V]$。那么对于任意满足 $[\![N]\!]P \neq \bot$或 $[\![\tau]\!]P \neq \bot$ 的进程 P，我们有 $[\![M]\!]P = (\lambda x:[\![\tau]\!]P \cdot [\![N]\!]P)\ [\![V]\!]P$，且$[\![M']\!]P = [\![N]\!]P[x:=[\![V]\!]P]$,。对于任意其他的 P′，我们有 $P' \notin ip(typeof(V))$，因此 $[\![M]\!]P' = [\![M']\!]P' = \bot$。由此，对于所有 $P \in ip(typeof(\lambda x:\tau.N)$，我们得到 $P[[\![M]\!]P] \xrightarrow{\tau}_{[\![D]\!]} [\![M']\!]P$，并定义 $\mathcal{N} = \prod_{P \in ip(typeof(\lambda x:\tau.N))} P[[\![M']\!]P] \mid \prod_{P' \notin 进程标识集合(类型((\lambda x:\tau.N))} P'[\bot]$，结论随之成立。
- 假设 $M = \Lambda X :: K.\ N\ \tau$, $\tau \equiv v$，且 $M' = N[X:=v]$。那么，若 $K \in \{P, P \setminus \rho\}$，则对任意进程 P，$[\![M]\!]P = \Lambda X :: K.\ AmI\ X\ ?\ [\![N[t:=P]]\!]P$ & $[\![N]\!]P\ \ [\![\tau]\!]P$，结论由引理 4 和 7 以及规则 [NBabs], [NIamr], 和 [NIaml]得出。若 $K \notin \{Proc, Proc \setminus \rho\}$，那么对于任意满足 $[\![N]\!]P = \bot$ 且 $K = K' \setminus (\{P\} \cup \rho)$ 的进程 P，我们有 $[\![M]\!]P = [\![M']\!]P = \bot$；对于任意其他 P′，我们有 $[\![M]\!]P' = (\Lambda X.\ [\![N]\!]P')\ [\![\tau]\!]P'$，且 $[\![M']\!]P' = [\![N]\!]P'[X:=[\![v]\!]P']$。因此，对所有 P，我们得到 $P[[\![M]\!]P] \to^*_{[\![D]\!]} M\ P$，结论随之成立。

- 假设 $M = N\ M''$，$M' = N'\ M''$，且 $N \to_D N'$。那么，对于任意满足 $[\![N]\!]P = [\![M'']\!]P = \bot$ 的进程 P，根据归纳法可得 $[\![N']\!]P = \bot$，因此 $[\![M]\!]P = [\![M']\!]P = \bot$。对于任意满足 P′ 且 $P' \in ip(typeof(N))$ 或 $[\![N]\!]P' \neq \bot \neq [\![M'']\!]P'$ 的进程 P，$[\![M]\!]P' = [\![N]\!]P'\ [\![M'']\!]P'$ 且 $[\![M']\!]P' = [\![N']\!]P'\ [\![M'']\!]P'$。对于其他任意满足 $[\![N]\!]P'' = \bot$ 的进程 P″，根据归纳法可得 $[\![N']\!]P'' = \bot$，因此$[\![M]\!]P'' = [\![M']\!]P'' = [\![M'']\!]P''$。对于其他任意满足 $[\![M'']\!]P''' = \bot$ 的进程 P‴，可得 $[\![M]\!]P''' = [\![N]\!]P'''$ 且 $[\![M']\!]P''' = [\![N']\!]P'''$。并且根据归纳法，$[\![N]\!] \to^*_{[\![D]\!]} \mathcal{N}\ N\ \mathcal{N}\ N \sqsupseteq [\![N']\!]$。因此，对于任意进程 P，都有 $[\![N]\!]P \xrightarrow{\mu_0}_{[\![D]\!]} \xrightarrow{\mu_1}_{[\![D]\!]} \cdots LP$，其中 $LP \sqsupseteq [\![N']\!]P$，且存在若干迁移序列 $\xrightarrow{\mu_0}_{[\![D]\!]} \xrightarrow{\mu_1}_{[\![D]\!]} \cdots$；根据网络语义，我们得到

$$[\![M]\!] \to^* \prod_{[\![N]\!]_P = [\![M'']\!]_P = \bot} P[\bot] \mid \prod_{P' \in ip(typeof(N)) \text{ or } [\![N]\!]_{P'} \neq \bot \neq [\![M'']\!]_{P'}} P'[L_{P'}\ [\![M'']\!]_{P'}] \mid \prod_{[\![M]\!]_{P''} = [\![M'']\!]_{P''}} P''[[\![M'']\!]_{P''}] \mid \prod_{[\![M]\!]_{P'''} = [\![N]\!]_{P'''}} P''[L_{P''}] = \mathcal{N}$$

并且$M' \to_D N'\ M''$。由于 $[\![N]\!] \to^*_{[\![D]\!]} \mathcal{N}'$并且 $[\![N']\!] \to^*_{[\![D]\!]} \mathcal{N}'_N$，我们知道这些迁移序列可以在必要时同步；如果 $[\![N]\!]_{P''''} \neq [\![N']\!]_{P''''} = \bot$ 那么我们就可以执行额外的应用来消去这个单位。

- 假设$M = V\ N$，$M' = V\ N'$，并且$N \to_D N'$。这与前面的 case 类似，使用引理3 来确保每个进程都可以使用规则 [NApp2]。
- 假设 M =case N 的结果为 inl x ⇒ N′; inr x′ ⇒ N″，$M' = $ case M″ 的结果为 inl x ⇒ N′; inr x ⇒N″，且 $N \to_D M''$。那么对于任意满足 $P \in ip(typeof(N))$ 的进程 P，我们有 $[\![M]\!]P =$ case $[\![N]\!]P$ 的结果为 inl x ⇒ $[\![N']\!]P$; inr x′ ⇒ $[\![N'']\!]P$，且$[\![M']\!]P =$ case $[\![M'']\!]P$ 的结果为 inl x ⇒ $[\![N']\!]P$; inr x′ ⇒ $[\![N'']\!]P$。对于其他任意进程 P′

满足 $[\![N]\!]_{P'} = [\![N']\!]_{P'} = [\![N'']\!]_{P'} = \bot$，由归纳法可得 $[\![M'']\!]_{P'} = \bot$，因此 $[\![M]\!]_{P'} = [\![M']\!]_{P'} = \bot$。对于其他任意满足 $[\![N]\!]_{P''} = \bot$ 的进程 P''，我们得到$[\![M]\!]_{P''} = [\![M']\!]_{P''} = [\![N']\!]_{P''} \sqcup [\![N'']\!]_{P''}$。对于其他任意满足 $[\![N']\!]_{P'''} = [\![N'']\!]_{P'''} = \bot$ 的进程 P'''，我们有 $[\![M]\!]_{P'''} = [\![N]\!]_{P'''}$ 且 $[\![M']\!]_{P'''} = [\![M'']\!]_{P'''}$。对于其他任意进程 P''''，我们有 $[\![M]\!]_{P''''} = (\lambda x : \bot.[\![N']\!]_{P''''} \sqcup [\![N'']\!]_{P''''})\ [\![N]\!]_{P''''}$ 且 $[\![M']\!]_{P''''} = (\lambda x.[\![N']\!]_{P''''} \sqcup [\![N'']\!]_{P''''})\ [\![M'']\!]_{P''''}$，其中 $x \notin fv(N') \cup fv(N'')$。其余部分可通过与第二种情况类似的简单归纳法得出。

- 假设 $M = \text{case 左注入}_\tau\ V\ \text{of 左注入}\ x \Rightarrow N;\ \text{右注入}\ x' \Rightarrow N'$，且 $M' = N[x := V]$。那么对于任意进程 $P \in ip(typeof(\text{左注入}_\tau\ V))$，我们有 $[\![M]\!]P = \text{case 左注入}_{[\![\tau]\!]}[\![V]\!]P\ \text{of 左注入}\ x \Rightarrow [\![N]\!]P;\ \text{右注入}\ x' \Rightarrow [\![N']\!]P$，且 $[\![M']\!]P = [\![N[x := [\![V]\!]P]]\!]P$。根据引理 6，$[\![N[x := [\![V]\!]P]]\!]P = [\![N]\!]P[x := [\![V]\!]P]$。对于任意其他进程 $P' \notin ip(typeof(\text{左注入}_\tau\ V))$，$[\![\text{左注入}\ V\ \tau P']\!] = \bot$，因此 $[\![M]\!]P' = [\![N]\!]P' \sqcup [\![N']\!]P' \sqsupseteq [\![N]\!]P' = [\![M']\!]P'$。结论成立。

- 假设 $M = \text{case 右注入}_\tau\ V \text{of 左注入}\ x \Rightarrow N;\ \text{右注入}\ x' \Rightarrow N'$ 且 $M' = N'[x' := V]$。此情形与前一种情形类似。
- 假设 $M = \text{通信} P_\tau, Q\ V$ 且 $M' = V[Q := P]$。那么，若 $Q \neq P$，则 $[\![M]\!]P = \text{接收} Q\ [\![V]\!]P, [\![M']\!]P = [\![V\ [Q := P]]\!]P = [\![V]\!]Q[Q := P]$，$[\![M]\!]Q = \text{发送} P\ [\![V]\!]Q$，$[\![M']\!]Q = [\![V[Q := P]]\!]Q = [\![V]\!]P[Q := P],;$ 并且对于任意 P' 使得 $[\![\tau]\!]P' \neq \bot$，我们有 $[\![M]\!]P' = sub[Q \mapsto P]\ [\![V]\!]P'$，且 $[\![M']\!]P' = [\![V[Q := P]]\!]P' = [\![V]\!]P'[Q := P],;$ 对于任意其他 P''，则有 $[\![M]\!]P'' = [\![M']\!]P'' = \bot$。因此我们得到 $[\![M]\!]P \xrightarrow{\text{接收}Q\ [\![V]\!]Q[Q:=P]} [\![V]\!]P \longrightarrow_{[\![D]\!]} [\![M']\!]P, [\![M]\!]Q \xrightarrow{\text{发送}P\ [\![V]\!]P\ Q\ V\ P\ Q:\ P\ D} [\![M']\!]Q$，以及 $[\![M]\!]P' \xrightarrow{\tau}_{[\![D]\!]} [\![M']\!]P'$。我们定义 $\mathcal{N} = \mathcal{N}' = [\![M']\!]$，结论由此得出。若 $Q = P$，则 $[\![M]\!]P = (\lambda x.x)\ [\![V]\!]P$ 且 $[\![M']\!]P = [\![V]\!]P$，结论由此得出 $\mathcal{N} = \mathcal{N}' = [\![M']\!]$。

- 假设 $M = \text{selectQ,P}\ \ell\ M'$。那么 $[\![M]\!]Q = \oplus P\ \ell\ [\![M']\!]Q$，$[\![M]\!]P = \&Q\ \{\ell : [\![M']\!]P\}$，并且对于任意 $P' \notin \{Q, P\}$，$[\![M]\!]P' = [\![M']\!]P'$。因此我们得到 $[\![M]\!] \xrightarrow{\tau}_{[\![D]\!]} [\![M]\!] \setminus \{P, Q\} \mid P[[\![M']\!]P] \mid Q[[\![M']\!]Q]$，结论由此得证。
- 假设 $M = (N,\ N')$，$N \to_D N''$，且 $M' = (N'',\ N')$。然后通过简单归纳法即可得出结论。
- 假设 $M = (V,\ N)$，$N \to_D N'$，且 $M' = (V,\ N')$。然后通过简单归纳法即可得出结论。

- 假设 $M = \text{fst}\ (V, V')$ 且 $M' = V$。那么对于任意满足 $[\![V]\!]P \neq \bot$ 或 $[\![V]\!]P \neq \bot$ 的进程 P，$[\![M]\!]P = \text{fst}\ ([\![M']\!]P\ [\![V']\!]P)$，并且对于任何其他满足 $P' \notin ip(typeof((M', V'))$ 的进程 P，我们有 $[\![M]\!]P' = \bot$ 且 $[\![M']\!]P' = \bot$。我们定义 $\mathcal{N} = \mathcal{N}' = [\![M']\!]$，结论由此得证。

- 假设$M = \text{snd}\,(V, V')$并且$M' = V'$那么该 case 与前一个类似。
- 假设 $M = f$ 且 $M' = D(f)$。那么该结果可由 $[\![D]\!]$的定义得出。

■

定理 6（可靠性）。给定一个封闭编排 M 和一个函数映射 D，若 $\Theta;\ \Gamma \vdash M :\ \tau$，$\Theta;\ \Gamma \vdash D$，并且 $[\![M]\!] \to^*_{[\![D]\!]} \mathcal{N}$对于某个网络 $\mathcal{N}$，则存在一个编排 M′和一个网络 $\mathcal{N}'$，使得：$M \to^*_D M'$，$\mathcal{N} \to^* \mathcal{N}'$，并且 $\mathcal{N}' \sqsupseteq [\![M']\!]$。

与完备性一样，在证明可靠性之前，我们需要一些辅助引理。为此，我们需要一种从网络中移除进程的概念。

定义 3. 给定一个网络 $\mathscr{N} = \prod_{P \in \rho} P[\![L_P]\!]$，有 $\mathscr{N} \setminus \rho' = \prod_{P \in (\rho \setminus \rho')} P[\![L_P]\!]_P$

首先，我们证明网络中的动作不会影响转换标签中未提及的角色。

引理 8. 对于任意进程 P 和网络 $\mathscr{N}$，若 $\mathscr{N} \xrightarrow{\tau_{\mathscr{P}}}_d \mathscr{N}'$ 且 $P \notin \mathscr{P}$ 则 $\mathscr{N}(P) = \mathscr{N}'(P)$。

证明 由网络语义可直接得出。 ■

接下来，我们证明从网络中移除进程不会妨碍网络执行涉及其他进程的动作。

引理 9. 对于任意进程集合 ρ 和网络 $\mathscr{N}$，若 $\mathscr{N} \xrightarrow{\tau_{\mathscr{P}}} \mathscr{N}'$ 且 $\mathscr{P} \cap \rho = /0$ 则 $\mathscr{N} \setminus \rho \xrightarrow{\tau_{\mathscr{P}}}_d \mathscr{N}' \setminus \rho$。

证明 由网络语义直接得出。 ■

最后，我们证明：如果某个编舞类型的投影等价于一个局部类型值，那么原始编舞类型就等价于一个编舞类型值。

引理10. 给定一个闭类型 τ_1 和进程 P，若 $\Theta; \Gamma \vdash \tau_1 :: K$ 且 $[\![\tau_1]\!]_P \equiv_P \nu$，则存在一个类型 ν' 使得：$\tau_1 \equiv \nu'$ 且 $[\![\nu']\!]_P = \nu$。

证明 我们对 τ_1 进行结构归纳。除一种情况外，其余情况均可由简单归纳法得出。

假设 $\tau_1 = \tau_2\ \tau_3$。那么，若 $[\![\tau_2]\!]_P = [\![\tau_2]\!]_P = \bot$，我们有 $[\![\tau_1]\!]_P = \bot$，因此 $\nu = \bot = \nu'$。否则，若 $[\![\tau_3]\!]_{P'} = \bot$ 且 $\mathrm{kindof}(\tau_3) = \mathrm{K} \setminus (\{\mathrm{P}\} \cup \rho)$，则得到 $[\![\tau_1]\!]_P = [\![\tau_2]\!]_P$，结论由归纳法得出。否则，若 $[\![\tau_2]\!]_P = \bot$，则得到 $[\![\tau_1]\!]_P = [\![\tau_3]\!]_P$，结论由归纳法得出。否则，我们得到 $[\![\tau]\!]_P = [\![\tau_2]\!]_P\ [\![\tau_3]\!]_P$。由归纳假设，$[\![\tau_2]\!]_P \equiv_P \nu_2$，且存在 ν'_2 使得 $\tau_2 \equiv \nu'_2$、$[\![\nu'_2]\!]_P = \nu_2$ 和 $[\![\tau_3]\!]_P \equiv_P \nu_3$ 成立；并且存在 ν'_3 使得 $\tau_3 \equiv \nu'_3$ 和 $[\![\nu'_3]\!]_P = \nu_3$ 成立。由于 τ_1 可定种类，存在种类 K' 使得 $\Theta; \Gamma \vdash \tau_2 :: \mathrm{K}' \Rightarrow \mathrm{K}$ 且 $\Theta; \Gamma \vdash \tau_3 :: \mathrm{K}'$。这意味着 $\nu'_2 = \lambda \mathrm{X}.\ \mathrm{K}' \nu_4$，并且若 $\mathrm{K}' \in \{\mathrm{Proc}, \mathrm{Proc} \setminus \rho\}$，则 $\nu_2 = \lambda \mathrm{X}.\ \mathrm{AmI\ X\ ?}\ [\![\nu_4[\mathrm{X} := \mathrm{P}]]\!]_P\ \&\ [\![\nu_4]\!]_P$；否则 $\nu_2 = \lambda \mathrm{X}.\ [\![\nu_4]\!]_P$。于是我们得到 $\nu \equiv_P [\![\nu'_4[\mathrm{X} := \nu_3]]\!]_P$ 和 $\nu' \equiv \nu'_4[\mathrm{X} := \nu'_3]$，并且由于 X 和 ν'_3 都是基本类型，因此 $[\![\nu'_4[\mathrm{X} := \nu_3]]\!]_P$ 和 $\nu'_4[\mathrm{X} := \nu'_3]$ 也都是基本类型。■

接下来，我们可以证明可靠性。

证明 [定理的证明 6] 我们对 M 进行结构归纳。

- 假设 $M = V$。那么对于任意进程 P，$[\![\mathrm{M}]\!]_P = \mathrm{U}$, 因此 $[\![\mathrm{M}]\!] \not\xrightarrow{\tau_{\mathscr{P}}}$。
- 假设 $\mathrm{M} = \mathrm{N1\ N2}$。于是，对于任意满足 $[\![\mathrm{N_1}]\!]\mathrm{P} = [\![\mathrm{N_2}]\!]\mathrm{P} = \bot$ 的进程 P，都有 $[\![\mathrm{M}]\!]\mathrm{P} = \bot$。对于任意满足 P' 使得 $\mathrm{P}' \in \mathrm{ip(typeof(N1))}$ 或 $[\![\mathrm{N_1}]\!]\mathrm{P}' \neq \bot \neq$ 的进程 P

$[\![N_2]\!]_{P'}$， $[\![M]\!]_{P'} = [\![N_1]\!]_{P'}\ [\![N_2]\!]_{P'}$。对于任意其他进程 P'' 满足 $[\![N_2]\!]_{P''} = \bot$，可得 $[\![M]\!]_{P''} = [\![N_1]\!]_{P''}$。对于任意其他进程 P'''，可得 $[\![M]\!]_{P'''} = [\![N_2]\!]_{P'''}$。于是我们得到 2 种情形。

– 假设 $N_2 = V$。根据定理 3，有 $[\![N_2]\!]_P = U$；对于任意满足 P' 的 P，$' \notin ip(typeof(N_2)) \subseteq ip(typeof(N_1))$。根据定理 6，$[\![N_2]\!]_{P'} = \bot$，因此$[\![M]\!]_{P'} = [\![N_1]\!]_{P'}$，我们有 5 种情形。

* 假设 $N_1 = \lambda x : \tau.\, N_3$。那么，对于任意满足 $[\![N_3]\!]_P \neq \bot$ 或$[\![\tau]\!]_P \neq \bot$的进程 P，$[\![N_1]\!]_P = \lambda x : [\![\tau]\!]_P\,.\,[\![N_3]\!]_P$。对于任何其他进程，$[\![N_1]\!]_P = \bot$。因此，任意进程唯一可用的迁移都会使用规则 [NAbsApp]。这意味着存在满足 P'' 的 P，使得 $P = P''$。于是得到 $[\![M]\!] \xrightarrow{\tau} P \longrightarrow [\![M]\!] \setminus \{P''\}\ |P''[[\![N_3]\!]_{P''}[x := [\![N_2]\!]_{P''}]]$。我们称 $M' = N_3[x := N_2]$，结论可由在每个满足 $[\![M]\!]_P \neq \bot$ 的进程 P 中使用规则 [NAbsApp] 并应用归纳法得出。* 假设 $N_1 = 通信^{Q}_{\tau,P}$。那么，如果 $Q \neq P$，$[\![M]\!]_Q = 发送_P\ [\![N_2]\!]_Q$，$[\![M]\!]_P = 接收_P\ \bot$；对于任意 $P' \in ip(\tau)$，$[\![M]\!]_{P'} = sub[Q \mapsto P]\ [\![V]\!]_{P'}$；对于任何其他进程 P''，$[\![N_1]\!]_{P''} = \bot = [\![M]\!]_{P''}$。而如果 $Q = P$，则 $[\![N_1]\!]_P = \lambda x.x$。若 $\mathscr{P} = Q, P$，则 $\mathscr{N} = [\![M]\!] \setminus \{Q, P\} \mid Q[[\![N_2]\!]_P\ [Q := P]] \mid P[[\![N_2]\!]_Q[Q := P]]$。由于 $[\![N_2]\!]_P = \bot$ 且 $[\![N_2]\!]_Q = U$，$N_2 = V$。因此 $M \xrightarrow{\mathscr{P}}_D V\,[Q := P]$；对于任意 $P' \in ip(\tau)$，根据规则 [NSub]，$\mathscr{N}(P') \xrightarrow{\tau}_{[\![D]\!]} [\![V\,[Q := P]]\!]_{P'}$，结论由归纳法得出。若 $\mathscr{P} = P$，则 $Q = P$ 或 $[\![N_1]\!]_P = sub[Q \mapsto P]$。若 $Q = P$，则 $\mathscr{N} = [\![M]\!] \setminus \{P\} \mid P[[\![N_2]\!]_P]$，其余部分与上文类似。若 $[\![N_1]\!]_P = sub[Q \mapsto P]$，则此情形与另外两种情形之一类似。* 否则，$N_1 \neq V$，且 $P = P$ 或 $P = P, Q$。若 $\mathscr{P} = P$，则或者 $[\![N_1]\!]_P \xrightarrow{\tau} L$ 且 $P \in ip(typeof(N_1))$，$\mathscr{N} = [\![M]\!] \setminus \{P\}\ |P[L\ [\![N_2]\!]_P]$。因此有 $[\![N_1]\!] \xrightarrow{\tau} P \longrightarrow [\![N_1]\!] \setminus \{P\} \mid P[L],$，并由归纳法，存在 $N_1 \to^*_D N_1'$，使得 $[\![N_1]\!] \setminus \{P\}\ |P[L] \to^* \mathscr{N}_1 \sqsupseteq [\![N_1']\!]$。由于所有这些迁移都可以在网络中越过 N_2 进行传播，并且在任何涉及的进程 P'中都可以越过 $[\![N_2]\!]_{P'}$ 传播，因此可得 $M' = N_1'\ N_2$的结论。若 $\mathscr{P} = P, Q$，则此情形类似。

– 如果 $N_2 \neq V$，那么有 2 种情形。

* 如果 $\mathscr{P} = P$，那么要么是 $[\![N_1]\!]_P \xrightarrow{\tau} L$，要么是 $[\![N_1]\!]_P = U$ 和 $[\![N_2]\!]_P \xrightarrow{\tau} L$，这种情形与前一种类似。 * 如果 $\mathscr{P} = Q, P$，那么存在 U，使得要么 $[\![N_1]\!]_Q$ $sendP\ U \longrightarrow L_Q$ 或$[\![N_2]\!]_Q\ sendP\ U \longrightarrow L_Q$ 和 $[\![N_1]\!]_P\ recvQ\ U \xrightarrow{[\ =\ ]} Q{:}\ P\ L_P$，要么 $[\![N_2]\!]_P\ recvQ\ U \xrightarrow{[\ =\ ]} Q{:}\ P\ L_P$。如果 $[\![N_1]\!]_Q$ $sendP\ U \longrightarrow L_Q$，那么 $[\![N_1]\!]_Q \neq U'$，因此 $[\![N_1]\!]_P\ recvQ\ U \xrightarrow{[\ =\ ]} Q{:}\ P\ L_P$，这种情形与前一种类似。如果 $[\![N_2]\!]_Q\ sendP\ U \longrightarrow L_Q$，那么 $[\![N_1]\!]_Q = U'$，因此 $[\![N_2]\!]_P\ recvQ\ U \xrightarrow{[\ =\ ]} Q{:}\ P\ L_P$，这种情形与前一种类似。

• 假设 $M = N\ \tau$。那么，对于任何满足 $[\![N]\!]P = [\![\tau]\!]P = \bot$ 的进程 P，我们有$[\![M]\!]P = \bot$。对于任何满足 $[\![\tau]\!]P' = \bot$ 且 $kindof(\tau) = K \setminus (\{P\} \cup \rho)$ 的进程 P'，$[\![M]\!]P' = [\![N]\!]P'$。对于任何其他满足 $[\![\tau]\!]P'' = \bot$ 的进程 P''，我们得到 $[\![M]\!]P'' =$

$[\![N]\!]_{P''}$。对于任何其他进程 P‴，我们得到 $[\![M]\!]_{P'''} = [\![N]\!]_{P'''}$ $[\![\tau]\!]_{P'''}$。这种情况与前一种情况类似，除非 $N = \lambda t :: K_{\circ}$ N' 且 $\tau = B$。

若 N = Λ X :: K. N′ 且 τ ≡ ν 成立，则有两种情形：K ∈ {Proc, Proc \ ρ} 成立，或不成立。若 K ∈ {Proc, Proc \ ρ} 成立，则对于任意 P′，$[\![M]\!]_{P'} = \Lambda$ X. AmI X ?$[\![$N′[t := P′]$]\!]$P & $[\![N']\!]_{P'}$ $[\![\nu]\!]_{P'}$。由于 $[\![\nu]\!]_{P'}$ =P 对某个 P 成立，唯一可用的迁移是使用规则 [NBabs],，因此我们得到 $\mathscr{P}$ =P″ 对某个 P″ 成立，且$\mathscr{N} = [\![M]\!] \setminus \{P''\} \mid P''[$AmI P ? $[\![$N′[X := P″]$]\!]$P″ & $[\![N']\!]_{P''}$。接着定义 M′ =N′[X := ν]，并可见，使用规则 [NIamr] 和 [NProam] 作用于 P″（若 P″ =P 成立），否则在所有其他进程处使用规则 [NIaml] 和 [NProam],，再使用规则 [NBAbs]，然后使用规则 [NIamr] 和 [NProam]，或规则 [NIaml] 和 [NProam]，即可得出结果；该结论由归纳法可得。

如果 K ∉ {Proc, Proc \ ρ} 那么此 case 与$N_1 = \lambda X :$ $\tau.\, N_3$ 上文类似。

- 假设 M = fst N。那么，要么 N ≠V，结论由归纳法得出；要么 N =(V ,V ′)，并且对于任意进程 P ∈ip(typeof((V ,V ′))), $[\![M]\!]$P =fst ($[\![V]\!]$P $[\![V']\!]$P)，对于任意其他进程 P′ ∉ip(typeof((V ,V ′))，由定理 6 可得 $[\![M]\!]P' = [\![N]\!]P' = \bot$，因此 $[\![M]\!]P' \not\to$。

  如果 $\mathscr{P}$ =P ∈ip(typeof((V, V′))) 那么 $\mathscr{N} = [\![M]\!] \setminus \{P\} \mid P[[\![V]\!]_P]$ 并且 $M P \to_D V$。利用规则 [NProj1] 和定理6 以及归纳法可得结论。
- 假设 N_1 =snd N2。此 case 与前一种情况类似。
- 假设 $M = (M_1, M_2)$。那么结论由简单归纳法得出。
- 假设 M = 情形 N 为 inl x ⇒ N′; inr x′ ⇒N″。那么，对于任意满足 P ∈ ip(typeof(N)) 的进程 P，我们有 $[\![M]\!]$P = case $[\![N]\!]$P 为 inl x ⇒ $[\![N']\!]$P; inr x′ ⇒ $[\![N'']\!]$P。对于任意其他满足 $[\![N]\!]P' = [\![N']\!]P' = [\![N'']\!]P' = \bot$ 的进程 P′，$[\![M]\!]P' = \bot$。对于任意其他满足 $[\![N]\!]P'' = \bot$ 的进程 P″，我们得到 $[\![M]\!]P'' = [\![N']\!]P'' \sqcup [\![N'']\!]P''$。对于任意其他满足 $[\![N']\!]P''' = [\![N'']\!]P''' = \bot$ 的进程 P‴，我们有 $[\![M]\!]P''' = [\![N]\!]P'''$。对于任意其他进程 P⁗，我们有 $[\![M]\!]P'''' = (\lambda x : \bot.[\![N']\!]P'''' \sqcup [\![N'']\!]P'''')$ $[\![N]\!]P''''$。我们分两种情形讨论。

  – 假设 $\mathscr{P}$ = P ∈ 进程标识集合(类型(N))。那么有三种情形。

    * 假设 $N = \text{左注入}_{\tau} V_{\circ}$ Then $[\![N]\!]_P = \text{左注入}_{[\![\tau]\!]_P} [\![V]\!]_P$ and $\mathscr{N} = [\![M]\!] \setminus \{P\} \mid P[[\![N'[x := [\![V]\!]_P]]\!]_P]$。我们定义 $M' = N'$ 并且由于 $[\![N']\!]_{P'} \sqsupseteq [\![N']\!]_{P'} \sqcup [\![N'']\!]_{P'}$ 该结果可由使用规则 [NAbsApp] 和 [NCasel] 以及归纳法得到。
    * 假设 $N = \text{右注入}_{\tau} V$。那么，该情形与前一个情形类似。
    * 否则，我们使用规则 [NCase]，并且有一个迁移 $[\![N]\!]_P \xrightarrow{\tau} L$ 满足

      $$\mathscr{N} = [\![M]\!] \setminus \{P\} \mid P[\text{case } L \text{ of inl } x \Rightarrow [\![N']\!]_P;\ \text{inr } x' \Rightarrow [\![N'']\!]_P]$$

      并且，该结果可由类似于上一个应用情形的归纳法得到。

  – 假设 $\mathscr{P}$ = Q， P。那么，其逻辑与前一个情形的第三个子情形类似。

- 假设 M =选择$_{Q}$ $_{,P}$ ℓ N。这与N_1 =通信$^{Q}_{\tau}$ $_{,P}$ 上述 case。
- 假设 M = f。那么对于任意进程 P，$[\![M]\!]$P = f。因此，存在某个进程 P，使得 $\mathscr{P}$ = P 且 $\mathscr{N} = ([\![M]\!] \setminus P) \mid P[[\![D]\!](f)]$。然后我们定义所需的编排 M′ = D(f) 和网络 $\mathscr{N}' = [\![M']\!]$，结论由此得证。

■

根据定理 3、4、5以及6，我们得到以下推论：由类型正确的封闭编排导出的网络可以持续规约，直至所有角色都只包含局部值。

推论 1. 给定一个封闭编排 M，以及一个包含 M 所有函数名的函数环境 D，若 $\Theta; \Gamma \vdash M : \mathrm{T}$ 且 $\Theta; \Gamma \vdash \mathrm{D}$，则：只要 $[\![M]\!] \to^*_{[\![D]\!]} \mathscr{N}$ 对于某个网络 $\mathscr{N}$ 成立，要么存在 $\mathscr{P}$ 使得 $\mathscr{N} \xrightarrow{\tau}_{\mathscr{P}[\![D]\!]} \mathscr{N}'$，要么 $\mathscr{N} = \prod_{P \in ip(\mathrm{M})} P[\mathrm{V}_P]$。

6 相关工作

6.1 编排

编排的灵感源自 Needham 和 Schroeder 提出的安全协议“‘Alice 和 Bob’表示法”（1978年），其中包含一个用于表示从一个参与者到另一个参与者的通信的项。这一理念后来也启发了多种用于对进程间交互进行建模的图形表示法，包括顺序图和消息时序图（对象管理组，2017年；国际电信联盟，1996年）。

最早用于表达编排的成熟语言，是为描述 Web 服务之间的交互而发明的。W3C 制定的 Web 服务编舞描述语言（WS-CDL）（W3C，2004年）是一种编舞语言，从全局视角描述 Web 服务之间预期可观察到的交互（Zongyan 等人，2007年）。Carbone 等人（2012年）随后基于 WS-CDL，将端点投影形式化为编舞理论，并特别引入了合并算子（我们对其进行了调整和扩展，以适用于我们的场景）。这推动了更多关于编排和投影的研究，并最终促成了编舞式编程的诞生——编排作为程序被编译为可执行代码——以及第一种编舞式编程语言 Chor（Montesi，2013年）。随着编舞式编程语言日益复杂，Cruz-Filipe 和 Montesi（2020年）开发了编舞核心演算（CC）。Montesi（2023年）在其关于编舞语言基础的著作中重新审视并推广了 CC。随后，Cruz-Filipe 等人（2021年，?）在 Coq 定理证明器中形式化了这一新版本及其性质（Coq 开发团队，2004年）。之后，Pohjola 等人（2022年）利用 HOL 定理证明器，开发了一个从 CC 的另一种变体到 CakeML 的经过认证的端到端编译器。

所有编舞式编程语言的主要设计目标之一是设计上无死锁（Carbone 和 Montesi，2013年）——编排与分布式网络之间的操作对应关系确保了网络无死锁。PolyChorλ实现了这一目标。

首个支持有限进程多态的编舞语言是过程式编舞（PC）（Cruz-Filipe 和 Montesi，2017年）。在 PC 中，编排附带一个预定义过程环境，这些过程以进程名称为参数，可由编排调用。与 PolyChor 的进程多态相比，这些过程存在若干限制λ：它们不能包含任何自由进程，也不能

被部分实例化，而且它们是低阶的——也就是说，它们必须在环境中定义，而不能作为更大编排的一部分。这些限制使投影函数能够确定过程将如何实例化；而在 PolyChorλ 中，我们可能需要先计算涉及哪些进程。这对投影有重大影响：在 PC 中，投影过程调用时，很容易判断涉及哪些进程，因此哪些进程需要保留该调用的本地副本。然而，PolyChorλ 的完全进程多态允许函数名称和进程名称分别封装在上下文中。这虽然为程序员提供了更大的灵活性，却迫使我们将进程多态函数投影到每个进程，并让各进程在运行时自行判断是否参与其中。

近年来，人们对编排中的高阶编程产生了相当浓厚的兴趣（Giallorenzo 等人，2020年；Hirsch 和 Garg，2022年；Cruz-Filipe 等人，2021年）。第一个高阶编舞编程语言 Choral（Giallorenzo 等人，2020年）是一种编译为 Java 代码的面向对象的编舞语言。因此，Choral 编排可以依赖其他编排，从而让程序员复用代码。Choral 也是第一个将 通信 视为一等函数的编舞语言。

虽然 Choral 为高阶编舞程序提供了实际实现，但并未确立其理论性质。两项彼此独立开发的不同研究填补了这一空白，其中包括 Chorλ，它是 PolyChorλ的基础。Chorλ是一种基于 λ演算的函数式编舞演算。在这项工作中，我们扩展了 Chorλ，为其加入类型多态和进程多态，以及发送编排等非局部值的能力。Chorλ以及由此衍生的 PolyChorλ为高阶编舞程序提供了核心语言，使我们能够确立其性质。由于原始的 Chorλ和 PC、Choral 一样采用参数化过程，因此它不具备 PolyChorλ的这一性质：在某个进程中发散的编排必须在每个进程中都发散。这使得 Chorλ不得不采用复杂的乱序执行概念，并放宽网络中的动作与编排之间的对应关系。

另一项确立高阶编舞式编程理论性质的工作是 Pirouette（Hirsch 和 Garg，2022年），它同样是一种基于简单类型 λ 演算的函数式编舞式编程语言。与 Chorλ （因而也与 PolyChorλ）不同，Pirouette 不允许进程发送用 Pirouette 编写的消息。相反，它从低阶编舞编程语言中汲取灵感；在这些语言中，消息（及生成消息所需的计算）是用各自独立的语言编写的。与（Montesi，2023年；Cruz-Filipe 等人，2021年）中的编舞语言一样，Pirouette 的设计由用于编写消息的语言参数化。因此，Pirouette 可以描述进程之间的通信模式，其局部计算可采用大量不同的语言。然而，这种设计意味着，与 PolyChorλ 不同，Pirouette 从根本上无法允许程序发送编排函数。

此外，与 Chorλ和 PolyChorλ不同，Pirouette 要求每个进程在调用函数时都进行同步。这一设计使 Pirouette 能够与其网络编程语言建立双模拟关系，并且这一结果已在 Coq 中得到形式化证明。由此可以通过构造给出传统且经过验证的无死锁性证明。然而，这种持续同步会成为实际系统中的瓶颈；PolyChorλ选择建立一种较弱但足够强的语言间联系，从而避免了这种高昂的代价。

6.2 并发函数式编程

函数式并发编程历史悠久，最初的尝试是将函数式程序的执行并行化（Burton，1987）。第一个支持在信道上进行通信的函数式编程语言是 Facile（Giacalone 等，1989）；与后来的编舞语言不同，它对进程标识符进行了抽象，这与进程多态非常相似。近期一项更接近编舞式编程的工作是 Links（Cooper 等，2006），其核心语言为 RPC 演算（Cooper 和 Wadler，2009）。与编排一样，Links 和 RPC 演算允许程序员描述计算的不同部分位于不同位置的程序，然后将其编译成各个位置对应的独立代码。有趣的是，尽管 Links 显式支持通信，RPC 演算在顶层却不支持通信；只有在投影位于另一进程上的函数时，才会在投影过程中创建通信。此外，RPC 演算不支持多个计算线程；相反，通信发生时，唯一的计算线程会移动到新位置，而其他位置则会阻塞。后来，RPC 演算扩展加入了位置多态，这与我们提出的进程多态以及 Facile 的进程多态非常相似（Choi 等，2020年）。不过，由于 RPC 演算只处理由两个进程（客户端和服务器）组成的系统，因此它们将进程抽象投影为一对，然后通过选择这对中的正确部分来确定位置。这种方案生成的网络语言更简单，但不适用于 PolyChorλ这类支持任意数量参与者的框架。此外，RPC 演算和 PolyChorλ一样，与传统编舞语言不同，顶层不支持乱序执行。

Gay 和 Vasconcelos（2010）将会话类型应用于一种支持异步通信的并发函数式演算。然而，他们的会话类型并不能保证无死锁性。相反，他们得到的是一个运行时安全性结果：只有在线程已经终止、尝试从空缓冲区读取，或尝试与尚未准备就绪的另一个线程建立新信道时，线程才无法执行操作。

ML5（Licata 和 Harper，2010；Murphy VII 等人，2007）是一种基于模态逻辑语义的函数式并发编程语言。然而，与编舞语言中的发送和接收项不同，它们有一个原语 get[w] M，它让另一个进程w 求值M 并返回结果。由于M 可能包含其他 get 操作，这一构造使 ML5 具备了某种类似于 PolyChorλ发送完整编排的能力。然而，对这一“编排”求值的结果必须先在接收方产生，然后再返回给发送方。

多层编程语言（如 ScalaLoci（Weisenburger 和 Salvaneschi，2020年））为描述分布式系统的全局视图提供了另一种范式。与 Chor 类似，ScalaLoci 构建于现有的面向对象语言之上：此处使用的是 Scala。在 ScalaLoci 和其他无层语言中，一个对象描述的是包含多个进程和函数的整个系统。与编舞式编程不同，多层编程不允许对预期的通信流进行建模。相反，通信通过远程函数调用隐式发生，而要遵循的具体协议在很大程度上由运行时中间件决定。关于编舞式语言和多层语言的更详细比较，请参阅 Giallorenzo 等人（2021年）的研究。

7 结论

本文介绍了 PolyChorλ，这是首个支持进程多态的高阶编舞式编程语言。PolyChorλ采用基于 System Fω的类型与种类系统，并对其进行了扩展，使进程名称成为种类为Proc的类型。此外，我们展示了如何通过引入一种新构造，在进程语言中并发解释 PolyChorλ程序；该构造对应于进程获知自身身份的能力。我们发现，如果进程变量可以实例化为其所在的进程，就必须使用这一构造；而使用编舞式编程语言则可抽象掉这一必要性。对进程多态的探索还使 PolyChorλ能够将从P 向Q 通信非局部值，描述为将消息中由P 拥有的部分发送给Q。这些非局部值包括完整的编排，由此可以通过通信一个描述受委托任务的分布式函数，以简单灵活的方式描述委托。这一创新要求我们重新定义通信：它是一种交换，在这种交换中，委托方发送编排后不会留下一个空值，而是留下一个函数，使其有可能参与受委托任务，例如在任务结束时接收结果。

进程多态填补了高阶编舞程序理论与实用语言之间的大部分空白。不过，仍有许多工作要做。例如，目前 PolyChorλ尚不支持递归类型。要支持递归类型，我们要么需要让端点投影能够投影出可能不终止的进程描述，要么限制递归类型导致类型计算无法终止的能力。

此外，我们可以设想允许进程发送类型和进程名称，以及值。例如，这样我们就能编写一个服务器程序，使其等待接收客户端的名称，然后与该客户端交互。由于这种形式的委托在实践中很常见，若能理解如何在编舞语言中提供这一能力，同时保留编舞式编程的保证，程序员便能将惯用的编程模式应用于编舞代码。

尽管缺乏针对局部进程的类型判断系统，我们仍然会投影局部类型。我们不同寻常的网络通信语义，使得为发送操作和接收操作定义局部类型规则变得困难，因此我们将局部类型判断（或者作为替代方案，类型擦除）留作未来工作。

最后，尽管确定的、即时的同步通信便于理论研究，但这些假设并不符合现实世界中的分布式系统。Cruz-Filipe 和 Montesi（2017年）通过运行时项表示传输中的消息，对编排中的异步通信进行了建模。我们可以通过让通信原语分两步归约来将这种方法应用于 PolyChor：λ首先归约为一个运行时项，然后再归约为已交付的值。不过，这一扩展并非易事，因为当传输中的消息是编排这类非局部值时，目前尚不清楚该如何表示它们。

尽管理论与实践之间的这些差距依然存在，PolyChorλ中的进程多态让我们离适用于分布式系统的现实编舞语言更近了一步。编舞程序有望让并发编程和

分布式编程变得更简单、更清晰，并提供更可靠的保证。借助高阶编舞编程和进程多态，这一愿景几乎触手可及。

参考文献

Burton, F. (1987) 并发与分布式编程的函数式编程。《计算机杂志》。30(5), 437–450. Caires, L. & Pfenning, F. (2010) 会话类型即直觉线性命题。并发理论 (CONCUR)。Carbone, M., Honda, K. & Yoshida, N. (2012) 面向 Web 服务的结构化通信中心编程。《编程语言与系统汇刊》(TOPLAS)。34(2). Carbone, M. & Montesi, F. (2013) 设计即无死锁：多方异步全局编程。编程语言原理 (POPL)。Castagna, G., Dezani-Ciancaglini, M. & Padovani, L. (2012) 关于全局类型与多方会话。《计算机科学逻辑方法》。8(1). Choi, K., Cheney, J., Fowler, S. & Lindley, S. (2020 197, 102499) 多态 RPC 演算。SCP. . Cooper, E., Lindley, S., Wadler, P. & Yallop, J. (2006) Links：无层级的 Web 编程。组件与对象形式化方法 (FMCO)。Cooper, E. E. & Wadler, P. (2009) RPC 演算。声明式编程原理与实践 (PPDP)。Cruz-Filipe, L., Graversen, E., Lugovic, L., Montesi, F. & Peressotti, M. (2021) 函数式编舞编程。CoRR. abs/2111.03701. Cruz-Filipe, L. & Montesi, F. (2017) 关于异步性与编排。交互与并发体验 (ICE)。Cruz-Filipe, L. & Montesi, F. (2017) 过程式编舞编程。分布式对象、组件与系统形式化技术 (FORTE)。Cruz-Filipe, L. & Montesi, F. (2020) 编舞编程的核心模型。《理论计算机科学》。802, 38–66. Cruz-Filipe, L., Montesi, F. & Peressotti, M. (2021) 编舞编译认证。国际计算理论方面研讨会 (ICTAC)。Cruz-Filipe, L., Montesi, F. & Peressotti, M. (2021) 在 Coq 中形式化一种图灵完备编舞语言。交互式定理证明 (ITP)。Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I. & Mauro, J. (2017) 动态编排：理论与实现。《计算机科学逻辑方法》。13(2). Dardha, O., Giachino, E. & Sangiorgi, D. (2012) 重新审视会话类型。声明式编程原理与实践 (PPDP)。DeYoung, H., Caires, L., Pfenning, F. & Toninho, B. (2012) 线性逻辑中的切消去即异步会话类型通信。计算机科学逻辑 (CSL)。Gay, S. J. & Vasconcelos, V. T. (2010) 异步会话类型的线性类型理论。《函数式编程杂志》(JFP)。20(1). Giacalone, A., Mishra, P. & Prasad, S. (1989) Facile：并发与函数式编程的对称集成。《国际并行编程杂志》。18, 121–160. Giallorenzo, S., Montesi, F. & Peressotti, M. (2020) 作为对象的编舞。Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G. & Weisenburger, P. (2021) 多方语言：编舞情形与多层情形（Pearl）。欧洲面向对象编程会议 (ECOOP)。Girard, J.-Y. (1972) 高阶算术的函数解释与切消去。博士论文。巴黎第七大学。Hirsch, A. K. & Garg, D. (2022) Pirouette：高阶类型化函数式编舞。编程语言原理

编程语言原理 (POPL)。Honda, K. (1993) 二元交互类型。并发理论 (CONCUR)。Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) 基于结构化通信的编程的语言原语与类型纪律。欧洲编程研讨会 (ESOP)。Honda, K., Yoshida, N. & Carbone, M. (2016) 多方异步会话类型。ACM期刊。63(1), 1–67. 国际电信联盟。(1996) Z.120号建议书：消息序列图。Jongmans, S. & van den Bos, P. (2022) 编舞的谓词变换器——计算编舞式编程中的前置条件。欧洲编程研讨会 (ESOP)。Licata, D. R. & Harper, R. (2010) ML5 的单子形式化。逻辑框架与元语言：理论与实践 (LFMTP)。L´opez, H. A., Nielson, F. & Nielson, H. R. (2016) 在故障感知通信系统中确保可用性。分布式对象、组件与系统形式化技术 (FORTE)。Coq 开发团队。(2004) Coq 证明助手参考手册。LogiCal项目。版本 8.0. Montesi, F. (2013) 编舞式编程。博士论文。哥本哈根信息技术大学。Montesi, F. (2023) 编排导论。剑桥大学出版社。已获出版许可。Murphy VII, T., Crary, K. & Harper, R. (2007) 使用 ML5 实现类型安全的分布式编程。可信全球计算机 (TGC)。Needham, R. & Schroeder, M. (1978年) 在大型计算机网络中使用加密进行身份验证。ACM通讯。21(12), 993–999. Needham, R. M. & Schroeder, M. D. (1978年) 在大型计算机网络中使用加密进行身份验证。ACM通讯 (CACM)。21(12). 对象管理组。(2017年) 统一建模语言，2.5.1版。Pohjola, J. ˚A., G´omez-Londo˜no, A., Shaker, J. & Norrish, M. (2022年) Kalas：一种经过验证的端到端编译器，适用于编舞语言。第13届国际交互式定理证明会议，ITP 2022，2022年8月7日至10日，以色列海法。莱布尼茨信息学中心 Schloss Dagstuhl。第27:1–27:18页。Richter, D., Kretzler, D., Weisenburger, P., Salvaneschi, G., Faust, S. & Mezini, M. (2022年) Prisma：一种用于在去中心化应用中实施合约客户端协议的无层级语言。欧洲面向对象编程会议 (ECOOP)。Scalas, A. & Yoshida, N. (2019年) 少即是多：重新审视多方会话类型。编程语言原理 (POPL)。(W3C), T. W. W. W. C. (2004) WS编排模型概述。访问日期：2021年1月29日。Wadler, P. (2012年) 命题即会话。国际函数式编程会议 (ICFP)。Weisenburger, P. & Salvaneschi, G. (2020年) 实现分布式系统语言：Scala 中类型级编程与宏编程的选择与经验。ASEP。4(3), 17. Zongyan, Q., Xiangpeng, Z., Chao, C. & Hongli, Y. (2007年) 迈向编排的理论基础。万维网大会 (WWW)。

1 Full PolyChorλ Typing Rules

$$[\textsc{Tunit}]\ \frac{\Theta;\Gamma\vdash \nu :: \mathsf{Proc}}{\Theta;\Gamma\vdash () \mathbin{@} \nu : () \mathbin{@} \nu} \qquad [\textsc{Tint}]\ \frac{\Theta;\Gamma\vdash \nu :: \mathsf{Proc}}{\Theta;\Gamma\vdash n \mathbin{@} \nu : \mathsf{Int} \mathbin{@} \nu}$$

$$[\textsc{Tapp}]\ \frac{\Theta;\Gamma\vdash N : \tau_1 \to_\rho \tau_2 \qquad \Theta;\Gamma\vdash M : \tau_1}{\Theta;\Gamma\vdash N\ M : \tau_2}$$

$$[\textsc{Tabs}]\ \frac{\Theta;\Gamma\vdash \tau_1 :: * \qquad \Theta;\Gamma'\vdash \nu :: \mathsf{Proc} \text{ for all } \nu\in\rho \qquad \Theta\cap(\rho\cup \mathsf{ip}(\tau_1)\cup\mathsf{ip}(\tau_2)\cup \mathrm{ftv}(\tau_1)\cup\mathrm{ftv}(\tau_2)); \Gamma, x:\tau_1\vdash M:\tau_2}{\Theta;\Gamma\vdash \lambda\, x:\tau_1.\, M : \tau_1 \to_\rho \tau_2}$$

$$[\textsc{Tsel}]\ \frac{\Theta;\Gamma\vdash \nu_1 :: \mathsf{Proc} \qquad \Theta;\Gamma\vdash \nu_2 :: \mathsf{Proc} \qquad \Theta;\Gamma\vdash M : \tau}{\Theta;\Gamma\vdash \mathsf{select}_{\nu_1,\nu_2}\ \ell\ M : \tau}$$

$$[\textsc{Tcom}]\ \frac{\Theta;\Gamma\vdash \tau :: \mathsf{Proc} \Rightarrow * \qquad \Theta;\Gamma\vdash \nu_1 :: \mathsf{Proc} \setminus (\mathsf{ip}(\tau)\cup\mathrm{ftv}(\tau)) \qquad \Theta;\Gamma\vdash \nu_2 :: \mathsf{Proc} \setminus (\mathsf{ip}(\tau)\cup\mathrm{ftv}(\tau))}{\Theta;\Gamma\vdash \mathsf{com}^{\tau}_{\nu_1,\nu_2} : (\tau\ \nu_1 \to_\varnothing \tau\ \nu_2)}$$

$$[\textsc{TappT}]\ \frac{\Theta;\Gamma\vdash M : \forall X :: \mathsf{K}.\, \tau_1 \qquad \Theta;\Gamma\vdash \tau_2 :: \mathsf{K}}{\Theta;\Gamma\vdash M\ \tau_2 : \tau[X\mapsto\tau_2]}$$

$$[\textsc{TabsT}]\ \frac{\begin{array}{c}\Theta';\Gamma', X :: \mathsf{K}\vdash M : \tau \\ \text{if } \exists \mathsf{K}', \rho.\ \mathsf{K} = \mathsf{K}' \setminus \rho \text{ then } \Gamma' = (\Gamma + X)\ \&\ \rho \setminus X \text{ else } \Gamma' = \Gamma + X \\ \text{if } \mathsf{K} = \mathsf{Proc} \text{ or } \exists \rho.\ \mathsf{K} = \mathsf{Proc}\setminus\rho \text{ then } \Theta' = \Theta, X \text{ else } \Theta' = \Theta\end{array}}{\Theta;\Gamma\vdash \Lambda X :: \mathsf{K}.\, M : \forall X :: \mathsf{K}.\, \tau}$$

$$[\textsc{Teq}]\ \frac{\Theta;\Gamma\vdash M : \tau_1 \qquad \tau_1 \equiv \tau_2 \qquad \Theta;\Gamma\vdash \tau_2 :: *}{\Theta;\Gamma\vdash M : \tau_2}$$

$$[\textsc{Tdefs}]\ \frac{\forall f \in \mathsf{domain}(D).\ f:\tau\in\Gamma \wedge \emptyset;\Gamma\vdash D(f):\tau}{\Theta;\Gamma\vdash D} \qquad [\textsc{Tvar}]\ \frac{x:\tau\in\Gamma}{\Theta;\Sigma;\Gamma\vdash x:\tau}$$

$$[\textsc{Tcase}]\ \frac{\Gamma\vdash N : \tau_1 + \tau_2 \qquad \Theta;\Gamma, x:T_1\vdash M' : \tau \qquad \Theta;\Gamma, x':T_2\vdash M'' : \tau}{\Theta;\Gamma\vdash \mathsf{case}\ N\ \mathsf{of}\ \mathsf{inl}\ \ x\Rightarrow M';\ \ \mathsf{inr}\ \ x'\Rightarrow M'' : \tau}$$

$$[\textsc{Tfun}]\ \frac{f:\tau\in\Gamma}{\Theta;\Gamma\vdash f:\tau} \qquad [\textsc{Tpair}]\ \frac{\Theta;\Gamma\vdash M:\tau_1 \qquad \Theta;\Gamma\vdash N:\tau_2}{\Theta;\Gamma\vdash (M,N):\tau_1\times\tau_2}$$

$$[\textsc{Tproj1}]\ \frac{\Theta;\Gamma\vdash M:\tau_1\times\tau_2}{\Theta;\Gamma\vdash \mathsf{fst}\,M:\tau_1} \qquad [\textsc{Tproj2}]\ \frac{\Theta;\Gamma\vdash M:\tau_1\times\tau_2}{\Theta;\Gamma\vdash \mathsf{snd}\,M:\tau_2}$$

$$[\textsc{Tinl}]\ \frac{\Theta;\Sigma;\Gamma\vdash M:\tau_1}{\Theta;\Sigma;\Gamma\vdash \mathsf{inl}_M\,\tau_2:\tau_1+\tau_2} \qquad [\textsc{Tinr}]\ \frac{\Theta;\Sigma;\Gamma\vdash M:\tau_2}{\Theta;\Sigma;\Gamma\vdash \mathsf{inr}_M\,\tau_2:\tau_1+\tau_2}$$

2 Full PolyChorλ Operational Semantics

$$[\textsc{AppAbs}]\ \frac{}{(\lambda\, x:\tau.\, M)\ V\to_D M[x\mapsto V]}$$

$$[\textsc{App1}]\ \frac{M_1\to_D M_2}{M_1\ N\to_D M_2\ N} \qquad [\textsc{App2}]\ \frac{N_1\to_D N_2}{V\ N_1\to_D V\ N_2}$$

$$[\textsc{AppTAbs}]\ \frac{\tau\equiv\nu}{(\Lambda X::\mathsf{K}.\, M)\ \tau\to_D M[X\mapsto\nu]} \qquad [\textsc{MTApp1}]\ \frac{M_1\to_D M_2}{M_1\ \tau\to_D M_2\ \tau}$$

$$[\textsc{MTApp2}]\ \frac{\tau_1\to_D\tau_2}{V\ \tau_1\to_D V\ \tau_2}$$

$$[\textsc{Inl}]\ \frac{M_1\to_D M_2}{\mathsf{inl}_\tau\ M_1\to_D \mathsf{inl}_\tau\ M_2} \qquad [\textsc{Inl}]\ \frac{M_1\to_D M_2}{\mathsf{inr}_\tau\ M_1\to_D \mathsf{inr}_\tau\ M_2}$$

$$[\textsc{Case}]\ \frac{N_1\to_D N_2}{\mathsf{case}\ N_1\ \mathsf{of\ inl}\ \ x\Rightarrow M_1;\ \mathsf{inr}\ \ y\Rightarrow M_2\to_D \mathsf{case}\ N_2\ \mathsf{of\ inl}\ \ x\Rightarrow M_1;\ \mathsf{inr}\ \ y\Rightarrow M_2}$$

$$[\textsc{CaseL}]\ \frac{}{\mathsf{case}\ \mathsf{inl}_\tau\ V\ \mathsf{of\ inl}\ \ x\Rightarrow M_1;\ \mathsf{inr}\ \ y\Rightarrow M_2\to_D M_1[x\mapsto V]}$$

$$[\textsc{CaseR}]\ \frac{}{\mathsf{case}\ \mathsf{inr}_\tau\ V\ \mathsf{of\ inl}\ \ x\Rightarrow M_1;\ \mathsf{inr}\ \ y\Rightarrow M_2\to_D M_1[x\mapsto V]}$$

$$[\textsc{Pair1}]\ \frac{M_1\to_D M_2}{(M_1,N)\to_D (M_2,N)} \qquad [\textsc{Pair2}]\ \frac{N_1\to_D N_2}{(V,N_1)\to_D (V,N_2)}$$

$$[\textsc{Fst}]\ \frac{M_1\to_D M_2}{\mathsf{fst}\ M_1\to_D \mathsf{fst}\ M_2} \qquad [\textsc{Snd}]\ \frac{M_1\to_D M_2}{\mathsf{snd}\ M_1\to_D \mathsf{snd}\ M_2}$$

$$[\textsc{Proj1}]\ \frac{}{\mathsf{fst}\ \ (V_1,V_2)\to_D V_1} \qquad [\textsc{Proj2}]\ \frac{}{\mathsf{snd}\ \ (V_1,V_2)\to_D V_2}$$

$$[\textsc{Def}]\ \frac{}{f\to_D D(f)}$$

$$[\textsc{Sel}]\ \frac{}{\mathsf{select}_{\mathsf{P},\mathsf{Q}}\ \ell\ M\to_D M} \qquad [\textsc{Com}]\ \frac{}{\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}\ V\to_D V[\mathsf{P}\mapsto\mathsf{Q}]}$$

3 Full Network and Local Program Operational Semantics

$$[\textsc{NSend}]\ \mathsf{send}_{\mathsf{P}}\ V \xrightarrow{\mathsf{send}_{\mathsf{P}}\ V\ V'}_d V' \qquad [\textsc{NRecv}]\ \mathsf{recv}_{\mathsf{P}}\ V \xrightarrow{\mathsf{recv}_{\mathsf{P}}\ V'\ V}_d V'$$

$$[\textsc{NCom}]\ \frac{L_1 \xrightarrow{\mathsf{send}_{\mathsf{P}}\ V\ V'[\mathsf{Q}:=\mathsf{P}]}_d M_1' \qquad L_2 \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ V[\mathsf{Q}:=\mathsf{P}]\ V'}_d M_2'}{\mathsf{Q}[M_1] \mid \mathsf{P}[M_2] \xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_d \mathsf{Q}[M_1'] \mid \mathsf{P}[M_2']}$$

$$[\textsc{NCho}]\ \oplus_{\mathsf{P}}\ \ell\ M \xrightarrow{\oplus_{\mathsf{P}}\ \ell}_d M \qquad [\textsc{NOff}]\ \&_{\mathsf{P}}\ \{\ell_1 : M_1, \ldots, \ell_n : M_n\} \xrightarrow{\&_{\mathsf{P}}\ \ell_i}_d M_i$$

$$[\textsc{NSel}]\ \frac{L_1 \xrightarrow{\oplus_{\mathsf{P}}\ \ell}_d M_1' \qquad L_2 \xrightarrow{\&_{\mathsf{Q}}\ \ell}_d M_2'}{\mathsf{Q}[M_1] \mid \mathsf{P}[M_2] \xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_d \mathsf{Q}[M_1'] \mid \mathsf{P}[M_2']} \qquad [\textsc{NAmIR}]\ \mathsf{Aml}\ \mathsf{P}\ ?\ M\ \&\ M' \xrightarrow{\mathsf{P}}_d M$$

$$[\textsc{NAmIL}]\ \frac{\mathsf{Q} \neq \mathsf{P}}{\mathsf{Aml}\ \mathsf{P}\ ?\ M\ \&\ M' \xrightarrow{\mathsf{Q}}_d M'} \qquad [\textsc{NAbsApp}]\ (\lambda\, x : \tau.\, M)\ V \xrightarrow{\tau}_d M[x \mapsto V]$$

$$[\textsc{NSub}]\ \mathsf{sub}[\mathsf{P} \mapsto \mathsf{Q}]\ V \xrightarrow{\tau}_d V[\mathsf{P} \mapsto \mathsf{Q}] \qquad [\textsc{NProam}]\ \frac{M \xrightarrow{\mathsf{P}}_d M'}{\mathsf{P}[M] \xrightarrow{\tau_{\mathsf{P}}}_d \mathsf{P}[M']}$$

$$[\textsc{NBAbs}]\ \frac{\tau \equiv_{\mathsf{P}} \nu}{(\Lambda X.\, M)\ \tau \xrightarrow{\mathsf{P}}_d M[X \mapsto \nu]}$$

$$[\textsc{NApp1}]\ \frac{M_1 \xrightarrow{\mu}_d M_2}{M_1\ M' \xrightarrow{\mu}_d M_2\ M'} \qquad [\textsc{NApp2}]\ \frac{M \xrightarrow{\mu}_d M'}{V\ M \xrightarrow{\mu}_d V\ M'} \qquad [\textsc{NTApp1}]\ \frac{M \xrightarrow{\mu}_d M'}{M\ \tau \xrightarrow{\mu}_d M\ \tau}$$

$$[\textsc{NPro}]\ \frac{M \xrightarrow{\tau}_d M'}{\mathsf{P}[M] \xrightarrow{\tau_{\mathsf{P}}}_d \mathsf{P}[M']} \qquad [\textsc{NInl}]\ \frac{M \xrightarrow{\mu}_d M'}{\mathsf{inl}_\tau\ M \xrightarrow{\mu}_d \mathsf{inl}_\tau\ M'} \qquad [\textsc{NInr}]\ \frac{M \xrightarrow{\mu}_d M'}{\mathsf{inr}_\tau\ M \xrightarrow{\mu}_d \mathsf{inr}_\tau\ M'}$$

$$[\textsc{NCase}]\ \frac{M \xrightarrow{\mu}_d M'}{\mathsf{case}\ M\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow L_2 \xrightarrow{\mu}_d \mathsf{case}\ M'\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2}$$

$$[\textsc{NCaseL}]\ \mathsf{case}\ \mathsf{inl}_\tau\ V\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2 \xrightarrow{\tau}_d M_1[x \mapsto V]$$

$$[\textsc{NCaseR}]\ \mathsf{case}\ \mathsf{inr}_\tau\ V\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2 \xrightarrow{\tau}_d M_2[x \mapsto V]$$

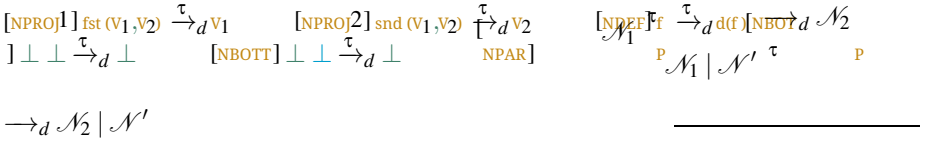
$$[\textsc{NPair1}]\ \frac{M_1 \xrightarrow{\mu}_d M_1'}{(M_1, M_2) \xrightarrow{\mu}_d (M_1', M_2)} \qquad [\textsc{NPair2}]\ \frac{M_2 \xrightarrow{\mu}_d M_2'}{(M_1, M_2) \xrightarrow{\mu}_d (M_1, M_2')}$$

$$[\textsc{Fst}]\ \frac{M_1 \to_D M_2}{\mathsf{fst}\ M_1 \to_D \mathsf{fst}\ M_2} \qquad [\textsc{Snd}]\ \frac{M_1 \to_D M_2}{\mathsf{snd}\ M_1 \to_D \mathsf{snd}\ M_2}$$

[NPROJ1] fst (V1,V2) $\xrightarrow{\tau}_d$ V1 [NPROJ2] snd (V1,V2) $\xrightarrow{\tau}_d$ V2 [NDEF] f $\xrightarrow{\tau}_d$ d(f) [NBOT $\longrightarrow_d \mathcal{N}_2$

] $\perp \perp \xrightarrow{\tau}_d \perp$ [NBOTT] $\perp \perp \xrightarrow{\tau}_d \perp$ NPAR] $\mathcal{N}_1$ P $\mathcal{N}_1 \mid \mathcal{N}'$ τ P

$\longrightarrow_d \mathcal{N}_2 \mid \mathcal{N}'$



此图“by.png”提供以下“PNG”格式：

http://arxiv.org/ps/2303.04678v1