arXiv:2303.01278v1 [cs.PL] 2 Mar 2023

内在类型化会话与回调

函数式珍珠

PETER THIEMANN，弗赖堡大学，德国

所有会话类型的形式化都依赖线性类型来保证可靠性，因为会话类型化通信通道必须在每次操作时改变其类型。会话类型的嵌入式语言实现也遵循这一做法。它们要么依赖巧妙的类型构造来静态保证线性性，要么依赖用于近似线性性的运行时检查。

我们提出了一种新的会话类型嵌入式语言实现，其灵感来自控制反转设计原则。采用我们的方法，所有应用程序从本质上都具有会话类型，并且从构造上无法破坏线性性。线性性仍是一个证明义务，但只需针对一个微小的封装库完成；在构建该库时即可一次性完成证明。

我们证明，所提出的设计可扩展到会话类型系统的广泛特性：分支、递归、多通道会话和高阶会话，以及上下文无关会话。多通道扩展提供了一种会话类型的嵌入式实现，并通过构造保证无死锁性。

本文所报告的开发工作完全由经过类型检查的 Agda 代码支持。

附 加关键词和短语：会话类型、领域特定语言、依赖类型、Agd a

ACM 引用格式：
Peter Thiemann。2023。具有回调的内在类型化会话：函数式珍珠。1，1（2023年3月），27页。
https://doi.org/XXXXXXX.XXXXXXX

1 引言

会话类型为并发编程系统中的协议提供了一种类型规约。每个会话类型算子都描述通信方向和载荷的类型（即传输值）。此外，还有一些方法可用于指定协议的控制结构，如顺序执行、分支和循环。

Honda 等人 [18, 19, 52] 最初在 π 演算的背景下开始研究会话类型 [38]。后来，这些概念被应用于其他编程范式，例如函数式编程 [17, 60] 和面向对象编程 [16, 46]。

早期研究主要关注二元会话类型，即涉及两个参与方的协议。随后，人们提出了多方会话类型 [20]，放宽了参与方必须为两方的限制。Betty 项目促成了多篇综述文章和著作的出版，为该领域提供了良好的概述 [2, 4, 15, 24]。本文主要讨论二元情形，但也从多方会话类型中汲取了一些灵感。

会话类型通过 Curry–Howard 对应与线性类型密不可分地联系在一起。这一联系已被发现并进行了相当深入的研究 [5, 6, 61]，并产生了诸多影响 [3]。此外，这种联系不仅具有理论意义，还影响了实现，并且（可以说）阻碍了广泛采用。语言

作者地址：Peter Thiemann，thiemann@acm.org，弗赖堡大学，德国。

允许免费制作本作品全部或部分内容的数字或纸质副本，供个人或课堂使用，但前提是不得为盈利或商业利益制作或分发副本，且副本须保留本声明及首页上的完整引文。本作品中由非 ACM 所有的组件，其版权必须得到尊重。注明出处后可以摘录。若要以其他方式复制、重新出版、发布到服务器或分发至列表，则须事先获得具体许可和/或支付费用。请向 permissions@acm.org 申请许可。© 2023 计算机协会。XXXX-XXXX/2023/3-ART $15.00https://doi.org/XXXXXXX.XXXXXXX

围绕会话类型构建的通常是专用语言，它们贯彻线性性，因此其类型检查器会拒绝违反线性性的行为。这样的例子很多：Links [36],SEPI [14, 59],SILL[56], C0 [62], 等等。尽管这些语言及其实现推动了研究并鼓励了实验，但它们并未得到广泛使用。

为了推动会话类型的应用，人们投入了大量工作，致力于将会话类型嵌入主流语言中，而其中大多数语言并不原生支持线性性。对于函数式语言（如 Haskell [40, 45, 49] 和 OCaml [26, 43],）以及面向对象语言（如 C# [29], 、Scala [50], 和 Java [23]），都有许多此类嵌入的实例。大多数方法忽略了编译时线性性这一问题；有些方法则完全忽略了它。有些方法（例如 [43]）依赖运行时检查，另一些则依赖使用透镜 [25] 或单子 [45]对线性性进行编码。还有一些扩展语言配有独立的检查器，可将会话 [41] 或更一般的类型状态 [32] 添加到底层 Java 程序中。我们将在相关工作（第 9 节）中评论一些近期依赖 Rust 的实现。

近期关于多方会话类型的工作 [39, 64] 提出了一种不依赖线性性的替代方法。该方法受控制反转这一设计原则的启发，这一原则对于程序员来说因 GUI 编程而广为熟悉。这些工作中描述的系统会将多方会话类型的描述转换为一个库，该库封装了所有通信的实现。对于每个通信操作，该库都会提供一个接口，程序员可在其中为该特定操作指定一个回调函数。

为阐明这一思想，我们给出一个非常简单的示例，并在第 2 节中，结合函数式语言的背景继续展开和扩展。与所引用的工作不同，我们的工作以及此示例都依赖于二元会话类型。我们从以下类型语法开始 T 和会话类型 S。

$$T ::= \mathsf{int} \mid \mathsf{bool} \qquad\qquad S ::= !T{\cdot}S \mid ?T{\cdot}S \mid \mathsf{end}$$

会话类型 $!T{\cdot}S$($?T{\cdot}S$) 描述一个已准备好发送（接收）负载类型为 T 的值、并将继续作为 S的通道。会话类型 end 描述一个只能关闭的通道。

传统设置 (参见 [17]) 会话类型通信的传统接口由诸如以下原语操作构成

$$\mathtt{send} : !T{\cdot}S \otimes T \multimap S \qquad \mathtt{recv} : ?T{\cdot}S \multimap (T \otimes S) \qquad \mathtt{close} : \mathsf{end} \multimap ()$$

用于在通道上发送、从通道接收以及关闭通道。关键的观察是，类型系统必须以线性方式处理通道，才能确保协议保真性。程序通常如下所示：

negp - server : ? int .! int . end ⊸() negp - server c0 =-- c0 : ? int .! int . end令 (x, c1) =接收 c0 于 -- c1 : ! int . end令 c2 = 发送 (c1 , -x) 于 -- c2 : end关闭 c2

根据线性性，recv 操作会消耗 c0；否则，还可以再次对 c0 执行 recv，从而破坏协议。类似的论证也适用于 c1 和 c2，例如，通道 c2 一旦关闭，就不能再次关闭。

回调方法 本文提出的用于会话类型通信的回调接口由两部分组成。

（1）一种命令数据类型 Cmd，由应用状态 A 和会话类型索引。该数据类型构成了通信函数式程序的内禀会话类型编码。

（2）一个封装的解释器 exec，用于执行命令。[1]

```
data 命令 (A : 集合) : 会话 → 集合 where CLOSE : 命令 A
end SEND : (A  → A × T⟦ T ⟧) → 命令 A S  → 命令 A (! T · S)
RECV : (T⟦ T ⟧ → A  → A) → 命令 A S  → 命令 A(? T · S)
```

SEND 命令通过一个回调，从应用状态中获取要发送的值。类似地，RECV 命令通过一个回调，将接收到的值注入应用状态。两者都接受一个类型为 命令 *A S*、用于处理续延会话的命令 *S*。 CLOSE 命令表示协议结束。

程序表示为一个类型为 命令的值。它看起来与传统形式相似，我们选择 ℤ整数作为应用状态。这种编码高度依赖函数式编程的核心思想：将函数（回调）作为一等值。[2]

```
negp-command :  命令 ℤ (? 整数 · ! 整数 · end)negp-command  =
RECV (λ x a  → x) $ SEND (λ a  →  ⟨ a , -a ⟩ ) $ CLOSE
```

最简单的解释器接收一条命令、合适的初始应用状态和一个无类型通道，并返回一个输入输出动作，该动作会产生最终的应用状态。

```
exec :  命令A S → A → 通道 → 输入输出 A
```

这个解释器在封装库中一次性实现。从某种意义上说，它构成了我们方法的可信计算基，因为我们有义务证明它会根据通道的会话类型索引，在通道上执行命令。

1.1 贡献

- 我们在依赖类型函数式编程的背景下，提出了针对二元会话类型的回调方法。我们在 Agda 语言中将其作为概念验证规范加以实现，但我们预计，这一开发成果可以通过编译或翻译移植到 Haskell [10]。
- 我们通过在一个小型解释器中验证命令执行的线性处理，来确保会话处理的线性性；该解释器构成了我们方法的可信计算基。宿主语言的类型系统无需支持线性类型，也无需使用巧妙的类型构造来模拟线性性。
- 我们证明，该方法可扩展至大多数常见的会话类型构造：分支（第3节）、递归（第4节）、多通道和高阶会话（第7节）。在第4.3 节中，我们提出了对基于应用程序接口的递归处理方法的一项新颖且重要的改进。
- 扩展为可在多个通道上运行意义重大，而且基本上与其他特性正交。我们的方法受 Wadler 的 GV 演算 [61] 启发，因此能够从构造上保证程序无死锁。
- 我们提出一种适用于分支会话类型的新型动态选择操作（第3节）。
- 我们将回调方法扩展到上下文无关会话类型（包含分支和递归）；这也要求采用比常规会话类型更精巧的、依赖类型编码的命令表示（第6节）。
- 对于单子爱好者，第 5 节介绍了一种采用单子式编码实现回调的版本。

[1]函数 T⟦_⟧ : 类型 → 集合 将类型语法映射为其作为 Agda 类型的解释。更多详情请参见第 2 节。 [2]运算符 _$_ 表示中缀函数应用。它按从右向左的方式结合。（第二个 _$_ 可以省略。）

本文档的源文件包含若干文学化 Agda 脚本，这些脚本将作为匿名补充材料提交（之后会整理成一个制品）。所有以彩色排版的代码行都已通过 Agda 检查。目前，解释器基于一组精简的单子式输入输出操作应用程序接口实现，用于操作无类型通道。这个应用程序接口可以使用 Agda 的外部函数接口在 Haskell 中实现。[3]

作为一篇函数式珍珠，本文专注于库的设计，不包含对库解释器所承担证明义务的形式化证明（即多通道情形下的线性性和无死锁性）。第8 节中的讨论提出了一些处理这一任务的建议。

掌握 Agda 并不是理解本文的硬性要求。我们会在遇到 Agda 特有的功能时加以解释，力求让熟悉 Haskell 的读者也能理解代码。

2 有限非分支会话类型

让我们直接从最简单的实例——有限非分支简单会话类型——开始，以传达这一方法的要旨。后续章节将展示如何加入会话类型的大多数常见功能。

二元会话类型描述两个对等方之间的双向通信，我们称它们为服务器和客户端。会话类型附加在通信信道的类型上。[4]

```
数据类型: 集合 其
中整数 : 类型布尔
值:  类型

数据 会话 :  集合 其中
  !_-_ :  类型  →会话  → 会话?_-_ :
  类型  →  会话  →  会话结束 :  会
  话
```

这些 Agda 类型对应于会话类型的标准语法，其中 T 是可传输负载值的类型，而 S 是会话的类型。

$$T ::= \mathsf{int} \mid \mathsf{bool} \qquad\qquad S ::= !T{\cdot}S \mid ?T{\cdot}S \mid \mathsf{end}$$

会话类型 $!T{\cdot}S$（$?T{\cdot}S$）描述了一个已准备好发送（接收）负载类型为 T 的值，然后继续执行 S。会话类型 结束 描述了一个只能关闭的通道。

以下是两个会话类型示例：分别是二元运算和一元运算服务器的类型。

```
binaryp  = ?整数 · ?整数 · ! 整数 ·
结束unaryp  =? 整数 · ! 整数 · 结束
```

在 GV（一种受到广泛研究的函数式会话类型理论）中， [17], 有用于发送和接收值以及关闭通道的原语，其类型如下：

$$\mathsf{send} : (!T{\cdot}S \otimes T) \multimap S \qquad \mathsf{recv} : ?T{\cdot}S \multimap (T \otimes S) \qquad \text{关闭:}\ \ \text{结束} \multimap ()$$

这些类型表明，我们必须以线性方式处理会话类型的通道值：send 操作会消耗 一个已准备好发送的通道，并将其与载荷配对，然后返回处于某个状态的通道

[3]这是一项尚未完成的编程练习，我们计划在提交制品时完成。 [4]Agda 支持混合固定语法，标识符中的下划线表示参数的位置。例如，!_-_ 和 ?_-_ 是接受两个参数的运算符。我们声明这些运算符向右结合，以省去括号。

该状态由 *S*描述；recv 操作会消耗 一个已准备好接收的通道，并返回一个由接收到的值和更新后的通道组成的对；close 操作会消耗 该通道，并返回一个 unit 值。为保证可靠性，必须强制执行这种线性性。

在这项工作中，我们采用了一种受回调式编程启发的不同方法。不再向程序员提供发送和接收原语，而是要求程序员通过实现一个命令值来定义“应用逻辑”，其类型命令由会话类型索引。此定义依赖于将类型解释为 Agda 类型。

```
T⟦_⟧: 类型 → 集
合 T⟦整数 ⟧ = ℤT⟦
布尔值 ⟧ = 布尔值

数据 命令 (A: 集合) : 会话 → 集合 其中
  CLOSE : 命令 A结束 SEND : (A → A × T⟦ T ⟧) → 命令
  A S → 命令 A (! T · S) RECV : (T⟦ T ⟧ → A → A) → 命
  令 A S → 命令 A (? T · S)
```

在此定义中，类型参数 *A* 体现了应用状态。每个SEND 命令都接受一个状态变换器，用于从当前应用状态中提取要发送的值；每个RECV命令都接受一个由接收值索引的状态变换器；CLOSE 命令会终止会话。事实上，我们可以通过作用于应用状态的状态单子中的操作来提供应用逻辑 *A*。我们将把转向单子式接口推迟到第5节，届时我们将掌握全貌。

继续这个例子，我们定义实现以下协议服务器的命令：unaryp和 binaryp，其操作分别实例化为取反和加法。

```
negp-command : 命令 ℤ (?整数 · ! 整数 · end)negp-command =
RECV (λ x a → x) $ SEND (λ a → 〈 a, - a 〉) $ CLOSE
```

```
addp-命令: 命令 ℤ binaryp
addp-command = RECV (λ x a → x) $ RECV (λ y a → y + a) $ SEND (λ a → 〈 a, a〉) $ CLOSE
```

为清晰起见，我们使用显式的 λ抽象，而不采用花哨的缩写和库中的组合子；变量 *a* 表示应用状态， *x* 和 *y* 分别表示从通道接收的值。对于无点式定义的爱好者，我们使用标准库中的函数，给出一个更简洁的加法命令版本：

```
addp-command′ : 命令 ℤbinarypaddp-command′ = RECV const $
RECV _+_ $ SEND <id ,id > $CLOSE
```

为了执行命令，我们编写了一个解释器，它依赖于 输入输出 单子提供的原语操作。我们的方法的关键在于，该解释器是唯一一个需要证明其以线性方式处理通道的定义。

```
公设
  通道: 集合 原语接受 : 输入输出 通道 – 接受连接，返回新通道原语关闭 :
  通道 → 输入输出 ⊤ – 关闭连接原语发送 : A → 通道 → 输入输出 ⊤ – 发
  送 A 类型的值原语接收 : 通道 →输入输出 A – 接收 A 类型的值
```

此应用程序接口应当一目了然。[5] 它声明了一种无类型原始通道的抽象类型，并提供接受连接、关闭通道，以及通过通道发送和接收值的操作。它略过了序列化等问题，这些问题可以通过使用类型类约束来解决，例如在 Haskell A 中对 原语发送 和 原语接收使用 Serialize。

解释器本身通过对类型进行归纳来定义 命令。实际计算在 输入输出 单子中进行，并使用 do记法表达，这两者在 Haskell 中都很熟悉。

```
exec : 命令 A S → A → 通道 → 输入
输出 A exec CLOSE 状态 ch = do 原
语关闭 ch pure 状态 exec (SEND
getx cmd) 状态 ch = do let 〈状态′,
x〉 = getx 状态 原语发送 x ch
exec cmd 状态′ ch exec (RECV
putx cmd) 状态 ch = do x ←原语接
收 ch let 状态′ = putx x 状态 exec
cmd 状态′ ch
```

要实际运行服务器，还需要提供一个包装器，用于接受连接并调用解释器。

```
记录 接受 A S: 集合 其中构造
器 ACC字段cmd : 命令 A S

接受器 : 接受 A S → A → 输入输出 A
接受器 (ACC 命令) a = primAccept »= exec 命令 a
```

检查解释器后，我们终于看到了完整的单子结构。我们需要一个单子变换器栈，顶部是用于应用状态的状态单子，其上是提供通道的读取器单子，最底层是输入输出单子。

我们对解释器的证明义务归结为：验证每个命令的解释都会执行由相应会话类型算子指定的单个通信操作。会话类型所要求的正确顺序执行由 输入输出 单子所施加的顺序约束保证。事实上，正是这一观察促使我们在函数式语言中为基于状态的操作 API 使用单子 [28]。

3 选择与 CHOICE

在我们的开发中加入分支非常直接。标准的会话类型理论允许使用以下语法在有限集的标签上进行分支：

$$S ::= \cdots \mid \oplus\{\ell : S_\ell \mid \ell \in L\} \mid \&\{\ell : S_\ell \mid \ell \in L\}$$

这里 L 是一个有限且非空的标签集合，每次使用该类型运算符时都可以选择不同的标签。类型构造子 $\oplus$ 对应程序的内部选择 。select 原语会发送类型中可用的某个标签（例如 $\ell \in L$），并根据 S_ℓ继续：

$$\texttt{select}\ \ell : \oplus\{\ell : S_\ell \mid \ell \in L\} \multimap S_\ell$$

[5]类型 ⊤ 类似于 Haskell 的单位类型，其中只有一个元素tt。

类型构造子 & 对应于外部选择。原语分支会接收类型中提到的某个标签，并根据该标签选择一个延续。在和类型与线性性的支持下，该原语可以按如下方式进行类型检查 [44]。

$$\mathsf{branch} : \&\{\ell : S_\ell \mid \ell \in L\} \multimap +\{\ell : S_\ell \mid \ell \in L\}$$

我们在 Agda 中的建模扩展了第 会话、命令 和 exec 节中的定义： k大小为 有限类型 $k = \{0, \ldots\ldots, k-1\}$ 的标签集由类型表示，而备选延续会话则由从标签到 会话 的函数表示（与会话向量同构，参见第8.1节）。

```
数据 会话 : 集合 其中
  ⊕′ : (Si : (i : 有限类型 k) → 会话) → 会话&
  ′ : (Si : (i : 有限类型 k) → 会话) → 会话

数据 命令 (A: 集合) : 会话 → 集合 其中
  SELECT : ∀ {Si} → (i : 有限类型 k) → 命令 A (Si i) → 命令 A (⊕′ Si)
  CHOICE : ∀{Si} → ((i : 有限类型 k) → 命令 A (Si i)) → 命令 A (&′ Si)
```

SELECT 和 CHOICE 命令仅在括号的位置上有所不同： SELECT接受一个标签以及与该标签对应的命令，而 CHOICE 接受一个将标签映射到其命令的函数。这两种命令都不需要应用状态：选择是静态的，不依赖当前状态；一旦作出外部选择，所选命令便可在应用状态中体现这一选择。

```
exec : 命令 A S → A → 通道 → 输入输出 A

exec (SELECT i 命令)状态 ch = do
  原语发送 ich
  exec 命令 状态 ch

exec (CHOICE cont)state ch =
dox ←原语接收 chexec (cont
x)state ch
```

扩展我们的运行示例，我们定义算术服务器的类型，该服务器在二元运算和一元运算之间提供选择。此定义使用智能构造器 & 实现外部选择；该构造器接收一个会话类型向量，并将其转换为相应的函数。[6]

```
算术谓词 = & [binaryp , unaryp ]
```

服务器的命令以显然的方式扩展。向量技巧不适用于 Cmd 构造器 SELECT 和 CHOICE ，因为它们的参数是依赖函数，而对于不同的参数，这些函数通常会返回不同的类型。

```
arithp-command ： Cmd ℤ
arithparithp-command = CHOICE λ 其
中zero →addp-command(suczero) →
negp-command
```

最后，我们注意到，所提出的接口支持比标准会话类型更具动态性的选择算子。标准的 select 算子以一个固定的标签作为索引

[6]此转换 tion 依赖于大小为 k 的向量与定义域为有限类型 k的非依赖函数之间的同构。

在编译时，我们可以提供一个动态选择器命令，其标签由回调在运行时计算：getl：

```
数据 命令 (A: 集合): 会话 → 集合 其中
  DSELECT : ∀ {Si} → (getl : A → A × Fin k)
                   → ((i : Fin k) → Cmd A (Si i))
                   → Cmd A (⊕′ Si)
```

将 exec 扩展到此命令非常简单。此命令的类型仍有改进空间。我们将在第6节回到这一问题。

4 循环往复

递归类型是会话类型的常见特征。对于需要反复执行相同功能的服务器，要对其协议进行建模，就必须使用递归类型。我们的示例将是一个允许客户端反复执行一元操作，直到客户端退出的服务器。

笔-a 递归类型的语法依赖于类型变量和一个 μ 运算符，如下所示：[7]

$$S ::= !T{\cdot}S \mid ?T{\cdot}S \mid \mathsf{end} \mid \mu\, X.S \mid X \mid \ldots$$

μ运算符的预期语义是，$\mu\, X.S$等价于 $S[X \mapsto \mu\, X.S]$，即展开；在展开时，我们用递归类型本身替换其主体中的变量 X。

对于 Agda 形式化，我们采用约束变量的标准德布鲁因编码。参数 n 表示会话类型中作用域内的类型变量数量。μ 运算符会开启一个新作用域，而‘ i 引用第 i 个变量，最内层的绑定为0。

```
数据会话(n: ℕ): 集合其中
  !_·_ : 类型 → 会话 n → 会话n ?_·_ : 类
  型 →会话 n → 会话 n 结束 : 会话 n⊕′ :
  (Si : (i : 有限类型 k) → 会话n) → 会话
  n &′ : (Si : (i : 有限类型 k) → 会话 n) →
  会话 nμ_ : 会话 (suc n) → 会话 n ‘ : 有
  限类型 n →会话 n_
```

进一步扩展这个贯穿全文的示例，我们将协议 unaryp重新定义为一个函数：它接收协议的其余部分，并将其封装为递归类型。会话类型 many-unaryp 是一种递归类型，它要么运行一元函数并递归，要么直接结束协议。我们按照第 & 节中的做法，将其定义为智能构造器：3。

```
unaryp : 会话 n →会话
nunaryp S = ?整数 · ! 整数 · S

多重一元谓词 : 会话0多重一元谓词 = μ
& [ unaryp (‘ zero) , end ]
```

[7]为避免类型变得更加复杂，我们略过递归类型的守卫性（或压缩性）问题。

$$\left|\begin{array}{ll} 0 & \mapsto \text{会话}0 \end{array}\right| \quad \left|\begin{array}{ll} 0 & \mapsto \text{会话}1 \\ 1 & \mapsto \text{会话}0 \end{array}\right| \quad \left|\begin{array}{l} \cdots \end{array}\right| \quad \left|\begin{array}{ll} 0 & \mapsto \text{会话}\ n \\ 1 & \mapsto \text{会话}\ (n-1) \\ & \vdots \\ n-1 & \mapsto \text{会话}0 \end{array}\right|$$

图 1。CmdStore 中的会话类型索引，在 1、2 个以及CmdStore 之后的 n 循环条目

4.1 命令

该 命令 类型获得一个新参数 n，以匹配用作索引的会话类型中的参数。我们只展示两个新增情况。

data 命令 (n : ℕ) (A: 集合) : 会话 n → 集合
where LOOP : 命令 (suc n) A S → 命令 n A (μ
S) CONTINUE : (i : 有限类型 n) → 命令 n A (‘ i)

有了这种类型，我们就可以实现一个服务：它会在接收到数字后反复将其相加，并且每次都将部分和作为响应发送出去。

addup-command： 命令 n ℤ S →命令 n ℤ (unaryp S)addup-command
cmd = RECV (λ x a → x + a) $ SEND (λ a → 〈 a , a〉) $ *cmd*

runningsum-command: 命令 0 ℤ 多重一元谓词
runningsum-command = LOOP $ CHOICE λ 其中
zero →addup-command (CONTINUE zero)(suczero) →
CLOSE

4.2 解释

执行带递归的命令需要在执行器中添加一个新组件。每当执行进入一个 LOOP 命令时，该组件就会将整个循环保存到栈中。当遇到 CONTINUE i 命令时，我们从栈中取出相应的循环并继续执行。设置这个新组件有一些复杂之处，我们称之为 CmdStore。

它的类型需要仔细考虑。顶层类型是一个 会话 0。当我们进入第一个循环时，它的类型是会话 0。因此，CmdStore 包含一个条目（图 1中的左列）。进入下一个循环时，会将一个类型为 会话 1 的循环压入栈中（中间列）。进入第 n个循环后，我们得到图 1右侧所示的结构。下面给出一个合适的类型定义。

CmdStore： ℕ → 集合 → 集合
CmdStore *n A* = (*i* : 有限类型 *n*) → ∃[*S*] (命令 (toℕ(opposite *i*)) *A S*)

对于某个 (*i* : 有限类型 (suc *n*))，函数 opposite *i* 返回 $n - i$（作为同一类型的元素），而函数 toℕ 将有限类型的元素注入自然数中（在类型意义上，它是一个恒等函数）。棘手的部分通过该结构上带类型的推入和弹出操作来实现。

push : CmdStore n A → 命令 n A S → CmdStore (suc n) A pop1 :
CmdStore (suc n) A → CmdStore n A pop : CmdStore (suc n) A → (i : 有限类
型 (suc n)) → CmdStore (suc (toℕ(opposite i))) A

这个push 函数接收一个CmdStore 和一个合适的命令，并返回扩展了该命令的存储（位于零位置）。pop1 函数会从CmdStore中弹出第一个条目。它用于定义函数pop的归纳步骤，该函数会从栈中弹出任意（合法）数量的条目。 *i* 这些定义很简单，但在文中省略了，因为它们需要调用一些关于从有限类型 *n* 到有限类型 (suc *n*)的单射（即恒等函数）的技术性引理。

```
燃料 = ℕ
exec : 燃料 → 命令 n A S → CmdStore n A → (init :A) → 通道 → 输入输出 A

exec g (LOOP 命令) cms 状态 ch = execg 命令 (push cms (LOOP 命令)) 状态 ch
exec {suc n} {A} zero (CONTINUE i) cms 状态 ch = pure 状态 – 注意：黑客警报！
exec {suc n} {A} (suc g) (CONTINUE i) cms 状态 ch
  包含cms i
... | 〈 _ , cmd-i 〉 = exec g cmd-i(pop1 (pop cms i)) 状态 ch
```

执行 LOOP 只需将循环压入 CmdStore ，然后执行循环体。对于 CONTINUE *i* ，则会跳转到选定的循环，并从存储中弹出所有介于其间的延续。这还揭示了另一个复杂之处。对 exec 的递归调用作用于从 CmdStore中取出的命令。这个命令与当前命令无关，因此 Agda 的终止性检查器会因此受阻。为了解决这个问题，我们引入一个 燃料 参数，并在每次执行CONTINUE时将其递减。

问题是，燃料耗尽时该怎么办？至少有三种可能 s.

(1) 我们将当前形式化视为概念验证，并让读者参考 Haskell 中的“真正”实现，在那里不需要燃料参数。我们认为实现这样的版本并无问题，但工作量会更大，因为依赖类型尚未在 Haskell 中得到充分集成。(2) 我们将这个缺陷变成一项特性，并要求每次 CONTINUE 都与通道另一端通信，交换各自的燃料级别；如果任一端燃料耗尽，就退出循环。此选择意味着我们必须为 CONTINUE 添加一个续延参数，它的作用类似于异常处理的 finally 子句，并以有序的方式结束协议。(3) 我们可以修改 exec，使其每次只执行一步，并返回一个延续和中间状态，或返回类型为 *A*的最终结果。在包含循环的场景中，这种修改很直接：类型为 CmdStore (suc *n*) *A* 的值 cms 可以充当延续。要重启延续，只需对 cms zero 调用 exec。为明确起见，我们给出续延类型和重启函数的实现。

```
CmdCont : Set →Set CmdCont A = ∃[ n ] (CmdStore (suc n) A × A) restart :
CmdCont A →Channel → IO (CmdCont A ⊎ A) restart 〈 n , 〈 cms , st 〉 〉 ch
with cms zero ... | 〈 s₀ ,cmd₀ 〉 rewrite toℕ-fromℕ n = exec cmd₀ (pop1 cms) st
ch
```

采用这种方法时，重启的管理必须纳入可信计算基，因为它还必须保证对通道进行线性处理。

当前的形式化对这个问题尚未给出答案，只是破坏了协议。我们首选的解决方案是用 Haskell 实现，它省略了燃料，因此遵循协议。u

4.3 客户端呢?

到目前为止，我们介绍的递归会话命令适用于反复执行相同操作的服务器。然而，客户端可能希望在每次迭代中执行不同的操作。虽然这种行为可以通过应用状态来编码，但对程序员来说并不是一种愉快的体验。

因此，我们提出一种 展开 命令，使我们能够一次为一个循环迭代指定命令。

```
展开 : 命令 (后继 n)A S → 命令 n A (μ S) → 命令 n A (μ S)
```

它为会话类型的循环体指定一个命令，并为整个类型指定一个延续命令，以覆盖后续迭代。执行此命令意味着执行循环体，并将延续命令压入栈中。

```
exec g (展开 循环体命令 下一条命令) 命令流 状态 通道 = exec g 循环体命令 (将 命令流 下一条命令压入栈中) 状态 通道
```

例如，我们为 多重一元谓词 协议编写一个客户端，使其在终止前迭代该协议两次。每一轮中，它都会发送一个整数并忽略响应。

```
runningsum-client : 命令 0 ⊤(对偶 多重一元谓词)
runningsum-client =
  展开 (SELECT zero $ SEND (λ x → ⟨ tt , + 17 ⟩ ) $ RECV constᵣ (CONTINUE
  zero)) $ 展开 (SELECT zero $ SEND (λ x → ⟨ tt , + 4 ⟩ ) $ RECV constᵣ
  (CONTINUE zero)) $ LOOP (SELECT (suc zero) CLOSE)
```

5 转向单子式

我们此前已指出，回调以及 exec 函数的实现中都明显体现出单子式结构。事实上，改用单子式接口或许能让我们的会话程序更加简洁。

我们从一种略经重构的命令类型开始。更准确地说，我们引入一个新的 SKIP命令，它仅通过唯一的回调执行状态转换。接着，我们使用状态单子重新表述 SEND和 RECV 的回调类型。

```
data Cmd (A: Set) : 会话 → 集合₂ where CLOSE : Cmd Aend SKIP : (单子
M ⇒ StateT A M ⊤) → Cmd A S → Cmd A S SEND : (单子 M ⇒ StateT A
M T⟦ T ⟧) → Cmd A S → Cmd A (send T S) RECV : (单子 M ⇒ (T⟦T ⟧ →
StateT A M ⊤)) → Cmd A S → Cmd A (recv T S) SELECT : ∀ {Si} → (i : 有
限类型 k) → Cmd A (Si i) → Cmd A (⊕′ Si) CHOICE : ∀{Si} → ((i : 有限类
型 k) → Cmd A (Si i)) → Cmd A (&′ Si)
```

下面具体介绍回调的类型。每个回调都在一个状态单子中运行，该状态单子负责处理应用状态 A，并构建在另一个单子之上 M。这一构造通过单子变换器StateT 应用于应用状态 A 和底层单子 M来表达。回调的类型所透露的信息也就仅此而已。

不过请注意，使用 Agda 标准库时，我们必须说明 M 具有符合单子的类型，并且它实现了接口RawMonad （一种包含单子基本操作的记录）。幸运的是，我们可以抽象掉这些问题，采用受 Haskell 启发的语法，并用直接明了的 Agda 定义来实现。[8]

```
单子的 : ((集合 → 集合₁) →集合₁) →集合₂单子的 f = ∀
{M : 集合 → 集合₁} → {{RawMonad M}} → f M
语法 单子的 (λ M → X) = 单子 M ⇒ X
```

我们的运行示例变得（甚至更简洁了吗？）：

```
addp-command: 命令 ℤ binarypaddp-command = RECV put $ RECV
(modify ∘′ _+_)$ SEND get$ CLOSE
negp-command: 命令 ℤ unarypnegp-command =
RECV（put ∘ -_）$SEND get $ CLOSE
算术命令 : 命令 ℤ 算术 算术命令
= CHOICE λ 其中零 → 加法命令
（后继 零） → 否定命令
```

这里，放置、修改和获取 都是在状态单子变换器中定义的单子函数。它们的完整类型相当难以阅读，但直观上我们可以这样理解：

$$\text{put} : A \to ST\ A\ \top \qquad \text{modify} : (A \to A) \to ST\ A\ A \qquad \text{get} : ST\ A\ A$$

放置 使用其参数覆盖状态；修改 h 将函数 h 应用于状态并返回新状态；而获取 = 修改 id返回当前状态。

既然如此，我们也以单子风格定义 exec函数。此时，执行单子的完整类型显现出来了。我们已经遇到过用于处理应用状态的状态变换器 ontop。下面是一个读取器单子（同样使用单子变换器实现），用于处理对通道的访问。最底层则和之前一样，是输入输出单子。

```
exec: 命令 A S → 状态变换器 A（读取器单子 通道 输入输出） ⊤执行
CLOSE = 询问 »= 提升输入输出 ∘ 原语关闭 exec（SKIP act cmd） = act » 执行
cmd exec （ SEND getx cmd ） = getx »= λ x → 询问 »= 提升输入输出 ∘ 原
语发送 x » 执行 cmd exec（RECV putx cmd） = 询问 »= 提升输入输出 ∘ 原语
接收 »= putx » 执行 cmd exec（SELECT i cmd） = 询问 »= 提升输入输出 ∘
原语发送 i » 执行 cmd exec（CHOICE f-cmd） = 询问 »= 提升输入输出 ∘ 原
语接收 »= λ x → 执行（f-cmd x）
```

这项实现最引人注目之处，恰恰在于它并不引人注目。不过请记住，单子早已被用于封装线性资源，其中最著名的例子莫过于单子式 IO 的出现 [28]。也就是说，在这种单子式设定下，我们关于通道线性处理的证明义务变得更加简单。

我们以定义单子接受器结束本节。它本质上会调用 reader 单子和状态单子的运行器，并执行底层的 IO 动作。

[8]Agda 的 语法 命令定义了一个宏，使我们能够对绑定子进行抽象。新引入的语法，即被定义项，位于等号的右侧。将记录 RawMonad M 放在双大括号中，即可启用单子运算符的重载 [13]。

record Accepting A s : 集合$_2$ where constructor
ACC field pgm : 命令 A s acceptor : Accepting A
s → A → IO A acceptor (ACC pgm) a = do ch ←
primAccept 〈 final , _ 〉 ← runReaderT
(runStateT (exec pgm) a)ch pure final

6 上下文无关会话类型

上下文无关会话类型 [1, 44, 54] 旨在摆脱会话类型对尾递归的限制。消除这一限制能让会话类型化编程更具组合性，并支持树结构序列化等底层编程任务。

T其基本思路 [54] 是按如下方式重新组织会话类型的类型语言。

$$S ::= !T \mid ?T \mid S ⨟ S \mid \mathsf{skip}$$

现在$!T(?T)$仅表示发送（接收）一个类型为 T的值这一动作。要组合两个会话类型，我们必须使用带有单位元 $S⨟S$ 的顺序组合skip。分支类型和递归与之前相同，因此这里不再赘述。

这种结构的 Agda 编码可以直接与前几节积累的成果结合起来。

数据会话(n: ℕ) : 集合其中
?_ : 类型 →会话 n !_ : 类型 → 会话 n⊕′ :
(si : (i : 有限类型 k) → 会话 n) → 会话
n &′ : (si : (i : 有限类型 k) → 会话 n) →
会话 n_⨟_ : 会话 n → 会话 n → 会话 n
跳过 : 会话 nμ_ : 会话 (suc n) → 会话
n ‘ : Finn → 会话 n_

修订后的命令结构有一些需要解释的注意事项。

变量$V\ W$: Vec Set$n$

data 命令 : 集合 → 集合 → 向量 集合 n → 向量 集合 n → 会话 n → 集合
$_1$ 其中 SKIP : (A → B) → 命令 A B V Wskip SEND : (A → B × T⟦T ⟧) →
命令 A B V W (! T) RECV : (T⟦ T ⟧ → A → B) → 命令 A B V W (? T)
SELECT : ∀{Si} {F : 有限类型 k → 集合} → (A → Σ (有限类型 k) F)→ ((i :
有限类型 k) → 命令 (F i) B V W (Si i))→ 命令 A B V W(⊕′ Si) CHOICE : ∀
{Si} → ((i : 有限类型 k) → 命令 A B V W (Si i)) → 命令 A B V W (&′ Si)

$$[_]_⨟[_]_[_] : (A \to A_1 \times A') \to \mathsf{Cmd}\ A_1\ B_1\ V\ W\ S_1 \to (A' \to B_1 \to A_2) \to \mathsf{Cmd}\ A_2\ B_2\ V\ W\ S_2 \to (B_1 \to B_2 \to B) \to \mathsf{Cmd}\ A\ B\ V\ W\ (S_1 ⨟ S_2)$$

LOOP : 命令 A B (A :: V) (B :: W) S → 命令 A B V W(μ S) CONTINUE :
(i : 有限类型 n) → 命令 (查找 V i) (查找 W i) V W (‘ i)

该 Cmd 数据类型现在除了会话类型之外，还携带四个与类型相关的参数。类型为 Cmd $A\ B\ V\ W\ S$ 的命令首先是一种操作，它接收一个类型为 A的输入，产生一个类型为 B 的输出，并根据 S执行一个会话。额外的参数 V和W是类型向量，用于控制当前待处理循环的类型判断；这些循环将在 LOOP 和 CONTINUE命令中说明。目前，我们可以将它们视为这些待处理循环的输入类型和输出类型栈。

SKIP 命令对应于会话类型中的 跳过 。它附带一个函数，用于将 A转换为 B。

SEND 和 RECV命令不再接收命令参数来处理续延会话。这项功能现在由组合运算符一次性提供。

命令 SELECT 规定了动态选择。它接收一个回调，将输入类型 A 的值映射为由标签 i 和类型为 $F\ i$的值构成的依赖对。[9] 延续利用这一细粒度类型信息，将标签 i 与输入类型 $F\ i$ 关联起来，该输入类型由回调生成。

CHOICE 命令与之前相同，只是增加了类型参数。

两个命令的组合运算符还有三个额外的回调参数，我们称之为 split、cross 和 join。第一个回调 split 将输入类型拆分为两部分：一部分由第一个命令消耗，另一部分绕过该命令。第二个回调 cross 将输出类型与绕过的部分合并为第二个命令的输入类型。第三个回调 join则合并两个命令的输出。

在上下文无关的情况下，LOOP 命令的行为比第 4 节中的简单尾递归迭代更加通用。它接受一个循环体，将 A转换为 B，同时将这些相同的类型压入栈中。

CONTINUE i命令调用第 i 个待处理的循环。为此，当前类型必须与该循环的类型匹配；我们通过在栈上位置 i 查找输入类型和输出类型来确定该循环的类型。

6.1 示例

在深入探讨解释器之前，让我们通过回顾之前的示例以及一个展示上下文无关会话额外表达能力的新示例，来总结一下我们取得的成果。

– 二元函数的服务协议binaryp：会话
*n*binaryp = ?整数 ⨟ ?整数 ⨟ ! 整数

– 一元函数的服务协议unaryp： 会话
*n*unaryp = ? 整数 ⨟ ! 整数

– 用于在二元函数与一元函数之间进行选择的服务协议arithp： 会话 *n*arithp =
& [binaryp , unaryp]

– 多个一元函数

[9]该 F参数的负出现不允许出现在 集合 数据类型的构造器中。它的存在会迫使命令类型上移一级，进入集合$_1$。

```
多重一元谓词: 会话 n
多重一元谓词 = μ (& [ unaryp ⨟ ‘ zero , skip ])
```

将这些类型与第4节中相应的类型进行比较：在该节中，协议片段是以续延会话为参数的元函数。上下文无关会话类型无需这样的参数化。它们本质上具有组合性和模块化。

最后一个示例 多重一元谓词 包含一个尾递归部分。选择的第二个分支必须使用 SKIP 类型，因为选择分支不能为空。

让我们来考虑实现这些协议的服务器。

```
addp-command : 命令 ⊤ ⊤V W binarypaddp-command = RECV
const ⨟′ RECV _+_ ⨟′ SEND (λ x → 〈 tt ,x〉 )
negp-command : 命令 ⊤ ⊤V W unarypnegp-command =
RECV (const ∘′ -_) ⨟′ SEND λ x → 〈 tt , x〉
arithp-command : 命令 ⊤ ⊤V W
arithparithp-command = CHOICE λ
其中zero → addp-command(suc zero)
→ negp-command

many-unaryp-command : 命令 ℤ ℤ VW 多重一元谓词
many-unaryp-command = LOOP $ CHOICE λ 其中zero →
(RECV _+_ ⨟′ SEND < id , id >) ⨟′ CONTINUE zero(suczero) →
SKIP id
```

服务器的代码看起来与之前并没有太大不同。不过，与之前不同的是，现在每个服务器都可以作为更大协议实现的一部分重复使用。一种迹象是类型中使用了参数 V和W，另一种迹象则是缺少了CLOSE 这个用于规定关闭连接的构造器。

由于这些协议仍然是尾递归的，其实现采用了一种针对这一用例设计的命令二元组合运算符。分割函数将所有内容传递给第一个命令；交叉操作忽略被绕过的值，只将第一个命令的输出传递给第二个命令；合并操作则只传递第二个命令的输出。

```
_⨟′_ : 命令 A B V W S₁ →命令 B C V W S₂ →命令 A C V W (S₁ ⨟ S₂)命
令₁ ⨟′ 命令₂ = [ (λ x → 〈 x , tt 〉 ) ] 命令₁ ⨟[ 常量ᵣ ] 命令₂ [ 常量ᵣ ]
```

为了进一步明确上下文无关会话与正则会话之间的区别，我们最后介绍一种协议，它通过超越尾递归，发挥上下文无关会话类型的强大能力：接收并发送二叉树的服务器。

```
数据 IntTree : 集合 其中叶节点 : ℤ →
IntTree分支 : IntTree →IntTree →
IntTree

leafp : 会话 n
leafp = ? 整数
branchp: 会话(suc n)
branchp = ‘零 ⨟‘ 零
```

```
treep: 会话n
treep = μ & [ leafp , branchp ]
```

会话类型 leafp 和 branchp 对二叉树类型 IntTree 的叶节点和分支进行接收编码。会话类型 treep 提供外围递归，以及在叶节点协议和分支协议之间进行选择。由于 branchp 包含两个递归调用，协议 treep 不再是尾递归。

```
recvTree : 命令 ⊤ IntTree V W treep recvTree = LOOP $ CHOICE λ where zero
 →RECV (const ∘′叶节点) (suc zero) → [ (λ x → 〈 x , x 〉 ) ] (CONTINUE zero)
 ⨟[ const ] (CONTINUE zero) [ 分支 ]
```

叶节点（选择 zero）的接收方会将收到的值包装在 叶节点构造器中并返回。分支（选择 suc zero）的接收方通过递归调用获取左子树和右子树CONTINUE zero , 并使用分支构造器组合结果。 Split 和cross回调只会操作单位类型的值。

树的发送方充分利用了用于命令组合的回调。

```
IntTreeF: 有限类型 2 → 集合
IntTreeF零 = ℤIntTreeF (后继 零) =
IntTree × IntTree
splitTree : IntTree → Σ (有限类型2)
IntTreeFsplitTree (叶节点 x) = 〈 零 , x 〉
splitTree (分支 t₁ t₂) = 〈 后继 零 , 〈 t₁ , t₂ 〉 〉
sendTree : 命令 IntTree ⊤V W (dual treep) sendTree = LOOP $
SELECT splitTree λ 其中 zero → SEND (λ z → 〈 tt , z 〉 ) (suc zero)
→ [ id ] (CONTINUE zero) ⨟[ const ] (CONTINUE zero) [const ]
```

函数 splitTree 本质上实现了展开类型 IntTree中递归所对应同构的一半；函数IntTreeF 有助于构造其类型。利用这些辅助函数，可以直接构造树的发送方。将 splitTree 用作 SELECT 的回调后，我们发现，零 分支会接收一个整数，而 后继 零 分支会接收一对子树。Split 回调就是恒等映射，因此左子树会传递给第一次递归调用，右子树则传递给第二次。

6.2 解释

我们展示完整的 exec 函数，并在最后说明本节的回调接口为何不是单子的。

```
exec: Cmd{n} A B V W S →CmdStore n V W →A →(ReaderT
Channel 输入输出) B exec (SKIP act)cms a = 纯 (act a) exec (SEND
getx)cms a = 执行 let 〈 b , x 〉 = getx a ask »= liftIO ∘ 原语发送 x
纯 b exec (RECV putx)cms a = 执行 x ←ask »= liftIO ∘ 原语接收
```

纯 (putx x a) exec (SELECT getx cont) cms a with getx a ... | 〈 i , ai 〉 = 执行 ask »= liftIO ◦ 原语发送 i exec (cont i) cms ai exec (CHOICE cont) cms a = 执行 i ← ask »= liftIO ◦ 原语接收 exec (cont i) cms a exec [Split] 命令$_1$ ⨾[cross] 命令$_2$ [join] cms a = 执行 let 〈 a_1 , a′ 〉 = Split a b_1 ←exec 命令$_1$ cms $a_1$let a_2 = cross a′ b_1b_2 ←exec 命令$_2$ cms a_2纯 (join b_1 b_2) exec (LOOP 命令) cms a = exec 命令 (push cms 命令) a exec{suc n} (CONTINUE i) cms a with cms i ... | 〈 s-i , 命令-i 〉 = exec 命令-i (pop cms i) a

的实现SKIP 只执行该操作。发送SEND、接收RECV和选择CHOICE 与通常相同。相较于第3节，动态选择得到了改进。此前，被选中的标签与所执行的延续之间没有连接，因此解释器中的错误可能会导致不匹配。在当前的解释器中，不可能出现这种不匹配，因为调用除cont i（其类型为 ai）之外的任何其他延续都会产生类型错误。

组合首先使用split 回调拆分输入。它执行左侧命令，并在 b_1中获取其最终状态，然后使用cross将其与旁路值 a' 组合，接着执行右侧命令。它在 b_2 中获取其最终状态，并返回join a_1 a_2。关于LOOP 和CONTINUE 的实现与第4节之前的实现相同。区别在于，CONTINUE 操作现在可能出现在组合的上下文中，而该组合会提供一个非平凡的延续。在第4节中，exec 函数是尾递归的，但在这里不是！

现在我们来讨论单子式回调接口的问题。检查代码后可以发现，状态单子已不再是描述回调的最恰当模型。乍看之下，类型 $A \to M\ B$ 与 ReaderT $A\ M\ B$相同。然而，组合的情况表明，读取器并不适用，因为在解释器的递归调用之间，读取器源的类型 A 会发生变化。此外，组合的回调完全不符合读取器单子的模式。因此，我们不采用单子式接口。

7 处理多个通道

要完整涵盖传统的二元会话类型，我们还需补充一些要素：线程可以同时打开并操作多个通道，而且通道可以被委托。到目前为止，我们的接口仅限于单个通道，且只能传输纯数据（即不包含通道）。现在，我们来解除这些限制。

为了聚焦于新的问题，我们先对第 3 节中介绍过的会话类型稍作改写。具体而言，我们仅考虑二元分支，并将其扩展为有限分支以及加入递归的工作留作读者练习。与之前一样，这些类型描述的是单个通道上的通信。这里有两个新方面：我们

将通信方向的规范拆分出来，并新增一种用于信道委托的特殊类型，即发送或接收一个通道。

```
数据方向：集合其中
  输入 输出：方向

数据会话：集合其中
  传输：(d：  方向) → 类型 → 会话  → 会话委托：(d:
  方向) → 会话 → 会话 → 会话分支：(d：  方向)
  → 会话 → 会话 → 会话结束：  会话
```

一个实际的多信道会话类型描述了所有通道上交错进行的通信。它具有与多方会话类型非常相似的特性和限制 [20]。

```
数据MSession：  ℕ →集合变
量M M₁ M₂：  MSession n
因果性：  有限类型 n → MSessionn → MSession n →
SetCheckDual0：  MSession (suc m) → MSession (suc n) → Set

数据 MSession 其中
  传输 : (d : Direction) → (c : Fin n) → (T : Type) → MSession n →
  MSession n 分支 : (d : Direction) → (c : Fin n) → (M₁ : MSession n) → (M₂ :
  MSession n)→ 因果性 c M₁ M₂ → MSessionn 关闭 : (c : Fin (suc n)) →
  MSessionn → MSession (suc n) 终止 : MSession zero 连接 : Split m n →
  (M₁ : MSession (suc m)) → (M₂ : MSession (suc n))→ CheckDual0 M₁ M₂ →
  MSession (m + n) – 假设新通道在两个线程中的地址均为零 委托输出 : (c j :
  Fin (suc n)) →c ≢ j →Session →MSession n → MSession (suc n) 委托输入 :
  (c : Fin n) → MSession (suc n) → MSession n – 接收的通道在延续中的地址
  为零
```

多通道会话类型 MSession n，以 M表示，按其所管理的通道数 n 进行索引。多通道会话类型大致基于 Wadler 的 GV 演算[61]。具体而言，多通道程序的连接拓扑受到与 Wadler 的 GV 相同的限制，因此可以保证不会发生死锁，这一点令人欣喜。

通道的名称由 de Bruijn 指标表示。与其他嵌入式实现不同，多通道类型涵盖了通信应用的完整编排：连接（分叉）到新进程与通道创建、分支以及像往常一样传输值相结合，还包括关闭通道、委托，以及在所有协议完成且各自的通道关闭后终止应用。

除 connect 和 terminate之外，每种多通道类型都接受一个参数 c，用于标识命令所操作的通道。用于传输值的类型（transmit）与之前相同，只是方向和通道参数有所不同。分支（branch）也与之前相同，但增加了 因果性函数：它通过确保除 c 之外的每个通道上的会话不受分支影响，来保证分支是合理的。在多方会话类型的语境中，这项限制称为“因果性”。之所以需要这项限制，是因为只有通道 c 另一端的参与方知道发生了分支，而其他参与方并不知道。

由于编排过程中可以关闭某个通道，而不终止多通道协议，因此 close 类型需要一个续延参数。当所有通道都关闭后，可以使用类型 terminate来结束编排。

类型 连接 表示创建一个运行协议的新线程 M_1，而当前线程则继续执行 M_2。当前打开的通道根据 Split 参数分配给各个线程。通过新建一个通道来建立与新线程的通信，该通道在两个线程中的地址都映射为 零 。此外，CheckDual0 谓词确保该通道两端的会话类型互为对偶。[10]

委托类型用于通过一个通道发送（接收）另一个通道。它们的类型有足够大的差异，因此无法将方向抽象掉。要使用 j 委托输出发送通道，我们需要知道所传输的通道并非发送所用的同一通道、它的简单会话类型，以及延续命令少一个通道。要使用委托输入 接收通道时，只需在续延会话中将其映射到地址 零 即可。

函数 投影 类似于多方会话类型中从全局类型到局部类型的投影。我们使用操作 de Bruijn 指标的辅助函数： locate-split sp- c c 确定通道 c 的目标线程及其局部索引； adjust 会调整索引，以计入从编排中移除的通道。 因果性 和 CheckDual0 均通过显而易见的公式实现。

投影 : 有限类型 n → M会话 n → 会话 投影 c (连接 sp-c M_1 M_2 _)，并以 locate-split sp-c c ... | inj_1 x = 投影 (suc x) M_1... | inj_2 y = 投影 (suc y) M_2投影 c (分支 d x M_1 M_2 因果性)，并以 c ? = x ... | no c≠x = 投影 c M_1 – 我们有 (因果性 c c≠x : 投影 c M_1 ≡ 投影 c M_2) ... | yes refl = 分支 d (投影 c M_1) (投影 c M_2) 投影 c (传输 d x t M)，并以 c ? = x ... | no c≠x = 投影 c M ... | yes refl = 传输 d t (投影 c M) 投影 {suc n} c (close x M)，并以 c ? =x ... | no c≠x = 投影 (adjust c x c≠x) M ... | yes refl = end 投影 c (delegateOUT x j x≠j sj M)，并以 c ? =x ... | yes refl = 委托输出 sj (投影 (adjust c j x≠j) M) ... |no c≠x，并以 c ? = j ... | yes refl =sj ... | no c≠j = 投影 (adjust c j c≠j) M 投影 c (delegateIN x M)，并以 c ? = x ... | yes refl = 委托输入 P (投影 zero M) (投影 (suc c) M) ... |no c≠x = 投影 (suc c) M 因果性 {n} i M_1 M_2 = ∀(j : 有限类型 n) → j ≠ i → 投影 j M_1 ≡ 投影 j M_2对偶性检查 M_1 M_2 = 投影 zero M_1 ≡ 对偶 (投影 zero M_2)

[10]We 由于篇幅所限，省略了论文中关于Split类型以及对偶性的细节。请参阅补充材料。

根据前面的讨论以及前几节中的讨论，命令的类型可以直接得出。

```
data 命令 (A : 集合) : (n : ℕ) → MSession n → 集合₁ where CLOSE : ∀ c → (A → A) → 命令 A n M → 命令 A (suc n) (close c M) SEND : ∀ c → (A → T⟦ T ⟧ × A) → 命令 A n M → 命令 A n (send c T M) RECV : ∀ c → (T⟦ T ⟧ → A → A) → 命令 A n M → 命令 A n (recv c T M) SELECT : ∀ c → (causal : 因果性 c M₁ M₂) → (A → 布尔值 × A)→命令 A n M₁ → 命令 A n M₂ → 命令 A n (select c M₁ M₂ causal) CHOICE : ∀c → (causal : 因果性 c M₁ M₂)→命令 A n M₁ → 命令 A n M₂ → 命令 A n (choice c M₁ M₂ causal) CONNECT : ∀ {M₁ : MSession (suc m)} {M₂ : MSession (suc n)} (check : CheckDual0 M₁ M₂)→ (split : A → A × A)→(sp : Split m n)→ 命令 A (suc m) M₁ → 命令 A(suc n) M₂→ 命令 A (m + n) (connect sp M₁ M₂ check) SENDCH : ∀{sj} → ∀ c j → (c≠j : c ≠ j)→ 命令 A n M → 命令 A (suc n) (委托输出 c j c≠j sj M) RECVCH : ∀c → 命令 A (suc n) M → 命令 A n (委托输入 c M) END : 命令 A 0 终止 同理，解释器 exec 是一道简单的练习。exec : 命令 A n M →A →Vec 通道 n → 输入输出 A exec (SENDCH c j f≠j cmd) state chns = do 原语发送 (查找 chns c) (查找 chns j) exec cmd state (移除 chns j) exec (RECVCH c cmd) state chns = do ch ← 原语接收 (查找 chns c) exec cmd state (ch :: chns) exec (CONNECT _ split split-ch cmds₁ cmds₂) state chns =let 〈 state₁ , state₂ 〉 = s plit state in let 〈 chns₁ , chns₂ 〉 = 应用拆分 split-ch chns in 新建通道 »= λ{ 〈 ch₁ , ch₂ 〉 →分叉输入输出 (exec cmds₁ state₁ (ch₁ :: chns₁)) » exec cmds₂ state₂ (ch₂ :: chns₂) } exec (CLOSE c gend cmd) state chns = do 原语关闭 (查找 chns c) exec cmd (gend state) (移除 chns c) exec (SEND c getx cmds) state chns =let 〈 x , state′ 〉 = getx state in 原语发送 (查找 chns c) x » exec cmds state′ chns exec (RECV c putx cmds) state chns =原语接收 (查找 chns c) »= λ x →let state′ = putx x state in exec cmds state′ chns
```

exec (SELECT c _getx cmds$_1$ cmds$_2$) state
chns = do let 〈 b , a 〉 = getx state 原语发
送 (查找 chns c) b if bthen (exec cmds$_1$ a
chns) else (exec cmds$_2$ a chns) exec
(CHOICE c _cmd$_1$ cmd$_2$) state chns = do b ←
原语接收 (查找 chns c) if bthen exec cmd$_1$
state chns else exec cmd$_2$ state chns exec
END state [] = do pure state runCmd : 命令
A 0 M → A → 输入输出 A runCmd cmd i
nit =do exec cmd init []

8 讨论

8.1 使用向量进行选择和选择

读者可能会想，为什么我们不使用长度为 n的向量来定义选择和选择的构造器，而不是使用从有限类型 n → 会话 出发的函数（两者是同构的）。使用函数可以直接扩展到命令 类型的定义，其中延续命令的会话 索引取决于函数参数 i： 有限类型 n。若要实现类似的表达能力，我们就必须定义一种特殊的依赖向量类型。

8.2 替代表示

我们可以不解释命令值，而是效仿相关工作 [39, 64]，将其编译到一个自定义库中。通过针对单个命令对命令解释器进行特化，即可得到这样的编译器。

由此得到的指称式实现 对应于一种库实现：它将命令（在我们的方法中，这些命令被具体化为一种数据类型）作为函数公开。Reynolds [48] 率先提出了这种方法，之后该方法也应用于部分求值和程序分析等领域 [7, 53]。

H下面摘录了这种实现的一部分，它源自第 2 节中的解释器 .

XCmd： 集合 → 会话 → 集合$_1$
XCmd A s = A →通道 输入输出 A

xclose : XCmd A end xclose state ch = 执行 原语关
闭 ch，返回 state xsend : (A → A × T⟦T ⟧) →
XCmd A S → XCmd A (! T · S) xsend f xcmd = λ
state ch → 执行 let〈 state′ , x 〉 = f state 原语发送
x ch xcmd state′ ch

其他章节中的解释器也可以用类似的方式重新表述，而不会牺牲我们方法的优势。不过，在程序开发中，基于命令的方式更具吸引力，因为 Agda 的交互式编程支持为命令提供了自动补全功能，但不支持组合子。

8.3 多方会话类型

我们认为，将本文提出的方法扩展到参与方超过两方的协议并无问题。为避免额外的复杂性，本文没有进行这种扩展。第 7节已对完整的多方版本所需的条件提供了一些见解。

8.4 验证

我们的方法有个美中不足之处：它需要证明解释器不会破坏线性性。由于解释器非常简单，人们很容易想依靠人工检查。

为了获得形式化且有望实现机械化的证明，我们设想采用自由单子 [30, 51] 来定义语义，其中包含对原始通道 IO API 操作的未解释出现。实际证明或许可以借助交互树 [63], 来进行；这是一种用于表示递归和非纯程序并对其推理的机械化框架。

9 相关工作

在本节中，我们回顾不同的会话类型库实现如何处理线性性。具体而言，我们不考虑专用的语言实现，例如 SILL [56],SEPI [14],FreeST [1], 或 Links [36]。这些实现配备了专用的类型检查器，能够在编译时恰当地处理线性性。

有些库对会话类型规约采用完全动态的强制执行方式 [21, 37]。由于它们必须检查每个通信操作，因此会产生运行时开销；此外，它们缺乏保证，因为一旦在某个对等方检测到错误，就会终止协议。

有些库在运行时执行线性性检查。虽然对资源至多一次（仿射）使用的检查只需一个比特即可完成 [57], ，但线性性检查的开销更高，不过仍被认为是轻量级的 [22]。

有些库通过使用参数化单子 [45, 49],、多态 [26], 或高阶抽象语法 [35]对其进行编码，从而在静态上强制执行线性性。尽管这些编码巧妙地利用了宿主语言提供的功能并支持类型推断，但它们难以解释，而且生成的类型对程序员而言并不易读。在我们的方法中不需要这种巧妙的技巧；类型具有可读性，交互式 Agda 系统有助于构造类型和程序，但不会推断会话类型。

现有的嵌入方案都不具备类似我们动态选择的功能。不过，动态选择可以视为标签依赖的会话类型的一种特殊情况 [55], ，因此我们的方法也实现了该理论的部分内容。

虽然动态库和面向对象的库支持会话子类型化，但我们的方法目前不支持子类型化。

最相关的工作由 Miu、Ferreira、Yoshida 和 Zhou 完成 [39]。他们通过根据协议规范生成自定义库，在 TypeScript 中实现了多方会话类型。他们的实现保证不会出现通信错误，包括死锁、通信不匹配、通道使用违规或取消错误。他们生成了回调风格的 TypeScript API，其中通信端点背后的有限状态机通过接口来具体表示。发送和接收操作均通过回调编码，视具体情况交由库或用户程序处理。生成的 API 封装了所有基本通信操作。

9.1 Haskell

Neubauer 最早的 Haskell 实现 [40] 着重于使用幻影类型对单个会话的类型进行建模。对偶性通过带有函数依赖的类型类来实现。该实现未考虑线性性。

Sackman [49] 和 Tov [58] 使用参数化单子对多个通道进行建模，该单子由从通道名称到其当前会话类型的映射进行索引。每次执行操作时，映射都会发生变化，以跟踪所有通道的当前状态。单子式接口保证了线性性。

这种方法比我们在第 7 节提出的方法赋予程序员更大的自由度。虽然在他们的方法中，各个通道类型彼此独立，但我们的多通道会话类型会对所有连接施加一种编排，这在理念上更接近多方会话类型。

Lindley 和 Morris [35] 以 GV [17]的风格，为线性 λ 演算的高阶抽象语法编码扩展了单子和会话原语。他们将这种方法称为“参数化无标签” [7]，这意味着他们将 λ 演算的语法和会话原语编码为类型类中参数化函数的形式，并随后提供具体实现。

Orchard 和 Yoshida [42] 讨论了会话类型与效应系统之间的联系。他们通过基于分级单子的效应系统编码，在 Haskell 中实现了会话类型。在这里，“分级”跟踪当前处于活动状态的通道及其会话类型，而单子结构则提供适当的顺序执行。

9.2 OCaml

FuSe 的实现 [43] 由构建在无类型通道之上的类型化层组成（与我们的方法一样）。它提供了一个受 GV 启发的应用程序接口，支持类型推断，并在运行时检查线性性。该方法可以扩展到上下文无关会话类型 [44] ，但需要一些用户注解（同时保留类型推断）。

Session OCaml [26] 通过基于 Garrigue[11] 技术的参数多态来强制实施线性性，并进行了大幅扩展以处理会话类型。它可以同时处理固定数量的通道，并使用全局定义的通道名称（槽名称）。该技术依赖于全局槽名称的多态类型。会话类型可从使用类似 FuSe 和 GV 的表示法编写的程序中推断出来。

9.3 Java

“基于会话的 Java 分布式编程” [23] 中提出的早期接口依赖于用于会话通信的特殊语法和预处理器，以限制别名。Mungo [32] 和 Bica [16] 采用类似的思想来实现类型状态和会话。

Hu 和 Yoshida [22]率先提出了用于实现多方会话类型的代码生成方法。他们根据多方会话类型为 Java 生成协议专用端点 API，但声称其方法可广泛适用于主流语言。他们从一个观察出发：通信端点的行为可以用有限状态机表示。每个状态都被具体化为一种状态通道类型，其中的方法隐含了与状态转换相对应的通信操作。由于 Java 不限制对象的使用方式，他们在生成的 API 中部署了“非常轻量的运行时检查，以确保通道类型实例遵循线性使用规范”。他们的抽象 I/O 状态接口与我们提供的功能最为接近。

尽管他们的方法将通信端点可能处于的不同状态具体化了，但在我们的方法中，这些状态是隐含的。此外，基于有限状态机的应用程序接口完全

[11]参见 https://github.com/garrigue/safeio。

适用于服务器；我们还可以通过 展开第 4.3 节中所述的命令，为递归协议的客户端实现提供更大的灵活性。

Scala 的工作在 Scala [50] 中遵循类似的思路，并通过运行时检查来保证线性性。较新的工作 [9] 通过依赖 Scala 3 中的高级类型特性，提高了灵活性。

9.4 Rust

最近有若干会话类型的实现 [8, 11, 12, 27, 31, 33, 34]依赖于 Rust；Rust 是一种具有唯一性类型和所有权的语言，二者均在编译时进行检查。尽管唯一性类型与仿射类型非常相似（描述至多只能使用一次的值），但它们尚未达到会话所需的程度：通道具有仿射会话类型意味着代理可以在未完成协议的情况下随时丢弃连接，这会导致连接另一端发生死锁。借助适当的线性会话类型，每个代理都必须履行协议，直至连接关闭。

10 结论

我们的论文表明，使用基于回调的方法来实现会话类型化程序，与（轻度）依赖类型函数式编程完美契合。这种内在会话类型化的程序编写方式，通过构造保证了协议保真性，并且在某些情况下保证无死锁性。结合回调方法，宿主语言无需支持线性性；因此，一旦封装库经过验证，程序便具有静态安全性。

有趣的是，改变研究视角揭示了两种新构造的可能性及其必要性：动态选择，以及用于递归会话的展开 命令。前者得益于我们的依赖类型宿主语言。后者则源于编写递归协议客户端程序的需要。

第8 节的讨论提出了若干未来研究方向。此外，探讨如何整合子类型化也很有意义（一种显而易见的做法是坚持使用显式强制转换）。

参考文献

[1] Bernardo Almeida、Andreia Mordido、Peter Thiemann 和 Vasco T. Vasconcelos。2022年。带有上下文无关会话类型的多态 λ 演算。Inf. Comput. 289, 2022 104948 10 1016 2022 104948 Part ( ), 。https://doi.org/ 。/ j.ic. 。[2] Davide Ancona、Viviana Bono、Mario Bravetti、Joana Campos、Giuseppe Castagna、Pierre-Malo Deniélou、Simon J. Gay、Nils Gesbert、Elena Giachino、Raymond Hu、Einar Broch Johnsen、Francisco Martins、Viviana Mascardi、Fabrizio Montesi、Rumyana Neykova、Nicholas Ng、Luca Padovani、Vasco T. Vasconcelos 和 Nobuko Yoshida。2016年。编程语言中的行为类型。Found. Trends Program. Lang. 3, 2 3 2016 95 230 10 1561 2500000031 - ( ), – 。https://doi.org/ . / Stephanie Balzer 和 Frank Pfenning。 。使用会话类型实现显式共享。PACMPL ICFP ( ), : – : 。https://doi.org/ 10 1145 3110281 . / Massimo Bartoletti、Ilaria Castellani、Pierre-Malo Deniélou、Mariangiola Dezani-Ciancaglini、Silvia Ghilezan、Jovanka Pantovic、Jorge A. Pérez、Peter Thiemann、Bernardo Toninho 和 Hugo Torres Vieira。2015年。将行为类型与安全分析相结合。J. Log. Algebraic Methods Program. 84, 6 (2015), 763–780。https://doi.org/10.1016/j.jlamp. 2015.09.003[5] Luís Caires 和 Frank Pfenning。2010年。作为直觉主义线性命题的会话类型。载于 CONCUR（LNCS，第6269卷）。Springer，法国巴黎，222 236 10 1007 978 3 642 15375 4_16 – 。https://doi.org/ . / - - - - Luís Caires、Frank Pfenning 和 Bernardo Toninho。 。作为会话类型的线性逻辑命题。MSCS 26, 3 (2016 367 423 10 1017 0960129514000218), – 。https://doi.org/ . /S Jacques Carette、Oleg Kiselyov 和 Chung-chieh Shan。 。终于实现无标签、部分求值：用于更简单类型化语言的无标签分阶段解释器。函数式编程杂志 19, 5 2009 509 543 10 1017 0956796809007205 ( ), – 。https://doi.org/ . /S Ruofei Chen、Stephanie Balzer 和 Bernardo Toninho。 。Ferrite：会话类型在 Rust 中的判断式嵌入。载于 ECOOP 2022（LIPIcs，第222卷），Karim Ali 和 Jan Vitek（编）。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，德国柏林，22 1 22 28 10 4230 2022 22 : – : 。https://doi.org/ . /LIPIcs.ECOOP. 。Guillermina Cledou、Luc Edixhoven、Sung-Shik Jongmans 和 José Proença。2022年。多方会话类型的 API 生成：重新审视并使用 Scala 3 修订。载于 ECOOP 2022（LIPIcs，第222卷），Karim Ali 和 Jan Vitek（编）。

Schloss Dagstuhl - Leibniz-Zentrum für Informatik，德国柏林，27:1–27:28。https://doi.org/10.4230/LIPIcs. ECOOP.2022.27[10] Jesper Cockx、Orestis Melkonian、Lucas Escot、James Chapman 和 Ulf Norell。2022年。合理的 Agda 是正确的 Haskell：使用 agda2hs 编写经验证的 Haskell 程序。载于 Haskell '22，Nadia Polikarpova（编）。ACM，斯洛文尼亚卢布尔雅那，
108 122 10 1145 3546189 3549920 – 。https://doi.org/ . / 。Zak Cutner 和 Nobuko Yoshida。 。Rust 中安全的基于会话的[11] 异步协调。载于 COORDINATION 2021（《计算机科学讲义》，第12717卷），Ferruccio Damiani 和 Ornela Dardha（编）。Springer，马耳他瓦莱塔，
80 89 10 1007 978 3 030 78142 2_5 – 。https://doi.org/ . / - - - - Zak Cutner、Nobuko Yoshida 和 Martin Vassor。[12] 。使用多方会话类型在 Rust 中实现无死锁异步消息重排序。载于 PPoPP '22，Jaejin Lee、Kunal Agrawal 和 Michael F. Spear（编）。ACM，大韩民国首尔，
246 261 10 1145 3503221 3508404 – 。https://doi.org/ . / 。Dominique Devriese 和 Frank Piessens。[13] 。类型类的积极一面：Agda 中的实例参数。载于 ICFP 2011，Manuel M. T. Chakravarty、Zhenjiang Hu 和 Olivier Danvy（编）。ACM，日本东京，143–155。https://doi.org/ 10 1145 2034773 2034796 . / 。Juliana Franco 和 Vasco Thudichum Vasconcelos。2013年。一种具有精化会话类型的并发[14]编程语言。载于《软件工程与形式化方法——SEFM 2013 联合研讨会：BEAT2、WS-FMDS、FM-RAIL-Bok、MoKMaSD 和 OpenCert，修订精选论文》（《计算机科学讲义》，第8368卷），Steve Counsell 和 Manuel Núñez（编）。Springer，西班牙马德里，
15 28 10 1007 978 3 319 05032 4_2 – 。https://doi.org/ . / - - - - Simon Gay 和[15] António Ravara（编）。 。行为类型：从理论到工具。River Publishers，丹麦 Gistrup。[16] Simon J. Gay、Nils Gesbert、António Ravara 和 Vasco Thudichum Vasconcelos。2015年。面向对象的模块化会话类型。Log. Methods Comput. Sci.
11, 4 2015 10 2168 11 4 12 2015 ( )。https://doi.org/ . /LMCS- (:) Simon J. Gay 和[17] Vasco Thudichum Vasconcelos。 。异步会话类型的线性类型理论。JFP 20, 1 2010 19 50 10 1017 ( )， – 。https://doi.org/ . /S 0956796809990165[18] Kohei Honda。 。二元交互的类型。载于 CONCUR '93（LNCS，第715卷）。
Springer，德国希尔德斯海姆，
509 523 10 1007 3 540 57208 2_35 – 。https://doi.org/ . / - - - Kohei Honda、Vasco Thudichum Vasconcelos 和 Makoto Kubo。1998年。用于基于结构化通信的编程的语言原语与类型规约。载于 ESOP '98（LNCS，第[19] 1381卷）。Springer，葡萄牙里斯本，122–138。https://doi.org/ 10 1007 0053567 . /BFb Kohei Honda、Nobuko Yoshida 和 Marco Carbone。2016年。多方异步会话类型。J. ACM 63, 1 (2016 9 1 9 67 10 1145 2827695)，: – : 。[20] https://doi.org/ . / Raymond Hu、Rumyana Neykova、Nobuko Yoshida、Romain Demangeon 和 Kohei Honda。2013年。实用的可中断对话——使用会话类型和 Python 进行分布式动态验证。载于 Runtime Verification 2013（《计算机科学讲义》，第8174卷），Axel Legay 和 Saddek Bensalem（编）。Springer，法国雷恩，130–148。https://doi.org/ 10 1007 978 3 642 40787 1_8 . / - - - - Raymond Hu 和 Nobuko Yoshida。[21] 。通过端点 API 生成实现混合会话验证。载于 FASE 2016（《计算机科学讲义》，第9633卷），Perdita Stevens 和 Andrzej Wasowski（编）。Springer，荷兰埃因霍温，
401 418 10 1007 978 3 662 49665 7_24 – 。https://doi.org/ . / - - - - Raymond Hu、Nobuko Yoshida 和[22] Kohei Honda。 。基于会话的分布式 Java 编程。载于 ECOOP 2008（《计算机科学讲义》，第5142卷），Jan Vitek（编）。Springer，塞浦路斯帕福斯，516–541。https: //doi.org/ 10 1007 978 3 540 70592 5_22 . / - - - - Hans Hüttel、Ivan Lanese、Vasco T. Vasconcelos、Luís Caires、Marco Carbone、Pierre-Malo Deniélou、Dimitris Mostrous、Luca Padovani、António Ravara、Emilio Tuosto、Hugo Torres Vieira 和 Gianluigi Zavattaro。2016年。会话类型与行为契约基础。ACM Comput. Surv.
49, 1 2016 3 1 3 36 10 1145 2873052 ( )，: – : 。https://doi.org/ . /[24] Keigo Imai 和 Jacques Garrigue。[25] 。使用透镜和单子的轻量级线性类型编程。J. Inf. Process.
27 2019 431 444 10 2197 27 431 ( )， – 。https://doi.org/ . /ipsjjip. 。Keigo Imai、Nobuko Yoshida 和[26] Shoji Yuen。 。Session-OCaml：具有极性和透镜的基于会话的库。Sci. Comput. Program.
172 2019 135 159 10 1016 2018 08 005 ( )， – 。https://doi.org/ . /j.scico. . . Thomas Bracht[27] Laumann Jespersen、Philip Munksgaard 和 Ken Friis Larsen。2015年。Rust 的会话类型。载于 WGP@ICFP 2015，Patrick Bahr 和 Sebastian Erdweg（编）。ACM，加拿大不列颠哥伦比亚省温哥华，13–22。https://doi.org/10. 1145/2808098.2808100[28] Simon L. Peyton Jones 和 Philip Wadler。1993年。命令式函数式编程。载于 POPL 1993，Mary S. Van Deusen 和 Bernard Lang（编）。ACM 出版社，南卡罗来纳州查尔斯顿，
71 84 10 1145 158511 158524 – 。https://doi.org/ . / 。Shunsuke Kimura 和 Keigo Imai。[29] 。C# 中的流式会话编程。载于 PLACES@ETAPS（EPTCS，第 卷），Stephanie Balzer 和 Luca Padovani（编）。爱尔兰都柏林，
61 75 10 4204 314 6 – 。https://doi.org/ . /EPTCS. 。Oleg Kiselyov 和 Hiromi Ishii。[30] 。Freer 单子：更具可扩展性的效应。载于 Haskell，Ben Lippmeier（编）。ACM，加拿大不列颠哥伦比亚省温哥华，94–105。https://doi.org/10.1145/2804302.2804319

[31] Wen Kokke。2019。《Rusty Variation：Rust 中具有故障处理能力的无死锁会话》。载于 ICE 2019（EPTCS，第 304 卷），Massimo Bartoletti、Ludovic Henrio、Anastasia Mavridou 和 Alceste Scalas（编）。丹麦哥本哈根，48–60。https://doi.org/10.4204 304.4 . /EPTCS. . Dimitrios Kouzapas、Ornela Dardha、Roly Perera 和 Simon J. Gay。2018。《使用[32] Mungo 和 StMungo 对协议进行类型检查：面向 Java 的会话类型工具链》。Sci. Comput. Program. 155 (2018)，52–75。https://doi.org/10.1016/j.scico. 2017.10.006[33] Nicolas Lagaillardie、Rumyana Neykova 和 Nobuko Yoshida。2020。《在 Rust 中实现多方会话类型》。载于 COORDINATION 2020（《计算机科学讲义》，第 12134 卷），Simon Bliudze 和 Laura Bocchi（编）。Springer，马耳他瓦莱塔，127 136 10 1007 978 3 030 50029 0_8 – 。https://doi.org/ 。 / - - - - Nicolas [34] Lagaillardie、Rumyana Neykova 和 Nobuko Yoshida。。《在恐慌情况下确保安全：使用多方会话类型进行仿射 Rust 编程》。载于 2022 ECOOP 2022（LIPIcs，第 222 卷），Karim Ali 和 Jan Vitek（编）。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，德国柏林，4 1 4 29 10 4230 2022 4 : – : 。https://doi.org/ 。 / [35] LIPIcs.ECOOP. 。 Sam Lindley 和 J. Garrett Morris。。《在 Haslell 中嵌入会话类型》。载于 Haslell，Geoffrey Mainland（编）。2016, ACM，日本奈良，133 145 10 1145 2976002 2976018 – 。https://doi.org/ 。 / 。 Sam Lindley 和 J. Garrett Morris。。[36]《轻量级函数式会话类型》。见 [15]。[37] Hernán C. Melgratti 和 Luca Padovani。2017。《面向高阶会话的监护合约》。2017 ACM 程序设计语言学报 1, 2017 35 1 35 29 10 1145 3110279 ICFP ( )，: – : 。https://doi.org/ 。 / Robin [38] Milner、Joachim Parrow 和 David Walker。。《移动进程演算 I》。IC 100，1 (1992)，1–40。https://doi.org/1992 10 1016 0890 5401 92 90008 4 . / - ( ) - Anson Miu、Francisco Ferreira、Nobuko Yoshida 和 Fangyi Zhou。2021。《使[39] 用路由多方会话类型在 TypeScript 中进行通信安全的 Web 编程》。载于 CC '21，Aaron Smith、Delphine Demange 和 Rajiv Gupta（编）。ACM，线上活动，大韩民国，94 106 10 1145 3446804 3446854 – 。https://doi.org/ 。 / 。 Matthias Neubauer 和 Peter Thiemann。。《会话类型的一种实现》。载于 PADL 2004 （《计算机科学讲义》，第 3057 卷），2004 [40] Bharat Jayaraman（编）。Springer，美国得克萨斯州达拉斯，56–70。https://doi.org/10.1007/978-3- 540-24836-1 5_[41] Nicholas Ng、Nobuko Yoshida、Olivier Pernet、Raymond Hu 和 Yiannos Kryftis。2011。《使用 Session Java 进行安全并行编程》。载于 COORDINATION 2011（《计算机科学讲义》，第 6721 卷），Wolfgang De Meuter 和 Gruia-Catalin Roman（编）。Springer，冰岛雷克雅未克，110 126 10 1007 978 3 642 21464 6_8 – 。[42] https://doi.org/ 。 / - - - - Dominic A. Orchard 和 Nobuko Yoshida。。《作为会话的效应，作为效应的会话》。载于 POPL，Rastislav 2016, 2016. Bodík 和 Rupak Majumdar（编）。ACM，美国佛罗里达州圣彼得堡，568 581 10 1145 2837614 2837634 – 。https://doi.org/ 。 / 。 Luca [43] Padovani。。《二元会话的一个简单库实现》。函数式编程杂志 ( )，e 。https: //doi.org/ 2017 27 2017 4 10 1017 0956796816000289 . /S 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 (2019)，9:1–9:37。https://doi.org/[44] 2019 10 1145 3229062 . / Riccardo Pucella 和 Jesse A. Tov。2008。《Haskell 会话类型：几乎不需要类型类》。载于 Haskell [45] 2008，Andy Gill（编）。ACM，加拿大不列颠哥伦比亚省维多利亚，25 36 10 1145 1411286 1411290 – 。https://doi.org/ 。 / 。 António Ravara 和 Vasco [46] Thudichum Vasconcelos。。《并发对象演算的行为类型》。载于 Euro-Par（LNCS，第1997 1300 554 561 10 1007 0002782 卷）。Springer，德国帕绍，– 。[47] https://doi.org/ 。 /BFb John C. Reynolds。。《用户定义类型与过程式数据结构：数据抽象的互补方法》。载于《算法语言的新方向》，1975 Stephen A. Schumann（编）。INRIA。转载于 [48]。[48] John C. Reynolds。1994。《用户定义类型与过程式数据结构：数据抽象的互补方法》。载于《面向对象编程的理论方面：类型、语义与语言设计》，Carl A. Gunter 和 John C. Mitchell（编）。麻省理工学院出版社，美国马萨诸塞州剑桥，13–23。最初发表于 [47]。[49] Matthew Sackman 和 Susan Eisenbach。2008。《Haskell 中的会话类型》。面向21世纪的消息传递更新。(2008)。https://spiral.imperial.ac.uk:8443/handle/10044/1/5918。[50] Alceste Scalas 和 Nobuko Yoshida。2016。《Scala 中的轻量级会话编程》。载于 ECOOP 2016（LIPIcs，第 56 卷），Shriram Krishnamurthi 和 Benjamin S. Lerner（编）。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，意大利罗马，21 1 21 28 10 4230 2016 21 : – : 。[51] https://doi.org/ 。 /LIPIcs.ECOOP. 。 Wouter Swierstra。。《按需组合的数据类型》。函数式编程杂志 18，4 (2008)，423–436。2008 https://doi.org/10.1017/ S0956796808006758[52] Kaku Takeuchi、Kohei Honda 和 Makoto Kubo。1994。《一种基于交互的语言及其类型系统》。载于 PARLE ' 94 817 398 413 10 1007 3 540 58184 7_118 （LNCS，第[53] 卷）。Springer，希腊雅典，– 。https://doi.org/ 。 / - - - Peter Thiemann。。《程序生成组合子》。函数式编程杂志 ( )，– 。https: //1999 9, 5 1999 483 525 doi.org/ 10 1017 0956796899003469 . /s Peter Thiemann 和 Vasco T. Vasconcelos。2016。《上下文无关会话类型》。载于[54] ICFP 2016，Jacques Garrigue、Gabriele Keller 和 Eijiro Sumii（编）。ACM，日本奈良，462–475。https://doi.org/10.1145/2951913.2951926

[55] Peter Thiemann 和 Vasco T. Vasconcelos。2020。《标签依赖的会话类型》。ACM 程序设计语言学报 4，POPL（ 2020 67 1 67 29 10 1145 3371135），：–：。https://doi.org/。 / Bernardo Toninho、Luís Caires 和 Frank [56] Pfenning。2013。《高阶进程、函数与会话：单子式集成》。载于 ESOP（LNCS，第 7792 卷）。Springer，意大利罗马，350–369。[57] Jesse A. Tov 和 Riccardo Pucella。2010。《仿射类型的有状态契约》。载于 ESOP 2010（《计算机科学讲义》，第 6012 卷），Andrew D. Gordon（编）。Springer，塞浦路斯帕福斯，550–569。https://doi.org/10.1007/978-3-642- 11957-6 29_ [58] Jesse A. Tov 和 Riccardo Pucella。2011。《实用仿射类型》。载于 POPL。ACM，美国得克萨斯州奥斯汀，447–458。https: // doi.org/ 10 1145 1926385 1926436 . / 。 Vasco Thudichum Vasconcelos。2009。《会话类型基础》。载于 SFM（L [59] NCS，第 5569 卷）。Springer，意大利贝尔蒂诺罗， 158 186 10 1007 978 3 642 01918 0_4 – 。 https://doi.org/。 / - - - - Vasco Thudichum Vasconcelos、Simon J. Gay 和 António Ravara。2006。《使用会话类型对多线程函 [60] 数式语言进行类型检查》。TCS 368, 1 2 2006 64 87 10 1016 2006 06 028 - ( ), – 。https://doi.org/。 /j.tcs. . . Philip Wadler。。 2014 24, 2 3 2014 384 418 《作为会话的命题》。JFP - ( ), – 。[62] Max Willsey 、 Rokhini Prabhu 和 [61] Frank Pfenning。2016。《并发C0 的设计与实现》。载于 LINEARITY 2016（EPTCS，第 238 卷），Iliano Cervesato 和 Maribel Fernández（编）。Open Publishing Association，葡萄牙波尔图， 73 82 10 4204 238 8 – 。 https://doi.org/。 /EPTCS. 。 Li-yao Xia、Yannick Zakowski、Paul He、Chung-Kil Hur、Gregory Malecha、Benjamin C. [63] Pierce 和 Steve Zdancewic。2020。《交互树：在 Coq 中表示递归和非纯程序》。ACM 程序设计语言学报 4，POPL (2020), 51 1 51 32 10 1145 3371119 : – : 。https://doi.org/。 / Fangyi Zhou、Francisco Ferreira、Raymond Hu、 [64] Rumyana Neykova 和 Nobuko Yoshida。2020。《多方协议的静态验证细化》。ACM 程序设计语言学报 4，OOPSLA (2020), 148:1–148:30。https://doi.org/10. 1145/3428216