结构化并发程序的约简

Namratha Gangamreddypalli[1], Constantin Enea[1], 和 Shaz Qadeer[2]

[1] LIX、巴黎综合理工学院、法国国家科学研究中心和巴黎综合理工学院，法国{namratha、cenea}@lix.polytechnique.fr[2] Microsoft shaz.qadeer@gmail.com

arXiv:2601.13341v1 [cs.PL] 19 Jan 2026

摘要。 基于 Lipton迁移操作的交换性推理是验证并发程序的一项强大技术。其核心思想是定义一种程序变换，在可交换操作的重新排序意义下保持初始交错执行集合的一个子集。如何将交换性推理扩展到软件系统中常用的过程和并行组合等特性，仍然是一项重大挑战。

在本文中，我们提出了一种针对结构化并发程序的新型约简技术，将两项关键进展统一起来。首先，我们提出了一种归约策略，能够以保持正确性的方式将并行组合替换为顺序组合。其次，我们推广了 Lipton 归约，使其支持包含（可能递归的）过程调用的原子区段。关键在于，这两种基础策略可以任意组合，从而极大拓展了基于归约的推理的适用范围和灵活性。我们在 Civl 中实现了该技术，并通过多个具有挑战性的案例研究展示了其有效性，包括快照对象、容错且线性化的寄存器、FLASH 缓存一致性协议，以及两阶段提交的一个非平凡变体。

1 引言

交换性推理是验证并发程序的一种强大技术。这种方法源于这样一种观察：某些并发执行的语句对可以重新排序而不影响程序行为，也就是说，这些语句可以交换。仅在可交换语句的顺序要求上有所不同的交错执行（即并发执行序列）被视为等价。因此，只需验证每个等价类中的一个代表性交错执行是否正确即可。这些技术也称为化简 技术，因为它们将推理范围缩小到更少的代表性交错执行。

在并发程序的静态验证中，利用交换性推理的一种标准方法是通过对原始程序进行语法变换来表示约简程序。随后，在变换后的程序上证明正确性，并利用化简论证将验证结果推广到原始程序。Lipton [34] 引入了原子区段，作为表示此类变换的一种简单方法。此后，原子

区段被广泛用于描述无干扰，并简化并发程序的推理 [19,17,11,22,15]。

本文聚焦于将交换性推理应用于并发程序的演绎验证。将交换性推理应用于真实世界的程序颇具挑战。软件系统通常使用过程来组织代码并扩展软件工程的规模。此外，并发系统以性能为导向，往往会并行启动多个任务，并在任务完成时收集结果。现有方法并未充分处理这些特性：过程和动态并发。Lipton [34] 仅处理具有有界线程和有限大小原子区段的并发程序。QED [11] 和 Civl [22]能够处理包含循环的原子区段，但无法处理过程调用（除非事先将其内联）或并行组合。Kragl 等人 [29] 提出了一种程序转换，通过证明被调用过程可概括为一个单独的原子动作，且该动作可交换到任何其他程序动作的左侧（即所谓的左移动操作），从而同步异步过程调用。虽然异步调用可视为并行组合的一种受限形式，但这种模型对我们的目的而言不够通用（参见第 9 节）。

我们为结构化并发程序提出了一种新颖的约简技术，统一了两项关键进展。首先，我们提出一种约简策略，可可靠地将并行组合替换为顺序组合，解决了一个与以往工作所探索的引入原子区段这一维度正交的问题。其次，我们将 Lipton 归约推广到支持包含（可能递归的）过程调用的原子区段。至关重要的是，这两种基础策略可以任意组合，大幅拓展了基于约简的推理的适用范围和灵活性。

我们的约简技术基于可移动操作或可交换语句 [34,11,22]这一概念。引入原子区段的可靠性取决于能否证明该区段由一系列右移动操作、任意语句以及一系列左移动操作依次组成。右移动操作是指在程序中可越过任意其他语句向右移动的语句，而左移动操作则可向左移动；这种说法设想时间从左向右流动。重要的是，移动语句分类取决于所考虑的动作集合：某条语句是右移动操作还是左移动操作，取决于它与程序中其他动作的交互方式。对可移动操作的这种直观理解具有迷惑性；要给出精确定义并非易事，因为定义必须考虑可能失败或具有非确定性的语句。

为处理结构化代码，我们通过一个分析过程体的类型系统 [19]，将可移动操作的概念扩展到过程。这使我们能够可靠地引入包含递归过程调用的原子区段（与之前一样，可靠性取决于确保右移动操作与左移动操作的顺序执行具有良好结构）。例如，如果某个过程 Q 的过程体仅由右移动器语句组成——包括嵌套的过程调用，这些调用也会递归地进行类型判定——那么 Q 就会被归类为右移动过程。左移动过程也有类似的分类，但还额外要求它们必须终止

在单独执行时。这一终止条件确保失败行为得以保留：否则，一个不终止的过程可能会不可靠地消除潜在失败。值得注意的是，尽管我们讨论的是并发，但这一条件仅取决于过程单独执行时的行为，这是我们框架中一个出人意料且实用的特点。

我们技术的第二个关键贡献，是能够可靠地将一种并行构造转换为其各自代码块的顺序组合：该构造会生成任意数量的线程，并立即将它们合并。这一转换同样基于可移动操作的概念，并通过迭代过程实现：先对左移动者代码块排序，再对右移动者代码块排序。

我们技术的两项贡献整合在一个统一框架中，该框架支持过程的任意顺序组合和并行组合。此外，这两项贡献相辅相成：转换并行构造可能会产生左移动或右移动过程，从而进一步促成原子区段的引入。我们的技术在一种核心编程语言中得以形式化，其中两条归约原则通过特定关键字调用，而类型系统则保证它们能够正确且可靠地组合。我们指出，我们对可移动操作的定义比 Civl 中所用的更一般；也就是说，某些语句在我们的定义下可交换，但在 Civl 的定义下不可交换。此外，我们的工作首次对 Civl 的归约理论的可靠性给出了形式化证明。这项证明在确定足以确保可靠性的最一般可交换性形式方面发挥了关键作用（第 4 节）。

这一化简框架的可靠性基于一些非平凡的论证。例如，化简是在代码层面定义的，这意味着即使是单个化简，在一次执行中也可能应用无界次（同一过程可能被多次调用）。因此，需要制定一种非平凡的策略，根据步骤的可交换性来重排执行中的步骤。此外，使用左移动操作或右移动操作的侧条件并不对称，这可能看起来有些违背直觉。例如，在并行归约中，不允许右移动操作失败；而左移动过程在无干扰地运行时必须终止——值得注意的是，后一条件涉及顺序执行，尽管它应用于并发程序。此外，右移动操作不得失败这一限制仅适用于并行归约，而不适用于顺序归约。

我们将该技术实现为 Civl 的扩展，同时保留了与其现有功能的兼容性。为了评估其有效性，我们将该实现应用于一系列具有挑战性的案例研究：快照对象的并行实现 [3],ABD 寄存器 [4] ，它通过消息传递模拟共享内存；FLASH 缓存一致性协议 [31], ；以及一种非平凡的两阶段提交协议变体。这些示例涵盖了多个不同领域，包括并发对象、分布式协议和硬件缓存一致性，展现了我们方法的广泛适用性。特别是，前两个案例研究是并发对象，我们证明了它们具有线性化性质 [23]。众所周知，要证明这些对象的线性化性质颇具挑

```
procedure scan() returns (snapshot: [int]StampVal) {
  var r1: [int]StampVal;
  var r2: [int]StampVal;
  while (true) {
    (call r1[1] := read(1)) par
    (call r1[2] := read(2));
    (call r2[1] := read(1)) par
    (call r2[2] := read(2));
    if (r1 == r2) {
      snapshot := r1;
      return;
    }
  }
}
```

```
datatype StampVal {
  StampVal(ts: int, value: Value)
}

var mem: [int]StampVal;

action read (i: int) returns (v:
     StampVal) {
  v := mem[i];
}

action write(i: int, v: Value) {
  mem[i] := StampVal(mem[i]->ts +
     1, v);
}

action scan_spec() returns
     (snapshot: [int]StampVal) {
  assume (snapshot := mem);
}
```

图 1：快照对象。扫描过程连续执行两次收集操作，也就是说，它并行读取整个内存两次。如果两次收集得到的结果相同，该过程就返回。否则，它会重新开始。

战，因为无法通过所谓的固定线性化点来证明，也就是说，方法调用的效果无法映射到方法体中的某条固定语句的执行，而且证明还需要预言变量 [1]。

对于我们的案例研究而言，化简不可或缺；每项研究都涉及由无界数量、动态创建的并发任务对共享状态进行细粒度访问。旧版 Civl 无法处理这些案例研究，因为化简仅适用于只包含动作（不含过程调用）的顺序代码片段，而且不具备并行化简的概念。据我们所知，没有其他基于化简的证明技术能够处理这些案例研究。如果不使用化简，仅依靠归纳不变式进行证明会复杂得多：所需的不变式规模庞大且难以构造。

2 概述

我们通过一个并发快照对象 [3] 的实现来展示我们的化简证明技术，该对象提供两种方法：`write(i,v)` 将值`v` 写入内存单元`i`，而`scan()` 则返回整个内存的快照。我们假设内存由一个数组表示。任意数量的线程都可以并发调用这些方法。我们首先介绍该实现以及我们试图证明的规范，然后详细说明如何应用我们的化简证明技术。

2.1 一个并发快照对象

实现。 图 1 列出了快照对象的代码（稍后将解释扫描规约 ）。每个内存单元都保存一个带时间戳的值（一个值及其整数时间戳）。为简单起见，我们考虑只有两个单元的内存。任意大小的情形将在第2节 2.4 中讨论。

代码采用一种支持常规过程调用和并行组合的编程语言，其中对共享内存的每次访问都封装

```
procedure scan() returns (snapshot: [int]StampVal) {
  var r1: [int]StampVal;
  var r2: [int]StampVal;
  while (true) {
    seq-reduce {
      par-reduce {
        (call r1[1] := read_f(1)) par
        (call r1[2] := read_f(2))
      }
      par-reduce {
        (call r2[1] := read_s(1)) par
        (call r2[2] := read_s(2))
      }
      if (r1 == r2) {
        snapshot := r1;
        return;
      }
    }
  }
}
```

```
right action read_f(i: int) returns
     (out: StampVal) {
  var k: int;
  var v: Value;
  if (*) {
    assume k < mem[i]->ts;
    out := StampVal(v, k);
  } else {
    out := mem[i];
  }
}

left action read_s(i: int) returns
     (out: StampVal) {
  var k: int;
  var v: Value;
  if (*) {
    assume k > mem[i]->ts;
    out := StampVal(v, k);
  } else {
    out := mem[i];}
}
```

图 2：抽象扫描。与原始版本相比，两个内存读取分别调用抽象动作 read f 和 read s。在这些动作中，$*$ 是非确定性选择，局部变量最初被赋予任意值。注释 seq-reduce 和 par-reduce 与我们的约简技术有关。

在一个所谓的动作中。假设动作在单个不可分割步骤中原子执行。在这种情况下，我们有两个动作：`read(i)` 读取`i`号内存单元，而`write(i,v)` 则用值`v` 更新`i`号内存单元，并将其时间戳增加 1。

过程 `scan` 由一个“自旋”循环组成：当对整个内存连续进行两次读取并得到相同结果时，循环便退出。对内存的读取采用并行化方式，每个内存单元都由不同的线程读取。这里使用关键字 `par` ，将对动作`read` 的两次调用隔开（动作的调用方式与过程相同）。对于两个语句 s和 s' ， s `par` s' 的含义是， s和 s' 在两个不同的线程中执行，并在执行下一条语句之前合并。这确保了对整个内存的两次读取不会在时间上重叠。对于熟悉相关知识的读者来说，这对应于分叉-汇合模型。

过程 `write(i,v)` 省略了；它只需调用同名动作。

规范：线性化。我们的目标是证明该对象是可线性化的，也就是说，每次对 `scan` 或`write` 的并发调用，似乎都会在调用与返回之间的某个时刻瞬间生效。换言之，每次包含多个调用的并发执行，都对应一个线性化序列——一个有效的调用序列，其中每次`scan`都会返回由此前所有`write` 操作所产生的内存状态。

前述的 `scan` 顺序语义由图 1 中的动作 `scan spec` 定义，该动作假设返回值等于对内存的一次瞬时读取（请注意，动作在单个不可分割步骤中执行）。线性化可以归结为证明 `scan` 是 `scanspec` 的精化，具体含义将在后文中说明。请注意，`scan spec` 可能

阻塞，而这是合理的，因为线性化本身并不蕴含任何进展概念。

线性化证明。Attiya 等人[5] 的工作表明，任何“未约简”的线性化证明都需要预言变量；所谓“未约简”证明，是指试图为任意数量的调用所产生的每一种可能执行建立线性化序列的证明。线性化等价于并发对象与原子（顺序）规范之间一种标准的轨迹包含 概念；在该规范中，每次调用都执行一个不可分割步骤。轨迹是由调用与返回事件构成的序列，其中记录了输入和值。已知轨迹包含等价于存在由正向模拟关系和反向模拟关系组成的复合关系 [36]。Attiya 等人 [5] 证明，不存在从该快照对象到相应原子（顺序）规范的正向模拟，这意味着必须使用反向模拟（正向模拟对应于使用所谓固定线性化点的证明）。已知在演绎验证的语境中，反向模拟对应于使用预言变量 [1,36]。

接下来，我们介绍一种基于化简技术的证明，该技术避免使用众所周知极具挑战性的预言变量。经过一步抽象后，读取操作将变为可移动操作，并且可以重新排序，形成一个原子区段，直接精化扫描规约。

这超越了以往的化简技术，因为扫描实现将语句的并行组合与顺序组合嵌套在一起。

2.2 使用抽象实现化简

显然，动作读取 和写入 不能交换。为了实现化简，我们为读取引入两种抽象：右移动作抽象读取 f和左移动操作抽象读取 s，它们列于图 2 右侧。一般而言，-动作的抽象会对其对全局状态的影响以及可能返回值的集合进行过近似。

如果读取 抽象能够返回一个时间戳早于内存中时间戳的值，那么它就可以交换到写入操作的右侧；这意味着它在写入之前返回的任何值，在写入之后仍然有效。我们通过一种非确定性选择来引入这种行为：读取 f 可以返回内存中的带时间戳的值，也可以返回任意带时间戳的值，但其时间戳必须严格小于内存中的时间戳。左移动操作抽象读取 s 与此非常相似，区别在于“任意”情况下返回值的时间戳必须严格大于内存中的时间戳。我们指出，设计此类抽象需要创造力，这与任何其他演绎证明系统一样。其优势在于，这些抽象能让证明更加简洁易用，既更精炼，也不那么繁琐。此外，移动操作抽象的可靠性是一种局部性质，因为它只涉及成对的动作；由此产生的化简也大大简化了后续推理。

图 2 列出了 扫描过程的一个抽象，该过程在第一次和第二次读取内存期间分别调用抽象动作 读取 f和 读取 s。下面将解释 顺序归约和并行归约的出现，目前应忽略它们。该抽象是可靠的，因为

原始快照实现的任何并发执行，在用此抽象版本替换扫描后仍然是可能的。这种可靠性直接源于读取 f和读取 s对读取行为进行了过度近似。

接下来，我们将展示，这种对扫描的抽象是原子动作扫描规约的一种精化，由此完成线性化证明。

2.3 并行语句的归约

我们证明，将图 2 中抽象扫描 里的所有读取操作都视为原子执行，且不受其他线程干扰，是合理的。第一步化简移除了并行组合的使用，并用注释并行归约表示。第一次出现并行归约 时，依据是读取 f(2) 是一个右移动操作，因此，

```
(call r1[1] := read_f(1)) par (call r1[2] := read_f(2))
```

可以改写为

```
call r1[1] := read_f(1); call r1[2] := read_f(2)
```

其中，并行组合并行 被顺序组合;取代。这为两个动作确定了执行顺序，但两次调用之间仍允许发生干扰（即在读取 f(1) 完成之后、读取 f(2)开始之前）。

事实上，对于任何一种交错，只要 读取 f(2) 先于 读取 f(1)执行，读取 f(2)的右移动性就意味着，可以将它可靠地交换到所有执行至 读取 f(1) 的动作以及 读取 f(1) 本身之后（此处，可靠性指保留最终状态以及动作或过程的所有返回值）。

反过来，第二次出现的 并行归约 依赖于 读取 s(1) 是左移动操作，从而将并行组合 并行 化简为顺序组合。

```
seq-reduce { // -> atomic {
  call r1[1] := read_f(1);
     // right
  call r1[2] := read_f(2);
     // right
  call r2[1] := read_s(1);
     // left
  call r2[2] := read_s(2);
     // left
  if (r1 == r2) {
    snapshot := r1;
    return;
  }
}
```

图 3：化简后的循环迭代。

两个并行语句化简后的结果如图 3 左侧所示。为简单起见，我们只写循环迭代。读取序列现在由一系列右移动操作和随后的一系列左移动操作组成，因此我们可以使用 Lipton 化简，将其重写为一个原子区段（条件语句以及读取之后的赋值操作访问的是局部变量，可以按任意方向重新排序，此处是向左移动）。通过关键字 顺序归约即可应用这一化简原则。扫描 最终的化简形式会将所有读取操作和条件判断组合在一起

一个原子区段，使用关键字 原子 标记。现在很容易证明，扫描细化了原子动作 扫描规约:

– 每次条件语句失败的迭代都不会影响返回值，– 如果条件语句成功，那么对于每个单元，`read f` 和 `reads` 返回的时间戳都相同。这表明两次读取都访问了当前内存状态，且这些值并非任意选取。具体而言，如果 `readf` 返回的时间戳小于内存中的时间戳，

```
procedure scan() returns (snapshot:
     [int]StampVal){
  var r1: [int]StampVal;
  var r2: [int]StampVal;
  while (true) {
    seq-reduce {
      call r1 := collect_f(n); // right
      call r2 := collect_s(n); // left
      if (r1 == r2) {
        snapshot := r1;
        return;
      }
    }
  }
}
```

```
right procedure collect_f(n: int)
     returns (r: [int]StampVal) {
  var out: StampVal;
  if (n == 0) { return; }
  else {
    par-reduce {
      (call r := collect_f(n-1)) par
      (call out := read_f(n))
    }
    r[n] := out;}
}
left procedure collect_s(n: int) returns
     (r: [int]StampVal) {
  var out: StampVal;
  if (n == 0) { return; }
  else {
    par-reduce {
      (call out := read_s(n)) par
      (call r := collect_s(n-1))
    }
    r[n] := out;}
}
```

图 4：对无界大小内存的扫描过程抽象应用化简。

那么 read_s 就不可能返回相同的时间戳，因为它只会返回严格大于当前内存中时间戳的时间戳（这些时间戳单调递增）。出于类似原因，动作 reads 不可能返回大于内存中时间戳的时间戳。

2.4 无界内存情形：结构化代码的化简

图 4 列出了一个化简证明，针对的是将先前的 扫描 实现扩展到无界大小内存的情形。内存读取在两个递归过程 collect f 和 collects 中执行，这两个过程使用相应的read f和 read s动作来读取内存单元。

值得注意的是，这展示了如何将 Lipton 归约扩展到结构化程序，也就是包含过程调用的代码。为实现组合性，我们引入过程的移动性类型，并以类似于 Lipton 归约的方式使用这种移动性类型。经过一个并行化简步骤（下文将作说明）后，collect f和 collect s 分别被判定为右过程和左过程。这样便可进行一个化简步骤，得到一个原子区段，其中涵盖了外层扫描 循环的整个迭代，与前一种情形相同。

现在，并行化简分别在两个递归过程中进行，并依赖于与前面类似的论证（读取 f和读取 s 动作的移动性）。经过这一步化简后，它们不再包含并行组合；又由于它们执行的所有动作都具有相同的移动性类型，因此可以在过程层面导出这种类型。左移动情形要求过程在单独执行时终止；显然这里满足这一点，因为参数递减且有下界 0。

应用这两个化简步骤后，要证明扫描过程是扫描规约的精化，与前面介绍的有界情形类似。

形式化证明这种化简技术的正确性并非易事，因为它既适用于结构化代码，也适用于并行组合。下一节将

$$
\begin{array}{rl}
A \in \mathit{ActionName} & Q \in \mathit{ProcName} \quad X \in \mathit{ActionName} \cup \mathit{ProcName} \\
\mathit{Val} & \ni \lightning \\
v \in \mathit{Var} & = \mathit{GVar} \cup \mathit{LVar} \\
g \in \mathit{GStore} & = \mathit{GVar} \to \mathit{Val} \\
\ell \in \mathit{LStore} & = \mathit{LVar} \rightharpoonup \mathit{Val} \\
\sigma \in \mathit{Store} & = \mathit{Var} \rightharpoonup \mathit{Val} \\
\rho \in \mathit{Gate} & = 2^{\mathit{Store}} \\
\tau \in \mathit{Trans} & = 2^{\mathit{Store} \times \mathit{Store}} \\
\iota, o \in \mathit{IOMap} & = \mathit{LVar} \rightharpoonup \mathit{LVar} \\
x \in \mathit{LVar} & \\
I, O \in 2^{\mathit{LVar}} & \\
M \in \mathit{MoverType} & = \{B, R, L, N, \top\} \\
F \in \{\mathit{true}, \mathit{false}\} &
\end{array}
$$

$$
\begin{array}{rl}
s \in \mathit{Stmt} ::= & \mathsf{skip} \mid \text{条件语句 } x\ s\ s \\
& \mid s;\ s \mid s \text{ 并行 } s \\
& \mid \text{调用 } (X, \iota, o) \\
& \mid \text{原子 } s \\
& \mid \text{并行归约 } s \text{ 并行 } s \\
& \mid \text{顺序归约 } s \\
\mathit{Action} ::= & (I, O, \rho, \tau, M, F) \\
\mathit{ProcSig} ::= & (I, O, M, F) \\
\mathit{as} \in & \mathit{ActionName} \to \mathit{Action} \\
\mathit{ps} \in & \mathit{ProcName} \to \mathit{ProcSig} \\
\mathcal{P} \in \mathit{Prog} = & \mathit{ProcName} \to \mathit{Stmt}
\end{array}
$$

图 5：RedPL：语法

介绍一种简单而富有表现力的编程语言，用于在后续各节中推理其正确性。

3 RedPL：语法与语义

本节介绍核心编程语言 RedPL，以形式化我们对化简的方法。我们的语言受到 RefPL [30] 的启发。图5总结了 RedPL 的语法。

变量与存储。我们假设存在固定的全局变量 *GVar*集合和固定的局部变量 *LVar* 集合，且$GVar$ 与$LVar$ 互不相交。变量 *Var* 集合是$GVar$ 与$LVar$的并集。存储 σ 是从变量到值的偏映射。我们用 $\sigma' \subseteq \sigma$ 表示 σ 是 σ' 的扩展，用 $\sigma|_V$ 表示将 σ 限制到 V，用 $\sigma - V$ 表示 $\sigma|_{dom(\sigma)\setminus V}$，用 $\sigma[\sigma']$ 表示在 dom(σ') 上与 σ' 相同、其他位置与 σ 相同的存储，并用 $g{\cdot}\ell$ 表示全局存储 g与局部存储 ℓ的组合。

动作。RedPL 使用原子动作 [11,29] 对线程的不间断执行进行建模。我们假设存在一个固定的动作集合，其名称取自 ActionName 集合。对全局变量的所有访问都限定在动作中。动作映射 as 将每个 $A \in$ ActionName 映射到一个元组 as(A) $= (I, O, \rho, \tau, M, F)$。输入变量集合 I 和输出变量集合 O 均为 LVar 的子集。门 ρ 是一个存储集合，满足对于每个 $g{\cdot}\ell \in \rho$，dom(ℓ) $= I$。转移关系 τ 是一个关于存储的关系，对于每个 $(g{\cdot}\ell, g'{\cdot}\ell') \in \tau$，满足 dom($\ell$) $= I$ 且 dom(ℓ') $= O$。从不满足门的存储 σ 执行动作（即 $\sigma \notin \rho$）会导致执行失败。否则，τ 中的每个迁移 (σ, σ') 都描述了一种可能的原子状态转移：从 σ（在 GVar$\cup I$ 上）转移到 σ'（在 GVar $\cup O$ 上）。动作的移动类型 M 是集合 MoverType [19] 的一个成员；它简洁地概括了此动作相对于由 as 定义的其他动作的可交换性性质。失败类型 F 是一个布尔值，表示此动作是否可能失败。若 $F =$ 为 false，则门必须包含 GVar $\cup I$ 上的所有可能存储。

我们的形式化并未为原子动作的主体提供具体语法，而是选择用符号转移系统对其进行抽象建模。我们采用的建模方法具有通用性，允许动作是任意的

并且可能失败的、针对全局变量、输入变量和输出变量的计算。具体而言，动作可以对多种语句进行建模，包括断言、假设、（非确定性）赋值、选择和顺序执行。

过程。RedPL 使用过程对可抢占的并发执行进行建模。我们假设存在一组固定的过程，其名称来自集合 $ProcName$，该集合与 $ActionName$ 不相交。我们将过程的规范拆分为两个映射——签名ps 和程序 $\mathcal{P}$。形式化过程的一个重要方面，是在保持签名不变的同时转换过程体。将过程行为的规范拆分为签名映射和程序，有助于我们的形式化工作。

签名映射 ps将每个 $Q \in ProcName$映射到一个元组 $(I,\ O,\ M,\ F)$。输入变量集合 I 和输出变量集合 O 分别是$LVar$的子集。当调用 Q时，其局部存储会为 $LVar$中的每个变量建立一个绑定。移动类型 M 是集合$MoverType$的成员。失败类型 F 是一个布尔值，用于指示该过程的执行是否可能失败。稍后将介绍的类型检查器会检查 M 和 F与过程体的一致性。

程序 $\mathcal{P}$ 将每个 $ProcName$ 映射到语句 s，当 Q 被调用时，该语句就会执行。基本语句 skip 不执行任何操作；它是稍后将介绍的形式操作语义中的一个标记。语句 if $x\ s_1\ s_2$ 会查找局部存储中 x 的值，并在该值为 $true$ 时继续执行 s_1，或在该值为 $false$ 时继续执行 s_2。语句 s_1 ; s_2 会先执行 s_1，再执行 s_2。语句 s_1 par s_2 会并行执行 s_1 和 s_2。

语句 call (X, ι, o) 会调用动作或过程。参数传递通过从 I 到 $LVar$ 的输入映射 ι，以及从 O 到 $LVar$ 的单射输出映射 o 来表示。对于 ι 和 o，定义域均为被调用方形参，值域均为调用方实参。输入变量是不可变的，因为输出映射不会将其映射到任何位置，而且过程的变量不会在其他地方被修改。动作调用是访问全局变量或修改全局存储、局部存储的唯一方式。

语句 atomic s 在禁用抢占的情况下执行 s，也就是说，语句 s 会先执行完成，然后才会调度任何其他并发执行。语句 par-reduce s_1 par s_2 表达了程序员意图：将并行计算 s_1 par s_2 归约为顺序计算 s_1 ; s_2。语句 seq-reduce s 表达了程序员意图：将语句 s 归约为 atomic s。如果语句 s 中没有出现 atomic，则称其为无原子操作。如果语句 s中没有出现 seq-reduce 或 par-reduce，则称其为无归约操作。如果对于所有 $Q \in$dom($\mathcal{P}$)， $\mathcal{P}(Q)$ 都是无原子操作，则称程序 $\mathcal{P}$ 为无原子操作。

虽然 RedPL 不显式支持循环和非确定性选择，但两者都可以建模。我们可以用递归过程来建模循环。我们可以通过先将一个非确定性选取的值赋给局部变量（通过一个动作），再使用条件语句 if $x\ s\ s$ 来建模非确定性选择。

SC

$f ::= (\ell, SC[s])\ \ t\ =\ f \mid f\ t :: \mathsf{Lf}\ \mathsf{Nd} :: = \{t, \ldots, t\} :: (g,)$

$\mathcal{T}$ $c\ =\ \mathcal{T}$

$\mid \lightning$ $::= \bullet_s \mid SC;$ $s \mid$ SC原子区段 内部顺序规约

$\mid$

SC $= \{TC\} \uplus \mathcal{T}$

TC $::$ $= \bullet_t \mid f\ t TC t\, \mathsf{Nd} :: LC\ = PC[\ \bullet_\ell :: \mathsf{Lf}\ (, SC)]$(过

PC

程调用) $(g, PC[\mathsf{Lf}\ (\ell, SC[$调用 $(Q, \iota, o)])] \to (g, PC[\mathsf{Nd}\ (\ell, SC[$调用 $(Q, \iota, o)])\ \mathsf{Lf}\ (\{v \mapsto \text{☠} \mid v \in LVar\}[\ell \circ \iota], \mathcal{P}(Q)))]$ (返回) $(g, PC[\mathsf{Nd}\ (\ell, SC[$调用 $(Q, \iota, o)])\ \mathsf{Lf}\ (Q, \ell', \mathsf{skip})] \to (g, PC[\mathsf{Lf}\ (\ell[\ell' \circ o^{-1}], SC[\mathsf{skip}])])$ (分叉) $(g, PC[\mathsf{Lf}\ (\ell, SC[s_1\, \mathsf{par}\ \ s_2])]) \to (g, PC[\mathsf{Nd}\ (\ell, SC[s_1\ \mathsf{par}\ \ s_2])\ \mathsf{Lf}\ (\ell, s_1)\ \mathsf{Lf}\ (\ell, s_2)])$ (合并) $(g, PC[\mathsf{Nd}\ (\ell, SC[s_1\, \mathsf{par}\ \ s_2])\ \mathsf{Lf}\ (\ell_1, \mathsf{skip})\ \mathsf{Lf}\ (\ell_2, \mathsf{skip})]) \to (g, PC[\mathsf{Lf}\ (\ell[\ell_1|_{mod(s_1)}][\ell_2|_{mod(s_2)}], SC[\mathsf{skip}])])$ (动作调用) $as(A) = (, \rho, \tau)\ (g{\cdot}(\ell \circ \iota)\ \ g'{\cdot}\hat{}\ell) \in \rho \circ \tau^{\ell'} = \ell[\hat{}_{\ell \circ o^{-1}}](g, LC[\ell][$调用 $(A, \iota, o)]) \to (g', LC[\ell'][\mathsf{skip}])$ (动作失败) $as(A) = (, \rho, , ,)\ g{\cdot}(\ell \circ \iota) \notin \rho$ (分支) $s' = \begin{cases} s_1 & \ell[x] = true \\ s_2 & \ell[x] = false \end{cases} (g, LC[\ell][$ 调用 $(A, \iota, o)]) \to \lightning\ (g, LC[\ell][$ 条件

语句 $x\ s_1\ s_2]) \to (g, LC[\ell][s'])$ (skip) $(g, LC[\ell][\mathsf{skip}\ ;\ s]) \to (g, LC[\ell][s])$ (停止) $(g, \{\mathsf{Lf}\ (, \mathsf{skip})\} \uplus \mathcal{T}) \to (g, \mathcal{T})$ (原子进入) $(g, LC[\ell][\mathsf{atomic}\ \ s]) \to (g, LC[\ell][$原子区段 $s])$ (原子退出) $(g, LC[\ell][$原子区段 skip$]) \to (g, LC[\ell][\mathsf{skip}])$ (并行归约进入) $(g, PC[\mathsf{Lf}\ (\ell, SC[$并行归约 $s_1\ \mathsf{par}\ \ s_2])]) \to (g, PC[\mathsf{Nd}\ (\ell, SC[$并行归约 $s_1\, \mathsf{par}\ \ s_2])\ \mathsf{Lf}\ (\ell, s_1)\ \mathsf{Lf}\ (\ell, s_2)])$ (并行归约退出) $(g, PC[\mathsf{Nd}\ (\ell, SC[$并行归约 $s_1\, \mathsf{par}\ \ s_2])\ \mathsf{Lf}\ (\ell_1, \mathsf{skip})\ \mathsf{Lf}\ (\ell_2, \mathsf{skip})]) \to (g, PC[\mathsf{Lf}\ (\ell[\ell_1|_{mod(s_1)}][\ell_2|_{mod(s_2)}], SC[\mathsf{skip}])])$ (顺序归约进入) $(g, LC[\ell][$顺序归约 $s]) \to (g, LC[\ell][$内部顺序规约 $s])$ (顺序归约退出) $(g, LC[\ell][$内部顺序规约 skip$]) \to (g, LC[\ell][\mathsf{skip}])$ 图 6: RedPL: 程序 $\mathcal{P}$ 的操作语义

3.1 语义

图 6 展示了 RedPL 的操作语义，它是关于配置的转移关系$\rightarrow$。每个配置要么是失败 $\lightning$，要么是由全局存储 g 和线程的有限多重集 $\mathcal{T}$ 组成的二元组（$g,\ \mathcal{T}$）。每个线程都是一棵树（对调用栈作了推广）；通过调用语句和并行语句创建新的叶节点（Lf）。这两种语句都会在内部节点 Nd 中阻塞调用者，直到叶节点执行完毕。每个树节点都包含一个帧 （$\ell,\ s$），其中 ℓ 是局部存储，s 是待执行的剩余语句。

在 $\rightarrow$的定义中，我们使用了若干求值上下文，每个上下文都有一个唯一的洞 $\bullet$，用于标记求值位置；填入该洞记作 $\cdot[\cdot]$。SC 是语句上下文，即带有一个洞 $\bullet$的语句。$SC[s]$ 是一个语句，其中 s 处于求值位置。除了洞 $\bullet$之外，还有另外三种语句上下文。上下文 $SC\,;s$ 会先完成SC 的求值，再继续执行 s。进入原子区段时会引入上下文 in-atomicSC 。进入内部顺序规约时会引入上下文 in-seq-reduceSC 。PC 是由线程树组成的多重集，其中某棵树带有一个洞 $\bullet$ 。这个多重集PC还满足以下条件：(1) 不含洞 $\bullet$ 的树中不包含原子区段语句。(2) 如果 t 是带有洞 $\bullet$的树，且其中存在任何原子区段语句，那么该语句必须位于从 t 的根节点到该洞的唯一路径上。这些条件确保原子语句的执行不会受到干扰。PC 中的洞可以填入任意树。LC是PC 的一种特化形式，其中洞由一个叶节点填充，而该叶节点内部有两个洞，分别用于局部存储和下一条待执行语句。

$$
\begin{aligned}
mod(\texttt{skip}) &= \varnothing \\
mod(\texttt{call}\ (X, \iota, o)) &= \mathrm{img}(o) \\
mod(s_1\,;s_2) &= mod(s_1) \cup mod(s_2) \\
mod(s_1\ \texttt{par}\ s_2) &= mod(s_1) \cup mod(s_2) \\
mod(\texttt{atomic}\ s) &= mod(s) \\
mod(\texttt{par-reduce}\ s_1\ \texttt{par}\ s_2) &= mod(s_1) \cup mod(s_2) \\
mod(\texttt{seq-reduce}\ s) &= mod(s) \\
mod(\texttt{if}\ x\ s_1\ s_2) &= mod(s_1) \cup mod(s_2)
\end{aligned}
$$

图 7： mod 函数

图 6 为固定程序 $\mathcal{P}$ 提供语义，该程序 $\mathcal{P}$ 由一组规则组织而成，空缺处的每种语句各有一条规则。用于在局部存储 ℓ 的上下文中执行调用 ($Q,\ \iota,\ o$) 的规则（**过程调用**）会创建一个新的叶节点，并用一个局部存储初始化其帧，其中 Q 的输入变量从 $\ell \circ \iota$ 获取其值（$\circ$ 表示函数或关系复合），其余变量则初始化为默认值 ☣。用于从调用中返回的规则（**返回**）会使用输出映射 o，将被调用者局部存储中的值更新到调用者的局部存储中。

规则 （**分叉**）用于执行 s_1 par s_2 ，它会创建两个叶节点来分别执行s_1 和 s_2，每个节点都会获得父节点局部存储的一份副本。父节点会被阻塞，直到两个子节点都执行完毕。随后，每个子节点中已修改的变量都会按照规则 （**合并**）写回父节点的局部存储。RedPL 的类型检查器（见第 5 节）会检查 s_1 修改的局部变量是否未被 s_2 访问，反之亦然。此检查可确保将子节点的局部存储更新到父节点的局部存储时不存在冲突。图 7 所示的 mod 函数会近似计算语句所修改的局部变量集合。

原子动作直接在调用者的上下文中执行。如果当前存储不满足所执行动作的门条件，执行便会停止并进入失败格局 $\lightning$（规则**（动作失败）**）。否则，执行会根据该动作的转移关系进行一步迁移（规则**（动作调用）**）。我们将原子动作执行时发生的迁移称为 $action$ 迁移，而其他所有迁移称为 $local$ 迁移。对于在帧中局部存储为 ℓ 的叶节点上，以输入映射 ι_A 调用动作A 时发生的任意动作转换 t_A ，我们将该迁移的输入参数绑定定义为$in\text{-}bind(t_A) = \ell \circ \iota_A$。

规则 **（分支）**执行条件语句。规则 **（skip）**将求值上下文向前推进。规则**（停止）**将已完成的树从树的多重集中移除。规则 **（原子进入）**进入原子区段，规则 **（原子退出）**则退出该区段。规则**（并行归约进入）**和 **（并行归约退出）**分别类似于**（分叉）**和**（合并）**。规则 **（顺序归约进入）**和**（顺序归约退出）**用于进入和退出顺序归约块。

4 原子动作的可交换性

我们将介绍一些基本概念，并在本节末尾用它们定义良构动作映射。直观地说，如果动作映射 as 中每个原子动作的移动类型相对于as中的整个原子动作池都是正确的，那么该映射就是良构的。对于接下来的几个定义，我们固定两个动作 $X = (I_X,\ O_X,\ \rho_X,\ \tau_X,\ _,\ _)$ 和 $Y = (I_Y,\ O_Y,\ \rho_Y,\ \tau_Y,\ _,\ _)$，

最弱自由前置条件最弱自由前置条件 弱最弱前置条件$(X,\ \rho_Y)$ 是满足以下条件的所有三元组 $(g,\ \ell_X,\ \ell_Y)$ 的集合： X 在 $g{\cdot}\ell_X$ 中不会失败，并且从 $g{\cdot}\ell_X$ 执行 X 总是会得到一个全局存储 g' ，使得 Y 的门在 $g'{\cdot}\ell_Y$ 上成立。

$$\begin{aligned} wlp(X, \rho_Y) = \{(g, \ell_X, \ell_Y) \mid g{\cdot}\ell_X \in \rho_X \wedge \\ \big(\forall g', \ell'_X : (g{\cdot}\ell_X, g'{\cdot}\ell'_X) \in \tau_X \implies g'{\cdot}\ell_Y \in \rho_Y\big)\} \end{aligned}$$

这个定义的一个推论是，如果某个状态满足 X 的门，但不满足 弱最弱前置条件$(X,\ \rho_Y)$，那么就存在一种执行 X 的方式，使其到达一个 Y 的门不成立的状态。下文将形式化地说明这一推论；它在证明我们的归约定理的可靠性时发挥了重要作用。

$$\begin{aligned} \forall g, \ell_X, \ell_Y : g{\cdot}\ell_X \in \rho_X \ \wedge\ (g, \ell_X, \ell_Y) \notin wlp(X, \rho_Y) \implies \\ \exists \hat{g}, \hat{\ell} :\ (g{\cdot}\ell_X, \hat{g}{\cdot}\hat{\ell}) \in \tau_X \ \wedge\ \hat{g}{\cdot}\ell_Y \notin \rho_Y \end{aligned}$$

交换我们称 X 与 Y 交换，记作 交换$(X,\ Y)$，如果先执行 X 再执行 Y 所到达的状态，也可以通过先执行 Y 再执行 X 到达。

$$\begin{aligned} \forall g, g', \bar{g}, \ell_X, \ell_Y, \ell'_X, \ell'_Y : (g, \ell_X, \ell_Y) \in wlp(X, \rho_Y) \wedge (g, \ell_Y, \ell_X) \in wlp(Y, \rho_X) \\ \wedge\ (g{\cdot}\ell_X, \bar{g}{\cdot}\ell'_X) \in (\rho_X \circ \tau_X) \wedge (\bar{g}{\cdot}\ell_Y, g'{\cdot}\ell'_Y) \in (\rho_Y \circ \tau_Y) \\ \implies \exists \hat{g} :\ (g{\cdot}\ell_Y, \hat{g}{\cdot}\ell'_Y) \in (\rho_Y \circ \tau_Y) \wedge (\hat{g}{\cdot}\ell_X, g'{\cdot}\ell'_X) \in (\rho_X \circ \tau_X) \end{aligned}$$

成功保持 我们称 X 保持了 的成功，记作保持-成功$(X,\ Y)$，如果从某个状态出发，只要 Y 的门和 X 的门均成立，那么 X 的任何迁移都会到达一个同样满足 Y 的门的状态。

$$\forall g, \ell_X, \ell_Y :\ g{\cdot}\ell_X \in \rho_X \wedge g{\cdot}\ell_Y \in \rho_Y \implies (g, \ell_X, \ell_Y) \in wlp(X, \rho_Y)$$

失败保持 我们称 X 保持了 的失败，记作保持-失败(X， Y)，如果从某个状态出发， X 不会失败而 Y 会失败，那么存在 X 的一次迁移，使其从该状态到达 Y 失败的状态。等价地，如果从某个状态出发， X 不会失败，且无法执行会导致 Y 失败的迁移，那么 Y 不会从该状态失败。

$$\forall g, \ell_X, \ell_Y : (g, \ell_X, \ell_Y) \in wlp(X, \rho_Y) \implies g{\cdot}\ell_Y \in \rho_Y$$

良构动作映射。 我们有 5 种类型：右移动作 (**R**)、左移动者 (**L**)、双向移动动作 (**B**)、非移动者 (**N**)、顶 (⊤)。这些类型在由以下关系定义的偏序下构成一个格：**B** $\sqsubseteq a \sqsubseteq$ **N** $\sqsubseteq \top$ 其中 $a \in \{\mathbf{R}, \mathbf{L}\}$。将该偏序的并运算符记为 $\sqcup$。动作映射 as 在满足以下条件时是良构的： 对于所有动作 $A \in ActionName$， 若 $as(A) = (I_A, O_A, \rho_A, \tau_A, M_A, F_A)$，则满足以下条件：

1. $M_A \neq \top$。2. 若 $M_A \sqsubseteq \mathbf{L}$，则对于所有 $X \in ActionName$：

$$\begin{array}{ll} \texttt{(L1)} & preserves\text{-}success(X, A) \\ \texttt{(L2)} & preserves\text{-}failure(A, X) \\ \texttt{(L3)} & commutes(X, A) \end{array}$$

3. 若 $M_A \sqsubseteq \mathbf{R}$，则对于所有 $X \in ActionName$：

$$\begin{array}{ll} \texttt{(R1)} & preserves\text{-}success(A, X) \\ \texttt{(R2)} & commutes(A, X) \end{array}$$

4. 若 $F_A = false$，则 $\rho_A = \{\, g{\cdot}\ell_A \mid g \in GStore \wedge \ell_A \in LStore \wedge \mathsf{dom}(\ell_A) = I_A \,\}$.

我们的良构动作映射条件与先前关于 Civl [27] 的工作中的条件有重要差异。我们的条件更弱，而且我们在撰写化简技术可靠性证明（定理 1）的过程中发现了这些条件。

首先，我们对 交换(X, Y) 的定义更弱。Civl 中对应的条件只假设在初始状态下满足 ρ_X 和 ρ_Y，同时验证X 和 Y 可以重新排序。相比之下，我们的条件在初始状态下作出更强的假设：弱最弱前置条件(X, ρ_Y) 和 弱最弱前置条件(Y, ρ_X)。直观地说，我们的检查只要求对这样的初始状态重新排序 X 和 Y：从这些状态出发，无论以何种顺序执行 X 和 Y，都不可能失败。以下示例说明了这一区别。设 S 为一个共享集合，并定义动作 A 和 B 如下：$A(j)$： $S := S \setminus \{j\}$, $B(i)$：断言 $i \in S$，其中该断言就是 B 的门。按照我们的交换性条件， A 是右移动操作， B 是左移动操作，但按照 Civl 原有的条件，这一点并不成立。

其次，Civl 对左移动操作分别执行两项检查：向后保持和非阻塞。向后保持要求左移动操作 A证明，对于每个动作 X ，如果条件语句 ρ_X 在执行 A 的一步后成立，那么它在这一步之前也成立。非阻塞检查要求 A从每个初始状态出发，要么失败，要么执行一步。相比之下，我们只进行一项失败保持检查，其对 A 的要求低于向后保持和非阻塞要求的组合。直观而言，我们的检查仅要求 A在试图保持动作 X的失败时执行一步。例如，如果 X 没有任何会导致失败的行为，那么失败保持就会自动成立，但对 A 进行非阻塞检查仍可能并非易事。

5 种化简类型

本节中，我们利用原子动作的移动类型，检查程序中并行归约和顺序归约的应用是否适用，从而实现可靠规约。为此，我们定义两个针对语句的辅助函数——可能-失败 和 移动-类型。函数 可能-失败 通过保守的静态分析，将动作的失败类型传播到语句。函数移动-类型 则利用效应系统 [19]将动作的移动类型提升到语句。结合这两个函数，我们可以定义类型良好的程序，并由此保证程序中的化简应用有效。

语句的失败类型判定 我们通过保守地传播原子动作的失败类型，计算任意语句 s 的可能-失败(s)。

$$
\begin{aligned}
\mathit{may\text{-}fail}(\texttt{skip}) &= \mathit{false} \\
\mathit{may\text{-}fail}(\texttt{call}\ (A, \iota, o)) &= \mathit{may\text{-}fail}(A) \\
\mathit{may\text{-}fail}(\texttt{call}\ (Q, \iota, o)) &= \mathit{may\text{-}fail}(Q) \\
\mathit{may\text{-}fail}(s_1\,;\, s_2) &= \mathit{may\text{-}fail}(s_1) \vee \mathit{may\text{-}fail}(s_2) \\
\mathit{may\text{-}fail}(s_1\ \texttt{par}\ s_2) &= \mathit{may\text{-}fail}(s_1) \vee \mathit{may\text{-}fail}(s_2) \\
\mathit{may\text{-}fail}(\texttt{atomic}\ s) &= \mathit{may\text{-}fail}(s) \\
\mathit{may\text{-}fail}(\texttt{par-reduce}\ s_1\ \texttt{par}\ s_2) &= \mathit{may\text{-}fail}(s_1) \vee \mathit{may\text{-}fail}(s_2) \\
\mathit{may\text{-}fail}(\texttt{seq-reduce}\ s) &= \mathit{may\text{-}fail}(s) \\
\mathit{may\text{-}fail}(\texttt{if}\ x\ s_1\ s_2) &= \mathit{may\text{-}fail}(s_1) \vee \mathit{may\text{-}fail}(s_2)
\end{aligned}
$$

图 8： *may-fail* 函数。

语句的移动类型判定 给定一个良构动作映射 as 和一个过程签名映射 ps，我们定义移动-类型 函数，该函数为语句分配移动类型。为定义此函数，我们首先定义下表中的移动类型顺序组合。利用此表，我们可以递归地定义移动-类型 如下：

	B	L	R	N	⊤
B	B	L	R	N	⊤
L	L	L	⊤	⊤	⊤
R	R	⊤	R	N	⊤
N	N	⊤	N	N	⊤
⊤	⊤	⊤	⊤	⊤	⊤

$\mathit{mover\text{-}type}(\text{skip}) = \mathbf{B}$；$\mathit{mover\text{-}type}(s_1 \text{ 并行 } s_2) = \mathit{mover\text{-}type}(s_1) \sqcup \mathit{mover\text{-}type}(s_2)$

$\mathit{mover\text{-}type}(\text{调用 } (A, \iota, o)) = \mathit{mover\text{-}type}(A)$

$\mathit{mover\text{-}type}(\text{调用 } (Q, \iota, o)) = \mathit{mover\text{-}type}(Q)$

$\mathit{mover\text{-}type}(\text{原子 } s) = \mathit{mover\text{-}type}(s)$

$\mathit{mover\text{-}type}(\text{顺序归约 } s) = \mathit{mover\text{-}type}(s)$

$\mathit{mover\text{-}type}(\text{并行归约 } s_1 \text{ 并行 } s_2) = \mathit{mover\text{-}type}(s_1; s_2) \sqcup \mathit{mover\text{-}type}(s_1\,;\, s_2) = \mathit{mover\text{-}type}(s_1); \mathit{mover\text{-}type}(s_2)$

$\mathit{mover\text{-}type}(\text{条件语句 } x\ s_1\ s_2) = \mathit{mover\text{-}type}(s_1) \sqcup \mathit{mover\text{-}type}(s_2)$

图 9： *mover-type* 函数。

例如，考虑概述中示例里的以下顺序归约语句。分配给顺序归约区块的移动类型将为$\mathbf{N}$。

```
seq-reduce {
    par-reduce {(call r1[1] := read_f(1)) par (call r1[2] := read_f(2))};
    par-reduce {(call r2[1] := read_s(1)) par (call r2[2] := read_s(2))};
}
```

第一次并行归约中对读取 f 的每次调用都具有$\mathbf{R}$ 类型。随后，根据表格，并行归约语句会获得移动类型$\mathbf{R};\mathbf{R} = \mathbf{R}$ 。类似地，第二次并行归约包含对读取 s 的调用，每次调用的类型都是$\mathbf{L}$，而并行归约

语句会获得移动类型$\mathbf{L};\mathbf{L} = \mathbf{L}$。现在，顺序归约中的语句按顺序组合了两个并行归约块：第一个的类型是$\mathbf{R}$，第二个的类型是$\mathbf{L}$。顺序组合$\mathbf{R};\mathbf{L}$ 的结果是$\mathbf{N}$类型，该类型随后被赋予整个顺序归约块。

上述规则意味着，如果 s 嵌套在 s' 中，并且 $\mathit{mover\text{-}type}(s) = \top$，那么 $\mathit{mover\text{-}type}(s') = \top$。由于 $\mathit{mover\text{-}type}(s_1 \text{ 并行 } s_2) = \top$，如果 s_1 并行 s_2 嵌套在语句 s 中，那么 $\mathit{mover\text{-}type}(s) = \top$。在下面讨论类型良好的程序的规则时，我们将回到这一观察。

类型良好的程序 我们定义谓词 $\mathit{well\text{-}typed}(s,\ l)$，其中 $s \in$语句且 $l \in 2^{LVar}$。谓词 $\mathit{well\text{-}typed}(s,\ l)$ 检查语句的两个方面。首先，它检查 s是否只访问其被允许访问的局部变量。之所以需要进行这项检查，是因为如果 s包含两个并行执行的嵌套语句 s_1 和 s_2，那么这两个语句所修改的局部变量必须彼此不相交。实际上，我们检查的是一个更强但更易检查的性质：s_1 所修改的局部变量既不会被 s_2读取，也不会被其写入，反之亦然。这一要求仅用于简化形式化过程。也可以改为考虑局部变量的新副本（注意，并行构造中的分支数量是固定的）。在我们的实现中，并行构造的各个分支只能是过程调用，因此这一限制实际上转化为要求它们的输入和输出参数彼此不相交，这既自然又易于检查。其次，我们检查 par-reduce 和 seq-reduce 的应用是否在其目标语句上具有适当的可移动操作和失败类型。检查的这一部分使用了先前定义的函数 $\mathit{mover\text{-}type}$ 和 $\mathit{may\text{-}fail}$。

$$\mathit{well\text{-}typed}(\text{skip } l) = \mathit{true}\mathit{well\text{-}typed}(\text{原子 } s, l) = \mathit{well\text{-}typed}(s, l)\mathit{well\text{-}typed}(\text{调用 } (X, \iota, o)\ l) = \text{img}(\iota) \subseteq l \wedge \text{img}(o) \subseteq \mathit{lwell\text{-}typed}(\text{条件语句 } x\ s_1\ s_2, l) = \mathit{well\text{-}typed}(s_1, l) \wedge \mathit{well\text{-}typed}(s_2, l) \wedge x \in \mathit{lwell\text{-}typed}(s_1\,;\, s_2, l) = \mathit{well\text{-}typed}(s_1, l) \wedge \mathit{well\text{-}typed}(s_2, l)\mathit{well\text{-}typed}(s_1 \text{ par } s_2, l) = \mathit{well\text{-}typed}(s_1, l - \mathit{mod}(s_2)) \wedge \mathit{well\text{-}typed}(s_2, l - \mathit{mod}(s_1))\mathit{well\text{-}typed}(\text{顺序归约 } s, l) = \mathit{well\text{-}typed}(s, l) \wedge \mathit{mover\text{-}type}(s) \sqsubseteq \mathbf{N}\mathit{well\text{-}typed}(\text{并行归约 } s_1 \text{ par } s_2, l) = \mathit{well\text{-}typed}(s_1, l - \mathit{mod}(s_2)) \wedge \mathit{well\text{-}typed}(s_2, l - \mathit{mod}(s_1)) \wedge (\mathit{mover\text{-}type}(s_1) \sqsubseteq \mathbf{L} \vee \mathit{mover\text{-}type}(s_2) \sqsubseteq \mathbf{R} \wedge \neg \mathit{may\text{-}fail}(s_2))$$

图 10： $\mathit{well\text{-}typed}$ 函数

上述定义中的大多数规则都很直观。我们仔细考察并行归约和顺序归约的规则，重点关注移动器相关检查。par-reduce s_1 par s_2 会检查以下两种情况之一是否成立：要么 s_1 是左移动操作，要么 s_2 是右移动操作且不得失败。直观来说，在这两种情况下，s_1 中的所有代码都可以移到 s_2中的所有代码之前。例如，下面的例子说明了为什么不允许右移动器语句 s_2 失败。设`x` 是一个共享整数变量，`read` 和`inc` 是动作，其中`read`是一个右移动操作，它首先断言`x > 0` （这对应于它的门），然后

```
var x: int;

right action read() returns (out: int) {
  assert x > 0;
  assume out <= x;
}

action inc() {
  x := x + 1;
}
```

```
procedure Q {
  par-reduce {
    call inc() par (call read())
  }
}

procedure Q' {
  call inc();
  call read();
}
```

Fig. 11：说明为何右移动操作不得失败这一条件必不可少的示例

读取x 的值（恰好等于或小于该值），而inc是一个非移动者，它将x加 1（它不是左移动者，因为它不满足失败保持；它也不是右移动者，因为它无法移到读取操作之后）。设 Q 是原始程序 P 中的一个过程，并假设通过应用并行归约将其约简为约简程序 P' 中的 Q'（如上面的代码片段所示）。假设初始状态为x = 0，则 P 可能会失败（如果read先执行），但 P' 不存在会失败的执行（因为inc 总是在read之前执行）。这会带来问题，因为这种归约会“隐藏”失败，有悖于可靠推理。因此，我们要求右移动操作不会失败。这正是辅助函数 *may-fail* 所检查的内容。

顺序归约 s 检查 可移动操作-类型(s) $\sqsubseteq \mathbf{N}$，这意味着经过 s 的任何执行都具有如下形式：$\mathbf{R}^* \cdot \mathbf{N}? \cdot \mathbf{L}^*$，因此可以将 s 转换为原子区段 [19]。在这种情况下，语句 s 中不能嵌套任何尚未归约的并行语句（其移动类型为 $\top$）。但 s 中可以嵌套已归约的并行语句（其移动类型可能不同于 $\top$）。这种灵活性在实践中十分重要，对我们在第8节介绍的案例研究尤其有用。

如果对于所有 $Q \in \in \mathrm{dom}(\mathcal{P})$，当 $ps(Q) = (\ , M, F)$ 时，下列条件均成立，则称程序 $\mathcal{P}$ 类型良好：(1) 类型良好($\mathcal{P}(Q)$, $LVar$)；(2) 移动类型($\mathcal{P}(Q)$) $\sqsubseteq M$；以及 (3) 可能-失败($\mathcal{P}(Q)$) $\Rightarrow F$。对于程序中的每个过程 Q，我们分别计算其类型良好谓词，所依据的是 Q 调用的所有过程和动作的签名。首先，检查 Q 的主体，确认其相对于所有局部变量的集合而言类型良好。其次，Q 主体的移动类型必须强于 Q 的注释移动类型 M。这项检查确保即使在调用点内联了 Q，调用 Q 的过程也能通过类型检查。本质上，这意味着无论完成对某个过程的调用时采取何种具体执行，该过程的移动类型都有效。第三，检查 Q 的失败类型，确认它是 Q 主体失败类型的保守近似。我们在第6节中使用类型良好的程序这一概念来陈述约简技术的可靠性定理。

6 RedPL 程序的化简

本节中，我们为 RedPL 程序中的顺序归约和并行归约注解赋予含义，并陈述相关的可靠性定理。可靠性是通过程序之间的精化 关系来陈述的，该关系将在下文定义。

若配置 $(g,\ \mathcal{T})$ 包含任意数量的线程，且这些线程即将执行一条类型正确语句，则称该配置为初始 配置；也就是说，对于所有 $t \in \mathcal{T}$，都存在ℓ和 s，使得 $t = Lf(\ell,\ s)$、类型 -正确$(s,\ LVar)$，并且 s 无原子操作且无归约操作。若配置 $(g,\ \mathcal{T})$ 满足 $\mathcal{T} = \varnothing$，则称其为最终 配置；也就是说，所有线程都已成功执行完毕。失败格局 $\lightning$ 也属于最终配置。

给定两个类型良好的程序 $\mathcal{P}$ 和 $\mathcal{P}'$，若对所有初始配置 $(g,\ \mathcal{T})$，以下两个性质均成立，则称 $\mathcal{P}$ 细化 $\mathcal{P}'$ （记作$\mathcal{P} \preccurlyeq \mathcal{P}'$）：(P1) 如果从初始配置 $(g,\ \mathcal{T})$ 出发存在一条 $\mathcal{P}$ 的执行会失败，那么从同一初始配置出发也存在一条 $\mathcal{P}'$ 的执行会失败：

$$(g, \mathcal{T}) \xrightarrow{\mathcal{P}}{}^{*} \lightning \implies (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} \lightning$$

(P2) 如果存在一条从初始配置 $(g,\ \mathcal{T})$ 开始、到达最终配置 $(g',\ \varnothing)$ 的 $\mathcal{P}$ 执行，那么从相同初始配置出发也存在一条 $\mathcal{P}'$ 的执行，该执行要么到达相同的最终配置，要么导致失败：

$$(g, \mathcal{T}) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \implies (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} (g', \varnothing) \vee (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} \lightning$$

细化关系是传递的，即若 $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$ 和 $\mathcal{P}_2 \preccurlyeq \mathcal{P}_3$，则 $\mathcal{P}_1 \preccurlyeq \mathcal{P}_3$。

记号 $s[s_2/s_1]$ 表示在语句 s中，将所有语句 s_1 替换为语句 s_2 后的结果。类似地，记号 $\mathcal{P}[s_2/s_1]$表示一个新程序：对于每个过程 Q，将 Q 的主体中所有 s_1 （定义见 $\mathcal{P}$）替换为 s_2：

$$\mathcal{P}[s_2/s_1] = \{Q \mapsto \mathcal{P}(Q)[s_2/s_1] \mid Q \in ProcName\}$$

语句 s 在程序 $\mathcal{P}$ 中是终止的，条件是从任意配置 $(g,\ \mathrm{Lf}\,(\ell,\ s))$ 出发，都不存在无限执行，且类型良好$(s,\ LVar)$。类型良好的程序 $\mathcal{P}$ 是终止的，条件是对于所有满足可移动操作-类型(Q) $\sqsubseteq \mathbf{L}$ 的 $Q \in \mathrm{dom}(\mathcal{P})$，$Q(Q)$ 在 $\mathcal{P}$ 中终止。

定理 1。 设 $\mathcal{P}_s$ 是无原子操作且类型良好的程序。设

$$\mathcal{P}_i = \mathcal{P}_s[s_1\,;s_2 \ /\ \texttt{par-reduce}\ s_1\ \texttt{par}\ s_2]$$
$$\mathcal{P}_r = \mathcal{P}_i[\texttt{atomic}\ s\ /\ \texttt{seq-reduce}\ s]$$

那么，程序 $\mathcal{P}_i$ 和 $\mathcal{P}_r$ 类型良好。此外，若 $\mathcal{P}_i$ 终止，则：*(1)* $\mathcal{P}_s \preccurlyeq \mathcal{P}_i$，且 *(2)* $\mathcal{P}_i \preccurlyeq \mathcal{P}_r$。因此， $\mathcal{P}_s \preccurlyeq \mathcal{P}_r$。

[**证明概要**] 首先，我们证明 $\mathcal{P}_s \preccurlyeq \mathcal{P}_i$，从而说明对 par-reduce 内部的并发行为进行顺序化是可靠的。我们分别证明细化性质(P1) 和(P2) 。顶层策略是通过对尚未归约的 par-reduce 应用次数进行归纳，将 $\mathcal{P}_s$ 的一次执行重写为 $\mathcal{P}_i$ 的一次执行，使得对于每次 par-reduce 规则的应用，语句 s_1 都在 s_2 之前执行。注意， $\mathcal{P}_i$ 会消除所有 par-reduce 的出现，包括嵌套在 seq-reduce 内部的那些。因此，任何 seq-reduce 应用内部都不存在并行性。

其次，我们证明 $\mathcal{P}_i \preccurlyeq \mathcal{P}_r$，从而表明为顺序归约中的所有代码块定义原子区段是合理的。这一步同样通过对尚未归约的顺序归约应用次数进行归纳来完成。我们通过将 $\mathcal{P}_i$ 的一次执行改写为 $\mathcal{P}_r$ 的一次执行来证明这一点，其中顺序归约中的每个代码块都具有 $\mathbf{R}^{*}\mathbf{N}?\mathbf{L}^{*}$ 的形式。

更多详情请参见附录 A。

7 实现

我们已在用于分层并发程序的 Civl [28] 验证器中实现了我们的证明规则 [27]。我们的实现涵盖了形式化的所有方面，唯独未涵盖左移过程终止性的侧条件以及右移语句中不存在失败这一侧条件。对于第 8 节所报告的示例，这些侧条件均通过人工验证。

Civl 是 Boogie 验证器针对顺序程序的扩展 [7] 。与 Boogie 类似，Civl 的实现大致分为类型检查器和验证条件生成器。类型检查器负责基本的类型分析，还会检查变量、让出不变式、动作和过程上的层注释是否彼此一致。它还会检查每个过程的移动类型是否与根据过程体推断出的移动类型一致。Civl 中的验证条件生成器会从输入程序中消除所有并发特性，并生成一组带有规约注释的顺序过程。这些顺序过程编码了与原子动作的移动类型相关的检查、每个过程的精化检查，以及与让出不变式相关的非干扰检查。随后，这些顺序过程会交由标准的 Boogie 流程处理，该流程将每个过程转换为逻辑约束，并使用 SMT 求解器进行检查。

我们的实现对 Civl 作了修改和扩展，具体如下。首先，我们根据第 4 节中针对良构动作映射所列出的规则，修改了用于检查动作移动类型的验证条件。修订后的规则更为通用，因此适用于更多场景。针对左移动子的失败保持检查也经过修订，在调试未能通过的证明时能提供更直观的思路。

其次，我们为过程添加了移动类型，并按照第 5 节所述，实现了针对过程体检查这些类型的功能。在确定连续程序层之间允许进行的程序变换程度时，我们的类型检查器也会考虑过程的移动类型。按照第 6 节所述，以完全嵌套的方式规约程序的能力十分重要，它使单个程序层能够完成大量证明工作，从而降低各层的证明开销。我们允许使用前置条件和后置条件对标注了移动类型的过程进行摘要。现在，无需内联这些过程，就可以分析包含对此类过程调用的原子代码片段。我们的实现还以与递归过程相同的方式直接处理循环。

最后，我们允许使用本文提出的并行归约技术对并行调用进行化简。我们并行执行过程调用，而不是并行执行语句。这一设计选择简化了调用者与被调用者之间的参数传递。本文所述的已修改局部变量分析并无必要，可由一项简单检查取代：并行调用所有分支中使用的输出变量彼此均不相同。我们的实现将此前提出的同步异步调用证明规则纳入同一框架 [29]。

第 3 节对 RedPL 的形式化明确展示了源程序中每一处并行归约和顺序归约的应用。这些注释并未在 Civl 程序中显式声明；相反，我们的类型检查器会自动推断与之等价的信息。

8 评估

我们在一组多样且具有挑战性的案例研究中评估上述实现：并行化的快照对象（第2节）、Attiya、Bar-Noy 和 Dolev [4](ABD) 提出的经典消息传递式共享内存模拟、FLASH缓存一致性协议的实现 [31],以及两阶段提交协议的一个版本。这些实现自然地分解为多个过程，并大量使用动态线程创建。消息传递采用远程过程调用（RPC）的方式建模：广播并等待响应被表示为过程调用的并行组合，每个调用都会修改接收方的状态并返回确认。

示例	#LOC 总计	#LOC 实现与规范	Time sec
快照	119	82	0.4
ABD	389	206	1.4
一致性 608		401	5
2PC	146	111	1.8

表1：评估指标。

评估结果表明，每个案例研究都涉及并行归约和顺序归约的大量嵌套，并同时利用左移动操作和右移动操作。这种方法有助于避免复杂不变式的需求，否则，细粒度交错执行会导致这类不变式的产生。与 Civl 中的常见做法一样，证明被分解为一系列精化步骤，其中一些步骤是与化简无关的“抽象”步骤（它们会引入非确定性抽象或幽灵变量）。此外，这些证明还依赖 Civl 的其他功能，例如归纳不变式和基于权限的推理。后者对于实现交换性推理尤其有用。许多时候，动作之间的可交换性可由其（部分）输入互不相同这一点推出，而这通过权限来编码。

表1列出了评估中的定量指标。我们报告了每个证明的代码总行数，并分别统计实现和规范部分的行数。我们还报告了执行每个证明所需的墙上时钟时间。证明注释行数与实现和规范总行数之比，从 0.31（两阶段提交）到 0.88（ABD）不等。此外，所有这些证明都可以在几秒钟内完成。Civl代码库还包含约 50 个其他示例，其中包括规模更大的基准测试（数百至数千行），并适用顺序归约规则。本文重点介绍四个案例研究，因为它们同时采用顺序归约和并行归约，因此最能清晰地展示这两条规则的交互与用法。

下面，我们将详细介绍每个案例研究，但已在第2节中描述过的快照对象除外。我们将介绍实现、所证明的规范以及化简的使用。证明文件可在补充材料中获取。

8.1 ABD 寄存器

```
type TimeStamp // a set with a total order and a
lower bound
TS: TimeStamp // global timestamp used to order
operationsvalue_store: Map TimeStamp Value
procedure ReadClient(pid) returns (val) {
old_ts := Begin(pid)ts, val := Read(pid, old_ts)
End(pid, ts);}procedure WriteClient(pid, val) {
old_ts := Begin(pid)ts := Write(pid, old_ts)
End(pid, ts);}
```

```
action Begin(pid) returns (ts) {
  ts = TS;
}
action Read(pid, old_ts) returns
     (ts, val) {
  assume old_ts <= ts
  assume ts in value_store
  val := value_store[ts]
}
action Write(pid, val) returns (ts) {
  assume old_ts < ts
  assume ts not in value_store
  value_store[ts] := val
}
action End(pid, ts) {
  TS := max(TS, ts)
}
```

图 12：ABD 的线性化规范。全局变量 TS 模拟一个时钟：操作开始时读取，操作结束时递增。

ABD 算法在消息传递之上实现了一个读写寄存器（共享内存）。它在一个由 n 个副本组成、用于容错的寄存器上提供两个操作读取 和写入。少于一半的副本可以崩溃。一组客户端会并行调用这些操作。

每个副本都存储一个带时间戳的值（时间戳，值），其中时间戳取自一个有下界的全序集（例如自然数）。读取 和写入操作都分为两个阶段：查询阶段 和更新阶段。在查询阶段，它们会并行向所有副本发送查询消息，等待法定人数（至少一半）的回复，并从回复中取得时间戳t最大的回复(t,v) 。然后进入更新阶段，向所有副本发送更新消息。读取操作发送Update(t, v) 消息，而写入 操作发送Update(t+1,v') 消息，其中v' 是要写入的新值。两者都会等待收到法定人数的确认后再返回。收到Update(t, v) 消息时，如果 t 大于副本当前的时间戳，该副本就会将其存储更新为(t, v) ，无论是否执行更新，都会回复确认。收到查询 消息时，它会回复当前存储的副本。

规范：线性化。 我们的目标是证明该寄存器实现是线性化的，为此我们将问题归约为一次精化检查。为了表达每个操作似乎都在其调用与返回之间原子地生效这一条件，我们在操作开始时为其添加读取全局时钟的步骤，并在操作结束时推进时钟。该时钟与 ABD 时间戳保持一致：每次读取都会返回一个时间戳大于或等于调用时钟的值，而每次写入都会写入一个时间戳严格大于调用时钟的值。操作结束时，时钟会推进至该操作读取或写入的时间戳。所得规范如图 12 所示，其中ReadClient 和WriteClient 分别用开始和结束动作封装相应操作，以访问时钟。每个方法都只包含一个“内部”步骤（调用读取或写入动作），这体现了它们必须瞬时生效的要求。由写入更新、由读取访问的映射值存储可确保读取返回先前写入的值。

```
right procedure {:layer 3} QueryPhase(i: int,
     old_ts: TimeStamp)
  returns (max_ts: TimeStamp, max_value: Value) {
  ...
  par-reduce {
    call max_ts, max_value :=
      QueryPhase(i + 1, old_ts)
    par call ts, value :=
      Query(i, old_ts)  // right
  }
  if (less_than(max_ts, ts)) {
    max_ts := ts; max_value := value;
  }
}
```

```
procedure Read(pid: ProcessId, old_ts:
     TimeStamp) returns (ts:
     TimeStamp, value: Value) {
  ...
  seq-reduce {
    call ts, value :=
      QueryPhase(0, old_ts);   // right
    call UpdatePhase(0, ts, value); //
     left
  }
}
```

图 13：查询阶段和读取过程。

我们证明，将调用同名 ABD 过程替换为对动作 `ReadClient` 和`WriteClient` 的“具体”版本，是图 12 中抽象规范的精化。

应用化简。 目标是应用化简，以表明 ABD 过程 `Read` 和 `Write` 可以改写为在单个 原子区段内执行，然后证明该原子区段精化了图 12 中的抽象`Read` 和`Write` 动作。为此，我们引入 查询 处理程序的一个抽象，该处理程序是右移动操作，进而确保更新处理程序成为左移动操作。该抽象支持在查询阶段 和更新阶段中进行并行化简；例如，在下面所示的查询阶段 中，它允许通过将所有查询处理程序串行化，来化简递归过程（这些处理程序最初是并行执行的）。反过来，由于查询 处理程序（右移动操作）之后紧跟着更新 处理程序（左移动操作），这种串行化使得能够在读取 和写入 过程中进行顺序化简——下面以读取为例说明。

查询的抽象允许其返回低于当前副本时间戳的时间戳。不过，它不能返回任意低的时间戳——该时间戳应大于或等于在开始时获取的时钟时间戳`oldts` 。这种化简消除了归纳不变式的需求，即无需跟踪读取操作和写入操作在其查询阶段或更新阶段的不同阶段之间的关系。

8.2 缓存一致性

我们实现了 FLASH缓存一致性协议 [31]，该协议是一种基于目录的 MESI缓存一致性协议。该协议使用集中式目录来管理共享内存多处理器系统中多个缓存之间的一致性。

图 14 展示了我们的内存（`mem`）、目录（`dir`）和一组缓存（`cache`）模型。内存使用内存地址（`MemAddr`）进行索引，而每个缓存则使用本地缓存地址（`CacheAddr`）进行索引。由于内存地址空间大于缓存地址空间，因此哈希函数会将每个内存地址映射到一个缓存地址，使多个内存地址可以对应同一个缓存地址。每个缓存都以缓存行的形式存储数据，缓存行包含一个内存地址、一个值和一个状态（已修改、独占、共享、无效）。目录跟踪各个内存地址在所有缓存中的状态。如果某个内存地址由某个缓存以已修改或独占状态持有，目录就会将其记录为 `Owner(i)`，其中 `i` 是该缓存的 ID

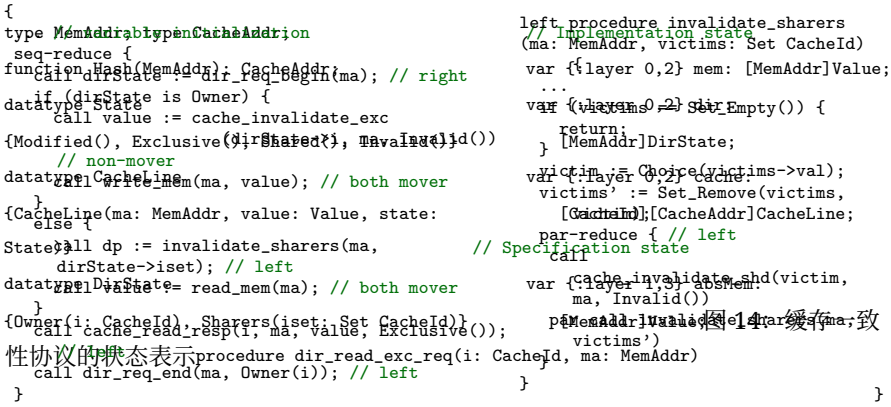

Fig. 15：目录对独占状态请求应用的化简

所有者缓存。否则，目录会将其记录为 Sharers(iset)，其中iset 是处于共享状态且包含该内存地址的缓存标识符集合。

我们实现了缓存的 5 种顶层操作：

– 缓存读取 和缓存写入，分别用于读取和写入缓存条目。– 缓存驱逐请求 用于发起缓存行驱逐。– 缓存共享读请求 和 缓存独占读请求 分别用于发起将内存地址载入缓存并置于共享模式和独占模式的操作。

现在详细介绍缓存独占读请求 的操作，以及缓存与目录之间的交互。缓存通过图 15 所示的dirread exc req 向目录发起独占请求。如果请求的内存地址对应的目录状态为所有者，目录会调用cache invalidate exc向所有者发送失效请求。随后，所有者应将其状态更改为无效，并将数据发送回目录；目录再调用write mem将数据写入内存。如果目录状态为共享者，则会调用cache invalidate shd，并行向共享者列表中的所有缓存发送失效请求。随后，目录调用readmem读取内存，并通过cache read resp将响应发送回原始缓存，最后调用dir req end结束请求。

规范。我们引入一种抽象内存，absMem。目标是证明，缓存写入和缓存读取 操作是直接从absMem读取和写入的原子动作的精化。我们隐藏目录以及所有与之交互的缓存操-

作。该规范自然地刻画了缓存一致性属性。

应用化简。引入规范中使用的幽灵变量absMem 后，我们定义了实现中若干动作的抽象，使其成为可移动操作。内存操作read mem 和write mem 被设为双向移动操作，共享失效请求cache invalidateshd 被设为左移动操作，缓存对读请求的响应cache read resp 被设为左移动操作，而用于读取和更新目录状态的动作dirreq begin 和dir req end 则分别被设为右移动操作和左移动操作。这些可移动操作使实现中的许多位置都能进行化简。具体而言，它们使失效循环（invalidatesharers）能够进行并行归约，从而将其顺序化；随后，还能对整个过程dir read exc req 的主体进行顺序归约，以便将一个内存地址载入缓存并设为独占模式。图 15 对此作了精确说明。

这种化简有助于完成精化证明，从而隐藏目录和所有缓存，使缓存上的读取和写入操作成为对absMem执行的原子操作的精化。

8.3 两阶段提交 (2PC)

两阶段提交是一种经典的分布式协议，用于实现并发事务。多个协调器进程会让多个副本就并发事务的顺序达成一致。每个事务都关联一个开始时间和结束时间，并提交给一个协调器。如果两个事务的时间区间重叠，则它们互相冲突。目标是确保所有已提交事务两两之间都不冲突。

协调器分两个阶段运行。在投票阶段，它向所有副本发送投票请求，副本则回复YES 或NO （接受或不接受某个事务）。副本会将其已投票YES 但仍待处理（尚未提交或中止）的事务集合存入所谓的锁定集合，并且仅当收到的投票请求涉及的事务与锁定集合中的所有事务都不冲突时，才回答YES 。在最终确定阶段，如果所有回复都是YES，协调器就发送提交请求；否则发送中止请求。如果副本收到中止请求，就会将该事务从锁定集合中移除。

规范。 我们添加一个幽灵变量已提交事务，用于跟踪所有已提交的事务。在将事务添加到此集合之前（在协调器的代码中），我们断言它不会与任何先前已提交的事务发生冲突。

应用化简。 为启用化简，我们对投票请求处理程序进行抽象，使其能够非确定性地返回NO ，而不修改状态。这种抽象使投票请求处理程序成为右移动操作，而中止请求处理程序则成为左移动操作。提交请求处理程序是双向移动操作，因为它不会改变状态。举例来说，考虑由同一副本处理的两个连续投票请求（在不同副本上的请求可交换，因为它们访问的状态互不相交）。如果事务发生冲突，一个处理程序可能返回YES ，另一个则返回NO。没有这种抽象时，重新排列这些处理程序是不可靠的：如果第二个先执行，它可能会返回YES，这会

破坏可交换性。不过，有了这种抽象，第二个处理程序就可以非确定性地返回NO，从而允许重新排列执行顺序，同时第一个处理程序仍返回YES。类似的推理也适用于其他处理程序组合。

与前述情况一样，抽象使并行归约与顺序归约得以结合。如下文以投票请求为例，并行归约用于将两个阶段顺序化；随后，这种顺序化使我们能够证明事务的整个计算可以在一个原子区段内执行。

```
right procedure vote_all(xid: TransactionId, i:
     ReplicaId)
  returns (votes: [ReplicaId]Vote) {
  ...
  if (1 <= i) {
    vr = VoteRequest(xid, i);
    par-reduce {
     (call votes := vote_all(xid, i-1))
     par (call out := vote(vr)); } // right
    }
    votes[i] := out;
}}
```

```
procedure TPC(xid: TransactionId) {
  ...
  seq-reduce {
    call votes := vote_all(xid, n);
     // right
    // locally calculate decision
     based on votes
    call finalize_all(decision,
     xid); // left
  }
  ...
}
```

图 16：全部投票和 TPC 过程。

9 相关工作

我们回顾了有关使用交换性推理来证明并发或分布式系统正确性的研究。

演绎验证中的交换性推理。Lipton 归约理论 [34]引入了移动操作 这一概念，用以定义一种能够生成有界大小原子块的程序变换。这项工作假设使用一种不支持过程调用的简单编程语言，并且线程数量固定。QED [11]通过引入对原子动作和抽象的迭代应用，扩展了 Lipton 理论的适用范围。此外，原子区段可以包含循环，但不能包含过程调用或动态线程创建。Civl [22]基于 QED 的基础，加入了不变式 [37,24], 精化层，并通过线性类型系统 [27], 和待处理异步调用 [29,26]实现基于权限的推理。待处理异步调用可以视为仅限于执行单个原子步骤且无法合并的线程。它们用于概括异步过程调用，并定义一种归约方案，将异步过程调用转换为同步过程调用 [29]。该归约方案基于以下证明：被异步调用的过程可以概括为一个左移动待处理异步调用。此思路已通过归纳原理 [26]扩展，用于将创建无界数量待处理异步调用的异步程序顺序化。在这项工作中，我们引入了更灵活的归约方案，以提升可扩展性。这些方案允许原子区段包含顺序和并行过程调用，从而支持更强的组合性。此外，它们还通过支持基于左移动和右移动操作的交换重排，扩展了归约能力。

Anchor [15]将化简应用于一种低级面向对象语言，其中为对象字段的读取和写入访问分配移动操作注释。该方法引入了一种类型系统，用于证明整个过程的原子性，

该方法建立在 Lipton 归约之上。相比之下，我们的工作基于一种更抽象的语言，支持关于过程的组合式归约推理，并考虑并行组合。 [16] 研究了将归约与依赖-保证推理相结合的问题，而这超出了本文的范围。

CSPEC [8] 采用了与 Civl 类似的方法，但在 Rocq 定理证明器 [38] 中将整个元理论机制化，从而实现灵活性和可靠的可扩展性。Armada [35] 也拥有灵活且机制化的元理论，其效用通过实现多种程序变换得以展现，其中包括适用于细粒度并发和弱内存模型的变换。Iron- Fleet [21] 将 TLA 风格状态机建模 [32] 嵌入 Dafny 验证器 [33] ，以将高层分布式系统规约细化为低层可执行实现。它们的证明以相当临时的方式将化简推理嵌入 Dafny。

移动操作也被用于定义一种保持等价性的变换，以消除消息传递程序 [6,39] 中的缓冲区。这些工作定义了一类受限的程序，并证明：对这类程序的会合 执行集合进行推理是完备的，也就是说，任何其他执行都等价于某个会合执行，区别至多在于移动子动作的重新排序。例如， [39] 引入了若干基于语法的启发式方法，以化简给定程序。这些启发式方法不适用于我们的案例研究，而且很难设想怎样扩展它们才能使其适用。例如，有时可以通过抽象动作（消息处理程序）来实现化简，而这无法通过语法论证来处理。两阶段提交（2PC）是这类工作的典型基准，已在多种系统中多次得到验证，其中也包括自动化系统 [10,39]。在我们的 2PC 中，副本使用非平凡逻辑来确定其投票，而这些系统所用的版本并非如此。在先前的工作中，副本以非确定性方式投赞成 或反对 票，这大大简化了正确性论证：所有消息处理程序都是左移动操作，无需进行任何抽象 [29]。相比之下，在我们的 2PC 版本中，经过适当的抽象后，有些消息处理程序是右移动操作，有些则是左移动操作。

算法验证中的交换性推理。在算法验证的背景下，交换性推理体现为所谓的偏序化简技术 [20,18,2,25] ，这类技术主要针对有限状态系统或长度有界的执行。

在面向无限状态程序的自动化证明合成领域，现有工作大多聚焦于线程数量有界的程序 [9,12,13]。文献 [14] 提出了一种用于参数化程序的插桩方案，其中无界数量的线程执行相同的代码。该方案使得在这类场景中能够表示可靠的归约。此外，作者还形式化定义了归约有效性的概念，并指出，合适的归约可以使证明所需的幽灵变量更少或更简单。

致谢。 本研究部分得到了法国国家研究署（项目 SCEPROOF）的资助。

参考文献

1. Mart ´ *Theor. Comput. Sci.* 82 2 253 284, 1991 10 1016 0304 3975 91 90224 ın Abadi 和 Leslie Lamport。精化映射的存在性。《》，（）：–。doi：。/—（）—P。2. Parosh Aziz Abdulla、Stavros Aronis、Bengt Jonsson 和 Konstantinos Sagonas。最优动态偏序约简。载于 Suresh Jagannathan 和 Peter Sewell 编，*The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*，*POPL '14*，*San Diego*，*CA*，*USA*，*January 20-21*，2014 年，第 373–384 页。ACM，2014 年。doi:10.1145/2535838.2535845。3. Yehuda Afek 、 Hagit Attiya、Danny Dolev、Eli Gafni、Michael Merritt 和 Nir Shavit。共享内存的原子快照。*J. ACM*，40(4):873–890，1993 年。doi: 10.1145/153724.153741。4. Hagit Attiya、Amotz Bar-Noy 和 Danny Dolev。消息传递系统中的稳健内存共享。 *J. ACM*，42(1):124–142，1995 年。doi:10.1145/200836.200869。5. Hagit Attiya 和 Constantin Enea。将强线性化置于上下文中：保留使用并发对象的程序中的超属性。载于 Jukka Suomela 编，*33rd International Symposium on Distributed Computing DISC*2019，*October 14-18*，2019，*Budapest*，*Hungary*，第 146 卷，收录于 *LIPIcs*，第 2:1–2:17 页。施洛斯·达格施图尔—莱布尼茨信息学中心，2019 年。网址：https://doi.org/10.4230/LIPIcs.DISC.2019.2，doi:10.4230/LIPICS.DISC.2019.2。6. Alexander Bakst、Klaus von Gleissenthall、Rami G¨ okhan Kici 和 Ranjit Jhala。通过规范顺序化验证分布式程序。*Proc. ACM Pro-* ，(*gram. Lang.* 1 110 1 110 27, 2017 10 1145 3133934 OOPSLA): : – : 。doi：。/。7. Michael Barnett、Bor-Yuh Evan Chang、Robert DeLine、Bart Jacobs 和 K. Rus-tan M. Leino。Boogie：一种用于面向对象程序的模块化可复用验证器。载于 Frank S. de Boer、Marcello M. Bonsangue、Susanne Graf 和 Willem P. de Roever 编，*Formal Methods for Components and Objects*，*4th International Symposium FMCO 2005 Amsterdam The Netherlands November 1-4 2005 Revised* ，，，，，，，，*Lectures* 4111 *Lecture Notes in Computer Science* 364 387 第 卷，收录于 ，第 – 页。施普林格，2005 年。doi:10.1007/11804192_17。8. Tej Chajed、M. Frans Kaashoek、Butler W. Lampson 和 Nickolai Zeldovich。使用移动操作在 CSPEC 中验证并发软件。载于 Andrea C. Arpaci-Dusseau 和 Geoff Voelker 编，*13th USENIX Symposium on Operating Systems Designand Implementation*，*OSDI 2018*，*Carlsbad*，*CA*，*USA*，*October 8-10*，2018 年，第 306–322 页。USENIX 协会，2018 年。网址：https://www.usenix.org/conference/osdi18/presentation/chajed。9. Duc-Hiep Chu 和 Joxan Jaffar。将偏序约简与状态插值协同结合的框架。载于 Eran Yahav 编，*Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference HVC 2014 Haifa Israel November 18-20 2014. Proceedings* 8855 ，，，，，第 卷，收录于 *Lecture Notes in Computer Science*，第 171–187 页。施普林格，2014 年。doi: 10.1007/978-3-319-13338-6_14。10. Cezara Drˇagoi、Thomas A. Henzinger 和 Damien Zufferey。PSync：一种用于容错分布式算法的部分同步语言。 *SIGPLAN Not.*，51 1 400 415，2016 10 1145 2914770 2837650 ( ): – January。doi: . / .。11. Tayfun Elmas、Shaz Qadeer 和 Serdar Tasiran。原子动作演算。载于 Zhong Shao 和 Benjamin C. Pierce 编，*Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL* ，，，，*2009 Savannah GA USA* *2009* 2 15 2009 *January 21-23*，，第 – 页。ACM，。doi: 10.1145/1480881.1480885。

12. Azadeh Farzan、Dominik Klumpp 和 Andreas Podelski。用于并发程序验证的可靠顺序化。载于 Ranjit Jhala 和 Isil Dillig（编）， *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17*，2022年，第 506– 521 页。ACM，2022年。doi:10.1145/3519939.3523727。13. Azadeh Farzan、Dominik Klumpp 和 Andreas Podelski。并发程序验证算法中的分层交换性。 *Proc. ACM Program. Lang.*，7 1426 1453，2023 10 1145 3571242 (POPL): – 。doi: . / . 14. Azadeh Farzan、Dominik Klumpp 和 Andreas Podelski。交换性简化参数化程序的证明。 *Proc. ACM Program. Lang.*，8(POPL):2485– 2513，2024年。doi:10.1145/3632925。15. Cormac Flanagan 和 Stephen N. Freund。用于阻塞与非阻塞并发软件的 Anchor 验证器。 *Proc. ACM Program. Lang.*，4(OOPSLA):156:1– 156:29，2020年。doi:10.1145/3428224。16. Cormac Flanagan 和 Stephen N. Freund。移动器逻辑：用于化简和依赖-保证推理的并发程序逻辑。载于 Jonathan Aldrich 和 Guido Salvaneschi（编）， *38th European Conference on Object-OrientedProgramming, ECOOP 2024, September 16-20*，2024年， *Vienna, Austria*，第 313 卷，收录于 *LIPIcs*，第 16:1– 16:29 页。Schloss Dagstuhl - Leibniz 信息学中心，2024年。URL: https://doi.org/10.4230/LIPIcs.ECOOP.2024.16，doi:10. 4230/LIPICS.ECOOP.2024.16。17. Cormac Flanagan、Stephen N. Freund 和 Shaz Qadeer。利用纯性实现原子性。 *IEEE Trans. Software Eng.*，31(4):275–291，2005年。doi:10.1109/TSE. 2005.47。18. Cormac Flanagan 和 Patrice Godefroid。用于软件模型检测的动态偏序约简。载于 Jens Palsberg 和 Mart´ın Abadi（编）， *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of ProgrammingLanguages, POPL 2005, Long Beach, California, USA, January 12-14*，2005年，第 110–121 页。ACM，2005年。doi:10.1145/1040305.1040315。19. Cormac Flanagan 和 Shaz Qadeer。用于原子性的类型与效应系统。载于 Ron Cytron 和 Rajiv Gupta（编）， *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003 USA June 9-11 2003* 338 349 2003 ， *San Diego, California*, , , , 第 – 页。ACM，。doi:10.1145/781131.781169。20. Patrice Godefroid。 *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem* 1032 *Lecture Notes in Computer Science* ，第 卷，1996 10 1007 3 540 60761 7 收录于 。施普林格，。doi: . / - - - 。21. Chris Hawblitzel、Jon Howell、Manos Kapritsos、Jacob R. Lorch、Bryan Parno、Michael Lowell Roberts、Srinath T. V. Setty 和 Brian Zill。Ironfleet：证明实用分布式系统的正确性。载于 Ethan L. Miller 和 Steven Hand（编）， *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey CA USA October 4-7 2015* 1 17 2015 10 1145 , , , ,, 第 – 页。ACM，。doi: . / 2815400.2815428。22. Chris Hawblitzel 、 Erez Petrank 、 Shaz Qadeer 和 Serdar Tasiran。面向并发程序的自动化模块化精化推理。载于 Daniel Kroening 和 Corina S. Pasareanu（编）， *Computer Aided Verification - 27th International Conference CAV 2015 San Francisco CA USA July 18-24 2015 Proceedings Part II* 9207 *Lecture Notes in Computer Science* 449 465 , , , , , , , ,, 第 卷，收录于 ，第 – 页。施普林格，2015年。doi:10.1007/978-3-319-21668-3_26。

23. Maurice Herlihy 和 Jeannette M. Wing。线性化：并发对象的一种正确性条件。 *ACM Trans. Program. Lang. Syst.*，12(3):463–492，1990年。doi:10.1145/78969.78972。
24. Cliff B. Jones。程序的规范与设计（并行）。载于 R. E. A. Mason（编），
Information Processing 83 Proceedings of the IFIP 9th World ，，，，，第 – 页。*Computer Congress Paris France September 19-23 1983* 321 332 North-Holland/IFIP，1983年。25. Michalis Kokologiannakis、Iason Marmanis、Vladimir Gladstein 和 Viktor Vafeiadis。真正无状态的最优动态偏序约简。
Proc. ACM ，(POPL): – 。*Program. Lang.* 6 1 28, 2022 10 1145 3498711 doi: . / . 26. Bernhard Kragl、Constantin Enea、Thomas A. Henzinger、Suha Orhun Mutluergil 和 Shaz Qadeer。异步程序的归纳式串行化。载于 Alastair F. Donaldson 和 Emina Torlak（编），*Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation PLDI 2020 London UK June 15-20 2020* 227 242 2020 ，，，，，，第 – 页。ACM，。doi:10.1145/3385412.3385980。27. Bernhard Kragl 和 Shaz Qadeer。分层并发程序。载于 Hana Chockler 和 Georg Weissenbacher（编），
Computer Aided Verification - 30th In-ternational Conference, CAV 2018, Held as Part of the Federated Logic Con-ference, FloC 2018, Oxford, UK, July 14-17，2018年， *Proceedings, Part I*，第 10981 卷，收录于 *Lecture Notes in Computer Science*，第 79–102 页。施普林格，2018年。doi:10.1007/978-3-319-96145-3_5。28. Bernhard Kragl 和 Shaz Qadeer。Civl 验证器。载于 *Formal Methods inComputer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22*，2021 143 152 2021 10 34727 2021，第 – 页。IEEE，。URL: https://doi.org/ . / /isbn. 978-3-85448-046-4 23, doi:10.34727/2021/ISBN.978-3-85448-046-4_23。29. Bernhard Kragl、Shaz Qadeer 和 Thomas A. Henzinger。同步异步程序。载于 Sven Schewe 和 Lijun Zhang（编），
29th International Con-ference on Concurrency Theory, CONCUR 2018, September 4-7，2018年， *Beijing, China* 118 *LIPIcs* 21 1 21 17 ，第 卷，收录于 ，第 : – : 页。Schloss Dagstuhl - Leibniz 信息学中心，2018年。URL: https://doi.org/10.4230/LIPIcs.CONCUR.2018.21，doi:10.4230/LIPICS.CONCUR.2018.21。30. Bernhard Kragl、Shaz Qadeer 和 Thomas A. Henzinger。结构化并发程序的精化。载于 Shuvendu K. Lahiri 和 Chao Wang（编），
Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles CA USA July 21-24 2020 Proceedings Part I 12224 *Lec-* ，，，，，，第 卷，*ture Notes in Computer Science* 275 298 2020 10 1007 收录于 ，第 – 页。施普林格，。doi: . / 978-3-030-53288-8_14。31. Jeffrey Kuskin、David Ofelt、Mark A. Heinrich、John Heinlein、Richard Simoni、Kourosh Gharachorloo、John Chapin、David Nakahira、Joel Baxter、Mark Horowitz、Anoop Gupta、Mendel Rosenblum 和 John L. Hennessy。斯坦福 FLASH 多处理器。载于 David A. Patterson（编），*Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago IL USA* ，，，
April 1994 302 313 1994 10 1109 ，第 – 页。IEEE 计算机学会，。doi: . /ISCA. 1994.288140。32. Leslie Lamport。
Specifying Systems The TLA+ Language and Tools for Hardware and Software Engineers ，。2002 Addison-Wesley，。URL: http://research.microsoft. com/users/lamport/tla/book.html。33. K. Rustan M. Leino。Dafny：用于函数正确性的自动程序验证器。载于 Edmund M. Clarke 和 Andrei Voronkov（编），*Logic for Pro-*

gramming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010年, *Revised Selected Papers, Lecture Notes in Computer Science* 第 6355 卷，348–370 页。施普林格，2010年。doi:10.1007/978-3-642-17511-4_20。34. Richard J. Lipton。《化简：证明并行程序性质的一种方法》。*Commun. ACM*，18(12):717–721，1975年。doi:10.1145/361227.361234。35. Jacob R. Lorch、Yixuan Chen、Manos Kapritsos、Bryan Parno、Shaz Qadeer、Upamanyu Sharma、James R. Wilcox 和 Xueyuan Zhao。《Armada：低成本验证高性能并发程序》。载于 Alastair F. Donald- son 和 Emina Torlak（编），*Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020 London UK June 15-20 2020* 197 210 2020 , , , , , 第 – 页。ACM，。doi: 10.1145/3385412.3385971。36. Nancy A. Lynch 和 Frits W. Vaandrager。《正向与反向模拟：I. 无时钟系统》。 *Inf. Comput.*，121(2):214–233，1995年。URL: https://doi.org/10. 1006/inco.1995.1134，doi:10.1006/INCO.1995.1134。37. Susan S. Owicki 和 David Gries。《验证并行程序的性质：一种公理化方法》。*Commun. ACM*，19(5):279–285，1976年。doi:10.1145/360051. 360224。38. Coq 开发团队。《Coq 证明助手》，8.11.0 版，2020年。doi: 10.5281/zenodo.3744225。39. Klaus von Gleissenthall、Rami G¨okhan Kici、Alexander Bakst、Deian Stefan 和 Ranjit Jhala。《假装同步：异步分布式程序的同步验证》。 *Proc. ACM Program. Lang.*，3(POPL):59:1–59:30，2019年。doi:10.1145/3290372。

A 第6节证明（RedPL 程序的化简）

A.1 并行归约

本节将给出定理 1 证明的第一部分。

我们证明，所有并行归约的出现都可以顺序化，即 $\mathcal{P}_s$细化 $\mathcal{P}_i$。我们证明，定义精化的两个性质（P1）和（P2）均成立。

证明精化性质（P1）给定 $\mathcal{P}_s$ 的一个失败执行 π ，我们将其改写为 $\mathcal{P}_i$中的一个失败执行 π'，使得在改写后的执行中，每个并行归约实例 s_1 par s_2 都满足以下两个要求：

1. 顺序要求： s_1 的所有转换都必须先于 s_2的所有转换。 2. 完成：如果存在 s_2 的转换，那么 s_1 必须已完整执行。

其思路是首先满足其中不再嵌套任何其他并行归约语句的实例的这两个要求。之后，我们继续满足嵌套实例已经满足要求的实例，依此类推，直到所有实例都满足这两个要求。

给定一个执行过程，我们可以按如下方式识别一个不再包含任何嵌套应用的并行归约应用：在并行归约进入转换所产生的配置中，将指针放在新引入的内部节点（Nd*）上。如果在执行过程的剩余部分中，该节点（Nd*）的任何后代节点都没有触发并行归约进入转换，那么此实例就不再包含任何嵌套的并行归约应用。

令 par-reduce s_1 par s_2 为这样一个实例。要使其成为良类型语句，需要满足以下与移动类型相关的条件：移动类型(s_1) $\sqsubseteq \mathbf{L}$ ∨(移动类型 (s_2) $\sqsubseteq \mathbf{R}$ ∧ ¬可能-失败(s_2))。首先，考虑 移动类型(s_1) $\sqsubseteq \mathbf{L}$ 的情况，我们称之为左归约。

满足顺序要求（左归约）除了顺序要求之外，在左归约的情况下，我们还需要 s_1 的迁移互不干扰（连续发生），以确保我们的完成过程（稍后解释）能够终止。因此，在这种情况下，我们在进行顺序调整时，也会消除 s_1的迁移之间的干扰。

为了满足顺序要求，我们从 s_1 中最左侧且顺序不正确的迁移开始，尝试逐步向左移动它（过程中可能会将它替换为另一个迁移），直到它到达正确位置。我们继续这一过程，直到 s_1 的所有迁移都符合顺序要求；每次都选取 s_1 中下一个尚未符合顺序要求的最左侧迁移，并将其放在前一个已排序迁移之后。

逐步向左移动时，唯一值得关注的情况是两个动作转换相邻；否则，由于要交换的转换互不影响，交换是显而易见的。因此，我们只考虑这种情况。设 $L = (\ , \rho_L, \ , M_L,)$ 是我们试图向左移动其迁移 (t_L) 的动作，设 X 是其迁移 (t_X)

紧邻在 t_L 之前的动作。设 ℓ_L =输入-绑定(t_L) ℓ_X =输入-绑定(t_X) 分别是它们的输入绑定。设 ρ_L 和 ρ_X 分别是它们各自的门。

注意，由于可移动操作-类型(s_1) $\sqsubseteq \mathbf{L}$，动作 L 必须具有左移动者性质，即 $M_L \sqsubseteq \mathbf{L}$。

我们的目标是在保留从初始状态出发的失败结果的同时，重新排列执行顺序；不要求保留中间状态，而且重新排列后的执行可能更短。

情形 1： t_L 是一次失败转移：

$$(g_1, \mathcal{T}_1) \xrightarrow{t_X} (g_2, \mathcal{T}_2) \xrightarrow{t_L} \lightning$$

根据保留-成功(X， L)，我们知道，只要某个状态满足 X 的门和 L的门，那么 X 的任意一次迁移都会到达一个满足 L 的门的状态。利用这一条件的逆否命题，我们得到：如果从 (g_1， $\mathcal{T}_1$) 出发， X 的一次迁移到达状态 (g_2， $\mathcal{T}_2$)，且该状态不满足 L 的门，那么在 (g_1， $\mathcal{T}_1$) 中， X 的门或 L 的门至少有一个不满足。由于在我们的情形中 $g_1{\cdot}\ell_X \in \rho_X$，因此必定有 $g_1{\cdot}\ell_L \notin \rho_L$。所以，我们可以从 ($g_1$， $\mathcal{T}_1$) 得到 L 的失败，并且可以将 t_X 从执行中删去。

情形 2： t_L 是一次非失败转移：

$$(g_1, \mathcal{T}_1) \xrightarrow{t_X} (g_2, \mathcal{T}_2) \xrightarrow{t_L} (g_3, \mathcal{T}_3)$$

情形 2.1：(g_1， ℓ_X， ℓ_L) $\in wlp$(X， ρ_L)，且 (g_1， ℓ_L， ℓ_X) $\in wlp$(L， ρ_X)。
存在一种执行方式，先执行 L，再执行 X，并到达状态 (g_3， $\mathcal{T}_3$)；根据交换(X， L) (L3) 左移动者的性质，从而保留 π 中的原始失败。

情形 2.2：（g_1， ℓ_X， ℓ_L） $\notin wlp$（X， ρ_L）。
我们已经有了 $g_1{\cdot}\ell_X \in \rho_X$，这意味着可以从 ($g_1$， $\mathcal{T}_1$) 执行 X，并得到 L 的失败转移。因此，我们有一个 X 的转移，随后是 L 的失败转移，这与情形 1 相同。

情形 2.3：（g_1， ℓ_L， ℓ_X） $\notin wlp$（L， ρ_X）。
若 $g_1{\cdot}\ell_L \notin \rho_L$，则我们从 ($g_1$， $\mathcal{T}_1$) 执行 L 并导致失败，从而消除t_X；否则，若 $g_1{\cdot}\ell_L \in \rho_L$，则存在一种从 ($g_1$， $\mathcal{T}_1$) 运行 L 的方式，并导致 X 的一次失败迁移。

令 π_l 为所得的执行，其中 s_1 的所有转换均已排序且无干扰。

满足完成要求（左归约）假设在 π_l 中， s_1 未完全执行，且 s_2 至少有一次迁移。在这种情况下，我们需要完成 s_1。令 $seq(s) = s[s'; s'' \;/\; 并行归约\; s' 并行\; s'']$。为了完成 s_1，我们选择与 $seq(s_1)$ 匹配的调度顺序。 s_1 中尚未执行的部分可能包含并行归约语句，但这些语句都尚未开始（否则就会出现嵌套并行规约）。因此， s_1 在 π_l 中已执行的转换与 $seq(s_1)$ 的一个前缀匹配；接下来，我们通过扩展 s_1 的执行来继续，使其转换始终与 $seq(s_1)$ 的一个前缀匹配。

设失败前执行的最后一个状态为 $(g, \mathcal{T})$，并设 t_X 为动作 X 从该状态出发的失败转移。从 $(g, \mathcal{T})$ 开始，我们允许 s_1

执行，并进行与$seq(s_1)$ 的调度顺序一致的迁移。

如果这是一个非动作转换，那么可以在保留失败的同时轻易地引入它。因此，我们假设它是一个动作转换 t_L，对应于动作 L。引入它时，可能出现以下两种情形之一：

情形 1：t_L 导致失败。
这样我们就保留了失败。
情形 2：t_L 不会失败。
根据左移动者的保留-失败$(L,\ X)$ (L2) 性质，如果 X 从一个使 L 不失败的状态出发会失败，那么就存在一种执行 L 并使 X 失败的方法。我们利用这一性质，从 $(g,\ \mathcal{T})$ 执行 L，到达一个使 X 失败的状态。

在引入 t_L 并保留失败的同时，我们现在可以像之前一样对其排序，使其位于正确位置。我们继续这一过程，直到 s_1 完全执行，或在排序过程中消除 s_2 的所有转换。

完成的终止性 下面说明完成 s_1 的过程为何会终止。s_1 中唯一可能导致非终止的因素是过程调用。假设 s_1 中有一个对某个过程 Q 的调用。由于 $s_1 \sqsubseteq \mathbf{L}$，因此 可移动操作-类型$(Q) \sqsubseteq \mathbf{L}$。根据 $\mathcal{P}_i$ 是终止的这一假设，我们知道，$\mathcal{P}_i(Q)$ 从任意带有输入绑定的存储开始单独运行（即不受干扰）时会终止。鉴于 $seq(\mathcal{P}_s(Q)) = \mathcal{P}_i(Q)$，在完成 s_1 的同时执行过程调用 Q 也会终止。

因此，所得执行满足此实例的顺序要求和完成要求。

现在我们考虑 可移动操作-类型$(s_2) \sqsubseteq \mathbf{R}$ 的情形。我们称此情形为右归约。

满足顺序要求（右归约）为了满足顺序要求，我们的策略是从最右侧的乱序迁移 s_2开始，将其逐步向右移动，直到它处于正确的位置。然后，我们继续选择下一个最右侧的乱序迁移，重复这一过程，直到所有迁移都排列正确。

在逐步向右移动的过程中，唯一值得关注的情况仍然是两个动作转换相邻。设 $R = (\ , \rho_R, , M_R,)$ 是我们试图向右移动其迁移（t_R）的动作，X 是其迁移（t_X）紧接在 t_R 之后的动作。设 $\ell_R =$输入-绑定(t_R) $\ell_X =$输入-绑定(t_X) 分别是它们的输入绑定，ρ_R 和 ρ_X 分别是它们的门。

由于可移动操作-类型$(s_2) \sqsubseteq \mathbf{R}$，动作 R 必须具有右移动性质，即 $M_R \sqsubseteq \mathbf{R}$。

情形 1： t_X 是一次失败转移

$$(g_1, \mathcal{T}_1) \xrightarrow{t_R} (g_2, \mathcal{T}_2) \xrightarrow{t_X} \lightning$$

在这种情况下，使用右移动操作的保持-成功$(R,\ X)$（R1）性质的逆否形式（类似于我们在左归约的情形 1 中对其的使用）

我们可以使 X 从（g_1， $\mathcal{T}_1$）中失败，从而消除动作 R 的迁移。

情形 2： t_X 是非失败转移

$$(g_1, \mathcal{T}_1) \xrightarrow{t_R} (g_2, \mathcal{T}_2) \xrightarrow{t_X} (g_3, \mathcal{T}_3)$$

情形 2.1：(g_1， ℓ_X， ℓ_R) $\in wlp$(X， ρ_R) 和 (g_1， ℓ_R， ℓ_X) $\in wlp$(R， ρ_X)。然后，可以先执行 X，再执行 R，并通过 交换(R， X) (R2) 到达状态 (g_3，$\mathcal{T}_3$)，从而保留 π 中的原始失败。**情形 2.2**：(g_1， ℓ_X， ℓ_R) $\notin wlp$(X， ρ_R)。

根据 ¬可能-失败(s_2)，我们知道动作 R 不会失败，因此有 wlp(X， ρ_R) $=\rho_X$。由此可知 $g_1\ell_X \notin \rho_X$，因此从 (g_1， $\mathcal{T}_1$) 执行 X 会导致 X 失败，从而消除 R 的迁移。

情形 2.3：(g_1， ℓ_R， ℓ_X) $\notin wlp$(R， ρ_X)。根据 ¬may-失败(s_2)，我们知道每个状态都满足 ρ_R，因此 ($g_1 \cdot \ell_R$) $\in \rho_R$。所以，必定存在一种执行 R 的方式，使其从 (g_1， $\mathcal{T}_1$) 执行并产生 X 的失败。现在，我们进入情形 1。

令 π_r 为所有 s_1 的转换都已排序后得到的执行。

满足完成要求（右归约）假设在 π_r中， s_1 尚未完成，且 s_2至少有一个转换。由于 s_1 可能是不可移动操作，我们无法通过以保留失败的方式扩展执行来完成它。因此，我们的策略是从执行中移除 s_2 的所有转换，以满足完成要求。请注意，如果失败转移属于 s_2，那么我们就无法移除 s_2的转换。这就是为什么类型检查要求 ¬may-失败(s_2)。现在可以假设失败转移不属于 s_2。从 s_2 的最后一个转换开始，我们逐步将其向右移动，直到将其移除。与之前一样，通过应用排序过程来实现这一点，并将正确位置设在失败转移之后。我们继续这一过程，直到s_2 的所有转换都被移除。

现在，我们得到了一次执行，其中单个并行归约应用实例（不包含任何进一步嵌套的并行归约应用）的顺序要求和完成要求均已满足。现在，我们可以继续以类似方式调整其他此类实例，包括嵌套实例已处理完毕的实例。一旦某个实例经过调整而满足顺序要求和完成要求，即使调整其他实例，这些性质仍会保持。原因如下：

1. 顺序要求不会被违反，因为在调整某个特定实例时，我们从不交换两个都属于其他实例的迁移。因此，其他实例的迁移之间的相对顺序保持不变。2. 关于完成，唯一值得担心的是，在调整另一个实例时，执行可能会因引入失败而缩短。结果，我们可能会失去某个已调整实例的后缀。不过，这并不会违反完成要求，因为对于满足顺序要求和完成要求的实例的所有前缀，完成要求都成立。

因此，当所有实例都处理完毕后，所有实例都满足这两项要求。

展示精化性质（P2）给定一个从初始配置（g，$\mathcal{T}$）开始并以最终配置（g'，$\varnothing$）结束的执行 π of $\mathcal{P}_s$，我们旨在证明存在一个执行 π' in $\mathcal{P}_i$，它从（g，$\mathcal{T}$）开始，并以相同的最终配置结束，或者以失败告终。

$$(g, \mathcal{T}) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \quad \Longrightarrow \quad \left((g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} (g', \varnothing) \quad \vee \quad (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} \lightning\right)$$

我们可以通过应用相同的重写过程来强制对 π 施加所需的顺序要求，从而证明这一点。只有当重排引入失败时，才需要完成。如果没有引入失败——也就是说，左右归约都仍处于情形 2.1——那么最终状态（g'，$\varnothing$）会由左右移动操作的交换性质保持不变。否则，执行将以失败告终。

A.2 顺序归约

现在我们证明定理 1 的第二部分。

我们证明，seq-reduce 内部的所有代码块都可以变为原子操作，即 $\mathcal{P}_i$精化 $\mathcal{P}_r$。

展示精化性质（P1）给定一个失败执行 π of $\mathcal{P}_i$，我们将其改写为一个失败执行 π' of $\mathcal{P}_r$，使得在改写后的执行中，对于每个形式为 seq-reduce s 的顺序规约应用，以下两项要求均得到满足：

1. 无干扰：不存在其他转换与语句 s 的转换交错。 2. 完成：如果改写后的执行中至少存在一个语句 s 的转换，那么语句 s 必须完整执行或以失败结束。

为了满足每个实例的无干扰和完成要求，我们的策略是让每个最外层实例（即未嵌套在任何其他顺序规约应用中的实例）都满足这两项要求。一旦最外层实例满足了这两项要求，所有嵌套实例也会满足，因为它们的转换被视为最外层实例的一部分。

我们可以按如下方式识别顺序规约应用中的最外层实例：在顺序规约进入转移所产生的配置中，我们标记引入内部顺序规约的叶节点（Lf*）。如果该节点以及从 Lf* 到根节点的路径上的任何节点中都不存在内部顺序规约，那么我们就识别出了一个最外层应用。

满足无干扰要求 我们设计了一种针对迁移的交换策略：将迁移划分为若干区块，并根据块类型执行交换，从而在无法再进行交换时，使所得执行满足无干扰要求。

区块 我们将属于特定最外层实例的语句 s 的所有迁移视为一个区块的一部分。

根据 seq-reduce$s$ 类型良好这一事实可知，移动类型(s) $\sqsubseteq \mathbf{N}$，这意味着语句s的任意执行路径中的动作移动类型都符合模式 $\mathbf{R}^*\mathbf{N?L}^*$。

区块的提交转换 我们将区块的提交转换定义如下（如果存在）：

– 如果区块包含一个非右移动作转换，则提交转换是该区块中的第一个此类转换。– 否则，如果语句 s 执行完毕或以失败结束，则提交转换分别是区块中 s 的最后一个转换或失败转换。

所有其他未应用 seq-reduce 的转换均标记为提交转换，并且每个此类转换都被视为单独构成一个区块。

区块没有提交转换的唯一情况是：它包含一个来自 seq-reduce 实例应用的语句，并且该应用 (1) 不包含任何非右移动作转换，(2) 未能执行完毕，且 (3) 并未以失败结束。具有提交转换的区块称为已提交区块；否则称为未提交区块。

区块中转换的移动者标记 在每个已提交块中，我们为提交转换分配非移动者标签（$\mathbf{N}$）。提交转换之前的所有转换均分配右移动者标签（$\mathbf{R}$），而提交转换之后的所有转换均分配左移动者标签（$\mathbf{L}$）。未提交块中的所有转换均分配右移动者标签。对于给定的迁移 t let$label$(t) 表示分配给它的标签。请注意，对于动作 t_A of an action A 的任何动作转换，其移动类型 A 至少与 t_A 的标签一样精确（即，移动者 -类型(A) $\sqsubseteq$标签(t_A)）。例如，如果某个动作的移动类型为 $\mathbf{R}$，我们会将该转换标记为 $\mathbf{R}$ 或 $\mathbf{N}$，而不会标记为 $\mathbf{L}$。完成标记后，我们发现每个已提交块的标记都符合模式$\mathbf{R}^*\mathbf{NL}^*$，其中唯一的 $\mathbf{N}$ 表示提交转换；而每个未提交块的标记都符合模式 $\mathbf{R}^+$。

我们定义迁移上的两个谓词：如果迁移t_X 属于已提交块，则 committed(t_X)成立；如果迁移 t_X属于未提交块，则 uncommitted(t_X)成立。我们可以通过 block(t_X) 获取迁移 t_X 所属的块，并通过 block(t_X)。commit 获取对应的提交迁移。

交换条件 令 $<_E$ 表示迁移上的执行顺序。我们在未提交块上定义一个任意的全序 $<_O$，并使其保持固定。

执行中连续发生的两个迁移 t_A 和 t_B 是可交换 的，当且仅当满足以下条件之一：

1. 未提交(t_A) ∧已提交(t_B) 2. 已提交(t_A) ∧已提交(t_B) ∧区块(t_B).提交 $<_E$ 区块(t_A).提交

3. uncommitted(t_A) ∧uncommitted(t_B) ∧区块(t_B) $<_O$ 区块(t_A)。

引理1。 如果 t_A和 t_B可交换，那么要么标签(t_A) $=\mathbf{R}$，要么标签 (t_B) $=\mathbf{L}$。

证明。 如果它们满足条件 (1) 或 (3)，那么 t_A 必须具有右移动者标签，因为所有未提交转换都具有右移动者标签。如果它们满足条件 (2)，那么条件 (2) 中的 block(t_B).commit $<_E$ block(t_A).commit 合取式只有在 t_A 具有右移动者标签或 t_B具有左移动者标签时才成立。否则，我们就违反了条件 (2)。

1. 如果我们选择 标签(t_A) $=\mathbf{L}$和 标签 (t_B) $=\mathbf{R}$，那么根据提交转换的定义，block(t_A).commit $<_E$ t_A和 t_B $<_E$ block(t_B).commit。由于每个区块都具有 $\mathbf{R}^*\mathbf{N}\mathbf{L}^*$ 的形式，我们得到

$$\mathsf{block}(t_A).\mathsf{commit} <_E t_A <_E t_B <_E \mathsf{block}(t_B).\mathsf{commit},$$

   这违反了条件 (2)。
2. 如果我们选择标签(t_A) $=\mathbf{N}$ 和标签(t_B) $=\mathbf{R}$，那么

$$t_A = \mathsf{block}(t_A).\mathsf{commit} <_E t_B <_E \mathsf{block}(t_B).\mathsf{commit},$$

   再次违反条件 (2)。
3. 如果我们选择标签(t_A) $=\mathbf{L}$ 和标签(t_B) $=\mathbf{N}$，那么

$$\mathsf{block}(t_A).\mathsf{commit} <_E t_A <_E \mathsf{block}(t_B).\mathsf{commit} = t_B,$$

   这也违反了条件 (2)。
4. 最后，如果我们选择标签(t_A) $=\mathbf{N}$ 和标签(t_B) $=\mathbf{N}$，那么

$$t_A = \mathsf{block}(t_A).\mathsf{commit} <_E \mathsf{block}(t_B).\mathsf{commit} = t_B,$$

   再次违反条件 (2)。

□

引理2。 交换会保留失败，且该过程会终止。

证明。 由于交换涉及局部迁移的两个迁移显然会保留失败，我们只需证明交换两个动作调用迁移 t_A和 t_B会保留失败。令 $A = (\ ,\ \rho_A、\ 、\ M_A、\ )$ 和 $B = (\ ,\ \rho_B、\ 、\ M_B、\ )$。根据我们先前的观察——如果 t_A 和 t_B 可交换，那么要么标签(t_A) $=\mathbf{R}$ ，要么 标签(t_B) $=\mathbf{L}$——我们可以得出 $M_A \sqsubseteq \mathbf{R}$ 或$M_B \sqsubseteq \mathbf{L}$成立。令 $\ell_A =$ 输入-绑定 (t_A) $\ell_B =$输入-绑定(t_B) 表示它们的输入绑定。**情形 1**： t_B 是一个失败迁移

$$(g_1, \mathcal{T}_1) \xrightarrow{t_A} (g_2, \mathcal{T}_2) \xrightarrow{t_B} \lightning$$

情形 1.1: $M_A \sqsubseteq$**R**W

e know f

从 保留-成功(A, B) 性质 (R1) 可知，右移动操作具有这样的性质：每当 A 和 B 的门在某个状态中成立时， A 从该状态出发的每个迁移都会到达一个 B 的门成立的状态。使用这一条件的逆否命题，我们得到如下结论：如果从 (g_1, $\mathcal{T}_1$) 出发，A 存在一个迁移到达状态 (g_2, $\mathcal{T}_2$)，且 B 的门不成立，那么在 (g_1, $\mathcal{T}_1$) 中，A 的门不成立，或者在 (g_1, $\mathcal{T}_1$) 中， B 的门不成立。由于在我们的情形中， A 的门在 (g_1, $\mathcal{T}_1$) 中成立，因此必定是 B 的门在 (g_1, $\mathcal{T}_1$) 中不成立。因此，我们可以消去 A，并从 (g_1, $\mathcal{T}_1$) 得到 B 的失败。

情形 1.2: $M_B \sqsubseteq$**L**

在此情形中，我们再次使用左移动操作在逆否命题中的保持-成功(A, B) （L1） 来消除 A 并获得 B 的失败（使用与前一种情形相同的推理）。

情形 2: t_B 不是失败迁移

$$(g_1, \mathcal{T}_1) \xrightarrow{t_A} (g_2, \mathcal{T}_2) \xrightarrow{t_B} (g_3, \mathcal{T}_3)$$

情形 2.1：(g_1, ℓ_A, ℓ_B) $\in wlp$(A, ρ_B) 且 (g_1, ℓ_B, ℓ_A) $\in wlp$(B, ρ_A) 由于我们有 $M_A \sqsubseteq$ R 或 $M_B \sqsubseteq$ L，因此我们使用交换(A, B) (R2) 或 (L3) 来保持终止状态。情形 2.2：(g_1, ℓ_A, ℓ_B) $\notin wlp$(A, ρ_B) 我们已经有 $g_1 \cdot \ell_A \in \rho_A$。因此，我们可以从 ($g_1, \mathcal{T}_1$) 执行 A ，并获得 B的失败。情形 2.3：(g_1, ℓ_B, ℓ_A) $\notin wlp$ (B, ρ_A)。如果 $g_1 \cdot \ell_B \notin \rho_B$ ，那么我们从 ($g_1, \mathcal{T}_1$) 执行 B 并获得失败。否则，如果 $g_1 \cdot \ell_B \in \rho_B$，则存在一种方式，可以从 ($g_1, \mathcal{T}_1$) 执行 B 并获得 A的失败。

终止性 在上述所有情况下，如果引入了新的失败，我们就会得到一条严格更短的失败执行，并在这条新执行上重新启动交换过程，因为已提交块和未提交块的集合可能已经改变。

一旦该过程不再重新启动，且已提交块和未提交块的集合稳定下来，每对迁移至多交换一次。事实上，两条迁移交换后，促成该次交换的条件就不再适用于这一对迁移。□

引理 3. 当不再有可进行的交换时，就不存在未提交块。

证明。 首先，我们证明不存在未提交块。为反证起见，假设存在一条来自未提交块的迁移。取执行中最后一条未提交迁移（t_X）。 t_X 不可能是失败的迁移，因为失败的迁移属于已提交块。这意味着在 t_X 之后存在一条迁移（称之为 t_Y）。 t_Y 不可能来自未提交块，否则 t_X 就不会是最后一条未提交迁移。因此， t_Y 必须已提交，而在这种情况下，交换操作可用，从而导致矛盾。□

现在，我们证明不存在带有干扰的块。由于我们已经证明未提交块不存在，因此可以假设执行仅由已提交块构成。我们注意到，已提交块的形式要么是 $\mathbf{R}^*\mathbf{N}$，要么是 $\mathbf{R}^*\mathbf{N}\mathbf{L}^*$。如果一个已提交块 B 存在干扰，这意味着执行中存在另一块的一条迁移（称为干扰迁移），它在块 B 的各条迁移之间执行。如果干扰迁移在块 B 的提交转换之前执行，我们就称块 B 存在提交前干扰。在任何提交前干扰中，都可以找到块 B 的一条右标记迁移，它紧接在干扰迁移之前。如果干扰迁移在块 B 的提交转换之后执行，我们就称块 B 存在提交后干扰。在任何提交后干扰中，都可以找到一条紧接在干扰迁移之后的左标记迁移。

将块的顺序定义为它们在执行中提交转换发生的先后顺序。

引理 4. 当无法再进行交换时，不存在有提交前干扰的块。

证明。 为导出矛盾，假设存在某个块有提交前干扰。选取最右侧的块 B（其提交转换在执行中最靠右的块），使其存在提交前干扰。

设 t_R 是已提交块 B 的右标记迁移，它紧邻干扰迁移 t_X 之前。注意，t_X 也属于一个已提交块（根据引理 3）。

情形 1： t_X 的标签为**L**或**N**，则 t_R 和 t_X之间可以进行交换，因为区块(t_X)。提交 $<_E$ 区块(t_R)。提交，这与“不再有启用的交换”这一假设相矛盾。

情形 2： t_X 的标签为**R**，且区块(t_X)。提交 $<_E$ 区块(t_R)。提交，则 t_R 和 t_X之间同样可以进行交换，这与“不再有启用的交换”这一假设相矛盾。

情形 3： t_X 的标签为**R**，且区块(t_R)。提交 $<_E$ 区块(t_X)。提交，这意味着我们有 $t_X <_E$ 区块(t_R)。提交 $<_E$ 区块(t_X)。提交，也就是说，我们在区块 t_X中存在提交前干扰，其中干扰迁移为区块(t_R)。提交，这与“区块 B 是具有提交前干扰的最右区块”这一假设相矛盾。□

引理 5。 当不再有启用的交换时，不存在具有提交后干扰的区块。

证明。 假设存在一个发生提交后干扰的区块。选取最左侧的此类区块 B（即提交迁移在执行中最靠左的区块），且该区块发生提交后干扰。

令 t_L 为紧跟在干扰迁移 t_X 后面的左标记迁移。注意， t_X 也属于一个已提交块（根据引理 3）。

情形 1： t_X 的标签为 **R** 或**N**，
那么可以进行交换，因为区块(t_L)。提交 $<_E$ 区块(t_X)。提交，这与不再有可执行交换的假设相矛盾。

情形 2： t_X 具有标签 **L**，且区块(t_L)。提交 $<_E$ 区块(t_X)。提交，那么 t_L 和 t_X 之间仍然可以进行交换，这与不再有可执行交换的假设相矛盾。

情况 3： t_X 具有标签 **L**，且区块(t_X)。提交 $<_E$ 区块(t_L)。提交意味着我们有区块(t_X)。提交 $<$ 区块(t_L)。提交 $<_E t_X$，这意味着区块(t_X)中存在提交前干扰，其中干扰迁移是区块(t_L)。提交，这与假设相矛盾：区块 B 是最左侧存在提交前干扰的区块。□

引理6。 当不再允许交换时，不存在未提交块，也不存在有干扰的区块。

证明。 当不再允许交换时，根据引理 3，我们得知不存在未提交块。利用引理 4 和引理 5，我们可以得出不存在有干扰的已提交块。由于执行中的所有块都是已提交的（根据引理 3），因此可以断定所有块都无干扰。□

满足完成要求 完整块是指与 seq-reduce s 的某个实例相对应的块，其中属于 s 的所有转换均已执行，或者属于 s 的某个转换执行失败。否则，该块称为不完整块。我们需要完成这些不完整块。请注意，任何不完整块只需左移动者即可完成。如果它需要右移动者或非移动者，就不会有提交转换——这会使该块成为未提交块；而我们已经从执行中消除了所有未提交块。我们选择最靠左的不完整块，并按照处理并行归约的方式完成它，即满足 par-reduce 左归约情形的完成要求。完成一个块后，执行仍然没有干扰，也不存在未提交块，因此我们继续完成下一个最靠左的不完整块。

展示精化性质（P2） 给定从初始配置 $(g,\ \mathcal{T})$ 开始、以最终配置 $(g',\ \varnothing)$ 结束的 $\mathcal{P}_i$ 的一次执行 π，我们旨在证明，在 $\mathcal{P}_r$ 中存在一次执行 π'，它从 $(g,\ \mathcal{T})$ 开始，并以相同的最终配置结束或执行失败。

$$(g,\mathcal{T}) \xrightarrow{\mathcal{P}}{}^{*} (g',\varnothing) \quad \Longrightarrow \quad (g,\mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} (g',\varnothing) \quad \vee \quad (g,\mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} \lightning$$

我们可以通过对 π 应用相同的交换算法来证明这一点。如果不引入任何失败，那么我们可以保证保留终止状态（$g',\ \varnothing$），因为这源于左移动者和右移动者的交换性质。否则，我们将以失败告终。