arXiv:2304.13898v1 [cs.PL] 27 Apr 2023

使用锁不变式证明逻辑原子性*

Roshan Sharma[1], Shengyi Wang[2[0000−0002−2286−8703]]、AlexanderOey[3], Anastasiia Evdokimova[4], Lennart Beringer[2[0000−0002−1570−3492]]和 William Mansky[4[0000−0002−5351−895X]]

[1] Amazon，美国[2] 普林斯顿大学，美国[3] Cisco，美国[4] 芝加哥伊利诺伊大学，美国

摘要。逻辑原子性已被广泛接受为并发分离逻辑中数据结构的一种规约格式。尽管无锁数据结构和基于锁的数据结构都已经根据逻辑原子规约完成了验证，但后者中的大多数还同时以锁的原子规约作为起点。在本文中，我们将这种方法与基于较早的锁不变式规约的方法进行比较。我们表明，使用这些较早的规约，仍然可以为采用细粒度锁定的数据结构证明逻辑原子规约，但与使用原子锁规约的证明相比，这些证明要复杂得多。我们的证明技术已在已验证软件工具链中实现。该工具链的可靠性证明依赖于较早的锁规约；我们还将该技术应用于基于锁的并发数据结构的 C 实现。

关键词：并发分离逻辑、细粒度锁定、逻辑原子性、已验证软件工具链、Iris

1 引言

并发分离逻辑（CSL） [13] 是证明并发程序正确性的有用工具。早期的研究主要关注使用锁实现的数据结构，以及涉及内存安全和数据竞争自由的规约 [5,6], ；而较新的逻辑能够证明采用底层原子内存操作 [14,9], 的无锁实现的线性化风格的功能正确性，即使是在弱内存模型 [3]下也能做到。尤其是，逻辑原子性 [14] 已成为并发数据结构规约的黄金标准，它所表达的思想是：数据结构操作应当表现得仿佛瞬间完成，且不存在外部可见的中间状态。逻辑原子操作的效果会在线性化点变得可见；线性化点是一条发布数据结构新状态的单条指令（通常是一次原子内存操作，或对逻辑原子函数的调用）。

直观而言，基于锁的实现中的临界区，作用与无锁数据结构中的原子指令相同，而且在基于锁的实现中识别线性化点并不困难——保护已更新数据的锁释放时，更新的效果便会变得可见。这种转换的

* 由美国国家科学基金会通过编号为 #1811894 的资助项目提供支持。

然而，将这种直觉转化为证明取决于锁操作所使用的规约。CSL文献包含两种不同的锁获取与释放规约：

1. 在 Gotsman/Hobor 风格的规约中， [5,6], 每个锁都与一个锁不变量 R 相关联，该不变量由获取操作获得，并由释放操作恢复。2. 在 TaDA 风格的规约中， [14], 获取操作以原子方式将锁从未锁定状态移至锁定状态，而释放操作执行相反的操作。

众所周知，风格 2 的规约可以蕴含风格 1 的规约，并且在最近的大多数验证工作中，人们都选择了这种风格 [4,10]。然而，Zhang [15] 未发表的工作表明，风格 1 同样可以用于为数据结构证明原子规范。

在本文中，我们将这一方法应用于已验证软件工具链 [1], ，这是一个用于证明 C 程序正确性的系统，最近已扩展支持一般幽灵状态和原子性证明。VST [2] 的可靠性证明将 CompCert 编译器 [12] 的单线程正确性定理提升至并发环境，并且明确依赖于风格 1 的锁规约；修改这一证明以改为依赖原子操作，在理论和实践上都十分艰巨。我们证明了：1）从风格 1 的锁规约出发，我们的方法可以用于为采用细粒度锁定的有趣数据结构证明原子规范；2）这些证明需要一些额外的复杂处理，而使用风格 2 的锁规约则可以避免这些处理。其关键区别在于，在风格 1 中，线程必须先获得组件锁的句柄所有权，才能获取该锁并读取或修改组件；而在风格 2 中，锁和组件都可以视为单个抽象状态的一部分，每次访问锁时都对其进行访问。然而，为数据结构证明的顶层规约并不受锁规约复杂性的影响，因此，数据结构客户端的证明不会比其他情况下更加复杂。

据我们所知，这是对这两种锁规约风格的首次形式化比较；我们的技术也应适用于 VST 等工具，因为旧式风格已深度集成于这些工具中。这也是首次针对逻辑原子规约对 C 实现进行的基础性验证；此前，VeriFast [7]等工具虽已用于验证 C 实现，但并未证明其相对于 C 语义是可靠的，因此可以避开锁规约在并发可靠性方面带来的复杂性。

2 背景

2.1 逻辑原子规约

对于许多并发数据结构，理想的正确性条件是：即使多个线程同时访问，数据结构的行为也与顺序实现相同。这一直观条件已通过线性化（操作看起来按某种全序生效）或原子性（客户端线程永远不会观察到操作的中间状态）形式化。在分离逻辑中，原子性可以用如下形式的原子三元组 [14] 表示：

$$\langle a.\, P_l \mid P_p(a) \rangle\ c\ \langle Q_l \mid Q_p(a) \rangle$$

其中 P_l 和 Q_l 是类似于普通霍尔三元组的局部前置条件和后置条件，而 P_p 和 Q_p 是以抽象值为参数的公共前置条件和后置条件

共享数据结构的 a。该三元组表示：如果调用前 P_l 为真，并且共享状态（例如全局不变量）中保存着对于某个 a 值 P_p 为真的事实，那么在 c 的线性化点之前，P_p 将继续对某个（可能不同的）a 值为真；在线性化点处，Q_p 将针对同一个 a 值以原子方式变为真（并且在 c 结束后 Q_l 为真）。当其他线程修改数据结构时，只要始终保持 P_p(a)，a 的具体值（共享抽象状态）就可以变化。例如，该规范

$$\langle s.\ \mathsf{is_stack}\ p \mid \mathsf{stack}\ s\rangle\ \mathtt{push}(v)\ \langle \mathsf{is_stack}\ p \mid \mathsf{stack}\ (v :: s)\rangle$$

表达了这样一个事实：并发栈的 push 操作正确实现了顺序 push 的行为，在到达线性化点并将当前栈 s 原子地替换为 v :: s 之前，不会对现有的栈 s 做出任何可见更改。栈 本身被视为一种公开所有的资源，只有持有is_栈 断言的线程才能以原子方式访问和修改它。

2.2 两种锁规范风格

在并发分离逻辑引入后不久， [13], Gotsman 等人 [5]和 Hobor 等人 [6] 对其进行了扩展，使其支持 C 风格的可存储锁，其中锁是一种特殊的内存位置，用于协调对其他资源的访问。他们的锁规范实质上是等价的，形式如下：

$$\{\ell \boxdot\!\!\to R\}\ \mathtt{acquire}(\ell)\ \{R * \ell \boxdot\!\!\to R\} \qquad \{R * \ell \boxdot\!\!\to R\}\ \mathtt{release}(\ell)\ \{\ell \boxdot\!\!\to R\}$$

每个锁 l 都与一个“锁不变量”R 相关联。R 是表示受保护资源的任意分离逻辑谓词，由获取锁的线程获得，并且在线程释放锁时必须恢复。当时，CSL 尚未对原子操作进行形式化处理，因此实际的锁实现不在验证范围内：这些规约被视为公理，或者通过直接援引特定内存模型的数据竞争自由保证来证明。在不会释放锁的垃圾回收语言中，断言 $l \boxdot\!\!\to R$ 可以自由复制，并在访问 R 的线程之间共享；如果我们希望跟踪所有权并最终释放锁，则可以将该断言拆分为分数份额，使得 $l \boxdot\!\!\to_{\pi_1} R * l \boxdot\!\!\to_{\pi_2} R = l \boxdot\!\!\to_{\pi_1+\pi_2} R$，然后仅在拥有完整份额 $l \boxdot\!\!\to_1 R$ 时才允许释放。

后来，TaDA 逻辑 [14], 将逻辑原子性的概念编纂成形，并提出了另一组锁规范：

$$\langle b.\ (\mathsf{L}(\ell) \wedge \neg b) \vee (\mathsf{U}(\ell) \wedge b)\rangle\ \mathtt{acquire}(\ell)\ \langle \mathsf{L}(\ell) \wedge b\rangle \qquad \langle \mathsf{L}(\ell)\rangle\ \mathtt{release}(\ell)\ \langle \mathsf{U}(\ell)\rangle$$

获取操作可以反复访问锁，检查它处于锁定（L）或未锁定（U）状态；如果它看到未锁定状态（条件 b 成立），就可以原子地将其转换为锁定状态并返回。释放操作则只需原子地将锁从锁定状态转换为未锁定状态。这些操作并未规定锁保护的内容，但由于它们是原子的，因此可以与任何全局不变量（用 TaDA 的术语来说，即“共享区域”）交互：特别是，结合不变量（$\mathsf{U}(l) * R) \vee \mathsf{L}(l)$,

可以用来推导基于锁不变式的规范。还值得注意的是，这些锁规范并非公理：它们是针对一种简单的自旋锁实现证明得出的，该实现使用原子操作（例如比较并交换），而这些正是 TaDA 及其后续逻辑中的实际原语操作。在这种风格中，追踪锁的所有权与锁断言本身正交；可使用可取消不变式 [3] 之类的机制，使锁断言 U/L 可公开使用，并在线程全部结束后将其释放。

2.3 VST 和 Iris

我们采用两种基于 Coq 的验证系统相结合的方式：已验证软件工具链（VST）和 Iris。VST [1] 是一种基于 Coq 的系统，用于证明 C 程序的分离逻辑规约。它不仅针对 CompCert 编译器的 C 语言语义进行了可靠性证明，还针对编译器正确性定理 [2], 进行了证明，从而确保经过验证的程序在编译和运行时能够正确执行。VST 直接实现了 Hobor 等人 [6], 提出的并发分离逻辑，并且不变式风格的锁规范从根本上被纳入其可靠性证明之中。

Iris [9] 是一个框架，它综合了多种并发分离逻辑的思想，并秉持“幽灵状态是你所需要的一切”这一理念。Iris 从一个包含任意高阶幽灵状态的小型核心逻辑出发，并从幽灵状态的特定实例中推导出包括不变量和逻辑原子性在内的特性。Iris 被设计为独立于语言和逻辑的框架，而 VST 的近期版本利用了这一点，在基础模型中融入了 Iris 风格的幽灵状态，并直接使用 Iris 策略来推理 C 实现的不变量和原子性。本文描述的所有证明都已在这一 VST+Iris 环境中形式化。Iris 也被用作带有原子锁规范的证明的基础，例如 Krishna 等人 [10]的工作。

3 锁与原子性

本节概述我们将锁不变量与原子规约连接起来的方法，并将其与基于原子锁的方法进行对比。

考虑二叉搜索树的顺序实现。该实现包含插入、查找和删除函数，并已证明满足以下规约：

$$\{\mathsf{BST}\ p\ t\}\ \texttt{insert}(p,k,v)\ \{\mathsf{BST}\ p\ (t[k \mapsto v])\}$$

$$\{\mathsf{BST}\ p\ t\}\ \texttt{lookup}(p,k)\ \{v.\ \mathsf{BST}\ p\ t \wedge t(k) = v\}$$

$$\{\mathsf{BST}\ p\ t\}\ \texttt{delete}(p,k)\ \{\mathsf{BST}\ p\ (t[k \mapsto _])\}$$

其中，二叉搜索树 p t 表示从位置 p 开始的内存中存有一棵抽象二叉搜索树 t。例如，t 可以是 Coq 中递归定义的树类型的一个元素，而二叉搜索树 可以定义为一个递归谓词，用内存中的对象表示 t 的每个节点，这些对象包含键、值以及指向左子节点和右子节点的指针，根节点位于 p 处。

此数据结构的线程安全版本应满足与顺序规范[5]密切对应的原子规范：

$$\langle m.\ \mathsf{bst_ref}_g\ p\ |\ \mathsf{bst_abs}_g\ m\rangle\ \text{插入}(p, k, v)\ \langle \mathsf{bst_ref}_g\ p\ |\ \mathsf{bst_abs}_g\ (m[k\ \mapsto v])\rangle$$

$$\langle m.\ \mathsf{bst_ref}_g\ p\ |\ \mathsf{bst_abs}_g\ m\rangle\ \text{查找}(p, k)\ \langle v.\ \mathsf{bst_ref}_g\ p\ |\ \mathsf{bst_abs}_g\ m\ \wedge m(k)\ = v\rangle$$

$$\langle m.\ \mathsf{bst_ref}_g\ p\ |\ \mathsf{bst_abs}_g\ m\rangle\ \text{删除}(p, k)\ \langle \mathsf{bst_ref}_g\ p\ |\ \mathsf{bst_abs}_g\ (m[k\ \mapsto _])\rangle$$

这些规约断言，每个操作看起来都以原子方式执行，在不暴露任何中间状态的情况下访问并更新数据结构的状态——例如，查找不应找到一个部分插入的键，且其值不同于传递给插入调用的值。我们将 BST 断言拆分为两部分：bst_ref包含线程调用 BST 函数所需的指向和锁不变式断言，而 bst_abs 包含描述树整体状态的幽灵状态，并被视为以原子方式访问的共享资源。二者通过引用一个固定但任意的幽灵状态标识符 g 关联起来；通常我们会省略该标识符。我们将数据结构的抽象状态从树 t 改为键值映射 m，以反映客户端无法观察树的内部结构这一事实（当我们希望在并发访问期间重新排列节点时，这一点将很有用）。我们的顺序实现几乎肯定无法满足这些规约，因为其函数依赖于能够随时访问数据结构的任何部分：BST 断言包含对数据结构中每个位置的完整所有权。

要证明某个实现满足这些原子规范，我们必须证明：1）该实现能够在不持有公共断言 bst_abs 的任何部分的情况下安全执行，只以原子方式访问并恢复该断言，直至线性化点；2）在某个线性化点，该实现会以原子方式将当前映射 m 转换为满足公共后置条件的映射。线性化点可以是一个（物理或逻辑上的）原子操作，也可以出现在程序执行步骤之间（例如，如果当前线程持有数据结构中被修改部分的锁，那么在释放锁之前，其他线程无法区分原始状态和修改后的状态）。在接下来的章节中，我们将讨论使用两种锁规范的基于锁的实现的这类证明的一般结构。

3.1 使用锁不变式证明原子性

同步数据结构最简单的方法，是添加一把锁，让每个操作在开始时获取该锁，并在结束时释放。这类粗粒度锁定实现很容易使用不变式风格的锁规范来验证：锁的不变量可以包含整个数据结构断言（例如 BSTp t），这样线程在执行操作时始终拥有该数据结构的完整所有权。如果我们只想证明该操作是线程安全的，可能会选择锁不变量 $\exists t.\ \mathsf{BST}\ p\ t$，断言每次获取锁时，我们都会在某个状态 t 下获得树的完整所有权。

[5] 我们还有用于创建和释放树的函数，但这些函数不必是线程安全的，因为树在创建时不会被共享，在释放时也不得被共享。

另一方面，如果我们想证明描述数据结构变化的规约——例如原子规范 ⟨m. bst_abs m⟩ insert(p, k, v) ⟨bst_abs (m[k ↦v])⟩——我们就需要幽灵状态来追踪这些变化。

具体来说，我们可以定义一部分幽灵状态 幽灵资源_二叉搜索树 t，它在锁不变量与函数的公共前置条件和后置条件之间进行划分，将内存中的树与数据结构的公共抽象状态连接起来。例如，我们可以选择一个锁不变量 R^{cg}，∃t。 二叉搜索树 p t ∗ 幽灵资源_二叉搜索树$_{.5}$ t，然后设置 二叉搜索树_引用 p ≜p ⊡↦$_{\pi}$ R^{cg} and 二叉搜索树_抽象 t ≜ 幽灵资源_二叉搜索树$_{.5}$ t。这样，我们就可以证明上述原子三元组：在线性化点，我们访问 二叉搜索树_抽象，合并幽灵状态的两部分，并更新它们以反映具体数据结构的新状态，然后使用幽灵状态的一部分来满足原子后置条件，并将另一部分返回给锁不变量。原子后置条件反映了抽象状态已被访问或更改这一事实，而锁不变量则维持抽象状态与内存中实际数据结构之间的连接。这种通过幽灵状态使对加锁数据结构的更改可见的方法，源于 Zhang 和 Jung 在 Iris [15] 中对原子同步器的证明。

在细粒度实现中，数据结构的不同部分上存在多个锁，因此每个锁不变量的幽灵状态不可能是整个数据结构的抽象状态。相反，幽灵状态表示单独的加锁部分的抽象状态，而数据结构的抽象状态则由各个加锁组件状态的组合推导得出。每个加锁组件 c 都有一部分幽灵状态 幽灵资源_c，该状态在锁不变量与顶层抽象状态之间进行拆分。在 c 的临界区结束时，我们以原子方式访问顶层抽象状态，合并 幽灵资源_c 的两半，更新幽灵资源_c 以反映任何局部更改，然后证明以下两者之一：对 c 的更新不会改变数据结构的顶层状态；或者对 c 的更新会将顶层状态改变为满足该函数原子后置条件的状态。例如，从二叉搜索树中删除一个节点时，我们会旋转节点以重构出一棵有效的树；在每次旋转的临界区结束时，我们证明以下两者之一：树中的值未发生改变（并且树仍然保持良好的顺序）；或者我们已经删除了待删除的节点。正如我们将在第 4 节中看到的，每个组件的锁不变量还必须仔细说明该组件的锁以及相关节点的锁（例如树中的子节点锁）的所有权。

3.2 使用原子锁实现原子性

使用锁的原子规范证明数据结构的原子规范更为直接。例如，考虑 Krishna 等人对搜索结构中手递手锁定模式的验证 [10]。所使用的锁规范[6] 是

$$\langle \mathsf{inFP}(n) \mid \mathsf{CSS}(r, C)\rangle \ \mathtt{lockNode}\ n\ \langle \mathsf{CSS}(r, C) * \mathsf{N}(n)\rangle$$

其中，inFP(n) 断言节点 n 位于数据结构中，CSS 是抽象状态，而N(n) 是节点 n 的资源（包括指向断言和幽灵

[6] 事实上，该规范源自基本的 TaDA 风格原子锁规范。

状态）。抽象状态断言 CSS 包括全局幽灵状态，以及当前未被锁定的每个节点的每节点状态 N 的组合，因此获取节点 n 的锁后，可以从抽象状态中移除N(n)。

请注意，在这种情况下，局部前置条件inFP(n) 仅表示知道 n 位于数据结构中，并不涉及任何所有权——尤其是，调用者不需要拥有根节点锁的份额。线程可以通过原子访问CSS，提取与数据结构中节点 n 对应的子断言，并将其切换到锁定状态，从而获取该节点的锁。更一般地说，在这种方法中，无需跟踪各个节点锁的所有权；正如我们将在第 4.1 节中看到的那样，在基于锁不变式的证明中，这正是造成相当复杂性的一个因素。

4 使用交替持锁的已验证二叉搜索树

我们通过在 VST 中验证一棵采用细粒度锁定的二叉搜索树（BST），来展示如何使用基于不变式（类型 1）锁规约证明逻辑原子规约；我们采用第 3.1 节的方法为二叉搜索树操作证明原子规约，同时对锁使用不变式风格规范（这是 VST 可靠性证明所要求的）。如第 3 节所述，我们的目标是定义断言bst_ref （二叉搜索树的每线程句柄）和bst_abs （共享抽象状态），使得并发二叉搜索树的操作满足类似如下的原子三元组

$$\langle m.\ \mathsf{bst_ref}\ p \mid \mathsf{bst_abs}\ m\rangle\ \texttt{insert}(p,k,v)\ \langle \mathsf{bst_ref}\ p \mid \mathsf{bst_abs}\ (m[k \mapsto v])\rangle$$

本节中，我们将介绍如何构造这两个谓词，并证明采用交替持锁的二叉搜索树实现中的各项操作；其中，锁使用基于锁不变量的规格。

4.1 交替持锁

交替持锁（也称为锁耦合）是一种用于遍历数据结构的细粒度锁定模式：在释放当前节点的锁之前，先获取下一个节点的锁，从而确保遍历期间这两个节点之间的链接不会被移除或重新排列。交替持锁是需要引用其他锁不变量的锁不变量这一模式的经典示例，其历史可追溯至 CSL 中关于“堆中的谓词”的最早论文 [5]。诀窍是为数据结构中的每个节点分配一个锁不变量，其中包含其子节点的锁断言；例如，对于链表，我们可以写作

$$R(n) \triangleq n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\mapsto R(n') \tag{1}$$

其中，n′ 是 n 的子节点，l′ 是 n′ 的锁。获取 n 的锁后，我们便能获知 n′ 的锁断言，并可按需获取其锁，然后再释放 n 的锁。

当我们考虑锁的所有权时，这个不变量会变得稍微复杂一些；如果我们最终希望释放锁并归还其资源，这种考虑是必需的。在这种情况下，锁断言携带一个份额 π （通常表示为有理数，位于

范围 $(0,1]$ 内）；释放锁需要完整份额 $\pi = 1$，而获取或释放锁则只需任意份额即可。我们可以将锁不变量修改为

$$R(n) \triangleq n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\rightarrow_{\pi} R(n') \tag{2}$$

对于某个固定份额 π，但随后，释放 n 上的锁时，l′ 的份额会怎样？从持有 n 的锁 l 的状态开始，我们得到：

$\{\ell \boxdot\!\!\rightarrow_{\pi} R(n) * n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\rightarrow_{\pi} R(n')\}$
`acquire`(ℓ');
$\{\ell \boxdot\!\!\rightarrow_{\pi} R(n) * n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\rightarrow_{\pi} R(n') * n' \mapsto (d, n'', \ell'') * \ell'' \boxdot\!\!\rightarrow_{\pi} R(n'')\}$
`release`(ℓ);
$\{\ell \boxdot\!\!\rightarrow_{\pi} R(n) * n' \mapsto (d, n'', \ell'') * \ell'' \boxdot\!\!\rightarrow_{\pi} R(n'')\}$

在调用 release 时，我们放弃资源，以重新确立 l 的锁不变量，其中包括 l′ 的锁断言。因此，我们剩下 l 的一份份额（即我们刚离开的节点的锁），而不再持有 l′ 的任何份额（即当前节点的锁）。

这两个问题都可以通过将锁设为递归锁来解决，即在 l 自身的锁不变量中包含 l 的一份份额[7]。我们将节点 n 的锁 l 的所有权拆分为两份：π_1 由 l 自身持有，π_2 由 n 的父节点的锁持有。然后，我们修订定义 1，使其成为

$$R(n) \triangleq \ell \boxdot\!\!\rightarrow_{\pi_1} R(n) * n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\rightarrow_{\pi_2} R(n')$$

现在，同样的操作序列给出 u s:

$\{\ell \boxdot\!\!\rightarrow_{\pi_1} R(n) * n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\rightarrow_{\pi_2} R(n')\}$
`acquire`(ℓ');
$\{\ell \boxdot\!\!\rightarrow_{\pi_1} R(n) * n \mapsto (d, n', \ell') * \ell' \boxdot\!\!\rightarrow_{\pi_2} R(n') *$
$\ell' \boxdot\!\!\rightarrow_{\pi_1} R(n') * n' \mapsto (d, n'', \ell'') * \ell'' \boxdot\!\!\rightarrow_{\pi_2} R(n'')\}$
`release`(ℓ);
$\{\ell' \boxdot\!\!\rightarrow_{\pi_1} R(n') * n' \mapsto (d, n'', \ell'') * \ell'' \boxdot\!\!\rightarrow_{\pi_2} R(n'')\}$

获取 l′ 时，我们持有其锁断言的 π_1 和 π_2 两份；释放 l 时，我们返还 l 的份额 π_1 以及 l′ 的份额 π_2，但保留 l′ 的份额 π_1，以便稍后释放它。定义一个包含自身锁断言份额的锁不变量，其机制可能因分离逻辑而异（VST 提供了 selflock 断言，正是用于此目的）；但只要存在这样的机制，就可以使用这种模式实现带份额核算的交接式锁不变量，从而遍历按此模式构建的数据结构，并在线程全部处理完毕后回收其所有资源。

这种模式是我们的证明中复杂性的主要来源，而使用原子锁规范的证明中不存在这种复杂性。例如，在 Krishna 等人 [10], 的交替持锁证明中，节点关于其子节点的唯一信息是它们位于数据结构中这一事实；它们的锁断言与其余资源一起保存在全局抽象状态中。这显著简化了交替持锁同步的规约和证明。

[7] Gotsman 等人则使用分离断言 Locked，该断言始终由获取锁的线程持有，并且释放锁时必须满足：这在功能上等同于额外的一份锁断言。

4.2 手递手二叉搜索树的规范

我们二叉搜索树中节点的 C 实现如下：

typedef struct tree {int 键; void *值; struct tree_t *左节点, *右节点;} tree; typedef struct tree_t {tree*t; lock_t *锁;} tree_t;

每个节点（类型为 tree_t）都有一个锁字段，用于保存保护该节点的锁；其 t 字段要么为 NULL（对于叶节点而言，该节点不包含任何键或值），要么指向一个 tree 结构体，其中包含该节点的键、值和子节点指针。为了验证针对这一数据结构的操作，我们需要将内存中的 tree_t 与抽象状态（从键到值的数学映射）关联起来，并在断言 bst_ref （每个客户端线程持有的资源）和 bst_abs （数据结构的抽象状态）之间划分这些资源的所有权。如第 3.1 节所述，我们的做法是将一份幽灵状态与树中的每个节点关联起来，这份状态将在该节点的锁与 bst_abs 中的抽象状态之间共享，同时，bst_ref 将是一个指向根节点锁的指针。我们还需要确保锁不变量支持手递手模式。

键范围 当我们遍历二叉搜索树以查找键 k 时，最终会到达包含 k 的节点，或到达一个空节点——如果 k 位于树中，它本应出现在该节点处。为了证明这一过程的正确性，我们需要使用幽灵状态来跟踪每个键在当前树中“应出现”的位置。正如 Krishna 等人所观察到的那样， [11], 可以通过为每个节点关联一个下界和上界来实现，这两个界限定了以该节点为根的子树中出现的键。我们将这两个界所构成的组合称为一个范围。节点的范围继承自其父节点，并基于父节点的键确定：如果节点 n 的范围为 (l,r)，键为 k，则其左子节点的范围为 (l,k)，右子节点的范围为 (k,r)。图1展示了一个二叉搜索树示例，其中每个节点都标有其范围；范围从根节点处的 (−∞, +∞) 开始，并传播到空叶节点。

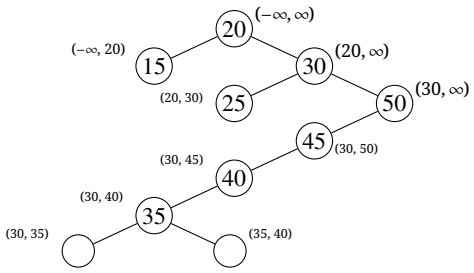


图1：每个节点都标注其范围的二叉搜索树；省略了除35以外节点的空子节点

在叶节点处，这些范围将树中不存在的所有键所构成的空间划分开来。如果我们到达一个范围为 (a,b) 的叶节点，就知道介于 a 和 b 之间的键 1）当前不在树中，

并且 2）可以正确地插入到该叶节点中。例如，假设我们想将键 38 插入图1中的树。键为 35 的节点的右子节点是一个范围为 (35,40) 的叶节点，因此可以保证这里是插入 38 的正确位置。

每节点与全局幽灵状态 重要的是，即使其他线程中的操作在我们遍历期间重构了树，叶节点的范围仍然会划分所有可能键的空间。此外，树中更高位置的操作只会扩大其下方节点的范围，因此，如果我们找到一个范围包含目标键的节点，就能保证找到了正确的位置。这使范围非常适合作为幽灵状态：一个线程持有的范围信息不会被其他线程作废，并且足以证明二叉搜索树操作的正确性。更准确地说，每个节点的幽灵状态由幽灵名称 g 标识，其中包含一对范围和内容；对于叶节点，内容为 无 ；对于包含键 k、值 v 以及标识符为 g_l 和 g_r 的左子节点和右子节点的内部节点，内容为 有 (k, v, g_l, g_r)。谓词 节点_幽灵资源$_g$ ((i, j), n) 断言标识符为 g 的节点的范围为 (i, j)，内容为 n；该谓词将在节点的锁不变量与抽象状态断言 二叉搜索树 抽象之间划分。_

二叉搜索树_抽象 谓词描述了抽象映射 m 由树的幽灵状态实现时所需满足的条件。直观来说，我们应当能够将各节点的幽灵状态汇集成一棵实现映射 m 的抽象树 t，也就是说，m 中的每个键值对都出现在 t 的某个节点中。我们通过先定义形成抽象树所需的各节点幽灵状态片段集合，将这一点形式化：

$$\begin{aligned}&\text{public_halves}(t, g, (i, j)) \triangleq \text{match } t \text{ with} \mid \text{Leaf} \Rightarrow \text{节点_幽灵资源}_g^5\\&((i, j), \text{None}) \mid \text{Node } (k, v, t_l, g_l, t_r, g_r) \Rightarrow \text{节点_幽灵资源}_g^5((i, j), \text{Some}\\&(k, v, g_l, g_r)) * \text{public_halves}(t_l, g_l, (i, k)) * \text{public_halves}(t_r, g_r, (k, j))\end{aligned}$$

我们从范围为 (i, j) 的节点 g 开始，递归遍历这棵树，并将每个节点_幽灵资源 断言收集到一个单独的断言中。我们按照上述方法计算各节点子节点的范围：如果当前节点的键为 k，那么左子节点的范围是 (i, k)，右子节点的范围是 (k, j)。

这些按节点划分的幽灵状态集合描述了树的内容和结构，但获取某个节点锁的线程还需要知道该节点确实存在于树 t 中。为此，我们定义幽灵状态断言 幽灵资源_节点(S) 和 在_树 g，其中 S 是标识符集合，g 是一个标识符，并且 在_树 g $*$幽灵资源_节点(S) 意味着 g $\in$S。于是，二叉搜索树_抽象 的完整定义为：

$$\begin{aligned}\text{bst_abs}_g\ m \triangleq\ &\exists t.\ t \text{ implements } m \wedge \text{public_halves}(t, g, (-\infty, +\infty)) *\\&\text{ghost_nodes}(\text{ids}(t))\end{aligned}$$

其中，ids(t) 是抽象树 t 中所有节点标识符的集合；当且仅当 t 中的键值对与 m 中的完全相同时，t 才实现 m。根节点（名为 g）的范围是（$-\infty, +\infty$），而 幽灵资源_节点 断言保证：如果任何线程或锁持有关于树 g_i 的断言 在_树 g_i 中，那么 g_i 就是 t 中的一个节点。由于对 t 进行了存在量化，只要 t 仍然是良构树，我们就可以随时重新排列其中的节点

同时保持节点标识符集合和键值对不变。验证删除函数时，我们将利用这一点。

接下来，我们通过将每个节点的锁与一个锁不变式断言 l ⊡→ R 关联起来，把 C 实现中的各节点锁与全局抽象状态联系起来，其中 R 同时包含该节点的具体指向断言以及相应的抽象节点_幽灵资源。位置 p 处节点的锁不变量需要将抽象内容 c 映射为 p 的 t 字段在内存中的值：

$$\begin{array}{l}\text{match } c \text{ with}\\ \mid \text{None} \Rightarrow p.\mathtt{t} = \mathtt{NULL}\\ \mid \text{Some } (k, v, g_l, g_r) \Rightarrow \exists p_l, p_r.\ p.\mathtt{t} \mapsto (k, v, p_l, p_r)\end{array}$$

然后，我们将其与第 4.1 节的交替锁定不变式模式结合起来，得到最终的锁不变量：

$$\begin{array}{l}\text{node_inv}_g(p) \triangleq \exists i\ j\ c.\ \text{node_ghost}_g^5\ ((i, j), c) * p.\mathtt{lock} \boxdot\!\!\rightarrow_{\pi_1} \text{node_inv}_g *\\ \quad \text{match } c \text{ with}\\ \quad \mid \text{None} \Rightarrow p.\mathtt{t} = \mathtt{NULL}\\ \quad \mid \text{Some } (k, v, g_l, g_r) \Rightarrow k \in (i, j) \wedge \exists p_l, p_r.\ p.\mathtt{t} \mapsto (k, v, p_l, p_r) *\\ \qquad p_l.\mathtt{lock} \boxdot\!\!\rightarrow_{\pi_2} \text{node_inv}_{g_l} * p_r.\mathtt{lock} \boxdot\!\!\rightarrow_{\pi_2} \text{node_inv}_{g_r}\end{array}$$

交替锁定模式允许我们获取 p_l.lock 或 p_r.lock，然后释放 p.lock，而不会丢失锁断言的任何份额。

现在，我们可以补全第 3 节中概述的规约：

$$\langle m.\ \text{二叉搜索树_ref}_g\ p \mid \text{二叉搜索树_abs}_g\ m\rangle\ \text{插入}(p, k, v)\ \langle \text{二叉搜索树_ref}_g\ p \mid \text{二叉搜索树_abs}_g\ (m[k \mapsto v])\rangle$$

$$\langle m.\ \text{bst_ref}_g\ p \mid \text{bst_abs}_g\ m\rangle\ \text{查找}(p, k)\ \langle v.\ \text{bst_ref}_g\ p \mid \text{bst_abs}_g\ m \wedge m(k) = v\rangle$$

$$\langle m.\ \text{bst_ref}_g\ p \mid \text{bst_abs}_g\ m\rangle\ \mathtt{delete}(p, k)\ \langle \text{bst_ref}_g\ p \mid \text{bst_abs}_g\ (m[k \mapsto _])\rangle$$

其中 $\text{bst_abs}_g m$ 断言，树的全局幽灵状态实现了映射 m，如上文所述；而 bst_ref （客户端持有的数据结构句柄）只是对根节点锁的引用：

$$\text{bst_ref}_g\ b \triangleq \exists p.\ b \mapsto p * p.\mathtt{lock} \boxdot\!\!\rightarrow \text{node_inv}_g(p)$$

接下来，我们将证明每个二叉搜索树函数都满足其规范。

4.3 证明：插入和查找

插入和查找函数的代码如图 2 所示。每个函数中的遍历过程相同：我们访问当前节点的键，将其与目标键 k 进行比较；如果 k 小于当前键，则移动到左子节点；如果 k 大于当前键，则移动到右子节点，并使用交接式锁。因此，两个循环具有相同的不变量。在每次迭代开始时，我们位于指针 p 所指向的节点处，其幽灵标识符为 g，并持有该节点的锁，从而获得锁不变量节点_不变量$_g$(p) 以及某个范围 (i, j)。循环不变量要求 k ∈ (i, j)，也就是说，目标键始终属于

```
1 void 插入(tree_t** t, int k, void *v){ void *查找(tree_t** t, int k){ 2 tree_t *tgt =*t;
tree_t *tgt = *t; 3 acquire(tgt->锁); acquire(tgt->锁); 4 while(1) { while(1) { 5 tree* p = t
gt->树; tree *p = tgt->树; 6 if (p == NULL) { if (p == NULL) { 7 ... // 使用 k、v 创建一个新的
节点 8 release(tgt->锁); release(tgt->锁); 9 return; return NULL; 10 } } 11 if (k < p->键){ if (
k < p->键){ 12 void *l_old =tgt->锁; void *l_old = tgt->锁; 13 tgt =p->left; tgt =p->left; 14
acquire(tgt->锁); acquire(tgt->锁); 15 release(l_old); release(l_old); 16 } else if (p->键 <k){ }
else if (p->键 < k){ 17 ... // 改为移动到 p->right ... // 改为移动到 p->right 18 } else { } else { 19
p->值 = v; void* v =p->值; 20 release(tgt->锁); release(tgt->锁); 21 return; return v; 22 } }
23 } } 24 } }
```

图 2：插入和查找的代码框架

当前节点的子树中。这保证了如果我们到达一个空的叶节点，该叶节点就是应该存放键 k 的位置。每个操作的实际逻辑发生在我们到达空的叶节点（第 7 行）或找到一个以 k 为其键的节点（第 19 行）时；此外，这两个函数还有各自的线性化点。

当插入操作遇到一个键为 k 的节点时，只需将该节点的值更改为 value。如果到达叶节点，则分配一个键为 k、值为 v 的新节点，并将其插入该位置。在这两种情况下，我们都必须证明，可以将树的抽象状态从bst_abs m 原子地更新为 bst_abs(m[k $\mapsto$ v])，方法是证明对内存中树的更新对应于抽象树中的局部更改，该更改插入了新的键值对。证明分为两部分：一部分证明 C 代码实现了局部更新，另一部分证明对幽灵资源状态树的局部更新实现了全局更新。在第一种情况下，局部更改是在一个范围包含 k 的空叶节点处插入 (k,v)，并添加两个新的空叶节点作为其子节点；在第二种情况下，我们只需更改一个键已经是 k 的内部节点的值。

查找的证明类似，但更简单。我们必须证明，如果在树中找到了键，那么抽象状态中就存在相同的键值对；如果没有找到该键，那么它就不在抽象状态中。前者由锁不变量节点_不变量保证，该不变量确保内存中的键和值与幽灵节点中的键和值相匹配；后者则由范围保证，因为如果抽象树

```
1 void pushdown_left (tree_t *tgt){ 2 while(1) { 3 tree *p  =
tgt->t; 4 tree_t *rc   =p->right; 5 void *lq   =rc->lock; 6 acquire(rc->
lock); 7 tree *q  =  rc->t; 8 if (q   ==  NULL) { 9 tree_t *lc  =  p->left;
10 ... // 将 lc->t 移至 tgt；释放 p、lc 和 rc 11 release(tgt->lock); 12
return; 13 } else { 14 turn_left(tgt, rc); 15 tgt   =q->left; 16
release(tgt->lock); 17 } 18 } 19 }
```

图 3：pushdown_left 的代码概要

包含一个范围为 (i, j) 的叶节点，使得 k ∈ (i, j)，则它不可能包含键为 k 的节点。无论哪种情况，抽象状态都保持不变。

4.4 删除

删除函数使用与插入和查找相同的遍历逻辑，但找到具有目标键的节点时，会调用图 3 中概述的辅助函数 pushdown_left。该函数会重新排列树中的节点，将待删除节点沿树向下移动，直到其右子节点为空，同时维持二叉搜索性质。具体做法是反复旋转三个节点组成的结构：目标节点的右子节点 rc 向上移动并成为其父节点，目标节点位于其左侧，而该右子节点的右子节点则移至目标节点之下，如图 4 所示。待删除节点没有右子节点后，用其左子节点的内容替换该节点的内容，并将其子节点从树中移除。

pushdown_left函数通过两种方式改变树的结构。首先，旋转turn_left（第14行）改变待删除节点及其右子节点的范围（例如，图4第一步中的节点40和55）；这是一个局部变化，因为所有其他节点的范围（甚至重新排列节点的子节点的范围）都保持不变。这不是线性化点，因此必须认为该旋转不会改变树的抽象状态。我们通过在bst_abs的定义中对抽象树t进行量化来处理这一点；在不改变键值映射的情况下重新排列节点，仍然实现相同的抽象状态，并且其他线程不能假定它们已经观察到的树结构仍是当前的树结构。其次，最终删除节点（第10行）会增大其下方子树的范围；例如，

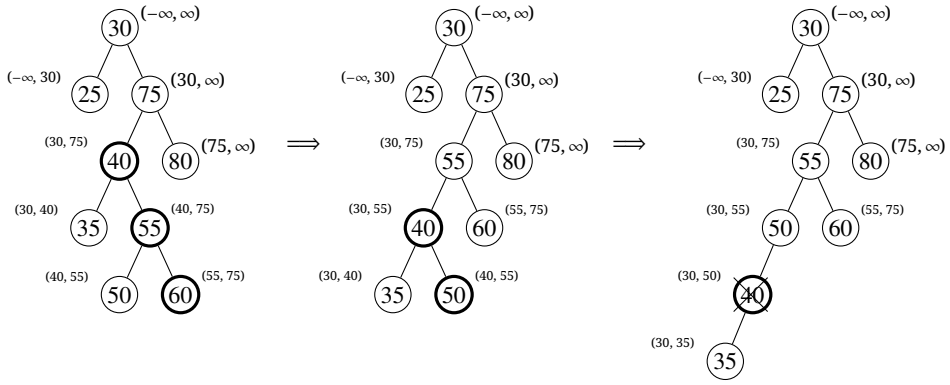


图4：通过 pushdown_left 从树中删除节点 40

在图4的最后一步中，删除节点35时，该节点的范围从（30,40）变为（30,50）。这是函数的线性化点。该变化发生时，pushdown_left仅持有被删除节点的锁，因此范围幽灵状态的代数必须允许节点的范围随时增大，即使该节点的锁正被持有。幸运的是，插入和查找仅使用范围来断言目标键位于当前节点的范围内；即使范围增大，这一点仍然成立。

5 使用规约

为了说明基于锁的证明的复杂性对潜在用户是隐藏的，我们验证了二叉搜索树的一个简单客户端程序，如图 5 所示。运行 update_tree 函数的生产者线程填充一棵树，而运行 retrieve_value 的消费者线程则等待，直到键 2 与指向值 4 的指针相关联，然后返回与键 1 关联的值。这是消息传递惯用模式的数据结构级版本，展示了原子规范的优势之一：我们可以利用它们来同步线程，并通过数据结构操作传递资源所有权。二叉搜索树 t 的不变量是一个简单的状态机，根据抽象映射 m 的内容分为三种状态：1) m(2) = NULL；2) 对于某个整数 i，m(2) $\mapsto$ i，且 i $\neq$ 4；以及 3) m(2) $\mapsto$ 4 $\wedge$ m(1) $\mapsto$ 3。在最终状态中，不变量还持有要从生产者转移给消费者的资源。我们还加入了幽灵状态，以反映恰好存在一个生产者线程这一事实（因此生产者始终确切知道树中有哪些值），以及一个消费者线程（因此它可以检索资源，而不必担心干扰）。客户端的证明只使用二叉搜索树操作的原子规范，不受锁规范风格的影响。

6 相关工作

Gotsman 等人 [5], 在介绍不变式风格的锁规范的同一篇论文中，还验证了一个采用手递手加锁的链表，该方法后来成为一种常见的

```
1 void* update_tree(void* t){ int retrieve_value(treebox t){ 2 a = 1; int y =-
1; 3 b = 2; do { 4 c = 3; searchResult = 查找 (t, 2); 5 d = 4; if (searchResult !=
NULL) { 6 插入(t, 1, &a); y = *((int *)searchResult); 7 插入(t, 2, &b); } 8 插入(t, 1, &
c); } while (y != 4); 9 插入(t, 2, &d); void* searchResult = 查找 (t, 1); 10 return
NULL; return *((int *)searchResult); 11 } }
```

图5：客户端代码

这是处理细粒度锁定的 CSL 示例。其中最新的工作之一是 Krishna 等人 [10], 使用原子锁规范证明了同一数据结构的正确性。这项工作以现代规约更新了 Gotsman 等人的工作，并直接将结果与 Krishna 等人的结果进行比较。

VeriFast [8], 是一种基于分离逻辑的 C语言 和 Java 验证器，支持基于锁的并发和原子并发 [7], ，并已用于验证一种类似于 Krishna 等人的手递手加锁链表。锁操作和链表本身的规约使用了 TaDA 逻辑原子性的前身。VeriFast 并非基础性的，但其基本逻辑已在 Coq 中针对一种玩具语言的语义进行了验证。

7 结论

虽然大多数针对基于锁的数据结构的新版 CSL 证明都为其锁采用 TaDA 风格的原子规范，但转而不再使用基于不变量的规范所带来的影响，此前并未得到深入考察。我们现已证明，基于不变量的规范仍可用于证明较大数据结构的逻辑原子规范（即使是细粒度实现），但与采用原子锁规范相比，这需要更复杂一些的证明（尤其是对于细粒度实现）。旧式锁规范仍可用于 VST 之类的系统，因为切换规范会使现有的自动化机制或可靠性证明失效；在这些系统中，我们提出的方法可用于为细粒度锁数据结构获得强正确性属性。不过，在可能的情况下，我们更倾向于对锁使用原子规范，以简化证明。事实上，我们打算研究如何调整 VST 的基础，使其使用原子操作并推导出锁的原子规范：尽管更新可靠性证明将是一项艰巨的工作，但采用较新的锁规范风格将简化今后所有关于基于锁的数据结构的证明。

致谢 本研究受益于与 Ralf Jung 的大量通信。他指出了 Zhang 的 syncer 方法，并解释了 Iris 中逻辑原子性的细微之处。

参考文献

1. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.:《经认证编译器的程序逻辑》。剑桥大学出版社，美国（2014年） 2. Cuellar, S., Giannarakis, N., Madiot, J.M., Mansky, W., Beringer, L., Cao, Q., Appel, A.: 并发场景下的编译器正确性：从并发分离逻辑到共享内存汇编语言。技术报告，普林斯顿大学（2020年） 3. Dang, H.H., Jourdan, J.H., Kaiser, J.O., Dreyer, D.: RustBelt 遇上宽松内存。ACM《编程语言会议录》4(POPL)（2019年12月）。https://doi.org/10.1145/3371102, https://doi.org/10 3371102 4 1145. / . D'Osualdo, E., Farzan, A., Gardner, P., Sutherland, J.: TaDA Live：用于细粒度并发程序终止性的组合式推理。《ACM编程语言与系统汇刊》（TOPLAS）。（2021年） 5. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: 可存储锁和线程的局部推理。载于：Shao, Z.（编）《编程语言与系统》。第19–37页。施普林格海德堡，柏林，海德堡（2007年） 6. Hobor, A., Appel, A.W., Nardelli, F.Z.: 并发分离逻辑的预言机语义。载于：Drossopoulou, S.（编）《编程语言与系统》。第353–367页。施普林格海德堡，柏林，海德堡（2008年） 7. Jacobs, B., Piessens, F.: 富有表现力的模块化细粒度并发规范。载于：第38届 ACM SIGPLAN-SIGACT 编程语言原理年会论文集。第271–282页。POPL '11，计算机协会，纽约，美国（2011年）。https://doi.org/10.1145/1926385.1926417, https://doi.org/10 1926417 8 1145 1926385 . / . Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast：一种功能强大、可靠、可预测且快速的 C语言和 Java 验证器。载于：Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R.（编）《NASA形式化方法》。第41–55页。施普林格海德堡，柏林，海德堡（2011年） 9. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris：作为并发推理正交基础的幺半群与不变式。载于：第42届 ACM SIGPLAN-SIGACT 编程语言原理年会论文集，POPL 2015，印度孟买，2015年1月15–17日。第637–650页。ACM（2015年）。https://doi.org/10 2676980 10 1145 2676726 2676980, 10 1145 2676726 . / . https://doi.org/ . / . . Krishna, S., Patel, N., Shasha, D., Wies, T.: 验证并发搜索结构模板。载于：第41届 ACM SIGPLAN 编程语言设计与实现会议论文集。第181–196页。PLDI 2020，计算机协会，纽约，美国（2020年）。https://doi.org/10.1145/3385412.3386029, https://doi.org/10 3386029 11 1145 3385412 . / . . Krishna, S., Shasha, D., Wies, T.: 顺势而为：并发数据结构的组合式抽象。ACM 《编程语言会议录》2(POPL)（2017年12月）。https://doi.org/10 3158125 12 1145 3158125, 10 1145 . / https://doi.org/ . / . Leroy, X.: 实用编译器的形式化验证。《ACM通讯》52(7)，107–115（2009年7月 1538814 13 10 1145 1538788 1538814, 10 1145 1538788）。https://doi.org/ . / . https://doi.org/ . / . . O'Hearn, P.W.: 资源、并发与局部推理。《理论计算机科学》375(1)，271–307（2007年）。https://doi.org/https://doi.org/10.1016/j.tcs.2006.12.035, https://www.sciencedirect.com/science/article/pii/S030439750600925X, 献给 John C. Reynolds 70岁生日的纪念文集 14. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA：一种用于时间与数据抽象的逻辑。载于：Jones, R.（编）《ECOOP 2014——面向对象编程》。第207–231页。施普林格海德堡，柏林，海德堡（2014年）

15. Zhang, Z.：iris-atomic（2016年12月），
https://gitlab.mpi-sws.org/FP/iris-atomic/-/raw/master/docs/atomic.pdf