Verifying a Binary Search Tree with Fine-Grained Locking

WILLIAM MANSKY

ACM Reference Format:
William Mansky. 2017. Verifying a Binary Search Tree with Fine-Grained Locking . *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (October 2017), 6 pages. https://doi.org/10.1145/3133911

In this section, we will focus on safety proofs of concurrent binary search tree. Here safety means thread-safe. An operation of any concurrent data-structure is thread-safe if it functions correctly during simultaneous execution by multiple threads. In VST, we can use lock_inv assertion to assert that there exists a lock in memory with a given invariant: lock_inv sh p R means that the current thread owns share sh of a lock at location p with invariant R. We use the term *lock invariant* for R, which is a predicate representing the resources protected by a lock. The lock_inv predicate is sufficient to prove the safety properties of the concurrent data-structure. We have taken binary search tree with fine-grained concurrent control implemented using hand-over-hand locking mechanism. In hand-over-hand locking mechanism, we acquire the lock for a successor before releasing the lock for a predecessor. The code for `insert` and `lookup` methods of concurrent binary search tree that we are interested to verify is presented in Figure 1.

3.1 Lock invariants for hand-over-hand locking

A node of CBST have the pointer to its children, so lock invariant of a node should have the knowledge about its children. Whenever we release the lock for the parent node, we have to recover all resources that the parent node accessed while acquiring the lock. By releasing the parent node's lock, we would lose the information about the lock_inv of the node and we wouldn't be able to release the node's lock later. To solve this problem, we need to use *recursive* lock, one whose invariant includes a share of lock itself. In VST, we can make such invariant using selflock function along with the lemma selflock_eq: $\forall Q\ sh\ p, \mathsf{selflock}\ Q\ sh\ p = Q * \triangleright \mathsf{lock\_inv}\ sh\ p\ (\mathsf{selflock}\ Q\ sh\ p)$. We can define *lock invariants* and lock_inv predicate for *recursive* lock *lock* as follows: lock_inv $lsh1\ p$ (selflock $P\ lsh2\ lock$) where P represents the knowledge about the children of current node, $lsh1$ and $lsh2$ are the two halves of the writable share Ews.

3.2 Specification and Verification

4 CORRECTNESS PROOFS

In the previous section, we showed that how to prove concurrent data-structure is thread safe, but not functionally correct. To prove the correctness, our threads need to be able to record the information about the actions they have performed on the shared state, instead of sealing all knowledge of the shared data structure inside the lock invariant. We can accomplish this with ghost variables, a simple form of auxiliary state. Aside from the `permission` on shared resources held by each thread that we used in the safety proofs, the verification of correctness properties also involves the `ghost state`. In VST, any Coq type can be used as ghost state, as long as we can describe what happens when two elements of that type are joined together. We can use own predicate in VST to represent ghost state assertion as follows: own $g\ a\ pp$, where g is a ghost name, a is the value associated with g which can be of any type, and pp is a separation logic predicate.

4.1 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. This is often implemented with either a lock-free series of atomic memory accesses, in which case all but the last access must make changes that are considered "invisible", or a coarse-grained lock, in which case any

Author's address: William Mansky.

2475-1421/2017/10-ART1
https://doi.org/10.1145/3133911

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

void insert (treebox t, int x, void *value) {
  struct tree_t *tgt = *t;
  struct tree *p;
  void *l = tgt>lock;
  acquire(l);
  for(;;) {
    p = tgt->t;
    if (p==NULL) {
      tree_t *p1 = (struct tree_t *) surely_malloc (sizeof *tgt);
      tree_t *p2 = (struct tree_t *) surely_malloc (sizeof *tgt);
      p1 ->t = NULL;
      p2 ->t = NULL;
      lock_t *l1 = (lock_t *) surely_malloc(sizeof(lock_t));
      makelock(l1);
      p1->lock = l1;
      release2(l1);
      lock_t *l2 = (lock_t *) surely_malloc(sizeof(lock_t));
      makelock(l2);
      p2->lock = l2;
      release2(l2);
      p = (struct tree *) surely_malloc (sizeof *p);
      tgt->t = p;
      p->key=x; p->value=value; p->left=p1; p->right=p2;
      release2(l);
      return;
    } else {
      int y = p->key;
      if (x<y){
        tgt = p->left;
        void *l_old = l;
        l = tgt->lock;
        acquire(l);
        release2(l_old);
      } else if (y<x){
        tgt = p->right;
        void *l_old = l;
        l = tgt->lock;
        acquire(l);
        release2(l_old);
      }else {
        p->value=value;
        release2(l);
        return;
      }
    }
  }
}
```

Fig. 1. Insert Method

```
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;
  while (p!=NULL) {
    int y = p->key;
    if (x<y){
      tgt=p->left;
      void *l_old = l;
      l = tgt->lock;
      acquire(l);
      p=tgt->t;
      release2(l_old);
    }else if (y<x){
      tgt=p->right;
      void *l_old = l;
      l = tgt->lock;
      acquire(l);
      p=tgt->t;
      release2(l_old);
    }else {
      v = p->value;
      release2(l);
      return v;
    }
  }
  release2(l);
  return NULL;
}
```

Fig. 2. Lookup Method

changes may be made and no other thread can access the data structure until the operation is complete. Fine-grained locking presents an interesting middle ground between these two approaches: other threads may continue to access the data structure as long as they do not require a section that is currently locked, and the visible changes may be implemented with multiple non-atomic operations in the locked section.

In an Iris-style separation logic, the lock-free approach means that each atomic operation gets access to the state of the data structure stored in an atomic shift, and must either maintain its previous state or (at the linearization point) move to a new one; in the coarse-grained approach we can store the state of the data structure in a lock invariant, and the function has access to all of it throughout the critical section. With fine-grained locking, neither of these approaches quite applies as-is: there are no atomic operations by which we access the state of the data structure, but neither is there a lock whose invariant controls access to the entire structure. However, we can take a modified form of the lock-free approach, where we treat each critical section as an atomic operation that can access the atomic shift. Typically, we will perform a sequence of non-atomic operations in the body of the critical section without access to the global state of the data structure, and then (before exiting the critical section) access the atomic shift and show that we have either maintained the original state or satisfied the postcondition of the operation. In this section, we describe a general approach to associating fine-grained locks with pieces of the abstract state of a data structure, allowing them to interact with atomic shifts and be used to prove atomic specifications. (Note that this approach is not novel to us; it is adapted from something Ralf Jung worked on. We need to be more clear about the relationship.)

technical material: definition of sync_inv, sync_commit and sync_rollback

4.2 Insert and Lookup

In this section, we will verify an atomic specification for insert and lookup method of Binary Search Tree with fine-grained concurrency. In order to write the atomic specification, we need to define a predicate that describes the state of the data structure, both concrete and abstract. We know that abstract state of data structure corresponds to the public pre- and postcondition of an atomic specification and concrete information of the node that a thread needs to access the data structure corresponds to the private pre- and postcondition of an atomic specification. For instance, an atomic specification for any function might look like as follows:

$$\forall t.\ \langle \mathsf{nodebox\_rep}(p, g\_root) \mid \mathsf{tree\_rep}(g\_root, t) \rangle\ \mathtt{cbst\_op}(p)\ \langle \mathsf{nodebox\_rep}(p, g\_root) \mid \mathsf{tree\_rep}(g\_root, t') \rangle$$

where p and g_root are the pointer and *gname* representing the root node of a tree, and t and t' are the abstract state of a tree before and after the function's execution. The *nodebox_rep* is a predicate describing the local information about the node almost similar to the predicate we defined for the safety proof in section 3. In this case, the lock invariant of recursive lock have, along with the information about the children node,my_half predicate which holds the partial information describing the current thread's contribution to the shared state (i.e. *tree_rep*). The *tree_rep* predicate describes the abstract state of the data structure in terms of collection of ghost states each describing the information about separate node in the tree.

4.2.1 Ghost States and Range. The main challenge in proving the properties of concurrent data structure is to abstract the data structure which helps to reason about it in the concurrent setting. According to the flow interface paper, we can use the product of lower-bound and upper-bound on each node of binary tree by propagating from each node to its child the appropriate bounds on the values presents in the sub-tree. We use the term *range* to represent the product of lower- and upper-bound. The range for root node is always $(-\infty, +\infty)$ and we can calculate the range for all nodes in the tree by enforcing each node to propagates appropriate bounds. For the left child, range will be lower-bound for the node(lower-bound) and a key in the node (upper-bound), while for the right child range will be a key in the node (lower-bound) and upper-bound for the node (upper-bound). This properties in each node help us to prove that the thread have changed the state of data-structure to the valid state. For instance, we confirm that a key-value have been inserted in the right place if a key is inside the range of the node.

We introduce the *range* in our proof using the ghost variables. This *ranges* along with other information about the node represent the abstract state of the data-structure. We need another type of ghost state to track the collection of all these per-node ghost states. So, we have two kind of ghost state in our proof: one is per-node ghost state which product of range and node information and another is over-all ghost states which represent the set of all per-node ghost variable. Both type of ghost states follow the `reference` pattern in which each thread holds partial information describing its contribution to the shared state, and the shared resource holds a "reference" copy that records all of the contributions.

For per-node ghost state, we first define the range ghost instance as follows:

```
Program Instance range_ghost : Ghost :=
  { G := (number*number); valid g := True; Join_G a b c := c =  merge_range a b }.
```

where number $\subsetneq (-\infty, +\infty)$ and *merge_range* function merges two ranges into one. We extend this range information with the other information about the node: key, value and *gname* for its child. We define our total ghost states as follows:

```
Definition ghost_info : Type := (key * val * gname * gname)%type.
Instance node_ghost : Ghost := prod_PCM range_ghost (exclusive_PCM (option ghost_info)).
```

Now we can create the contribution part and reference part discussed above using `my_half` and `public_half` function in VST. `my_half` predicate will be used in the private pre- and post-condition of atomic specification while `public_half` will be used in the public pre- and post-condition of atomic specification.

We define the contribution and reference part of the over-all ghost state as follows:

```
Definition ghost_ref g r1 := ghost_reference(P := set_PCM) r1 g.
Definition in_tree g r1 := EX sh: share, ghost_part(P := set_PCM) sh (Ensembles.Singleton r1) g.
```

We use *ghost_ref* and *in_tree* in the public and private part of the atomic specification respectively. The global invariant, which we introduced in our atomic specification as *tree_rep* predicate above, ties together `public_half` and `ghost_ref` as follows:

$$\mathsf{tree\_rep}(T, g) \triangleq [\ \forall n.\mathsf{public\_half}_{\mathrm{gn}}\ (range_n * ghost\_info_n)\ ] * \mathsf{ghost\_ref}\ g\ \{gn\ \}$$

Similarly, `nodebox_rep` predicate ties together `my_half` and `in_tree` inside the `lock_inv` assertion of each node's lock as follows:

$$\mathsf{nodebox\_rep}(p, g, g\_root) \triangleq p \mapsto\ (lock, tp) * \mathsf{lock\_inv}_{\mathrm{lock}}(\mathsf{my\_half}_{\mathrm{g\_root}} *\ \mathsf{in\_tree}\ g\ g\_root\ *\ \exists\ pa,\ pb,\ ga,\ gb,\ tp \mapsto\ (pa * pb)\ *\ \mathsf{nodebox\_rep}(pa, g, ga)\ *\ \mathsf{nodebox\_rep}(pb, g, gb))$$

4.2.2 insert. The code for the insert method of our concurrent binary search tree is shown in Figure 1. A node has a *lock*, key, value, and pointers to the left and right child. Each leaf node in tree are empty node with lock. Whenever a thread try to insert key-value pair in the tree, it first spans the tree to find the right position for new key using hand-over-hand locking mechanism; acquire the lock for child before releasing the lock for node. After locating right leaf node to insert new key-value, thread creates two new empty leaf nodes, insert key-value in the current node, and link the newly created leaf nodes to the current node. If the key already exist in the tree, then thread simply swaps the old value associated with node with the new value keeping the child pointers as it is.

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

$$\forall t.\ \langle \mathsf{nodebox\_rep}(p, g\_root) \mid \mathsf{tree\_rep}(g\_root, t) \rangle$$
$$\texttt{insert}(p, k, v)$$
$$\langle \mathsf{nodebox\_rep}(p, g\_root) \mid \mathsf{tree\_rep}(g\_root, t[k \mapsto v]) \rangle$$

We are not guaranteed that the state of a tree at the end of the insertion will be $t[k \mapsto v]$ where t is the tree state when insert is called; rather, we know that p always holds some tree during the function's execution, and at some point the function will take that tree t, add $[k \mapsto v]$,and then eventually return while in the meantime another thread may have modified the state of tree from $t[k \mapsto v]$ to any other arbitrary state that the current thread do not know. Since we have given the specification which is strong enough to specify both safety and correctness properties of insert method, we can now verify that body of the function satisfy that specification using various VST and Iris (encoded in VST) tactics. To prove the loop inside insert method, we need loop-invariant as follows:

$$\mathsf{insert\_inv}(b, x, g\_root, range, info) \triangleq \exists\ lock,\ g\_current,\ np,\ !!(x \in range)\ \&\&\ \mathsf{my\_half}(g\_current, range, info)\ *\ R\ np\ *$$
$$\mathsf{lock\_inv}(lock, lsh2, R')\ *\ \mathsf{nodebox\_rep}(b, g\_root)\ *\ \mathsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q)$$

Where P_p and Q_p are $\mathsf{tree\_rep}(g\_root, t)$ and $\mathsf{tree\_rep}(g\_root, t[k \mapsto v])$ respectively. This loop invariant has some existential variables which characterized the state of each iteration of loop in the C code. We use $forward\_while$ tactic just before the loop with above *insert_inv* which leaves four subgoals. The first subgoal is to prove that the current precondition in our proof satisfies the insert invariant, the second subgoal is about type-checking the loop-test expression which is often solve automatically by *entailer*! tactic, the third subgoal is to prove that the loop body preserves the loop invariant for that we must forward-symbolic-execute through the loop body, and final subgoal is to prove that the post condition of the loop body entails the insert invariant.

Figure 3 shows the *insert* function annotated with separation logic specification.

4.2.3 lookup.

4.3 Delete

5 RELATED WORK

6 CONCLUSION

$$\left\{ \text{nodebox\_rep}(b, g\_root) \ * \ \text{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \right\}$$

```
void insert (treebox t, int x, void *value) {
  struct tree_t *tgt = *t;
  ....
  acquire(l);
```

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

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

$$\left\{ \ insert\_inv \ \right\} \triangleq \left\{ \begin{array}{l} !!(x \in range) \&\& \text{my\_half}(g\_current, range, info) * \ R\ np \ * \text{lock\_inv}(lock, lsh2, R') * \\ \text{nodebox\_rep}(b, g\_root) \ * \ \text{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array} \right\}$$

```
    p = tgt->t;
    if (p==NULL) {
      tree_t *p1 = (struct tree_t *) surely_malloc (sizeof *tgt);
      tree_t *p2 = (struct tree_t *) surely_malloc (sizeof *tgt);
      ....
      p = (struct tree *) surely_malloc (sizeof *p);
      tgt->t = p;
      p->key=x; p->value=value; p->left=p1; p->right=p2;
```

$$\left\{ \begin{array}{l} \text{own } g1\ a1\ pp \ * \ \text{own } g2\ a2\ pp \ * \ \text{my\_half}(g\_current, range, None) * \ R\ np \ * \text{lock\_inv}(lock, lsh2, R') * \\ \text{nodebox\_rep}(b, g\_root) \ * \ \text{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \end{array} \right\}$$
$$\Downarrow \textbf{sync\_commit}$$
$$\left\{ \ \text{my\_half}(g\_current, range, None) * \ R\ np \ * \text{lock\_inv}(lock, lsh2, R') \ * \ \text{nodebox\_rep}(b, g\_root) \ * \ Q \ \right\}$$

```
      release2(l);
```

$$\left\{ \ \text{nodebox\_rep}(b, g\_root) \ * \ Q \ \right\}$$

```
      return;
    } else {
      int y = p->key;
      if (x<y){
        ....
      } else if (y<x){
        ....
      }else {
        p->value=value;
```

$$\left\{ \ \text{my\_half}(g\_current, range, None) * \ R\ np \ * \text{lock\_inv}(lock, lsh2, R') \ * \ \text{nodebox\_rep}(b, g\_root) \ * \ \text{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) \ \right\}$$
$$\Downarrow \textbf{sync\_commit}$$
$$\left\{ \ \text{my\_half}(g\_current, range, None) * \ R\ np \ * \text{lock\_inv}(lock, lsh2, R') \ * \ \text{nodebox\_rep}(b, g\_root) \ * \ Q \ \right\}$$

```
        release2(l);
```

$$\left\{ \ \text{nodebox\_rep}(b, g\_root) \ * \ Q \ \right\}$$

```
        return;
      }
    }
  }

}
```

Fig. 3. The `insert` function annotated with separation logic specification