隔离事务的编舞式编程

Ton Smeele荷兰开放大学 荷兰海尔伦

Sung-Shik Jongmans荷兰开放大学，荷兰海尔伦；荷兰数学与计算机科学研究中心（CWI），荷兰阿姆斯特丹；ssj@ou.nl

实现分布式系统很困难；编舞式编程旨在让这件事变得更容易。本文介绍了一种新型编舞式编程语言的设计，该语言支持重叠进程集之间的隔离事务。第一个想法是跟踪每个变量允许哪些进程使用。第二个想法是使用模型检查来证明隔离性。

1 引言

1.1 背景：编舞式编程

实现分布式系统很难；编排式编程旨在让这件事变得更简单[8, 10, 37]。图 1 展示了这一理念。

最初，分布式系统被编写为一个全局程序 G（“编舞”）。它以顺序编程风格实现所有进程的整体行为（易于编写，但难以作为分布式系统运行）。例如，下面的全局程序实现了这样一个分布式系统：首先，数据对象从 Alice 传送到 Bob；其次，传送该对象的哈希值。

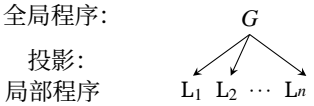


图 1：方法

$$G_{\mathsf{ab}} = (\mathsf{a.\texttt{"foo"}} \twoheadrightarrow \mathsf{b.x})\,;(\mathsf{a.hash} := \mathtt{md5(\texttt{"foo"})})\,;(\mathsf{a.hash} \twoheadrightarrow \mathsf{b.y})$$

这里，p.e ↠ q.y 和 q.y := e 表示进程间通信和进程内计算。通信 p.e ↠ q.y 将进程 p 处表达式 e 的值输出，并在进程 q 处将其输入变量 y；传输是异步、可靠且遵循先进先出的。计算 q.y := e 将进程 q 处表达式 e 的值存储在变量 y 中。

随后，分布式系统作为一组局部程序 L1, . . . ,L$_n$ 运行，这些程序通过投影从全局程序中自动提取。局部程序以并行编程风格分别实现每个进程的行为（易于作为分布式系统运行，但难以编写）。例如，以下局部程序实现了 Alice 和 Bob：

$$L_{\mathsf{a}} = (\mathsf{ab!}\texttt{"foo"})\,;(\mathsf{a.hash} := \mathtt{md5(\texttt{"foo"})})\,;(\mathsf{ab!hash}) \qquad L_{\mathsf{b}} = (\mathsf{ab?x})\,;(\mathsf{ab?y})$$

这里，发送 pq!e 和接收 pq?y 分别通过从 p 到 q 的通道实现输出和输入。

编舞式编程的核心保障是操作等价：具体而言，全局程序及其对应的一组局部程序可以保证具有相同的行为。要证明一组局部程序的性质，操作等价使我们能够转而证明全局程序的性质。这通常更简单。这类性质的典型例子是不存在死锁。

编舞式编程起源于 Carbone 等人 [7, 8]（使用二元会话类型 [31]）以及 Carbone 和 Montesi [10, 37]（使用多方会话类型 [32]）；此后该领域取得了长足进展。Montesi 和 Yoshida 构建了组合式编舞编程理论

Ilaria Castellani 和 Alceste Scalas（编）：第14届并发与通信中心软件编程语言方法研讨会（PLACES2023）。EPTCS 378，2023年，第49–60页，doi:10.4204/EPTCS.378.5

© T. Smeele 与 S. Jongmans 本作品依据知识共享署名许可协议授权。

支持开放式分布式系统 [38]；Carbone 等人研究了编舞式编程与线性逻辑之间的联系 [6, 11]；Dalla Preda 等人将编舞式编程与动态适应相结合 [39–41]；Cruz-Filipe 和 Montesi 开发了一种极简的图灵完备全局程序语言 [21]；Cruz-Filipe 等人和 Kjær 等人提出了从局部程序族中提取全局程序的技术 [17, 35]；Giallorenzo 等人研究了编舞式编程与多层语言之间的对应关系 [27]；Jongmans 和 Van den Bos 将编舞式编程与演绎验证相结合 [34]；Hirsch、Garg 以及 Cruz-Filipe 等人开发了函数式编舞式编程语言 [16, 30]。其他研究成果包括案例研究 [18],、过程抽象 [20], 、异步通信 [19], 、多元通信 [22, 29], 、可实现性 [26], , 以及在 Coq 中的形式化／机械化 [23,24,30]。这些理论进展也得到了多种工具的实践支持 [4, 10, 27, 40, 41]。

1.2 开放问题：隔离事务

假设我们需要实现一个满足以下要求的分布式系统：

1. 一个数据对象及其哈希值由 Alice 和 Carol 并行传送给 Bob。 2. 最终，Alice 的数据对象及其哈希值会存储在 Bob 处，或者 Carol 的（但不会混合存储） .

要求 1可以在支持并行组合（自由交错）的编舞式编程语言中轻松实现，如以下全局程序所示：

$$G^{\mathrm{v1}}_{\mathsf{acb}} = G^{\S 1.1}_{\mathsf{ab}} \parallel G_{\mathsf{cb}} \qquad G_{\mathsf{cb}} = (\mathtt{c."bar"} \twoheadrightarrow \mathtt{b.x})\,;\,(\mathtt{c.hash} := \mathtt{md5("bar")})\,;\,(\mathtt{c.hash} \twoheadrightarrow \mathtt{b.y})$$

相比之下，我们所知的任何编舞式编程语言都无法满足需求 2（也就是说，§1.1 中引用的编舞式编程语言似乎都不具备这一能力）。我们需要一种机制，将 $\mathsf{G_{ab}}$ 和 $\mathsf{G_{cb}}$ 作为隔离事务运行。

一种可能的做法是使用标准的非确定性选择算子 $+$ 来扩展语言。在这种情况下，系统可以实现为 $(\mathsf{G_{ab}}\,;\,\mathsf{G_{cb}}) + (\mathsf{G_{cb}}\,;\,\mathsf{G_{ab}})$。然而，这种将并行组合显式展开为选择的做法，通常会导致全局程序的规模呈指数级增长（相对于事务数量），同时模糊系统的设计意图。例如，如果将 Dave 添加为 Bob 的第三个客户端，我们就需要编写以下全局程序：

$$\begin{aligned} G^{\mathrm{v1}}_{\mathsf{abcd}} = {} & (G_{\mathsf{ab}}\,;\,((G_{\mathsf{cb}}\,;\,G_{\mathsf{db}}) + (G_{\mathsf{db}}\,;\,G_{\mathsf{cb}}))) + \\ & (G_{\mathsf{cb}}\,;\,((G_{\mathsf{ab}}\,;\,G_{\mathsf{db}}) + (G_{\mathsf{db}}\,;\,G_{\mathsf{ab}}))) + \\ & (G_{\mathsf{db}}\,;\,((G_{\mathsf{ab}}\,;\,G_{\mathsf{cb}}) + (G_{\mathsf{cb}}\,;\,G_{\mathsf{ab}}))) \end{aligned} \qquad \begin{aligned} G_{\mathsf{db}} = {} & (\mathtt{d."baz"} \twoheadrightarrow \mathtt{b.x})\,; \\ & (\mathtt{d.hash} := \mathtt{md5("baz")})\,; \\ & (\mathtt{d.hash} \twoheadrightarrow \mathtt{b.y}) \end{aligned}$$

此外，如果我们希望允许使用互不相交的变量集的独立事务片段相互重叠，以提升性能（即，它们的交错执行不会破坏隔离性），那么采用非确定性选择的方法会进一步增加编程难度。

为避免这些问题，本文提出了一种粒度更细的方法，既支持最终一致性，又允许隔离事务交错执行。在我们的方法中，无需通过枚举可接受的通信序列来手动实现隔离事务；相反，隔离性是通过编程语言的显式支持自然实现的。

1.3 本文的贡献

我们介绍一种支持隔离事务的新型编舞式编程语言的设计。

第一个想法是跟踪每个变量允许哪些进程使用。初始时，每个进程都允许使用每个变量。之后，进程 p 可以获取独占权限。

使用进程q的变量 y。当获得许可后，非 p 对 y的每次使用都会被阻塞，直到 p 释放其独占权限。使用权限的管理对程序员是透明的；这是编程语言的一项功能。以下全局程序展示了语法，并满足 §1.2 中的需求 2：

$$G^{\text{v2}}_{\text{acb}} = ((\text{a } \textbf{acq } \text{b.x}) \,;\, G^{\S1.1}_{\text{ab}} \,;\, (\text{a } \textbf{rel } \text{b.x})) \parallel ((\text{c } \textbf{acq } \text{b.x}) \,;\, G^{\S1.2}_{\text{cb}} \,;\, (\text{c } \textbf{rel } \text{b.x}))$$
$$G^{\text{v2}}_{\text{acdb}} = G^{\text{v2}}_{\text{acb}} \parallel ((\text{d } \textbf{acq } \text{b.x}) \,;\, G^{\S1.2}_{\text{db}} \,;\, (\text{d } \textbf{rel } \text{b.x}))$$

我们注意到，$G^{\text{v2}}_{\text{acdb}}$ 是由$G^{\text{v2}}_{\text{acb}}$以组合方式构造而成的，无需引用子程序$G_{\text{ab}}$ 和G_{cb}；当并行组合被显式展开为选择时，这是不可能的。

因此，跟踪使用权限——并阻止那些被禁止的使用行为——使程序员能够编写更加精简的全局程序，从而更好地保留系统的意图。然而：

– 此功能本身并不能保证隔离性；它只是实现隔离性的一种手段。换言之，仍然需要一种单独的机制来检查隔离性，并确保投影能够保持隔离性。– “阻止那些被禁止的用法”也会带来一个不利的副作用：竞争使用相同变量的进程可能会陷入死锁。例如，以下全局程序实现了这样一个系统：Alice 试图获得使用 Bob 的变量 x 和 y 的许可，而 Carol 也试图获得使用相同变量的许可，但顺序相反：

$$(\text{a } \textbf{acq } \text{b.x} \,;\, \text{a } \textbf{acq } \text{b.y} \,;\, \cdots) \parallel (\text{c } \textbf{acq } \text{b.y} \,;\, \text{c } \textbf{acq } \text{b.x} \,;\, \cdots)$$

当 Alice 获得使用 x 的许可，而 Carol 获得使用 y 的许可时，就会发生死锁，因此两人都无法获得使用第二个变量的许可。[1]

为了解决这些问题，本文的第二个想法是用时序逻辑描述各种性质，例如隔离性和无死锁，并使用模型检查来证明这些性质得到满足。我们认为，将这两者与编舞式编程结合起来是一个新颖的做法。

2 设计

我们定义了一种语言，其中既可以表达全局程序，也可以表达局部程序族。图2展示了设计。它包含四个层次：每个系统都由程序（一个全局程序，或多个局部程序）、存储（每个进程一个）和通道来定义（每对进程之间一个）；每个程序、存储或通道都由动作、进程/通道名称和数据定义；每个动作本身也由名称和数据定义。

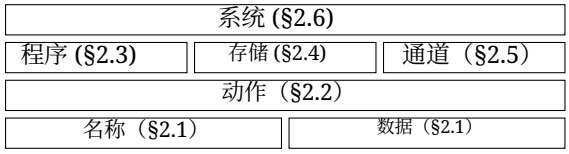


图2：设计

2.1 名称与数据

首先，我们定义：名称语法（定义1）；数据语法（定义2）。由于我们关注的主题是“进行通信的进程”，而非“被通信的数据”，因此省略了大部分细节。

[1]我们注意到，这是一种不同于编舞式编程通常避免的通信死锁（即等待一条永远不会发送的消息）的死锁来源。

定义1. 令 $\mathbb{R} = \{a,b,c \ldots\}$ 表示由p、q、r遍历的进程名称集合。令 $\mathbb{R} \times \mathbb{R} \setminus \{(r, r) \mid r \in R\}$ 表示通道名称集合。 □

定义2. 令 $\mathbb{X} = \{_, x、y、z \ldots\}$ 表示变量集合，其元素为 x、y、z。令 $\mathbb{V} = \{unit、true、false、0、1、2、...、acq、rel\}$ 表示值集合，其元素为 u、v、w。令 $\mathbb{E}$ 表示表达式集合，其元素为 E；其定义如下：

$$E ::= x \mid u \mid E_1 == E_2 \mid \sim E \mid E_1 \,\&\&\, E_2 \mid E_1 + E_2 \mid \cdots$$ □

Symbol是一种特殊变量，会丢弃写入其中的所有数据，类似于 Unix 中的 /dev/null。Symbol acq 和 rel 是用于控制变量使用权限的特殊值（§2.4）。

2.2 动作

接下来，我们定义：进程可以执行的动作的语法（定义 3）；用于获取动作的“主体”和“对象”的函数（定义 4）。主体是执行动作的进程；对象是执行动作所经过的通道（如果有）。

D定义 3. 令 $\mathbb{A}$ 表示动作的集合，其元素记作 α；定义如下：

$$\alpha ::= p.E \mid q.y := E \mid pq!E \mid pq?E \mid \tau$$ □

动作 p.E 在进程 p 处测试表达式 E。动作 q.y := E 在进程 q 处将表达式 E 的值赋给变量 y。动作 pq!E 和 pq?E 分别表示将表达式 E 的值从进程 p 异步发送至进程 q，以及从进程 p 接收该值。动作 τ 表示空闲。

定义 4. 令 subj(α) 和 obj(α) 分别表示 α的主体和对象；定义如下：

$$\begin{array}{lll} \mathsf{subj}(p.E) = p & \mathsf{subj}(q.y := E) = q & \mathsf{obj}(pq!E) = pq \\ \mathsf{subj}(pq!E) = p & \mathsf{subj}(pq?y) = q & \mathsf{obj}(pq?E) = pq \end{array}$$ □

2.3 程序

接下来，我们定义：程序的语法（定义 5）；从全局程序中提取局部程序的函数（定义 6）；程序的操作语义（定义 7）。

定义 第 5节。令 $\mathbb{P}$ 表示程序集合，其元素分别以 P、G、L 表示；定义如下 ：

$$P ::= \mathbf{1} \mid \alpha \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P_1 ; P_2$$ □

程序 1 实现一次空执行。程序 $P_1 + P_2$ 实现 P_1 和 P_2 之间的选择。程序 $P_1 \parallel P_2$ 实现 P_1 和 P_2 的交错执行。程序 $P_1 ; P_2$ 实现 P_1 和 P_2 的顺序执行。此外，我们使用以下简写记法：

$$\begin{array}{rl} p.E \twoheadrightarrow q.y & \text{instead of } pq!E ; pq?y \\ p \ \mathbf{acq} \ q.y & \text{instead of } (p.\mathtt{acq} \twoheadrightarrow q.y) ; (q.\mathtt{unit} \twoheadrightarrow p._) \\ p \ \mathbf{acq} \ q.[y_1, \ldots, y_n] & \text{instead of } p \ \mathbf{acq} \ q.y_1 ; \cdots ; p \ \mathbf{acq} \ q.y_n \\ p \ \mathbf{rel} \ q.y & \text{instead of } p.\mathtt{rel} \twoheadrightarrow q.y \\ p \ \mathbf{rel} \ q.[y_1, \ldots, y_n] & \text{instead of } p \ \mathbf{rel} \ q.y_1 ; \cdots ; p \ \mathbf{rel} \ q.y_n \\ \mathbf{if} \ p.e \ P_1 \ P_2 & \text{instead of } (p.E ; P_1) + (p.\sim E ; P_2) \end{array}$$

如果一个程序中至少出现两个主体，则它是全局程序；如果其中至多出现一个主体，则它是局部程序。可以通过投影从全局程序 G 中提取进程 r 的局部程序。其思路是将 G 中所有主体不是 r 的动作替换为 τ。

定义 n 6. 令 P ↾ r 表示 P 在 r 上的投影；它由以下方程导出。 :

$$\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}} \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'} \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 + P_2 \xrightarrow{\alpha} P_2'} \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2} \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1 \parallel P_2'}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 ; P_2 \xrightarrow{\alpha} P_1' ; P_2} \qquad \frac{\mathsf{subj}(\alpha) \notin \{\mathsf{subj}(\hat{\alpha}) \mid P_1 \to \cdots \xrightarrow{\hat{\alpha}}\} \qquad P_2 \xrightarrow{\alpha} P_2'}{P_1 ; P_2 \xrightarrow{\alpha} P_2'}$$

(a) 程序。令 $\to \cdots \to$ 表示由零个或多个归约组成的序列。

$$\frac{S[\![E]\!]_p = \mathtt{true}}{S \xrightarrow[p.\mathtt{true}]{p.E} S} \qquad \frac{S[\![E]\!]_q = v}{S \xrightarrow[q.y:=v]{q.y:=E} S[y \mapsto v]_q} \qquad \frac{S[\![E]\!]_p = u}{S \xrightarrow[pq!u]{pq!E} S} \qquad \frac{}{S \xrightarrow[pq?v]{pq?y} S[y \mapsto v]_p} \qquad \frac{}{S \xrightarrow[\tau]{\tau} S}$$

(b) 存储

$$\frac{}{C \xrightarrow[p.v]{} C} \qquad \frac{}{C \xrightarrow[q.y:=v]{} C} \qquad \frac{|\vec{v}| < n}{(\vec{v}, n) \xrightarrow[pq!u]{} (u \cdot \vec{v}, n)} \qquad \frac{}{(\vec{u} \cdot v, n) \xrightarrow[pq?v]{} (\vec{u}, n)} \qquad \frac{}{C \xrightarrow[\tau]{} C}$$

(c) 通道

$$\frac{P \xrightarrow{\alpha} P'}{\{\mathrm{P}\} \cup \mathcal{P} \xrightarrow{\alpha} \{\mathrm{P}'\} \cup \mathcal{P}} \qquad \frac{\mathrm{S} \xrightarrow[\underline{\alpha}]{\alpha} S' \qquad \mathsf{subj}(\alpha) = \mathrm{r}}{\{\text{主语}(\alpha) \mapsto \mathrm{S}\} \cup \mathcal{S} \xrightarrow[\underline{\alpha}]{\alpha} \{\mathsf{subj}(\alpha) \mapsto \mathrm{S}'\} \cup \mathcal{S}}$$

$$\frac{C \xrightarrow[\underline{\alpha}]{} C' \qquad \mathsf{obj}(\underline{\alpha}) = \mathrm{pq}}{\{\mathrm{pq} \mapsto \mathrm{C}\} \cup \mathcal{C} \xrightarrow[\underline{\alpha}]{} \{\mathrm{pq} \mapsto \mathrm{C}'\} \cup \mathcal{C}} \qquad \frac{\mathcal{P} \xrightarrow{\alpha} \mathcal{P}' \qquad \mathcal{S} \xrightarrow[\underline{\alpha}]{\alpha} \mathcal{S}' \qquad \mathcal{C} \xrightarrow[\underline{\alpha}]{} \mathcal{C}'}{(\mathcal{P}, \mathcal{S}, \mathcal{C}) \xrightarrow[\underline{\alpha}]{\alpha} (\mathcal{P}', \mathcal{S}', \mathcal{C}')}$$

(d) 系统

图 3：操作语义

$$\alpha \upharpoonright \mathsf{subj}(\alpha) = \alpha \qquad \mathbf{1} \upharpoonright r = \mathbf{1}$$
$$\alpha \upharpoonright r = \tau \quad \textbf{if: } r \neq \mathsf{subj}(\alpha) \qquad P_1 \circ P_2 \upharpoonright r = (P_1 \upharpoonright r) \circ (P_2 \upharpoonright r) \quad \textbf{if: } \circ \in \{+, \parallel, ;\}$$ □

我们通过带标签的归约关系来定义程序的操作语义。

定义 7. 令 $\mathrm{P} \xrightarrow{\alpha} \mathrm{P}'$ 表示从 P 到 P′ 的归约，且带有 α；其定义见图 3a。 □

大多数规则都是标准的。唯一特殊的是顺序结构的第二条规则：它允许动作序列乱序执行，只要这些动作在不同的进程中执行（也就是说，它们彼此独立；在并行环境中强行要求顺序执行是不合理的）。也就是说，该规则的左前提蕴含 α 的主体不会出现在 P_1 中（参见全局多方会话类型的操作语义）。例如，在 a.x := 5; b.y := 6 中，Alice 和 Bob 处的赋值可以乱序执行。相反，在 a.x := 5; a.x+1 → b.y 中，赋值和通信必须按顺序执行。

2.4 存储

接下来，我们定义：存储语法（定义 8）；从存储中读取表达式并向其中写入值的函数（定义 9）；存储的操作语义（定义 10）。

定义 8. 令 $\mathbb{S} = (\mathbb{X} \setminus \{_\}) \rightharpoonup (\mathbb{V} \times 2^{\mathbb{R}})$ 表示存储集合，记作 S。 □

存储 $S(x) = (u, R)$ 表示变量 x 的值为 u，且 R 中的进程获准使用它。通常，$R \in \{\mathbb{R}\} \cup \{\{r\} \mid r \in \mathbb{R}\}$：要么所有进程都获准使用 x（若 $R = \mathbb{R}$），要么只有一个进程获准使用（若 $R = \{r\}$ 对于某个 $r \in \mathbb{R}$）。每个进程都有自己的存储，但通过通信，其他进程也可以使用它。

定义 9. 令 $S[\![E]\!]_r$ 和 $S[y \mapsto v]_r$ 分别表示进程 r 在 S 中读取 E，以及进程 r 在 S 中将 v 写入 y；定义如下：

$$
\begin{array}{lll}
S[\![x]\!]_r = u \quad \textbf{if: } S(x) = (u, R) \textbf{ and } r \in R & & S[\![E_1 \texttt{ == } E_2]\!]_r = \ldots \\
S[\![u]\!]_r = u & & S[\![\texttt{~}E]\!]_r = \ldots \\
S[_ \mapsto v]_r = S & \begin{cases} \{y \mapsto (v, R)\} & \textbf{if: } \texttt{acq} \neq v \neq \texttt{rel} \\ \{y \mapsto (u, \{r\})\} & \textbf{if: } \texttt{acq} = v \neq \texttt{rel} \\ \{y \mapsto (u, \mathbb{R})\} & \textbf{if: } \texttt{acq} \neq v = \texttt{rel} \end{cases} & S[\![E_1 \texttt{ \&\& } E_2]\!]_r = \ldots \\
S[y \mapsto v]_r = \{x \mapsto S(x) \mid x \neq y\} \cup & & S[\![E_1 \texttt{ + } E_2]\!]_r = \ldots \\
\quad \textbf{if: } y \neq _ \textbf{ and } S(y) = (u, R) \textbf{ and } r \in R & & \quad \vdots \qquad \vdots \ \vdots
\end{array}
$$

□

写入 $S[y \mapsto \text{acq}]_r$ 和 $S[y \mapsto \text{rel}]_r$ 表示进程 r 尝试获取或释放使用 y 的独占权限，但不改变其值；只有当 r 已拥有使用权限（可能不是独占权限）时，此操作才会成功。

该定义的关键在于，当 r 不允许使用 E 或 y 中出现的变量时，$S[\![E]\!]_r$ 和 $S[y \mapsto v]_r$ 均未定义。存储的操作语义（下一个定义）利用了这种未定义性。我们注意到，当对不兼容类型的子表达式执行操作时，$S[\![E]\!]_r$ 也未定义。例如，$S[\![5 + \text{true}]\!]_r$ 未定义。类型系统可用于静态捕获此类错误；这与本文的目标无关。

我们通过带标签的归约关系来定义存储的操作语义。每次归约都有两个标签：一个动作（写在箭头上方）和该动作的“基底”版本（写在下方）。在基底版本中，每个表达式都被替换为其值（如果有）。

定义 10。令 $S \xrightarrow[\underline{\alpha}]{\alpha} S'$ 表示从 S 到 S′ 的归约，其标签为 α 和 $\underline{\alpha}$；定义见图 3b。 □

第一条规则规定，若 E 的值为真且 p 拥有足够的权限，则通过读取 E，在存储上执行测试 p.E。第二条规则规定，若 q 拥有足够的权限，则通过读取 E 并将 E 的值写入 y，执行赋值 q.y := E。第三条规则规定，若 p 拥有足够的权限，则通过读取 E 执行发送 pq!E。第四条规则规定，若 p 有权使用 y（而非 q；本质上，我们将接收视为远程赋值），则通过将 v 写入 y，执行接收 pq?y 及其基底版本 pq?v。如果进程没有足够的权限来应用某条规则，存储就无法规约，因此该动作处于阻塞状态。

2.5 通道

接下来，我们定义：通道语法（定义 11）；操作语义（定义 12）。此后，我们用 $\vec{u}$ 表示值列表，用 $v{\cdot}\vec{u}$ 和 $\vec{u}{\cdot}v$ 表示添加前缀和添加后缀。

D定义 11。令 $\mathbb{C} = \mathbb{V}^* \times \{0, 1, 2, \ldots, \infty\}$ 表示一个通道集合，其范围由 C 表示 . □

通道（$\neg u$，n）表示其容量为 n 的缓冲区包含 $\neg u$ 中的值；该缓冲区可靠且先进先出 .

我们通过带标签的归约关系定义通道的操作语义。由于通道包含值，每个归约都有一个标签：基动作（写在箭头下方）。

定义 12。令 $C \xrightarrow[\alpha]{} C'$ 表示从 C 到 C' 且带有 $\underline{\alpha}$ 的归约；其定义见图 3c。 □

第一条和第二条规则指出，测试和赋值会在通道上执行，但并未真正使用该通道。第三条规则指出，如果缓冲区未满，则通过将值入队到缓冲区来执行发送。第四条规则指出，如果缓冲区非空，则通过从缓冲区出队一个值来执行接收。此后，在归约标签无关紧要时，我们将省略它们。

2.6 系统

最后，我们定义：系统语法（定义 13）；操作语义（定义 14）；操作等价（定义 15）

定义 13. 令 $\mathbb{P} = 2^{\mathbb{P}} \setminus \{\emptyset\}$ 表示由（非空）程序集合构成的集合，其元素记为 $\mathcal{P}$。令 $\mathbb{S} = \mathbb{R} \rightharpoonup \mathbb{S}$ 表示存储集合族，其元素记为 $\mathcal{S}$。令 $\mathbb{C} = \mathbb{R} \times \mathbb{R} \rightharpoonup \mathbb{C}$ 表示通道集合族，其元素记为 $\mathcal{C}$。令 $\mathbb{P} \times \mathbb{S} \times \mathbb{C}$ 表示系统集合，其元素记为 $\mathcal{D}$。 □

系统 $(\mathcal{P},\mathcal{S},\mathcal{C})$ 表示 $\mathcal{P}$ 中的程序、$\mathcal{S}$ 中的存储和 $\mathcal{C}$ 中的通道共同执行。如果存在进程集合 $R = \{r1, \ldots, r_n\}$，使得 $\mathcal{S}$ 的定义域为 R（每个进程都有一个存储），且 $\mathcal{C}$ 的定义域为 $R \times R$（每对进程之间都有一个通道），则该系统是良构的，并且：

$$\mathcal{P} \in \{\{P\} \mid P \text{ 是全局的，并且 } P \text{ 中出现的每个主体都出现在 } R\} \cup \{\{P1, \ldots, P_n\} \mid \text{对于每个 } 1 \leq i \leq n,\ P_{r_i} \text{ 都是局部的，并且 } P_{r_i} \text{ 中出现的每个主体都是 } r_i\}$$

我们通过带标签的归约关系定义系统的操作语义。 .

定义 14. 设 $(\mathcal{P},\mathcal{S},\mathcal{C}) \xrightarrow[\underline{\alpha}]{\alpha} (\mathcal{P},\mathcal{S},\mathcal{C})'$ 表示从 $(\mathcal{P},\mathcal{S},\mathcal{C})$ 到 $(\mathcal{P},\mathcal{S},\mathcal{C})$ 的归约，其中 α 和 $\underline{\alpha}$；其定义见图 3d。 □

第一、第二和第三条规则将归约从单个程序、存储和通道提升到程序集合、存储集合族和通道集合族。第四条规则将它们连接起来。

如果两个系统具有相同的行为，则它们在操作上等价。我们用分支双模拟等价 [28]（而非通常采用的迹等价）来形式化“具有相同的行为”，因为：它对空闲不敏感；它保持许多时序逻辑中的公式有效性（包括 LTL、CTL、CTL^* 和 μ-演算，但需满足一定条件），而这正是我们用来描述全局程序性质所需要的。两个系统（分别是进程、存储、通道、进程集合、存储集合族、通道集合族）分支双模拟等价，当且仅当它们能够反复模拟彼此的归约，允许空闲。

定义15. 令 $\{\approx_1, \approx_2, \ldots\}$ 表示分支双模拟的集合，以 $\approx$为其元素；其定义如下，采用余归纳方式：

- 对于每个 $\mathcal{D}_1 \xrightarrow[\tau]{\tau}{}^* \mathcal{D}_1\dagger \xrightarrow[\underline{\alpha}]{\alpha} \mathcal{D}_1\ddagger \xrightarrow[\tau]{\tau}{}^* \mathcal{D}'_1$，存在某个 $\mathcal{D}_2 \xrightarrow[\tau]{\tau}{}^* \mathcal{D}_2\dagger \xrightarrow[\underline{\alpha}]{\alpha} \mathcal{D}_2\ddagger \xrightarrow[\tau]{\tau}{}^* \mathcal{D}'_2$，使得 $\mathcal{D}_1\dagger \approx \mathcal{D}_2\dagger, \mathcal{D}_1\ddagger \approx \mathcal{D}_2\ddagger, \mathcal{D}'_1 \approx \mathcal{D}'_2$
- 对于每个 $\mathcal{D}_2 \xrightarrow[\tau]{\tau}{}^* \mathcal{D}_2\dagger \xrightarrow[\underline{\alpha}]{\alpha} \mathcal{D}_2\ddagger \xrightarrow[\tau]{\tau}{}^* \mathcal{D}'_2$，对于某个 $\mathcal{D}_1 \xrightarrow[\tau]{\tau}{}^* \mathcal{D}_1\dagger \xrightarrow[\underline{\alpha}]{\alpha} \mathcal{D}_1\ddagger \xrightarrow[\tau]{\tau}{}^* \mathcal{D}'_1$，$\mathcal{D}_1\dagger \approx \mathcal{D}_2\dagger, \mathcal{D}_1\ddagger \approx \mathcal{D}_2\ddagger, \mathcal{D}'_1 \approx \mathcal{D}'_2$

$$\mathcal{D}_1 \approx \mathcal{D}_2$$

令 $\equiv = \approx_1 \cup \approx_2 \cup \cdots$ 表示操作等价（即最大的分支双模拟）。 □

以下命题指出，程序集合的操作等价意味着它们所构成的系统也操作等价。具体而言，若 P 是一个全局程序，且 $\{\mathrm{P}\} \equiv \{\mathrm{P} \upharpoonright \mathrm{r} \mid \mathrm{r} \text{ 是 } \mathrm{P}\}$ 的一个主体，那么从 P 中提取的局部程序在任意初始存储和通道下都与 P 具有相同的行为。在本文所述的不含循环的情况下，检查 $\mathcal{P}_1 \equiv \mathcal{P}_2$ 显然是可判定的；而存在循环时则不可判定。我们将 $\equiv$ 的可判定近似（例如，通常针对选择语法规定的良构条件）留待未来研究，届时我们将把循环纳入工作范围。

命题 1. 对所有 $\mathcal{S},\mathcal{C}$，如果 $\mathcal{P}_1 \equiv \mathcal{P}_2$，那么 $(\mathcal{P}_1,\mathcal{S},\mathcal{C}) \equiv (\mathcal{P}_2,\mathcal{S},\mathcal{C})$. □

2.7 性质

为了证明性质，我们采用一种具有 CTL [25] 风格的基于状态的时序逻辑。我们主要关注两类性质（当然，也可以指定其他类别）：事务隔离性和无死锁；我们的逻辑设有专门的谓词来表述这类性质。之所以需要明确证明无死锁，是因为本文中的系统并非通过构造保证无死锁。例如，由以下程序组成的任何系统都可能发生死锁（§1.3 中最后一个例子的详细说明）：

$$G^{\mathrm{v3}}_{\mathsf{acb}} = ((\mathsf{a}\ \mathbf{acq}\ \mathsf{b}.[\mathrm{x},\mathrm{y}])\ ;\ G^{\S1.1}_{\mathsf{ab}}\ ;\ (\mathsf{a}\ \mathbf{rel}\ \mathsf{b}.[\mathrm{x},\mathrm{y}])) \parallel ((\mathsf{c}\ \mathbf{acq}\ \mathsf{b}.[\mathrm{y},\mathrm{x}])\ ;\ G^{\S1.2}_{\mathsf{cb}}\ ;\ (\mathsf{c}\ \mathbf{rel}\ \mathsf{b}.[\mathrm{x},\mathrm{y}]))$$

问题在于，Alice 按照先获取 x 再获取 y 的顺序操作，而 Carol 则按照先获取 y 再获取 x 的顺序操作。

Defi义 16. 令 $\mathbb{F}$ 表示由公式构成的集合，其元素记为 φ；定义如下：

$$\varphi \ ::= \ \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{EG}(\varphi) \mid \mathsf{EU}(\varphi_1,\varphi_2) \mid p.E \mid \mathsf{AX}_{q.y}(\varphi) \mid \mathsf{dead}$$ □

公式 $\top$ 表示真。公式 $\neg\varphi$ 和 $\varphi_1 \wedge \varphi_2$ 表示否定和合取。公式 EG(φ)表示在某个分支中，φ 始终为真。公式 EU(φ_1,φ_2)表示在某个分支中，φ_1为真，直到 φ_2 为真。公式 p.E 表示 p 处的命题 E。公式 $\mathsf{AX}_{q.y}(\varphi)$表示如果进程 q 中的变量 y 发生了变化，则 φ下一步为真。公式 dead 表示存在死锁。此外，我们使用以下简写符号（标准记法）：

$$\bot \text{ instead of } \neg\top \qquad \mathsf{AG}(\varphi) \text{ instead of } \mathsf{EU}(\top,\neg\varphi)$$
$$\phi_1 \vee \phi_2 \text{ instead of } \neg(\neg\phi_1 \wedge \neg\phi_2) \qquad \mathsf{AU}(\varphi_1,\varphi_2) \text{ instead of } \neg(\mathsf{EU}(\neg\varphi_2,\neg(\varphi_1 \vee \varphi_2)) \vee \mathsf{EG}(\neg\varphi_2))$$

定义17. 令 $\mathcal{D} \models \varphi$表示 $\mathcal{D}$对 φ 的蕴涵；其定义如下：

$$\frac{}{\mathcal{D} \models \top} \qquad \frac{\mathcal{D} \not\models \varphi}{\mathcal{D} \models \neg\varphi} \qquad \frac{\mathcal{D} \models \varphi_1 \qquad \mathcal{D} \models \varphi_2}{\mathcal{D} \models \varphi_1 \wedge \varphi_2} \qquad \frac{\mathcal{D} \to \mathcal{D}' \qquad \mathcal{D} \models \varphi \qquad \mathcal{D}' \models \mathsf{EG}(\varphi)}{\mathcal{D} \models \mathsf{EG}(\varphi)}$$

$$\frac{\mathcal{D} \models \varphi_2}{\mathcal{D} \models \mathsf{EU}(\varphi_1,\varphi_2)} \qquad \frac{\mathcal{D} \to \mathcal{D}' \qquad \mathcal{D} \models \varphi_1 \qquad \mathcal{D}' \models \mathsf{EU}(\varphi_1,\varphi_2)}{\mathcal{D} \models \mathsf{EU}(\varphi_1,\varphi_2)}$$

$$\frac{\mathcal{S} \xrightarrow[p.\mathtt{true}]{p.E} \mathcal{S}}{(\mathcal{P},\mathcal{S},\mathcal{C}) \models p.E} \qquad \frac{\text{for each } (\mathcal{P},\mathcal{S},\mathcal{C}) \to (\mathcal{P}',\mathcal{S}',\mathcal{C}') \text{ if } \mathcal{S}(q)(y) \neq \mathcal{S}'(q)(y), \text{ then } (\mathcal{P}',\mathcal{S}',\mathcal{C}') \models \varphi}{(\mathcal{P},\mathcal{S},\mathcal{C}) \models \mathsf{AX}_{q.y}(\varphi)} \qquad \frac{\mathcal{P} \to}{(\mathcal{P},\mathcal{S},\mathcal{C}) \models \mathsf{dead}}$$ □

前两行的规则是计算树逻辑（CTL）的标准规则。第三行的第一条规则指出，如果相应的测试成功，则命题为真。第三行的第二条规则指出，每个会改变进程 q 中变量 y 的归约都必须使 φ 为真。第三行的第三条规则指出，如果程序集合可以归约，而系统不能归约（即程序归约被存储和/或通道阻塞），则死锁的存在为真。

在没有循环的情况下（如本文所述），模型检查问题是可判定的：只需稍作调整，便可将经典的 CTL 模型检查算法（例如 Clarke 等人 [13] 的算法）扩展为支持我们的公式 p.E、$\mathsf{AX}_{q.y}$(φ) 和 dead。如果全局程序 G 满足操作等价，那么只需对由 G 构成的系统进行模型检查，无须对由 G 的投影构成的系统进行模型检查；前者通常效率高得多，因为 G 的投影的状态空间可能比 G 的状态空间大几个数量级（这是由于投影的 τ-归约所致）。

2.8 示例

我们以一些示例结束本节。设：

$$\mathcal{S} = \{\mathsf{a} \mapsto \{\mathtt{hash} \mapsto 0\}, \mathsf{b} \mapsto \{\mathtt{x} \mapsto \texttt{""}, \mathtt{y} \mapsto 0\}, \mathsf{c} \mapsto \{\mathtt{hash} \mapsto 0\}\}$$
$$\mathcal{C} = \{pq \mapsto (\epsilon, \infty) \mid p, q \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \text{ and } p \neq q\}$$

换言之，$\mathcal{S}$ 是一个初始存储族（分别属于 Alice、Bob 和 Carol），其中所有变量都具有默认值；而 $\mathcal{C}$ 是一个初始空通道族（位于 Alice、Bob 和 Carol 之间）。此外，除了 §1.2 中的 $\mathsf{G}^{\mathsf{v1}}_{\mathsf{acb}}$、§1.3 中的 $\mathsf{G}^{\mathsf{v2}}_{\mathsf{acb}}$ 和 §2.7 中的 $\mathsf{G}^{\mathsf{v3}}_{\mathsf{acb}}$ 之外，设：

$$\mathsf{G}^{\mathsf{v4}}_{\mathsf{acb}} = ((\mathsf{a}\ \mathbf{acq}\ \mathsf{b.x})\ ;\ G^{\S 1.1}_{\mathsf{ab}}\ ;\ (\mathsf{a}\ \mathbf{rel}\ \mathsf{b.x}))\ \|\ G^{\S 1.2}_{\mathsf{cb}} \qquad G^{\mathsf{v5}}_{\mathsf{acb}} = (G^{\S 1.1}_{\mathsf{ab}}\ ;\ G^{\S 1.2}_{\mathsf{cb}}) + (G^{\S 1.2}_{\mathsf{cb}}\ ;\ G^{\S 1.1}_{\mathsf{ab}})$$

– 关于事务隔离性，需要证明的性质可指定如下：

$$\varphi = \mathsf{AG}(\mathsf{AX}_{\mathsf{b.x}}(\mathsf{AU}(\mathsf{AX}_{\mathsf{b.x}}(\bot) \wedge \mathsf{AX}_{\mathsf{b.y}}(\bot), \mathsf{AX}_{\mathsf{b.y}}(\mathtt{b.(md5(x) == y)})))))$$

也就是说，在所有分支中，始终（AG）满足：如果 x 在 Bob 处发生变化（$\mathsf{AX}_{\mathsf{b.x}}$），那么在 y 于 Bob 处发生变化（$\mathsf{AX}_{\mathsf{b.y}}$）之前，x 不会再次发生变化（$\mathsf{AX}_{\mathsf{b.x}}(\bot)$），且 x 和 y 保持一致（b.(md5(x) == y)）。系统（$\{\mathsf{G}^{\mathsf{v1}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）违反了 φ，正如 §1.2 中的非正式说明。系统（$\{\mathsf{G}^{\mathsf{v2}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）满足φ，因为 Alice 和 Carol 获得了在 Bob 处使用 x 和 y 的独占权限。系统（$\{\mathsf{G}^{\mathsf{v3}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）也满足 φ：系统发生死锁时，Bob 处的 x 尚未发生变化；系统未发生死锁时，Alice 和 Carol 获得独占权限。系统（$\{\mathsf{G}^{\mathsf{v4}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）违反了 φ：虽然 $\mathsf{G}^{\S 1.1}_{\mathsf{ab}}$ 作为隔离事务运行（因为 Alice 确实获得了独占权限），但 $\mathsf{G}^{\S 1.2}_{\mathsf{cb}}$ 并非如此（因为 Carol 没有获得独占权限）。系统（$\{\mathsf{G}^{\mathsf{v5}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）也满足 φ，但违反了操作等价。——关于不存在死锁，要证明的性质可表述为 $\varphi = \mathsf{AG}(\neg\mathsf{dead})$。系统（$\{\mathsf{G}^{\mathsf{v1}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）、（$\{\mathsf{G}^{\mathsf{v2}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）、（$\{\mathsf{G}^{\mathsf{v4}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）和（$\{\mathsf{G}^{\mathsf{v5}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）均满足 φ。相比之下，系统（$\{\mathsf{G}^{\mathsf{v3}}_{\mathsf{acb}}\}$，$\mathcal{S}$，$\mathcal{C}$）违反了 φ。

3 结论

3.1 相关工作

编舞式编程方面的进展已在 §1.1 中提及。除编舞式编程之外，与我们工作最接近的是会话类型相关文献中用于保障互斥的机制。在二元会话类型相关文献中，Balzer 等人 [1]（不保证无死锁）以及 Balzer 等人和 Kokke 等人 [2, 36]（保证无死锁）的工作，通过针对线性和共享通道的类型规则，支持互斥及相关模式。在多方会话类型相关文献中，Voinea 等人 [42] 的工作通过针对线性和共享通道的类型规则支持互斥，适用于多个进程共同实现单个角色这一特殊情形。更一般地，并行组合在多方会话类型的语境下以多种方式得到研究：通过类型的静态交错（例如，[32, 33]）；通过程序的动态交错（例如，[3, 14]）；以及结合上述两种方式（例如，以嵌套的形式，[9, 12]）。

3.2 本文工作

我们介绍了一种新的编舞式编程语言的设计，该语言支持重叠进程集之间的隔离事务。第一个想法是追踪每个变量允许哪些进程使用。第二个想法是使用模型检查来证明隔离性。本文是我们首次深入探索这些想法的工作。我们认为，无论是探索其他设计方案，还是改进本文所介绍的工作，都仍有很大的空间。例如，可以在编舞式编程语言中加入新的原语来实现程序，也可以在时序逻辑中加入新的模态来描述性质。

3.3 未来工作

在理论方面，我们看到了三个主要方向。首先，我们计划为编舞式编程语言扩展一些原语，从而在构造上保证隔离性并避免死锁。一种可能的设计是采用“isolate P”形式的原语，将 P 实现为一个隔离事务。难点在于定义操作语义，使变量的独占权限能够尽可能晚地自动获取，并尽可能早地释放，同时避免死锁（例如，通过对变量施加全序）。其次，我们计划研究为编舞式编程语言添加循环的扩展。第三，我们计划研究用于证明性质的符号方法。

在实践方面，我们目前正在开发一个概念验证实现，将设计构建为从编舞式编程语言到 mCRL2 [5, 15] 的编译器。给定一个全局程序，编译器通过投影提取一组局部程序，并将全局程序及其对应的一组局部程序都转换为 mCRL2 规格。借助 mCRL2 工具集，我们便可以检查全局程序的性质（我们的 CTL 公式的 μ-演算版本）以及操作等价性。

参考文献

[1] Stephanie Balzer & Frank Pfenning（2017年）：基于会话类型的显式共享。《ACM 程序设计语言会议论文集》1（ICFP），第 37:1–37:29 页，doi:10.1145/3110281。[2] Stephanie Balzer、Bernardo Toninho & Frank Pfenning（2019年）：共享会话类型的显式无死锁性。载于：ESOP，《计算机科学讲义》11423，Springer，第 611–639 页，doi:10.1007/978-3-030-17184-1_22。[3] Lorenzo Bettini、Mario Coppo、Loris D’Antoni、Marco De Luca、Mariangiola Dezani-Ciancaglini & Nobuko Yoshida（2008年）：动态交错的多方会话中的全局进展。载于：CONCUR，《计算机科学讲义》5201，Springer，第 418–433 页，doi:10.1007/978-3-540-85361-9_33。[4] Petra van den Bos & Sung-Shik Jongmans（2023）：VeyMont：并行化已验证程序，而非验证并行程序。载于：FM，《计算机科学讲义》14000，Springer，第 321–339 页，doi:10.1007/978-3-031-27481-7_19。[5] Olav Bunte、Jan Friso Groote、Jeroen J. A. Keiren、Maurice Laveaux、Thomas Neele、Erik P. de Vink、Wieger Wesselink、Anton Wijs & Tim A. C. Willemse（2019年）：用于分析并发系统的 mCRL2 工具集——表达能力和易用性的改进。载于：TACAS（2），《计算机科学讲义》11428，Springer，第 21–39 页，doi:10.1007/978-3-030-17465-1_2。[6] Marco Carbone、Lu´ıs Cruz-Filipe、Fabrizio Montesi & Agata Murawska（2018）：多方经典编舞。载于：LOPSTR，《计算机科学讲义》11408，Springer，第 59–76 页，doi:10.1007/978-3-030-13838-7_4。[7] Marco Carbone、Kohei Honda & Nobuko Yoshida（2007）：面向 Web 服务的结构化、以通信为中心的编程。载于：ESOP，《计算机科学讲义》4421，Springer，第 2–17 页，doi:10.1007/978-3-540-71316-6_2。

[8] Marco Carbone、Kohei Honda & Nobuko Yoshida（2012）：面向 Web 服务的结构化、以通信为中心的编程。《ACM 程序设计语言与系统汇刊》34(2)，第 8:1–8:78 页，doi:10.1145/2220365.2220367。[9] Marco Carbone、Sam Lindley、Fabrizio Montesi、Carsten Sch¨urmann & Philip Wadler（2016）：相干性推广对偶性：多方会话类型的逻辑解释。载于：CONCUR，LIPIcs 59，Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，第 33:1–33:15 页，doi:10.4230/LIPIcs.CONCUR.2016.33。[10] Marco Carbone & Fabrizio Montesi（2013）：设计即无死锁：多方异步全局编程。载于：POPL，ACM，第 263–274 页，doi:10.1145/2429069.2429101。[11] Marco Carbone、Fabrizio Montesi & Carsten Sch¨urmann（2018）：从逻辑角度看编舞。《分布式计算》31(1)，第 51–67 页，doi:10.1007/s00446-017-0295-1。[12] Marco Carbone、Fabrizio Montesi、Carsten Sch¨urmann & Nobuko Yoshida（2017）：作为相干性证明的多方会话类型。《信息学学报》54(3)，第 243–269 页，doi:10.1007/s00236-016-0285-y。[13] Edmund M. Clarke、E. Allen Emerson & A. Prasad Sistla（1986）：使用时序逻辑规约自动验证有限状态并发系统。《ACM 程序设计语言与系统汇刊》8(2)，第 244–263 页，doi:10.1145/5397.5399。[14] Mario Coppo、Mariangiola Dezani-Ciancaglini、Nobuko Yoshida & Luca Padovani（2016）：动态交错的多方会话的全局进展。《计算机科学中的数学结构》26(2)，第 238–302 页，doi:10.1017/S0960129514000188。[15] Sjoerd Cranen、Jan Friso Groote、Jeroen J. A. Keiren、Frank P. M. Stappers、Erik P. de Vink、Wieger Wesselink & Tim A. C. Willemse（2013）：mCRL2 工具集及其最新进展概述。载于：TACAS，《计算机科学讲义》7795，Springer，第 199–213 页，doi:10.1007/978-3-642-36742-7_15。[16] Lu´ıs Cruz-Filipe、Eva Graversen、Lovro Lugovic、Fabrizio Montesi & Marco Peressotti（2022）：函数式编舞编程。载于：ICTAC，《计算机科学讲义》13572，Springer，第 212–237 页，doi:10.1007/978-3-031-17715-6_15。[17] Lu´ıs Cruz-Filipe、Kim S. Larsen & Fabrizio Montesi（2017）：编舞提取路径。载于：FoS-SaCS，《计算机科学讲义》10203，第 424–440 页，doi:10.1007/978-3-662-54458-7_25。[18] Lu´ıs Cruz-Filipe & Fabrizio Montesi（2016）：实践中的编舞。载于：FORTE，《计算机科学讲义》9688，Springer，第 114–123 页，doi:10.1007/978-3-319-39570-8_8。[19] Lu´ıs Cruz-Filipe & Fabrizio Montesi（2017）：在编舞中编码异步性。载于：SAC，ACM，第 1175–1177 页，doi:10.1145/3019612.3019901。[20] Lu´ıs Cruz-Filipe & Fabrizio Montesi（2017）：过程式编舞编程。载于：FORTE，《计算机科学讲义》10321，Springer，第 92–107 页，doi:10.1007/978-3-319-60225-7_7。[21] Lu´ıs Cruz-Filipe & Fabrizio Montesi（2020）：编舞编程的核心模型。《理论计算机科学》802，第 38–66 页，doi:10.1016/j.tcs.2019.07.005。[22] Lu´ıs Cruz-Filipe、Fabrizio Montesi & Marco Peressotti（2018）：重新审视编舞中的通信。载于：SAC，ACM，第 1248–1255 页，doi:10.1145/3167132.3167267。[23]Lu´ıs Cruz-Filipe、Fabrizio Montesi & Marco Peressotti（2021）：编舞编译的认证。载于：ICTAC，《计算机科学讲义》12819，Springer，第 115–133 页，doi:10.1007/978-3-030-85315-0_8。[24] Lu´ıs Cruz-Filipe、Fabrizio Montesi & Marco Peressotti（2021）：在 Coq 中形式化图灵完备的编舞语言。载于：ITP，LIPIcs 193，Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，第 15:1–15:18 页，doi:10.4230/LIPIcs.ITP.2021.15。[25] E. Allen Emerson & Edmund M. Clarke（1982）：使用分支时态逻辑综合同步框架。《科学计算机程序》2(3)，第 241–266 页，doi:10.1016/0167-6423(83)90017-5。[26] Saverio Giallorenzo、Fabrizio Montesi & Maurizio Gabbrielli（2018）：应用编舞。载于：FORTE，《计算机科学讲义》10854，Springer，第 21–40 页，doi:10.1007/978-3-319-92612-4_2。

[27] Saverio Giallorenzo、Fabrizio Montesi、Marco Peressotti、David Richter、Guido Salvaneschi 和 Pascal Weisenburger（2021年）：多方语言：编舞式与多层案例（Pearl）。载于：ECOOP，LIPIcs 194，Schloss Dagstuhl - Leibniz-Zentrum für Informatik，第 22:1–22:27 页，doi:10.4230/LIPIcs.ECOOP.2021.22。[28] Rob J. van Glabbeek 和 W. P. Weijland（1996年）：互模拟语义中的分支时间与抽象。J.ACM43(3)，第 555–600 页，doi:10.1145/233551.233556。[29] Thomas T. Hildebrandt、Tijs Slaats、Hugo A. L´opez、Søren Debois 和 Marco Carbone（2019年）：声明式编舞与活性。载于：FORTE，《计算机科学讲义》11535，Springer，第 129–147 页，doi:10.1007/978-3-030-21759-4_8。[30] Andrew K. Hirsch 和 Deepak Garg（2022年）：Pirouette：高阶类型化函数式编舞。Proc. ACM Program. Lang. 6(POPL)，第 1–27 页，doi:10.1145/3498684。[31] Kohei Honda、Vasco Thudichum Vasconcelos 和 Makoto Kubo（1998年）：基于结构化通信的编程的语言原语与类型纪律。载于：ESOP，《计算机科学讲义》1381，Springer，第 122–138 页，doi:10.1007/BFb0053567。[32] Kohei Honda、Nobuko Yoshida 和 Marco Carbone（2008年）：多方异步会话类型。载于：POPL，ACM，第 273–284 页，doi:10.1145/1328438.1328472。[33] Kohei Honda、Nobuko Yoshida 和 Marco Carbone（2016年）：多方异步会话类型。J.ACM 63(1)，第 9:1–9:67 页，doi:10.1145/2827695。[34] Sung-Shik Jongmans 和 Petra van den Bos（2022年）：编舞的谓词转换器——计算编舞式编程中的前置条件。载于：ESOP，《计算机科学讲义》13240，Springer，第 520–547 页，doi:10.1007/978-3-030-99336-8_19。[35] Bjørn Angel Kjær、Lu´ıs Cruz-Filipe 和 Fabrizio Montesi（2022年）：从无穷到编舞——无界系统的提取。载于：LOPSTR，《计算机科学讲义》13474，Springer，第 103–120 页，doi:10.1007/978-3-031-16767-6_6。[36] Wen Kokke、J. Garrett Morris 和 Philip Wadler（2020年）：迈向线性逻辑中的竞态。Log. Methods Comput. Sci. 16(4)，doi:10.23638/LMCS-16(4:15)2020。[37] Fabrizio Montesi（2013年）：编舞式编程。博士论文，哥本哈根信息技术大学。https://www.fabriziomontesi.com/files/choreographic-programming.pdf。[38] Fabrizio Montesi 和 Nobuko Yoshida（2013年）：组合式编舞。载于：CONCUR，《计算机科学讲义》8052，Springer，第 425–439 页，doi:10.1007/978-3-642-40184-8_30。[39] Mila Dalla Preda、Maurizio Gabbrielli、Saverio Giallorenzo、Ivan Lanese 和 Jacopo Mauro（2015年）：动态编舞——分布式应用的安全运行时更新。载于：COORDINATION，《计算机科学讲义》9037，Springer，第 67–82 页，doi:10.1007/978-3-319-19282-6_5。[40] Mila Dalla Preda、Maurizio Gabbrielli、Saverio Giallorenzo、Ivan Lanese 和 Jacopo Mauro（2017年）：动态编舞：理论与实现。Log. Methods Comput. Sci. 13(2)，doi:10.1007/BF01221097。[41] Mila Dalla Preda、Saverio Giallorenzo、Ivan Lanese、Jacopo Mauro 和 Maurizio Gabbrielli（2014年）：AIOCJ：用于安全自适应分布式应用的编舞框架。载于：SLE，《计算机科学讲义》8706，Springer，第 161–170 页，doi:10.1007/978-3-319-11245-9_9。[42] A. Laura Voinea、Ornela Dardha 和 Simon J. Gay（2019年）：通过基于能力的多方会话类型实现资源共享。载于：IFM，《计算机科学讲义》11918，Springer，第 437–455 页，doi:10.1007/978-3-030-34968-4_24。