Proving Logical Atomicity using Lock Invariants*

Roshan Sharma[1], Shengyi Wang[2][0000–0002–2286–8703], Alexander Oey[3], Anastasiia Evdokimova[4], Lennart Beringer[2][0000–0002–1570–3492], and William Mansky[4][0000–0002–5351–895X]

[1] Amazon, USA
[2] Princeton University, USA
[3] Cisco, USA
[4] University of Illinois Chicago, USA

Abstract. Logical atomicity has been widely accepted as a specification format for data structures in concurrent separation logic. While both lock-free and lock-based data structures have been verified against logically atomic specifications, most of the latter start with atomic specifications for the locks as well. In this paper, we compare this approach with one based on older lock-invariant-based specifications for locks. We show that we can still prove logically atomic specifications for data structures with fine-grained locking using these older specs, but the proofs are significantly more complicated than those that use atomic lock specifications. Our proof technique is implemented in the Verified Software Toolchain, which relies on older lock specifications for its soundness proof, and applied to C implementations of lock-based concurrent data structures.

Keywords: concurrent separation logic, fine-grained locking, logical atomicity, Verified Software Toolchain, Iris

1 Introduction

Concurrent separation logic (CSL) [13] is a useful tool for proving correctness of concurrent programs. While early iterations focused on data structures implemented with locks and specifications involving memory safety and data-race-freedom [5,6], more recent logics are able to prove linearizability-style functional correctness of lock-free implementations that use low-level atomic memory operations [14,9], even under weak memory models [3]. In particular, *logical atomicity* [14] has become the gold standard for concurrent data structure specifications, capturing the idea that data structure operations should appear to take place instantaneously with no externally visible intermediate states. The effects of a logically atomic operation become visible at a *linearization point*, a single instruction (usually an atomic memory operation or call to a logically atomic function) that publishes the new state of the data structure.

Intuitively, a critical section in a lock-based implementation serves the same role as an atomic instruction in a lock-free data structure, and it is not hard to identify linearization points in lock-based implementations—the effects of an update become visible when the lock protecting the updated data is released. The translation of this

* Supported by the National Science Foundation via grant #1811894.

intuition into proof, however, depends on the specifications used for lock operations. The CSL literature contains two distinct specifications for lock acquire and release:

1. In Gotsman/Hobor-style specs [5,6], each lock is associated with a lock invariant *R*, which is gained by `acquire` operations and restored by `release` operations.
2. In TaDA-style specs [14], `acquire` atomically moves a lock from the unlocked state to the locked state, and `release` does the reverse.

Style 2 specs are known to imply style 1 specs, and have been the style of choice for most recent verifications [4,10]. However, unpublished work by Zhang [15] shows that style 1 can also be used to prove atomic specifications for data structures.

In this paper, we adapt this approach to the Verified Software Toolchain [1], a system for proving correctness of C programs that has recently been extended with support for general ghost state and atomicity proofs. The soundness proof for VST [2] lifts the single-threaded correctness theorem of the CompCert compiler [12] to a concurrent setting, and relies explicitly on style 1 lock specs; modifying this proof to rely on atomic operations instead is both theoretically and practically daunting. We demonstrate that 1) **our approach can be used to prove atomic specs for interesting data structures with fine-grained locking, starting from style 1 lock specs**, and 2) **these proofs require some additional complications that can be avoided with style 2 lock specs**. The key difference is that in style 1 a thread must gain ownership of a handle to a component's lock before acquiring the lock and reading/modifying the component, while in style 2 the locks and components alike can be considered part of a single abstract state that is accessed at each lock access. However, the top-level specifications proved for the data structure are not affected by the complexity of the lock specs, so proofs of clients of the data structure are no more complex than they would be otherwise.

To our knowledge, this is the first formal comparison of the two styles of lock specifications, and our technique should be useful in tools like VST where the older style is deeply integrated into the tool. This is also the first *foundational* verification of a C implementation against logically atomic specifications; prior tools such as VeriFast [7] have been used to verify C implementations, but are not proved sound against a C semantics and so can avoid the complexities of lock specs w.r.t. concurrent soundness.

2 Background

2.1 Logically Atomic Specifications

For many concurrent data structures, the ideal correctness condition is that the data structure behaves the same as a sequential implementation, even when accessed simultaneously by multiple threads. This intuitive condition has been formalized in terms of linearizability (operations appear to take effect in some total order) or atomicity (client threads never observe intermediate states of operations). In separation logic, atomicity can be expressed in the form of *atomic triples* [14] of the form

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

where P_l and Q_l are *local* pre- and postconditions similar to an ordinary Hoare triple, and P_p and Q_p are *public* pre- and postconditions, parameterized by an abstract value

a of the shared data structure. This triple says that if P_l is true before a call, and the fact that P_p is true for some value of a is held in shared state (e.g. a global invariant), then P_p will continue to be true for some (possibly different) value of a up until the *linearization point* of c, at which point Q_p will become true atomically for the same value a (and Q_l will be true after c ends). The specific value of a (the shared abstract state) can vary as other threads modify the data structure, as long as $P_p(a)$ is always maintained. For instance, the specification

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

expresses the fact that the push operation of a concurrent stack correctly implements the behavior of a sequential push, making no visible changes to whatever stack s is in place until, at the linearization point, the current stack s is replaced atomically with $v :: s$. The stack itself is treated as a publicly owned resource, and can only be accessed and modified atomically by threads holding the is_stack assertion.

2.2 Two Styles of Lock Specification

Soon after the introduction of concurrent separation logic [13], both Gotsman et al. [5] and Hobor et al. [6] extended it with support for C-style storable locks, where a lock is a special memory location designated as coordinating access to some other resource. Their lock specifications were essentially equivalent, and of the form:

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

Each lock ℓ is associated with a "lock invariant" R, an arbitrary separation logic predicate representing the protected resource, which is gained by a thread that acquires the lock and must be restored when the lock is released. At the time, there was no formal treatment of atomic operations in CSL, so actual lock implementations were beyond the scope of verification: these specs were taken as axioms, or proved through direct appeal to data-race-free guarantees of specific memory models. In garbage-collected languages that do not deallocate locks, the assertion $\ell \boxdot\!\!\to R$ can be freely duplicated and shared among threads accessing R; if we want to track ownership and eventually deallocate the lock, we can instead split the assertion into fractional shares such that $\ell \boxdot\!\!\to_{\pi_1} R * \ell \boxdot\!\!\to_{\pi_2} R = \ell \boxdot\!\!\to_{\pi_1+\pi_2} R$, and then allow deallocation only when we own the full share $\ell \boxdot\!\!\to_1 R$.

Later, the TaDA logic [14], which codified the notion of logical atomicity, presented an alternative set of lock specifications:

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

An acquire operation may access the lock repeatedly, finding it in either the locked (L) or unlocked (U) state; if it sees the unlocked state (b holds), then it can atomically move to a locked state and return. A release simply moves atomically from the locked state to the unlocked state. These operations do not specify what is protected by the lock, but because they are atomic, they can interact with any global invariant ("shared region" in TaDA's terminology): in particular, in combination with the invariant $(\mathsf{U}(\ell) * R) \lor \mathsf{L}(\ell)$, they can be used to derive the lock-invariant-based specs. It is also worth noting that these lock specifications are not axioms: they are proved for a simple spinlock implementation that uses atomic operations (e.g. compare-and-swap), which are the actual primitive operations in TaDA and the logics that follow it. In this style, tracking ownership of a lock is orthogonal to the lock assertion itself; a mechanism such as cancelable invariants [3] can be used to both make the lock assertion U/L publicly available and deallocate it once all threads are finished.

2.3 VST and Iris

We employ a combination of two Coq-based verification systems, the Verified Software Toolchain (VST) and Iris. VST [1] is a Coq-based system for proving separation logic specifications of C programs. It is proved sound against not only the C semantics of the CompCert compiler, but also the compiler correctness theorem [2], so that verified programs are guaranteed to behave correctly when compiled and run. VST directly implements the concurrent separation logic of Hobor et al. [6], and the invariant-style lock specifications are incorporated into its soundness proof at a fundamental level.

Iris [9] is a framework that synthesizes ideas from a range of CSLs with the idea that "ghost state is all you need". Iris begins with a small core logic including arbitrary higher-order ghost state, and derives features including invariants and logical atomicity from particular instances of ghost state. Iris is designed as a language- and logic-independent framework, and recent versions of VST take advantage of this, incorporating Iris-style ghost state in the foundational model and directly using Iris tactics to reason about invariants and atomicity of C implementations. All proofs described in this paper have been formalized in this VST+Iris setting. Iris has also been used as the basis for proofs with atomic lock specifications, such as the work of Krishna et al. [10].

3 Locks and Atomicity

In this section, we outline our approach to connecting lock invariants and atomic specifications, and contrast it with the atomic-lock-based approach.

Consider a sequential implementation of a binary search tree. It has functions `insert`, `lookup`, and `delete`, proven to satisfy the following specifications:

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

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

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

where $\mathsf{BST}\ p\ t$ asserts that some abstract binary search tree t is represented in memory starting at location p. For instance, t could be an element of a recursively defined tree type in Coq, and BST could be defined as a recursive predicate that represents each node of t as an object in memory, with fields for key, value, and pointers to left and right children, with the root node located at p.

A thread-safe version of this data structure should satisfy atomic specifications that closely correspond to the sequential specs[5]:

$$\langle m.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m\rangle\ \texttt{insert}(p,k,v)\ \langle \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ (m[k \mapsto v])\rangle$$

$$\langle m.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m\rangle\ \texttt{lookup}(p,k)\ \langle v.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m \wedge m(k) = v\rangle$$

$$\langle m.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m\rangle\ \texttt{delete}(p,k)\ \langle \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ (m[k \mapsto \_])\rangle$$

These specifications assert that each operation appears to execute atomically, accessing and updating the state of the data structure without exposing any intermediate states—for instance, a `lookup` should not find a partially inserted key with a value other than the one passed to its `insert` call. We split the BST assertion into two pieces: bst_ref contains the points-to and lock-invariant assertions that a thread needs in order to call a BST function, and bst_abs contains the ghost state that describes the overall state of the tree and is treated as an atomically accessed shared resource. They are connected by reference to a fixed but arbitrary ghost state identifier g, which we will generally omit. We change the abstract state of the data structure from a tree t to a key-value map m to reflect the fact that clients cannot observe the internal structure of the tree (which will be useful when we want to rearrange nodes during concurrent access). Our sequential implementation will almost certainly not satisfy these specifications, because its functions relied on being able to access any part of the data structure at any time: the BST assertion included full ownership of every location in the data structure.

To show that an implementation satisfies these atomic specifications, we must show that 1) the implementation can execute safely without owning any piece of the public assertion bst_abs, only accessing it atomically and restoring it up until the linearization point, and 2) at some linearization point, the implementation atomically transforms the current map m into one that satisfies the public postcondition. The linearization point may be a (physically or logically) atomic operation, or it may occur between steps of the program (e.g., if the current thread holds a lock on the modified section of the data structure, so other threads cannot distinguish between the original state and the modified state until the lock is released). In the following sections, we discuss the general structure of these proofs for lock-based implementations using each of the two kinds of lock specifications.

3.1 Atomicity with lock invariants

The easiest way to synchronize a data structure is to add a lock that each operation acquires at the start and releases at the end. These *coarse-grained* locking implementations are easy to verify with invariant-style lock specs: the lock's invariant can contain the whole data structure assertion (e.g. BST $p\ t$), so that a thread always has full ownership of the data structure when performing an operation. If we only want to prove that the operation is thread-safe, we might choose the lock invariant $\exists t.$ BST $p\ t$, asserting that we gain full ownership of the tree in some state t every time we acquire the lock.

[5] We also have functions to create and deallocate trees, but these functions need not be thread-safe, since the tree will not be shared when created and must not be shared when deallocated.

On the other hand, if we want to prove specifications that describe changes to the data structure—as in the atomic specification $\langle m.\ \mathsf{bst\_abs}\ m\rangle$ `insert`(p, k, v) $\langle \mathsf{bst\_abs}\ (m[k \mapsto v])\rangle$—we need ghost state to track the changes.

Specifically, we can define a piece of ghost state $\mathsf{ghost\_bst}\ t$ that is divided between the lock invariant and the public pre- and postconditions of the functions, connecting the tree in memory to the public abstract state of the data structure. For instance, we might choose a lock invariant $R_{cg} \triangleq \exists t.\ \mathsf{BST}\ p\ t * \mathsf{ghost\_bst}_{.5}\ t$, and then set $\mathsf{bst\_ref}\ p \triangleq p \boxdot\!\!\rightarrow_{\pi} R_{cg}$ and $\mathsf{bst\_abs}\ t \triangleq \mathsf{ghost\_bst}_{.5}\ t$. Then we can prove the atomic triples above, where at the linearization point we access $\mathsf{bst\_abs}$, combine the two halves of the ghost state and update them to reflect the new state of the concrete data structure, and then fulfill the atomic postcondition with one half of the ghost state and return the other half to the lock invariant. The atomic postcondition reflects the fact that the abstract state has been accessed or changed, while the lock invariant maintains the connection between the abstract state and the actual data structure in memory. This approach to making changes to a locked data structure visible via ghost state is due to Zhang and Jung's proofs of an atomic syncer in Iris [15].

In a *fine-grained* implementation, there are multiple locks on different pieces of the data structure, so the ghost state of each lock invariant cannot be the abstract state of the entire data structure. Instead, the ghost state represents the abstract state of the locked section alone, and the abstract state of the data structure is derived from the composition of the states of the locked components. Each locked component c has a piece of ghost state $\mathsf{ghost\_}c$ that is split between the lock invariant and the top-level abstract state. At the end of a critical section for c, we atomically access the top-level abstract state, combine the two halves of $\mathsf{ghost\_}c$, update $\mathsf{ghost\_}c$ to reflect any local changes, and then show either that the update to c does not change the top-level state of the data structure, or that the update to c changes the top-level state to one that satisfies the atomic postcondition for the function. For instance, when deleting a node from a binary search tree, we rotate nodes to reconstruct a valid tree; at the end of the critical section for each rotation, we show either that the values in the tree are unchanged (and the tree is still well-ordered), or that we have removed the node to be deleted. As we will see in section 4, the lock invariant for each component must also carefully account for the ownership of both that component's lock and the locks of related nodes (e.g., child nodes in a tree).

3.2 Atomicity with atomic locks

Proving atomic specifications for a data structure using atomic specifications for locks is more immediate. As an example, consider Krishna et al.'s verification of a hand-over-hand locking pattern for search structures [10]. The lock specification used[6] is

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

where $\mathsf{inFP}(n)$ asserts that the node n is in the data structure, CSS is the abstract state, and $\mathsf{N}(n)$ is the resources for the node n (including both points-to assertions and ghost

[6] In fact, this specification is derived from the basic TaDA-style atomic lock specificaton.

state). The abstract state assertion CSS includes both global ghost state and the composition of per-node state N for each node that is not currently locked, so that $\mathsf{N}(n)$ can be removed from the abstract state when n's lock is acquired.

Note that in this case, the local precondition $\mathsf{inFP}(n)$ is merely the knowledge that n is in the data structure, and does not involve any ownership—in particular, a caller does not need a share of the lock for the root node. A thread can acquire the lock for any node n in the data structure by atomically accessing CSS, extracting the sub-assertion corresponding to n, and switching it to the locked state. More generally, in this approach there is no need to track ownership of the locks of individual nodes, which as we will see in section 4.1 is a source of considerable complexity in lock-invariant-based proofs.

4 A Verified Binary Search Tree with Hand-over-Hand Locking

We demonstrate our approach to proving logically atomic specifications with invariant-based (type 1) lock specs by verifying a binary search tree (BST) with fine-grained locking in VST, using the approach of section 3.1 to prove atomic specifications for the BST operations while using invariant-style specs for our locks (as required by VST's soundness proof). As described in section 3, we aim to define assertions $\mathsf{bst\_ref}$ (the per-thread handle to the BST) and $\mathsf{bst\_abs}$ (the shared abstract state) such that the operations of the concurrent BST satisfy atomic triples along the lines of

$$\langle m.\ \mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ m\rangle\ \mathtt{insert}(p,k,v)\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ (m[k \mapsto v])\rangle$$

In this section, we describe the construction of the two predicates and the proofs of the operations for a BST implementation with hand-over-hand locking, using lock-invariant-based specs for our locks.

4.1 Hand-over-Hand Locking

Hand-over-hand locking (also called lock coupling) is a fine-grained locking pattern for traversing a data structure in which we acquire the lock for the next node before releasing the lock for the current node, thus guaranteeing that the link between the two nodes is not removed or rearranged while we traverse it. Hand-over-hand locking is a classic example of a pattern that requires lock invariants that refer to other lock invariants, dating back to the earliest papers on "predicates in the heap" in CSL [5]. The trick is to assign each node in the data structure a lock invariant that includes the lock assertion for its child nodes; for a linked list, for instance, we might write

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

where n' is the child of n and ℓ' is n''s lock. By acquiring n's lock, we learn the lock assertion for n', and can acquire its lock as needed before releasing n.

This invariant becomes slightly more complicated when we consider *ownership* of locks, as is required if we want to eventually free the lock and return its resources. In this case the lock assertion carries a share π (often represented as a rational number in the range $(0, 1]$), where the full share $\pi = 1$ is required to free the lock while any share is sufficient to acquire or release it. We can amend our lock invariant to

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

for some fixed share π, but then, what happens to the share of ℓ' when we release the lock on n? Starting from a state in which we hold n's lock ℓ, this gives us:

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

In the call to `release`, we give up resources to reestablish ℓ's lock invariant, including the lock assertion for ℓ'. As a result, we are left with a share of ℓ (the lock for the node we left behind) and no share of ℓ' (the lock for the current node).

Both of these problems can be solved by making the lock *recursive*, including a share of ℓ in ℓ's own lock invariant[7]. We split ownership of the lock ℓ for a node n into two shares: π_1 is held by ℓ itself, and π_2 is held by the lock of n's parent. Then we amend definition 1 to

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

Now the same sequence of operations gives us:

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

When we acquire ℓ', we hold both π_1 and π_2 of its lock assertion; when we release ℓ, we return share π_1 of ℓ along with share π_2 of ℓ', but retain share π_1 of ℓ', so we can release it later. The mechanism for defining a lock invariant that includes a share of its own lock assertion may vary across separation logics (VST provides a selflock assertion for exactly this purpose), but as long as such a mechanism exists, this pattern can be used to implement hand-over-hand lock invariants with share accounting, allowing us to traverse a data structure built with this pattern and then reclaim all of its resources once all threads are finished with it.

This pattern is the main source of complexity in our proofs that is not present in proofs with atomic lock specifications. For instance, in the hand-over-hand locking proofs of Krishna et al. [10], the only information a node holds about its children is the fact that they are in the data structure; their lock assertions are held in the global abstract state along with the rest of their resources. This significantly simplifies the specification and proof of the hand-over-hand synchronization.

[7] Gotsman et al. instead use a separation assertion Locked that is always held by the thread that acquires a lock, and is required to free the lock: this is functionally equivalent to an extra share of the lock assertion.

4.2 Specification of the hand-over-hand BST

The C implementation of a node in our BST is:

```
typedef struct tree {int key; void *value;
                     struct tree_t *left, *right;} tree;
typedef struct tree_t {tree *t; lock_t *lock;} tree_t;
```

Each node (of type `tree_t`) has a `lock` field that holds the lock protecting the node, and a `t` field that is either `NULL` (for a leaf, which contains no key or value) or points to a `tree` struct containing the node's key, value, and child pointers. To verify operations on this structure, we need to relate a `tree_t` in memory to an abstract state (a mathematical map from keys to values), and divide ownership of those resources between the assertions bst_ref (the resources held by each client thread) and bst_abs (the abstract state of the data structure). As described in section 3.1, our approach is to associate a piece of ghost state with each node in the tree, which will be shared between that node's lock and the abstract state in bst_abs, while bst_ref will be a a pointer to the lock of the root node. We also need to make sure that our lock invariants support the hand-over-hand pattern.

Key Ranges When we traverse the BST looking for a key k, we will reach either a node containing k, or an empty node where k would appear if it was in the tree. To prove correctness of this procedure, we need ghost state that tracks where each key "should appear" in the current tree. As observed by Krishna et al. [11], this can be done by associating each node with a lower and upper bound on the keys appearing in the subtree rooted at that node. We refer to the product of these bounds as a *range*. The range of a node is inherited from its parent, based on the parent's key: if node n has range (l, r) and key k, then its left child has range (l, k) and its right child has range (k, r). Figure 1 shows an example of a BST with each node labeled with its range, starting with $(-\infty, +\infty)$ at the root and propagated to the empty leaf nodes.

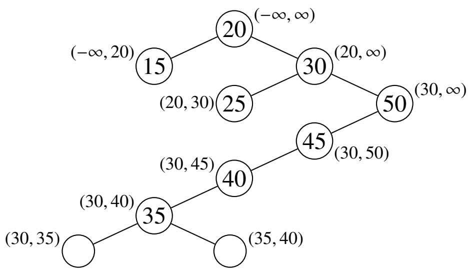

Fig. 1: A BST with each node labeled with its range; empty children of nodes other than 35 are omitted

At the leaves, these ranges partition the space of all keys not in the tree. If we reach a leaf node with range (a, b), then we know that keys between a and b 1) are not currently in the tree and 2) can correctly be inserted at this leaf. For example, suppose we want to insert a key 38 into the tree in figure 1. The right child of the node with key 35 is a leaf with range $(35, 40)$, so it is guaranteed to be the right place to insert 38.

Per-Node and Global Ghost State Importantly, the fact that the leaf ranges partition the space of possible keys remains true even if operations in other threads restructure the tree during our traversal. Furthermore, operations further up the tree can only *increase* the range of nodes below them, so if we find a node whose range includes our key, we are guaranteed to have found the right place for it. This makes ranges ideal for use as ghost state: range information held by one thread will not be invalidated by other threads, and is sufficient to show correctness of BST operations. More precisely, each node's ghost state, identified by a ghost name g, contains a pair of its range and its contents, where the contents are None for a leaf node and $\mathsf{Some}\ (k, v, g_l, g_r)$ for an internal node with key k, value v, and left and right children with identifiers g_l and g_r. The predicate $\mathsf{node\_ghost}_g\ ((i, j), n)$ asserts that the node with identifier g has range (i, j) and contents n, and will be divided between the lock invariant of the node and the abstract state assertion $\mathsf{bst\_abs}$.

The $\mathsf{bst\_abs}$ predicate describes the conditions under which an abstract map m is implemented by the ghost state of a tree. Intuitively, we should be able to gather the per-node ghost states into an abstract tree t that implements the map m, in that each key-value pair in m appears in one of the nodes of t. We formalize this by first defining the collection of per-node ghost state fragments needed to form an abstract tree:

$$\begin{aligned}
&\mathsf{public\_halves}(t, g, (i, j)) \triangleq \mathsf{match}\ t\ \mathsf{with}\\
&\quad \mid \mathsf{Leaf} \Rightarrow \mathsf{node\_ghost}^{.5}_g\ ((i, j), \mathsf{None})\\
&\quad \mid \mathsf{Node}\ (k, v, t_l, g_l, t_r, g_r) \Rightarrow \mathsf{node\_ghost}^{.5}_g\ ((i, j), \mathsf{Some}\ (k, v, g_l, g_r)) *\\
&\qquad\quad \mathsf{public\_halves}(t_l, g_l, (i, k)) * \mathsf{public\_halves}(t_r, g_r, (k, j))
\end{aligned}$$

We recursively walk through the tree, starting from a node g with range (i, j), and collect the $\mathsf{node\_ghost}$ assertions of each node into a single assertion. We compute the range of each node's children as described above: if the key in the current node is k, then the left child has range (i, k) and the right child has range (k, j).

This collection of per-node ghost state describes the contents and structure of the tree, but a thread that acquires a node's lock also needs to know that the node actually appears in the tree t. For this purpose, we define ghost state assertions $\mathsf{ghost\_nodes}(S)$ and $\mathsf{in\_tree}\ g$, where S is a set of identifiers and g is an identifier, such that $\mathsf{in\_tree}\ g * \mathsf{ghost\_nodes}(S)$ implies that $g \in S$. Then the full definition of $\mathsf{bst\_abs}$ is:

$$\begin{aligned}
\mathsf{bst\_abs}_g\ m \triangleq\ &\exists t.\ t \text{ implements } m \wedge \mathsf{public\_halves}(t, g, (-\infty, +\infty)) *\\
&\mathsf{ghost\_nodes}(\mathrm{ids}(t))
\end{aligned}$$

where $\mathrm{ids}(t)$ is the set of all node identifiers in the abstract tree t, and t implements m iff the key-value pairs in t are exactly those in m. The range of the root node (named g) is $(-\infty, +\infty)$, and the $\mathsf{ghost\_nodes}$ assertion guarantees that if any thread or lock holds an assertion $\mathsf{in\_tree}\ g_i$, then g_i is a node in t. Thanks to the existential quantification over t, we can rearrange its nodes at any time, as long as it remains a well-formed tree with the same set of node IDs and key-value pairs. We will take advantage of this fact when verifying the `delete` function.

Next, we connect the per-node locks of our C implementation with the global abstract state of bst_abs by associating each node's lock with a lock invariant assertion $\ell \mapsto R$, where R includes both the concrete points-to assertion for the node and the corresponding abstract node_ghost. The lock invariant for a node at location p needs to map the abstract contents c to the values present in memory in the `t` field of p:

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

We then combine this with the hand-over-hand invariant pattern of section 4.1, giving us a final lock invariant of:

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

The hand-over-hand pattern allows us to acquire $p_l.$`lock` or $p_r.$`lock` and then release $p.$`lock` without losing any shares of the lock assertions.

Now we can fill in the pieces of the specifications outlined in section 3:

$$\langle m.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m\rangle\ \mathtt{insert}(p, k, v)\ \langle \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ (m[k \mapsto v])\rangle$$

$$\langle m.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m\rangle\ \mathtt{lookup}(p, k)\ \langle v.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m \wedge m(k) = v\rangle$$

$$\langle m.\ \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ m\rangle\ \mathtt{delete}(p, k)\ \langle \mathsf{bst\_ref}_g\ p \mid \mathsf{bst\_abs}_g\ (m[k \mapsto \_])\rangle$$

where $\mathsf{bst\_abs}_g\ m$ asserts that global ghost state of the tree implements a map m, as described above, and bst_ref (the client's handle to the data structure) is simply a reference to the lock for the root node:

$$\mathsf{bst\_ref}_g\ b \triangleq \exists p.\ b \mapsto p * p.\mathtt{lock} \mapsto \mathsf{node\_inv}_g(p)$$

Next, we will prove that each of our BST functions satisfies its specification.

4.3 Proofs: Insert and Lookup

The code for the insert and lookup functions is shown in figure 2. The traversal process is the same in each function: we access the current node's key, compare it to the target key `k`, and move to the left child if `k` is less than the current key and right if it is greater, using hand-over-hand locking. Thus, both loops have the same invariant. At the beginning of each iteration, we are at a node at pointer p with ghost identifier g and hold the lock for that node, giving us the lock invariant $\mathsf{node\_inv}_g(p)$ with some range (i, j). The loop invariant requires that `k` $\in (i, j)$, that is, the target key always belongs

```
 1 void insert(tree_t** t, int k, void *v){ void *lookup(tree_t** t, int k){
 2   tree_t *tgt = *t;                          tree_t *tgt = *t;
 3   acquire(tgt->lock);                        acquire(tgt->lock);
 4   while(1) {                                 while(1) {
 5     tree* p = tgt->t;                          tree *p = tgt->t;
 6     if (p == NULL) {                           if (p == NULL) {
 7       ... // make a new node with k, v
 8       release(tgt->lock);                        release(tgt->lock);
 9       return;                                    return NULL;
10     }                                          }
11     if (k < p->key){                           if (k < p->key){
12       void *l_old = tgt->lock;                   void *l_old = tgt->lock;
13       tgt = p->left;                             tgt = p->left;
14       acquire(tgt->lock);                        acquire(tgt->lock);
15       release(l_old);                            release(l_old);
16     } else if (p->key < k){                    } else if (p->key < k){
17       ... // move to p->right instead            ... // move to p->right instead
18     } else {                                   } else {
19         p->value = v;                            void* v = p->value;
20         release(tgt->lock);                      release(tgt->lock);
21         return;                                  return v;
22     }                                          }
23   }                                          }
24 }                                          }
```

Fig. 2: Code outlines for `insert` and `lookup`

in the subtree of the current node. This guarantees that if we reach an empty leaf, that leaf is the place where key k should be held. The actual logic of each operation happens when we either reach an empty leaf node (line 7) or find a node with k as its key (line 19); there are also the linearization points of the two functions.

When insertion encounters a node with key k, it simply changes that node's value to `value`. If it reaches a leaf, it allocates a new node with key k and value v, and inserts it at that position. In both cases, we must show that we can atomically update the abstract state of the tree from bst_abs m to bst_abs(m[k $\mapsto$ v]), by showing that the update to the tree in memory corresponds to a local change in the abstract tree that inserts the new key-value pair. The proofs are split into a proof that the C code implements the local update, and a proof that the local update to a ghost-state tree implements the global update. In the first case, the local change is the insertion of (k, v) at an empty leaf whose range includes k, and the addition of two new empty leaves as its children; in the second case, we simply change the value of an internal node whose key is already k.

The proof of `lookup` is similar, but simpler. We must show that if the key is found in the tree, then the same key-value pair exists in the abstract state, and that if the key is not found then it is not present in the abstract state. The former follows from the lock invariant node_inv, which guarantees that the key and value in memory match the key and value in the ghost node; the latter follows from the ranges, since if the abstract tree

```
 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        ... // move lc->t to tgt; deallocate p, lc, and 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  }
```

Fig. 3: Code outline for `pushdown_left`

contains a leaf node with range (i, j) such that $\mathtt{k} \in (i, j)$, it cannot contain a node with key `k`. In either case, the abstract state remains unchanged.

4.4 Delete

The `delete` function uses the same traversal logic as `insert` and `lookup`, but when it finds the node with the target key, it calls a helper function `pushdown_left`, outlined in figure 3. This function rearranges the nodes in the tree, moving the node to be deleted down through the tree until its right child is empty while maintaining the binary search property. This is accomplished by repeatedly rotating sets of three nodes: the target node's right child `rc` is moved up to become its parent, with the target node on the left and the right child's right child moved under the target node, as illustrated in figure 4. Once the node to be deleted has no right child, its contents are replaced with those of its left child, and its children are removed from the tree.

The `pushdown_left` function changes the structure of the tree in two ways. First, the rotation `turn_left` (line 14) changes the ranges of the node to be deleted and its right child (e.g., nodes 40 and 55 in the first step of figure 4); this is a local change, as the ranges of all other nodes (even the children of the rearranged nodes) are preserved. This is not a linearization point, so the rotation must be considered not to change the abstract state of the tree. We accounted for this by quantifying over the abstract tree t in the definition of $\mathsf{bst\_abs}$; rearranging nodes without changing the key-value map still implements the same abstract state, and other threads cannot assume that the tree structure they have observed is still the current structure of the tree. Second, the final removal of the node (line 10) increases the ranges of the subtree below it; for example,

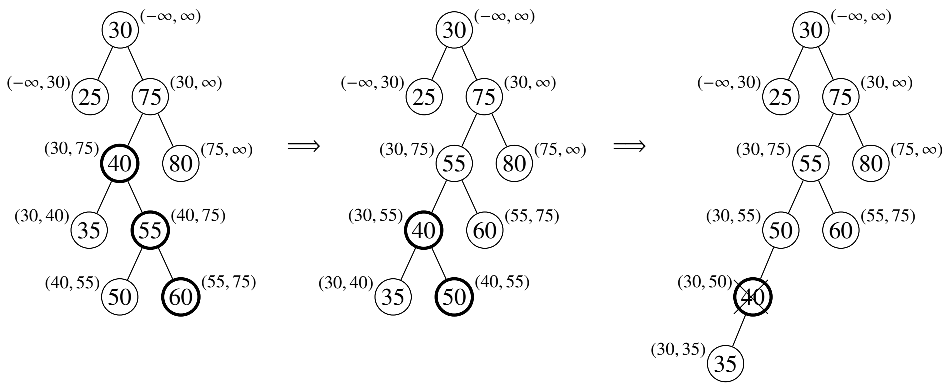

Fig. 4: Deleting node 40 from the tree via `pushdown_left`

in the last step of figure 4, the range of node 35 changes from $(30, 40)$ to $(30, 50)$ when the node is removed. This is the linearization point of the function. The change occurs while `pushdown_left` holds only the lock of the deleted node, so the algebra for range ghost state must allow a node's range to increase at any time, even when the node's lock is held. Fortunately, `insert` and `lookup` only use ranges to assert that the target key is in the range of the current node, which is still true if the range increases.

5 Using the Specifications

To demonstrate that the complexities of our lock-based proofs are hidden from prospective users, we verified a simple client program for the binary search tree, shown in figure 5. A producer thread running the `update_tree` function populates a tree, while a consumer thread running `retrieve_value` waits until key 2 is associated with a pointer to the value 4, and then returns the value associated with key 1. This is a data-structure-level version of the message-passing idiom, and demonstrates one of the advantages of atomic specifications: we can use them to synchronize threads and pass ownership of resources via data structure operations. The invariant for the BST `t` is a simple state machine with three states, depending on the contents of the abstract map m: 1) $m(2) = \texttt{NULL}$, 2) $m(2) \mapsto i$ for some integer $i \neq 4$; and 3) $m(2) \mapsto 4 \wedge m(1) \mapsto 3$. In this final state, the invariant also holds the resources to be transferred from the producer to the consumer. We also include ghost state to reflect the fact that there is exactly one producer thread (so the producer always knows exactly which values are in the tree) and one consumer thread (so it can retrieve the resources without worrying about interference). The proof of the client uses only the atomic specifications of the BST operations, and is not affected by the style of lock specification.

6 Related Work

Gotsman et al. [5], in the same paper in which they introduced invariant-style lock specs, also verified a linked list with hand-over-hand locking, which became a common

```
void* update_tree(void* t){
  a = 1;
  b = 2;
  c = 3;
  d = 4;
  insert(t, 1, &a);
  insert(t, 2, &b);
  insert(t, 1, &c);
  insert(t, 2, &d);
  return NULL;
}
```

```
int retrieve_value(treebox t){
  int y = -1;
  do {
     searchResult = lookup (t, 2);
     if (searchResult != NULL) {
        y = *((int *) searchResult);
     }
  } while (y != 4);
  void* searchResult = lookup (t, 1);
  return *((int *) searchResult);
}
```

Fig. 5: Client code

example for CSLs that handled fine-grained locking. Among the most recent of these is the work of Krishna et al. [10], who used atomic lock specs to prove correctness of the same data structure. This work updates the work of Gotsman et al. with modern specifications, and directly compares the results to those of Krishna et al.

VeriFast [8], a separation-logic-based verifier for C and Java, supports lock-based and atomic concurrency [7], and has been used to verify a hand-over-hand-locking linked list similar to that of Krishna et al. The specifications of the lock operations and the list itself use a precursor of TaDA's logical atomicity. VeriFast is not foundational, but its basic logic is verified against the semantics of a toy language in Coq.

7 Conclusion

While most new CSL proofs of lock-based data structures use TaDA-style atomic specs for their locks, the implications of the switch away from invariant-based specs had not been thoroughly examined. We have now demonstrated that invariant-based specs can still be used to prove logically atomic specs for larger data structures (even for fine-grained implementations) and that this requires somewhat more complex proofs than using atomic lock specs (especially for fine-grained implementations). The old style of lock spec may still be used in systems like VST, where switching would invalidate existing automation or soundness proofs, and in these systems the approach we have outlined can be used to obtain strong correctness properties for fine-grained-locking data structures. When possible, however, we would prefer to use atomic specifications for locks to simplify our proofs. Indeed, we intend to investigate shifting the foundations of VST to use atomic operations and derive atomic specs for locks: while updating the soundness proof will be a considerable effort, using the newer style of lock specification will simplify all future proofs about lock-based data structures.

Acknowledgements This work benefited greatly from correspondence with Ralf Jung, who pointed out Zhang's syncer approach and explained the finer points of logical atomicity in Iris.

References

1. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, USA (2014)
2. Cuellar, S., Giannarakis, N., Madiot, J.M., Mansky, W., Beringer, L., Cao, Q., Appel, A.: Compiler correctness for concurrency: from concurrent separation logic to shared-memory assembly language. Tech. rep., Princeton University (2020)
3. Dang, H.H., Jourdan, J.H., Kaiser, J.O., Dreyer, D.: RustBelt meets relaxed memory. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371102, https://doi.org/10.1145/3371102
4. D'Osualdo, E., Farzan, A., Gardner, P., Sutherland, J.: TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. ACM Transactions on Programming Languages and Systems (TOPLAS). (2021)
5. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Shao, Z. (ed.) Programming Languages and Systems. pp. 19–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
6. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) Programming Languages and Systems. pp. 353–367. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
7. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 271–282. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926417, https://doi.org/10.1145/1926385.1926417
8. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 637–650. ACM (2015). https://doi.org/10.1145/2676726.2676980, https://doi.org/10.1145/2676726.2676980
10. Krishna, S., Patel, N., Shasha, D., Wies, T.: Verifying concurrent search structure templates. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 181–196. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386029, https://doi.org/10.1145/3385412.3386029
11. Krishna, S., Shasha, D., Wies, T.: Go with the flow: Compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158125, https://doi.org/10.1145/3158125
12. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (jul 2009). https://doi.org/10.1145/1538788.1538814, https://doi.org/10.1145/1538788.1538814
13. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science **375**(1), 271–307 (2007). https://doi.org/https://doi.org/10.1016/j.tcs.2006.12.035, https://www.sciencedirect.com/science/article/pii/S030439750600925X, festschrift for John C. Reynolds's 70th birthday
14. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014 – Object-Oriented Programming. pp. 207–231. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

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