arXiv:2304.03764v1 [cs.PL] 7 Apr 2023

参数化代数协议

Andreia Mordido[1],Janek Spaderna[2], Peter Thiemann[2], 和 Vasco T. Vasconcelos [1]

[1] LASIGE，里斯本大学理学院，葡萄牙里斯本[2] 弗赖堡大学，德国弗赖堡

摘要。 我们提出了代数协议，它支持定义协议模板和会话类型，类似于利用代数数据类型定义领域特定类型。参数化代数协议涵盖所有正则会话类型，以及大多数上下文无关和嵌套会话类型；同时，它以线性时间运行的名义检查取代了开销高昂的超线性类型检查算法。代数协议与多态相结合，通过提供参数化和组合会话类型的新方式，提升了表达能力和模块化程度。

1 引言

自 20 世纪 80 年代以来，人们便已认识到使用代数数据类型（ADTs）对数据进行建模的优势 Burstall [1977], Burstall 等人 [1980]。代数数据类型易于定义和使用，能够支持构建领域特定类型，也适用于高效的名义类型检查。函数式编程语言很快便采用了代数数据类型及模式匹配这一概念，而模式匹配至今仍日益受到重视。

代数数据类型在单个声明中整合了递归类型、和类型、积类型和生成性（每个 ADT 定义都会创建一个新类型）。不同类型概念的平滑整合使 ADT 易于学习和使用。它还简化了类型检查器的实现，因为类型检查器可以省去用于检查等递归类型等价性的超线性算法（即将递归类型视为与其展开形式相等），转而对名义类型进行线性比较。

我们展示了用于定义递归数据类型的 ADT 方法也可以应用于代数协议（AP），这是一种定义一大类递归协议的新方法。与 ADT 类似，AP 在单个声明中整合了会话类型理论中的概念 Honda [1993], Takeuchi 等人[1994] （递归类型、选择类型和序列类型）以及生成性。与 ADT 类似，AP 易于使用，也适用于高效的类型检查。

我们通过类比代数数据类型来构建代数协议。请考虑以下用于整数列表的代数数据类型。

```
data IntList =Nil | Cons 整数 IntLi st
```

数据类型 `IntList` 包含两个数据构造函数：`Nil` 用于构造空列表，`Cons`用于通过一个 `Int` 类型的元素和一个`IntList`类型的列表来构造列表。构造函数之间的选择对应于和类型，而 `Cons` 的两个参数则产生积类型。显然，`IntList` 是一种递归类型。此外，它的定义将 `IntList` 引入为一种新类型，不同于任何其他类型。类型等价很简单：我们只需按名称比较代数数据类型。

虽然`IntList` 描述了一种（静态）数据类型，但它隐含了动态行为，读者很自然就能将此类型的元素想象成两个通信伙伴之间的有限消息序列：如果还有后续元素，发送方就选择构造子`Cons`，

发送一个 整数 类型的值，然后递归。如果没有后续元素，发送方就选择Nil 并停止传输。在另一端，接收方等待一个构造子。接收到Cons 后，它就接收一个整数 并递归。接收到Nil后，它就停止。

这种观点与二元会话类型理论有着显著的相似之处 Honda [1993], Honda 等人 [1998], Takeuchi 等人 [1994]。会话类型为通信信道提供了丰富的类型纪律，可静态保证协议保真性以及通信信道间的类型可靠性。下面给出与 IntList ADT 相关协议的一种会话类型，采用现有系统的语法表示 Almeida 等人 [2022], Gay 和 Vasconcelos [2010], Lindley 和 Morris[2017] ，并从发送方的视角来看：[3]

类型 IntListS = $\mu\alpha.\ \oplus\{\texttt{Cons} : !\text{整数}.\alpha, \texttt{Nil} : \text{结束}_!\}$

前导符号 $\mu\alpha\ldots$ 构成一个等递归类型。其主体是一个内部选择类型 $\oplus$，包含两个由标签 Cons 和 Nil 表示的选项。如果发送方选择 Cons，就必须继续遵循 $!\text{整数}.\alpha$，也就是说，它必须发送一个整数，然后递归。如果发送方选择Nil，协议便如类型 $\text{结束}_!$ 所示结束。类型 $\text{结束}_!$ 表示一个已准备好关闭的信道端点。对偶类型 $\text{结束}_?$ 则表示另一个等待关闭的端点。顾名思义，会话类型描述了通信伙伴之间从开始到结束的完整交互会话。

与这个例子类似，大多数会话类型系统依赖结构化类型和等递归；这样一来，展开不会引发通信。传统的会话类型系统将递归限制在尾位置（如 IntListS），因此类型实际上是具有正则轨迹语言的有限自动机。因此，这类系统的类型等价和子类型关系可在多项式时间内判定。近年来，人们开始研究更丰富的递归形式，取消了对尾递归的限制，使得类型类似于确定性上下文无关文法等 Almeida 等人[2022], Das 等人 [2021], Padovani [2017, 2019], Thiemann和Vasconcelos [2016]。遗憾的是，为类型赋予更丰富的递归形式后，类型等价会变得更加复杂（达到双指数级 Das 等人 [2021]）。为这些系统实现类型等价算法并非易事 Almeida 等人 [2020]；人们使用了不完备算法来加快类型等价检查 Das 等人 [2021]；也有人提出注释 Padovani [2019] ，使类型检查更加高效。但在所有这些工作中，类型等价算法的渐近复杂度仍然是超线性的。

遗憾的是，结构化会话类型相当冗长，对普通程序员来说也过于复杂，因此阻碍了它们的普及。尽管等递归类型表达能力很强，但程序员难以理解它们的等价性，而且它们在语言实现中的效率也不高

如果使用富有表现力的递归协议编程能像使用代数数据类型编程一样简单，那会怎样？本文提出的代数协议理论与实践，正好提供了这样一种方法。通过以 System F 的风格加入参数和多态，代数协议在定义和实现通信协议时，为模块化和抽象提供了一种独特简洁且高效的方法。

这 是与会话类型 IntListS 对应的代数协议声明：[4]

协议 IntListP =Nil | Cons 整数 IntListP

关键字 协议取代数据，表示代数 协议 类型，而非代数数据 类型。协议类型就像一个模板，用于描述会话中可组合的片段。

[3] 用 IntListS 表示与 IntList 对应的会话类型。[4] 用 IntListP 表示与 IntList 对应的协议类型。

它既不描述完整的会话，也不限定特定的通信方向。相反，协议类型必须通过指定一个方向并给出后续会话，才能具体化为会话类型：我们用 `!IntListP.S` 表示发送一串整数，并继续使用会话类型 `S` ；或用 `?IntListP.S`表示接收一串整数，并继续使用 `S`，因此会话类型 `IntListS` 对应于 `!IntListP.`结束。因此，协议类型与会话类型的关系，正如可组合的后续与后续之间的关系 Danvy and Filinski [1992],Felleisen[1988]：前者描述一段交互（执行），而后者描述直至结束的交互（执行）。

就像代数数据类型一样，我们可以为代数协议配备参数，例如`ListP a`:

协议 `ListP a = Nil | Cons a ListP`

在代数数据类型定义中，参数取值范围是类型；而在代数协议定义中，参数取值范围是协议！这一功能是代数协议实现模块化的关键。例如，协议`ListP Q` 会重复协议`Q`。通过考虑协议`ListP -Q`，也可以翻转`Q` 的方向：重复协议`Q` ，但反转发送方和接收方的角色（详见第 2.2 节）。借助这种模块化设计，我们可以定义由任何可表示为代数协议的对象构成的有限流。之前会话类型系统中的外部选择和内部选择，由代数类型的和积结构体现——和类型替代内部/外部选择，积类型则替代顺序组合，这与 Caires 等人 [2016] 对会话类型的线性逻辑解释一致。

代数协议类型具有名义性，并采用同构递归。这些特性的结合，将类型等价的复杂度从渐近意义上的双指数 Almeida 等人 [2020],Das 等人 [2021] 降低到了线性。值得注意的是，代数协议的表达能力与具有非正规递归的会话类型相当，Das 等人 [2021], Thiemann和Vasconcelos [2016],因此，在此类理论中可表达的所有协议都可以转换为代数协议（而无需让编译器承担实现复杂或不完备算法的代价）。有趣的是，由于顺序组合的基本性质，类型现在还具有同构结合性，从而免去了定义硬编码结合律的需要，Almeida 等人 [2022], Thiemann和Vasconcelos [2016]。

贡献

– 代数协议（AP）。我们提出一种定义递归协议的新方法，可组合地 y
受参数化代数数据类型的启发。该方法以类似于代数数据类型的方式，将命名、递归和序列类型选择结合起来。它为标签的组成协议引入了一种新颖的方向概念。由于 AP 采用名义类型，类型等价的检查变得简单高效（最坏情况下与类型的大小呈线性关系）。– 代数会话类型（AlgST）。我们定义了一种基于带有（非线性）递归的线性System F的核心演算，并支持基于 AP 的会话类型。AlgST 的类型检查由双向类型规则规定，这些规则可直接转化为算法。我们通过保持性和进展性证明类型可靠性。– 完整实现以公开可用的制品形式发布，见 Spaderna 等人 [2023], ，其中包含实用扩展（参见第 5 节）。

概述

第2 节通过示例对代数协议进行了非正式介绍；我们在第2.3 节中说明了相较于其他工作的表达能力和模块化方面的提升。第 3 节讨论了

AlgST 的类型结构，并确立了高效类型检查的关键结果：类型规范化的正确性以及类型等价的最坏情况复杂度。第4节定义了表达式和进程的静态语义与动态语义，并介绍元理论结果。第5节讨论实现；由于所有规则都是算法性的，按照恰当的方式解读这些规则即可直接得到类型检查器和解释器。第6节和第7节讨论相关工作并作总结。证明、辅助结果、更多示例、将上下文无关会话类型嵌入 AlgST 的非正式概述以及进一步讨论，均作为补充材料提供。实现已由 Spaderna等人公开发布 [2023]。

2 代数协议与会话类型

本节从程序员的视角介绍代数会话类型（AlgST）。内容以示例为主，并引入代数协议和会话类型的概念。我们的语法受到 Haskell 的启发，并已在我们的制品中实现。示例广泛使用了形式系统的实用扩展（参见第5节）。

2.1 以代数数据类型作为协议

假设我们要传输一种语言的抽象语法树，其中包含整数和加法。其类型可以像在 Haskell 或 ML 中那样，通过代数数据类型定义来定义。

```
数据 Ast = 构造函数 整数 | 加法 Ast Ast
```

此声明定义了类型 `Ast` 以及构造函数的类型。这些构造函数可用于构造 `Ast` 值，也可用于对它们进行模式匹配：

```
Con : Int → Ast
Add : Ast → Ast → Ast
```

类型 `Ast` 是递归的，其 `Add` 构造函数接受两个 `Ast` 值作为参数。要定义用于传输 `Ast`类型值的协议，我们只需将关键字`data` 改为 `protocol` ，如下所示：

```
协议 AstP = ConP 整数 | AddP AstP AstP
```

此声明定义了协议类型 `AstP` 以及标签 `ConP` 和 `AddP`，它们用于引入和消除协议中的分支。由此得到的 `AstP` 协议可用于两个方向：发送数据时使用 `!AstP` ，接收数据时使用 `?AstP` 。[5] 协议类型只描述行为，不对运行时值进行分类。它们属于专用的种类 `P`。

我们通过一个用于发送 `Ast` 值的函数来演示 `AstP` 协议。我们采用信道传递风格编写函数：函数以通道为参数，在处理完通道会话类型的某个前缀后返回该通道。函数的类型对未探索部分通道的延续类型`s` 进行全称量化。参数类型在 `s` 前添加一个前缀，用来描述函数所执行的会话部分。在本例中，该前缀为`!AstP`。

```
sendAst : Ast → ∀(s: S). !AstP .s →s
sendAst t [s]c = 匹配 t，其中 { Con x → 选择 ConP [s] c ▷ sendInt [s] x , Add l r →选择 AddP [s] c ▷sendAst l [!AstP .s] ▷sendAst r [s] }
```

[5] 该实现使我们能够将数据声明重用为协议声明，从而实现构造函数重载。

我们使用 ▷ 运算符来进行反向函数应用。其定义为 `x ▷ f = f x`，该运算符从左结合且优先级较低，因此 `x ▷ f ▷ g` 表示 `g (f x)`。方括号表示类型抽象（在左侧）和类型应用（在表达式中）。预定义的`sendInt` 操作具有类型 `∀(s:S). Int → !Int.s → s`。在代数数据类型上使用`case`进行模式匹配是标准做法。与会话类型系统中通常的情况一样，会话类型的通道值是线性的，而`select` 操作会在多个备选项之间进行内部选择。选择一个协议构造子会发送相应标签的编码，并将字段的传输“压入”通道类型中。即：

```
选择 ConP [s]:!AstP.s →!整数.s选择 AddP [s]:!
AstP.s →!AstP.!AstP.s
```

处理此协议接收端的函数必须接受一个类型为 `?AstP.s` 的参数。

```
recvAst : ∀(s: S).?AstP.s →(Ast, s) recvAst [s]c = 匹配 c with { ConP c →let (x,
c)=receiveInt [s] c in (Con x, c), AddP c →let (tl, c) =recvAst [?AstP.s] c in let
(tr, c) =recvAst [s] c in (Add tl tr, c) }
```

对会话类型通道执行的 匹配 会接收一个标签，并据此进行分支。分支也会“压入”通道类型中字段的传输，这与 选择 操作类似：

```
c:?AstP.s ⊢匹配 c与{ConP c →...-- c:?整数.sAddP c →...}-- c:?AstP.?AstP.s
```

竖线左侧 ⊢，我们写出 `c` 在 匹配 之前的类型。在 匹配 的主体中，我们分别写出 `c` 与选择器 `ConP` 和 `AddP` 匹配后的类型。匹配 `c` （左侧带有类型）会消耗通道`c`的线性绑定。模式 `ConP c` 会重新引入变量`c`，将其绑定到处理选择后得到的通道，并为相应分支的主体赋予更新后的类型（位于右侧）。模式`AddP c` 的处理方式类似。分支的选择由外部决定，取决于接收到的选择器标签。

本小节中的协议超出了大多数会话类型系统的能力范围，因为它们只支持尾递归会话类型。因此，函数 `sendAst` 和 `recvAst` 在传统会话类型系统中没有有意义的类型 Gay 和 Vasconcelos [2010], Lindley 和 Morris[2017]。然而，它们可以使用上下文无关类型或嵌套会话类型来编写 Almeidaetal. [2022],Das 等人 [2021], Thiemann和Vasconcelos [2016]。我们将在第 6 节讨论其中的差异。

2.2 带极性的简单协议

第 2.1 节中的协议类型`AstP` 描述了单向通信。本节为协议类型扩展了极性，以描述双向通信。算术服务器（参见 Gay 和 Hole [1999]）的协议类型`Arith` 就是一个例子。服务器接收一条命令，`Neg` 或`Add`，接受该命令的一个或两个输入，并生成输出。

```
协议Arith = Neg 整数-整数 | Add 整数 整数-整数
```

此声明中的新元素是极性 “-”，它表示通信方向的反转。客户端会以“正向”使用此协议，也就是使用形如`!Arith.s` 的会话类型；而服务器则会实现`?Arith.t`，其中某个会话类型`t`与`s`对偶。在`!Arith.s`上，标签选择的工作方式与第 2.1 节中相同，但负极性会翻转方向：

```
选择 Neg [s] : ! Arith .s →!整数 .?整数 .s选择 Add [s] : !
Arith .s →!整数 .!整数 .?整数.s
```

分支在假设中的类型上以类似方式运作：

```
c : ? Arith .s ⊢匹配 c 与 { Neg c →... -- c : ?整数 .! 整数 .sAdd c →... }-- c : ?整数 .? 整数 .! 整数 .s
```

实　　现服务器的实现很直接。（客户端代码留给读者完成。）

```
serveArith : ∀(s: S). ? Arith .s →s serveArith [s]c = 匹配 c 与 {
Neg c →let (x , c) =接收Int [! 整数 .s] c in sendInt [s] (0 - x) c ,
Add c →let (x , c) =接收Int [? 整数 .! 整数 .s] c in let (y , c) =接收I
nt [! 整数 .s] c in sendInt [s] (x+y) c }
```

标签声明Neg Int -Int 表明标签参数本身也是协议，而不是普通类型。AstP 被用作AstP 的标签参数，以及-Int 被用作Arith的标签参数，都印证了这一点。像Int 这样的普通类型会被提升为协议类型。

2.3 参数化协议与模块化

代数协议可以像代数数据类型具有类型参数一样，具有协议参数。这项功能为声明模块化协议提供了一种强大而灵活的方法。首先，我们声明一个能够反复运行另一个协议的协议：

```
协议 Stream a = Next a (Stream a)
```

使用这个协议，我们定义一个进程，在通道上无限次发送 1 ：

```
ones : ! Stream Int.End! →Unitones c =选择 Next [Int ,End!] c ▷sendInt [! Stream Int .
End!] 1 ▷ones
```

使用Stream 协议时，延续会话类型结束! 并不重要，因为该协议永远不会到达延续。对于参数化协议，选择 接受与协议参数相对应的类型参数（整数）以及延续会话（结束!）。代数协议避开了递归定义收缩性和协议守卫性方面的问题：我们必须始终通过选择Next 标签来显式推进。

到目前为止，我们所展示的示例经过一些努力，都可以用以往关于会话类型的研究成果来定义。以下示例则利用了使 AlgST 脱颖而出的特性：

– 传统会话类型缺乏可组合性，因为只有在尾位置才能对行为进行抽象。Almeida 等人的多态上下文无关会话类型 [2022]放宽了这一限制，但代价是类型等价算法在最坏情况下需要双指数时间。AlgST 允许对协议的任意部分进行抽象，同时提供线性时间的类型等价算法。

– 协议种类是 AlgST 的一项独特特性，它支持对整个子协议及其在协议中任意位置的方向进行抽象。这项功能使我们能够定义一套受正则语言运算启发的泛型协议运算符工具箱，并使用否定运算符在客户端和服务器视图之间切换。

泛型服务器 参数化协议支持构建可对从属协议进行抽象的泛型服务器。我们的第一个示例创建了一个整数流，但如果我们想要一个无限运行 Arith 协议的 流算术 服务器，该怎么办？ 为此，我们编写一个名为 stream 的泛型服务器，它实现适用于任意 a 的 Stream a 协议，并调用一个用于子协议 a 的从属服务器。我们用一个参数化类型别名来抽象这个从属服务器的类型：

```
类型 服务 a = ∀(s:S).?a.s →s
```

有了这些准备，我们就可以编写通用服务器，并用它运行流算术。

```
流： ∀(a: P). 服务 a →?Stream a.结束! →单位类型流 [a] serve c = 匹
配 c with {Next c →serve? Stream a.结束!] c ▷流 [a] serve}

streamArith : ?Stream Arith.结束! →单位类型
streamArith = 流 [Arith] serveArith
```

我们把以这种方式构造的服务器称为被动 服务器，因为它会等待接收Next 标签，然后再运行辅助协议。与此相对，我们也可以定义一个主动 服务器。

主动服务器——对偶性与取负的练习 主动服务器向客户端提供 Next 标签，以表明它已准备好接受请求。这样的服务器是对 一次性 服务器的推广。我们只需翻转类型中的两个位：

```
streamAct： ∀(a: P). 服务 a →!Stream -a.结束! →单位类型streamAct [a]
服务c =
  选择 Next [-a , End!] c ▷svc [!Stream -a.End!] ▷streamAct [a] sv
                                                                  c
```

为什么类型中必须写 !Stream -a ？ 人们可能会认为，主动服务器应运行在一个通道上，而该通道的类型是 ?Stream a.End! 的对偶，也就是!Stream a.End?。但方向的改变也会影响参数 a 中的子协议，如 select 的类型所示：

```
  c : ! Stream Arith.End! ⊢选择 Next [Arith , End!] c : ! Arith . ! Stream a.
  End!
```

但是，这个类型并不是类型 算术服务 = ∀(s:S). ?Arith.s。借助极性，我们可以适配为 c : !Stream -Arith ， 从而切换为输出方向：

```
  选择 Next [-Arith , 结束!] c : ? Arith . ! Stream - Arith . 结束!
```

这样，我们可以按如下方式重用 serveArith 。

```
streamActArith : ! Stream -Arith .结束! →单位类型
streamActArith = streamAct [Arith] serveArith
```

同一个主动服务器也可以提供无限的 1 流：

```
streamActOnes : ! Stream 整数.结束! →单位类型
streamActOnes =streamAct [-整数] ( sendInt 1)
```

有趣的是， streamActOnes 的类型涉及极性的双重否定。注意sendInt 1 : Service -整数，因此传输的流实际上是Stream -(-整数) ≡ Stream 整数。

泛型服务器的工具箱 受协议 `Stream a` 和`ListP a` （来自第 1 节）的启发，我们可以构建一个包含这些参数化协议及其泛型服务器的库。这些协议具体化了标准类型构造子，让人联想到正则语言运算符。

```
协议 Seq ab = Seq ab -- 乘积协议 Either ab = Left a | Right b -- 和类
型协议 Repeat a = More a (Repeat a) | Quit -- 迭代
```

相应的被动泛型服务器（`seq`、`either` 和 `repeat`）的直接实现，以及它们的主动变体，留给读者完成。

这些协议类型及其泛型服务器足以为所有流程可用正则语言描述的协议构造骨架服务器。例如，我们可以通过将泛型服务器与取负和加法的服务器组合起来，构建 `Repeat Arith` 的服务器。

```
类型 Neg = Seq 整数 -整数类型 Add=Seq整数 (
Seq整数 -整数 )类型 Arith =Either Neg Add
```

```
serveNeg : 服务 Neg
serveNeg [s]c = 匹配 c with {Seqc →let(x,c) =接收整
数 [!整数 .s] cinsendInt [s](0-x) c}
```

```
serveAdd : 服务 Add
serveAdd [s]c = 匹配 c with { Seq c →let (x , c) =接收整数 [? Seq 整
数 -整数 .s] c in 匹配 c with { Seq c →let (y , c) =接收整数 [! 整数 .s] c i
n sendInt [s](x+y) c }}
```

```
serveArith : 服务 算术服务
serveArith =Either [Neg] serveNeg [Add ] serveAdd
```

```
serveAriths : 服务 ( Repeat 算术服务) serveAriths =
重复 [Arith] serveArith
```

与第 2.2 节中展示的算术服务器相比，这里会产生一些额外开销。之前，客户端除了处理 `Repeat` 协议外，只需处理 `Neg` 和 `Add` 标签。而在这里，`Left` 和 `Right` 取代了 `Neg` 和 `Add` ，客户端还需额外处理 `Seq` 标签。因此，由泛型部分构造的协议会产生一些额外开销，详见补充材料 A.6。补充材料 B 中还提供了更多泛型服务器的示例。

3 类型

AlgST 的底层类型结构由三部分组成：种类结构、类型语言和协议定义。类型语言与协议定义互为递归。为简化阐述，形式化语言基于线性System F。虽然我们对协议定义进行了形式化，但没有对（线性）代数数据类型及模式匹配的标准处理进行形式化，尽管我们在示例中使用了它。

类型语言涵盖了具有不同特征的类型（如第 2 节所示）。为了正确进行类型抽象，我们需要通过种类为类型建立分类。AlgST 区分三种种类。

$\mathbf{s}$ 用于分类会话类型，而会话类型又用于分类通信信道的端点。$\mathbf{t}$ 用于分类所有用于表示运行时值的类型。$\mathbf{p}$ 用于分类协议类型。种类 $\mathbf{p}$ 通过隐式提升涵盖其他所有种类，该提升会将一种类型关联到适合传输该类型的协议。协议类型本身用于分类纯行为，而不表示任何运行时值。

种类按照子种类关系线性排序；该关系是严格序 $\mathbf{s} < \mathbf{t} < \mathbf{p}$的自反传递闭包。类型形成包含一条标准的种类包含规则。

类型的语法如下：

$$
\begin{array}{rcll}
S, T, U & ::= & \texttt{Unit} \mid T \to U \mid T \otimes U \mid \forall \alpha : \kappa . T \mid \alpha \mid & \text{functional types} \\
& & ?T.S \mid !T.S \mid \texttt{End}_? \mid \texttt{End}_! \mid \texttt{Dual}\ S \mid & \text{session types} \\
& & \rho\ \overline{T} \mid -T & \text{protocol types}
\end{array}
$$

我们主要使用元变量 S表示会话类型（种类 $\mathbf{s}$）。元变量 T、U表示函数类型（种类 $\mathbf{t}$）和协议类型（种类 $\mathbf{p}$）。函数类型包含线性 System F 中常见的类型：单位类型、函数、积、全称量化以及类型变量。会话类型运算符包括接收、发送、被动与主动终止，以及对偶运算符。对偶运算符会交换会话类型链中的所有通信方向。因此，会话类型是由 结束$_?$ 或 结束$_!$终止的通信序列，或者可通过类型变量 α 扩展；该变量的种类为 $\mathbf{s}$。

协议类型$\rho\ T$ 指的是声明为 ρ 且具有参数 T的协议。运算符“$-$”反转其协议参数的通信方向。对偶性将会话类型从外向内转换，而反转运算符则将协议从内向外转换。

种类上下文 Δ 将类型变量映射到种类，并将协议名称映射到一阶箭头种类；这些种类决定协议的参数数量。形式上，对于不带参数的协议构造子，其种类应写作 $\varepsilon \to \mathbf{p}$，但我们通常将其简写为$\mathbf{p}$。

$$
\Delta ::= \cdot \mid \Delta, \alpha : \kappa \mid \Delta, \rho : \overline{\mathbf{p}} \to \mathbf{p}
$$

图 1 给出了类型形成规则。我们从一开始就以算法形式呈现这些规则。判断式 $\Delta \vdash T \Rightarrow \kappa$ 为类型 T 分配其最小种类 κ，此分配是在种类上下文Δ下进行的。相应的检查判断式 $\Delta \vdash T \Leftarrow \kappa$ 以 T 和 κ 为输入，并检查合成的种类是否为预期种类的子种类。

单位类型（T-Unit）、函数（T-Arrow）、对（T-Pair）和全称多态（T-Poly）的类型，其种类 $\mathbf{t}$ 与其组成部分相同。变量可以具有分配给它们的任何种类（T-Var）。AlgST 的会话类型概念区分会话的被动 结束$_?$ 和主动 结束$_!$（T-End?、T-End!），以及对偶运算符（T-Dual）。传统会话类型要求为消息中交换的值指定类型；而这里我们要求的是协议（T-In和 T-Out）。规则 T-Protocol根据协议名称 ρ构造协议类型。我们假定种类上下文包含 ρ 的绑定，以确定其元数。协议的所有参数本身都必须是协议。后者并无限制，因为任何类型都可以提升为协议。协议类型可以使用逆运算符改变方向（规则 T-MsgNeg）。

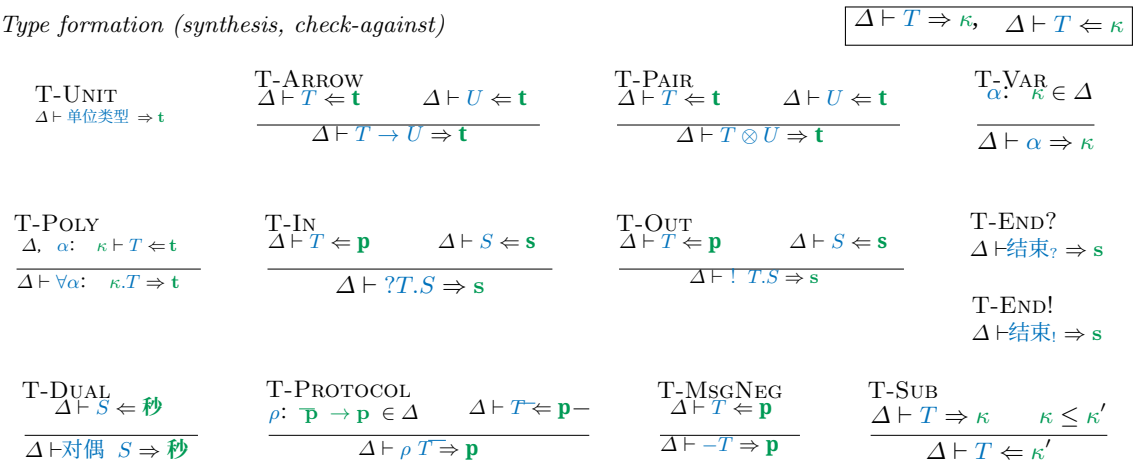


图 1：算法类型形成规则

代数协议类型 $\rho\ \alpha$ 的定义与代数数据类型类似。

$$\texttt{protocol}\ \rho\ \overline{\alpha} = \{C_i\ \overline{T_i}\}_{i \in I}$$

此声明定义了协议类型构造函数 ρ，其参数数量由 α 指定。该协议包含由 $i \in I$ 索引的选择。每个选择都带有一个选择器标签 C_i，该标签保护一系列子协议 T_i，这些子协议按顺序处理。我们假设选择器标签全局唯一。程序可以使用任意完整应用 $\rho\ U$ 作为协议类型。我们使用类型形成规则检查单个协议声明，其中 Δ 可以引用先前定义的协议：[6]

$$\frac{\texttt{protocol}\ \rho\ \overline{\alpha} = \{C_i\ \overline{T_i}\}_{i \in I} \qquad \Delta, \rho\colon \overline{\mathbf{p}} \to \mathbf{p}, \overline{\alpha}\colon \overline{\mathbf{p}} \vdash \overline{T_i} \Leftarrow \mathbf{p} \qquad (\forall i \in I)}{\Delta \vdash \rho \Rightarrow \overline{\mathbf{p}} \to \mathbf{p}}$$

类型遵循一种按种类索引的转换关系。图 2 展示了声明式规则。转换具有自反性、对称性和传递性（C-Refl、C-Sym、C-Trans）。我们将对偶性定义为会话类型上的一种转换。这些规则表明，对偶性会改变从外部发起的传输方向（C-DualEnd?、C-DualEnd!、C-DualIn、C-DualOut），并且对偶性是自反的（C-DualInv）。种类包含关系被提升为转换（C-Sub）。最后一行规定了逆运算符与会话类型构造函数之间的交互：逆运算符会反转从内部发起的方向（C-NegIn、C-NegOut）。逆运算符也是自反的（C-NegInv）。我们省略了标准的同余规则。

虽然类型形成是算法化的，但类型转换的声明式定义更加清晰。对于接下来的算法化表达式类型判断关系，我们定义了一个类型规范化算法，它相对于声明式规则是可靠且完备的。

在范式类型中，逆运算符在顶层（类型种类为 $\mathbf{p}$）最多只能出现一次，并且出现在协议类型的参数顶层。对偶运算符只出现在会话类型脊柱末端的类型变量上。因此，计算算法

[6] 对于互 递的协议 ρ，我们在检查每个单独的 ρ 时，加入所有 ρ 的假设 $_i$.

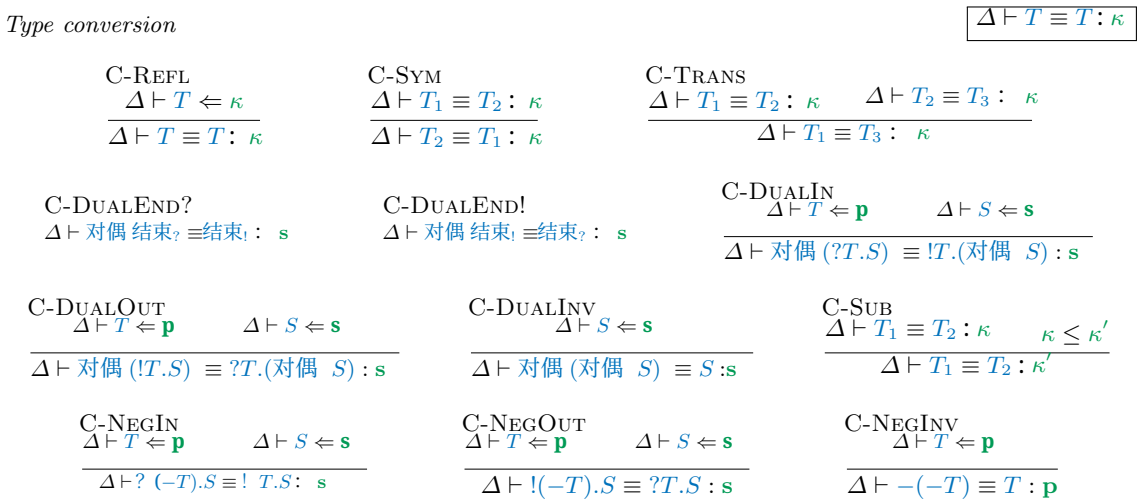


图 2：类型转换（省略同余规则）

范式必须通过应用转换规则，将所有对偶运算符沿会话类型的主干向下压入 C-DualIn，C-DualOut，C-DualEnd?，以及 C-DualEnd!，从左到右应用。如果沿主干遇到另一个对偶运算符，则应用规则 C-DualInv。我们无法进一步化简对偶 α，因此它是一个范式。

逆运算符仅适用于协议类型。因此，它只能出现在如下消息类型中：$?(-T).S$，以及协议类型的参数中。对于消息类型，必须从左到右穷尽应用规范化规则 C-NegIn 和 C-NegOut，从而从消息类型中移除逆运算符。对于协议类型的参数，规范化会穷尽应用规则 C-NegInv。从奇数个逆运算符开始，最终会剩下一个运算符；从偶数个开始，则会移除所有逆运算符。

遵循这一思路，图 3 中通过两个相互递归函数 $\mathrm{nrm}^+(\,)$ 和 $\mathrm{nrm}^-(\,)$ 以及三个辅助函数来定义归一化。正向归一化函数 $\mathrm{nrm}^+(\,)$ 遍历并重构所有非会话类型构造。对于会话类型，它还会应用于传输值的类型，即应用于 T 类型中的 $!T.S$。对 对偶 运算符进行归一化时，会在正向和负向归一化之间来回切换，即在 $\mathrm{nrm}^+(\,)$ 和 $\mathrm{nrm}^-(\,)$ 之间切换。

带上标 $^-$ 的负向归一化表示有一个待处理的对偶 类型构造器，因此函数 $\mathrm{nrm}^-(\,)$ 仅用于会话类型。归一化会沿着会话类型的主干压入待处理的对偶构造器，以实现规则 C-DualIn、C-DualOut、C-DualEnd?和 C-DualEnd!。待处理的对偶化会在类型变量上具体化。

规范化反向协议 $-T$会对 T 的范式应用负向函数 $-(T')$，以强制执行转换规则 C-DualInv。

Type normalization $\boxed{\mathrm{nrm}^+(T) = T \quad \mathrm{nrm}^-(T) = T}$

nrm^+(单位类型) $=$ 单位类型 nrm^+(对偶 T) $= \mathrm{nrm}^-(T)$ $\mathrm{nrm}^+(T \to U) = \mathrm{nrm}^+(T) \to \mathrm{nrm}^+(U)$ $\mathrm{nrm}^+(\rho\ T) = \rho\,\mathrm{nrm}^+(T)$ $\mathrm{nrm}^+(T \otimes U) = \mathrm{nrm}^+(T) \otimes \mathrm{nrm}^+(U)$ $\mathrm{nrm}^+(-T) = -(\mathrm{nrm}^+(T))$ $\mathrm{nrm}^+(\forall\alpha\colon \kappa.T) = \forall\alpha\colon \kappa.\mathrm{nrm}^+(T)$ nrm^-(对偶 T) $= \mathrm{nrm}^+(T)$ $\mathrm{nrm}^+(\alpha) = \alpha$ $\mathrm{nrm}^-(\alpha) =$对偶 $\alpha\mathrm{nrm}^+(?\ T.S) = \S(-(\mathrm{nrm}^+(T)))$。$\mathrm{nrm}^+(S)$ $\mathrm{nrm}^-(?\ T.S) = \S(+(\mathrm{nrm}^+(T)))$。$\mathrm{nrm}^-(S)$ $\mathrm{nrm}^+(!T.S) = \S(+(\mathrm{nrm}^+(T)))$。$\mathrm{nrm}^+(S)$ $\mathrm{nrm}^-(!T.S) = \S(-(\mathrm{nrm}^+(T)))$。$\mathrm{nrm}^-(S)$ $\mathrm{nrm}^+(\text{结束}_?) = \text{结束}_?$ $\mathrm{nrm}^-(\text{结束}_?) = \text{结束}_!$ $\mathrm{nrm}^+(\text{结束}_!) = \text{结束}_!$ $\mathrm{nrm}^-(\text{结束}_!) = \text{结束}_?$

Materialization and the directional operators $\boxed{\S(T).T = T \quad -(T) = T \quad +(T) = T}$

$$\S(-T).U = ?T.U \qquad -(-T) = +(T) \qquad +(-T) = -(T)$$
$$\S(T).U = !T.U \qquad -(T) = -T \qquad +(T) = T$$

在第二行中，$T \neq -U$ 在所有情况下。

图 3：会话类型和协议类型上的类型规范化及辅助元函数

这里有一个正向范式逐步计算的一个示例：对偶 $(?(-\text{整数}).\alpha)$.

nrm^+(对偶 $(?(-\text{整数}).\alpha)$)=$\mathrm{nrm}^-(?(-\text{整数}).\alpha)$ 负向归一化会记住待处理的对偶= $\S(+(\mathrm{nrm}^+(-\text{整数}))).\mathrm{nrm}^-(\alpha)$ $\mathrm{nrm}^+(\,)$ 作用于负载，$\mathrm{nrm}^-(\,)$ 作用于主干= $\S(+(-(\mathrm{nrm}^+(\text{整数}))))$.对偶 α 负向归一化会在类型变量上重构对偶= $\S(-\text{整数})$。对偶 α 将范式具体化后应用.....= ?整数。对偶 α .....确定方向

为检验类型等价目标 $\Delta \vdash T \equiv U : \kappa$ 是否成立，我们计算算法类型等价，记作 $T \equiv_A U$，它通过比较范式来定义：$\mathrm{nrm}^+(T) = \mathrm{nrm}^+(U)$，模 α-等价。以下命题确立了规范化算法的可靠性与完备性，以及其最坏情况下的运行时间。

定理 1（可靠性）。设 $\Delta \vdash T \Rightarrow \kappa$ 和 $\Delta \vdash U \Rightarrow \kappa$。

1. 若$\mathrm{nrm}^+(T) =_\alpha \mathrm{nrm}^+(U)$，则 $\Delta \vdash T \equiv U : \kappa$。*2.* 若 $\kappa = \mathbf{s}$ 和$\mathrm{nrm}^-(T) =_\alpha \mathrm{nrm}^-(U)$，则 $\Delta \vdash T \equiv U : \mathbf{s}$。

定理 2（完备性）。设 $\Delta \vdash T \equiv U : \kappa$。

1. $\mathrm{nrm}^+(T) =_\alpha \mathrm{nrm}^+(U)$。若 $\kappa = \mathbf{s}$，则 $\mathrm{nrm}^-(T) =_\alpha \mathrm{nrm}^-(U)$。

Types for constants $\boxed{\text{typeof}(c) = T}$

$$\text{typeof}(*) = \text{单位类型} \qquad \text{typeof}(\text{分叉}) = (\text{单位类型} \to \text{单位类型}) \to \text{单位类型}$$

$$\text{typeof}(\texttt{new}) = \forall\alpha{:}\,\mathbf{s}.\alpha \otimes \text{对偶}\ \alpha \qquad \text{typeof}(\text{终止}) = \text{结束}_! \to \text{单位类型}$$

$$\text{typeof}(\text{接收}) = \forall\alpha{:}\,\mathbf{t}.\forall\beta{:}\,\mathbf{s}.?\alpha.\beta \to \alpha \otimes \beta \qquad \text{typeof}(\text{等待}) = \text{结束}_? \to \text{单位类型}$$

$$\text{typeof}(\text{发送}) = \forall\alpha{:}\,\mathbf{t}.\forall\beta{:}\,\mathbf{s}.\alpha \to !\alpha.\beta \to \beta$$

$$\text{typeof}(\text{选择}\ C_k) = \forall\alpha{:}\overline{\mathbf{p}}.\forall\beta{:}\,\mathbf{s}.!(\rho\ \alpha).\beta \to \S(+(T_k))\overline{.\beta}\ \text{if}\ \text{协议}\ \ \rho\ \alpha = \{C_i\ T_i\}_{i\in I}\ \text{和}\ \overline{k} \in I$$

图4：常量的类型

定理3（类型等价的时间复杂度）。类型的测试 $T \equiv_A U$ 运行时间为 $O(|T| + |U|)$。

4 表达式与进程

本节介绍表达式与进程的概念、表达式的算法式类型推导，以及基于带标签迁移关系的表达式与进程操作语义。常量 c、值 v、表达式 e 和进程 p 的语法由以下文法定义。

$$\begin{array}{rcl}
c & ::= & * \mid \texttt{fork} \mid \texttt{new} \mid \texttt{receive} \mid \texttt{send} \mid \texttt{select}\,C \mid \texttt{wait} \mid \texttt{terminate} \\
v & ::= & c \mid x \mid \lambda x{:}\,T.e \mid \texttt{rec}\ x{:}\,T.v \mid \Lambda\,\alpha{:}\,\kappa.v \mid \langle v, v\rangle \mid \texttt{rec}\ x{:}\,T.v \mid \texttt{receive}[T] \\
 & & \texttt{receive}[T][T] \mid \texttt{send}[T] \mid \texttt{send}[T][T] \mid \texttt{send}[T][T]\,v \mid \texttt{select}\,C[\overline{T}] \\
e & ::= & v \mid e\,e \mid e[T] \mid \texttt{let}\,* = e\,\texttt{in}\,e \mid \langle e, e\rangle \mid \texttt{let}\,\langle x, x\rangle = e\,\texttt{in}\,e \mid \\
 & & \texttt{match}\ e\ \texttt{with}\ \{C_i\ x_i \to e_i\}_{i\in I} \\
p & ::= & \langle e\rangle \mid p \mid p \mid (\nu xy)\,p
\end{array}$$

所有构造函数都是线性函数式语言或会话类型语言中的标准构造。从函数式语言中，我们有函数引入和消去（$\lambda x{:}\ T.e$ 和 $e_1\,e_2$）、线性对的引入和消去（$\langle e_1, e_2\rangle$ 和 $\texttt{let}\,\langle x, y\rangle = e_1\,\texttt{in}\ e_2$）、线性单位类型的引入和消去（* 和 $\texttt{let}\,* = e_1\,\texttt{in}\,e_2$）。从会话语言中，我们可以找到大多数以常量形式出现的构造函数，包括用于生成新线程的分叉、用于创建新通道的 new、用于在给定通道上接收值的接收、用于在给定通道上发送值的发送、用于等待通道关闭的等待、用于强制关闭通道的终止，以及 $\texttt{select}\,C$，用于在给定通道上执行一种选择，该选择由选择器名称 C 指定。用于在某个通道上提供选择的运算符无法用我们类型语言中的类型表示，因此我们将其设为表达式（匹配 e 与 $\{C_i\ x_i \to e_i\}_{i\in I}$）。在进程层面，我们有作为线程的表达式$\langle e\rangle$、并行组合 $p_1 \mid p_2$，以及通过声明通道的两个端点 x、y 来引入通道：$(\nu xy)\ p$。

表达式类型检查相对于（线性）函数式或会话类型语言而言再次基本遵循标准方式。类型检查的定义将一组协议声明作为隐式参数。对 选择 C （图4）和 匹配 （图5， E-Match）的类型检查会访问这些协议声明。图4中的 typeof 运算符返回范式中常量的类型。表达式的类型规则定义于图5中。类型系统是算法化的，由两个相互

Type synthesis $\boxed{\Delta \mid \Gamma \vdash e \Rightarrow T \mid \Gamma}$

$$\textsc{E-Const}\quad \Delta \mid \Gamma \vdash c \Rightarrow \mathrm{typeof}(c) \mid \Gamma$$

$$\textsc{E-Var}\quad \frac{\Delta \vdash T \Leftarrow \mathbf{t}}{\Delta \mid \Gamma, x\colon T \vdash x \Rightarrow T \mid \Gamma}$$

$$\textsc{E-Var}\star\quad \frac{\Delta \vdash T \Leftarrow \mathbf{t}}{\Delta \mid \Gamma, x{:}^\star T \vdash x \Rightarrow T \mid \Gamma, x{:}^\star T}$$

$$\textsc{E-Let}^*\quad \frac{\Delta \mid \Gamma_1 \vdash e_1 \Leftarrow \mathtt{Unit} \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 \Rightarrow T \mid \Gamma_3}{\Delta \mid \Gamma_1 \vdash \mathtt{let}\ * = e_1\ \mathtt{in}\ e_2 \Rightarrow T \mid \Gamma_3}$$

$$\textsc{E-Abs}\quad \frac{\Delta \vdash T \Leftarrow \mathbf{t} \qquad \mathrm{nrm}^+(T) = V \qquad \Delta \mid \Gamma_1, x\colon V \vdash e \Rightarrow U \mid \Gamma_2 \qquad x \notin \Gamma_2}{\Delta \mid \Gamma_1 \vdash \lambda x\colon T.e \Rightarrow V \to U \mid \Gamma_2}$$

$$\textsc{E-App}\quad \frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow T \to U \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 \Leftarrow T \mid \Gamma_3}{\Delta \mid \Gamma_1 \vdash e_1\, e_2 \Rightarrow U \mid \Gamma_3}$$

$$\textsc{E-Pair}\quad \frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow T \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 \Rightarrow U \mid \Gamma_3}{\Delta \mid \Gamma_1 \vdash \langle e_1, e_2 \rangle \Rightarrow T \otimes U \mid \Gamma_3}$$

$$\textsc{E-Rec}\quad \frac{\Delta \vdash T \Leftarrow \mathbf{t} \qquad \mathrm{nrm}^+(T \to U) = V \qquad \Delta \mid \Gamma, x{:}^\star V \vdash v \Leftarrow V \mid \Gamma, x{:}^\star V}{\Delta \mid \Gamma \vdash \mathtt{rec}\ x\colon T \to U.v \Rightarrow V \mid \Gamma}$$

$$\textsc{E-TAbs}\quad \frac{\Delta, \alpha\colon \kappa \mid \Gamma_1 \vdash v \Rightarrow T \mid \Gamma_2}{\Delta \mid \Gamma_1 \vdash \Lambda\, \alpha\colon \kappa.v \Rightarrow \forall \alpha\colon \kappa.T \mid \Gamma_2}$$

$$\textsc{E-TApp}\quad \frac{\Delta \mid \Gamma_1 \vdash e \Rightarrow \forall \alpha\colon \kappa.U \mid \Gamma_2 \qquad \Delta \vdash T \Leftarrow \kappa}{\Delta \mid \Gamma_1 \vdash e[T] \Rightarrow \mathrm{nrm}^+(U[T/\alpha]) \mid \Gamma_2}$$

$$\textsc{E-Let}\quad \frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow T \otimes U \mid \Gamma_2 \qquad \Delta \mid \Gamma_2, x\colon \mathrm{nrm}^+(T), y\colon \mathrm{nrm}^+(U) \vdash e_2 \Rightarrow V \mid \Gamma_3 \qquad x, y \notin \Gamma_3}{\Delta \mid \Gamma_1 \vdash \mathtt{let}\ \langle x, y \rangle = e_1\ \mathtt{in}\ e_2 \Rightarrow V \mid \Gamma_3}$$

$$\textsc{E-Match}\quad \frac{\begin{array}{c}\mathtt{protocol}\ \rho\ \overline{\alpha} = \{C_i\ \overline{T_i}\}_{i \in I} \qquad \Delta \mid \Gamma_1 \vdash e \Rightarrow ?(\rho\ \overline{U}).S \mid \Gamma_2 \\ \forall i \in I: \qquad \Delta \mid \Gamma_2, x_i\colon \S(-(\overline{T_i}[\overline{U}/\overline{\alpha}])).S \vdash e_i \Rightarrow V_i \mid \Gamma_i \qquad x_i \notin \Gamma_i \qquad V =_\alpha V_i \qquad \Gamma_3 =_\alpha \Gamma_i\end{array}}{\Delta \mid \Gamma_1 \vdash \mathtt{match}\ e\ \mathtt{with}\ \{C_i\ x_i \to e_i\}_{i \in I} \Rightarrow V \mid \Gamma_3}$$

Type against $\boxed{\Delta \mid \Gamma \vdash e \Leftarrow T \mid \Gamma}$

$$\textsc{E-Check}\quad \frac{\Delta \mid \Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2 \qquad T =_\alpha U}{\Delta \mid \Gamma_1 \vdash e \Leftarrow T \mid \Gamma_2}$$

图 5：表达式的算法类型规则

递归判断式：一个用于综合表达式的类型，另一个用于检查表达式是否符合某个类型。类型规则依赖于第 3 节中引入的类型变量上下文 Δ 、为项变量分配类型的项变量上下文 Γ 分配类型 T 给项变量 x。项变量上下文中有两种条目：线性条目，记作 $x\colon T$，以及非受限条目，记作 $x{:}^\star T$。后者仅用于递归函数（参见规则 E-Var$\star$ 和 E-Rec），其灵感来自 Atkey [2018], 、Bernardy 等人 [2018], 和 McBride [2016]关于定量类型论的研究。

类型系统维护这样一个不变量：项变量上下文中的条目始终是归一化的：写入上下文的规则，即 E-Abs、 E-Rec 和 E-Let，会对新条目进行归一化。判断式的形式为 $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$ 或 $\Delta \mid \Gamma_1 \vdash e \Leftarrow T \mid \Gamma_2$，其中 Γ_2 表示上下文 Γ_1 Walker [2005] 中未被 e 消耗的部分，而 T 处于范式中。公理（E-Const 和 E-Var）主要将输入上下文复制到输出中。规则 E-App 是上下文传递的一个很好的例子：为了对表达式 $e_1\ e_2$进行类型判定，将表达式 e_1 赋予输入上下文 Γ_1，产生 Γ_2，再将其传递给 e_2；后者进而产生 Γ_3，构成应用 $e_1\ e_2$的输出上下文。向上下文添加线性条目的规则（E-Abs、 E-Let 和E-Match）都会检查新变量是否不在输出上下文中，从而确保相应的值已被完全消耗。规则 E-TApp 会显式地对输出类型进行归一化，因为归一化不会通过替换得到保持；所有在上下文中引入新条目的规则（E-Abs、 E-Rec、E-Let和 E-Match）都会确保类型是归一化的，从而保证上下文只包含归一化的类型。

规则 E-Rec 和 E-Var$\star$ 规定了递归函数的类型判断。规则 E-Rec 通过非线性绑定 $x{:}^\star$ 绑定递归函数 x，因此可以在函数体中通过 E-Var$\star$进行任意次数的递归调用。非线性绑定保证递归函数能够终止。这类函数不能使用环境中的线性变量；为此，我们要求传入上下文 Γ 与传出上下文相等。

真正全新的规则是 E-Match，它表明所有规则都相对于一组隐式的协议声明而言是参数化的：对于每个在 Δ中具有种类判断的协议 ρ，都有一个协议声明。该规则首先综合待匹配表达式的输入类型；对于某个协议构造子 ρ，其类型为 $?(\rho\ U).S$。此时，会在隐式声明集合中查找协议 ρ 的声明。随后依次对选择器 C_i 的每种选择所对应的匹配分支进行类型判断。协议的参数 α 用参数 U 实例化，从而生成类型序列 $T_i[U/\alpha]$。如第 3 节所述，该类型序列会转换为规范化的会话类型，并将其赋给变量 $\overline{x_i}$。现在，每个分支都会为 $i \in I$综合出类型V_i 和上下文 Γ_i。在所有分支中，输出类型和上下文必须分别与匹配表达式的输出类型和上下文一致（在 α 等价意义下），即分别与 V 和 Γ_3一致。实际上，实现会选择一个分支，例如 $k \in I$，并将其他所有输出类型和上下文分别与 V_k 和 Γ_k进行比较。在此规则中，以及图4中 select 的类型规则中，我们通过如下映射，将方向运算符和具体化（来自图 3）扩展到参数序列： $\pm(\overline{T}) = \pm(T)$，以及 $\S(\varepsilon).\overline{S} = S$和$\S(TT).S = \S(T).\S(T).S$。

规则 E-Check 依赖先前的判断，检查表达式 e 是否符合类型 T，至多相差α-等价。这种简单的比较已足够，因为合成会输出范式形式的 U，而检查必须以范式形式的 T 作为输入。

表达式和进程的操作语义通过一个迁移关系定义，该关系由以下动作标记。

σ	$::= \quad x?v \mid x!v \mid x?C \mid x!C \mid x? \mid x!$	会话操作的标签
λ	$::= \quad \sigma \mid \beta \mid \texttt{fork}\, v \mid (\nu xx) : T$	表达式的标签
π	$::= \quad \sigma \mid \tau \mid (\nu xx)x!x \mid (\pi \mid \pi)$	进程的标签

这些标签表示会话上的六种操作：接收、发送、匹配、选择、等待和终止。标签 λ 用于表达式：β 表示内部动作，其余两个表示分叉和新建。标签 π 用于进程： τ 表示静默动作（对应于表达式中的 β），$(\nu ab)x!a$ 或 $(\nu ab)x!b$ 表示打开 (νab) 绑定的作用域，而 $\pi \mid \pi$ 表示并行组合。

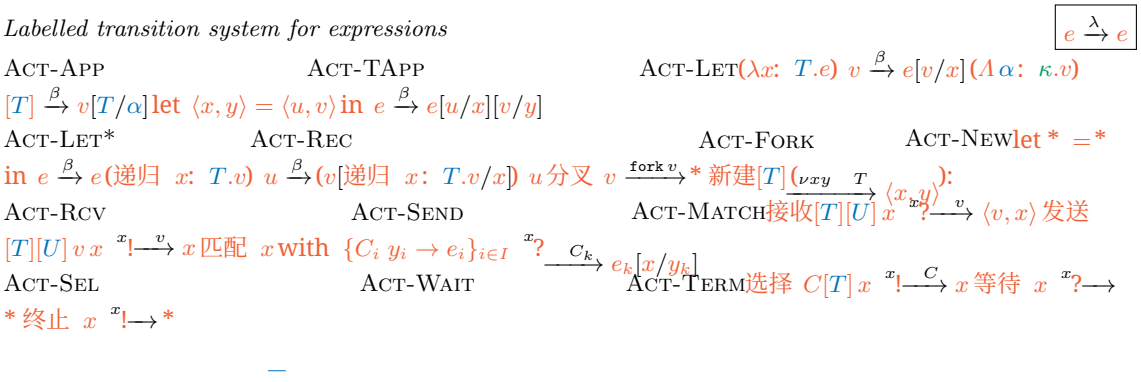


图 6：表达式的标记转换系统（所选规则）。完整版本包括按值调用的结构规则，见补充材料 D，图 11

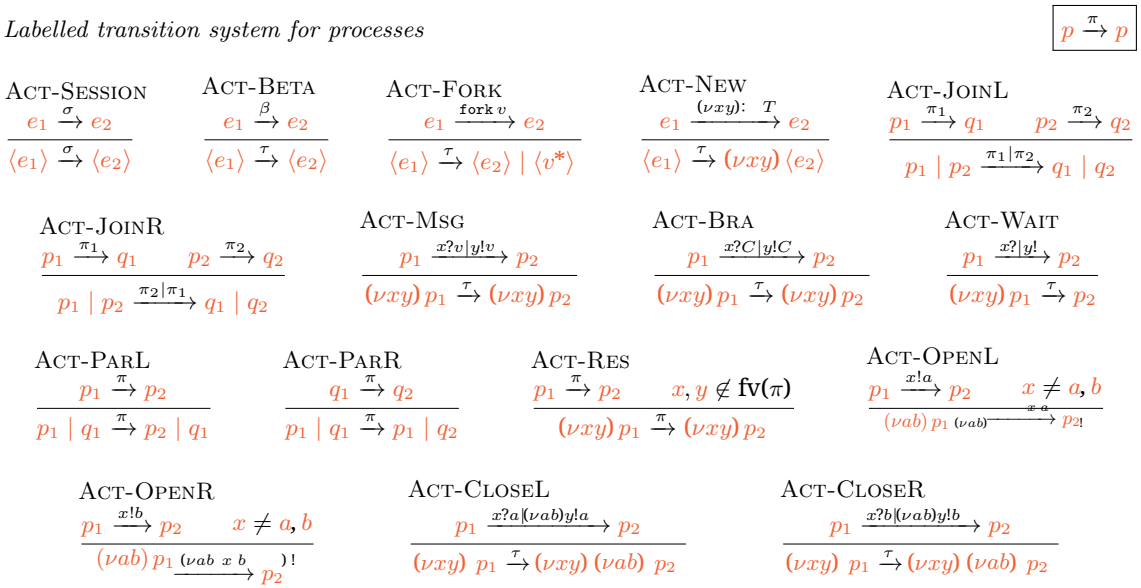


图 7：进程的标记转换系统

表达式的标记转换系统的公理由图 6 中的规则定义（补充材料 D的图 11包含完整的规则集）。我们将多态λ演算的通用归约规则归入标签 β 下。Act-Fork和 Act-New 在标签中记录分叉出的值以及所创建的信道端点的名称和类型。随后，这两个转换将在进程层面进行处理。接下来是六个通用会话操作的转换。

Typing rules for processes $\boxed{\Gamma \vdash p}$

$$\text{P-Exp}\ \frac{\cdot \mid \Gamma \vdash e \Leftarrow \text{单位类型} \mid \cdot}{\Gamma \vdash \langle e \rangle} \qquad \text{P-Par}\ \frac{\Gamma_1 \vdash p_1 \qquad \Gamma_2 \vdash p_2}{\Gamma_1,\ \Gamma_2 \vdash p_1 \mid p_2} \qquad \text{P-New}\ \frac{\cdot \vdash T \Leftarrow \textbf{秒} \qquad \Gamma,\ x\colon\ \mathrm{nrm}^{+}\ (T)\ \ y\colon\ \mathrm{nrm}^{-}\ (T) \vdash p}{\Gamma \vdash\ (\nu xy)\ p}$$

图 8：进程类型规则

进程的标记转换系统由图7 中的规则定义。这些规则改编自 Fowler 等人 [2021a,b]，而后者又受到Montesi 和 Peressotti [2021] 的启发。前四条规则将表达式转换转换为进程转换：会话迁移σ 直接从表达式传递到进程（规则Act-Session）。接下来的三条规则都会产生静默（τ）转换：Act-Fork启动一个新线程，而 Act-New 创建一个新通道。规则Act-JoinL 和Act-JoinR 说明了并行组合的交换性质。规则Act-ParL、Act-ParR 和 Act-Res 允许在并行组合和作用域限制之下进行转换。用于打开和关闭作用域的规则（左版本和右版本）改编自 Milner 等人 [1992] 针对 π-演算的标记转换系统中的双绑定子方案。之所以需要这些规则，是因为我们使用自由输出，这不同于上述引用的研究：Fowler 等人[2021a,b], 以及 Montesi 和 Peressotti [2021] 使用的是绑定输出。

只有当前缀下的约简（规则 Act-Res）中的信道端点 x 和 y （两个绑定变量）不会捕获标签中的自由变量时，才能发生约简。为此，我们将进程标签 π 的自由变量定义如下。

$$\mathrm{fv}(x?v) = \mathrm{fv}(x!v) = \{x\} \cup \mathrm{fv}(v) \qquad \mathrm{fv}(x?C) = \mathrm{fv}(x!C) = \mathrm{fv}(x?) = \mathrm{fv}(x!) = \{x\}$$
$$\mathrm{fv}(\tau) = \emptyset \qquad \mathrm{fv}((\nu wx)y!z) = \{y,z\} \setminus \{w,x\} \qquad \mathrm{fv}(\pi \mid \pi') = \mathrm{fv}(\pi) \cup \mathrm{fv}(\pi')$$

我们在本节中给出该语言的主要结果。为此，我们需要讨论上下文和进程类型判断。类型形成（图 1）逐点提升到上下文，得到判断 $\Delta \vdash \Gamma \Leftarrow \kappa$。进程类型规则见图 8。线程规则要求表达式 e 消耗上下文 Γ中给出的全部资源。规则 P-Par 将上下文拆分为两部分，并分别分配给两个并行进程。该规则不是算法化的，不过我们计划只对表达式进行类型检查；进程是运行时产物。规则 P-New 猜测信道端点 T 的类型 x，然后信道端点 y 获得类型 对偶 T，两者均为范式。同样，这条规则不是算法化的，也无需如此。带标签的转换通过其标签传达行为。由于我们使用带剩余项的算法化类型系统，因此需要为类型上下文建立一个标记转换系统，以说明标签所表示的行为对上下文的影响。保持性的论证依赖于补充材料 D 中定义的上下文标记转换系统（图 12）。

定理 4（保持性）。

1. 如果 $e_1 \xrightarrow{\lambda} e_2$ 且 $\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \Rightarrow T \mid \Gamma_3$ 且 $\Gamma_2 \xrightarrow{\lambda} \Gamma_2'$ 且 $\Delta \vdash \Gamma_1, \Gamma_2, \Gamma_3 \Leftarrow t$，则 $\Delta \mid \Gamma_1, \Gamma_2', \Gamma_3 \vdash e_2 \Rightarrow T' \mid \Gamma_3$ 且 $T' \equiv_A T$. *2.* 如果 $e_1 \xrightarrow{\lambda} e_2$ 且 $\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \Leftarrow T \mid \Gamma_3$ 且 $\Gamma_2 \xrightarrow{\lambda} \Gamma_2'$ 且 $\Delta \vdash \Gamma_1, \Gamma_2, \Gamma_3 \Leftarrow t$，则$\Delta \mid \Gamma_1, \Gamma_2', \Gamma_3 \vdash e_2 \Leftarrow T \mid \Gamma_3$.*3.*如果 $p_1 \xrightarrow{\pi} p_2$ 且 $\Gamma_1, \Gamma_2 \vdash p_1$ 且 $\Gamma_2 \xrightarrow{\pi} \Gamma_2'$ 且 $\cdot \vdash \Gamma_1, \Gamma_2 \Leftarrow t$，则 $\Gamma_1, \Gamma_2' \vdash p_2$.

我们将下面的标签集合称为进展集用于 x， y：

$$\{\tau,\ (x?v \mid y!v),\ (x?C \mid y!C),\ (x? \mid y!),\ (x?a \mid (\nu ab)y!a),\ (x?b \mid (\nu ab)y!b)\}$$

已完成的进程 具有以下形式 $\langle * \rangle \mid \ldots \mid \langle * \rangle$。如果一个进程满足以下条件，则它是 死锁：对于其所有形如 $(\nu xy)\,p$ 的子项，以及所有转换 $p \xrightarrow{\pi} p'$，均有 π 不属于通道 x 的 y。要识别死锁进程，我们找出它的所有 ν 子项，并找出每个子项的所有转换。然后，如果所有转换都不在进展集中，该进程就是死锁的。当种类上下文 Δ 可从上下文中看出时，我们用 $\Gamma^{\mathbf{s}}$ 表示仅由会话类型组成的类型上下文，即满足以下条件的上下文 $x\colon\ T \in \Gamma$ 蕴含 $\Delta \vdash T \Leftarrow \mathbf{s}$。

定理 5（可能导致死锁的进展）。

1. 若 $\Delta \mid \Gamma_1^{\mathbf{s}} \vdash e_1 \Rightarrow T \mid \Gamma_2$，则要么 e_1 是一个值，要么 $e_1 \xrightarrow{\lambda} e_2$。
2. 若 $\Delta \mid \Gamma_1^{\mathbf{s}} \vdash e_1 \Leftarrow T \mid \Gamma_2$，则要么 e_1 是一个值，要么 $e_1 \xrightarrow{\lambda} e_2$。
3. 若 $\Gamma^{\mathbf{s}} \vdash p_1$，则 p_1 已完成，$p_1 \xrightarrow{\pi} p_2$，或者 p_1 陷入死锁。

5 实现

AlgST 的实现由一个类型检查器和一个解释器组成，两者均使用 Haskell 编写。为了支持更贴近实际的程序，该实现加入了一些实用扩展，包括对形式系统的扩展，以及一个用于命名空间管理的简单模块系统。

类型检查 该实现的语法与本文中的示例基本一致。有些地方需要额外的种类注释，因为该实现的类型系统并不限于线性类型。其种类结构区分了线性种类和非限制种类 $\mathbf{s}$ 和 $\mathbf{t}$。子种类关系允许我们将类型的非限制版本提升为线性版本，即 $\mathbf{t}^{\mathbf{un}} <: \mathbf{t}^{\mathbf{lin}}$。这种子种类关系形成了一种简单的子类型关系，实质上允许我们在需要线性值的地方提供非限制值。协议种类 $\mathbf{p}$ 不拆分为两个版本，但仍然可以将任何类型提升为协议类型。Almeida 等人 [2022] 研究了一个具有类似种类结构的系统的元理论（但不包含种类 $\mathbf{p}$）。

类型检查器的实现紧密遵循论文中给出的双向规则。它必然包含图 3 所示的类型规范化算法的实现。该算法经过扩展，可处理类型同构 $\forall\alpha.(T \to U) \cong T \to \forall\alpha.U$ (if α not free in T)。这样，只要保留类型应用的顺序，就可以更自由地放置类型应用。

规范化算法严格按照图 5 中算法化类型规则的指示调用。规则经过精心设计，可使类型假设和推理结果保持范式，从而尽量减少调用规范化的次数。

类型检查器还包含更多检查规则，以便在不添加类型注释的情况下编写某些 lambda 抽象。这些规则改编自 Dunfield 和 Krishnaswami[2021] 的知名规则。例如：

$$\text{E-Abs'}\quad \frac{\Delta \mid \Gamma_1, x\colon T \vdash e \Leftarrow U \mid \Gamma_2 \qquad x \notin \Gamma_2}{\Delta \mid \Gamma_1 \vdash \lambda x.e \Leftarrow T \to U \mid \Gamma_2}$$

$$\text{E-App'}\quad \frac{\Delta \mid \Gamma_1 \vdash e_2 \Rightarrow T \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_1 \Leftarrow T \to U \mid \Gamma_3}{\Delta \mid \Gamma_1 \vdash e_1\ e_2 \Leftarrow U \mid \Gamma_3}$$

```
--- AlgST 语法中的协议和类型 --- 协议 Repeat x = 更多 x ( Repeat x) | 退出 ?
Repeat 整数 . !( 字符 , 结束 ) . 结束 --- FreeST 语法中的对应类型 --- ( rec
repeat0 : 1S . &{ 更多 : ? 整数 ; repeat0 ; 跳过 , 退出 : 跳过 }) ; (!( 字符 , 结束 ) ;
结束 ) --- 等价 AlgST 类型的示例 --- 对偶 (! Repeat 整数 . ?( 字符 , 结束) . 对偶 结
束) --- 非等价 AlgST 类型的示例 --- ? Repeat 字符串 . !( 字符 , 结束) . 结束
```

图 9：一个 AlgST 类型实例、其 FreeST 对应形式，以及遵循测试套件生成器规则的等价与非等价 AlgST 类型示例。

实现全面支持将数据构造函数重载用作协议运算符。我们可以针对 `List` 数据类型的构造函数进行模式匹配，并在为 `!List` 协议选择分支时使用相同的构造函数。此外，`case` 运算符也被重载为`match` 运算符。

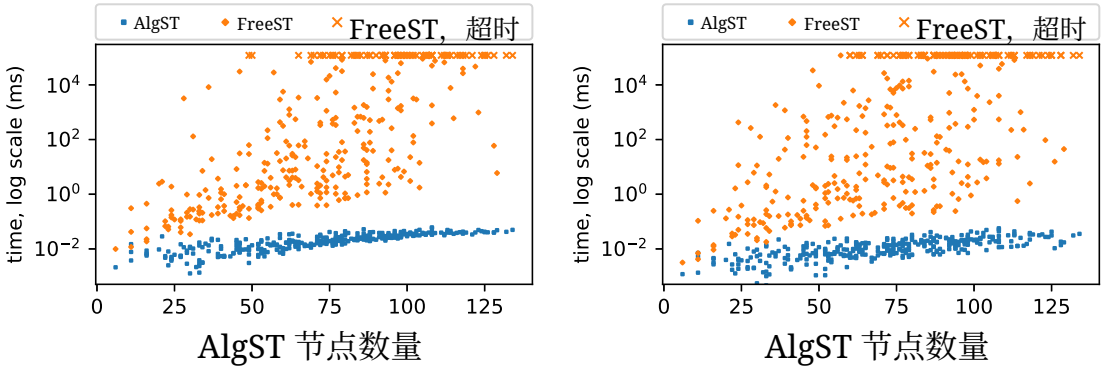
解释器 解释器采用标准技术处理函数式部分，并在运行时表示值。它使用一个全称类型，其中包含语言中每种类型对应的构造函数。AlgST 中的进程会映射为 Haskell 线程。语言中的分叉通过 Haskell 的 forkIO 实现。新线程会递归调用解释器。

通信信道通过 Haskell 的 Control.Concurrent 库提供的共享内存抽象来实现。同步通信（如本文所述）通过一对 MVar 实现；MVar 是一种类似信号量的并发数据结构，Jones 等人 [1996] 指出，它的行为类似于容量为 1 的缓冲区。该实现还提供一个选项，可使用 stm 库中的有界队列（TBQueue）切换为异步通道。

由于通道是在共享内存中实现的，发送 原语自然是多态的。在内部，解释器会在发送 和接收 操作之间传递上述全称类型的值。

基准测试 我们通过一项实验来证实这一主张：将类型等价的最坏情况下超线性算法替换为线性算法，确实能提升实际运行时性能。我们将 AlgST 的类型等价算法与 FreeST 进行比较。FreeST 是一款可自由获取的编程语言，其中实现了多态上下文无关会话类型的类型等价算法 Almeida 等人 [2020, 2022]。由于类型检查器经常需要进行类型等价检查，因此应将等价算法的性能视为类型检查器性能的下界。

为了创建一组等价和不等价的测试用例，我们实现了一个 AlgST 类型实例生成器。一个实例由一组相互递归的代数协议以及一个引用这些协议的会话类型组成。我们谨慎地限制协议和类型，从而能够将AlgST 实例转换为FreeST类型：该生成器避免多态递归和嵌套递归，并将取负运算符的出现限制在协议构造子参数的顶层。对于每个实例 T，我们随机应用规范化的性质，以生成一个等价的 AlgST类型 T'（见图 3）。这些对（T，T'）构成了我们的等价测试套件



(a) 等价测试用例 （b）不等价测试用例

图 10：AlgST 和 FreeST 类型等价算法在等价与不等价测试用例上的执行时间，按 AlgST 抽象语法树中的节点数绘制，并以对数尺度表示。

类型。不等价测试通过对每个 T 引入额外的量词，或将类型的某个子部分更改为同类的其他内容来获得。可能的替换项包括带量化的类型变量以及 AlgST 中内置的类型，例如 整数 或 字符串。AlgST 类型会被转换为FreeST 中的会话类型。协议在每个使用点以内联方式转换为递归分支类型或选择类型，具体取决于它出现在发送上下文还是接收上下文中。对于单构造函数类型，转换会省略构造器标签。构造函数的参数会被转换为由单个交互组成的嵌套序列。示例见图 9。

基准测试使用 gauge 软件包 v0.2.5 O'Sullivan [（最后访问于 2023] 年 3 月）进行，超时时间为 2 分钟。AlgST 使用 GHC 9.2.7 编译，FreeST 使用 GHC 8.10.7 编译。每个测试套件包含 324 项测试。FreeST 的正向测试发生了 69 次超时，负向测试发生了 77 次超时。图 10 对比了 AlgST 和 FreeST 类型等价算法在等价与不等价测试用例上的执行时间。这些图表证实了 FreeST 与 AlgST 在执行时间上的差异，并展示了随着 AlgST 类型抽象语法树中的节点数增加，AlgST 的执行时间呈线性增长，而 FreeST 则呈指数级增长。

6 相关工作

这篇相关工作综述聚焦于二元会话类型及其对递归的处理。该领域的许多研究侧重于不同方面，并将递归非正式地视为附加功能。特别是，Honda [1993], Takeuchi 等人 [1994]的早期工作并未考虑递归类型。以对偶直觉主义逻辑的序列演算为基础的会话类型学派 Caires and Pfenning [2010], Caires 等人 [2016] 最初关注有限行为。尽管递归会破坏其系统的逻辑一致性，他们仍考虑了包含复制的依赖类型变体 Pfenning et al. [2011]。

带递归的会话类型系统。一般而言，递归类型分为两种：等递归和同构递归 Abadi 和 Fiore [1996], ，这两种方法对类型

检查的影响已得到充分研究 Pierce [2002]。研究发现，这两种方法的表达能力相同 Patrignani 等人 [2021]。文献中的大多数会话类型系统都支持仅限于尾递归的等递归会话类型，并以非形式化方式处理它们。

递归类型最早由 Honda 等人 [1998] 引入会话类型领域，背景是 π-演算。他们的系统依赖等递归，并在语法上限制为尾递归。Gay 和 Hole [1999, 2005] 在相同设定下通过余归纳定义研究子类型关系。Castagna 等人 [2009] 将递归会话类型视为无限正则树。他们的类型没有标签，因为选择是依据所接受的值集合以及后续过程的可行性作出的。Gay 和 Vasconcelos [2010] 的函数式会话类型演算包含子类型关系和等递归类型，但后者仅限于尾递归会话类型。《会话类型基础》Vasconcelos [2012] 提出了一系列基于 π-演算、复杂度逐步提升的系统。其中一个重点是如何处理复制和不受限制的信道端点，这需要递归类型。在本研究中，递归类型被建模为无限正则树，并使用熟悉的 μ 记法表示，其主体为收缩的尾递归体。

Toninho 等人 [2013]扩展了基于线性逻辑的研究方向，并聚焦于函数与进程的结合，包括递归进程，以及由此产生的进程和函数的递归类型。示例表明其采用等递归变体。显然，递归的使用没有限制，并且顺序组合通过张量 $\otimes$实现。Toninho 等人 [2014]探讨了等递归的另一个方面。他们研究余递归协议定义，并分析其生产性。他们关注的主要问题是：余递归协议何时能在有限次内部步骤后产生下一个值。SILL Pfenning 和 Griffith [2015]是一种基于进程的语言设计，融合了线性类型和仿射类型，以及同步和异步通信。它直接建立在线性逻辑对会话类型的诠释之上。其示例使用了多态和等递归类型，形式为尾递归类型方程，但类型理论表述并未体现这两者。Bartoletti 等人 [2014] 基于标记转移系统对会话类型进行语义分析。他们的系统支持尾位置上的等递归。

Lindley 和 Morris [2016] 定义了两种带递归的函数式会话类型系统—— μGV 和μCP，并证明它们之间存在保类型且保语义的翻译。因此，以下我们只将其与 μGV 进行比较。 μGV 基于简单类型线性 λ 演算，并扩展了递归和余递归的概念。这些概念分别定义为严格正函子的最小不动点和最大不动点，类似于具有逻辑意义的代数（余）数据类型，并配有通用的折叠算子和展开算子。最后，他们将这种对递归数据的同构递归处理提升到会话类型层面：通过编码分支和递归，将相应的数据函子映射到会话类型上，并用承诺将其封装起来。我们得出结论， μGV 具有支持同构递归的会话类型，可用于编码 AlgST 的简单类型片段，但不支持参数化数据类型，也不支持 over 方向。 μGV 使用承诺（即独立的通道）来编码递归类型折叠操作的参数（即代数数据类型的构造器参数）；而 AlgST 则在同一通道上传输折叠操作的主体，从而避免了信道创建的开销。此外，μGV 主要是一项理论贡献，而 AlgST 已有完整实现。

上下文无关和嵌套会话类型系统。 Thiemann和Vasconcelos [2016] 提出了上下文无关会话类型，以支持对非正则协议进行类型安全编码，例如树结构的序列化，或引言中提到的栈协议，同时避免会话启动带来的开销。他们将传统会话类型语法的不对称结构 $?T.S$ 和$!T.S$（其中传输一个类型为 T 的项，随后会话继续为 S）改为对称的

结构，其中会话类型可以由原子传输 $?T$ 和 $!T$、选择和分支 以及 $S_1 \,\&\, S_2$构成，并配有一个带单位元的幺半群顺序组合算子 $;S_2$，其单位元为跳过。这种设计结合等递归会话，引发了若干技术难题。首先，类型具有非平凡的等式理论：顺序组合构成一个以跳过 为单位元的幺半群，并且可分配于选择和分支之上。其次，需要多态递归，因为递归调用可能需要以不同的后续会话进行实例化。第三，等递归与不再限制为尾递归之间的相互作用，导致了一个困难但可判定的类型等价关系。后续研究探讨了更易处理的类型等价算法。Almeida 等人 [2019, 2020]。Almeida 等人 [2022] 将上下文无关会话类型与 System F 风格的显式多态相结合。近期工作通过高阶类型扩展了这一方法。Costa 等人 [2022]。

Padovani [2017, 2019] 提出了一种不同的上下文无关会话类型方法，该方法体现在一种名为 FuSe$^{\{\}}$ 的函数式会话演算中。该演算采用上下文无关会话类型的类型语言，但通过引入恢复运算符 $\{e\}_c$，绕开了顺序组合运算符和类型等价方面的问题。该运算符会分解通道 c 上的一系列会话：$S_1; S_2$。它通过将通道重新定型为 $c : S_1$ 来实现这一点，以便求值 e；求值完成后，必须返回同一通道 S_1。更形式化地说，恢复操作遵循某种框架规则：若 $c : S_1 \vdash e : T \times \mathtt{1}$，则 $c : S_1; S_2 \vdash \{e\}_c : T \times S_2$。恢复操作会暂时从会话类型中移除延续 S_2，然后再将其接回。类型系统能够确保 e 返回的通道与 c 相同，这是可靠性所必需的。类型 $\mathtt{1}$ 并不表示会话结束，而是表示会话中某个已知部分已处理完毕，并将在上下文（即恢复操作中）继续。选择和分支操作的管理方式与 AlgST 非常接近。FuSe$^{\{\}}$ 通过在变体类型中传递构造器标签来实现选择，该变体类型用于选择不同的延续。其分支运算符会匹配接收到的构造器，从而选择所需的延续。然而，AlgST 对协议构造子的处理包括根据协议声明中给出的类型和方向构造延续类型。FuSe$^{\{\}}$ 没有与协议声明等价的机制，也没有用于交换传输方向的类型运算符 - 。Padovani [2019] 使用 OCaml 实现了 FuSe$^{\{\}}$，并证明，只要程序员审慎地安排恢复操作，上下文无关会话类型就可以进行类型推断。OCaml 的类型推断引擎已内置对等递归的支持。研究还表明，子类型关系是不可判定的。

Das 等人 [2021] 提出了嵌套会话类型的元理论，其中的会话类型定义带有嵌套多态递归参数。乍看之下，嵌套会话类型 $\mathtt{List}[\alpha] = \oplus\{\mathtt{nil}:\text{结束}, \mathtt{cons}: \alpha \otimes \mathtt{List}[\alpha]\}$ 与代数协议 协议 List α = Nil | Cons α (List α)非常相似，但两者存在显著差异。在代数一侧，我们可以将 List 协议具体化为一种类型，用于发送或接收整数列表；而嵌套类型 List 则已固定为一个具体方向。尽管类型嵌套能够表达代数会话所具有的模块化特征，但 AlgST 与嵌套会话类型之间仍有若干区别：Das 等人提出了结构化且等递归的类型定义观点，而非名义观点；并以余归纳地而非归纳地解释类型。这些特征显著影响嵌套会话类型等价性问题的复杂度及其实现：Das 等人提出了一种可靠但不完备的类型等价算法。通过采用代数会话类型的名义解释，我们将等价性问题的复杂度从双指数降低到线性，并设计了一种用于验证类型等价的可靠且完备的算法。Das 等人第 2 节中的示例 [2021]可转换为 AlgST，并可通过我们的实现进行类型检查。

Gay 等人 [2022] 考虑了由不同类型的方程组（有参数和无参数）定义的等递归协议。对参数化施加不同的限制，会产生不同类别的协议，其类型等价判定问题的复杂度从多项式时间到不可判定不等。他们证明，最大的一类可判定下推会话类型等价于嵌套会话类型，而我们可以用 AlgST 对其建模。

与所有其他竞争系统相比，Almeida 等人 [2022],Das 等人 [2021], Padovani [2019],Thiemann和 Vasconcelos [2020], AlgST 的具体化和方向运算符独具特色。这些运算符使我们能够构造协议，而不必预先确定特定方向。

线性类型和非线性类型。 我们的演算大体上是线性的，但允许递归函数使用非线性绑定，否则递归函数将毫无用处。我们的实现更进一步：除了确保会话类型通道按线性方式处理外，还允许功能完备的非线性计算。关于 λ 演算中的递归和线性性已有一些研究，Alves 等人 [2010] 将迭代器添加到线性自然数的演算中，从而避免了对类似 E-VAR⋆ 的规则的需求。

System F◦ Mazurak 等人 [2010] 扩展了带有种类的 System F，以整合多态与线性类型和非线性类型。这项工作为此后许多研究提供了蓝图，包括 LFST 和 FreeST（见下文），以及我们实现的语言。他们将所有对象（包括函数）归类为一次使用或多次使用的资源。这种风格可以追溯到 Walker [2005]。LFST Lindley 和 Morris [2017] 形式化了 Links 的会话类型系统；该系统融合了线性类型和非线性类型，并嵌入了一种标准的函数式编程语言。类型的次结构行为由精细的种类结构指定。我们实现中采用的种类判断受 LFST 启发，不过 LFST 还有用于描述行类型的额外种类，而我们不支持这些种类。Almeida 等人 [2022] 考虑将上下文无关会话类型与 System F 整合。他们的类型系统包含一种种类判断规则，用于管理普通类型和消息类型（发送和接收操作可能传递的负载）的线性与非线性版本。我们实现的系统以类似方式扩展了种类结构。

线性 Haskell（LH）Bernardy 等人 [2018] 提出了一种将线性类型整合到标准函数式编程中的方案。迄今讨论的工作刻画了资源，而 LH 支持 lollipop 类型 $S \multimap T$，用于分类恰好使用其参数一次的函数。最初，LH 强制程序员使用延续来分配资源。Spiwack 等人的后续工作 [2022] 已解决了这一限制。还有一些将线性类型与依赖类型整合的工作（例如 Krishnaswami 等人 [2015]），其做法是将语言划分为两个通过伴随关系连接的演算。

7 结论

代数协议将命名和递归与会话类型中常见的选择和序列类型构造子结合起来。与代数数据类型类似，它们支持参数化和相互递归。不同于以往关于会话类型的研究，代数协议不限定特定的通信方向，即使可以在类型级别指定方向反转。代数协议和会话类型涵盖了上下文无关和嵌套会话类型系统的表达能力，同时将类型检查的复杂度从双指数降至线性时间。

致谢

我们感谢审稿人提出的建议，这些建议大大促成了论文更为完善的版本。本研究获得了葡萄牙科学与技术基金会的资助，资助项目为 SafeSessions，编号 PTDC/CCI-COM/6453/2020；同时也获得了 LASIGE研究组的资助，编号 UIDB/00408/2020 和 UIDP/00408/2020，以及 COST 行动 EuroProofNet（CA20111）的资助。

软件可用性

所提出的系统及其实现已作为制品公开发布，Spaderna等人[2023]。感兴趣的读者可以按照制品文档中的步骤，设置预构建的 Docker 镜像或自行构建的 Docker 镜像。除了类型检查器和解释器的实现之外，我们还提供了论文中所有示例的程序，以及用于与相关工作进行比较的示例：Das 等人 [2021], Thiemann和Vasconcelos [2016]。读者还可以在制品中找到复现我们类型等价基准测试结果的说明。

参考文献

Mart´ın Abadi 和 Marcelo P. Fiore。关于递归类型的语法考量。载于 第*11*届年度 *IEEE*计算机科学逻辑研讨会，第242–252页，美国新泽西州新不伦瑞克，1996年。IEEE计算机学会。https://doi.org/10.1109/LICS.1996.561324。Bernardo Almeida、Andreia Mordido 和 Vasco T. Vasconcelos。Freest：函数式语言中的上下文无关会话类型。载于 Francisco Martins 和 Dominic Orchard 编，*PLACES@ETAPS 2019*，EPTCS 第291卷，第12–23页，捷克共和国布拉格，2019年。https://doi.org/10.4204/EPTCS.291.2。Bernardo Almeida、Andreia Mordido 和 Vasco T. Vasconcelos。判定上下文无关会话类型的双模拟等价。载于 Armin Biere 和 David Parker 编，*TACAS 2020*，《计算机科学讲义》第12079卷，第39–56页，爱尔兰都柏林，2020年。Springer。https://doi.org/10.1007/978-3-030-45237-7 3。Bernardo Almeida、Andreia Mordido、Peter Thiemann 和 Vasco T. Vasconcelos。带有上下文无关会话类型的多态λ演算。《信息与计算》，2022年。URL https://doi.org/10.1016/j.ic.2022.104948。Sandra Alves、Maribel Fern´andez、M´ario Florido 和 Ian Mackie。G¨odel 系统的线性性与迭代器类型。《高阶符号计算》，23(1):1–27，2010年。https://doi.org/10.1007/s10990- 010-9060-x。Robert Atkey。定量类型论的语法与语义。载于 Anuj Dawar 和 Erich Gr¨adel 编，*LICS2018*，第56–65页，英国牛津，2018年。ACM。https://doi.org/10.1145/3209108.3209189。URL https://doi.org/10.1145/3209108.3209189。Stephanie Balzer 和 Frank Pfenning。会话类型中的显式共享。《ACM程序设计语言汇刊》，1(ICFP):37:1–37:29，2017年。https://doi.org/10.1145/3110281。Massimo Bartoletti、Alceste Scalas 和 Roberto Zunino。会话类型的语义解构。载于 Paolo Baldan 和 Daniele Gorla 编，*CONCUR 2014*，《计算机科学讲义》第8704卷，第402–418页，意大利罗马，2014年。Springer。https://doi.org/10.1007/978- 3-662-44584-6 28。

Giovanni Bernardi 和 Matthew Hennessy。使用高阶契约对会话类型建模（扩展摘要）。载于 *CONCUR*，收录于《计算机科学讲义》第8704卷，第387–401页，意大利罗马，2014年。Springer。https://doi.org/10.1007/978-3-662-44584-6 27。Giovanni Bernardi 和 Matthew Hennessy。使用高阶契约对会话类型建模。计算机科学中的逻辑方法，12(2)，2016。https://doi.org/10.2168/LMCS-12(2:10)2016。Jean-Philippe Bernardy、Mathieu Boespflug、Ryan R. Newton、Simon Peyton Jones 和 Arnaud Spiwack。线性 Haskell：高阶多态语言中的实用线性性。《*ACM*程序设计语言汇刊》，2(POPL):5:1–5:29，2018。https://doi.org/10.1145/3158093。网址：https://doi. org/10.1145/3158093。Rod M. Burstall。函数式编程语言的设计考量。载于信息技术现状会议：软件革命，哥本哈根，1977年10月。Rod M. Burstall、David B. MacQueen 和 Donald Sannella。HOPE：一种实验性的应用式语言。载于*1980*年*LISP*会议论文集，第136–143页，美国加利福尼亚州斯坦福，1980年。ACM。https://doi.org/10.1145/800087.802799。Lu´ıs Caires 和 Frank Pfenning。会话类型作为直觉主义线性命题。载于 Paul Gastin 和 Fran¸cois Laroussinie 编，*CONCUR 2010*，收录于《计算机科学讲义》第6269卷，第222–236页，法国巴黎，2010年。Springer。https://doi.org/10.1007/978-3-642-15375- 4 16。Lu´ıs Caires、Frank Pfenning 和 Bernardo Toninho。线性逻辑命题作为会话类型。数学结构与计算机科学，26(3):367–423，2016。https://doi.org/10.1017/S0960129514000218。Giuseppe Castagna、Mariangiola Dezani-Ciancaglini、Elena Giachino 和 Luca Padovani。会话类型基础。载于 Ant´onio Porto 和 Francisco Javier L´opez-Fraguas 编，*PPDP2009*，第219–230页，葡萄牙科英布拉，2009年。ACM。https://doi.org/10.1145/1599410.1599437。Diana Costa、Andreia Mordido、Diogo Po¸cas 和 Vasco T. Vasconcelos。System F 中的高阶上下文无关会话类型。载于 Marco Carbone 和 Rumyana Neykova 编，第*13*届并发与通信中心软件编程语言方法国际研讨会论文集，*PLACES@ETAPS 2022*，*2022*年*4*月*3*日，德国慕尼黑，收录于*EPTCS*第356卷，第24–35页，2022年。https://doi.org/10.4204/EPTCS.356.3。网址：https: //doi.org/10.4204/EPTCS.356.3。Olivier Danvy 和 Andrzej Filinski。控制的表示：对 CPS 变换的研究。数学结构与计算机科学，2(4):361–391，1992年。https://doi.org/10.1017/S0960129500001535。Ankush Das、Henry DeYoung、Andreia Mordido 和 Frank Pfenning。嵌套会话类型。载于 Nobuko Yoshida 编，编程语言与系统——第*30*届欧洲编程研讨会，*ESOP 2021*，收录于《计算机科学讲义》第12648卷，第178–206页，卢森堡卢森堡市，2021年。Springer。https://doi.org/10.1007/978-3-030-72019-3 7。Jana Dunfield 和 Neel Krishnaswami。双向类型检查。*ACM*计算调查，54(5):98:1–98:38，2021年。https://doi.org/10.1145/3450952。Matthias Felleisen。一等提示的理论与实践。载于 Jeanne - Ferrante 和 Peter Mager 编，*POPL*，第180–190页，美国加利福尼亚州圣迭戈，1988年。ACM出版社。https://doi.org/10.1145/73560.73576。Simon Fowler、Wen Kokke、Ornela Dardha、Sam Lindley 和 J. Garrett Morris。平滑分离会话。载于 Serge Haddad 和 Daniele Varacca 编，第*32*届并发理论国际会议，*CONCUR 2021*，*2021*年*8*月*24–27*日，收录于*LIPIcs*第203卷，第36:1–36:18页，线上会议，2021a。Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik。https://doi.org/10.4230/LIPIcs.CONCUR.2021.36。

Simon Fowler、Wen Kokke、Ornela Dardha、Sam Lindley 和 J. Garrett Morris。平滑地分离会话。*CoRR*，abs/2105.08996，2021b。URL https://doi.org/10.48550/arXiv. 2105.08996。FreeST。最自由的编程语言。http://rss.di.fc.ul.pt/tools/freest/，（最后访问于2023年3月）。Simon J. Gay 和 Malcolm Hole。客户端-服务器交互的类型与子类型。载于 S. Doaitse Swierstra（编），《编程语言与系统》，第*8*届欧洲编程研讨会，*ESOP'99*，《计算机科学讲义》第1576卷，第74–90页，荷兰阿姆斯特丹，1999年。Springer。https://doi.org/10.1007/3-540-49099-X 6。Simon J. Gay 和 Malcolm Hole。pi演算中的会话类型子类型关系。《信息学学报》，42(2-3):191–225，2005。https://doi.org/10.1007/s00236-005-0177-z。Simon J. Gay 和 Vasco Thudichum Vasconcelos。异步会话类型的线性类型理论。《函数式编程杂志》，20(1):19–50，2010。https://doi.org/10.1017/S0956796809990268。Simon J. Gay、Diogo Po¸cas 和 Vasco T. Vasconcelos。无限会话类型的不同面貌。载于 Patricia Bouyer 和 Lutz Schr¨oder（编），*FOSSACS 2022*，《计算机科学讲义》第13242卷，第347–367页，德国慕尼黑，2022年。Springer。https://doi.org/10.1007/978-3-030-99253-8 18。Cordelia V. Hall、Kevin Hammond、Simon L. Peyton Jones 和 Philip Wadler。Haskell中的类型类。《*ACM*程序设计语言与系统汇刊》，18(2):109–138，1996。https://doi.org/10.1145/227699.227700。Kohei Honda。用于二方交互的类型。载于 Eike Best（编），*CONCUR '93*，《计算机科学讲义》第715卷，第509–523页，德国希尔德斯海姆，1993年。Springer。https://doi.org/10.1007/3-540-57208-2 35。Kohei Honda、Vasco Thudichum Vasconcelos 和 Makoto Kubo。基于结构化通信的编程语言原语与类型纪律。载于 Chris Hankin （编），《编程语言与系统——*ESOP'98*，第*7*届欧洲编程研讨会》，《计算机科学讲义》第1381卷，第122–138页，葡萄牙里斯本，1998年。Springer。https://doi.org/10.1007/BFb0053567。Simon L. Peyton Jones、Andrew D. Gordon 和 Sigbjørn Finne。并发Haskell。载于 Hans- Juergen Boehm 和 Guy L. Steele Jr.（编），*POPL'96*，第295–308页，美国佛罗里达州圣彼得斯堡海滩，1996年。ACM出版社。https://doi.org/10.1145/237721.237794。Steve Klabnik、Carol Nichols 和 Rust Community。《Rust编程语言》。https: //doc.rust-lang.org/book/，2018年。Neelakantan R. Krishnaswami、Pierre Pradic 和 Nick Benton。整合线性类型与依赖类型。载于 Sriram K. Rajamani 和 David Walker（编），*POPL 2015*，第17–30页，印度孟买，2015年。ACM。https://doi.org/10.1145/2676726.2676969。Sam Lindley 和 J. Garrett Morris。会话类型的结构化递归：香蕉对话。载于 Jacques Garrigue、Gabriele Keller 和 Eijiro Sumii（编），第*21*届*ACM SIGPLAN*国际函数式编程会议论文集，*ICFP 2016*，第434–447页，日本奈良，2016年。ACM。https://doi.org/10.1145/2951913.2951921。Sam Lindley 和 J. Garrett Morris。轻量级函数式会话类型。载于 Simon Gay 和 Ant´onio Ravara（编），《行为类型：从理论到工具》。River Publishers，丹麦吉斯特鲁普，2017年。扩展版见 https://homepages.inf.ed.ac.uk/slindley/papers/ fst-extended.pdf。Karl Mazurak、Jianzhou Zhao 和 Steve Zdancewic。System F度中的轻量级线性类型。载于 Andrew Kennedy 和 Nick Benton（编），*TLDI 2010*，第77–88页，西班牙马德里，2010年。

ACM。https://doi.org/10.1145/1708016.1708027。网址 https://doi.org/10.1145/1708016.1708027。Conor McBride。我啥都不缺。在 Sam Lindley、Conor McBride、Philip W. Trinder 和 Donald Sannella（编），《改变世界的成功集锦》——献给 *Philip Wadler* 的论文集，庆祝他 *60* 岁生日，第 9600 卷，《计算机科学讲义》，第 207–233 页，爱丁堡，苏格兰，2016。Springer。https://doi.org/10.1007/978-3- 319-30936-1 12。网址 https://doi.org/10.1007/978-3-319-30936-1_12。Robin Milner、Joachim Parrow 和 David Walker。移动进程演算，II。《信息与计算》，100(1):41–77，1992年。https://doi.org/10.1016/0890-5401(92)90009-5。Fabrizio Montesi 和 Marco Peressotti。线性逻辑、 π-演算及其元理论：证明即进程的配方。*CoRR*，abs/2106.11818，2021年。网址 https://arxiv.org/abs/ 2106.11818。Martin Odersky、Olivier Blanvillain、Fengyun Liu、Aggelos Biboudis、Heather Miller 和 Sandro Stucki。Simplicitly：隐式函数类型的基础与应用。《*ACM*程序设计语言汇刊》，2(POPL):42:1–42:29，2018年。https://doi.org/10.1145/3158130。Bryan O'Sullivan。gauge：用于性能测量与分析的小型框架。https: //hackage.haskell.org/package/gauge-0.2.5/，（最后访问于2023年3月）。Luca Padovani。上下文无关会话类型推断。载于 Hongseok Yang（编），《编程语言与系统——第*26*届欧洲编程研讨会，*ESOP 2017*》，第 10201 卷，《计算机科学讲义》，第 804–830 页，乌普萨拉，瑞典，2017年。Springer。https://doi.org/10.1007/978-3-662-54434-1 30。Luca Padovani。上下文无关会话类型推断。《*ACM*程序设计语言与系统汇刊》，41(2): 9:1–9:37，2019年。https://doi.org/10.1145/3229062。Marco Patrignani、Eric Mark Martin 和 Dominique Devriese。递归类型的语义表达能力。《*ACM*程序设计语言汇刊》，5(POPL):1–29，2021年。https://doi.org/10.1145/3434302。Frank Pfenning 和 Dennis Griffith。极化次结构会话类型。载于 Andrew M. Pitts （编），*FoSSaCS 2015*，第 9034 卷，《计算机科学讲义》，第 3–22 页，伦敦，英国，2015年。Springer。https://doi.org/10.1007/978-3-662-46678-0 1。Frank Pfenning、Lu´ís Caires 和 Bernardo Toninho。会话类型进程演算中的携带证明的代码。载于 Jean-Pierre Jouannaud 和 Zhong Shao（编），*CPP 2011*，第 7086 卷，《计算机科学讲义》，第 21–36 页，垦丁，台湾，2011年。Springer。https://doi.org/10.1007/978-3-642-25379-9 4。Benjamin C. Pierce。《类型与程序设计语言》。MIT出版社，2002年。ISBN 978-0-262-16209- 8。Janek Spaderna、Peter Thiemann、Andreia Mordido 和 Vasco T. Vasconcelos。参数化代数协议：制品。https://doi.org/10.5281/zenodo.7804667，2023年3月。Arnaud Spiwack、Csongor Kiss、Jean-Philippe Bernardy、Nicolas Wu 和 Richard A. Eisenberg。线性限定类型：能力与唯一性的泛型推断。《*ACM*程序设计语言汇刊》，6(ICFP):137–164，2022年。https://doi.org/10.1145/3547626。Kaku Takeuchi、Kohei Honda 和 Makoto Kubo。一种基于交互的语言及其类型系统。载于 Constantine Halatsis、Dimitris G. Maritsas、George Philokyprou 和 Sergios Theodoridis（编），*PARLE '94*，第 817 卷，《计算机科学讲义》，第 398–413 页，雅典，希腊，1994年。Springer。https://doi.org/10.1007/3-540-58184-7 118。Peter Thiemann 和 Vasco T. Vasconcelos。上下文无关会话类型。载于 Jacques Garrigue、Gabriele Keller 和 Eijiro Sumii（编），第*21*届*ACM SIGPLAN*国际

函数式编程会议，*ICFP 2016*，第462–475页，日本奈良，2016年。ACM。https://doi.org/10.1145/2951913.2951926。Peter Thiemann 和 Vasco T. Vasconcelos。依赖标签的会话类型。《*ACM*程序设计语言汇刊》，4(POPL):67:1–67:29，2020年。https://doi.org/10.1145/3371135。Bernardo Toninho、Lu´ıs Caires 和 Frank Pfenning。高阶进程、函数与会话：一种单子式集成。载于 Matthias Felleisen 和 Philippa Gardner 编，《编程语言与系统——第22届欧洲编程研讨会，ESOP 2013》，《计算机科学讲义》第7792卷，第350–369页，意大利罗马，2013年。Springer。https://doi.org/10.1007/978-3-642-37036-6 20。Bernardo Toninho、Lu´ıs Caires 和 Frank Pfenning。会话类型进程中的余递归与非发散性。载于 Matteo Maffei 和 Emilio Tuosto 编，《可信全局计算——第9届国际研讨会，TGC 2014》，《计算机科学讲义》第8902卷，第159–175页，意大利罗马，2014年。Springer。https://doi.org/10.1007/978-3-662-45917-1 11。Vasco T. Vasconcelos。会话类型基础。《信息与计算》，217:52–70，2012年。https://doi.org/10.1016/j.ic.2012.05.002。David Walker。类型与程序设计语言高级主题，第“次结构类型系统”章，第3–44页。MIT出版社，2005年。

A 常见问题

A.1 会话类型与协议类型

可以为信道端点分配会话类型，以描述该通道从开始到关闭的完整通信行为。因此，会话类型对一个运行时值进行分类，即一个信道端点。协议类型不对应任何运行时值，而是描述可组合的双向通信行为。协议类型通过使用！和？运算符，将其绑定到特定方向后，成为会话类型的一部分。

从某种意义上说，协议类型描述了通信协议中可组合的部分，或两个会话类型之间的差异。我们无处不在地采用信道传递风格，正是为了强调这一观点。

A.2 取反递归

可以定义一种代数协议，对其递归使用的极性取反。这种协议虽然不常见，但没有问题，因为每次展开递归时，方向都会改变。以下是一个示例：

```
协议 Flipper  =Flipper-整数 -Flipper
```

Flipper 的服务器可以用一对相互递归函数实现，甚至只用一个函数：

```
flipper:!Flipper.结束! →单位类型flipper  c   =  令 c   =  选择 Flipper   [结束!]
c  于令 (x,c)  =接收  [整数,?Flipper.结束!]c匹配 c其中 {Flipperc  →发送  [整数,!
Flipper.结束!]xc  ▷flipper}
```

A.3 相互递归

相互递归协议不会造成问题。一般来说，服务器可以用相互递归函数实现。

```
协议 Flip   =  Flip  -整数 Flop协议
Flop  =  Flop  整数 Flip

flip:!Flip.结束! →单位类型flipc  =选择 Flip  [结束!]c ▷接收  [整数 ,!Flop.结
束!] ▷flop

flop:(整数 ,!Flop.结束!) →单位类型flop  p   =  令 (x,  c)   =  p  Flop  [结
束!]c  ▷发送  [整数 ,!Flip.结束!]x  ▷flip
```

A.4 对偶性与极性

该 对偶 运算符作用于会话类型，并从外部反转通信方向。直观地说，对偶 运算符会遍历会话类型的主干，将所有 ! 翻转为 ?，反之亦然。它**不会**进入消息“载荷”的类型。例如（其中 T 是载荷类型，S 是后续会话）：

$$\texttt{Dual}\ (?T.S) \equiv !T.\texttt{Dual}\ S \qquad\qquad \texttt{Dual}\ (!T.S) \equiv ?T.\texttt{Dual}\ S$$

相比之下，极性算子- 作用于协议类型，并从内部反转通信方向。这种逆推发生在协议类型 ρ 转变为会话的负载类型时的边界处：

$$?(-P).S \equiv !P.S \qquad\qquad !(-P).S \equiv ?P.S$$

类型系统的可靠性保证，当执行到基本的发送或接收操作时，通信方向最终已经确定。

与对偶一样，- 算子也是 involutory 的：

$$\texttt{Dual}\ (\texttt{Dual}\ S) \equiv S \qquad\qquad -(-T) \equiv T$$

A.5 递归与对偶性

Bernardi 和 Hennessy [2014, 2016] 发现了对偶性与递归会话类型交互中的一个问题，而递归类型 $T = \mu X.!X.X$最简洁地揭示了这一问题。该类型的对偶为 对偶$(\mu X.!X.X)$ $= \mu X.?(\mu X.!X.X).X$，这可以通过对该类型的展开式取对偶得到

$$\texttt{Dual}\ (\mu X.!X.X) = \texttt{Dual}\ (!(\mu X.!X.X).(\mu X.!X.X)) = ?(\mu X.!X.X).\texttt{Dual}\ (\mu X.!X.X)$$

利用方程，我们可以通过 T定义类型 $T = !T.T$。将其转换为代数会话类型后，T 所对应的协议如下：

```
协议 X = 递归 T X类型 T
  =!X.结束!

选择Mu : T → !T.T 选择Mu c = 选择 Mu [
结束!] c

对偶T : 对偶 T → ?X.结束!对偶T c =
c

匹配Mu : 对偶 T →?T.( 对偶 T) 匹配Mu c' =匹配 c' 与 {Mu c'
  →c'}
```

这种行为与以下结果完全匹配：T 根据 Bernardi 和 Hennessy [2016]。函数 `selectMu` 负责展开 T，而 `matchMu` 负责展开对偶 T。恒等映射函数的类型 `dualT` 表明对偶 `T = ?X`。结束!。

A.6 泛型服务器的效率

在第 2.3 节中，我们通过一个泛型服务器示例来展示所提出的参数化协议的表达能力。与手工编写的版本相比，该示例有大量标记操作，因此开销显著。

本研究只是第一步，并未提供由形式化定理支持的解决方案。不过，我们把 ρ 无标签协议称为不含递归且只声明单个构造器标签的协议$C\ T_1 \ldots T_n$。类似于 Haskell 的`newtype`，实现可以通过省略选择与匹配操作，将无标签协议的运行时开销降至零。

A.7 等递归与同构递归会话类型

大多数关于会话类型的研究都使用等递归类型，原因有二。其一是历史原因：Gay 和 Hole [1999, 2005] 是最早研究（尾）递归会话类型上余归纳定义的关系的学者。他们定义了（如今已成为标准的）子类型关系概念，并证明该关系是可判定的。后续研究都以他们的成果为基础。另一原因是：“会话类型的等递归解释保证了展开递归类型时无需传递消息。”（Balzer 和 Pfenning [2017]）

代数协议会像代数数据类型那样，将递归的展开与分支合并。对于具有有限运行的协议，代数协议不会引入额外消息。由于这类协议采用由分支保护的递归，本来就需要发送一条消息来决定分支。

第 2.3 节中的协议 `Stream a` 确实会引入额外消息，因为它大致对应于类型 $\mu X.!a.X$ 的同构递归解读。

A.8 子类型关系、类型类、类型推断

对于上下文无关会话类型，子类型关系是不可判定的 Padovani [2019]。然而，由于 AlgST 的会话类型检查主要依赖名义类型，因此子类型关系有望实现可判定。我们已有一项具体方案，但尚未实现，其元理论也尚未经过检验。

在 ?? 中，用于发送和接收类型为 `a` 的数据的类型 `Send a` 和 `Recv a` 看起来像是具有类型类或特征的语言中用于字典的类型 Hall 等人. [1996], Klabnik 等人。 [2018]。未来的工作中，我们计划探索添加隐式参数的做法，如 Scala 中所示 Odersky 等人。 [2018]。

类型推断是未来的工作。

A.9 方向的确定

看到类似 $?T.S$ 和 $!T.S$这样的类型可能会令人困惑，因为方向之后可能发生变化。不过，这种行为完全可预测。只要 T 是协议类型变量 α： $\mathbf{p}$，方向就可能发生变化，因为 α 可以被某个协议 $-T'$替换。一旦 T 是种类为$\mathbf{t}$ 的普通类型（即使 T 是变量），方向就会固定。例如，在原语发送 `∀（a: M）`的类型中。 `∀（s: S）。a → !a.s → s` 中，负载类型 `a` 的种类为 $\mathbf{m}$，因此方向是固定的。

B 泛型服务器工具箱

类似于 Stream 协议，我们可以为子协议的有限次重复定义一种协议和一个通用服务器：

协议 重复 x = 更多 x（重复 x） | 退出

重复： ∀(p: P). 服务 p →服务（重复 p）重复 [p] serveP [s]c = 匹配 c 于 { 退出 c → c，更多 c → serveP [? 重复 p.s] c 重复 [p] serveP [s]}

随后可以像使用Stream那样使用此协议：

repeatArith = 重复 [Arith] serveArith

我们可以定义一个提供子协议 n 次的主动版本，并将其也用于Arith：

repeatActive : Int → ∀(p: P). 服务 p →服务 -(Repeat -p) repeatActiveArith : Int → 服务 -(Repeat - Arith)

虽然实现很直接，但敏锐的读者可能会问，为什么我们要在Service -(Repeat -p)中向服务 提供负参数。具体来说，我们能否使用对偶运算符来切换服务的方向?

要看出这个提议的问题，我们来展开这两个候选项：

对偶 (服务 (Repeat -p)) =对偶 (∀(s: S). ?Repeat -p.s →s)

由于右侧使用 对偶 不符合良构类型要求，因此该提案注定失败，因为在我们的系统中，全称类型和函数类型都不是会话类型。

服务 -(Repeat -p) = ∀(s: S). ?-(Repeat -p).s →s= ∀(s: S). !Repeat -p.s →s

否定 参数以精准方式将服务 中的接收运算符切换为发送 n.

类型的 C 证明

示例 *le 1.* 我们将引言中的示例推广为一个参数化栈协议。

协议 Stack a = 弹出 -a | 压入 a (Stack a) (Stack a)

我们设定 $\Delta = \mathtt{Neg} : \mathbf{p} \to \mathbf{p}, \mathtt{a} : \mathbf{p}$ 并得到直接的推导

$$\dfrac{\dfrac{\dfrac{\Delta \vdash \mathtt{a} \Rightarrow \mathbf{p}}{\Delta \vdash \mathtt{a} \Leftarrow \mathbf{p}}}{\Delta \vdash \mathtt{-a} \Leftarrow \mathbf{p}} \qquad \dfrac{\Delta \vdash \mathtt{a} \Rightarrow \mathbf{p}}{\Delta \vdash \mathtt{a} \Leftarrow \mathbf{p}} \qquad \dfrac{\mathtt{Stack} : \ \mathbf{p} \to \mathbf{p} \in \Delta \qquad \dfrac{\Delta \vdash \mathtt{a} \Rightarrow \mathbf{p}}{\Delta \vdash \mathtt{a} \Leftarrow \mathbf{p}}}{\Delta \vdash \text{Stack 一个} \ \Rightarrow \mathbf{p}}}{\Delta \vdash \text{Stack 一个} \ \Leftarrow \mathbf{p}}}{\vdash \mathtt{Stack} \Rightarrow \mathbf{p} \to \mathbf{p}}$$

这三个子推导分别对应协议构造子的参数。由此可知，协议 Stack a 的定义格式良好。

引理 1（具体化和方向运算符的性质）。

1. $\Delta \vdash \mathit{Dual}\ (\S(T).S) \equiv \S(-(T)).\mathit{Dual}\ S : \mathbf{s}$
2. $-(+(T)) = -(T)$
3. $-(-(T)) = +(T)$

证明。1. 根据 § 运算符的定义进行分类讨论（图 3）。情形 $T = -U$。由于 $\S(-U).S = ?U.S$，规则 C-DualIn 给出 $\Delta \vdash$ 对偶 $(\S(-U).S) \equiv !U.$对偶 $S : \mathbf{s}$。注意到 $-(-U) = +(U) = U$，我们得出 $\S(-(-U))$。对偶 $S = !U.$对偶 S。情形 $T \neq -U$ 由于 $\S(T).S = !T.S$，规则 C-DualOut 给出 $\Delta \vdash$ 对偶 $(\S(T).S) \equiv ?T.$对偶 $S : \mathbf{s}$。注意到 $-(T) = -T$，即可得出结论。2. 通过根据方向运算符的定义进行分类讨论即可证明。情形 $T = -U$。我们有：$-(+(-U)) = -(-(U)) = -(-U)$。情形 $T \neq -U$。我们有：$-(+(T)) = -(T)$。3. 通过根据方向运算符的定义进行分类讨论即可证明。情形 $T = -U$。我们有：$-(-(-U)) = -(+(U)) = -(U) = -U = +(-U)$。情形 $T \neq -U$。我们有：$-(-(T)) = -(-T) = +(T)$。

引理 2（类型转换的一致性）。如果 $\Delta \vdash T \equiv U : \kappa$，那么 $\Delta \vdash T \Leftarrow \kappa$ 和$\Delta \vdash U \Leftarrow \kappa$。

证明。对该假设进行规则归纳。

我们首先说明范式中的类型语法。良构类型的范式为 Q，定义如下文法：

$$Q ::= R \mid -R$$
$$R ::= \text{单位类型} \mid R \to R \mid R \otimes R \mid \forall \alpha{:}\ \kappa.R \mid \alpha \mid ?R.R \mid !R.R \mid \text{结束}_? \mid \text{结束}_! \mid \text{对偶}\ \alpha \mid \rho\ \overline{Q}$$

命题 1（种类保持性）。

1. 对所有 $\Delta \vdash T \Rightarrow \kappa$，$\Delta \vdash \mathrm{nrm}^+(T) \equiv T : \kappa$。2. 对所有 $\Delta \vdash T \Rightarrow \mathbf{s}$，$\Delta \vdash \mathrm{nrm}^-(T) \equiv \mathit{Dual}\ T : \mathbf{s}$。

证明。 参见 Agda 开发项目 nf-sound+ 和 nf-sound-。

证明（定理 *1*（可靠性）的证明）。第 1 项。假设 $\mathrm{nrm}^+(T) =_\alpha \mathrm{nrm}^+(U)$。由命题 1 可得 $\Delta \vdash \mathrm{nrm}^+(T) \equiv T : \kappa$ 且 $\Delta \vdash \mathrm{nrm}^+(U) \equiv U : \kappa$。由于类型相等在 $\equiv$中，利用转换的对称性和传递性即可得证。第 2 项的证明类似。

证明（定理 *2*（完备性）的证明）。参见 Agda 开发项目 `nf-complete` 和 `nf-complete-`。

为了获得计算范式的复杂度界，我们利用其句法形式。

引理 3. 如果 $\mathrm{nrm}^{\pm}(T) = T'$，那么 T' 可由 Q 描述。

证明。 参见 Agda 开发中的 nf-normal。

命题 2. 对于每个 $\Delta \vdash T \Rightarrow \kappa$，$\mathrm{nrm}^{\pm}(T)$ 的最坏情况运行时间属于 $O(|T|)$。

证明。 运行 $\mathrm{nrm}^{\pm}(T)$ 会访问输入类型中的每个 $|T|$ 节点一次。在每个节点，它要么根据子项的范式在 $O(1)$ 时间内重构类型，要么（对于传递规则）对范式进行极性校正。根据引理 3，范式开头至多有一个负极性，因此校正至多需要两步，耗时 $O(1)$。

Pro（定理*3*的证明（类型等价的时间复杂度））。引理 3 的推论 .

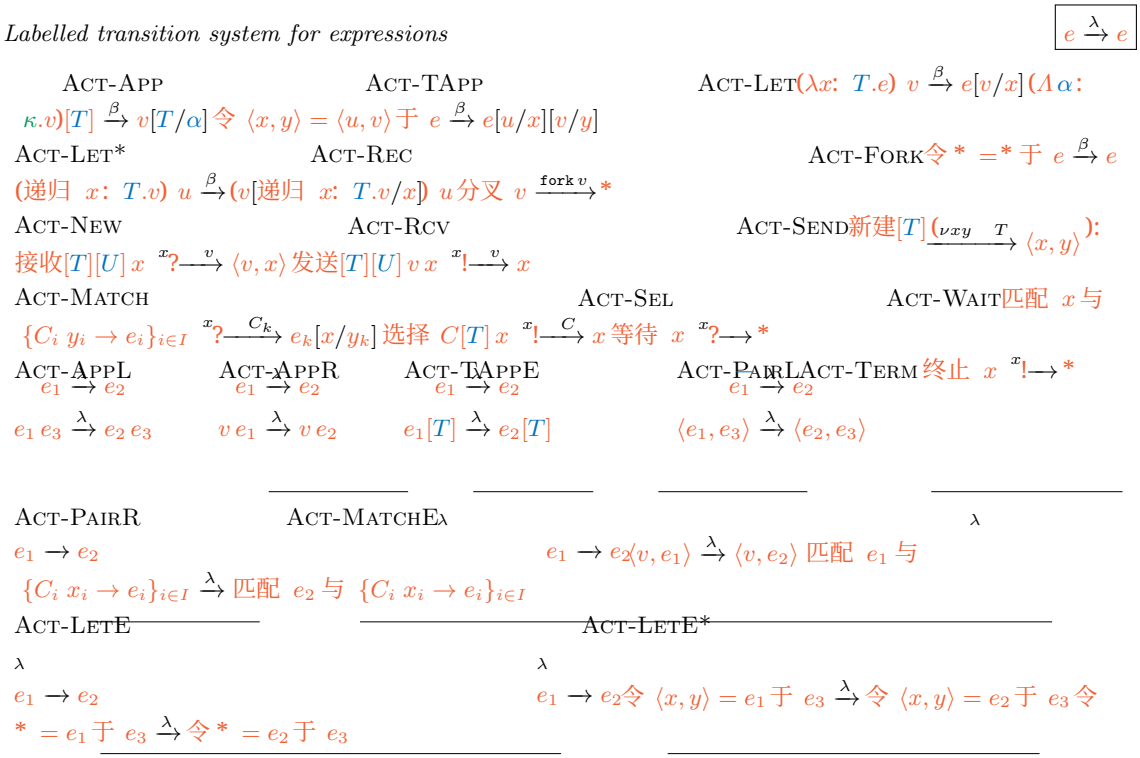


图 11：表达式的标记转换系统

D 表达式与进程的证明

引理 4。

1. 若 $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$ 且 $\Delta \vdash \Gamma_1 \Leftarrow \mathbf{t}$，则 $\Delta \vdash \Gamma_2 \Leftarrow \mathbf{t}$ 且 T 是范式。*2.* 若 $\Delta \mid \Gamma_1 \vdash e \Leftarrow T \mid \Gamma_2$ 且 $\Delta \vdash \Gamma_1 \Leftarrow \mathbf{t}$ 且 T 是范式，则 $\Delta \vdash \Gamma_2 \Leftarrow \mathbf{t}$。

证明。

1. 对第一个假设应用规则归纳。*2.* 使用第 *1* 项，并注意到满足 U 的类型 $U =_\alpha T$ 也处于范式。

引理 5（规范化的性质）。

1. 如果 $\Delta \vdash \mathit{nrm}^+(T) \Rightarrow \kappa$，那么 $\Delta \vdash T \Leftarrow \kappa$。*2.* 如果 $\Delta \vdash \mathit{nrm}^-(T) \Rightarrow \kappa$，那么 $\Delta \vdash T \Leftarrow s$。*3.* 如果 $\Delta \vdash \S(T).S \Rightarrow \kappa$，那么 $\Delta \vdash T \Leftarrow p$ 和 $\Delta \vdash S \Leftarrow s$。*4.* 如果 $\Delta \vdash \pm(T) \Rightarrow \kappa$，那么 $\Delta \vdash T \Leftarrow \kappa$。

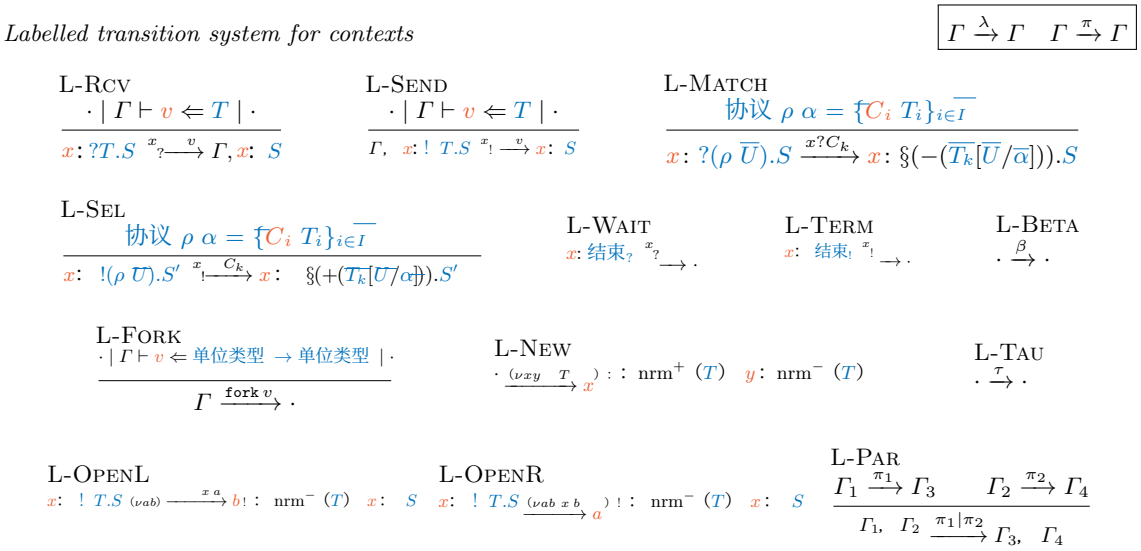


图 12：类型上下文的标记转换系统

证明。证明采用对各个假设进行相互归纳法。大多数情况都可通过直接归纳法得出。我们详细说明 $\mathrm{nrm}^-(?T.S) = \S(+(\mathrm{nrm}^+(T))).\mathrm{nrm}^-(S)$ 这一情形。条目 3 给出 $\Delta \vdash +(\mathrm{nrm}^+(T)) \Leftarrow \mathrm{p}$ 和 $\Delta \vdash \mathrm{nrm}^-(S) \Leftarrow \mathrm{s}$。规则 T-Sub 给出 $\Delta \vdash +(\mathrm{nrm}^+(T)) \Rightarrow \mathrm{p}$ 和 $\Delta \vdash \mathrm{nrm}^-(S) \Rightarrow \mathrm{s}$。条目 4 给出 $\Delta \vdash \mathrm{nrm}^+(T) \Rightarrow \mathrm{p}$。同样，规则 T-Sub 给出 $\Delta \vdash \mathrm{nrm}^+(T) \Leftarrow \mathrm{p}$。条目 1 给出 $\Delta \vdash T \Rightarrow \mathrm{p}$。再次，规则 T-Sub 给出 $\Delta \vdash T \Leftarrow \mathrm{p}$。根据 $\Delta \vdash \mathrm{nrm}^-(S) \Leftarrow \mathrm{s}$，归纳法给出 $\Delta \vdash S \Rightarrow \mathrm{s}$，而 T-Sub 给出 $\Delta \vdash S \Leftarrow \mathrm{s}$。最后，规则 T-In 给出 $\Delta \vdash ?T.S \Rightarrow \mathrm{s}$，规则 T-Sub 则推出 $\Delta \vdash ?T.S \Leftarrow \mathrm{s}$。

引理 6（替换）。

1. 若 $\Delta, \alpha : \kappa \vdash T \Rightarrow \kappa'$ 和 $\Delta \vdash U \Leftarrow \kappa$，则 $\Delta \vdash T[U/\alpha] \Rightarrow \kappa'$。*2.* 若 $\Delta, \alpha : \kappa \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$ 和 $\Delta \vdash U \Leftarrow \kappa$，则 $\Delta \mid \Gamma_1 \vdash e[U/\alpha] \Rightarrow \mathit{nrm}^+(T[U/\alpha]) \mid \Gamma_2$。*3.* 若 $\Delta \mid \Gamma_1, x : U \vdash e \Rightarrow T \mid \Gamma_2$ 和 $\Delta \mid \Gamma_3 \vdash v \Leftarrow U \mid \cdot$，则 $\Delta \mid \Gamma_1, \Gamma_3 \vdash e[v/x] \Rightarrow T \mid \Gamma_2$。*4.* 若 $\Delta \mid \Gamma_1, x :^* U \vdash e \Rightarrow T \mid \Gamma_2, x :^* U$ 和 $\Delta \mid \Gamma_3 \vdash v \Leftarrow U \mid \cdot$，则 $\Delta \mid \Gamma_1, \Gamma_3 \vdash e[v/x] \Rightarrow T \mid \Gamma_2$。

证明。 通过对第一个假设进行相互归纳即可得证。

引理 7（合意）。 令 $\Delta \vdash \Gamma_1 \Leftarrow \mathbf{t}$。

1. 若 $\Delta \mid \Gamma_1, x : T \vdash e \Rightarrow U \mid \Gamma_2$，则 $\Delta \vdash T \Leftarrow \kappa$。
2. 若 $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$，则 $\Delta \vdash T \Leftarrow \kappa$。*3.* 若 $\Gamma_1, x : T \vdash p$，则 $\cdot \vdash T \Leftarrow \kappa$。

证明。

1. 对第一个假设应用规则归纳。2. 对第一个假设应用规则归纳，并使用引理 6 和引理 2。

3. 对第一个假设应用规则归纳，使用 E-Check 和第 1 项。

单调性表明，在类型检查过程中不会创建资源 g.

引理 8（单调性）。如果 $\Delta \mid \Gamma_1 \vdash v \Rightarrow T \mid \Gamma_2$，那么 $\Gamma_2 \subseteq \Gamma_1$。

证明。对假设进行直接的规则归纳。基本情形为 E-Const 和 E-Var。E-Abs、E-TAbs、E-Pair 和 E-TApp 的情形可由归纳法以及子集关系的传递性得出。

引理 9（强化）。

1. 如果 $\Delta, \alpha\colon \kappa \vdash T \Rightarrow \kappa'$ 且 $\alpha \notin \mathrm{fv}(T)$，那么 $\Delta \vdash T \Rightarrow \kappa'$。*2.* 如果 $\Delta \mid \Gamma_1, x\colon U \vdash e \Rightarrow T \mid \Gamma_2, x\colon U$，那么 $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$。

证明。

1. 证明通过对第一个假设使用归纳假设，并使用 T-Sub .
2. 证明通过对第一个假设使用归纳假设。

引理 10（弱化）。若 $\Delta \mid \Gamma_1 \vdash e \Rightarrow T \mid \Gamma_2$，则 $\Delta \mid \Gamma_1, x\colon U \vdash e \Rightarrow T \mid \Gamma_2, x\colon U$。

证明。证明通过对该假设进行归纳得出。

保持性与进展 现在我们证明主要结果。

证明（定理 *4*（保持性）的证明）。通过对第一个假设进行互相规则归纳即可得证。

1. 规则 Act-App。在此情形下，$(\lambda x\colon U.e)\ v \xrightarrow{\beta} e[v/x]$ 和 $\Gamma_2 = \Gamma_2' = \cdot$。对 E-App 在 $\Delta \mid \Gamma_1, \Gamma_3 \vdash (\lambda x\colon U.e)\ v \Rightarrow T \mid \Gamma_3$ 进行逆推，得到 (a) $\Delta \mid \Gamma_1, \Gamma_3 \vdash \lambda x\colon U.e \Rightarrow V \to T \mid \Gamma'$ 以及 (b) $\Delta \mid \Gamma' \vdash v \Leftarrow V \mid \Gamma_3$。根据单调性（引理 8），我们知道 $\Gamma_3 \subseteq \Gamma'$；记为 $\Gamma' = \Gamma_1', \Gamma_3$。对 (a) 应用逆推，并对 (b) 应用强化（引理 9），得到 $\Delta \mid \Gamma_1, \Gamma_3, x\colon \mathrm{nrm}^+(U) \vdash e \Rightarrow T \mid \Gamma_1', \Gamma_3$ 以及 $\Delta \mid \Gamma_1' \vdash v \Leftarrow V \mid \cdot$（且 $V = \mathrm{nrm}^+(U)$）。根据 C-Refl 以及第二个假设上的一致性（引理 7），我们知道 $T \equiv_A T$。最后应用替换引理（引理 6，第 3 项）和弱化即可得证。

   Rule Act-TApp。再者，$\Gamma_2 = \Gamma_2' = \cdot$。对第二个假设应用对 E-TApp 的逆推 s, 随后 对其应用对 E-TAbs 的逆推，并使用 $T' = T$，根据引理 6 的第 2 项得出结论。
   规则 Act-Let。对第二个假设应用对 E-Let 的逆推，然后对 E-Pair 应用逆推，再应用替换（引理 6 的第 3 项），并使用 $T' = T$ 得出结论。
   规则 Act-Let*。在此情况下，令 * =* 属于 $e \xrightarrow{\beta} e$ 且 $\Gamma_2 = \Gamma_2' = \cdot$。对 E-Const 和 E-Check 应用 * 的逆推可知，E-Let* 的前提包括 $\Delta \mid \Gamma_1$、$\Gamma_3 \vdash e \Rightarrow T \mid \Gamma_3$，这正是我们要证明的，因为 $T \equiv_A T$。

   规则 Act-Rec。再次，$\Gamma_2 = \Gamma_2' = \cdot$。规则 E-App 的前提为 $\Delta \mid \Gamma_1, \Gamma_3 \vdash$ 递归 $x\colon V.v \Rightarrow U \to T \mid \Gamma_1, \Gamma_3$ 以及 $\Delta \mid \Gamma_1, \Gamma_3 \vdash u \Leftarrow U \mid \Gamma_3$。对前者应用 E-Rec 的逆推，得出 $\mathrm{nrm}^+(V) = U \to T$，然后应用 E-Check 的逆推和替换引理（引理 6），得到 $\Delta \mid \Gamma_1, \Gamma_3 \vdash v[$ 递归 $x\colon V.v/x] \Rightarrow U \to T \mid \Gamma_1, \Gamma_3$。应用 E-App 并利用 $T \equiv_A T$ 这一事实，得出结论。
   规则 Act-Fork。逆推 E-App 得到 $\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash$ 分叉 $\Rightarrow U \to T \mid \Gamma'$ 和 $\Delta \mid \Gamma' \vdash v \Leftarrow U \mid \Gamma_3$。对前者应用 E-Const，我们知道 $\Gamma' = \Gamma_1, \Gamma_2, \Gamma_3, U =$ 单位类型 $\to$ 单位类型

以及 $T = 单位类型$。根据 L-Fork，我们知道 $\Delta \mid \Gamma_2 \vdash v \Leftarrow 单位类型 \to 单位类型 \mid \cdot$，并且 $\Gamma_1 = \cdot$ 在此情况下。对 E-Const 应用于 $*$ 后，我们得出 $\Delta \mid \Gamma_3 \vdash * \Rightarrow 单位类型 \mid \Gamma_3$，正如我们所希望的那样。

规则 Act-New。在此情形下，new$[U]\xrightarrow{\nu x y \quad U}\langle x, y\rangle$):。根据 L-New，我们知道 $\Gamma_2 = \cdot$ and$\Gamma_2' = x$: $\mathrm{nrm}^+(U)$ y: $\mathrm{nrm}^-(U)$。对第二个假设应用 E-TApp and E-Const 的逆推，得到 $T = \mathrm{nrm}^+(U) \otimes \mathrm{nrm}^-(U)$ and $\Gamma_1 = \cdot$。利用 E-Var and E-Pair，我们得出$\Delta \mid \Gamma_3, x : \mathrm{nrm}^+(U)$ $y : \mathrm{nrm}^-(U) \vdash \langle x, y\rangle \Rightarrow \mathrm{nrm}^+(U) \otimes \mathrm{nrm}^-(U) \mid \Gamma_3$。

规则 Act-Rcv。在此情况下，receive$[U][V]\, x \xrightarrow{x?v} \langle v, x\rangle$。根据 L-Rcv，$\Gamma_2 = x : ?U.S$ and $\Gamma_2' = x : S$，Γ，其中 Γ 满足 $\cdot \mid \Gamma \vdash v \Leftarrow U \mid \cdot$。对第二个假设中的 E-App 应用逆推，得到 $\Delta \mid \Gamma_1, x : ?U.S, \Gamma_3 \vdash$ receive$[U][V] \Rightarrow W \to T \mid \Gamma_1'$ and $\Delta \mid \Gamma_1' \vdash x \Leftarrow W \mid \Gamma_3$。对后者中的 E-Var 应用逆推，得到 $\Gamma_1' = x : ?U.S, \Gamma_3$ and $W = ?U.S$。对 E-TApp 应用两次逆推，然后应用 E-Const，即可得出 $\Gamma_1 = \cdot$ and $T = U \otimes S$。利用 L-Rcv、E-Check 和弱化，我们知道 $\Delta \mid x : S, \Gamma, \Gamma_3 \vdash v \Rightarrow U \mid x : S, \Gamma_3$；利用 E-Var 和弱化，我们得到 $\Delta \mid x : S, \Gamma_3 \vdash x \Rightarrow S \mid \Gamma_3$。应用 E-Pair 并回想 $U \otimes S = T \equiv_A T$，即可得出结论。

规则 Act-Send。在此情况下，发送$[U][S]\, v\, x \xrightarrow{x!v} x$。根据 L-Send，$\Gamma_2 = \Gamma$，$x : \; !\, U.S$ 其中 Γ 满足 $\cdot \mid \Gamma \vdash v \Leftarrow U \mid \cdot$。与前一种情况类似，对 E-App进行两次逆推，接着应用 E-TApp，然后对 E-Const 在 发送 上进行逆推，从而得出 $S = T$。另一方面，对 E-Var 进行逆推，并对其他前提应用弱化后使用 L-Send，从而得出 $\Gamma_1 = \cdot$。应用 E-Var 并用 $\Gamma_2' = x$进行弱化，T，再利用 $T \equiv_A T$，即可得到结果。

Act-Match 规则。在此情况下，匹配 x 与 $\{C_i\ y_i \to e_i\}_{i \in I}$ $\xrightarrow{x?C_k} e_k[x/y_k]$。根据 L-Match，我们有 $\Gamma_2 = x : ?(P\ U).S'$，其协议为 $\rho\ \alpha = \{C_i\ T_i\}_{i \in I}$。E-Match的前提包括 (a)$\Delta \mid \Gamma_1, x : ?(P\ U).S'$，$\Gamma_3 \vdash x \Rightarrow ?(P\ U).S' \mid \Gamma_1, \Gamma_3$，(b) $\Delta \mid \Gamma_1, \Gamma_3, y_i : \S(-(T_i[U/\alpha])).S \vdash e_i \Rightarrow V_i \mid \Gamma_3$，以及 (c) $T =_\alpha V_i$。根据 L-Match，我们知道 $\Gamma_2' = x : \S(-(T_k[U/\alpha])).S$。应用 E-Var和 E-Check，得到 (d) $\Delta \mid x : \S(-(T_k[U/\alpha])).S \vdash x \Leftarrow \S(-(T_k[U/\alpha])).S \mid \cdot$。将替换引理（引理 6）应用于 (b) 和 (d)，即可得出结论。

Act-Sel 规则。我们有 select $C[U]\, x \xrightarrow{x!C} x$。使用 L-Sel，我们知道 $\Gamma_2 = x : !(P\ U).S'$对于协议 $\rho\ \alpha = \{C_i\ T_i\}_{i \in I}$。在第二个假设上对 E-App 和 E-TApp 使用逆推，得出 $T = \S(+(T_k[U/\alpha])).S'$。使用 E-Var 的逆推，得出 $\Gamma_1 = \cdot$。最后，使用 E-Var和弱化（引理 10），得到 $\Delta \mid x : \S(+(T_k[U/\alpha])).S', \Gamma_3 \vdash x \Rightarrow \S(+(T_k[U/\alpha])).S' \mid \Gamma_3$。观察到 $\S(+(T_k[U/\alpha])).S' = T \equiv_A T$，即可得出结论。

Act-Wait 规则。在这种情况下，$\Gamma_2 = x$: 结束$_?$和 $\Gamma_2' = \cdot$。在第二个假设上对 E-App 进行逆推，得到 $\Delta \mid \Gamma_1, x$: 结束$_?, \Gamma_3 \vdash$等待 $\Rightarrow U \to T \mid \Gamma'$和 $\Delta \mid \Gamma' \vdash x \Leftarrow U \mid \Gamma_3$。对前者应用 E-Const，对后者应用 E-Var，得出 $\Gamma_1 = \cdot$, $U =$ 结束$_?$ 和$T =$单位类型。使用 E-Const对 * 进行弱化（引理 10），并使用 $T' = T$，即可得到结果。Act-Term 规则类似。

规则 Act-AppR。在此情形下，$v\, e_1 \xrightarrow{\lambda} v\, e_2$。根据第二个假设和规则 E-App，我们知道 (a) $\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash v \Rightarrow U \to T \mid \Gamma_1', \Gamma_2, \Gamma_3$ 以及 (b) $\Delta \mid \Gamma_1', \Gamma_2, \Gamma_3 \vdash e_1 \Leftarrow U \mid \Gamma_3$。对条目 2 使用归纳假设，并利用 (b)，我们得到 $\Delta \mid \Gamma_1', \Gamma_2', \Gamma_3 \vdash e_2 \Leftarrow U \mid \Gamma_3$。对 (a) 应用弱化和强化，我们得到 $\Delta \mid \Gamma_1, \Gamma_2', \Gamma_3 \vdash v \Rightarrow U \to T \mid \Gamma_1', \Gamma_2', \Gamma_3$。应用 E-App，得出 $\Delta \mid \Gamma_1, \Gamma_2', \Gamma_3 \vdash v\, e_2 \Rightarrow T \mid \Gamma_3$。

规则 Act-AppL、Act-TAppE、Act-LetE、Act-LetE*、Act-PairL、Act-PairR 和 Act-MatchE 类似。

2. 对第二个假设中的 E-Check 进行逆推，得到 $\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \Rightarrow U \mid \Gamma_3$ 和$U =_\alpha T$。根据第 1 项的归纳假设，我们知道 $\Delta \mid \Gamma_1, \Gamma_2', \Gamma_3 \vdash e_2 \Rightarrow U' \mid \Gamma_3'$与

$U' \equiv_A U$。根据 C-Sym和 C-Trans，我们知道 $T \equiv_A U'$。利用 E-Check，我们得出$\Delta \mid \Gamma_1,\ \Gamma_2', \Gamma_3 \vdash e_2 \Leftarrow T \mid \Gamma_3'$。

3. 规则 Act-Session。对 Act-Session 进行逆推可得 $e_1 \xrightarrow{\sigma} e_2$，并且对第二个假设中的 P-Exp 进行逆推可推出 $\cdot \mid \Gamma_1,\ \Gamma_2 \vdash e_1 \Leftarrow$ 单位类型 $\mid \cdot$。对第 2 项应用归纳假设，可得 $\cdot \mid \Gamma_1, \Gamma_2' \vdash e_2 \Leftarrow$单位类型 $\mid \cdot$。应用 P-Exp 得出结论。规则 Act-Beta 类似，按照 L-Tau 和 L-Beta 的规定，考虑 $\Gamma_2 = \Gamma_2' = \cdot$。规则 Act-Fork。对第一个假设中的 Act-Fork 和第二个假设中的 P-Exp 进行逆推，可得 (a) $e_1 \xrightarrow{\text{fork } v} e_2$ 以及 (b) $\cdot \mid \Gamma_1 \vdash e_1 \Leftarrow$ 单位类型 $\mid \cdot$。根据 L-Fork 可知 (c)$\Gamma_1 \xrightarrow{\text{fork } v} \cdot$ 以及 (d) $\cdot \mid \Gamma_1 \vdash v \Leftarrow$ 单位类型 $\rightarrow$ 单位类型 $\mid \cdot$。对 (a)、(b) 和 (c) 中的第 2 项应用归纳假设，可得 $\cdot \mid \cdot \vdash e_2 \Leftarrow$ 单位类型 $\mid \cdot$。对 (d) 中的 E-Check 进行逆推，并对 * 应用 E-Const，从而可以应用 E-App 推出 $\cdot \mid \Gamma_1 \vdash v$* $\Rightarrow$ 单位类型 $\mid \cdot$。使用 E-Check、P-Exp 和 P-Par 得出结论。规则 Act-New。根据 L-Tau，$\Gamma_2 = \cdot$。对第二个假设中的 P-Exp 进行逆推，可得 $\cdot \mid \Gamma_1 \vdash e_1 \Leftarrow$ 单位类型 $\mid \cdot$。对 Act-New 进行逆推，并对第 2 项应用归纳假设，可得 $\cdot \mid \Gamma_1, x$: nrm$^+(T)$ y: nrm$^-(T) \vdash e_2 \Leftarrow$单位类型$\mid \cdot$。利用 P-Exp 可知$\Gamma_1, x$: nrm$^+(T)$ y: nrm$^-(T) \vdash \langle e_2 \rangle$.使用 P-New 得出结论。规则 Act-JoinL。P-Par 的逆推要求将上下文拆分为：$\Gamma_1, \Gamma_2^1 \vdash p_1$ 以及 $\Gamma_1, \Gamma_2^2 \vdash p_2$。对 L-Par 进行逆推，并对第 3 项应用归纳假设，可得 $\Gamma_1, \Gamma_3 \vdash q_1$ 以及$\Gamma_1, \Gamma_4 \vdash q_2$，其中 $\Gamma_2' = \Gamma_3, \Gamma_4$。使用 P-Par 和 L-Par 得出结论。规则 Act-JoinR 类似。规则 Act-Msg。在此情况下，$(\nu xy)\, p_1 \xrightarrow{\tau} (\nu xy)\, p_2$。通过对 Act-Msg 进行逆推，我们知道$p_1 \xrightarrow{x?_{v\mid y}!} \xrightarrow{v} p_2$。对于一个 τ-转换，第二个假设写作 $\Gamma_1 \vdash (\nu xy)\ p_1$（回忆规则 L-Tau）。根据 P-New，有 Γ, x: nrm$^+(T)$ y: nrm$^-(T) \vdash p_1$。根据 L-Par、L-Rcv 和 L-Send，我们知道 nrm$^+(T)$ $= ?U.S$ 以及 Γ_1, x: $?U.S, y$: $!U$.nrm$^-(S)$ $\xrightarrow{x} \xrightarrow{v} \Gamma_1 ?_{v\mid y}!, x$: S, y: nrm$^-(S)$ 并且

$\cdot \mid \Gamma_1 \vdash v \Leftarrow U \mid \cdot$。对第 3 项应用归纳假设，得到 Γ_1, x: S, y: nrm$^-(S) \vdash p_2$。应用 P-New 得出结论。规则 Act-Bra。P-New 的前提包括 (a) Γ_1, x: nrm$^+(T)$ y: nrm$^-(T) \vdash p_1$。根据 L-Match、L-Sel 和 L-Par，我们知道 nrm$^+(T)$ $= ?(P\ U).S'$，其中协议为 $\rho\ \alpha = \{C_i\ T_i\}_{i \in I}$ 和$C = C_k$，对于某个 $k \in I$。因此，我们可以将 (a) 重写为 Γ_1, x: $?(P\ U).S', y$: $!(P\ U)$.nrm$^-(S') \vdash p_1$。对 Act-Bra 应用逆推可得 $p_1 \xrightarrow{x?\ C\mid y\ C} p_2$。对第 3 项应用归纳假设，得到$\Gamma_1, x$: $\S(-(T_k[U/\alpha])).S, y$: $\S(+(T_k[U/\alpha]))$.nrm$^-(S') \vdash p_2$。应用引理 1、第 1 项和 P-New 得出结论。规则 Act-Wait。P-New 的前提包括 Γ_1, x: nrm$^+(T)$ y: nrm$^-(T) \vdash p_1$。根据 L-Par、L-Wait 和 L-Term，我们知道 nrm$^+(T)$ $=$ End?。对第 3 项应用归纳假设，得到 $\Gamma_1 \vdash p_2$。规则 Act-ParL。对 Act-ParL 应用逆推可得 $p_1 \xrightarrow{\pi} p_2$。P-Par 的前提为 (a) $\Gamma_1, \Gamma_2^1 \vdash p_1$和 (b) $\Gamma_1, \Gamma_2^2 \vdash q_1$，其中 $\Gamma_2 = \Gamma_2^1, \Gamma_2^2$。对 (a) 中的第 3 项应用归纳假设，得到 $\Gamma_1, \Gamma_3 \vdash p_2$，其中 $\Gamma_2^1 \xrightarrow{\pi} \Gamma_3$。结合 (b) 并应用 P-Par 得出结论。规则 Act-ParR 类似。规则 Act-Res。Act-Res 的前提为 $p_1 \xrightarrow{\pi} p_2$ 和 $x, y \notin \text{fv}(\pi)$。由于 $x, y \notin \text{fv}(\pi)$，我们知道 $\cdot \xrightarrow{\pi} \cdot$。P-New 的前提包括 Γ_1, x: nrm$^+(T)$ y: nrm$^-(T) \vdash p_1$。对第 3 项应用归纳假设，得到 Γ_1, x: nrm$^+(T)$ y: nrm$^-(T) \vdash p_2$。结合 P-New 得出结论。规则 Act-OpenL。根据 L-OpenL，我们知道 $\Gamma_2 = x$: $!T.S$。对 P-New 应用逆推可得Γ_1, x: $!T.S, a$: nrm$^+(U)$ b: nrm$^-(U) \vdash p_1$。对 L-Send 应用逆推，E-Check和 E-Var可得 nrm$^+(U) = T$。根据第 3 项的归纳假设，得出 Γ_1, x: S, b: nrm$^-(U) \vdash p_2$。规则 Act-OpenR 类似。

规则 动作-左关闭。根据 L-Tau，$\cdot \xrightarrow{\tau} \cdot$。对 P-New 应用逆推可得 $\Gamma_1, x\colon \mathrm{nrm}^+(T)\ y\colon \mathrm{nrm}^-(T) \vdash p_1$。根据 L-Rcv，我们知道 $\mathrm{nrm}^+(T) = ?U.S$。根据 L-Rcv、L-OpenL、L-Par 以及第 3 项的归纳假设，我们得到 $\Gamma_1, x\colon S, y\colon \mathrm{nrm}^-(S)\ b\colon \mathrm{nrm}^-(U)\ \Gamma \vdash p_2$，其中 $\cdot \mid \Gamma \vdash a \Leftarrow U \mid \cdot$。应用 E-Check 和 E-Var，我们知道 $\Gamma = a\colon U$。应用 P-New 两次得出结论。规则 Act-CloseR 类似。

规范形式告诉我们，值的形式取决于类型的形式。

引理 11（规范形式）。令 $\Delta \mid \Gamma_1 \vdash v \Rightarrow T \mid \Gamma_2$。

1. 若 $T = \mathtt{Unit}$，则 $v = *$。*2.* 若 $T = U \to V$，则 $v = \lambda x\colon U.e$ 或 $v = \mathtt{rec}\ x\colon T.e$ 或 $v = \mathtt{fork}$ 或 $v = \mathtt{wait}$ 或 $v = \mathtt{terminate}$ 或 $v = \mathtt{receive}[W][X]$ 或 $v = \mathtt{send}[W][X]$ 或 $v = \mathtt{send}[W][X]\ u$ 或 $v = \mathtt{select}\ C[W]$。*3.* 若 $T = U \otimes V$，则 $v = \langle u, w\rangle$。*4.* 若 $T = \forall \alpha\colon \kappa.U$，则 $v = \Lambda\, \alpha\colon \kappa.u$ 或 $v = \mathtt{receive}$ 或 $v = \mathtt{receive}[V]$ 或 $v = \mathtt{send}$ 或 $v = \mathtt{send}[V]$ 或 $v = \mathtt{select}\ C[V]$ 或 $v = \mathtt{select}\ \overline{C}$ 或 $v = \mathtt{new}$。*5.* 若 $\Delta \vdash T \Leftarrow \mathbf{s}$，则 $v = x$。

证明。对适用于判断 $\Delta \mid \Gamma_1 \vdash v \Rightarrow T \mid \Gamma_2$ 的规则进行直接分析。$T = U \to V$ 和 $T = \forall \alpha\colon \kappa.U$ 的情形来自规则 E-Abs，E-TAbs 以及具有这两种类型的常量。会话类型（种类为 $\mathbf{s}$ 的类型）的值只有变量（表示信道端点）。

引理 12（已完成的进程是封闭的）。若 p 已完成，则 $\cdot \vdash p$。

证明（定理 *5*（进展）的证明）。对第一个假设进行互归纳，在每种情形下均成立。

规则 E-Const、E-Var、E-Abs 和 E-TAbs。常量、变量、λ 抽象和类型抽象都是值。

规则 E-Let*。由于 $e = \mathtt{let}*\ = e_1\ \mathtt{in}\ e_2$ 不是值，因此必须归约。分两种情况讨论。当 e_1 是值时，规范形式（引理 11）给出 $e_1 = *$，且 e 通过规则 Act-Let* 发生转换。否则，规则 E-Let* 的前提包括 $\Delta \mid \Gamma_1^{\mathbf{s}} \vdash e_1 \Leftarrow$ 单位类型 $\mid \Gamma_2$，由归纳法可得 $e_1 \xrightarrow{\lambda} e_1'$（因为 e_1 不是值）。因此 e 根据规则 Act-LetE* 归约。

规则 E-Rec。表达式递归 $x\colon T.v$ 是一个值

规则 E-App。由于 $e = e_1\ e_2$ 不是值，因此它必须规约。分两种情况讨论。当 e_1 不是值时，根据归纳法，e_1 会规约，因此 e 也会规约（根据规则 ActAppL）。当 e_1 是值时，规范形式（引理 11）给出一系列情况，我们逐一分析。当 e_1 是抽象时，我们进一步分两种情况。如果 e_2 是值，则 e 根据规则 Act-App 规约。否则，单调性（引理 8）给出 $\Gamma_2 \subseteq \Gamma_1^{\mathbf{s}}$，因此 Γs2。归纳假设允许我们得出结论：e 根据规则 Act-AppR 规约。rec 的情况与 λ 类似。当 $e_1 =$ 是分叉时，我们进一步分两种情况。当 e_2 是值时，e 根据规则 Act-Fork 规约。否则，利用归纳法，表达式 e 根据规则 Act-AppR 规约。当 $e_1 =$ 是等待时，我们进一步分两种情况。当 e_2 是值时，规范形式给出 $e_2 = x$，因此 e 根据 Act-Wait 规则规约。否则，单调性（引理 8）使我们可以应用归纳假设，因此 e_2 规约（因为它不是值），而且 e 根据规则 Act-AppR 规约。terminate 的子情况类似。当 $e_1 =$ 接收 $[W][X]$ 时，我们进一步分两种情况。当 e_2 是值时，规范形式给出 $e_2 = x$，因此 e 根据规则 Act-Rcv 规约。否则，单调性（引理 8）使我们可以应用归纳假设，因此 e_2

会规约（因为它不是值），而 e 根据规则 Act-AppR 规约。关于 发送$[W][X]$和 发送$[W][X]\,u$ 以及 选择 $C[W]$ 的子情况类似。

规则 E-TApp。这些情况与规则 E-App 中的情况类似。

规则 E-Pair。分三种情况讨论 $e = \langle e_1, e_2\rangle$。当 e_1 和 e_2 都是值时，e 是一个值。当 e_1 不是值时，根据归纳法可知 e_1 会归约，因此 e 可通过 Act-PairL 规则进行转换。最后，当只有 e_1 是值时，规则 E-Pair 的前提包含 $\Delta \mid \Gamma_1^{\mathbf{s}} \vdash e_1 \Rightarrow T \mid \Gamma_2$。单调性（引理 8）给出 $\Gamma_2 \subseteq \Gamma_1^{\mathbf{s}}$，因此 $\Gamma_2^{\mathbf{s}}$，于是可应用归纳假设。最后应用 Act-PairR 规则。

规则 E-Let。与 E-Let*类似。

规则 E-Match。在这种情况下，e = 将 e' 与 $\{C_i\ x_i \to e_i\}_{i\in I}$匹配。该规则的前提包含 $\Delta \mid \Gamma_{\mathbf{s}_1} \vdash e' \Rightarrow S \mid \Gamma_2$ 以及 $\mathrm{nrm}^+(S) = ?(P\ U).S'$。根据一致性（引理 7），可得 $\Delta \vdash S \Leftarrow \kappa$。分两种情况讨论。当 e' 是值时，会话类型规范化（引理 5）和规范形式（引理 11）给出 $e' = x$，并且 e 通过 Act-Match 规则归约。否则，先应用归纳法，再应用 Act-MatchE 规则即可得出结论。

规则 P-Exp。在这种情况下 $p = \langle e\rangle$。分两种情况讨论。当 e 是一个值时，由规范形式可得 $e = *$ 且 p 已完成。否则，根据归纳法 $e \xrightarrow{\lambda} e'$，我们再根据标签 λ 分三种情况讨论：

λ	LTS 规则
分叉 v	Act-Fork
(νxy): T	Act-New
σ	Act-Session
β	Act-Beta

规则 P-Par。在此情况下 $p = p_1 \mid p_2$。根据归纳法，每个 p_i 要么已完成，要么可以规约，要么是死锁。我们必须分析八种情况。当两个 p_i 都已完成时，p 已完成。当其中一个 p_i 是死锁时，p也如此。如果两个 p_i 都可以规约，则 p 通过 ACT-JoinR（或 Act-JoinL）规约。否则，当 p_1 可以规约（且 p_2 已完成）时，p 根据规则 Act-ParL 规约。最后，当 p_2 可以规约（且 p_1 已完成）时，p 根据规则 Act-ParR 规约。

规则 P-New。在此情形下 $p = (\nu xy)\ p_1$。该规则的前提是 $\Gamma, x\colon \mathrm{nrm}^+(T)\ y\colon \mathrm{nrm}^-(T) \vdash p_1$。一致性（引理 7）以及对偶性仅定义在会话类型上的事实，表明 $(\Gamma, x\colon \mathrm{nrm}^+(T)\ y\colon \mathrm{nrm}^-(T))^{\mathbf{s}}$。根据归纳法，$p_1$ 已完成、可约化或陷入死锁。由于已完成的进程是封闭的（引理 12），p_1 不可能已完成。如果 p_1 陷入死锁，那么根据定义，p 也会如此。剩下需要检查的是 p 未陷入死锁且可约化的情形。假设 $p_1 \xrightarrow{\pi} p_2$，其中 π 属于 x, y 的进展集。那么 p 按照下表进行约化。

π	LTS 规则
σ	Act-Session
分叉 v	Act-Fork
(νab): T	Act-New
$x?v \mid y!v$	Act-Msg
$x?C \mid y!C$	Act-Bra
$x? \mid y!$	Act-Wait
$x?a \mid (\nu ab)y!$	a 动作-左关闭
$x?b \mid (\nu ab)y!$	b 动作-右关闭

E 嵌入上下文无关会话类型

本节非正式地探讨两个与 AlgST 和上下文无关会话类型（CFST；Almeida 等人 [2022]）相关的问题：我们能否将任意 AlgST 程序映射为等价的 CFST 程序？以及，我们能否将任意 CFST 程序映射为等价的 AlgST 程序？

这项研究必然是非正式的，因为 AlgST 和 CFST 的特性并不完全对应。AlgST 是 System F 的一种极简且大多为线性的版本，包含积类型、会话和代数协议。CFST 同样基于 System F，但具有更丰富的种类结构，支持非线性和线性特性、变体类型和记录类型，以及完整的一组上下文无关会话原语。

为了进行有意义的比较，我们聚焦于 CFST 的线性片段 Almeida 等人[2022] 采用 Costa 等人 [2022],的记法，将其限制为成对形式，并忽略 De Bruijn 指数。在 AlgST 中，我们只考虑不对协议进行多态化的类型，因为 CFST 中没有与之对应的形式。我们添加线性代数数据类型，以处理 CFST 中的变体类型。

将非参数化代数会话转换为（线性）上下文无关会话 不对协议类型进行参数化的代数会话类型（AlgST_0）到上下文无关会话类型（CFST）的转换由 $\wr T \wr$给出，它仅针对范式类型定义：

$$\wr \mathtt{Unit} \wr = \mathtt{unit} \quad \wr T \to U \wr = \wr T \wr \to \wr U \wr \quad \wr T \otimes U \wr = \wr T \wr \times \wr U \wr \quad \wr \forall \alpha : \mathbf{t}.T \wr = \forall \alpha : \mathbf{t}^{\mathbf{lin}}.\wr T \wr$$

$$\wr \alpha \wr = \alpha \qquad \wr ?T.U \wr = ?\wr T \wr ; \wr U \wr \qquad \wr !T.U \wr = !\wr T \wr ; \wr U \wr \qquad \wr \mathtt{End}_! \wr = \mathtt{skip} \qquad \wr \mathtt{End}_? \wr = \mathtt{skip}$$

$\wr ?\ \overline{P}\,\overline{U}.S \wr = P_U$； $\wr S \wr$其中 $P_U \doteq \&\{C_i : \wr \S(-(\overline{T_i[U/\alpha]})) \wr \text{。结束}_! \}_{i \in I}$对于协议 $\rho\ \overline{\alpha} = \{C_i\ \overline{T_i}\}_{i \in I}$

$\wr !\ \overline{P}\,\overline{U}.S \wr = P_U$； $\wr S \wr$其中$P_U \doteq \oplus\{C_i : \wr \S(+(\overline{T_i[U/\alpha]})) \wr \text{。结束}_! \}_{i \in I}$对于协议 $\rho\ \overline{\alpha} = \{C_i\ \overline{T_i}\}_{i \in I}$

函数类型和消息交换可以自然地映射到与之对应的形式。会话的结束映射为跳过，而代数协议则会转换为定义为内部选择或分支的类型名称，具体取决于它们出现在发送位置还是接收位置。选择标签与协议构造子一致。

我们声称，如果 $T \equiv_A U$，那么 $\wr T \wr \simeq \wr U \wr$。

嵌入代数会话的线性上下文无关会话 在这一方向上，我们省略多态，以获得更简洁明了的记法。我们的方法也适用于多态情形，但代价是需要承担较高的记法开销。

CFST 的线性和单态片段到 AlgST 的转换分为函数片段和会话类型片段两部分，这两部分相互定义，并展示于图 13。上下文无关会话类型 T 到代数设定的转换由 $[\![T]\!]$ 给出。函数算子在代数侧有自然的对应项。转换到同构递归设定时，函数递归类型表示为一个带有显式展开的数据类型。变体类型映射为数据类型，其构造函数就是相应的标签。会话类型通过协议定义具体化：每种会话类型都会生成一个协议，该协议源自将类型的顺序组合转换为一系列种类为$\mathbf{p}$的（代数）类型。这一转换由函数 $(\!|\cdot|\!)$ 控制，它将 跳过映射为空序列，并利用极性来约束通信方向。在会话类型片段中，递归类型

CFST to AlgST: Translation of the functional fragment $\boxed{[\![T : \mathsf{T}^{\mathsf{lin}}]\!] = T : \mathbf{t}}$

$$[\![\mathtt{unit}]\!] = \mathtt{Unit} \qquad [\![T \to U]\!] = [\![T]\!] \to [\![U]\!] \qquad [\![T \times U]\!] = [\![T]\!] \otimes [\![U]\!]$$

$$[\![\langle \ell : T_\ell \rangle_{\ell \in L}]\!] = X_L \text{ where } \mathtt{data}\ X_L = \{\ell\ [\![T_\ell]\!]\}_{\ell \in L}$$

$$[\![X : \kappa]\!] = X \text{ for } X \doteq T \text{ and } \kappa \neq \mathsf{S}^{\mathsf{lin}} \text{ where } \mathtt{data}\ X = \mathsf{Unfold}_X\ [\![T]\!]$$

$$[\![T : \mathsf{S}^{\mathsf{lin}}]\!] = !X_T.\mathtt{End}_! \text{ where } \mathtt{protocol}\ X_T = X_T\ (\!|T|\!) \text{ and } X_T \text{ is a fresh name.}$$

CFST to AlgST:Translation of the session typed fragment $\boxed{(\!|T : \mathsf{S}^{\mathsf{lin}}|\!) = \overline{T : \mathbf{p}}}$

$$(\!|\mathtt{skip}|\!) = \varepsilon \qquad (\!|!T|\!) = [\![T]\!] \qquad (\!|?T|\!) = -[\![T]\!] \qquad (\!|T;U|\!) = (\!|T|\!)(\!|U|\!)$$

$$(\!|\oplus\{\ell : T_\ell\}_{\ell \in L}|\!) = X_L \text{ where } \mathtt{protocol}\ X_L = \{\ell\ (\!|T_\ell|\!)\}_{\ell \in L}$$

$$(\!|\&\{\ell : T_\ell\}_{\ell \in L}|\!) = -X_L \text{ where } \mathtt{protocol}\ X_L = \{\ell\ (\!|\mathtt{dual}\ T_\ell|\!)\}_{\ell \in L}$$

$$(\!|X : \mathsf{S}^{\mathsf{lin}}|\!) = X \text{ for } X \doteq T \text{ where } \mathtt{protocol}\ X = \mathsf{Unfold}_X\ (\!|T|\!)$$

图 13：将线性上下文无关会话类型转换为代数会话类型

选择由协议定义表示。类型名称被视为生成式名称，我们放宽 AlgST 的限制，允许在不同协议定义中重复使用重载的构造器名称。

接下来需要在 CFST 依赖等递归类型的类型等价之处定义合适的适配器。为此，我们使用 Costa 等人提出的规则系统 [2022]，生成作为等价性证明见证的同构。构造此类同构的难点在于将其扩展到会话类型。为简化起见，我们只考虑同构 iso : $[\![T]\!] \to [\![U]\!]$ 的一侧，其中 $T \simeq U$ 为会话类型：

```
iso t = let (u, u') = new [[U]] in fork $ δ t u' ▷ u
```

此定义依赖于一个进程 δ : $[\![T]\!] \to$ 对偶 $[\![U]\!] \to$ 单位类型，该进程消耗两个（对偶）信道端点，并实现一个转发器，以弥补 T 与 U 之间的差异。见证进程 δ 根据规则，为每对等价的会话类型定义。我们在表 1 中简要展示了一些见证进程，并对该同构作更详细的分析。

为求简洁，在表 1 的所有情形中，我们省略了由 : S 的定义所产生的初始管理性 匹配 和选择步骤。对于 $T \simeq U$ 中 $T =$ 跳过且 $U =$ 跳过的情形，其形式化定义如下：

```
δ tu = 匹配 u with {XU u → let t
  = 选择 XT t in 等待 u▷终止 t }
```

进程 δ 匹配，在通道 u 上执行 X_U，并在通道 t 上执行选择，这正是由 [跳过] 得出的协议定义所规定的（见图 13）。根据 [跳过] 和 E-Match的定义，我们剩下类型为结束$_?$，在 u 和 结束$_!$，在 t，因此我们在上等待，并在 t 上终止。

一般而言， δ 会插入转换到等递归设定时所要求的显式展开。我们以等价关系 $Y \simeq U$为例展示我们的方法，其中类型 Y由 $T, Y \doteq T$ 定义。 δ 的定义遵循 Costa 等人提出的等价规则 [2022]，如下所示。用于嵌入 iso：$[\![Y]\!] \to [\![U]\!]$ 的见证进程δ 以一个见证 $T \simeq U$ 的进程 δ_1 为基础，定义如下：

```
δ y u = match u with {
  XU u → let y = select XY y in
         let t = select UnfoldY y in
         δ1 t u }
```

$$\frac{Y \doteq T \qquad T \text{ 对照} \qquad \delta_1 \colon T \simeq U}{\delta \colon Y \simeq U}$$

根据 $[\![Y \colon \mathsf{S}^{\mathsf{lin}}]\!]$的定义，进程 δ 在通道 u 上与 X_U 匹配，并在通道 y上选择 X_Y，这符合由 $[\![\mathrm{Y}]\!]$ 产生的协议定义（见图 13）。随后，选择由 $(\!|Y|\!)$ 引入的显式展开 Unfold_Y，并使用 δ_1 消耗 t 和 u 处的剩余项。对于 $X \simeq U$且 $X \doteq T$的情况，我们有一个见证 δ_1来证明 $T \simeq U$。我们选择由 $(\!|X|\!)$ 引入的显式展开 Unfold_X，然后使用 δ_1 消耗 t 和 u处的剩余项。

对于 $!T; V \simeq !U; W$ 的情况，我们不仅有一个见证 δ_1来消耗通道 v和 w的剩余部分，还有一个函数 $\theta \colon [\![T]\!] \to [\![U]\!]$，用于证明由 $T \simeq U$产生的同构。函数片段中类型之间同构的见证函数见表 2。给定 δ_1 和 θ，在初始的管理性匹配 和选择之后，进程 δ 接收s 在通道 w上接收 u，使用 θ^{-1} 转换 u，并通过通道 v发送结果。进程 δ_1 消耗 v 和 w的剩余部分。

为了见证与 $!T; V \simeq !U$where V 相对应的同构（关于已终止的会话，更多细节请参见 Costa 等人 [2022]），我们给定一个进程 σ，它会消费对 V 进行同构递归解释后产生的显式展开。表 3 展示了已终止会话的消费者进程 σ。

表 1 至表 3 中的其他情况也遵循类似的推理。对于单态类型，省略的情况同样适用这一方法。对于多态片段，我们需要考虑一种读取器单子，在其中可以存储从类型变量到相应进程的映射；这些进程用于消费相应的通道，或转换相应的值，具体取决于类型变量的种类是否为s。例如，在多态（非会话类型化）变量的公理 $n \simeq n$中，我们可以将 θ 视为恒等映射，但我们不知道应为绑定变量指定什么类型：$\theta = \lambda x \colon ?\ .x$。而要消费两个由多态会话类型变量标注的通道，我们就需要知道如何消费两个端点。用于消费或转换由多态变量标注的值的进程，则需要提供给 δ。对于多态片段中的类型翻译，我们需要收集类型中出现的自由变量，并使用参数化的数据或协议定义：

$$[\![X \colon \kappa]\!] = X\ \overline{\alpha} \text{ for } X \doteq T \text{ and } \kappa \neq \mathsf{S}^{\mathsf{lin}} \text{ where } \mathtt{data}\ X\ \overline{\alpha} = \mathsf{Unfold}_X\ [\![T]\!] \text{ and } \mathrm{fv}(T) = \overline{\alpha}$$

$$(\!|X \colon \mathsf{S}^{\mathsf{lin}}|\!) = X\ \overline{\alpha} \text{ for } X \doteq T \text{ where } \mathtt{protocol}\ X\ \overline{\alpha} = \mathsf{Unfold}_X\ (\!|T|\!) \text{ and } \mathrm{fv}(T) = \overline{\alpha}$$

尽管我们相信多态类型不会给此分析带来任何问题，但我们决定保持记法简单，并聚焦于单态片段。

在本分析中，我们只提供了单向同构的见证。对于同构的另一个方向，我们可以采用类似的方法来定义 $\mathrm{iso}^{-1} \colon [\![U]\!] \to [\![T]\!]$，然后证明 iso 和 iso^{-1} 复合后得到恒等映射。

等价规则 $T \simeq U$: S^{lin}	同构见证 $[\![T]\!] \to$ 对偶 $[\![U]\!] \to$ 单位类型
δ: 跳过 $\simeq$ 跳过	δ tu = 等待 u ▷终止 t
$\dfrac{X \doteq T \qquad T\ \mathsf{contr} \qquad \delta_1\colon\ T \simeq U}{\delta\colon\ X \simeq U}$	δ xu = let t = 选择 UnfoldX x 在 δ1 t u
$\dfrac{X \doteq T \qquad T\ \mathsf{contr} \qquad \delta_1\colon\ T;\ V \simeq U}{\delta\colon\ X;\ V \simeq U}$	δ xu = 令 t = 选择 UnfoldX x 在 δ1 t u
$\dfrac{\theta\colon\ T \simeq U \qquad \delta_1\colon\ V \simeq W}{\delta\colon\ !\,T;\ V \simeq !\,U;\ W}$	δ vw = 设 (u , w) = 接收 w 在 设 t = θ^{-1} u 令 v = 发送 tv 于 δ1 v w
$\dfrac{\theta\colon\ T \simeq U \qquad \sigma\colon\ V\checkmark}{\delta\colon\ !\,T;\ V \simeq !\,U}$	δ t u = 让 (u' , u) = 接收 u 在 令 t' = θ^{-1} u' 令 v = 发送 t' t 在 令 v = σ v 在 等待 u ▷终止 v
$\dfrac{\theta\colon\ T \simeq U}{\delta\colon\ !\,T \simeq !\,U}$	δ t u = 让 (u' , u) = 接收 u 在 设 t' = θ^{-1} u' 在 令 t = 发送 t' t 于 等待 u ▷终止 t
$\dfrac{\delta_\ell\colon\ T_\ell \simeq U_\ell \qquad (\forall \ell \in L)}{\delta\colon\ \oplus\{\ell\colon T_\ell\}_{\ell\in L} \simeq \oplus\{\ell\colon U_\ell\}_{\ell\in L}}$	δ tu = 匹配 u 与 { k u → 令 t = 选择 kt 于 δk t u}
$\dfrac{\delta_1\colon\ \oplus\{\ell\colon\ T_\ell;\ U\}_{\ell\in L} \simeq V}{\delta\colon\ \oplus\{\ell\colon\ T_\ell\}_{\ell\in L};\ U \simeq V}$	δ t u = δ1 t u
$\dfrac{\delta_1\colon\ T \simeq U}{\delta\colon$ 跳过; $T \simeq U}$	δ t u = δ1 t u

表 1：消耗两个对偶会话端点的进程。

等价规则 $T \simeq U$: κ，其中 $\kappa \neq S^{\mathsf{lin}}$	同构见证 θ: $[\![T]\!] \to [\![U]\!]$
θ: $()_{\mathsf{lin}} \simeq 0_{\mathsf{lin}}$	θ = λ x: 单位类型 .x
$\dfrac{\theta_1\colon\ T \simeq U \qquad \theta_2\colon\ V \simeq W}{\theta\colon\ T \to V \simeq U \to W}$	θ t = λ x: $[\![$U $]\!].\theta$2 $ t $ θ^{-1} x
$\dfrac{\theta_1\colon\ T \simeq U \qquad \theta_2\colon\ V \simeq W}{\theta\colon\ T \otimes V \simeq U \otimes W}$	θ t = <θ1 $ fst t,θ2 $ snd t>
$\dfrac{X \doteq T \qquad T\ \mathsf{contr} \qquad \theta_1\colon\ T \simeq U}{\theta\colon\ X \simeq U}$	θ x = 令 t = 选择 UnfoldX x 于 令 u = θ1 t 于 u

表 2：非会话类型表达式的转换。

已终止谓词见证 T: $\mathsf{S}^{\mathsf{lin}}$	σ: $[\![T]\!] \to$ 结束!
σ: 跳过✓	σt = t
$\dfrac{\sigma_1 : T\checkmark \qquad \sigma_2 : U\checkmark}{\sigma : T;U\checkmark}$	σ t = 设 t' = σ1 t 于 σ2 t'
$\dfrac{T \doteq U \qquad \sigma_1 : T\checkmark}{\sigma : X\checkmark}$	σx = 令 t = 选择 UnfoldX x 于 σ1 t

表 3：消费翻译所生成的显式展开的进程。