1 INTRODUCTION

The binary search tree (BST) is a common implementation of an ordered map, a widely used data structure. The concurrent version of the ordered map forms the bedrock of many parallel programs. We formally verified the functional correctness of two versions of fine-grained concurrent binary search trees (CBST) written in C. One adopts the hand-over-hand locking technique [Bayer and Schkolnick 1977], the other is lock free by using atomic primitives such as compare and swap (CAS). Both CBST implementations support insert, lookup, and delete operations; they also share the "same" specifications to some extent in our verification. All the proof is machine-checked in Coq.

Our specifications use the logical atomicity introduced in the TaDA logic [da Rocha Pinto et al. 2014], in the form of *atomic Hoare triples*. Intuitively, an atomic Hoare triple $\langle P \rangle\ \mathbb{C}\ \langle P \rangle$ means the program $\mathbb{C}$ "atomically updates" from P to Q. The program may actually take multiple steps, but evey step before the atomic update (linearization point) must preserve the assertion P. Meanwhile, the concurrent environment may also update the state before the linearization point, as long as the states satisfies P. The assertion Q must become true at the linearization point, then the environment can do whatever it likes. There is no guarantee that Q is still preserved when $\mathbb{C}$ returns. For example, the specification of our `insert` operation may be explained as follows: during the execution of `insert`, there is always *some* BST, and at some point the `insert` will take a BST t, insert a value with a certain key, and then eventually returns (meanwhile other threads may have further modified the inserted tree).

We employ the Verified Software Toolchain (VST) [Appel et al. 2014] to verify the correctness of CBST. Although the concurrent separation logic (CSL) has been formalized in VST rooted on the work of Hobor et al. [Hobor et al. 2008], we extend it with two descendants of CSL so as to accomplish the verification. One is the logical atomicity mentioned above; the other is the higher-order ghost state in the style of Iris [Jung et al. 2016]. The ghost states are used to contruct both the global invariants and the local state in our proofs. They will be further discussed in §??.

We highlights a few innovative aspects about our verification of CBST:

Range ghost We abstract the BST via a pair of values called *range* which represents the lower-bound and upper-bound of keys on each node to reason about the BST in the current settings. We prove that range with the merging operation forms a partial commutative monoid (PCM) so that it can be encoded in ghost states.

sync_inv pattern It is a particular approach combining locks with general invariants to solve the dilemma caused by the fine-grained locking mechanism: we do not have a lock to control the access of the entire BST nor can we access the state of the BST via atomic operations.

Our specific contributions are:

- To the best of our knowledge, this is the first mechanized verification of an concurrent search-based data structure written in a real programming language.
- We illustrate how to incorporate the CSL in VST, the higher-order ghost state in the style of Iris, and the logical atomicity from the TaDA logic together to verify the CBST.

We introduce the background about the verification of concurrent C programs in §2, including the tool-chain VST and Iris, the concept of ghost states, global invariants, and atomic specifications. The thread-safety proofs of operations on CBST are first explained in §??, where we show the *recursive* lock pattern for hand-over-hand locking mechanism. The functional correctness proofs are presented in §??. We detail the use of the sync_inv pattern, the combination of recursive lock invariants and ghost states together in the atomic specifications, and the proof skeleton of each operation on CBST. The related work is discussed in §6. We end with the conclusion of our work in §7.

2 BACKGROUND

2.1 VST and Iris

We use the Verified Software Toolchain [Appel et al. 2014] to prove the correctness of the concurrent binary search tree. VST is a Coq-based proof system that helps to prove separation logic properties of C programs with the assurance that the properties will be preserved in compiled code. First, we write programs in C programming and using CompCert's *clightgen*, the ASTs for C programs can be generated in Coq. We then state and prove specifications for them using Verifiable C. Verifiable C is a higher-order separation logic for reasoning about the functional correctness of C programs and carries a soundness proof that links high-level specifications to assembly code generated by the CompCert Verified compiler. Verifiable C supports reasoning about two separate kinds of concurrency: Pthreads-style concurrency with locks (used by implementing blocking data structures that can be fine-grained or coarse-grained), and concurrency using C11 atomic operations (i.e. lock-free data structures).

2.2 Ghost State and Global Invariants

We can prove memory safety and race freedom properties of shared-memory concurrent programs, by tracking the transfer of ownership of shared locations between threads that would happen if we ran the program. But, this idea is insufficient to prove that concurrently running threads are successful to accomplish some task, which is also called functional correctness. For instance, we can prove the memory safety properties of the increment example in 1 through the transfer of ownership of shared variable x between threads. But, we can not guarantee that the value of x will be 2 after all threads complete their execution, which is demonstrated in 1 using separation logic assertion. We must preserve the connection between the value

$$
\begin{array}{c}
\texttt{x = 0;} \\
I_{\pi}(x \mapsto v \land v \geq 0) \qquad \pi = \pi_1 \,.\, \pi_2 \\
\begin{array}{l||l}
I_{\pi_1}(x \mapsto v \land v \geq 0) & I_{\pi_2}(x \mapsto v \land v \geq 0) \\
\texttt{acquire(l);} & \texttt{acquire(l);} \\
x \mapsto v \land I_{\pi_1}(x \mapsto v \land v \geq 0) & x \mapsto v \land I_{\pi_2}(x \mapsto v \land v \geq 0) \\
\texttt{x++;} & \texttt{x++;} \\
x \mapsto v+1 \land I_{\pi_1}(x \mapsto v \land v \geq 0) & x \mapsto v+1 \land I_{\pi_2}(x \mapsto v \land v \geq 0) \\
\texttt{release(l);} & \texttt{release(l);} \\
I_{\pi_1}(x \mapsto v \land v \geq 0) & I_{\pi_2}(x \mapsto v \land v \geq 0)
\end{array} \\
I_{\pi}(x \mapsto v \land v \geq 0)
\end{array}
$$

Fig. 1. The increment example annotated with separation logic assertion

of shared resources and the work performed by each thread. A common approach to achieve this is to use *ghost variables*, an auxiliary state introduced later in the proof to track the local information about each thread. They do not appear in the original program. Program in 1 can be verified by creating ghost state, which tracks the latest action performed by a thread, for each thread with initial value of 0, and update with 1 after each thread increment the value of x. In VST, we can create *ghost state* for any Coq type in the form of an arbitrary *partial commutative monoid* (PCM), a set with a partially defined binary operation that is as associative as it can, commutative, and has a unit, as long as we can describe what happens when two elements of that type are joined together. VST provides ghost_var assertion to create a simple ghost state: ghost_var *sh a g* asserts that g is a *ghost name* (gname in Coq) associated with the value a, which may be of any type. We will see different kind of ghost states used to verify binary search tree in the following sections.

The main idea behind the logic from Iris [Jung et al. 2016], a mechnized higher-order concurrent separation logic framework, is the construction of *global invariant* as *ghost state. Global Invariant* is an invariant on the global (ghost and physical) state of program. We can open any invariant but must close it again before taking any steps of execution unless those execution are atomic. Thread can use the contents of *global invariant* during atomic operation if it can guarantee that no one will ever see an intermediate state in which invariant does not hold. The rules from Jung et al.[Jung et al. 2016] for creating and opening invariants are:

$$
\text{inv\_alloc}\;\frac{}{\triangleright P \vdash {\Rrightarrow}_{E}\ \mathsf{EX}\ i : \mathsf{iname}, \boxed{P}^{\,i}}
\qquad\qquad
\text{inv\_open}\;\frac{i \in E}{\boxed{P}^{\,i} \vdash {}^{E}{\Rrightarrow}^{E\setminus i}\ \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{E\setminus i}{\Rrightarrow}^{E}\ \mathsf{emp})}
$$

where invariants are provided in the form of assertion $\boxed{P}^{\,i}$, and states that P is maintained as an invariant on the global state with name i. The operatoris called “fancy update" operator,that allows to allocate, open, and close the invariants and the later $\triangleright$ operator is used for impredicativity (i.e. $\boxed{..\boxed{P}^{\,i}..}^{\,i}$).

2.3 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 can be formalized as 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* [da Rocha Pinto et al. 2014], written in the form $\forall a.\ \langle P_l \mid P_p(a)\rangle\ c\ \langle Q_l \mid Q_p(a)\rangle$, where P_l and Q_l are *private* 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. Intuitively, P_l and Q_l must be true before and after the call, while P_p must be true for some value of a at every point from the beginning of c until some designated linearization point, at which point Q_p becomes true atomically for the same value a. For instance, the specification

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

expresses the fact that the push operation of a concurrent stack correctly implements the behavior of a sequential push, atomically transitioning from some stack s to $v :: s$ at some point during its execution.

VST has encoded an atomic Hoare triple, and provide the way that matches the notation VST uses for normal specifications. Such specification is called *atomic specification*, and can be formalized in Coq as shown in 2. This is how we write pre- and postcondition in VST. W is the TypeTree representing the type of arguments passed with WITH clause; a is the abstract state for the triple; Ei and Eo are the sets of invariants names inside and outside the triple. The PROP clause describes things that that are true independent of program state, the LOCAL clause describes the values contained in C local variables, and the SEP clause

```
Program Definition insert_spec :=
DECLARE _insert
ATOMIC TYPE W OBJ a INVS Ei Eo
WITH ...
PRE [ ... ]
  PROP (...)
  LOCAL (...)
  SEP (P_l) | (P_p)
POST [ ... ]
  PROP ()
  LOCAL ()
  SEP (Q_l) | (Q_p)
```

Fig. 2. Atomic Specification in VST

represents the *separating conjunction* (*) of *spatial predicates*, predicates on some part of the memory. In VST, while proving that a function implements an atomic specification, the precondition will contains an atomic_shift assertion with the public pre- and postcondition, and the masks inside it. This atomic shift can be accessed through following two rules:

$$\text{atomic\_commit}\;\frac{\forall a, R * P_p\ a \Rrightarrow \mathsf{EX}\ y,\ Q_p\ a\ y * R'\ y}{\mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * R \Rrightarrow \mathsf{EX}\ y,\ Q\ y * R'\ y}$$

$$\text{atomic\_rollback}\;\frac{\forall a, R * P_p\ a \Rrightarrow P_p\ a * R'}{\mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * R \Rrightarrow \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * R'}$$

During commit, we must provide resources R which, in combination with the public precondition P_p, allow us to prove the public postcondition Q_p. Then, we gain access to an assertion Q required by the postcondition of the function, and leftover resources R'. During rollback, we provide resources R which, in combination with the public precondition, allow us to reestablish the public precondition; we then regain the atomic shift back in the proof, as well as leftover resources R'. The rollback is specially used to learn some relationship between pieces of information (e.g. ghost state) stored in the public precondition, while the commit is used to prove the public postcondition from the precondition, and obtain an assertion Q. To complete the proof of any function's specification, we must always perform commit to obtain Q; we can perform any number of roolbacks before that point, but after that point we lose the atomic shift and no access to the public precondition.

3 BST WITH HAND-OVER-HAND LOCKING

As described in section 2.3, we want the operations of our concurrent BSTs to satisfy atomic triples relating them to sequential operations:

$$\forall t.\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst}\ t \rangle\ \mathtt{insert}(p, k, v)\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst}\ (\mathsf{insert}\ t\ k\ v) \rangle$$

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

$$\forall t.\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst}\ t \rangle\ \mathtt{delete}(p, k)\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst}\ (\mathsf{delete}\ t\ k) \rangle$$

In our case, insert, lookup, and delete are simple Gallina functions on a functional tree data structure. The details of bst_ref and bst may vary by implementation, but we expect that bst_ref contains at least a pointer to the data structure, and bst contains some ghost state linking the concrete state of the data structure to the abstract tree t.

In the following sections, we describe the construction of these predicates, and the proofs of the operations, for BST implementations based on hand-over-hand locking and lock-free atomic operations.

3.1 Fine-grained locking and atomicity

To prove that an operation on a data structure satisfies a logically atomic specification, we must show that there are no visible intermediate states of the operation, i.e., that other threads see the data structure as unchanged until the linearization point at which the operation takes effect. In traditional concurrent separation logics such as VST's [Hobor et al. 2008], this is done by protecting the data structure with a single lock whose *lock invariant* describes consistent states of the data structure. In modern separation logics such as Iris [Jung et al. 2015] or FCSL [Sergey et al. 2015], we can make a series of atomic modifications to the data structure without continuously holding a lock, as long as one modification can be designated the *linearization point* at which the new state becomes public, and all prior changes are invisible to other threads. Fine-grained locking, in which the pieces of a data structure are each protected by a separate lock with a traditional lock invariant, requires us to synthesize these two approaches: we may make a series of non-atomic modifications to a part of the data structure as long as we hold its lock, but when we release the lock we need to ensure that either we have reached the linearization point, or our changes are invisible to other threads.

We accomplish this using a technique derived from unpublished work by Jung [Jung 2020] for treating the critical sections of locks as atomic operations. The key is to include in each lock invariant a piece of ghost state that represents the abstract state of the locked section. From the outside, the effect of the critical section is to atomically update that piece of ghost state, and all other changes are internal to the lock. These pieces of ghost state can then be composed to yield the abstract state of the entire data structure. In effect, this construction turns a traditional lock invariant into a *lock variant* $\ell \boxdot\!\!\rightarrow^{v} R$, where R is parameterized by a value that can be updated in the critical section, as long as it corresponds to a linearization point or invisible change to the abstract state of the data structure as a whole. This allows traditional lock invariants to interact with atomic shifts and be used to prove atomic specifications.

The construction of the lock variants is simple. We begin with a *part-reference* ghost state, a common construction for sharing information between an invariant and the threads that act on it, with elements $\mathsf{public\_half}(a)$ (the current state a) and $\mathsf{my\_half}_{\pi}(p)$ (partial information about the current state, annotated with a share π). When the share π is 1, we know that $p = a$. (do we want to define this ourselves, or reference it somewhere? note also that this is what Iris calls an authoritative algebra) A lock variant with an assertion $R(a)$ is then defined as

$$\ell \boxdot\!\!\rightarrow^{v}_{\pi} R \triangleq \ell \boxdot\!\!\rightarrow \exists a.\ R(a) * \mathsf{my\_half}_{\pi}(a)$$

The share π may be set to 1 if the lock's information about a is always fully up to date, or a smaller share if it can become outdated due to operations on other parts of the data structure. Since a is existentially quantified, a thread that changes the value of a forgets it after releasing the lock (as required in a lock invariant), but it must have synchronized with a $\mathsf{public\_half}(a)$ held elsewhere (usually in a global invariant) that will remember the change. Thus, the ghost state acts as a bridge between the component lock invariant and the global abstract state of the data structure.

From the atomic rules of section 2.3, we derive the following rules for accessing and modifying pieces of the global abstract state:

$$\text{sync\_commit}\ \frac{\begin{array}{c}\forall a.\ R * P_p\ a \Rrightarrow \exists x_1.\ \mathsf{public\_half}(g, x_1) * \exists x_0', x_1'.\ (x_0, x_1) \rightsquigarrow (x_0', x_1') \land \\ (\mathsf{my\_half}(g, sh, x_0') * \mathsf{public\_half}(g, x_1') \Rrightarrow \exists y.\ Q_p\ a\ y * R'\ y)\end{array}}{\mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * \mathsf{my\_half}(g, sh, x_0) * R \Rrightarrow \mathsf{EX}\ y,\ Q\ y * R'\ y}$$

$$\text{sync\_rollback}\ \frac{\forall a.\ R * P_p\ a \Rrightarrow \exists x_1.\ \mathsf{public\_half}(g, x_1) * (\mathsf{public\_half}(g, x_1) \Rrightarrow P_p\ a * R')}{\begin{array}{c}\mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * \mathsf{my\_half}(g, sh, x_0) * R \Rrightarrow \\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * \mathsf{my\_half}(g, sh, x_0) * R'\end{array}}$$

A thread that holds $\mathsf{my\_half}(a)$ (i.e., is in the critical section of a lock with a lock variant) may interact with the abstract state P_p held in an atomic_shift, as long as the corresponding $\mathsf{public\_half}()$ is part of the abstract state. It may either leave both halves as is and gain updated resources R' (including, e.g., a snapshot of the current abstract state), or update both halves to a new state that satisfies the public postcondition Q_p, executing the linearization point and obtaining the final postcondition Q. Typically we will use one of these rules at the end of each critical section, to show that the changes made in the critical section either are invisible to other threads (sync_rollback) or have fulfilled the linearization point (sync_commit). We can also derive simpler rules for special cases, such as a version of sync_commit for read-only operations that do not change the state of any component:

$$\text{sync\_commit\_same}\ \frac{\forall a.\ R * P_p\ a \Rrightarrow \exists x_1.\ \mathsf{public\_half}(g, x_1) * (\mathsf{my\_half}(g, sh, x_0) * \mathsf{public\_half}(g, x_1) \Rrightarrow \exists y.\ Q_p\ a\ y * R'\ y)}{\mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * \mathsf{my\_half}(g, sh, x_0) * R \Rrightarrow \mathsf{EX}\ y,\ Q\ y * R'\ y}$$

3.2 Specification of the hand-over-hand BST

To verify our tree implementation, we need to instantiate bst_ref (the resources held by each client thread) and bst (the abstract state of the data structure), and show how they interact to access and update the state of the data structure for each call. In a lock-based implementation, bst_ref will be a pointer to the lock of the root node, and acquiring that lock will give the thread access to more of the data structure. As described above, 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. We also need to make sure that our lock invariants support the hand-over-hand pattern.

3.2.1 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. [2017], 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 3 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. At the leaves, these ranges partition the space of all possible keys. 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 3. 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.

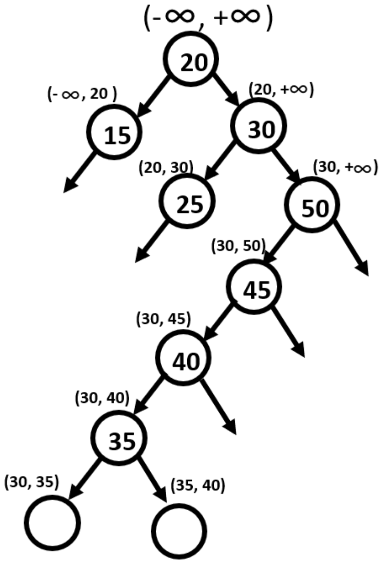


Fig. 3. An example of BST with each node labeled with *range*; lower and upper bound for a key

3.2.2 *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, the abstract state of a node in the tree consists of its range plus a representation of its contents: for a leaf node, None, and for an internal node, Some of its current key, value, and the identifiers of its left and right children. This per-node abstract state serves as a bridge between the per-node locks of the implementation and the abstract state of the entire tree (the bst predicate) in the specification.

The abstract state of a tree is a map from keys to values. The bst predicate then describes the conditions under which a map m is implemented by the ghost state of a tree. Roughly speaking, 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 public_half assertions—i.e., the authoritative copies of the per-node ghost states—needed to form an abstract tree:

$$\begin{aligned}&\mathsf{public\_halves}(t, g, (l, r)) \triangleq \mathsf{match}\ t\ \mathsf{with}\\ &\qquad \mid \mathsf{Leaf} \Rightarrow \mathsf{public\_half}_g\ ((l, r), \mathsf{None})\\ &\qquad \mid \mathsf{Node}\ (k, v, t_l, g_l, t_r, g_r) \Rightarrow \mathsf{public\_half}_g\ ((l, r), \mathsf{Some}\ (k, v, g_l, g_r)) * \mathsf{public\_halves}(t_l, g_l, (l, k)) * \mathsf{public\_halves}(t_r, g_r, (k, r))\end{aligned}$$

We recursively walk through the tree, starting from a node g with range (l, r), and collect the public_half 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 (l, k) and the right child has range (k, r).

This collection of public_half assertions can be combined with per-node lock variants to ensure that the nodes in an implementation actually form a valid tree. However, we also need to record the fact that each node in the implementation actually belongs to the tree t. We do this by augmenting public_halves with a piece of ghost state ghost_nodes(S) that records the set of all nodes in the tree, constructed so that each node can hold an in_tree g assertion guaranteeing that $g \in S$. Then our final definition of bst is:

$$\mathsf{bst}(m, g) \triangleq \exists t.\ t\ \text{implements}\ m \land \mathsf{public\_halves}(t, g, (-\infty, +\infty)) * \mathsf{ghost\_nodes}(\mathsf{node\_names}(t))$$

The range of the root node (named g) is $(-\infty, +\infty)$, and the ghost_nodes assertion guarantees that if any thread or lock holds an assertion in_tree g_i, then g_i is a node in the ghost tree t. Because we quantify over the abstract tree t, we can rearrange its nodes at any time, as long as it remains a well-formed tree with the same set of node names and key-value pairs. (We will take advantage of this property when verifying the `delete` function.)

As described in section 3.1, we connect the per-node locks of our C implementation with the global abstract state of bst by associating each node's lock with a lock variant assertion $\ell \boxdot\!\!\rightarrow_\pi R$, where R is parameterized by the abstract state of the node (range + possible contents). We will fix the share π to 1/2 so that the range in a node's lock can be out of date with respect to its public_half (in particular, it may have grown due to rearrangements further up the tree). We then need to choose a variant R describing the resources held by the lock for a given abstract state. The C implementation of a node 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;
```

The node data of each `tree` is wrapped in a `tree_t` struct, and is only accessible when the lock in the `lock` field is held. The lock variant for a node at location p then needs to describe the contents of the `t` field for each possible abstract state:

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

Because our implementation is doing hand-over-hand locking, in the internal case, we also need to obtain the lock variant assertions for the child nodes, so the full lock variant is:

$$\begin{array}{rl}
\text{node\_inv}((l, r), c) \triangleq & \text{match } c \text{ with} \\
& \mid \text{None} \Rightarrow \mathtt{t} \mapsto \mathtt{NULL} \\
& \mid \text{Some } (k, v, g_l, g_r) \Rightarrow k \in (l, r) \land \exists p_l, p_r.\ \mathtt{t} \mapsto (k, v, p_l, p_r) * \\
& \qquad p_l.\mathtt{lock} \boxdot\!\!\rightarrow^v \text{node\_inv} * p_r.\mathtt{lock} \boxdot\!\!\rightarrow^v \text{node\_inv}
\end{array}$$

While we do not know the ranges or contents of the child nodes, the lock-variant construction guarantees that they are consistent with the public_half assertions of the global ghost state.

Now we have everything we need to specify the tree operations:

$$\forall m.\ \langle \text{bst\_ref } p \mid \text{bst } m \rangle\ \mathtt{insert}(p, k, v)\ \langle \text{bst\_ref } p \mid \text{bst (insert } m\ k\ v) \rangle$$

$$\forall m.\ \langle \text{bst\_ref } p \mid \text{bst } m \rangle\ \mathtt{lookup}(p, k)\ \langle v.\ \text{bst\_ref } p \mid \text{bst } m \land \text{lookup } m\ k = v \rangle$$

$$\forall m.\ \langle \text{bst\_ref } p \mid \text{bst } m \rangle\ \mathtt{delete}(p, k)\ \langle \text{bst\_ref } p \mid \text{bst (delete } m\ k) \rangle$$

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

$$\text{bst\_ref}(b) \triangleq \exists p.\ b \mapsto p * p.\mathtt{lock} \boxdot\!\!\rightarrow^v \text{node\_inv}$$

Now we are prepared to prove that each of our BST functions satisfies its specification.

3.3 Proofs: Insert and Lookup

The basic verification style in VST is to divide proofs into two parts: proofs that the C implementations meet specifications that are Coq-level encodings of their behavior, and proofs that those encodings actually have the desired properties. In the single-threaded case, we might prove that some code implements a binary search tree written in Gallina, and then prove that that tree satisfies map axioms. Similarly, we divide our concurrency proofs into two parts: code-specific proofs that describe how the C code implements changes to per-node ghost state, and more abstract proofs describing how changes in local ghost state lead to the desired updates to the global state.

3.3.1 Insert. An outline of the C code for the insert function is shown in figure **??**. When a thread inserts a key k into a tree, it first crawls the tree to find the right position for k using the hand-over-hand locking mechanism, acquiring the lock for the next node before releasing the lock for the current node. If it encounters a node with key k, then it simply changes the value at that node. Otherwise, it will eventually reach a leaf node, where it creates two new empty leaf nodes, inserts the key and value into the current node, and links the newly created leaf nodes to the current node. The atomic specification for insert method is

$$\forall t.\ \langle \text{bst\_ref } p \mid \text{bst } t \rangle\ \mathtt{insert}(p, k, v)\ \langle \text{bst\_ref } p \mid \text{bst (insert } t\ k\ v) \rangle$$

The `insert` function should atomically add the key k with value v to the tree at p.

The key to the verification of the C code is the loop invariant for the top-level loop. We define the loop invariant for the insert function as follows:

$$\begin{array}{l}
\text{insert\_inv}(b, x, g, g\_root) \triangleq\ \exists\ lock, g\_current, np, range, info,\ (x \in range)\ \land\ \boxed{\text{my\_half}(range, info)}^{g\_current}\ *\ R\ np \\
*\ \text{lock\_inv}(lock, lsh2, R')\ *\ \text{bst\_ref}(b, g, g\_root)\ *\ \text{atomic\_shift}(P_p, E_i, E_o, Q_p, Q)
\end{array}$$

where P_p and Q_p are $\text{bst}(t, g)$ and $\text{bst}(t[k \mapsto v], g)$ respectively. The resources R, parameterized by a node pointer np, describes the concrete information about a node. Also, x is the key to be inserted that must be in the bound range. This loop invariant has some existential variables which characterize the state of each iteration of the loop.

Figure 6 shows the `insert` function with separation logic annotations. The proof starts with bst_ref and atomic_shift as the precondition. After acquiring the lock for the root node, the thread accesses the information inside the lock invariant of the root node. In order to prove while loop, we show that the precondition satisfies the insert_inv, then prove that the loop body preserves the loop invariant (cases inside elseif clauses at line 15 and 17 in **??**). Once we locate the empty node for inserting the new key-value pair (inside the first if clause), we open the global invariant encoded in the atomic_shift, create the ghost names g1 and g2 for two new empty child nodes of current node, and use the sync_commit rule—we have reached the linearization point and completed the insertion. The premise of sync_commit requires that we have enough information to know that inserting the key at this point will change the abstract state from some unknown t to $t[k \mapsto v]$, which we show in the second part of the proof.

The other branch of the code-level proof is the case where the key to be inserted already exists in the tree (the last `else` clause in the code). In this case, we change the value at the node but leave the rest of the tree intact, and this is the linearization point for the insertion. In the second part of the proof, we will show that this case also transforms t to $t[k \mapsto v]$.

In the other two cases, we compare the key of the current node with the new key `x` and move to the left or right child accordingly. Here we again need to access the atomic shift, to ensure that the range of the node we move to is included in the range of the current node. We can use sync_rollback to retrieve this information without making any changes to the abstract state of the tree.

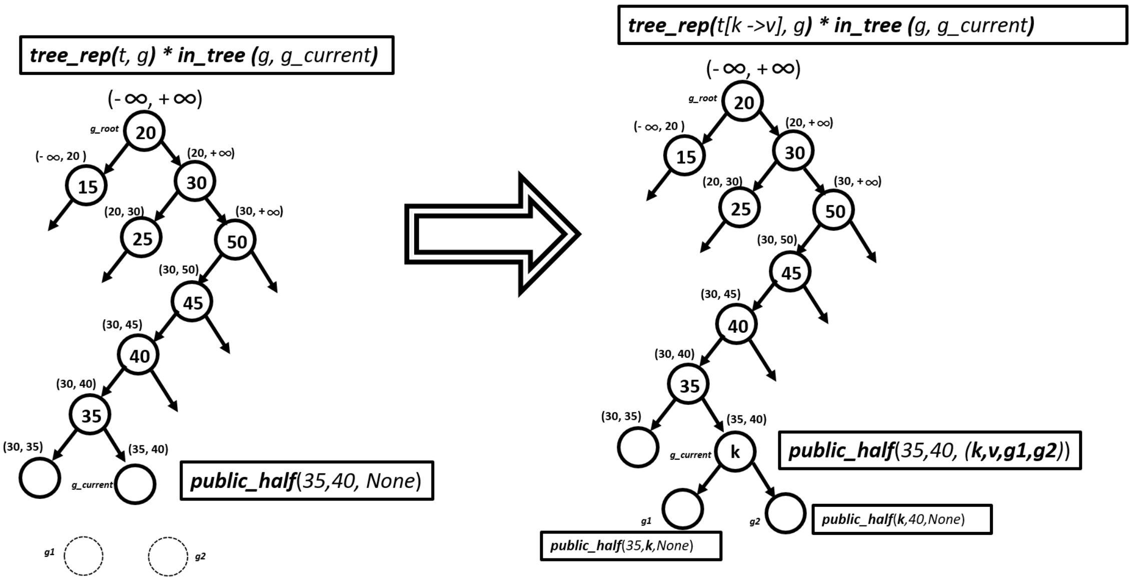


Fig. 4. A visual depiction of the change in global state of BST during insert(k, v) operation

Changing the Global State. describe "extract lemmas" here

3.3.2 *Lookup.*

The code for the lookup method is shown in Figure 7. It takes the location of the root pointer and a key as the arguments. A thread spans the tree to find a given key using hand-over-hand locking mechanism. Once a thread finds the key in the tree, it gets the value associated with that key, releases the current node's lock, and returns the value. The atomic specification for lookup is

$$\forall t.\ \langle \text{bst\_ref}\ p \mid \text{bst}\ t \rangle\ \texttt{lookup}(p, k)\ \langle v.\ \text{bst\_ref}\ p \mid \text{bst}\ t \wedge \text{lookup}\ t\ k = v \rangle$$

The `lookup` function should atomically look up the value of the key k in the tree at p.

As before, the key to the verification is the invariant for the main loop:

$$\text{lookup\_inv}(b, x, g\_root, range, info) \triangleq\ \exists\ lock,\ g\_current,\ np,\ (x \in range)\ \wedge\ \boxed{\text{my\_half}(range, info)}^{g\_current}$$
$$*\ R\ np\ *\ \text{lock\_inv}(lock, lsh2, R')\ *\ \text{bst\_ref}(b, g\_root)\ *\ \text{atomic\_shift}(P_p, E_i, E_o, Q_p, Q)$$

where P_p and Q_p are $\text{bst}(g\_root, t)$ and $(v = M[k])\ \wedge\ \text{bst}(g\_root, t)$ respectively. The proof steps for the `lookup` verification are similar to the steps used in `insert` verification, with a few differences which we discuss below.

Figure 7 shows the lookup function with separation logic annotations. The proof starts with bst_ref and atomic_shift assertions as the precondition and follows the same approach as the insert proof for the verification of loop body. Once we

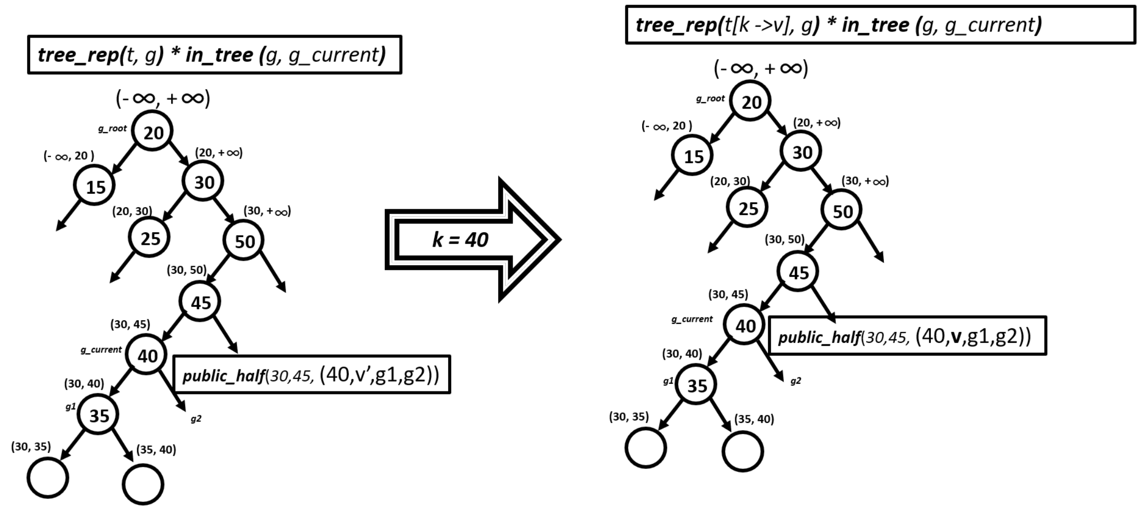

tree_rep(t, g) * in_tree (g, g_current)
(-∞, +∞)
g_root
20
(-∞, 20)
15
(20, +∞)
30
(20, 30)
25
(30, +∞)
50
(30, 50)
45
(30, 45)
g_current
40
public_half(30,45, (40,v',g1,g2))
(30, 40)
g1
35
g2
(30, 35)
(35, 40)
k = 40
tree_rep(t[k ->v], g) * in_tree (g, g_current)
public_half(30,45, (40,v,g1,g2))

Fig. 5. A visual depiction of the change in global state of BST during insert(k, v) operation, where key k already exists in the tree t

find the target key (the last else clause in the code), we need to confirm that the value present in the tree corresponds with the value we would find if we looked up the key in the current abstract state. We accomplish this by opening the global invariant encoded in the atomic_shift, and proving that lookup satisfies the public postcondition Q_p with the help of the sync_commit_same lemma described in section 3.1. This is the linearization point of the lookup operation and must be done before releasing the current node's lock. When we move into the left or right sub-tree, we need to establish lookup_inv at the end of the if or else if clause. Here, we need to show that the key we are searching for is still in the range of our new node, so we use sync_rollback to extract the bound of the left/right node encoded as the ghost state in the public precondition (bst(M, g, g_root)).

A critical difference from `insert` is that the `lookup` method does not change the shape of the tree. With the help of `sync_commit_same`, we need no longer the heavyweight *extract* lemma but instead a simpler *ramif* lemma to locate the current node in the global invariants:

```
Lemma ghost_tree_rep_public_half_ramif: forall tg g_root r_root g_in,
Ensembles.In (find_ghost_set tg g_root) g_in -> ghost_tree_rep tg g_root r_root |--
EX r: node_info, !! (range_info_in_tree r r_root tg) && (public_half g_in r *
(public_half g_in r -* ghost_tree_rep tg g_root r_root)).
```

3.4 Delete

The code for the delete method is shown in ??. It takes the location of the root pointer and a key as the arguments. A thread traverses the tree using hand-over-hand locking, until the thread finds the key to be deleted. To remove the key, the tree is changed through an operation called pushdown left such that the node to be deleted points to an empty leaf right child. To achieve that, the thread sets the right child's left child as the new right child of the to-be-deleted node and moves the right child's right child as the new parent of the to-be-deleted node. This operation is repeated until the right child of the to-be-deleted node is an empty leaf node. Then, the thread will delete the node and set the delete node's parent pointer to the left child of the deleted node.

The atomic specification for delete method can be written as follows:

$$\forall t.\ \langle \mathsf{bst\_ref}(p, g, g\_root) \mid \mathsf{bst}(t, g) \rangle$$

$$\texttt{delete}(p, k)$$

$$\langle \mathsf{bst\_ref}(p, g, g\_root) \mid \mathsf{bst}(\texttt{delete}(t, k), g) \rangle$$

The delete method is unique compared to insert and lookup because the pushdown left operation can change the *public* half range of more than one nodes. In addition, the pushdown left operation is one of the main reasons why the ghost tree of the binary search tree is not just a "copy" of the pure tree: the functional model of pushdown_left is unable to account for the change of deleted node's parent's pointer from pointing at the deleted node the to the deleted node's right child.

$$\left\{\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q)\ \right\}$$

```
insert(treebox t, int x, void *value)
  tree_t *tgt = *t;
  acquire(tgt->lock)
```

$$\left\{ \begin{array}{l} \boxed{\mathsf{my\_half}((-\infty,+\infty), info)}^{g\_root}\ *\ R\ b\ *\ \mathsf{lock\_inv}(lock, lsh2, R')\ * \\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array} \right\} \Rrightarrow \left\{\ insert\_inv\ \right\}$$

```
  for(;;) {
```

$$\left\{\ insert\_inv\ \right\} \triangleq \left\{ \begin{array}{l} (x \in range) \wedge \boxed{\mathsf{my\_half}(range, info)}^{g\_current}\ *\ R\ np \\ \mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array} \right\}$$

```
    p = tgt->t;
    if (p==NULL)
      tree_t *p1, *p2 = malloc();
      p = malloc(); tgt->t = p;
      p->key=x; p->value=value; p->left=p1; p->right=p2;
```

$$\left\{ \begin{array}{l} \boxed{\mathsf{own}\ (a1)}^{g1}\ *\ \boxed{\mathsf{own}\ (a2)}^{g2}\ *\ \boxed{\mathsf{my\_half}(range, None)}^{g\_current}\ *\ R\ np \\ \mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array} \right\} \Rrightarrow \textbf{sync\_commit}$$

$$\left\{\ \boxed{\mathsf{my\_half}(range, None)}^{g\_current}\ *\ R\ np\ * \mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{bst\_ref}(b, g\_root)\ *\ Q\ \right\}$$

```
      release2(l);
```

$$\left\{\ \mathsf{bst\_ref}(b, g\_root)\ *\ Q\ \right\}$$

```
      return;
     else
       ....
      else
       p->value=value;
```

$$\left\{ \begin{array}{l} \boxed{\mathsf{my\_half}(range, None)}^{g\_current}\ *\ R\ np\ * \mathsf{lock\_inv}(lock, lsh2, R')\ * \\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array} \right\} \Rrightarrow \textbf{sync\_commit}$$

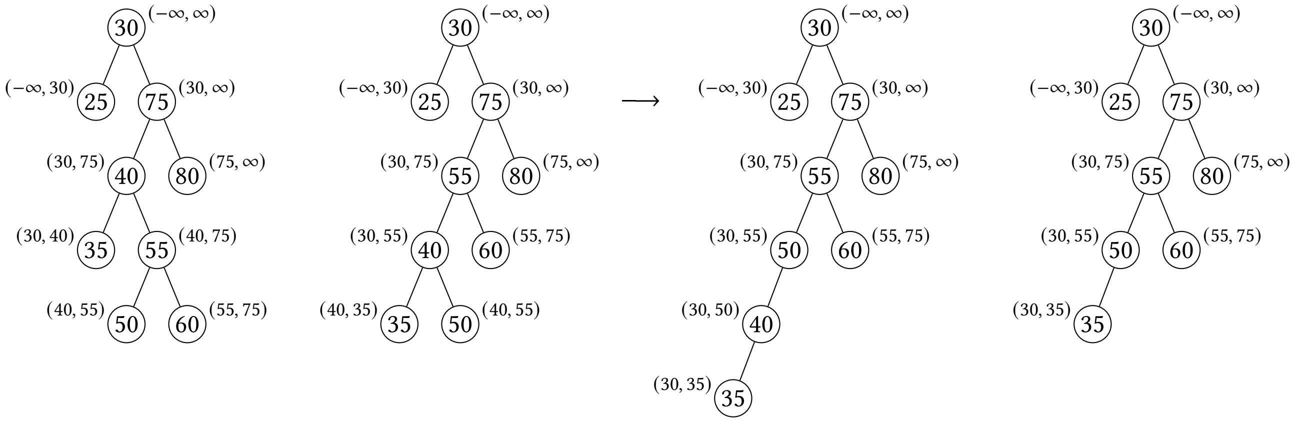
$$\left\{\ \boxed{\mathsf{my\_half}(range, None)}^{g\_current}\ *\ R\ np\ * \mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{bst\_ref}(b, g\_root)\ *\ Q\ \right\}$$

```
       release2(l);
```

$$\left\{\ \mathsf{bst\_ref}(b, g\_root)\ *\ Q\ \right\}$$

```
       return;
```

Fig. 6. The insert function annotated with separation logic specification



$$\left\{\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q)\ \right\}$$

```
void *lookup (treebox t, int x) {
  struct tree *p; void *v; struct tree_t *tgt;
  tgt = *t; void *l = tgt->lock;
  acquire(l); p = tgt->t;
```

$$\left\{\begin{array}{l} \boxed{\mathsf{my\_half}((-\infty, +\infty), info)}^{g\_root}\ *\ R\ b\ *\ \mathsf{lock\_inv}(l, lsh2, R')\ * \\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array}\right\} \Rrightarrow \left\{\ lookup\_inv\ \right\}$$

```
   while (p!=NULL) {
```

$$\left\{\ lookup\_inv\ \right\} \triangleq \left\{\begin{array}{l} (x \in range) \land \boxed{\mathsf{my\_half}(range, info)}^{g\_current}\ *\ R\ np\ * \\ \mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array}\right\}$$

```
   if (x<y){
     tgt=p->left;
     ....
   }else if (y<x){
     tgt=p->right;
    ....
   }else {
   v = p->value;
```

$$\left\{\begin{array}{l} \boxed{\mathsf{my\_half}(range, info)}^{g\_current}\ *\ R\ np\ * \mathsf{lock\_inv}(lock, lsh2, R')\ * \\ \mathsf{bst\_ref}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array}\right\} \Rrightarrow \textbf{sync\_commit\_same}$$

$$\left\{\ \boxed{\mathsf{my\_half}(range, info)}^{g\_current}\ *\ R\ np\ * \mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{bst\_ref}(b, g\_root)\ *\ Q\ \right\}$$

```
     release2(l);
```

$$\left\{\ \mathsf{bst\_ref}(b, g\_root)\ *\ Q\ \right\}$$

```
      return v;} }
  release2(l); return NULL; }
```

Fig. 7. The lookup function annotated with separation logic specification

3.4.1 delete. Similar with insert, delete starts by acquiring the root node lock, so that the thread can access the information inside the lock invariant of the root node. Then, we prove the while loop by showing that the precondition satisfies delete_inv and the loop body preserves the loop invariant. The first two cases are similar to insert and lookup in which the thread finds the target node, traversing by hand-over-hand locking. Once the target node is located, the thread will call pushdown_left and return after. Using the specification of pushdown_left, we show that the post-condition of pushdown_left implies the loop invariant. Note that before going into pushdown_left, the thread is holding the lock to the target node and will return to the delete method without holding any.

3.4.2 pushdown_left. The proof to pushdown_left is the key step of proving the delete method. First, we show that the precondition of pushdown_left satisfies the precondition of the loop invariant, which includes the assertion that the thread is holding the lock to the deleted node (i.e. the information inside the lock invariant is accessible) and an atomic_shift. Then, the thread accesses that information to acquire a second lock to the right child of the deleted node. Due to the hand-over-hand locking mechanism this will not cause a deadlock since the thread has already acquired the lock to the deleted node. Next, we prove that the action of deleting the node, with sync_commit by opening the global invariant and removing the relevant ghost names (the target node and its right sentinel node), satisfies the loop invariant and the postcondition of pushdown_left.

The second part of pushdown_left is proving that the turn_left operation satisfies the loop invariant. Using a variant of the sync_rollback rule, we show that the turn_left operation preserves the BST, apart from changes in the ranges of two nodes, the target node and its right child, and information about the nodes in those two nodes. To satisfy the structure of the ranges, we only need to prove that the new range for the target node is encompasses the old range of both nodes. This will imply that the range of the target node's old parent includes the target node's right child new range as the right child becomes the new parent (and the old parent points to the right child).

One key step of the proof is proving that two ghost trees are equivalent if the union of its ghost nodes are equal and the ghost tree is sorted. This allows the global invariant to be preserved during the turn_left operation in which no nodes were actually changed except for the structure of the tree.

4 LOCK-FREE BST

We want to prove that a lock-free BST implementation satisfies the same specification as our hand-over-hand implementation. Unfortunately, provably-correct deletion in a lock-free setting is a research topic in itself (cite?), so we begin with a lock-free BST that only supports insert and lookup. Once again, we want to prove

$$\forall t.\ \langle \text{bst\_ref}\ p \mid \text{bst}\ t \rangle\ \texttt{insert}(p, k, v)\ \langle \text{bst\_ref}\ p \mid \text{bst}\ (\text{insert}\ t\ k\ v) \rangle$$

$$\forall t.\ \langle \text{bst\_ref}\ p \mid \text{bst}\ t \rangle\ \texttt{lookup}(p, k)\ \langle v.\ \text{bst\_ref}\ p \mid \text{bst}\ t \land \text{lookup}\ t\ k = v \rangle$$

though the precise definitions of bst_ref and bst will differ.

5 USING THE SPECIFICATIONS

client and client proofs

6 RELATED WORK

Our work is based on the concurrent separation logic of VST in the style of Iris [Jung et al. 2016] and the logical atomicity from TaDA logic [da Rocha Pinto et al. 2014]. The notion of the *lower* and *upper* bound is taken from the flow interface paper [Krishna et al. 2017]. Our main technical contributions are the verification of two different implementation of binary search tree (using fine-grained locking and lock-free technique) with respect to the same abstract specification, and first c-level mechanized verification of concurrent search-based data structure (i.e. BST). Most of the other concurrent separation logic (CSL)-based verification works are on toy languages, while VST lets us use the same logic on real C programs.

Gotsman and Yang [Gotsman et al. 2007] is one of the earliest work on concurrent separation logic. They introduced the program logic to reason locally about the heap-manipulating program with the notion of dynamic ownership of heap parts by unbounded number of locks and threads, and shown the verification of singly-linked list with fine-grained locking. Xiong et al. [Xiong et al. 2017] have demonstrated the verification of ConcurrentSkipListMap from java.util.concurrent library using the recent advances in fine-grained concurrency reasoning. Their work is mainly based on the abstract atomicity from TaDa logic, and give two modular specifications for concurrent maps: one specification focus on the entire map structure which is suitable for verifying implementation, and another specification focus on the key-value pairs which appropriate for verifying clients. We use the same idea of atomicity (though implemented in Iris) in our work.

Krishna et al. [Krishna et al. 2017] have presented the proof technique for concurrent search structure templates based on the flow framework and the Iris separation logic, and verified the implementation of concurrent B-tree, hash tables, and linked lists based on the templates. Their work is closely related to our work; we took some inspiration from them in building our ghost state. But, their proofs are on a toy language rather than real C code.

7 CONCLUSION

REFERENCES

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, USA.

R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Inf.* 9, 1 (March 1977), 1–21. https://doi.org/10.1007/BF00263762

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–37.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–367.

Ralf Jung. 2020. personal communication.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. 51, 9 (2016). https://doi.org/10.1145/3022670.2951943

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. 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.* ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2017. Go with the Flow: Compositional Abstractions for Concurrent Data Structures (Extended Version). arXiv:1711.03272 [cs.LO]

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032).* Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *Proceedings of the 26th European Symposium on Programming (ESOP'17) (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 964–990. https://doi.org/10.1007/978-3-662-54434-1_36