arXiv:2602.05850v1 [cs.PL] 5 Feb 2026

通过代数效应对动态线程进行方程公理化

有限关系上的预层、标记偏序集与参数化代数理论

OHAD KAMMAR，爱丁堡大学，英国JACK LIELL-COCK，牛津大学，英国SAM LINDLEY，爱丁堡大学，英国CRISTINA MATACHE，爱丁堡大学，英国SAM STATON，牛津大学，英国

我们利用代数效应理论，为动态线程给出完备的方程公理化。我们的方法基于参数化代数理论；这类理论为函子范畴上的强单子提供了具体语法，也是处理名称与绑定的便利框架。

我们的程序由关键原语“分叉”和“等待”构成。“分叉”创建一个子线程，并将其名称（线程标识符）传递给父线程。“等待”使我们能够等待指定的子线程完成。我们给出一个由分叉和等待构成的参数化代数理论，其中还包括基本的原子动作，以及“分叉”的结合律等法则。

我们的等式公理化在两种意义上是完备的。首先，对于闭表达式，它完全刻画了标记偏序集（并发偏序集）相等；后者是业已确立的并发模型：模型完备。其次，若两个开放表达式在所有闭合代换下都相等，则它们可由公理证明相等：语法完备。

代数效应的优势在于，语义分析可以聚焦于分叉与等待这两种代数运算。随后，我们通过给出操作语义和指称语义，将分析扩展到一种简单的并发编程语言。指称语义采用参数化代数理论的方法构建；我们证明，对于标记偏序集观察，它是可靠的、适当的，并且在一阶情形下完全抽象。

CCS 概念：• **计算理论** → **指称语义**；**范畴语义**；**并发**。

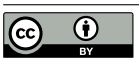
其他关键词和短语：参数化代数理论、Lawvere 理论、单子、预层、多线程编程抽象、类 Unix 派生、因果语义、并发偏序集、表示定理。

1 引言

代数效应理论提供了一种分析不同计算效应语义层面的方式：借助普遍代数中的代数理论，可以将这些效应彼此隔离，并与编程语言的其他层面分开研究。本文运用代数效应的方法对并发进行分析。

并发代数效应理论一直难以建立 [34, 49]。尽管进程代数中的等式推理和组合推理 [6, 16, 26, 41], 以及并发 Kleene 代数等并发等式理论 [14, 15]都已取得成功 [32, 42], , 但这更加令人费解：代数效应早已启发了强大的并发库 [32, 42], , 而这些软件实现却尚未在普遍代数和范畴论的意义上与代数效应理论相衔接。（关于相关文献的进一步讨论，请参见 §8 。）

作者联系信息：Ohad Kammar，爱丁堡大学，爱丁堡，英国，ohad.kammar@ed.ac. uk；Jack Liell-Cock，牛津大学，牛津，英国，jack.liell-cock@cs.ox.ac.uk；Sam Lindley，爱丁堡大学，爱丁堡，英国，Sam.Lindley@ed.ac.uk；Cristina Matache，爱丁堡大学，爱丁堡，英国，cristina.matache@ed.ac.uk；Sam Staton，牛津大学，牛津，英国，sam.staton@cs.ox. ac.uk。



本作品根据知识共享署名 4 .0 国际许可协议授权 .

我们工作的关键技巧是认真对待线程 *ID*。这需要一个支持抽象名称或标识符，并支持对它们进行绑定和传递的代数框架。为此，我们采用“参数化代数理论” [44, 45], ，它们与单子和代数效应已有紧密联系。我们的代数理论中有四种运算：

- 分叉：分叉出一个子线程。这是关键运算，写作 fork($a.x(a)$ y)。该运算会生成一个 ID 为 a 的新子线程，运行延续 y；同时在父线程中并发运行延续 x，并将子线程的 ID a 传递给它。

- 等待：在继续执行之前等待线程结束的命令。
- 停止：立即结束当前线程的命令。调用此命令将解除所有等待当前线程的线程的阻塞。

- 动作$_\sigma$：原子基本操作。附带说明：今后我们可以将其与其他代数效应结合，例如通过内存访问来查看并发共享内存；但目前为了专注于并发，我们仅考虑原子基本操作。

贡献。 我们提出了一个关于这四种运算之间八个方程的理论（§4）。我们给出了一个不依赖语法的项模方程表示定理（§5），并证明对于闭项，该表示恰好对应于基于标记偏序集（“并发偏序集”，定理 3.15）的、久已确立的真实并发模型。事实上，这可能是第一个针对标记偏序集的基本语法理论。对于带有自由变量的开放项，我们证明了一个完备性定理：在保持闭项上的带标签偏序集模型的同时，开放项上不可能存在其他方程（§6）。

代数效应使我们能够专注于某个特定理论，而不必操心其他编程语言原语；不过，在分析完代数效应后，通常很容易回到功能更完整的编程语言。在第 2 节中，我们给出一种典型的、带有并发原语和操作语义的函数式编程语言。在第 7 节中，我们利用代数效应和表示定理，为该编程语言构建指称语义，并证明它在一阶层面上是可靠的、适当的且完全抽象的。

1.1 将分叉和等待及线程 ID 作为语言原语的动机

在上一节中，我们引入了运算 fork($a.x(a)$ y)，其中 fork 的类型为 (tid →t) → t → t，且对 t 多态；tid 是线程 ID 类型。使用 fork 这类运算进行编程的方式并不常见，不过根据代数效应理论，代数运算在泛型效应 [35], 中有对应形式，这更符合带有效应的现实语言。‘fork’对应的泛型效应是一条命令：fork : unit → (tid option),

$$\underline{\mathsf{fork}}() = \mathsf{fork}(a.\mathsf{return}\ (\mathsf{Some}\ a), \mathsf{return}\ \mathsf{None}).$$

通过对 fork 的结果进行模式匹配并使用显式顺序执行，可以从命令 $\underline{\mathsf{fork}}$ 中还原出代数运算 fork。

我们在第 2 节给出一种带有操作语义的小型编程语言，它适用于线程池。在该语言中，分叉 会在线程池中生成一个新的子线程，并复制延续。分叉的调用者 分叉 可以通过查看分叉的返回值来判断自己是父线程还是子线程；如果是父线程，返回值将是子线程的标识符，否则就是 分叉，如果调用者是父线程，则会获得子线程的标识符；否则，返回 无。事实上，这种通用运算 分叉 让人想起 POSIX 标准中的 分叉 构造 [1],，在 POSIX 标准中，其类型为 `pid_t fork(void)`，它向父线程返回子线程标识符，向子线程返回 0。

除标准编程原语外，与代数运算 wait、act$_\sigma$、stop 相匹配的其他泛型效应包括：

$$\underline{\mathsf{wait}} : \mathsf{tid} \to \mathsf{unit} \qquad \underline{\mathsf{perform}}_\sigma : \mathsf{unit} \to \mathsf{unit} \qquad \underline{\mathsf{stop}} : \mathsf{unit} \to \mathsf{empty}.$$

其中：wait(a) 会将当前线程置于等待状态，并向调度器记录该线程 a正在等待的线程；perform$_\sigma$() 会立即执行该动作 σ，并在我们的转移系统中通过标签记录；而 stop() 会结束当前线程，使所有正在等待该线程的其他线程解除阻塞。这里，类型 empty 表明在此执行路径上不会再发生其他事情。

我们的操作语义使用标记转换系统来记录所执行的动作。受诸如异步转换系统（例如 [29]）等真并发模型的启发，我们还通过注明执行每个动作的线程标识符来纳入一些位置信息。在这种简单情形下，这足以不仅观察到动作轨迹，还能观察到不同动作之间的独立性。因此，我们可以从操作语义得到一个由动作标记的标记偏序 σ。标记偏序集有时被称为“并发偏序集”，是“真实”并发的另一种模型 [40]。对于我们的语义，偏序集的线性化恰好是程序的执行轨迹。

通过为我们这种特别简单的语言定义“良构配置”，我们可以证明程序永远不会死锁，大致是因为子线程永远不能等待其父线程。我们还证明，每个封闭程序都确定唯一的标记偏序集。这表明我们的语言非常简单：所有程序都会终止，并且不存在事件结构 [30],意义上的“冲突”，也不存在任何“竞争”。这些都是有用的性质，也有助于日后与指称语义建立联系，但它们也意味着，与动态线程在实践中的工作方式相比，我们研究的是一种非常理想化的情形。我们期望未来的工作能扩展到允许递归和冲突的其他原语。

1.2 标记偏序集（并发偏序集）的一个简单完备片段

这个简单的语言使我们能够构造所有标记偏序集；换言之，它完整描述了真实并发模型。为此，我们定义 node$_\sigma$: tidset $\to$ tid（其类型签名暂且非正式地给出）如下：

node$_\sigma$([$a_1, \ldots, a_n$]) 定义为=case fork() of
{Some $b \Rightarrow$返回 b;
None $\Rightarrow$等待(a_1); …; 等待(a_n); 执行$_\sigma$(); case 停止 of {}}。

在包含 node$_\sigma$ 和 stop、但不含分叉与等待的语言片段中，每个线程标识符恰好执行一个动作。因此，所诱导的标记偏序集是线程标识符上的偏序，记录了谁在等待谁，并为每个线程标识符标记其动作。命令 node$_\sigma$([$a_1, \ldots, a_n$]) 会向标记偏序集添加一个标记为 σ 的节点，将其直接前驱设为 a_1、……、 a_n，并返回新节点的名称。

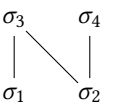


图 1. N形偏序集

例如，以下程序会生成 N形偏序集（图 1）。

let a_1 = 节点$_{\sigma_1}$([]) in let a_2 = 节点$_{\sigma_2}$([]) in let a_3 = 节点$_{\sigma_3}$([a_1, a_2]) in let a_4 = 节点$_{\sigma_4}$([a_2]) in 停止

我们可以用两条公理对标记偏序集进行完全公理化。这里先非正式地使用 ($\vec{\ }$) 和 [ ] 记号来表示线程标识符集合。正式表述见定理3.7 。

$$\text{let } c = \text{node}_{\sigma_1}(\vec{a}) \text{ in let } d = \text{node}_{\sigma_2}(\vec{b}) \text{ in } [c, d] \quad = \quad \text{let } d = \text{node}_{\sigma_2}(\vec{b}) \text{ in let } c = \text{node}_{\sigma_1}(\vec{a}) \text{ in } [c, d]$$

$$\text{let } b = \text{node}_{\sigma}(\vec{a}) \text{ in } [b] \quad = \quad \text{let } b = \text{node}_{\sigma}(\vec{a}) \text{ in } [b] +\!\!+ \vec{a}$$

第一条定律指出，只要遵循 ID 依赖关系，以何种顺序添加节点并不重要；第二条定律则刻画了偏序的传递性。我们在定理 3.15 中证明，这两条公理是完备的，并采用参数化代数

理论。关键在于，通过传递线程 ID，我们能够完整描述基于标记偏序集的既有真实并发模型。

1.3 技术设定：带绑定语法与带名称语义的函子范畴和自然性

为应对动态线程和线程名称数量的变化，我们沿袭了使用函子范畴的悠久传统。简言之，令 **FinRel** 表示由有限名称集及其间关系构成的范畴，令 **Set** 表示由所有集合及函数构成的范畴。那么，类型 A 处的计算构成一个函子 $[\![A]\!] : \mathbf{FinRel} \to \mathbf{Set}$，它将可用线程 ID 的集合 w 映射到至多使用这些线程 ID 的计算集合 $[\![A]\!](w)$。

根据这里的函子作用，对于每个关系 $R \subseteq w \times w'$，我们都有一个重索引函数 $[\![A]\!](w) \to [\![A]\!](w')$。其思路是：如果世界 w 中的某个计算会等待某个线程标识符 $a \in w$，那么我们可以将其转换为等待直接像中所有线程标识符的计算，即 $\{b \mid R(a,b)\}$。

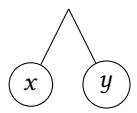
类型为 $A \to B$ 的程序被解释为函数族 $[\![A]\!](w) \to [\![B]\!](w)$，并且这些函数族是自然的。特别是，这保持了这样一个不变性：不能对线程标识符求和、猜测线程标识符，也不能按某种顺序比较它们。这类似于名义集合中名称的作用 [33]。

参数化代数理论框架是函子范畴上处理代数效应的一种成熟方法，并且具有具体语法，我们将其用于公理化。此外，每个参数化代数理论都会在函子范畴上诱导出一个强单子；在我们的情形中，该范畴是 [**FinRel**, **Set**]。函子范畴上的单子长期以来一直用于动态分配的指称语义 [31, 38]。在第 7.1 节中，我们使用由动态线程理论诱导出的强单子，为第 节的小型编程语言提供 Moggi 风格的指称语义 2。

1.4 一般情形下的分叉与等待、并行组合，以及带孔的标记偏序集

尽管节点$_\sigma$ 效应足以构造所有标记偏序集，但通过分叉与等待实现更高层次的并行，更能体现其典型用法。例如，我们可以定义一个程序 parallel : ((unit → 空类型)、(unit → 空类型)) → 空类型，使两个其他程序并行运行：只需生成两个线程并等待它们：

$$
\begin{aligned}
\mathsf{parallel}(x,y) = \mathsf{case}\,(\underline{\mathsf{fork}}())\,\mathsf{of}\{&\\
&\mathsf{Some}\,a \Rightarrow \mathsf{case}\,(\underline{\mathsf{fork}}())\,\mathsf{of}\,\{\\
&\qquad\mathsf{Some}\,b \Rightarrow \underline{\mathsf{wait}}(a); \underline{\mathsf{wait}}(b); \underline{\mathsf{stop}}()\\
&\qquad\mathsf{None} \quad\Rightarrow y()\}\\
&\mathsf{None} \quad\Rightarrow x()\}
\end{aligned}
$$



因此，我们在第 4.1 节为分叉与等待提供了一套等式理论。基本思路是，分叉$(a.t, u)$ 的行为类似于一个幺半群，而 等待(a)；停止() 则类似于单位元，不过需要谨慎处理线程标识符参数。

本文的**主要结果**是 fork/wait 理论的表示定理（定理 5.4）。这种表示沿用了标记偏序集的思路，不过现在可以用孔表示不同的续延。例如，‘并行组合’运算变成了图中所示的‘樱桃’图式。这并非易事，因为插入到 x 或 y 中的任何线程，都可能包含未被等待的子线程，也可能等待图中未出现的其他线程标识符。

我们还证明了一个完备性定理（定理6.1），该定理指出，如果无论将什么代入变量 x，y 等，两个表达式都产生相同的标记偏序集，那么这两个表达式就可以被证明相等。我们通过找出用于代入变量的特殊小工具来证明这一点。这可以视为一个完全抽象结果，我们将在定理 7.4 中说明这种联系。

最后补充一点，我们定义一个运算 series : ((单位元 → 空类型)，(单位元 → 空类型)) → 空类型，它会分叉出一个子线程并立即等待它：

$$\mathsf{series}(x, y) = \mathsf{case}\ \underline{\mathsf{fork}}() \ \mathsf{of}\ \{\mathsf{Some}\ a \Rightarrow \underline{\mathsf{wait}}(a); y() \mid \mathsf{None} \Rightarrow x()\}$$

我们可以用“串行”和“并行”来构造串并联图 [3, 25]，，并且很容易从我们的代数理论推导出串并联图的等式定律。但请注意，我们也可以表示 N 形，它并非串并联图。

虽然“串行”很容易编程，但它与编程语言中的顺序执行 $x(); y()$ 并不相同；“串行”要求 x 和 y 的返回类型相同。另一种理解是，(;) 的单位元是 (return ()) （返回一个值），而串行的单位元是 stop（结束当前线程）。特别是，$\mathsf{stop}(); x()$ 根本不会执行 x。还请注意，如果子线程在从未调用 stop 的情况下返回，那么等待该子线程的线程将永远不会解除阻塞。

“串行”与顺序执行表面上的相似性，可能是早先有人声称通过代数效应实现并发存在“非预期方程” [49, §8,p33]，（例如并行组合与顺序执行可交换）的原因。通过转而聚焦于分叉、等待和动态线程，我们清楚地展示了顺序执行与串行之间的区别：尽管“并行”与顺序执行可交换，但它并不像预期的那样与“串行”可交换。

论文摘要。操作语义和指称语义分别见第 2 节和第 7节；主要的编程语言结果是可靠性、充分性和完全抽象（§7.2）。构建指称语义的方法基于代数效应，在第 3–4节中阐述，并证明其具有一种表示（§5），由此证明其完备（§6）。

2 背景：并发编程语言

为准确说明这一情况，我们将讨论一种相当标准的并发编程语言及其操作语义。

2.1 语言与类型系统

我们考虑一种标准的高阶编程语言，它具有有限积类型和和类型（例如 [23, 28]）。类型的语法如下：

$$A, B ::= \mathsf{tid} \mid \prod_{i=1}^{k} A_i \mid \sum_{i=1}^{k} A_i \mid A \to B$$

当 $k = 0$ 时，我们得到空积和空和类型，分别记作 1 和 0，而非引言中的单位元和空类型。线程 ID 的基类型是针对我们的设定专门定义的。

该语言采用细粒度按值调用 [23]，，这意味着项分为值和计算。值是不会进行 β 归约且不会产生任何效应的项：

$$v ::= x \mid (v_1, \ldots, v_k) \mid \mathsf{inj}_i\, v \mid \lambda x.\, t \mid a \mid \mathtt{0} \mid v_1 \oplus v_2 \mid g$$

值包括变量、常见的积类型和和类型构造子，以及函数；函数体 t 是一个计算。符号 a 的取值范围是一个可数无限的实际线程标识集合 $\mathbb{T}$。常量 0 表示空线程 ID 集合，而 $\oplus$ 表示两个线程 ID 集合的并集。符号 g 的取值范围是一个固定的带类型项常量集合 $\mathbb{F}$：$A \to B$。这些常量使我们能够为语言添加并发特性。

计算的语法包含积类型、和类型及函数类型的常用析构子，以及用于显式顺序执行计算的 let构造：

$$t ::= \mathsf{return}\, v \mid \mathsf{proj}_i\, v \mid \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inj}_i(x_i) \Rightarrow t_i\}_{i=1}^{k} \mid v_1\, v_2 \mid \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$$

我们的语言包含以下值常量集合 $\mathbb{F}\ g\colon\ A \to B$，其中 σ 遍历一个固定的可观测动作集合 Σ:

$$\underline{\mathsf{fork}} : 1 \to \mathsf{tid} + 1 \qquad \underline{\mathsf{stop}} : 1 \to 0 \qquad \underline{\mathsf{wait}} : \mathsf{tid} \to 1 \qquad \underline{\mathsf{perform}}_\sigma : 1 \to 1 \tag{1}$$

直观来说，fork() 会分叉出一个新的子线程，并在和类型 tid + 1 的左分支中将其标识符返回给父线程。右分支供子线程使用，它会接收单位值 ()。fork() 的延续会运行两次：一次在父线程中，一次在子线程中。

计算 stop() 表示当前线程已完成，其延续将被丢弃。一旦线程调用了 stop()，就无法恢复运行。计算 wait(v) 会等待标识符属于 v 的所有线程调用 stop() 并结束，然后返回单位值。最后，perform$_\sigma$() 会执行可观测动作 σ，然后返回单位值。

编写程序时，我们会使用一些语法糖，例如用 $(t_1; t_2)$ 表示 let $x = t_1$ in t_2，其中 t_2 不依赖于 x；在含义明确时，也会用 case t of … 代替 let $x = t$ in case v of …。

例 *2.1*。考虑以下计算：

$$\begin{aligned}&\mathsf{let}\, y = \underline{\mathsf{fork}}() \,\mathsf{in}\, \mathsf{case}\ y \,\mathsf{of}\, \{\mathsf{inj}_1(x_1) \Rightarrow \underline{等待}(x_1); \underline{执行}_{\sigma_1}(); \underline{\mathsf{stop}}(),\ \mathsf{inj}_2() \Rightarrow 执行_{\sigma_2}(); \mathsf{stop}()\}\ \underline{\mathsf{let}}\\ &y = \underline{\mathsf{fork}}() \,\mathsf{in}\, \mathsf{case}\ y \,\mathsf{of}\, \{\mathsf{inj}_1(x_1) \Rightarrow 执行_{\sigma_1}(); 等待(x_1); \mathsf{stop}(),\ \mathsf{inj}_2() \Rightarrow 执行_{\sigma_2}(); \mathsf{stop}()\}\end{aligned}$$

两种情况下，主线程都会分叉出一个新的子线程，该子线程执行动作 σ_2 后停止；这个新线程的标识符绑定到 x_1。在第一种情况下，主线程会等待子线程结束后再执行动作 σ_1。因此，观察到的动作序列是 σ_2，随后是 σ_1。在第二种情况下，主线程不会等待，因此我们也可能看到相反的顺序，或者 σ_1 和 σ_2 同时发生。

值和计算分别有独立的类型判断 $\vdash^{\mathrm{v}}_w$ 和 $\vdash^{\mathrm{c}}_w$。标准判断见图2。所有判断都标注了一个有限的线程 ID 子集 $w \subseteq \mathbb{T}$，称为世界；在一次类型推导过程中，世界保持不变。世界 w 表示由环境创建的线程。项可以通过以下类型规则使用世界中的线程标识符：

$$\frac{a \in w}{\Gamma \vdash^{\mathrm{v}}_w a : \mathsf{tid}}$$

对于程序的等式推理，将线程标识符组合成复合线程标识符非常有帮助。例如，假设线程 a 已无事可做，但正在等待 b 和 c完成后才能结束。那么线程 a 就相当多余了，保留它的唯一原因是父线程可能会在某个时候等待 a。如果父线程可以改为同时等待 b 和 c，那么我们确实可以立即结束 a。（这是公理（17）的一个实例。）为了对此例进行推理，使用复合线程标识符$(b \oplus c)$ 来替代 a很有帮助。为便于进行本文旨在探讨的这种等式推理，我们给出以下复合线程标识符的构造；在操作语义中，它们被视为标识符集合：

$$\frac{}{\Gamma \vdash^{\mathrm{v}}_w 0 : \mathsf{tid}} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v_1 : \mathsf{tid} \qquad \Gamma \vdash^{\mathrm{v}}_w v_2 : \mathsf{tid}}{\Gamma \vdash^{\mathrm{v}}_w v_1 \oplus v_2 : \mathsf{tid}}$$

2.2 操作语义

我们现在为该语言定义一种操作语义，它基于格局上的标记转移关系。这些格局是线程池，其中一些线程已准备好运行，一些已完成，还有一些则卡住，必须等待其他线程完成之后才能运行。我们还包含一个关系，用以说明哪些线程正在等待哪些其他线程完成。

$$\frac{}{\Gamma, x : A, \Gamma' \vdash^{\mathrm{v}}_w x : A} \qquad \frac{\left(\Gamma \vdash^{\mathrm{v}}_w v_i : A_i\right)_{i=1}^{k}}{\Gamma \vdash^{\mathrm{v}}_w (v_1, \ldots, v_k) : \prod_{i=1}^{k} A_i} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v_i : A_i}{\Gamma \vdash^{\mathrm{v}}_w \mathsf{inj}_i\, v_i : \sum_{i=1}^{k} A_i} \qquad \frac{\Gamma, x : A \vdash^{\mathrm{c}}_w t : B}{\Gamma \vdash^{\mathrm{v}}_w \lambda x.\, t : A \to B}$$

$$\frac{\Gamma \vdash^{\mathrm{v}}_w v : A}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{return}\, v : A} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v : \prod_{i=1}^{k} A_i}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{proj}_i\, v : A_i} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v : \sum_{i=1}^{k} A_i \qquad \left(\Gamma, x_i : A_i \vdash^{\mathrm{c}}_w t_i : B\right)_{i=1}^{k}}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inj}_i(x_i) \Rightarrow t_i\}_{i=1}^{k} : B}$$

$$\frac{\Gamma \vdash^{\mathrm{v}}_w v_1 : A \to B \qquad \Gamma \vdash^{\mathrm{v}}_w v_2 : A}{\Gamma \vdash^{\mathrm{c}}_w v_1\, v_2 : B} \qquad \frac{\Gamma \vdash^{\mathrm{c}}_w t_1 : A \qquad \Gamma, x : A \vdash^{\mathrm{c}}_w t_2 : B}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2 : B} \qquad \frac{(g : A \to B) \in \mathbb{F}}{\Gamma \vdash^{\mathrm{v}}_w g : A \to B}$$

图 2. Standard类型规则，适用于细粒度按值调用编程语言。此处 $\mathbb{F}$ 给出于 (1).

2.2.1 另一种语言构造：act_σ。为了建立操作语义，考虑以下运算会比较方便

$$\mathrm{act}_\sigma : 1 \to 0$$

它执行动作 σ，然后立即结束。它与 $\mathrm{perform}_\sigma$：$1 \to 1$可以相互定义：

$\mathrm{act}_\sigma() = \mathrm{perform}_\sigma(); \mathrm{stop}()$，而 $\mathrm{perform}_\sigma() = \mathsf{let}\, x = \mathsf{fork}()\, \mathsf{in}\, \mathsf{case}\ x$ of $\{\mathsf{inj}_1(a) \Rightarrow \mathrm{wait}(a),\ \mathsf{inj}_2() \Rightarrow \mathrm{act}_\sigma()\}$。

也就是说，对于一个动作，若其后还可以执行其他命令，则可以通过分叉一个仅执行该动作的新线程来实现，然后等待该线程结束。

可以说，在编程中，$\mathrm{perform}_\sigma$ 更自然，但 act_σ 的语义更简单。我们关注语义，因此将 act_σ 视为原语，并将 $\mathrm{perform}_\sigma$ 视为派生形式。

2.2.2 配置。

定义 *2.2.* 一个 配置是一个三元组 $\langle w; \prec; \text{线程} \rangle$ 其中

- $w \subseteq \mathbb{T}$ 是参与此配置的线程 ID 的有限集合。
- $(\prec) \subseteq \mathbb{T} \times w$ 是一个关系，它将 w 中的线程 ID 与它们正在等待的、可能位于外部的 $\mathbb{T}$ 中的 ID 关联起来。
- 线程：$w \to \{\text{computations } t\} \uplus$ 已完成 是一个函数，它为每个活动线程标识符指定其运行的计算；若线程已完成，则指定为“已完成”。我们经常会枚举该映射，例如写作 $([a]t, [b]u)$，其中 $a \mapsto t$ 且 $b \mapsto u$。

我们将 在线程只有一个时，将该配置简写为 $[a]t$，而不写作 $\{a\}; \emptyset; [a]t$.

2.2.3 转换关系。转换系统通过图 3 中的归纳定义给出。我们定义配置之间的两种转换关系：静默归约 $\longrightarrow$ 和带标签归约 $\xrightarrow{\sigma}$，其中 σ 是一个动作。我们还用发生归约的线程（a）标注转换关系，沿用 [29]的做法；这并非必要，可以省略，但对元理论很有用。

关系 $a \prec b$ 表示线程 b 正在等待线程 a 完成。图 3 中的最后一条转换规则指出，只有当线程所等待的所有线程都已完成时，该线程才能执行一步。执行一步后，需要更新等待关系 $(\prec)$，以纳入任何新的等待项 $(\prec')$。

其他转换规则针对单个线程的归约。等待操作通过记录其等待的对象来归约。分叉会生成一个新的子线程 b，并将其标识符 b 传递给父线程 a。

在这种简单语言中，只有一个求值上下文，即 $\mathsf{let}\ x = [-]\, \mathsf{in}\, s$。要在此处进行求值，取决于上下文中的表达式（$t$）会归约、生成或完成哪些线程。我们

主要语言构造的归约规则。

$$[a]\underline{\text{等待}}(v) \longrightarrow_a \{a\};\ \{b \prec a \mid b \in \text{线程标识符}(v)\};\ [a]\text{返回 }()$$

$$[a]\underline{\text{fork}}() \longrightarrow_a \{a,b\};\ \emptyset;\ [a]\text{返回 }(\text{inj}_1\,(b))\ \ [b]\text{返回 }(\text{inj}_2\,()) \qquad (a \neq b)$$

$$\langle [a]\underline{\text{停止}}()\rangle \longrightarrow_a \langle [a]\text{已完成}\rangle$$

$$[a]\underline{\text{act}}_\sigma() \overset{\sigma}{\longrightarrow}_a \langle [a]\text{已完成}\rangle$$

积、和类型、函数和 let 绑定的标准归约规则。

$$\langle [a]\text{proj}_i\,(v_1,\dots,v_k)\rangle \longrightarrow_a \langle [a]\text{return}\,v_i\rangle$$

$$\langle [a]\text{case inj}_i\,(v)\text{ of }\{\text{inj}_k(x_k) \Rightarrow t_k\}_{k=1}^{l}\rangle \longrightarrow_a \langle [a]t_i[v/x_i]\rangle$$

$$\langle [a](\lambda x.t)\,v\rangle \longrightarrow_a \langle [a]t[v/x]\rangle$$

$$\langle [a]\text{let } x = \text{return } v \text{ in } t\rangle \longrightarrow_a \langle [a]t[v/x]\rangle$$

在求值上下文中进行归约：

$$\frac{\langle [a]t\rangle \overset{(\sigma)}{\longrightarrow}_a \langle w;\prec;\mathit{thread}\rangle}{\langle [a]\text{let } x = t \text{ in } s\rangle \overset{(\sigma)}{\longrightarrow}_a \left\langle w;\prec;\ \begin{array}{l}\{[b]\text{let } x = t' \text{ in } s \mid b \in w, \mathit{thread}(b) = t'\}\cup\\ \{[b]\mathit{finished} \mid b \in w, \mathit{thread}(b) = \mathit{finished}\}\end{array}\right\rangle}$$

将配置上的线程局部转移转换为全局转换：

$$\frac{[a]t \overset{(\sigma)}{\longrightarrow}_a \langle w;\prec';\text{线程}\rangle \qquad \forall b.\ b \prec a \implies \text{线程}_0(b) = \text{已完成}}{w_0 \uplus \{a\};\ \prec;\ \text{线程}_0 \uplus \{[a]t\} \overset{(\sigma)}{\longrightarrow}_a \langle w_0 \uplus w;(\prec \cup \prec' \cup \{(b,c) \mid b \prec a, c \in w\})^*;\text{线程}_0 \uplus \text{线程}\rangle}$$

其中 $(-)^*$ 表示传递闭包。

图 3. 我们的并发编程语言的操作语义（第 2.2）。我们用括号$\overset{(\sigma)}{\longrightarrow}$表示该规则有两个副本，一个带标签，另一个不带标签。这里，线程标识符$(a) = \{a\}$,线程标识符$(v \oplus v') =$ 线程标识符$(v) \cup$线程标识符(v')，且线程标识符$(0) = \emptyset$。

随后继续使用来自 w的所有线程对 let 表达式求值，每个线程都使用自己的延续副本 s继续执行；任何已完成的线程都会在不对 s求值的情况下结束。

$\prec$关系有几点微妙之处。首先，配置 $w;\ \prec$;线程 可能具有 $b \prec a$，其中某些 b不在 w中。因此，线程可能会等待当前线程池中不存在的线程 ID。这样做是为了允许我们将视角限制在特定线程上，但在顶层不会发生这种情况。

其次，即使 a和 b都已完成，配置也可能仍有 $a \prec b$。可以将这些冗余信息进行垃圾回收，就像任何高效的实现都会做的那样；但这并非必要，而且保留这些信息会让元理论更简单。

2.2.4 作为带标签偏序集的观测。 我们关注真并发语义，因此，除了考察线性轨迹之外，还纳入依赖序 $\prec$。由此得到一个标记偏序集（即部分有序多重集 [36, 40], , 或等价地，无冲突事件结构 [30]）。

定义 *2.3*。 设 Σ 为一个集合。 Σ-标记偏序集 是一个偏序集 $P = (X, \leq)$，并配备一个函数 $\ell : X \to \Sigma$。（在上下文明确时，我们省略 Σ。）

定义 *2.4*. $w;\ \prec$;线程是一个终止配置，其中所有线程均已完成：线程(a) = 已完成 对所有 $a \in w$ 均成立。

设 $(X,\ \leq,\ \ell)$ 是一个标记偏序集，且 C 是一个配置。我们记

$$C \Downarrow (X, \leq, \ell)$$

当存在某个终止配置 $C' = w; \prec$;线程，使得 C 允许一系列转移时

$$C \longrightarrow^* \xrightarrow{\sigma_1}_{a_1} \longrightarrow^* \xrightarrow{\sigma_2}_{a_2} \cdots \longrightarrow^* \xrightarrow{\sigma_n}_{a_n} \longrightarrow^* C',$$

并且满足关于 $(X,\ \leq,\ \ell)$ 的以下条件：$X = \{a_1,\ \ldots\ldots,\ a_n\}$，$X$ 上的序定义为：$a_i \leq a_j$ 当且仅当 $a_i \prec a_j$ 或 $i = j$，且 $\ell(a_i) = \sigma_i$。

回顾图 3 中的归约规则可知，每个动作 σ_i 都发生在线程中，并且线程会立即结束。因此，集合 X由所有不同的标识符组成，$a_1,\ \ldots\ldots,\ a_n$，它们对应执行动作的线程；ℓ 指定每个动作的内容，而 $\leq$ 则编码动作之间的因果依赖关系。在例 2.1中，第一个程序通过 $\Downarrow$ 与序 $(\sigma_2 < \sigma_1)$相关联，而第二个程序则与离散序 $\{\sigma_1,\ \sigma_2\}$相关联。参见 [36]，进一步讨论这些并发观测概念。

2.3 操作元理论

类型良好的程序永远不会发生死锁，而且会唯一确定一个观测到的标记偏序集。更复杂的语言不具备这些性质，但在这个简单的设置中，它们有助于与第7节的指称语义精确对应。

2.3.1 良构配置。类型良好的程序永远不会发生死锁，也就是说，不会有两个线程互相等待。为此，我们考虑存在一种线性序的良构配置，该线性序编码了线程的潜在创建顺序。其思路是，配置看起来就像最大的线程是一个父线程，它自身分叉出了配置中的所有其他线程；最小的线程是最先分叉出的子线程，接着是第二个子线程，依此类推。子线程只能引用更早分叉出的兄弟线程，因此这些线程在线性序中更小；父线程则可以引用任意子线程。请注意，线程可能并非按此线性序（或任何其他线性序）创建，而且父子关系也可能更为复杂。操作语义不依赖创建顺序，并且对于给定配置，可能存在多个彼此一致的线性序。

定义 *2.5*。考虑一个配置 $C = w; \prec$; 线程。考虑 w 上的一个线性序 $<$，以及一个类型 A。设 $w' \subseteq \mathbb{T}$ 与 w 不相交。（其思想是，$<$ 是创建顺序，而 w'描述当前线程池中不存在的一些线程；当我们聚焦于单个线程时，这可能会有用。）我们称 C 对于（$A,\ <,\ w'$）是良构的，当且仅当

- 等待序 $\prec$ 是传递的。
- 线程只等待池中或 w'中的线程：若 $a \prec b$，则 $a \in w \uplus w'$。
- 线程只等待较早创建的兄弟线程：若 $a \prec b$ 且 $a,\ b \in w$，则 $a < b$。
- 所有尚未完成的线程都具有类型 A，并且只依赖于先前创建的兄弟线程：对于所有 $a \in w$，都有 $\vdash_{\mathsf{c}} \{b<a\} \uplus w'$ 线程(a)：A；（父线程，即 $<$中最大的线程，可以依赖并等待其所有子线程）。

命题 2.6（保持性）。设 $C_1 = \langle w_1; \prec_1; \text{线程}_1 \rangle$ 和 $C_2 = \langle w_2; \prec_2; \text{线程}_2 \rangle$。若 C_1 对$(A, <_1, w')$及 $C_1 \xrightarrow{(\sigma)} C_2$ 是良构的，且 w' 与 w_2 不相交，则存在一个扩展 $<_1$ 的线性序 $<_2$，使得 C_2 对$(A, <_2, w')$是良构的。

证明笔记。对转移的推导进行归纳。 □

2.3.2 由空类型的项唯一确定的标记偏序集。

命题 *2.7*。考虑空世界 $\emptyset$ 中的空类型项 $\vdash^{c}_{\emptyset} t : 0$。*(1)* 配置 $\langle [a]t \rangle$ 可以到达一个终止配置，即存在一个标记偏序集 (P, ℓ)，使得 $\langle [a]t \rangle \Downarrow (P, \ell)$。*(2)* 若 $\langle [a]t \rangle \Downarrow (P_1, \ell_1)$ 且 $\langle [a]t \rangle \Downarrow (P_2, \ell_2)$，则这两个标记偏序集同构，即存在一个序同构 $f : P_1 \cong P_2$，使得 $\ell_1(e) = \ell_2(f(e))$。

证明概要。第 1 部分在更一般的情况下也成立：从一个类型良好的项开始的每个归约序列 $\vdash^{c}_{\emptyset} t : A$ 都是有限的。我们的证明直接结合了 Tait 方法 [47]和 Kőnig 树引理 [22]。程序的每个归约序列都会导出一棵有限分支树，其中每条分支对应于单个线程的顺序执行，且该线程不会提及其他线程。这些线程局部执行包括这样的转移步骤：环境将一个已知线程 ID 的状态改为已完成。每个无限归约序列都会导出这样一棵无限树，而 Kőnig 树引理蕴含它有一条无限分支。接着，我们运用 Tait 方法，即设计一种恰当的 Kripke 逻辑关系，证明在所有类型良好的程序中，每个线程都只有有限的顺序执行。该关系的 Kripke 性质是相对于线程标识符的单射重标记而言的。我们定义两种“值”关系：一种以闭值上的类型为索引，另一种以闭环境上的上下文为索引。以类型为索引、定义在闭计算上的计算关系表明，该计算不存在无限归约序列；并且每当该计算求值为返回 v 时，值 v 满足相应的值关系。随后，我们证明这些关系的基本性质：每个类型良好的值、计算和代换，都会将满足值关系的闭环境映射为满足该关系的值、计算和闭环境。

对于第 2 部分，我们证明一个合流性结果。首先，我们为 fork() 引入的新线程 ID 选定一个确定性命名方案，使新线程 ID 不受求值顺序影响。一种合适的方案（例如 [29]）是将线程 ID 定义为数字组成的有限序列，其含义是，ID $(m_1 m_2 m_3 \dots m_k m_{k+1})$ 表示线程 $(m_1 \dots m_k)$ 直接派生的第 m_{k+1} 个线程。接下来我们证明

(1) If $C \xrightarrow{(\sigma)}_a C_1$ and $C \xrightarrow{(\sigma)}_a C_2$ then $C_1 = C_2$; and
(2) If $C \xrightarrow{(\sigma)}_a C_1$ and $C \xrightarrow{(\tau)}_b C_2$ then there is C' such that $C_1 \xrightarrow{(\tau)}_b C'$ and $C_2 \xrightarrow{(\sigma)}_a C'$.

第一点是每个线程内部的局部确定性，这可以通过对转移推导的结构进行归纳轻松证明。第二点也通过对转移推导的结构进行归纳来证明。不过需要特别注意，对于某个可执行线程中的一步，局部到全局规则中的传递闭包不会引入依赖关系，导致另一个当前可执行的线程不得不等待。这里，对归纳假设作出的关键加强是证明

如果 $\langle w;\ \prec;\ thread \rangle \xrightarrow{(\sigma)}_a \langle w';\ \prec';\ thread' \rangle$ and $c \prec' b$ and $b \in w$ then either $c \prec b$ or $a = b$ or $a \prec b$

□

3 参数化代数理论：通过一种新的带标记偏序集理论加以说明

代数效应通过普遍代数中的代数理论形式化。本节回顾代数理论及其推广——参数化代数理论的概念，并介绍一个新颖的贯穿示例：带标记偏序集理论。

3.1 代数理论

定义 *3.1*。一个（一阶有限元）代数签名$\mathrm{O} = |\mathrm{O}|$，ar由运算$|\mathrm{O}|$ 以及函数 $\mathrm{ar} : |\mathrm{O}| \to \mathbb{N}$构成，该函数为每个运算关联一个自然数，称为其元数。

我们将元数为 n 的运算 O 写作 $\mathrm{O} : n$。上下文 $\Delta = a_1, \ldots, a_n$是由互不相同的变量组成的列表。上下文 Δ 中的项 归纳定义如下：

$$\frac{}{\Delta, a, \Delta' \vdash a} \qquad \frac{\Delta \vdash u_1 \quad \cdots \quad \Delta \vdash u_n \qquad \mathrm{O} : n}{\Delta \vdash \mathrm{O}(u_1, \ldots, u_n)}$$

定义 *3.2*。一个（一阶有限元数）代数理论 $\mathrm{T} = (\mathrm{O}\ E)$ 是由代数签名 O 和一组方程 E构成的二元组，其中一个方程是同一上下文中的两个项组成的有序对，$\Delta \vdash t_1$ 和 $\Delta \vdash t_2$，记作 $\Delta \vdash t_1 = t_2$。

例 *3.3*。半格的代数理论 L 有两个运算：$\oplus : 2$ 和 $0 : 0$。方程规定 $\oplus$ 满足结合律、对称律和幂等律，且 0 是其单位元：

$$a, b, c \vdash (a \oplus b) \oplus c = a \oplus (b \oplus c) \qquad a, b \vdash a \oplus b = b \oplus a \qquad a \vdash a \oplus a = a \qquad a \vdash a \oplus 0 = a$$

半格理论常用作带失败的非确定性选择的语义 [28]。上下文 Δ 中的项模方程对应于来自 Δ 的变量子集。

3.2 参数化代数理论

参数化代数理论是普通代数理论的扩展，允许绑定抽象参数。它们已用于公理化涉及某种资源的效应，例如局部状态中的新内存位置，以及 π-演算中的通道 [45]。

参数化代数理论是相对于某个普通代数理论参数化的，其含义见定理 3.2。在本文余下部分中，我们固定该普通代数理论为定理 3.3中的半格理论。我们回顾参数化代数理论的定义，并贯穿介绍一个新颖的带标记偏序集理论示例。

定义 *3.4*。一个参数化签名 $\mathrm{O} = |\mathrm{O}|$,ar 是一组运算 $|\mathrm{O}|$ 以及函数 $\mathrm{ar} : |\mathrm{O}| \to \mathbb{N} \times \mathbb{N}^*$，该函数为每个运算 O 指定一个元数，由一个自然数和一个自然数列表组成：$\mathrm{ar}(\mathrm{O}) = (p \mid m_1, \ldots, m_k)$。这意味着 O 接受 p个参数和 k 个延续，并在第 i 个延续中绑定 m_i 个参数。

例*3.5.* 考虑运算 $\mathrm{node}_\sigma : (1 \mid 1)$ 和 $\mathrm{end} : (0 \mid)$，其中 σ 取自一个固定的可观测动作集 Σ。node_σ 运算接受一个自由参数和一个绑定一个参数变量的延续；end 不接受参数，也不带续延。参数表示半格理论中的一个项。

参数化上下文 $\Gamma \mid \Delta$ 由两个部分组成：Δ 是一个参数变量列表，也就是底层参数化理论中的普通代数上下文；在我们的例子中，该理论是半格理论，参数是线程 ID。部分 Γ 是一个由不同计算变量组成的列表 $\Gamma = x_1 : m_1, \ldots, x_k : m_k$，其中每个 x_i都标注了它所使用的参数数量 m_i；x_i 是一个可以代入参数化理论中的项的变量。我们也常将 x_i 称为延续。上下文 $\Gamma \mid \Delta$ 中的项由以下规则归纳生成：

$$\frac{(\Delta \vdash u_i)_{i=1}^{m}}{\Gamma, x : m, \Gamma' \mid \Delta \vdash x(u_1, \ldots, u_m)} \qquad \frac{(\Delta \vdash u_i)_{i=1}^{p} \qquad \left(\Gamma \mid \Delta, b_1, \ldots, b_{m_i} \vdash t_i\right)_{i=1}^{k} \qquad \mathrm{O} : (p \mid m_1, \ldots, m_k)}{\Gamma \mid \Delta \vdash \mathrm{O}(u_1, \ldots, u_p,\ b_1 \ldots b_{m_1}.t_1,\ \ldots,\ b_1 \ldots b_{m_k}.t_k)}$$

其中 $\Delta \vdash u_i$是定理3.3中的普通半格代数理论里的项判断。两种上下文都允许所有常见的结构规则，并且我们将变量重命名后的所有项视为相同。

使用定理 3.5 中的签名，我们可以在上下文中构造以下项：

$$x : 1 \mid a \vdash \mathsf{node}_\sigma(a, b.x(b)) \tag{2}$$

$$x : 2 \mid a_1, a_2 \vdash \mathsf{node}_\sigma(a_1 \oplus a_2, b_1.\mathsf{node}_\tau(a_1, b_2.x(b_2, b_1))) \tag{3}$$

在项 (2) 中， a 是自由参数，而 b 在 x中是绑定参数。从并发的角度来看，我们将 节点$_\sigma(a, b.x(b))$解释为分叉出一个新的子线程，该子线程在标识符为 a的线程执行其动作之后，执行动作 σ。执行 σ的子线程的线程标识符是 b，而延续$x(b)$ 则并发执行。

代数运算节点$_\sigma$ 是泛型效应 节点$_\sigma$ 的对应形式，后者在第 1.2 节中讨论。我们可以用如下方式将两者相互编码：

$$\underline{\mathsf{node}}_\sigma(a) \;=\; \mathsf{node}_\sigma(a, b.\mathsf{return}\, b)$$
$$\mathsf{node}_\sigma(a, b.t) \;=\; \mathsf{let}\, b = \underline{\mathsf{node}}_\sigma(a)\, \mathsf{in}\, t$$

其中， a 现在表示一组线程 ID，这得益于我们使用了半格理论，因此 node$_\sigma$ 的类型是 tid $\to$ tid。

项 (3) 使用半格理论中的运算 $\oplus$ 来等待两个线程标识符 a_1和 a_2，然后再执行 σ。图 4 (c) 展示了项 (3) 的图形表示，其中 a_1 和a_2 是底部的输入，而 $x : 2$ 所接收的两个参数是顶部的输出。绑定参数的名称 b_1 和 b_2 不会出现。实线表示因果依赖。使用 node 和 end 构造的项至多包含一个计算变量，因此图形表示中不会记录该变量的名称。

我们在项上定义两种替换运算，一种用于参数变量，一种用于计算变量，采用标准的避免捕获方式，从而允许以下规则：

$$\frac{\Gamma \mid \Delta, a \vdash t \qquad \Delta \vdash u}{\Gamma \mid \Delta \vdash t[u/a]} \qquad\qquad \frac{\Gamma, x : m \mid \Delta \vdash t \qquad \Gamma \mid \Delta, b_1, \dots, b_m \vdash s}{\Gamma \mid \Delta \vdash t[b_1 \dots b_m.s/x]} \tag{4}$$

记号 $b_1 \dots b_m.s/x$强调， s中的绑定参数 b_1，…， b_m将被替换为传递给 x的参数。

以下是每种代换的示例。它们可以从图形上理解：第一种将图 4 （b）变换为图 4 （c），第二种将图4 （c）变换为图4 （d）。

$$\text{节点}_\sigma(a_3, b_1.\text{节点}_\tau(a_1, b_2.x(b_2, b_1)))\,[a_1 \oplus a_2/a_3] = \text{节点}_\sigma(a_1 \oplus a_2, b_1.\text{节点}_\tau(a_1, b_2.x(b_2, b_1))) \tag{5}$$

$$\text{节点}_\sigma(a_1 \oplus a_2, c_1.\text{节点}_\tau(a_1, c_2.x(c_2, c_1)))\,[b_1 b_2.y(b_1 \oplus b_2)/x] = \text{节点}_\sigma(a_1 \oplus a_2, c_1.\text{节点}_\tau(a_1, c_2.y(c_2 \oplus c_1))) \tag{6}$$

定义 *3.6.* A 参数化代数理论 T $= (\mathsf{O}\ E)$ 是由参数化签名 O 和方程集 E 组成的一对。方程是同一上下文 $\Gamma \mid \Delta$中的两个项组成的一对，记作 $\Gamma \mid \Delta \vdash t_1 = t_2$。

例 *3.7.* 带标号偏序集理论 C由定理 3.5中的签名构成，其中包含 节点$_\sigma$ 和 end，以及以下两个方程：

$$x : 2 \mid a_1, a_2 \vdash \mathsf{node}_\sigma(a_1, b_1.\mathsf{node}_\tau(a_2, b_2.x(b_1, b_2))) = \mathsf{node}_\tau(a_2, b_2.\mathsf{node}_\sigma(a_1, b_1.x(b_1, b_2))) \tag{7}$$

$$x : 1 \mid a \vdash \mathsf{node}_\sigma(a, b.x(b)) = \mathsf{node}_\sigma(a, b.x(a \oplus b)) \tag{8}$$

第一个方程表明，独立动作可以按任意顺序发生。第二个方程编码了因果依赖的传递性。没有涉及 end 的方程；直观地说，end 会结束整个程序的执行。

在第4 节中，我们将给出一个关于分叉线程的参数化代数理论的扩展示例，并通过定理4.2 至4.4中的示例展示等式推理。

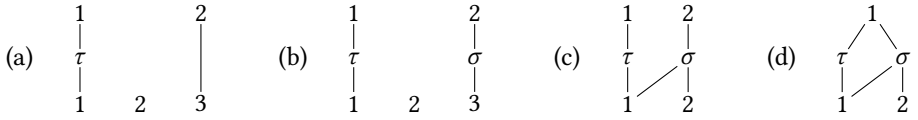


图 4. 由节点操作构造的项的图示示例。(a) 是项 $\mathrm{node}_\tau(a_1, b.x(b, a_3))$；变量 x 的两个输入上方的数字 1 和 2 分别对应于：2。(b) 是将 $\mathrm{node}_\sigma(a_3', a_3.-)$ 应用于 (a)。(c) 是方程 (3) 中的项；根据方程 (5)，将 $a_1 \oplus a_2$ 代入 a_3'，即可由 (b) 得到 (c)；根据方程 (6)，将一个项代入计算变量 x：2，即可由 (c) 得到 (d)。

给定一个参数化理论 $\mathsf{T} = (\mathsf{O}\ E)$，我们可以在 T 的项上构造一个等价关系 $=_{\mathcal{T}}$，称为可推导等式：将所有来自 E 等式公理的同时代换实例对自反性、对称性、传递性以及两条同余规则（分别针对变量和 O 中的运算）闭包即可。下面给出针对变量的同余规则；它使我们在对参数化项进行推理时，能够使用半格理论中的方程。

$$\frac{\left(\Delta \vdash u_i = u_i'\right)_{i=1}^{m}}{\Gamma, x : m \mid \Delta \vdash x(u_1, \ldots, u_m) =_{\mathcal{T}} x(u_1', \ldots, u_m')}$$

3.3 参数化代数理论的模型

我们通过类比普通代数理论的模型，回顾参数化代数理论的模型。对于本文而言，只需考虑参数化理论为半格理论的情形（定理 3.3），但也存在更一般的模型概念 [44–46]。在第 3.3.3 节中，我们通过考察定理 3.7 中的带标记偏序集理论来说明模型。

3.3.1 有限集与关系范畴和半格理论之间的联系。我们将自然数定义为范畴 **FinRel** 的对象 n，将关系定义为其态射 $n \to n'$ $R \subseteq [n] \times [n']$，其中 $[n]$ 表示集合 $\{1, \ldots, n\}$。$R \subseteq [n] \times [n']$与 $S \subseteq [n'] \times [n'']$ 的复合 $S \circ R$是通常的关系复合。我们也可以将 **FinRel** 视为幂集单子的 Kleisli范畴，并将其限制在有限集上。

对于每个 p，我们都可以在半格理论中项（的等价类）集合 $\{[a_1, \ldots, a_p \vdash u]\}$ 与态射集合 $\mathbf{FinRel}(1, p)$ 之间定义一个同构。（事实上，**FinRel** 是半格 Lawvere 理论的对范畴。）

$$[\![a_1, \ldots, a_p \vdash a_i]\!] = \{(1, i)\} \qquad [\![a_1, \ldots, a_p \vdash 0]\!] = \emptyset$$

$$[\![a_1, \ldots, a_p \vdash u_1 \oplus u_2]\!] = [\![a_1, \ldots, a_p \vdash u_1]\!] \cup [\![a_1, \ldots, a_p \vdash u_2]\!]$$

3.3.2 参数化理论在 $\mathbf{Set}^{\mathbf{FinRel}}$中的模型。普通代数理论的一个模型由一个集合（即载体）以及用于解释该理论中运算的结构组成，并且所有等式公理都得到满足。我们研究的是由半格理论参数化的理论，因此我们将考虑这样的模型：载体不是一个集合，而是一个由范畴 **FinRel**的对象所索引的集合族。

定义 *3.8*。设 O 是一个参数化签名。一个 O-结构 X 是函子范畴 $\mathbf{Set}^{\mathbf{FinRel}}$ 中的一个对象 X，并且对于每个运算 $\mathsf{O} : (p \mid m_1、\ldots、m_k)$，都配备了一个由自然数 n索引的函数族，并且相对于**FinRel**中的态射满足自然性：

$$\mathsf{O}_{X,n} : X(n + m_1) \times \ldots \times X(n + m_k) \to X(n + p)$$

给定一个 $\mathcal{O}$-结构 $\mathcal{X}$，运算的解释可以借助第 3.3.1 节中对半格项的解释扩展到所有项。项 $x_1 \colon m_1, \ldots, x_k \colon m_k \mid a_1, \ldots, a_p \vdash t$是

解释为一族函数

$$[\![t]\!]_{X,n} : X(n+m_1) \times \ldots \times X(n+m_k) \to X(n+p)$$

对 n自然，由结构递归定义。一个计算变量

$$x_1 : m_1, \ldots, x_k : m_k \mid a_1, \ldots, a_p \vdash x_i(u_1, \ldots, u_{m_i})$$

is 解释为先作投影，再对其半格项 inpu 进行解释 ts

$$X(n+m_1) \times \ldots \times X(n+m_k) \xrightarrow{\pi_i} X(n+m_i) \xrightarrow{X(n+[[\![u_1]\!],\ldots,[\![u_{m_i}]\!]])} X(n+p)$$

其中 $[\![u_1]\!]$，…，$[\![u_{m_i}]\!] : 1 \to p$是 **FinRel**中的态射。

形如 $\Gamma \mid a_1, \ldots, a_p \vdash \mathrm{O}(u_1, \ldots, u_{p'}, b_1 \ldots b_{m'_1}.t_1, \ldots, b_1 \ldots b_{m'_{k'}}.t_{k'})$ 的项被解释为 n-索引族，

$$X(n + [\mathrm{id}_p, [\![u_1]\!], \ldots, [\![u_{p'}]\!]]) \circ \mathrm{O}_{X,n+p} \circ \langle [\![t_1]\!]_{X,n}, \ldots, [\![t_{k'}]\!]_{X,n} \rangle,$$

其中 $\mathrm{O} : (p' \mid m'_1, \ldots, m'_{k'})$。映射 $[\mathrm{id}_p, [\![u_1]\!], \ldots, [\![u_{p'}]\!]] : p + p' \to p$使用参数变量 $a_1, \ldots, a_p$ 来解释 O 的 p'变元。

定义 *3.9.* 设 $\mathrm{T} = (\mathrm{O}\ E)$ 为一个参数化理论。若对于来自 $E, \Gamma \mid \Delta \vdash s = t$ 的每一条等式公理，以及每一个自然数 n，下列函数都相等，则 O-结构 X 是理论 T 的一个 模型：

$$[\![\Gamma \mid \Delta \vdash s]\!]_{X,n} = [\![\Gamma \mid \Delta \vdash t]\!]_{X,n}.$$

命题 3.10. 对于参数化理论 $\mathcal{T}$，可推导等式 $=_{\mathcal{T}}$是可靠的：若 $s =_{\mathcal{T}} t$ 可推导，则 $[\![s]\!]_{\mathcal{X}} = [\![t]\!]_{\mathcal{X}}$ 在任意 $\mathcal{T}$-模型 $\mathcal{X}$ 中成立。

3.3.3 带标记偏序集理论的一个模型。为说明上一节中的模型概念，我们为定理3.7中的标记偏序集参数化代数理论构造一个模型。为此，我们推广 Σ-标记偏序集的概念，该概念来自定理 2.3。

定义 *3.11*。An n-输入 m-输出 Σ-标记偏序集 $P = V_P, \leq_P, l_P$ 由一个元素集合 V_P、一个标记函数 $l_P : V_P \to \Sigma$以及集合 $[n] \uplus V_P \uplus [m]$上的偏序 $\leq_P$组成，其中 n 输入元素是极小元，m 输出元素是极大元。

这类偏序集的示例见图 4，输入位于底部，输出位于顶部。若没有输入和输出，定理 3.11 就退化为普通的 Σ-标记偏序集。两个 n-输入 m-输出 Σ-标记偏序集 P和 Q之间的同构是一个双射 $f : V_P \to V_Q$，它保持标签，并且满足 $\mathrm{id}_{[n]} \uplus f \uplus \mathrm{id}_{[m]}$ 保持并反映偏序关系。

对于每个自然数 n，定义集合 $S_m(n)$ 包含 同构类 n-输入 m-输出 Σ-标记偏序集。给定一个关系 $R \subseteq [n] \times [n']$，$S_m(R)$ 通过更新标记，使其作用于一个标记偏序集$i \leq e$为 $i' \leq e$，条件是 $(i, i') \in R$。图 4 (c) 中的偏序集由图 4 (b) 通过此动作得到。

对于每个自然数 m，我们为 S_m 配备一个运算族 $\mathrm{node}_{\sigma,\ m}$，每个 n对应一个：

$$(\mathrm{node}_{\sigma,m})_n : S_m(n+1) \to S_m(n+1)$$

它给定一个标记偏序集，为其$(n+1)$号输入标上 σ，并在 σ正下方创建一个新的$(n+1)$ 输入 σ。图 4 (b) 是对图 $(\text{节点}_{\sigma,\ 2})_3$ 施加于图 4 (a) 后的结果。

注。 标记偏序集可以组织成一个 PROP（参见 [24]），其中态射 $n \to m$ 是标记偏序集 $S_m(n)$，恒等态射由不含带标记元素的偏序集给出；复合是将一个偏序集的输出“接入”另一个偏序集的输入，而幺半复合则是并置。这种范畴化表述有助于证明定理3.12 和定理 3.15。同一个 PROP 在 [13] 中作为 L-标记传递有向无环图的 PROP 出现，并被刻画为由适当的等式理论自由生成。在 [27], 中，该 PROP 被证明是由某个重写系统呈示的幺半范畴。其他地方也出现了类似的范畴化思想，例如 [4, 7, 12],，但通常侧重一阶理论；而我们的目标是通过单子为编程语言提供语义（§7）。

对于每个计算变量的上下文 Γ，我们构造一个函子 S_Γ： **FinRel** $\to$ **Set**，它将按照定理 3.8 的意义，成为带标记偏序集理论的一个模型的载体。对于每个自然数 n，定义集合

$$\mathsf{S}_\Gamma(n) = \biguplus_{x:m \in \Gamma} S_m(n) \uplus S_0(n).$$

我们通过逐点应用 $(\mathrm{node}_\sigma)_n$，为 $\mathsf{S}_\Gamma(n)$ 配备一个运算：$\mathsf{S}_\Gamma(n+1) \to \mathsf{S}_\Gamma(n+1)$。令 $\mathrm{end}_n \colon 1 \to \mathsf{S}_\Gamma(n)$ 为这样一个函数：它从右注入 $S_0(n)$ 中选取带标记偏序集，该偏序集仅以 n 输入作为元素，并带有离散序。

命题 3.12。 对于每个上下文 Γ，函子 S_Γ，连同上文定义的自然变换node_σ 和 end，是定理 *3.7* 中参数化带标记偏序集理论的一个模型。

3.4 参数化代数理论的自由模型

现在我们回到对一般参数化代数理论模型的研究。利用 O-结构之间显然的同态概念，我们可以讨论相对于一组生成元 $X \in \mathbf{Set}^{\mathbf{FinRel}}$ 自由的 O-结构和 T-模型。

定义 *3.13*。 考虑一个载体为 $Y \in \mathbf{Set}^{\mathbf{FinRel}}$ 的 T-模型 Y，以及一个态射 $\eta_X \colon X \to Y$(属于 $\mathbf{Set}^{\mathbf{FinRel}}$)。如果对于任意其他模型 Z，以及任意态射 $f \colon X \to Z$（属于$\mathbf{Set}^{\mathbf{FinRel}}$），都存在唯一的模型同态 $\hat{f} \colon \mathrm{Y} \to \mathrm{Z}$ 延拓 f，即 $\hat{f} \circ \eta_X = f$属于 $\mathbf{Set}^{\mathbf{FinRel}}$，那么模型 Y 就是自由于 X的。

给定一个计算变量上下文 Γ，考虑函子 V_Γ，其中：

$$V_\Gamma(n) = \left\{ [x(u_1, \dots, u_m)]_{=_\mathcal{T}} \;\middle|\; \Gamma \mid a_1, \dots, a_n \vdash x(u_1, \dots, u_m) \right\}.$$

V_Γ 中项上的等价关系并非平凡，因为参数项 u_i 按照半格方程取商。

该项模型 由函子 $F_\mathcal{T}(V_\Gamma)$ 给出，其中包含项的等价类：

$$F_\mathcal{T}(V_\Gamma)(n) = \left\{ [t]_{=_\mathcal{T}} \;\middle|\; \Gamma \mid a_1, \dots, a_n \vdash t \right\} \tag{9}$$

对态射 $n \to n'$（即关系 $R \subseteq [n] \times [n']$）的动作由参数代换给出。利用句法项构造规则，可以将函子 $F_\mathcal{T}(V_\Gamma)$ 构造成一个 T-模型；通过将变量嵌入项中，我们可以构造一个态射 $\eta_{V_\Gamma} \colon V_\Gamma \to F_\mathcal{T}(V_\Gamma)$。我们使用项模型来证明下面的完备性结果。

命题 3.14。

(1) 函子 $F_\mathcal{T}(V_\Gamma)$ 是一个 $\mathcal{T}$-模型，而且还是以 V_Γ 为基的自由$\mathcal{T}$-模型。(2) 参数化代数理论中的可推导等式 $=_\mathcal{T}$ 是完备的：如果一个方程在每个 $\mathcal{T}$-模型中都成立，那么它可在 $=_\mathcal{T}$ 中推导出来。

现在，我们可以利用自由模型的泛性质，刻画定理 3.12 中的带标签偏序集模型。具体来说，由 $\{[\cdot \mid \cdot \vdash t]_{=_C}\}$ 构造的闭项的等价类节点 和 终止 (定理 3.7) 与普通的 Σ-标记偏序集构成双射。

定理 3.15。 对于每个上下文 Γ，函子 S_Γ 连同自然变换 节点$_\sigma$和 end 同构于定理 $F_C(V_\Gamma)$ 中带标记偏序集理论的自由模型 *3.7*。

证明注记。解释同态 $F_C(V_\Gamma) \to \mathsf{S}_\Gamma$ 由自由模型到其自身的唯一映射给出。我们定义逆映射 $\mathsf{S}_\Gamma \to F_C(V_\Gamma)$，它将带标记偏序集线性化为嵌套的 $\mathrm{node}_\sigma(u, a.-)$ 项，其中偏序集的每个元素都对应一个以动作 σ 标记的 node_σ 运算。固定一个偏序集元素，其在偏序关系中所有先于它的元素都由复合线程标识符 u 编码。方程（7）保证映射 $\mathsf{S}_\Gamma \to F_C(V_\Gamma)$ 与所选的线性化无关；方程（8）则通过要求因果依赖传递闭包化，保证所有项都与该映射像中的某个项 $(=_C)$ 相等。证明策略与定理 5.4 类似，后者我们会提供更详细的说明。 □

4 动态线程的参数化代数理论

本节将第 2 (分叉、等待、停止 和 act$_\sigma$节中的并发特性公理化为一个参数化代数理论。在第 5 节中，我们使用与第 3.3.3 节类似的标记偏序集，对该理论作语义解释。随后，在第 7 节中，我们扩展语义，以刻画第 2 节中的整个并发编程语言。

4.1 理论的表述

4.1.1 签名。 我们引入一个以半格理论为参数的动态线程理论T，其签名如下，其中 σ 取自一个固定的可观测动作集合 Σ:

$$\mathsf{fork} : (0 \mid 1, 0) \qquad \mathsf{wait} : (1 \mid 0) \qquad \mathsf{stop} : (0 \mid) \qquad \mathsf{act}_\sigma : (0 \mid)$$

在项 $\mathsf{fork}(a.x(a)\ y)$ 中，变量 x是父线程，而 y是子线程；参数 a 是子线程的线程标识符 y，并在 x中绑定。父线程可以等待名为 a 的子线程完成，然后以 z继续执行，使用运算 $\mathsf{wait}(a,\ z)$。运算 stop 没有延续，表示当前线程已完成执行；act$_\sigma$ 执行可观测动作 σ 后结束。参数具有半格结构，因此可以等待一个复合线程标识符，例如，$\mathsf{wait}(a_1 \oplus a_2,\ z)$ 会等待 a_1 和 a_2都完成；也可以不等待任何线程标识符，即 0。

例*4.1.* 项 t_1 表示动作 σ_1 顺序执行后接着执行 σ_2，而 t_2 和t_3 则表示 σ_1 与 σ_2并发执行:

$$t_1 = \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1}) \qquad t_2 = \mathsf{fork}(a.\mathsf{act}_{\sigma_2}, \mathsf{act}_{\sigma_1}) \qquad t_3 = \mathsf{fork}(a.\mathsf{act}_{\sigma_1}, \mathsf{act}_{\sigma_2})$$

不过， t_2和 t_3的预期语义略有不同。在项 $\mathsf{fork}(b.\mathsf{wait}(b,\ \mathsf{act}_\tau)\ t_2)$ 中，标识符 b 仅指线程 act$_{\sigma_2}$，而不指其子线程 act$_{\sigma_1}$，因此一种可能的执行方式是 $\sigma_2\tau\sigma_1$。但在 $\mathsf{fork}(b.\mathsf{wait}(b,\ \mathsf{act}_\tau)\ t_3)$ 中则无法这样执行，因为这里 σ_1必须先于 τ发生。

更一般地，在表达式 $\mathsf{fork}(a.x(a)\ y)$ 中，我们常将 x称为主线程，因为其标识符可供环境用于等待，而 y的标识符则只有 x可用。

4.1.2 方程。 动态线程理论的等式公理见图 5。不存在涉及可观测动作 act$_\sigma$的方程。方程（12）指出，如果 x正在等待 a 完成，那么将来再等待 a 就是多余的。分叉的交换律，即方程（14），仅当子线程 y 和 z 不使用彼此的标识符时才成立。同样，结合律，即方程（15），在父线程 x 不使用 b 的标识符时成立。 z方程（16）表示，分叉出子线程 x 并等待其完成，等同于将 x 作为主线程运行。等待是必要的，因为它会强制

Equa描述等待与线程 ID 半格结构之间交互的方程。

$$x : 0 \mid - \vdash \mathsf{wait}(0, x) = x \tag{10}$$

$$x : 0 \mid a, b \vdash \mathsf{wait}(a, \mathsf{wait}(b, x)) = \mathsf{wait}(a \oplus b, x) \tag{11}$$

$$x : 1 \mid a, b \vdash \mathsf{wait}(a, x(b)) = \mathsf{wait}(a, x(a \oplus b)) \tag{12}$$

等待和分叉运算可交换；分叉具有交换性和结合性。

$$x\colon 1, y\colon 0 \mid b \vdash \text{等待}(b, \text{分叉}(a.x(a)\ y)) = \text{分叉}(a.\text{等待}(b, x(a)), \text{等待}(b, y)) \tag{13}$$

$$x\colon 2, y\colon 0, z\colon 0 \mid - \vdash \text{分叉}(a.\text{分叉}(b.x(a, b)\ y)\ z) = \text{分叉}(b.\text{分叉}(a.x(a, b)\ z)\ y) \tag{14}$$

$$x\colon 1, y\colon 1, z\colon 0 \mid - \vdash \text{分叉}(a.x(a), \text{分叉}(b.y(b)\ z)) = \text{分叉}(b.\text{分叉}(a.x(a)\ y(b))\ z) \tag{15}$$

停止运算是分叉的单位元。

$$x\colon 0 \mid - \vdash \text{分叉}(a.\text{等待}(a, \text{停止})\ x) = x \tag{16}$$

$$x\colon 1 \mid b \vdash \text{分叉}(a.x(a), \text{等待}(b, \text{停止})) = x(b) \tag{17}$$

图5. 动态线程参数化代数理论 T 的方程。

要等待的环境 x 即使作为子线程执行时也是如此。方程（17）会移除不执行任何可观测动作的子线程；其中将参数 b 代换为 a 在 x中。

例*4.2.* 我们可以用 act_σ 来编码运算 $\mathsf{perform}_\sigma(x)$ 该运算执行动作 σ 然后继续执行 x。我们可以通过将 act_σ 从 $\mathsf{perform}_\sigma(x)$ 还原，方法是将 x 设为 stop 并使用方程（16）。

$$\mathsf{perform}_\sigma(x) \stackrel{\text{def}}{=} \mathsf{fork}(a.\mathsf{wait}(a, x), \mathsf{act}_\sigma)$$

代数操作act_σ 和$\mathsf{perform}_\sigma$ 是第2.2.1节讨论的泛型效应act_σ 和$\mathsf{perform}_\sigma$ 的对应物。 ___

例*4.3*。 节点$_\sigma$ 运算（定理3.7）可以编码为：

$$\mathsf{node}_\sigma(a, b.x(b)) \stackrel{\text{def}}{=} \mathsf{fork}(b.x(b), \mathsf{wait}(a, \mathsf{act}_\sigma))$$

方程（ 7）表示节点$_\sigma$ 运算的交换性，其中分叉（方程（14））；而方程（8）可以推导出 ：

$$\begin{aligned}
\mathsf{node}_\sigma(a, b.x(b)) &= \mathsf{fork}(b.x(b), \mathsf{wait}(a, \mathsf{fork}(c.\mathsf{wait}(c, \mathsf{stop}), \mathsf{act}_\sigma))) && \text{(eq. (16))}\\
&= \mathsf{fork}(b.x(b), \mathsf{fork}(c.\mathsf{wait}(a \oplus c, \mathsf{stop}), \mathsf{wait}(a, \mathsf{act}_\sigma))) && \text{(eq. (13) and eq. (11))}\\
&= \mathsf{fork}(c.x(a \oplus c), \mathsf{wait}(a, \mathsf{act}_\sigma)) && \text{(eqs. (15) and (17))}
\end{aligned}$$

例*4.4*。 在项 $t = \mathsf{fork}(a.\mathsf{wait}(a,\ \mathsf{act}_{\sigma_1}),\ \mathsf{fork}(b.\mathsf{stop},\ \mathsf{act}_{\sigma_2}))$ 中，线程标识符 a 仅指向线程 stop，而不指向其子线程 act_{σ_2}:

$$\begin{aligned}
t &= \mathsf{fork}(b.\mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_1}), \mathsf{stop}), \mathsf{act}_{\sigma_1}) && \text{(eq. (15))}\\
&= \mathsf{fork}(b.\mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_1}), \mathsf{wait}(0, \mathsf{stop})), \mathsf{act}_{\sigma_1}) && \text{(eq. (10))}\\
&= \text{分叉}(b.\mathsf{act}_{\sigma_1},\ \mathsf{act}_{\sigma_2}) && \text{(eq. (17) and eq. (10))}
\end{aligned}$$

4.2 范式

在本节中，我们确定一类便于处理的项子类，称之为范式。我们证明，动态线程理论 T 中的所有项在可推导等式 $=_{\mathcal{T}}$ 的意义下都与某个范式相等。我们将范式定义为具有如下形式的上下文中的项：

$$\Gamma \mid a_1, \ldots, a_n \vdash \mathsf{fork}(b_1. \ldots \mathsf{fork}(b_p.\mathsf{wait}(u_{p+1}, \mathsf{stop}), \mathsf{wait}(u_p, t_p)), \ldots \mathsf{wait}(u_1, t_1)) \tag{18}$$

其中每个子项 t_i要么是可观测动作 act_σ，要么是某个$x\colon m$（属于 Γ）上的变量 $x(u_{i1},\dots,u_{im})$。我们还要求参数（即复合线程标识符） $u_1,\dots,u_{p+1}$，以及出现在每个 $t_i = x(u_{i1},\dots,u_{im})$ 中的参数，满足下文所述的闭包条件。

直观地说，一个范式由一个父线程组成，它分叉出 p 个子线程，等待某些线程标识符的集合，u_{p+1}，然后结束。子线程必须恰好执行一个动作，或者是一个自由计算变量。根据项构造规则， u_i只能使用此前已分叉出的线程标识符 $b_1,\dots,b_{i-1}$，或上下文 $a_1,\dots,a_n$中的线程标识符。

示例 *4.5*。 项 分叉(b_1.等待(b_1,动作σ_2),动作σ_1)（定理 4.1中的项）不属于范式，但它与以下范式 ($=_{\mathcal{T}}$)相等：

$$\mathrm{nf}_1 = \mathrm{fork}(b_1.\mathrm{fork}(b_2.\mathrm{wait}(b_1 \oplus b_2, \mathrm{stop}), \mathrm{wait}(b_1, \mathrm{act}_{\sigma_2})), \mathrm{act}_{\sigma_1})$$

要证明这一点，使用等式（16）接着使用（13）和（11），证明子项 等待(b_1,动作σ_2) 等于

$$\mathrm{wait}(b_1, \mathrm{fork}(b_2.\mathrm{wait}(b_2, \mathrm{stop}), \mathrm{act}_{\sigma_2})) = \mathrm{fork}(b_2.\mathrm{wait}(b_1 \oplus b_2, \mathrm{stop}), \mathrm{wait}(b_1, \mathrm{act}_{\sigma_2})).$$

4.2.1 范式的闭包条件。 形如（18）的项要成为范式，必须满足以下闭包条件：如果标识符 b_j出现在 u_i中，那么 u_j中的所有标识符都包含在 u_i中；当 b_j出现在 u_{ik}中时，也要满足类似条件，其中 $t_i = x(u_{i1},\dots,u_{im})$；而且 u_i中的标识符必须出现在每个 u_{ik}中。施加这些闭包条件意味着范式包含了关于不同线程之间依赖关系的冗余信息，但这将有助于我们在第 5.1.2 节和定理 5.4中建立范式与标记偏序集之间的对应关系。

示例 *4.6*。 定理 4.5 中的范式满足这些闭包条件，下面的项也满足，其中自由标识符为 a_1 和 a_2：

$$\mathrm{nf}_2 = \mathrm{fork}(b_1.\mathrm{fork}(b_2.\mathrm{wait}(b_1 \oplus b_2 \oplus a_1、\ \mathrm{stop})、\ \mathrm{wait}(b_1 \oplus a_1、\ x(b_1 \oplus a_1 \oplus a_2)))、\ \mathrm{wait}(a_1、\ \mathrm{act}_{\sigma_1})) \quad (19)$$

定义 *4.7*。 固定一个由计算变量 Γ组成的上下文。对于每个自然数 n，定义集合 $\mathsf{NF}_\Gamma(n)$，其中包含范式，即形如（ 18 ）且满足上述闭包条件的项，再对以下等价关系取商：由重新排列彼此不依赖的子线程，以及复合线程标识符上的半格方程所生成的等价关系。此外，NF_Γ 在关系 $R \in [n] \times [n']$ 上具有由参数替换给出的函子作用。

根据此定义，同一范式等价类的两个代表元在动态线程理论中也($=_{\mathcal{T}}$)-相等（图5），因为独立子线程的重排对应于方程（14），而 $=_{\mathcal{T}}$根据定义，在半格方程下封闭。

定理4.8。动态线程理论$\mathcal{T}$中的每个项 $\Gamma \mid a_1, \dots, a_n \vdash t$ 都可推导出与来自$\mathsf{NF}_\Gamma(n)$的某个范式等价类相等；此等价类并非先验唯一。

我们使用图5中的方程对项进行归纳，以证明该定理。作为推论，范式通过恒等映射满射到项（的等价类）上。目前我们尚未证明每个项都等于唯一的范式等价类；我们将在定理5.5中证明这一点。

5 动态线程理论的表示定理

本节将使用一种与第 4 节中类似的标记偏序集概念，从语义上解释动态线程的参数化理论。在第 3.3.3 节中也使用了类似概念。我们将在第 5.1 节非正式地讨论标记偏序集，然后在第 5.2 节和5.3 节给出其形式化定义，并证明它们构成一个自由模型。在第 6 节中，我们将证明该模型在某种意义上是完备的。

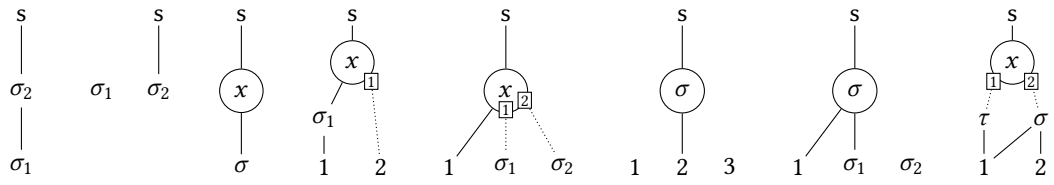


图 6. 标记偏序集的示例。(a) 是项 fork(*a*.wait(*a*, act$_{\sigma_2}$), act$_{\sigma_1}$)。(b) 是 fork(*a*.act$_{\sigma_2}$, act$_{\sigma_1}$)。(c) 是 fork(*a*.wait(*a*, *x*), act$_\sigma$)。(d) 是方程 (19) 中的范式。(g) 是在 (*e*) 中用 (f) 代入 (*x*: 2) 的结果。(h) 是使用本节引入的标记偏序集概念对图 4 (c) 的表示。

5.1 带孔的标记偏序集示例

我们通过示例介绍标记偏序集，并使用动态线程理论中的项（第 4 节）来阐释它们。我们将在定理 5.1 和5.2 中正式定义标记偏序集。

5.1.1 标记偏序集表示项。 考虑图 6 (a)：偏序集的两个元素由可观测动作标记 σ_1 和 σ_2。实线表示因果依赖，并诱导一个偏序： σ_1 必须发生在 σ_2之前。所有偏序集都包含一个特定的极大元 s ，它表示主线程的 *end*；我们将使用 s 来定义类似于 分叉的偏序集运算。在图 6 (b) 中， σ_2 属于主线程，但 σ_1 不属于，正如定理 4.1 中所讨论的。偏序集的元素可以由计算变量标记，例如， $x : 0$ 位于图 6 (c) 中。

项的自由线程标识符在其偏序集中表现为极小元素，例如在图 6 (d) 中，它们编号为 1 和 2。约束线程标识符不会出现在偏序集中。图 6(d) 描绘了以下项的偏序集：

$$x : 1 \mid a_1, a_2 \vdash \mathsf{fork}(b_1.\mathsf{wait}(b_1, x(b_1 \oplus a_2)), \mathsf{wait}(a_1, \mathsf{act}_{\sigma_1}))$$

在偏序集中，有一个标记为 x 的元素，它有一个输入。虚线不属于偏序；它表示传递给变量 x的线程标识符，并意味着 x 可能会等待 a_2，具体取决于计算 x 是什么。虚线所表示的是一种称为可见性的关系，假定该关系相对于偏序（实线）向下封闭，并包含偏序中位于 x 之下的所有元素；因此，我们省略从 σ_1 和 1 指向 x的虚线。

5.1.2 标记偏序集是范式。回想一下，范式具有如下形式，其中每个 t_i 要么是一个可观测动作，要么是来自 Γ的计算变量：

$$\Gamma \mid a_1, \dots, a_n \vdash \mathsf{fork}(b_1. \dots \mathsf{fork}(b_p.\mathsf{wait}(u_{p+1}, \mathsf{stop}), \mathsf{wait}(u_p, t_p)), \dots \mathsf{wait}(u_1, t_1))$$

相应的偏序集有 n与自由线程标识符相对应的最小元素 a_1， ……， a_n。对于项 t_1 至 t_p中的每一项，都有一个带标签的元素。停止的父线程对应于特殊的最大元素s。偏序（实线）编码了由复合线程标识符 u_1 至 u_{p+1}给出的依赖关系。可见性关系（虚线）对应于传递给变量的线程 ID $t_i = x$（u_{i1}， ……， u_{im}）。第 4.2.1 节中关于范式的闭包条件，对应于偏序的传递性，以及可见性关系相对于偏序是向下封闭的这一事实。

5.1.3 标记偏序集的代换。 与参数化理论中的项一样，标记偏序集也允许将另一个偏序集代入计算变量；其形式化定义将在第 5.3.2 节讨论。图 6 中的偏序集 (e) 和 (f) 分别表示以下项

$x : 2 \mid a_1 \vdash \mathsf{fo}\ \ \mathsf{rk}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(a_1, x(b_1, b_2)), \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1})$ and $\cdot \mid a_1, b_1, b_2 \vdash \mathsf{wait}(b_1, \mathsf{act}_\sigma)$

而 (g) 是将 (f) 代入 (e) 中的 $x : 2$ 后得到的项：

$$\cdot \mid a_1 \vdash \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(a_1, \mathsf{wait}(b_1, \mathsf{act}_\sigma)), \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1})$$

(e) 和 (f) 的输入 1 被识别，而 (f) 的输入 2 和 3 则映射到变量 x的两个输入。

用复合线程标识符替换偏序集的一个输入，对应于对项进行参数替换。我们在定理5.3中定义了偏序集上的这一运算。

5.1.4 与第 3.3.3 节中的标记偏序集以及普通标记偏序集的联系。定理 3.11 中具有 n 个输入和 m 个输出的 Σ-标记偏序集，是本节所述标记偏序集的一个特例。例如，图 4 (c) 中对应于项$x : 2 \mid a_1, a_2 \vdash$ 节点$_\sigma(a_1 \oplus a_2, b_1.$节点$_\tau(a_1, b_2.x(b_2, b_1)))$的偏序集，展示于图 6 (h) 中。变量 x 的两个输入对应于图 4 (c) 中偏序集的两个输出。

如果我们将标记主线程结束的结构 s 本身也视为一个标签，那么一个没有输入且没有任何顶点以计算变量为标签的标记偏序集，就是定理 2.3意义下的普通带标记偏序集，即一个偏序集 $(X, \leq)$ 及一个函数 $X \to \Sigma \uplus \{\mathsf{s}\}$。

5.2 带洞的标记偏序集：形式化定义

下面的两个定义将上一节关于标记偏序集的概念形式化。

定义 *5.1*。设 $\Gamma = x_1 : m_1, \ldots, x_k : m_k$是一个由计算变量组成的上下文，$\Sigma$是一个可观测动作集合。一个带有 n个输入的（Γ, Σ）-标记偏序集 $G = V_1, V_2, \leq_G, l_1, l_2$由以下部分组成：

- 输入顶点集合 $n\{1, \ldots, n\}$；
- 有限个两两不相交的顶点集合 V_1（以动作为标签）和 V_2（以变量为标签）；
- 一个特定顶点 s；
- 底集 $|G| \stackrel{\text{def}}{=} [n] \uplus V_1 \uplus V_2 \uplus \{\mathsf{s}\}$ 上的一个偏序 $\leq_G$（以实线表示）；
- 一个标记函数 $l_1 : V_1 \to \Sigma$，将 V_1 中的顶点映射到可观测动作；
- 一个函数 l_2，用取自 Γ 的变量（$x : m$）标记 V_2 中的顶点：

$$l_2 : V_2 \to \sum_{(x:m) \in \Gamma} \left(f : [m] \to \mathcal{P}(|G|)\right)$$

并且根据该变量的元数 m，l_2还指定一个函数 $f : [m] \to \mathrm{P}(|G|)$，其值域为 $|G|$ 的幂集。（每个 $f(i)$ 以虚线表示）。

如果没有输入，也没有由变量标记的顶点，即 $n = 0$ 和 $V_2 = \emptyset$，则标记偏序集成为标签取自 $\Sigma \uplus \{\mathsf{s}\}$ 的普通带标记偏序集。标记偏序集之间的同构 $\alpha : G \to G'$由两个双射 $\alpha_1 : V_1 \to V_1$与 $\alpha_2 : V_2 \to V_2'$组成，它们在适当意义上保持这两个标记函数，并且使得 $\mathrm{id}_{[n]} \uplus \alpha_1 \uplus \alpha_2 \uplus \mathrm{id}_{\{\mathsf{s}\}}$ 保持并反映该偏序。

为得到第 5.1.2 节所述的标记偏序集与范式之间的对应关系，我们将注意力限制在良构的标记偏序集上。

定义 *5.2*. 一个带有 n输入 $G = V_1, V_2, \leq_G, l_1, l_2$ 的 (Γ, Σ)-标记偏序集若满足以下条件，则称为良构：(1) 关于偏序 $\leq_G$，n 输入是极小元，且 s 是极大元；(2) 对每个满足 $l_2(e) = (x : m, f : [m] \to \mathrm{P}(|G|))$ 的 $e \in V_2$，以及每个 $i \in [m]$：$e \in f(i)$ 和 $\mathsf{s} \notin f(i)$，$f(i)$ 关于 $\leq_G$ 向下封闭。(3) 考虑由标记函数 l_2 诱导的可见性关系 $S \subseteq |G| \times |G|$：

$$(e', e) \in S \iff e \in V_2 \text{ and if } l_2(e) = (x : m, f) \text{ then } e' \text{ is in the image of } f.$$

关系 $(\leq_G) \cup S$ 的传递闭包是反对称的。

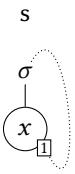


图 7. 一个不规范偏序集的示例。

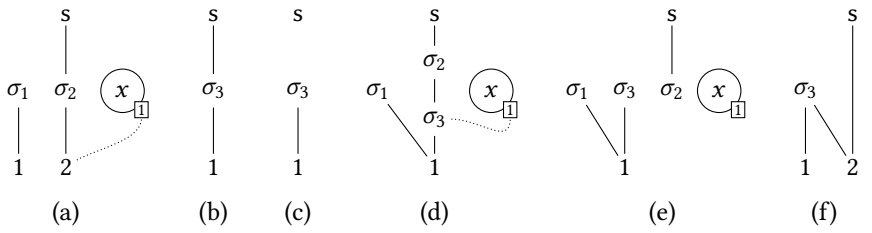


图 8. 标记偏序集上的分叉示例：将分叉父线程 (a) 与子线程 (b) 分叉，得到 (d)。若子线程是 (c)，则分叉的结果为 (e)。(f) 是对 (c) 应用 wait_1 后的结果。

第 5.1 节讨论的所有标记偏序集都满足上述良构条件。要求 s 为极大元，且 $s \notin f(i)$ 可确保子线程不知道主线程的标识符。向下闭包 $f(i)$ 和 $e \in f(i)$ 对应于范式的一些闭包条件（第 4.2.1节）。条件（3）确保在考虑可见性关系时，标记偏序集中不存在环。总体而言，良构性确保标记偏序集可以线性排序为形如（18）的范式。例如，图 7中的偏序集无法线性化，也不满足条件（3）。

5.3 标记偏序集模型与表示定理

5.3.1 模型结构。 为了基于标记偏序集构建动态线程理论的模型，我们将它们组织成如下函子 T_Γ： $\mathbf{FinRel} \to \mathbf{Set}$。

定义 *5.3.* 设 Γ 为计算变量的上下文。对每个自然数 n，定义集合 $T_\Gamma(n)$ 包含同构类，这些类由良构的 (Γ, Σ)-标记偏序集构成，并带有 n 输入。关系上的函子作用 $R \subseteq [n] \times [n']$ 作用于偏序集的输入：若 $(i, i') \in R$，则将 $i \le e$ 更新为 $i' \le e$，并相应地更新标记函数 l_2。

我们可以为 T_Γ 配备一个模型结构（在定理 3.8 的意义下），对应于 分叉、等待、停止 与 act_σ 这些动态线程理论的运算。对每个自然数 n，我们定义：

$$\mathsf{fork}_n : T_\Gamma(n+1) \times T_\Gamma(n) \to T_\Gamma(n).$$

分叉$_n(G_1, G_2)$ 的带标签顶点是来自 G_1 与 G_2 的那些顶点的并，且标签保持不变。分叉$_n(G_1, G_2)$ 的偏序通过将 G_1 的第 $(n+1)$ 个输入连接到 G_2 的 s 元，并在传递性下取闭包而得到。可见性关系经由同样的连接，由 G_1 与 G_2的相应关系得到。

图 8 展示了一个分叉示例。父线程 (a) 以及子线程 (b) 和 (c) 分别对应以下项：

$$x : 1 \mid a_1, a_2 \vdash t_1 = \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(a_2, \sigma_2), x(a_2)), \mathsf{wait}(a_1, \sigma_1))$$

$$x : 1 \mid a_1 \vdash t_2 = \mathsf{wait}(a_1, \mathsf{act}_{\sigma_3}) \qquad x : 1 \mid a_1 \vdash t_3 = \mathsf{fork}(c.\mathsf{stop}, \mathsf{wait}(a_1, \mathsf{act}_{\sigma_3}))$$

而 (d) 对应于 $\mathsf{fork}(a_2.t_1, t_2)$，(e) 对应于 $\mathsf{fork}(a_2.t_1, t_3)$。

运算 wait_n： $T_\Gamma(n) \to T_\Gamma(n+1)$新增一个输入 $n+1$，并将其连接到所有带标记元素以及 s。图 8(f) 是一个示例；它表示项 $\cdot \mid a_1, a_2 \vdash \mathsf{wait}(a_2, t_3)$。

运算 stop_n： $1 \to T_\Gamma(n)$选取了一个偏序集，其中仅有 n 输入和 s 作为元素，没有带标签顶点，并且采用离散偏序。运算 $\mathsf{act}_{\sigma,\ n}$： $1 \to T_\Gamma(n)$给出了一个偏序集，其中有一个标记为 σ 的顶点，该顶点在偏序中直接位于 s 下方，并且 n 输入不与任何对象相连。

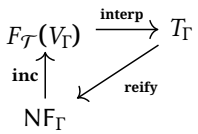
5.3.2 主定理。 现在我们证明，对于每个 T_Γ，标记偏序集构成动态线程理论的一个模型，并且该模型是自由的。回顾一下，我们在第 $F_{\mathcal{T}}(V_\Gamma)$ 节中将项模型描述为在 V_Γ 上的自由模型。第 3.4节中，我们利用7.1这一事实给出编程语言的一个解释： $T_\Gamma : \mathbf{FinRel} \to \mathbf{Set}$ 通过设定 T 在函子范畴 $\mathbf{Set}^{\mathbf{FinRel}}$ 上诱导出一个强单子 $T(V_\Gamma) = T_\Gamma$。

定理 5.4（表示定理）。对于每个上下文 Γ，函子 T_Γ （来自定理 *5.3*）以及标记偏序集上的运算分叉、等待、停止、act_σ 均满足图 *5*中的动态线程理论方程，因此构成一个模型，其含义见定理 *3.9*。

此外， T_Γ 与动态线程理论的项模型 $F_{\mathcal{T}}(V_\Gamma)$ 同构。

回想一下，在第 5.1.3 节中，我们曾非正式地讨论过标记偏序集的代换。利用上述定理中的模型同构，我们可以定义代换：先通过项替换将一个标记偏序集转换为项的等价类，再转换回偏序集。

5.3.3 表示定理的证明概要。为证明定理 5.4 ，我们考察下图。回想一下，函子 NF_Γ（来自定理 4.7）包含范式的等价类，而 $F_{\mathcal{T}}(V_\Gamma)$ 包含项的等价类。



该 **inc** 映射由范式到项的包含给出。映射**interp** 是通过实例化自由模型的泛性质而得到的唯一映射 $F_{\mathcal{T}}(V_\Gamma)$ (定理 3.13)，对于合适的 $V_\Gamma \to T_\Gamma$，它将计算变量映射到带标记偏序集；**interp** 本质上由第 $[\![-]\!]_{T_\Gamma}$ 节中标记偏序集模型里的项解释给出，参见 3.3.2。

对于每个自然数 n，我们定义一个函数 $\mathbf{reify}_n$，它借鉴第 5.1.2 节的直观思路，将标记偏序集线性化为范式。为了证明该函数定义良好且在 n中自然，我们使用良构标记偏序集的条件（定理 5.2）、闭包条件以及范式上的等价关系（第 4.2.1 节）。

我们证明 **reify** 既是复合映射 $\mathbf{interp} \circ \mathbf{inc}$ 的左逆，也是其右逆。证明它是左逆时，我们对范式中子线程的数量进行归纳；证明它是右逆时，则对偏序集中带标记元素的数量进行归纳。由于 **inc** 是满射（定理 4.8），这意味着 **interp** 是同构。根据自由模型的定义，我们已经知道 **interp** 是模型同态。

推论5.5。 范式的等价类到项的等价类的包含映射是单射。根据定理 *4.8*，每个项都等于唯一一个范式的等价类。

6 动态线程理论的完备性定理

定理 5.4 表明， $\Gamma \mid a_1, \ldots\ldots, a_n \vdash t$ 在动态线程理论中的项，与带有 n 个输入的(Γ, Σ)-标记偏序集完全对应。当上下文 Γ 为空且 $n = 0$时，这些标记偏序集实际上是普通标记偏序集，即带有函数$(X, \leq)$ 的偏序集 $X \to \Sigma \uplus \{s\}$。下一个定理表明，图5 中的$(\Gamma, \Sigma)$-标记偏序集的公理化，关于由普通标记偏序集的同构所诱导的等价关系，是完备的。

给定上下文 $\Gamma \mid \Delta \vdash t$ 中的一个项，闭合代换 γ 是指对 Γ中的每个变量（$x\colon m$）赋予一个项 $\gamma(x) = (\cdot \mid \Delta, b_1, \ldots\ldots, b_m \vdash s)$，且该项不含自由计算变量，使得 $\cdot \mid \Delta \vdash t[\gamma]$ 成立。闭合上下文 $\mathsf{C}[-]$是一个带有洞的项，使得给定项 $\cdot \mid \Delta \vdash t$时，判断式 $\cdot \mid \cdot \vdash \mathsf{C}[t]$ 成立。

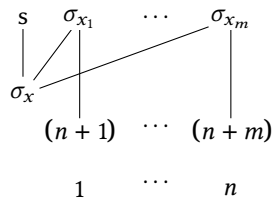
定理 6.1（完备性）。考虑动态线程理论中的项 $\Gamma \mid \Delta \vdash t_1, t_2$ 。若对于所有闭合代换 γ 以及所有闭合上下文 $C[-]$，这两个项都相等，即 $\cdot \mid \cdot \vdash C[t_1[\gamma]] = C[t_2[\gamma]]$，然后 $\Gamma \mid \Delta \vdash t_1 = t_2$。

证明概要。考虑不含自由计算变量的项 $\cdot \mid a_1, \ldots, a_n \vdash t_1, t_2$。定义一个上下文，将每个自由线程标识符绑定到一个可观测动作，并在主线程的末尾添加动作 $\mathsf{act}_{\sigma_{n+1}}$（$\sigma_1, \ldots, \sigma_{n+1}$ 与 t_1 和 t_2 中出现的任何可观测动作均不同）：

$$C[-] = \mathsf{fork}(a_1. \ldots \mathsf{fork}(a_n.\mathsf{fork}(a_{n+1}.\mathsf{wait}(a_{n+1}, \mathsf{act}_{\sigma_{n+1}}), [-]), \mathsf{act}_{\sigma_n}), \ldots \mathsf{act}_{\sigma_1})$$

回顾一下，$C[t_1]$和$C[t_2]$在 $T_\emptyset(0)$中被解释为标记偏序集。如果它们相等，那么根据定理5.4，标记偏序集 $[\![C[t_1]]\!]$ 与 $[\![C[t_1]]\!]$之间存在同构。我们将此同构调整为 $[\![t_1]\!], [\![t_2]\!] \in T_\emptyset(n)$ 之间的同构，并由此推出 $t_1 = t_2$，但省略细节。

现在考虑上下文 $\Gamma \mid a_1, \ldots, a_n \vdash t_1, t_2$中的项。考虑代换 γ，它将每个计算变量（$x : m$）从Γ映射到上下文$\cdot \mid a_1, \ldots, a_n, b_1, \ldots, b_m$中的项；右侧将其描绘为带有 $n + m$ 个输入的标记偏序集。我们假设，对于每个计算变量 x，动作σ_x,$\sigma_{x_1}, \ldots, \sigma_{x_m}$彼此不同，且不同于 t_1 和 t_2中的动作。我们可以将“新”动作编码为彼此不同的组合。即使只有一个动作，也可以通过交替使用不同的并行组合来实现。



假设我们有一个标记偏序集 $\alpha : [\![t_1[\gamma]]\!] \to [\![t_2[\gamma]]\!]$的同构，就可以构造一个同构 $\alpha' : [\![t_1]\!] \to [\![t_2]\!]$。在带标记元素上，$\alpha'$ 的作用与 α 相同，只是忽略标记为 σ_{x_1}、…、σ_{x_m} 的元素。$[\![t_1[\gamma]]\!]$ 中标记为 σ_x 的元素，与 $[\![t_1]\!]$中标记为 $(x : m)$ 的元素完全对应，而元素 σ_{x_i} 的依赖关系编码了 $[\![t_1]\!]$的可见性关系。为了证明 α' 是具有 n 个输入的 (Γ, Σ)-标记偏序集之间的同构，我们使用表示项的标记偏序集的良构性。特别是，我们依赖于定理 5.2 中的条件 (3)，该条件指出可见性关系不会导出环。□

7 指称语义、可靠性、充分性与完全抽象

第 3 节至第 6 节构建了一套基于分叉与等待的代数语言理论。代数效应的思想是，可以轻松地将其扩展为一门完整的语言。为此，我们回到第2节中的编程语言，概述如何利用完备表示为其赋予语义。相对于操作语义，该表示在一阶层面上是可靠、适当且完全抽象的。

7.1 解释

7.1.1 概述。我们将编程语言的所有类型 A 解释为函子 $[\![A]\!] \in \mathbf{Set}^{\mathbf{FinRel}}$。其思想是，$[\![A]\!](w)$ 是世界 w 中类型为 A 的值的解释所构成的集合，即 $\vdash^{\mathrm{v}}_w v : A$。类似地，我们将上下文 Γ 解释为函子 $[\![\Gamma]\!] \in \mathbf{Set}^{\mathbf{FinRel}}$，也就是以世界为索引的赋值集合。

我们将世界 w 中的值表达式 $\Gamma \vdash^{\mathrm{v}}_w v : A$ 解释为自然变换 $[\![v]\!] : \mathbf{FinRel}(w, -) \times [\![\Gamma]\!] \to [\![A]\!]$，其中取值于 $\mathbf{Set}^{\mathbf{FinRel}}$，而 $\mathbf{FinRel}(w, -)$ 是在 w 处的可表示函子。如果世界 w 为空类型，这便归结为自然变换 $[\![\Gamma]\!] \to [\![A]\!]$。为了解释计算表达式，我们根据表示在 $\mathbf{Set}^{\mathbf{FinRel}}$ 上构造一个单子 T，并将表达式$\Gamma \vdash^{\mathrm{c}}_w t : A$解释为自然变换 $[\![t]\!] : \mathbf{FinRel}(w, -) \times [\![\Gamma]\!] \to T[\![A]\!]$。

7.1.2 一阶类型的解释。线程 ID 类型的解释 s:

$$[\![\mathsf{tid}]\!] = \mathbf{FinRel}(1, -) \quad i.e.\ [\![\mathsf{tid}]\!](w) \cong \{w' \mid w' \subseteq w\}.$$

因此，我们将世界 w中的（复合）线程 ID 值解释为 w中（非复合）线程标识符的子集。

积类型和和类型的解释采用函子范畴 $\mathbf{Set}^{\mathbf{FinRel}}$中众所周知且典范的范畴结构。回顾一下，函子范畴中的积和余积是逐点计算的。此外，我们与第 3.4 节有着密切联系，因为对于一个

上下文 $(x_1\colon m_1 \dots x_k\colon m_k)$，变量函子是一个类型的解释：

$$[\![\prod_{i=1}^k A_i]\!](p) = \prod_{i=1}^k [\![A_i]\!](p) \quad [\![\sum_{i=1}^k A_i]\!](p) = \biguplus_{i=1}^k [\![A_i]\!](p) \quad V_{x_1:m_1\dots x_k:m_k} \cong [\![\textstyle\sum_{i=1}^k \text{线程 ID}^{m_i}]\!]。 \tag{20}$$

事实上，每个 一阶类型都与这种形式之一同构，因为积对和类型满足分配律 .

7.1.3 单子。我们按照 [44, 45] 的思路，将用于分叉与等待的参数化代数理论扩展为 $\mathbf{Set}^{\mathbf{FinRel}}$ 上的强单子。回想一下（例如见 [38]），普通代数理论（如幺半群）会在集合范畴上诱导出一个单子（如列表）：令 $T(X)$ 为该理论在集合 X 上的自由模型。对于参数化代数理论（如分叉与等待理论），我们可以在范畴 $\mathbf{Set}^{\mathbf{FinRel}}$ 上定义一个强单子 T，令 $T(X)$ 为该理论在函子 $X \in \mathbf{Set}^{\mathbf{FinRel}}$ 上的自由模型。回想第5.3节中，我们为每个上下文 Γ 构造了动态线程理论的一个模型 T_Γ，并证明它是 V_Γ 上的自由模型（定理 5.4）。因此，对于一阶类型（它们都被解释为某个 V_Γ），我们令 $T(V_\Gamma)$ 为函子 T_Γ。

旁注：每个保持筛余极限的 $\mathbf{Set}^{\mathbf{FinRel}}$ 上的强单子都以这种方式来源于一个参数化代数理论，从而给出单子—理论对应。要证明这一对应关系，可以将 [45, §5] 中的结果直接推广：从由不含方程的签名生成的参数化 Lawvere 理论，推广到任意 Lawvere 理论；在我们的情形中，就是$\mathbf{FinRel}^{\mathrm{op}}$。

7.1.4 高阶类型的解释。 众所周知，范畴 $\mathbf{Set}^{\mathbf{FinRel}}$ 是笛卡尔闭的。对于函子 G, H，我们有一个由柯里化同构确定的函子 H^G：给出一个自然变换 $F \times G \to H$，就等价于给出一个自然变换 $F \to H^G$。然后，我们使用单子并利用以下笛卡尔闭结构来解释函数类型 $A \to B$：

$$[\![A \to B]\!] = (T[\![B]\!])^{[\![A]\!]}.$$

7.1.5 项的解释。 我们利用 $T(X)$ 总是参数化代数理论的一个模型这一事实，来解释并发专用原语（分叉、 等待、 停止、 act），如下所示。这些映射有时被称为代数运算 [35] 的“泛型效应”。

$$[\![\underline{\mathsf{fork}}]\!] = \lambda().\mathsf{fork}(\lambda a.\eta(\mathsf{inj}_1(a)), \eta(\mathsf{inj}_2(\,))) : 1 \to T([\![\mathrm{tid}]\!] + 1) \qquad [\![\underline{\mathsf{stop}}]\!] = \lambda().\mathsf{stop} : 1 \to T(0)$$
$$[\![\underline{\mathsf{wait}}]\!] = \lambda a.\mathsf{wait}(a, \eta(\,)) : [\![\mathrm{tid}]\!] \to T(1) \qquad [\![\underline{\mathsf{act}_\sigma}]\!] = \lambda().\mathsf{act}_\sigma : 1 \to T(0)$$

如果省略第（20）个方程，我们也可以将这些视为规范项 s:

$$\underline{\mathsf{fork}}() = \mathsf{fork}(a.x(a), y(\,)) \qquad \underline{\mathsf{wait}}(a) = \mathsf{wait}(a, x) \qquad \underline{\mathsf{stop}}() = \mathsf{stop} \qquad \underline{\mathsf{act}_\sigma}() = \mathsf{act}_\sigma$$

值项和计算项的其余部分，采用的是在带有单子 [28] 的双笛卡尔闭范畴中对按值调用语言进行解释的经典方法。语言构造（和类型、积和函数）与范畴结构（余积、积和笛卡尔闭性）相对应。

$\mathsf{let}\, x = \dots \mathsf{in} \dots$ 的解释使用单子强度和乘法给出。对于一阶类型，这相当于对参数化代数理论进行代换。将这一点详细说明是有意义的。令 $A = \sum_{i=1}^k \mathrm{tid}^{m_i}$ 和 $B = \sum_{i=1}^{k'} \mathrm{tid}^{m'_i}$。考虑程序表达式：

$$\vdash^{\mathbf{c}}_{a_1\dots a_p} t : A \qquad\qquad x : A \vdash^{\mathbf{c}}_{a_1\dots a_p} u : B$$

并且我们解释 $[\![\mathsf{let}\ x = t \mathsf{in}\, u]\!]$。对于 $1 \le i \le k$，令 $\vdash^{\mathbf{c}}_{a_1\dots a_p, c_1\dots c_{m_i}} u_i \mathsf{def}= u[^{\mathsf{inj}_i(\vec{c})}/_x] : B$，其中每个 $\mathsf{inj}_i(\vec{c})$ 的类型为 $\sum_{i=1}^k$ 线程 ID^{m_i}。然后，根据式 (20) 中的第三款以及定理 5.4，我们可以将$[\![t]\!] \in T(V_{x:m_1\dots x_k:m_k})(\,\vec{a})$ 以及每个 $[\![u_i]\!] \in T(V_{x'_1:m'_1\dots x'_{k'}:m'_{k'}})(\,\vec{a}, \vec{c})$ 视为项

$$x_1 : m_1, \dots, x_k : m_k \mid a_1, \dots, a_p \vdash \bar{t} \qquad\qquad x'_1 : m'_1, \dots, x'_{k'} : m'_{k'} \mid a_1, \dots, a_p, c_1, \dots, c_{m_i} \vdash \bar{u}_i$$

在参数化代数理论中。于是，$\vdash^{\mathrm{c}}_{a_1 \dots a_p} \mathsf{let}\ x = t \,\mathsf{in}\, u \colon B \,\mathsf{in}\, T(V_{m'_1 \dots m'_{k'}})(\vec{a})$ 的解释相当于参数化代数理论中如下经代入的项：

$$x'_1 : m'_1, \dots, x'_k : m'_{k'} \mid a_1, \dots, a_p \vdash \bar{t}[\vec{u}_i/x_i]$$

例如，$[\![\underline{\mathsf{perform}_\sigma()}]\!] \in T(1)(0)$ 是左侧程序 $\mathsf{perform}_\sigma()$ 和右侧参数化代数理论中的项的语义：

$$\vdash^{\mathrm{c}}_{\emptyset} \mathsf{let}\ z = \underline{\mathsf{fork}}() \,\mathsf{in}\, \mathsf{case}\ z\ \mathsf{of}\ \begin{Bmatrix} \mathsf{inj}_1(a) \Rightarrow \underline{\mathsf{wait}}(a), \\ \mathsf{inj}_2() \Rightarrow \underline{\mathsf{act}_\sigma}(), \end{Bmatrix} : 1 \qquad x : 0 \mid \cdot \vdash \mathsf{fork}(a.\mathsf{wait}(a, x), \mathsf{act}_\sigma).$$

7.2 充分性、上下文等价、可靠性与完全抽象

我们证明一个充分性版本。通常，充分性指出，在基类型上，如果某个项的指称等于某个值的指称，那么该项就会归约为该值。我们利用充分性证明，指称相等是证明上下文等价的一种可靠方法。对于一阶类型的项，我们利用定理 6.1中的完备性结果证明其逆命题（一个部分完全抽象结果）。Plotkin 和 Power 已经证明了关于代数效应的一般充分性结果 [37]以及 Kavvos [21]，但这些结果无法直接应用于我们的情形，因为我们使用参数化签名来表达效应，并在函子范畴中对其建模。

引理 7.1（充分性）。对于 $\vdash^{\mathrm{c}}_{\emptyset} t : 0$，我们有 $[a]t \Downarrow [\![t]\!]$。

证明概要。我们分三步证明这一点。

(1) 我们将项解释扩展到 $[\![t]\!]$良构配置 $[\![C]\!]$。(2) 我们证明归约关系具有可靠性：归约会保持语义解释。例如，若 $C \longrightarrow C'$，则 $[\![C]\!] = [\![C']\!]$。这是一项直接的归纳证明，但陈述本身较为微妙，需要累积动作标签。(3) 我们从 $\langle [a]t \rangle$选取一条归约序列；根据命题 2.7，序列的选择无关紧要，而且该序列会终止。终结配置中只剩下等待关系 $\preceq$ 以及所有已停止的线程。加上累积的动作标签后，这个标记偏序集是 $[\![t]\!]$，因为归约会保持语义解释。 □

定义 7.2。令 Γ为一个类型上下文，A为一个类型。对于 Γ，A，程序上下文 $\mathsf{C}[-]$是一个类型为 0 且带有一个类型为 A的洞的程序。因此，如果 $\Gamma \vdash^{\mathrm{c}}_{\emptyset} t \colon A$，则 $\vdash^{\mathrm{c}}_{\emptyset} \mathsf{C}[t] \colon 0$。

两个程序 $\Gamma \vdash^{\mathrm{c}}_{\emptyset} t,\ u \colon A$称为上下文等价，记作 $t \,\mathrm{ctx}{=}\, u$，如果对于每个 $(\Gamma,\ A)$ 上下文 $\mathsf{C}[-]$，令 $(\cong)$ 表示标记偏序集的同构，则有

$$\langle [a]\mathsf{C}[t] \rangle \Downarrow (P, \ell_P) \ \&\ \langle [a]\mathsf{C}[u] \rangle \Downarrow (Q, \ell_Q) \implies (P, \ell_P) \cong (Q, \ell_Q)$$

根据 命题 2.7，$(P,\ \ell_P)$和 $(Q,\ \ell_Q)$分别由 $\mathsf{C}[t]$和 $\mathsf{C}[u]$唯一确定。

命题 7.3（可靠性）。假设 $\Gamma \vdash^{\mathrm{c}}_{\emptyset} t,\ u \colon A$。如果 $[\![t]\!] = [\![u]\!]$，则 $t \,\mathrm{ctx}{=}\, u$。

证明。我们如下由引理7.1 推出该结果。我们考虑任意类型上下文中的任意两个项 $\Gamma \vdash^{\mathrm{c}}_{\emptyset} t,\ u \colon A$，以及任意（$\Gamma,\ A$）-上下文$\mathsf{C}[-]$。假设 $[\![t]\!] = [\![u]\!]$。由引理7.1，$[a]\mathsf{C}[t] \Downarrow [\![\mathsf{C}[t]]\!]$，并且$[a]\mathsf{C}[u] \Downarrow [\![\mathsf{C}\ [u]]\!]$。由于指称语义是复合的，且 $[\![t]\!] = [\![u]\!]$，故亦有 $[\![\mathsf{C}[t]]\!] = [\![\mathsf{C}[u]]\!]$。因此 $t \,\mathrm{ctx}{=}\, u$。

□

特别　　地，标准的 β/η 定律是可靠的，图5中的所有方程亦然，例如（13）：

$$\underline{\mathsf{wait}}(b); \underline{\mathsf{fork}}() \ = \ \mathsf{let}\, x = \underline{\mathsf{fork}}() \,\mathsf{in}\, \underline{\mathsf{wait}}(b); \mathsf{return}\, x.$$

定理 7.4（一阶完全抽象）。假设 $a_1 \colon A_1, \dots, a_p \colon A_p \vdash^{\mathrm{c}}_{\emptyset} t,u \colon B$、$A_1 \dots A_p$和 B都是一阶的（没有函数类型）。那么 $[\![t]\!] = [\![u]\!]$ 当且仅当 $t \,\mathrm{ctx}{=}\, u$。

证明。从左到右由定理7.3可得。从右到左，我们先考虑 $p = 0$ 和 $B = 0$的情况。与空上下文的上下文等价尤其结合引理7.1，可得 $[\![t]\!] = [\![u]\!]$。

接下来考虑 $A_1 = A_2 = \ldots A_p = \mathsf{tid}$ 和 $B = \sum_{i=1}^{k} \mathsf{tid}^{m_i}$ 的情况。假设 $t \overset{\text{ctx}}{=} u$。根据（20），$[\![t]\!], [\![u]\!] \in T(V_{x_1:m_1 \ldots x_k:m_k})(p)$，也就是说，$t$ 和 u 直接在参数化代数理论中进行解释。我们必须证明它们相等。根据定理 6.1，只需证明对于所有代数上下文 $\mathsf{C}[-]$ 和代数替换 γ，$\mathsf{C}[[\![t]\!]][\gamma] = \mathsf{C}[[\![u]\!]][\gamma]$。我们通过将 $\mathsf{C}[-]$ 和 γ 转换为程序上下文（一阶的“完全可定义性”）来推导这一点，从而可以使用上下文等价 $t \overset{\text{ctx}}{=} u$。

首先，我们注意到，借助泛型效应 $\mathsf{act}_\sigma = \mathsf{act}_\sigma()$、$\mathsf{stop} = \mathsf{stop}()$ 和

$$\mathsf{fork}(t,u) = \mathsf{case}\ \underline{\mathsf{fork}}()\ \mathsf{of}\ \{\mathsf{inj}_1(a) \Rightarrow t,\ \mathsf{inj}_2(()) \Rightarrow u\} \qquad \mathsf{wait}(a,t) = \underline{\mathsf{wait}}(a); t \tag{21}$$

我们利用这一点，将代数上下文 $C[-]$ 转换为绑定自由变量的程序上下文$a_1 \cdots a_p$。此外，每个“代入项” $\gamma(x_i)$都不含计算变量，因此也可以视为一个在 (0 之下、类型为 21 的程序。现在我们定义计算程序表达式

$$t[\gamma] \overset{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{inj}_i(\vec{a}) \Rightarrow \gamma(x_i)\}_{i=1}^{k}$$

使得 $[\![t[\gamma]]\!] = [\![t]\!][\gamma] \in T(0)(p)$。因此 $C[[\![t]\!]][\gamma] = [\![C[t][\gamma]]\!] = [\![C[u][\gamma]]\!] = C[[\![u]\!]][\gamma]$如所需。最后，我们利用 β/η 和类型的法则，以及每种一阶类型都可定义地同构于形如 (20) □ 的类型这一事实，推出完整结果。

8 进一步的相关工作、未来工作与总结

在总结之 前，我们将讨论一些其他相关工作，以及本研究所开启的未来方向 .

8.1 进一步的相关工作

并发的代数效应。 如第 1 节简要讨论的那样，代数理论已被用于公理化过程演算的特性，其中包括采用代数效应的风格进行公理化。这包括对名称创建以及通过通道进行名称通信的代数效应分析，见π-演算 [43], , 以及使用代数效应和处理器对 CSP 的动作、选择和隐藏等特性的处理 [49]。从编程语言的角度来看，Abadi 和 Plotkin [2] 以及 Dvir 等人 [10, 11] 使用代数效应对非确定性和全局共享状态并存时的并发进行了建模。如第 1.4 节所述，我们的工作与这些先前工作的不同之处在于，程序的并行组合（即分叉）是等式公理化中的一项运算，而在先前的工作中，它是在代数效应的表示之上定义的。使这一点成为可能的关键在于，我们将线程 ID 视为原语，并使用参数化代数理论的框架来刻画线程创建。

迹语义。 Brookes 的开创性工作 [8]对一种带有全局共享状态的抢占式并发编程语言进行建模。程序表示为迹的闭集；这些迹表示一种协议，涉及程序及其环境对内存所做的更改。这种语义形式在变化和扩展下具有稳健性 [5, 48, 50], , 包括弱内存模型的变化 [10, 18]。Dvir 等人 [11] 为 Brookes 式的迹给出了一种双排序 代数理论。其表示定理在限定为其中一种排序时，恢复出 Brookes 的单子。有趣的是，在限定为另一种排序时，同一表示也恢复出 Abadi 和 Plotkin 的 [2] 共享状态合作式并发编程单子。Dvir 等人和 Abadi、Plotkin 的表述都预设非确定性选择是一种代数运算。相比之下，

在我们的参数化代数理论中，非确定性行为是从线程分叉这一更原始的行为中涌现出来的。

并发效应处理器。效应处理器源于对代数效应的理论研究，[39] 旨在支持非代数效应，例如用于捕获异常的运算。它们很快被采用为支持效应模块化编程的一项通用功能，[20]，并成为当前在 OCaml 5 [42]和 WebAssembly 中实现并发的核心机制。[32]它们为一系列不同的并发效应提供了基础，例如演员模型、异步/等待、协程、生成元和绿色线程。然而，使用效应处理器进行编程的实践与既有理论存在显著偏离：我们尚不知道如何使用任何形式的等式公理化来规定这些效应的语义，更不用说将其作为代数效应了。

Hazel 是一种用于效应处理器的分离逻辑 [9]，构建于并发分离逻辑 Iris 之上，并在 Rocq 证明助手中实现。Hazel 为推理关于由效应处理器实现的并发效应提供了强大的框架，但它与代数效应理论所提供的基础等式推理相去甚远，对于所定义效应的语义也几乎没有提供洞见。相比之下，我们的工作将一种特定的并发效应（动态线程）刻画为一种代数效应（具体而言，是参数化代数效应），并使之对应于一个自然的指称模型。未来的工作可以调整并扩展我们的方法，以支持广泛的不同并发效应，或将其与效应处理器及编程实践联系起来。

8.2 未来工作

代数理论框架允许对效应进行模块化组合 [17]。我们可以利用这一点，将基于动态线程的并发与其他效应相结合，例如线程间共享的全局状态与局部状态 [38]，并对线程的概率调度进行建模。

我们采用标记偏序集（部分有序多重集），它们是研究真实并发时的标准工具，例如 [40]，，并将其作为操作语义中的观察概念。我们希望未来能将我们的指称模型与基于交错轨迹的操作语义联系起来；后者在进程演算中更为常见，例如 [26]。

分叉与等待的语义变体不胜枚举，例如等待一个线程及其所有后代线程结束，或限制同一时刻可以存在的线程数量。另一种扩展是允许线程以一个值而非空类型结束，因此等待运算会将该值返回给父线程。这种扩展是线程间通信的一种抽象形式。本文聚焦于类 POSIX 线程的一个最小化理想片段，因此未来还可以尝试对许多线程特性进行建模，例如允许线程获知自己的标识符，或加入其他线程同步机制。

8.3 结语

我们利用参数化代数理论的框架，通过将线程 ID 视为抽象参数，研究了线程动态创建的语义。在第4 节中，我们给出了一个代数理论，用以公理化分叉和等待线程等运算。在第5 节中，我们给出了该理论中项的一种语法无关刻画（定理5.4），其基于标记偏序集的一种扩展，而标记偏序集在并发理论中已得到充分确立。随后，我们在第6节中表明，就普通标记偏序集的相等而言，我们的理论在某种意义上是完备的。

在第 2 节中，我们介绍了一种简单的并发编程语言及其操作语义。为了对这门语言进行指称建模，我们在第 7 节中使用了动态线程代数理论，以及代数理论与单子语义之间的联系。我们证明，该指称语义在一阶情况下是适当、可靠且完全抽象的。

总而言之，我们的简单语言表明，代数效应理论可直接应用于并发原语，且采用这种代数视角大有裨益。

致谢

Ohad Kammar、Jack Liell-Cock 和 Sam Staton 的研究部分得到了英国皇家学会大学研究奖学金、欧洲研究委员会 BLAST 项目、美国空军科学研究办公室（资助编号 FA9550-21-1-0038）、克拉伦登奖学金以及 ARIA 安全人工智能项目资助。Sam Lindley 和 Cristina Matache 得到了英国研究与创新署未来领袖奖学金“面向效应处理器的编程”（MR/T043830/1 和 MR/Z000351/1）以及华为爱丁堡联合实验室项目“EPOCH：面向能力硬件的效应式编程”的支持。为实现开放获取，作者已对本次投稿所产生的任何作者接受稿版本采用知识共享署名（CC BY）许可协议。

我们感谢匿名评审人以及许多人的宝贵讨论和有益建议，其中包括 Rob van Glabbeek 和 Sean Moss。

参考文献

[1] 2024。信息技术—可移植操作系统接口（POSIX®）基础规范，第 8 期。 https://pubs.opengroup.org/onlinepubs/9799919799/ 2024 年批准。由 IEEE 和开放组出版。[2] Martín Abadi 和 Gordon D. Plotkin。2010。协作线程模型。《逻辑方法与计算机科学》6, 4 (2010)。doi:10.2168/LMCS-6(4:2)2010[3] Rajeev Alur、Caleb Stanford 和 Christopher Watson。2023。串并联图的鲁棒理论。《ACM 编程语言汇刊》 7, 2023 37 10 1145 3571230 POPL ( )，文章 。doi: 。/ John C. Baez、[4] 2018 Brandon Coya 和 Franciscus Rebro。。网络理论中的 PROP。《范畴论及其应用》33 (2018), 727–783。 http://www.tac.mta.ca/tac/volumes/33/25/33-25.pdf [5] Nick Benton、Martin Hofmann 和 Vivek Nigam。2016。并发程序的效应依赖变换。载于 PPDP。ACM。doi:10.1145/2967973.2968602[6] Jan A. Bergstra、Alban Ponse 和 Scott A. Smolka（编）。2001。《过程代数手册》。Elsevier。doi:10.1016/B978- 0-444-82830-9.X5017-6[7] Filippo Bonchi、Paweł Sobociński 和 Fabio Zanasi。2017。信号流图演算 I：流上的线性关系。《信息与计算》（ 2017 2 29 10 1016 2016 03 002 ），– 。doi: 。/j.ic. . . Stephen D. Brookes。。共享变量并行语言[8] 1996 的完全抽象。《信息与计算》127, 2 (1996)。doi:10.1006/inco.1996.0056[9] Paulo Emílio de Vilhena 和 François Pottier。2021。效应处理器的分离逻辑。《ACM 编程语言汇刊》5, POPL (2021), 1–28。doi:10.1145/3434314[10] Yotam Dvir、Ohad Kammar 和 Ori Lahav。2024。释放/获取并发的指称方法。载于欧洲编程语言及系统研讨会，欧洲理论与实践软件科学会议（LNCS，第 *14577* 10 1007 978 3 031 57267 8_5 卷）。施普林格。doi: 。/ - - - - Yotam Dvir、Ohad[11] 2025 Kammar、Ori Lahav 和 Gordon D. Plotkin。。Brookes 共享状态指称语义的双排序代数分解。（ 2025 10 1007 978 3 031 90897 2_18 ）。doi: 。/ - - - - Uli Fahrenberg、Christian[12] Johansen、Georg Struth 和 Krzysztof Ziemiański。2022。作为并发模型的带接口偏序集。《信息与计算》 285 2022 10 1016 2022 104914 ( )。doi: 。/j.ic. . Marcelo Fiore 和 Marco Devesas[13] 2013 Campos。。有向无环图代数。施普林格柏林海德堡，柏林，海德堡， 37 51 10 1007 978 3 642 38164 5_4 – 。[14] 1988 doi: 。/ - - - - Jay L. Gischer。。pomset 等式理论。《理论计算机科学》 61 1988 10 1016 0304 3975 88 90124 7 ( )。doi: 。/ - ( ) -[15] 2011 Tony Hoare、Bernhard Möller、Georg Struth 和 Ian Wehrman。。并发 Kleene 代数及其基础。《逻辑与代数程序杂志》 80, 6 2011 10 1016 2011 04 005 ( )。doi: 。/j.jlap. . . Tony Hoare 和 Stephan van Staden。。程序[16] 2014 设计定律统一过程演算。《科学计算机编程》85 (2014 102 114 10 1016 2013 08 012), – 。doi: 。/j.scico. . . Martin Hyland、Gordon D. Plotkin 和 John Power。2006[17] 。组合效应：和与张量。《理论计算机科学》 357, 1 2006 70 99 10 1016 2006 03 013 (7 月)，– 。doi: 。/j.tcs. . . Radha Jagadeesan、Gustavo Petri 和 James[18] Riely。2012。Brookes 几乎是宽松的！载于 FOSSACS，欧洲理论与实践软件科学会议（LNCS，第 *7213* 180 194 10 1007 978 3 642 28729 9_12 卷），Lars Birkedal（编）。施普林格，– 。[19] doi: 。/ - - - - Ralf Jung、Robbert Krebbers、Jacques-Henri Jourdan、Ales Bizjak、Lars Birkedal 和 Derek Dreyer。2018。 Iris 从基础构建：高阶并发分离逻辑的模块化基础。《函数式编程杂志》28 (2018), e20。doi:10.1017/S0956796818000151

[20] Ohad Kammar、Sam Lindley 和 Nicolas Oury。2013。处理器的实际应用。载于 ICFP'13。ACM，145–158。doi:10.1145/ 2500365.2500590[21] G. A. Kavvos。2025。重新审视代数效应的充分性。《ACM 编程语言汇刊》9, OOPSLA1 (2025), 927–955。doi:10.1145/3720457[22] Dénes König。1926。关于集合的多值对应。《基本数学》8 (1926), 114–134。doi:10.4064/fm-8-1-114-134[23] Paul Blain Levy、John Power 和 Hayo Thielecke。2003。按值调用编程语言中的环境建模。《信息与计算》

185, 2 2003 182 210 10 1016 0890 5401 03 00088 9 ( ), – 。doi: 。/S - ( ) - Martin

[24] 2008
Markl。。多元运算与 PROPs。《代数手册》，第 卷。North-Holland，87–140。doi:10.1016/S1570- 5

7954 07 05002 4 ( ) - T.A. McKee。。串并联图：一种逻辑方法。《图论杂志》7, 2 (1983), 229–236。doi:10.

[25] 1983
1002 3190070206 /jgt。Robin Milner。。通信与并发。Prentice Hall。https://dl.acm.org/doi/book/

[26] 1989

10 5555 534666 . /

[27] 2014
Samuel Mimram。。呈现有限偏序集。载于 TERMGRAPH 。doi: 。/EPTCS. . *2014* 10 4204 183

1

[28] 1991 1991 10 Eugenio Moggi。。计

算的概念与单子。《信息与计算》（）。doi: 。1016/0890-5401(91) 90052-4[29] Madhavan Mukund 和 Mogens Nielsen。1992。CCS、位置与异步转换系统。载于 Proc. FSTTCS。doi:10.1007/3-540-56287-7_116[30] Mogens Nielsen、Gordon D. Plotkin 和 Glynn Winskel。1981。Petri网、事件结构与域，第一部分。《理论计算机科学》

13 1981 10 1016 0304 3975 81 90112 2 ( )。doi: 。/ - ( ) - Frank J. Oles。。类型代数、函子

[31] 1983
范畴与块结构。载于《语义学中的代数方法》。doi: 10 7146 12 156 7430 . /dpb.v i . Luna Phipps-Costin、

[32]
Andreas Rossberg、Arjun Guha、Daan Leijen、Daniel Hillerström、K. C. Sivaramakrishnan、Matija Pret nar 和 Sam Lindley。2023。使用效应处理器延续 WebAssembly。《ACM 编程语言汇刊》7, OOPSLA2 (2023), 460–485。doi:10.1145/3622814[33] Andrew M Pitts。2013。《名义集合：计算机科学中的名称与对称性》。CUP。https://dl.acm.org/doi/10.5555/ 2512979[34] Gordon Plotkin。2012。并发与效应的代数理论。载于 Proc. CO NCUR 2012。doi:10.1007/978-3- 642-32940-1 2_[35] Gordon Plotkin 和 John Power。2003。代数操作与泛型效应。《应用范畴结构》11, 1 (2003), 69–94。doi:10.1023/A:1023064908962[36] Gordon Plotkin 和 Vaughan Pratt。1996。团队能够观察到部分有序多重集。载于 OMIV '96：DIMACS 验证偏序方法研讨会论文集。https://homepages.inf.ed.ac.uk/gdp/publications/Teams.pdf

[37] 2001 *2001* 10 1007 3 540 45315 6_1

[38] 2002 *2002*

Gordon D. Plotkin 和 John Power。。代数效应的充分性。载于 FOSSACS。doi: 。/ - - - Gordon D. Plotkin 和 John Power。。计算概念决定单子。载于 FOSSACS。施普林格， 342 356 10 1007 3 540 45931 6_24 – 。

[39] 2013
https://doi.org/ 。/ - - - Gordon D. Plotkin 和 Matija Pretnar。。处理代数效应。《逻辑方法与计算机科学》9, 4 (2013)。doi:10.2168/LMCS-9(4:23)2013[40] Vaughan R. Pratt。1986。用偏序建模并发。《国际并行编程杂志》15, 1 (1986), 33–71。doi:10.1007/BF01379149[41] Davide Sangiorgi 和 David Walker。2001。《π演算：移动进程理论》。剑桥大学出版社。https://dl.acm.org/doi/ 10 5555 559050 . / K. C. Sivaramakrishnan、

[42]
Stephen Dolan、Leo White、Tom Kelly、Sadiq Jaffer 和 Anil Madhavapeddy。2021。将效应处理器移植到 OCaml。载于 PLDI '21。ACM，206–221。doi:10.1145/3453483.3454039[43] Ian Stark。2008。π演算的自由代数模型。《理论计算机科学》390, 2-3 (2008), 248–270。doi:10.1016/J. TCS.2007.09.024[44] Sam Staton。2013。谓词逻辑的代数表示——（扩展摘要）。载于 FOSSACS 2013。doi:10.1007/978- 3-642-37075-5 26_

[45] 2013 *2013* 10 1109 2013 58 Sa

[46] 2015 *2015* 10 1145

m Staton。。计算效应实例：代数视角。载于 LICS。doi: 。/LICS. . Sam Staton。。代数效应、线性性与量子编程语言。载于 POPL。doi: 。/ 2676726.2676999[47] W. W. Tait。1967。有限类型泛函的内涵解释 I。《符号逻辑杂志》32, 2 (1967), 198–212。doi:10.2307/2271658[48] Aaron Joseph Turon 和 Mitchell Wand。2011。用于精化并发对象的分离逻辑。载于 POPL。ACM。doi:10.1145/1926385.1926415[49] Rob J. van Glabbeek 和 Gordon D. Plotkin。2010。论 CSP 与效应的代数理论。载于《纪念 C. A. R. Hoare、A. W. Roscoe、Clifford B. Jones 和 Kenneth R. Wood 的学术贡献》。施普林格，333–369。

doi:10.1007/978-1-84882-912-1_15 [50] Qiwen Xu、Willem P. de Roever 和 Jifeng He。1997。用于验证共享变量并发程序的依赖-保证方法。*FormalAspects Comput.*9, 2 (1997)。doi:10.1007/BF01211617

收稿日期：2025-07-10；录用日期：2025-11-06