Verifying a Binary Search Tree with Fine-Grained Locking

William Mansky

1 Introduction

2 Background

2.1 VST and Iris

2.2 Atomic Specifications

3 Safety Proofs

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 CBST is safe, but not it is 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.

Linearizability is one of the most popular strong correctness condition for the most of concurrent data-structure, which implies that every operation appears to take place atomically, in some order, consistent with the real-time ordering of those operations.

4.1 Locking and atomicity

In order to record the action each thread performed as a linear history, we will define global invariants using the ghost state, which are similar to lock invariants but are not associated with any particular memory location. Instead, a global invariant is true before and after every step of a program, acting as a publicly accessible resource. A program instruction can use the contents of global invariant if it can guarantee that no one will ever see an intermediate state in which the invariant does not hold.

4.2 Insert and Lookup

4.3 Delete

5 Related Work

6 Conclusion

```
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;
      }
    }
  }
}
void *lookup (treebox t, int x) {
  struct tree *p; void *v;
  struct tree_t *tgt;
  tgt = *t;
```