Verifying Concurrent Data Structures Using Data-Expansion Technical Report

Tong Che[*] Rachid Guerraoui[†]

Abstract

We present the first thread modular proof of a highly concurrent binary search tree. This proof tackles the problem of reasoning about complicated thread interferences using only thread modular invariants. The key tool in this proof is the Data-Expansion Lemma, a novel lemma that allows us to reason about search operations in any given state. We highlight the power of this lemma when combined with our generalized version of the classical Hindsight Lemma, which enables us to prove linearizability by reasoning about the temporal properties of the operations instead of reasoning about the linearizability points directly.

The Data-Expansion Lemma provides an interesting solution to the proof blowup problem when reasoning about concurrent data structures by separating the verification of effectful and effectless operations. We show that our proof methodology is widely applicable to several published algorithms and argue that many advanced highly concurrent data structures can be surprisingly easy to verify using thread-modular arguments.

[*]École Polytechnique Fédérale de Lausanne, Switzerland, tong.che@epfl.ch

[†]École Polytechnique Fédérale de Lausanne, Switzerland, rachid.guerraoui@epfl.ch

1 Introduction

Highly concurrent algorithms are extremely hard to design and verify. On one hand, the vast number of interference possibilities makes formal proof impractical and human proof error-prone. On the other hand, thread modular proofs are usually impossible dreams, even for very simple algorithms. Hence, the verification of concurrent algorithms is a major research challenge and an important step to boost the reliability of concurrent programming.

Sophisticated concurrent objects, such as concurrent binary search trees, are becoming popular and are promising to replace traditional search structures , for example: skip lists. However, because of their complexity, many of these algorithms are published without rigorous mathematical proofs, not to mention formal ones. Meanwhile, the verification community spends most of its efforts on relatively simple data structures, such as linked lists and stacks.

This paper presents a surprisingly simple proof strategy for the linearizability of advanced concurrent algorithms, which is purely thread modular. Our proof strategy covers a number of algorithms, but in this paper, we focus on one simple example for concreteness — an external binary tree without rebalancing. This algorithm is simple but powerful, because many concurrent algorithms [4, 9] use similar mechanisms.

For the verification of highly complicated advanced algorithms, thread modular proofs seem to be the only feasible solution. Because of their complexity, the linearizability points of such algorithms are in many cases non-fixed. In traditional methods [6, 11], reasoning about such linearizability points was done by tracking the set of pending invocations and auxiliary states, which lead to non-local proofs. In such proofs, one has to construct the set of linearizability points before reasoning about the data abstraction. However, these methods are not adapted to advanced data structures, because the behavior of the pending calls are highly complicated, and it is hard, if not impossible, to avoid proof blowup.

A first purely thread modular proof of a simple linked list algorithm with non-fixed linearizability point was presented in [10]. It was shown that for linked lists, reasoning about invariants of tiny steps of every thread can lead to important mathematical conclusions, such as the Hindsight Lemma [10], and finally to proofs. In this work, our main purpose is to argue that this idea is actually widely applicable to many advanced data structures, some of which were previously considered too complicated for rigorous or formal proofs.

Our proof strategy proceeds as follows: First, we identify a set of thread local invariants preserved by every computation step of each thread. Then we prove that a small subset of these invariants implies our Generalized Hindsight Lemma as well as a new lemma: the Data-Expansion Lemma. Each lemma captures a specific aspect of the reason why the tree traversal works in both cases no matter whether the traversal encounters its target or not. At last, we prove the operations are linearizable using abstraction functions. The two lemmas give us direct explanations of the non-fixed linearizability points, avoiding thereby the use of extra auxiliary states.

The Data-Expansion Lemma is the main technical contributions of this work. It allows us to infer the nonexistence of a key in some past state when the tree traversal failed to encounter it without explicit construction of the linearizability points. This lemma can be applied to at least three main kinds of search data structures: *linked lists*, *skip lists*, and *search trees*. Combined with our generalized version of the Hindsight Lemma, it provides powerful tools to reason about operations with non-fixed linearizability points in advanced concurrent data structures.

The rest of this paper is organized as follows: In Section 2, we briefly explain our verification strategy. In Section 3, we present the verification target, a highly concurrent binary search tree algorithm. In Section 4, we introduce our computation models. In Section 5, we present our verification. In Section 6, we prove the most generalized version of Hindsight Lemma, and discuss its applications. In section 7, we discuss some possible extensions to other highly concurrent objects. In Appendix A and Appendix B, we formalize the programming language and the Generalized Hindsight Lemma. In Appendix C we give some definitions. We put the proof of the invariants in Appendix D.

2 Verification Overview

We describe our verification strategy intuitively, using the example of a concurrent set algorithm implemented with an optimistic external binary search tree. An external binary search tree is a variant of ordinary BST. Its keys are stored only in leaf nodes, and the internal nodes are used for routing. We further assume that for all nodes u, v, w, where u is an ancestor of v, w and v/w is located in the left/right subtree of u, then we have $v.key \leq u.key < w.key$.

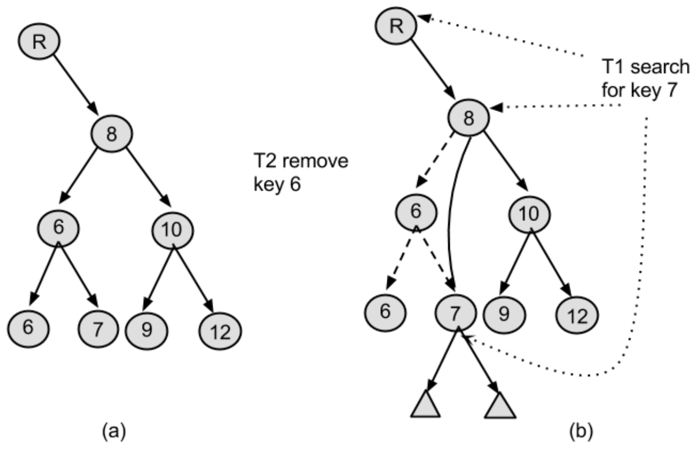

Figure 1: Concurrent BST

Heap Representation. The shared data of the threads underlying the set algorithm is an external binary search tree composed of dynamic allocated nodes of two types, leaf and internal, which we refer to as *heap*. Each internal node contains three fields, two pointer fields `child(1)`, `child(2)` pointing to its left and right children, and an integer field `key` storing the key of this node. Each leaf node contains only an integer field `key`. The `Root` node contains the key $-\infty$. For each state, some portion of the heap is *reachable* by following a sequence of child pointers from `Root`. We refer to this portion of the heap as *reachable heap*.

We view a computation of the algorithm as a sequence of shared program states. In each state, each leaf node in the reachable heap corresponds to a key in the set. The unreachable portion of the heap contains removed nodes.

Set Operations. There are three set operations, `add`, `remove`, and `contains`. Intuitively, they correspond to operations that add, remove or search for a key in a sequential binary search tree. All these operations need to traverse the tree first.

Generalized Hindsight Lemma. We use the example of the `contains` operation for illustration. We assume several threads are running the set algorithm. One of them is a `contains` operation, looking for key k in the binary search tree. If the operation reaches a leaf node with key k, can the operation return and claim that the set contains a node with key k at some linearizability point? We can separate two cases here:

- The leaf node is currently on the tree.
- The leaf node is removed from the tree in current state and is not in the reachable heap.

The first case is trivial. In the second case, the correctness (linearizability) of the operation is guaranteed by the Generalized Hindsight Lemma. Basically, this lemma claims the following:

If `add` and `remove` operations preserves certain simple thread modular invariants when modifying the data structure, each pointer link through which the `contains` operation has traversed was on the tree in some past state between the invocation and return.

The above lemma is not enough for our verification. A question remains open: if the operation reaches a leaf node with key $k' \neq k$, can the operation return and claim that the set does not contain a node with key k at some linearizability point? This is the question addressed by the Data-Expansion Lemma.

Static Bound. Given a state σ in a computation of the algorithm, for any node u on the tree, the range of keys which can be inserted to the subtree rooted at u is determined by the keys of the ancestors of u. This range is called the *static bound* at u in state σ. For example, in Figure 1(a), the static bound of the internal node with key 6 is $(-\infty, 8]$, because 6 is on the left side of 8.

Data-Expansion Lemma. The intuition behind our Data-Expansion Lemma is that a tree traversal should not miss the target node on the tree in the presence of thread interference. We assume a tree traversal targeting at key k arrives at an internal node u at state σ. If the leaf node with key k is on the tree but not on the subtree rooted at u, then the traversal would miss it.

Our Data-Expansion Lemma states that this will never happen if certain thread modular invariants are preserved. Namely, key k lies in the static bound of node u at state σ if some invariants are preserved by `add` and `remove` operations when they modify the heap. For example, in Figure 1(b), thread T_1 is searching for node with key 7, while T_2 is concurrently removing the node with key 6. If T_1 reaches the internal node with key 6 before the removal of T_2, the static bound of the node with key 7 is $(6, 8]$, after the removal the static bound of the node with key 7 is $(-\infty, 8]$. So the target key 7 is contained in the static bound in the presence of thread interference.

Verification of Linearizability. Our verification is a combination of formal proofs of the thread modular invariants and rigorous mathematical arguments, such as the Data-Expansion Lemma and Generalized Hindsight Lemma.

We treat two kinds of operations separately. *Effectful* operations are operations which successfully modify the heap. *Effectless* operations are read-only to the shared heap. Effectless operations do not have fixed linearizability points, so reasoning about their linearizability points using auxiliary states in such a complicated logic brings severe proof blowup. We will use our two lemmas to deal with two aspects of effectless operations, no matter the traversal encounters its target or not.

3 Verification Target

Our verification target is Listing 1. The algorithm implements a concurrent dictionary using binary search trees. It is essentially similar to [4], but is simpler, since this algorithm excludes mechanisms in [4] to achieve lock-freedom using only *CAS*. This algorithm can be viewed as a "template" implementation of concurrent binary search trees. In fact, many tree algorithms use similar mechanisms to achieve concurrency. The algorithm is optimistic and highly concurrent, for its atomic sections access only a very small portion (three nodes) of the data structure.

```
Node*,Node*,Node* search(KeyType k){
  Node *n := Root;
  while (n is not leaf node) {
    dir = k.compareTo(n.key);
    gp := p;
    p := n;
    n = n.child(dir);
 }
    return gp, p, n;
 }



bool contains(KeyType k){
  while(true) {
    _,_,n := search(k);
    if(k.compareTo(n.key) != 0)
       return false;
    return true;
  }
}

bool remove(KeyType k){
 while(true){
   gp,p,n := search(k);
   if(k.compareTo(n.key)!=0)
      return false;

   atomic{
      if(gp.isParentOf(p) &&
      p.isParentOf(n)&& !gp.marked) {
          n.marked = true;
          p.marked = true;
//gp change child from p to the child of p other than n.
            gp.changeChild(p,p.getOtherChild(n));
         return true;
      }}}}
```

```
 struct Leaf: Node {
    Key k;
    bool marked;
}
 struct Internal : Node {
     Key k;
     Node * children[2];
     marked;
    }
 void init() {
      Root = new Internal(-infty);
      Root.left = new Leaf(-infty);
      Root.right = new Leaf(+infty);

}

bool add(KeyType k){
   while(true){
       gp,p,n := search(k);
       dir = k.compareTo(n.key);
       if(dir == 0)
          return false;
       na = new Leaf(k);
       n1 = new Internal(n.key);
       n1.setChild(n,na);
       atomic{
         if(!p->marked
             && p.isParentOf(n))
              {  p.changeChild(n,n1);
        // change p's child from n to n1.
                  return true; }}}}
```

Listing 1: Implementation of the optimistic BST

We make several explanations of the algorithm in Listing 1. First, `compareTo` is a method to compare keys. The method `k1.compareTo(k2)` returns 0 if two keys are equal, or it returns -1 when $k1 < k2$, or 1 when $k1 > k2$. Second, the children choosing function `n.child(dir)` for internal node n returns the left child if $dir == 0$ or $dir == -1$, and the function returns the right child otherwise.

The most surprising part of this algorithm is the `search` operation. It traverses the data structure without any synchronization or retry. Many recent tree-based algorithms such as [2] share this property, and many of these algorithms can be verified with our method.

Dictionary Operations. The algorithm implements three common operations, `contains`, `add` and `remove`. Their sequential specifications are listed in the table below. They all use the helper operation `search` to locate the position where the operations take place. `add` and `remove` operations modify the heap under the protection of atomic sections. The atomic sections first check a set of validity conditions and retry if they are violated. After these validity checks, atomic sections perform the heap modification safely.

The sequential specification of the concurrent object can be viewed as a set of operations which operate on an abstract set S of keys.

Precondition	operation	Postcondition
$S = A$	`contains(k)`	$S' = A \wedge ret = k \in A$
$S = A$	`add(k)`	$S' = A \cup \{k\} \wedge ret = k \notin A$
$S = A$	`remove(k)`	$S' = A - \{k\} \wedge ret = k \in A$

4 Basic Definitions

States and Transitions. Program states are combinations of local stores and a shared heap. The ith local store s_i is a map from the local variables of thread i to values. A shared heap h is a finite map from memory locations L to values. The heap can be accessed by all threads. A memory state can be written as $\sigma = (s, h)$. In our specific setting, $h = h_a \cup h_b$, where h_a is the set of memory locations which can be accessed by following heap pointer links starting from `Root`, h_b is the locations which cannot be accessed from `Root`.

Backbone Nodes/Links. For a state σ, a link $u \rightarrow_\sigma v$ is a pair of nodes such that for some $i \in 1, 2$, $u.child(i) = v$ in state σ. A node/link is called a backbone node/link in state σ, if and only if in state σ, there is a link path from `Root` to the node/link. In any state σ, for two backbone nodes u, v we say that $u <_\sigma v$, if and only if there is a link path from v to u. The state may be omitted if it can be inferred from the context.

Computation Steps and Executions. For any thread t, we define a computation step s of t as a transition κ from state σ to σ'. We write $s = \sigma \rightarrow^t_\kappa \sigma'$, and denote $src(s) = \sigma$, $trg(s) = \sigma'$. A computation step of thread t is either an invocation of an operation, a return from an operation, or an atomic action in an operation invoked by thread t.

An execution Π is an alternating sequence of states and computation steps $\sigma_0, s_0, \sigma_1, s_1, \cdots$, where $\sigma_i = src(s_i)$ and $\sigma_{i+1} = trg(s_i)$. We define an execution trace of the execution by omiting all the computation step symbols, namely $\sigma_0, \sigma_1, \cdots$. An execution trace π can be simplified if we consider only heap computation steps, these simplified execution traces are called heap execution traces, they are simplifications of corresponding full execution traces.

Temporal Node Path, Temporal Backbone. In an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$, a sequence of consecutive pairs of different nodes $(u_0, u_1), (u_1, u_2), \cdots (u_{m-1}, u_m)$ is called a node path. It is called a temporal node path / backbone, if there is a sequence of integers $0 \leq i_1 \leq i_2 \cdots i_m \leq n$, such that (u_{k-1}, u_k) is a link / backbone link in state σ_{i_k} for each $1 \leq k \leq m$. Sometimes, we also call the sequence $u_0, u_1, \cdots u_m$ a temporal node path / backbone going through the subsequence $T_s = \{\sigma_{i_1}, \cdots \sigma_{i_m}\}$.

Bounds. For every state $\sigma = (s, h)$, from the state invariants in Figure 1, we know that the

Shape	ϕ_R	`Root` node exists.
Shape	ϕ_{loop}	Shared heap does not contain any loop.
Shape	ϕ_{c2}	Every internal node has two children.
Data	ϕ_∞	`Root` node has key $-\infty$.
Data	$\phi_<$	Data preserves tree order, for any node u, the keys on the left subtree $\leq$ $u.key <$ the keys on the right subtree.
Mark	ϕ_R	A node is marked $\Leftrightarrow$ it is a removed node.
Shape	δ_e	Child fields of removed nodes never changes.
Shape	δ_o	For a computation step (σ_1, σ_2), if $u <_{\sigma_2} v$, then $u <_{\sigma_1} v$
Shape	δ_R	`Root` never changes.
Shape	δ_{sn}	If a computation step removes a backbone node, the successors of the node remains unchanged in the next state.
Shape	δ_{Re}	A marked node can never become backbone again.
Data	δ_K	Key of any node can never change.

Table 1: Invariants of an external binary search tree.

reachable heap h_0 is actually a binary search tree. For every unmarked node u in heap h_0, there is a unique heap path $(u_0 = \texttt{Root}, u_1, u_2 \cdots u_m = u)$ from `Root`. We associate to u a real interval $S_\sigma(u) = (a, b]$, where $a = \max\{u_i.key | u_i.key < k, i \in [1, k]\}$, and $b = \min\{u_i.key | u_i.key \geq k, i \in [1, m]\}$. We refer to this interval as *Static Bound* of u at state σ.

Given a temporal node path $P = (v_0, \cdots, v_n)$ going through states T_s such that v_0 is `Root`, we define intervals $D_l(P) = (c, d]$ for $l \in \{1, 2 \cdots n\}$, where $c = \max\{v_i.key | v_i.key < k, i \in [1, l]\}$, and $d = \min\{v_i.key | v_i.key \geq k, i \in [1, l]\}$. $D_n(l)$ is called the *Dynamic Search Bound* of P_l.

Linearizability. Linearizability [6] is a widely-used correctness property of concurrent objects. Intuitively, it means each operation can be viewed as taking effect at some unique point in time between the invocation and response. We put the definition of linearizability in Appendix C.

5 Verification of the Algorithm

5.1 Thread-Local Invariants Needed for the Proof

Our proof relies on a set of thread modular invariants. Basically, we classify two main classes of invariants: state invariants and step invariants. State invariants are predicates $p(\sigma_0)$ on the state of shared heap σ_0, which can be written as separation logic formulas. Step invariants are predicates on single computation steps. Step invariants can also be written as separation logic formulas, taking account in both pre- and post-program states.

These invariants are natural to concurrent binary search tree algorithms, and most algorithms preserve at least some of them. These invariants can all be formally verified using separation logic. We list these invariants in Table 1. State invariants are named using ϕ, while step invariants are named using δ. Note that these invariants are by no means immutable. They can certainly be modified to verify other algorithms.

5.2 Generalized Hindsight Lemma

Lemma 1. ***Tree Version Hindsight Lemma***

Consider an execution trace satisfying the shape invariants in Table 1, $\sigma_0, \sigma_1, \cdots \sigma_n$. For $0 \leq i \leq j \leq n$, if there is a backbone link $u \rightarrow_{\sigma_i} v$, and a link $v \rightarrow_{\sigma_j} w$ (u, v, w are different nodes), then there is $i \leq k \leq j$, such that $v \rightarrow_{\sigma_k} w$ is a backbone link.

Proof. See proof of Lemma 7. □

Lemma 2. ***Tree Version Temporal Backbone Lemma***

Given an execution trace $T = (\sigma_0, \sigma_1, \cdots, \sigma_n)$ satisfying the shape invariants and a temporal node path $N = \{(u_0 = \texttt{Root}, u_1), (u_1, u_2), \cdots (u_{m-1}, u_m)\}$ going through $T_s = \{\sigma_{i_1}, \cdots \sigma_{i_m}\}$, Then there is another subsequence of execution trace $T'_s = \{\sigma_{j_1}, \cdots \sigma_{j_m}\}$ such that for all $1 \leq k \leq m-1$, $j_{k-1} \leq j_k \leq i_k$, and N is a temporal backbone going through T'_s.

Proof. Apply the Tree version Hindsight Lemma n times, and the theorem follows. □

For a `search` operation invoked by any thread t, the operation crosses the links to reach a leaf node. The search path of thread t is defined as a temporal node path $N = u_0, u_1, \cdots u_m$ of all the nodes visited by the search operation.

Corollary 5.1. *Consider an execution trace of the algorithm in Listing 1 satisfying shape invariants, $T = (\sigma_0, \sigma_1, \cdots \sigma_n)$. If this execution trace has an invocation to the search operation of a thread t, its search path is $N = u_0, u_1, \cdots u_m$. Then there is a subsequence $T_s = \{\sigma_{i_1}, \cdots \sigma_{i_m}\}$, such that N is a temporal backbone goes through T_s.*

5.3 Data-Expansion Lemma

To state the Data-Expansion Lemma, we reconsider our definition of static bound $S_\sigma(u)$ of a backbone node u in a state σ. We want to extend the definition to removed nodes. Since a removed node v must be on the backbone at some past state, we denote τ the last state when v was on the backbone. Then we define $S_\sigma(v) = S_\tau(v)$. Note this static bound will never change after a node is removed. We can prove the static and temporal versions of Data-Expansion Lemma.

Lemma 3. ***Static Data-Expansion Lemma***

Given an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$ satisfying shape, data and mark invariants, then for each $0 \leq i \leq j \leq n$, if internal node u exists from state σ_i, then we have

$$S_{\sigma_i}(u) \subseteq S_{\sigma_j}(u)$$

Proof. We only have to prove $S_{\sigma_i}(u) \subseteq S_{\sigma_{i+1}}(u)$ for each i. We distinguish 2 cases:

1. If u is a backbone node in both σ_i and σ_{i+1}. Let $A_k = \{v | v < u \text{ in } \sigma_k\}$. Then $A_{i+1} \subseteq A_i$, because from δ_o, each node such that $u <_{\sigma_{i+1}} v$ satisfied $u < v_{\sigma_i}$.

   So $A_{i+1} \subseteq A_i$. Since the static bound is determined by the the set A_i, and the key of u remain the same, so the static bound of u is non-decreasing.

2. If u is not a backbone node in σ_{i+1}, then the static bound is obviously the same in σ_i and σ_{i+1}.

□

Lemma 4. ***Data-Expansion Lemma***

Let T be an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$ satisfying shape, data and mark invariants. Let P be a temporal node path $P = \{v_0 = \texttt{Root}, v_1, \cdots v_m\}$ goes through subsequence $T_s = \{\sigma_{\tau(1)}, \cdots \sigma_{\tau(m)}\}$. For simplicity we assume v_m is a leaf node. Then the dynamic search bound $D_i(P)$ of the temporal node path is contained in the static bound $S_{\sigma_{\tau(i)}}(v_i)$. Namely, we have

$$D_i(P) \subseteq S_{\sigma_{\tau(i)}}(v_i), 0 \leq i < m$$

Proof. Because $v_0 = \texttt{Root}$, the temporal node path is also a temporal backbone goes through subsequence $T'_s = \{\sigma_{\gamma(1)}, \cdots \sigma_{\gamma(m)}\}$, such that $\gamma(k-1) \leq \gamma(k) \leq \tau(k)$ for each k.

We prove a stronger form of the lemma:

$$D_i(P) \subseteq S_{\sigma_{\gamma(i)}}(v_i), 0 \leq i < m$$

Due to the static Data-Expansion lemma, we have $S_{\sigma_{\gamma(i)}}(v_i) \subseteq S_{\sigma_{\tau(i)}}(v_i)$, so this stronger form implies our lemma.

We prove this lemma by induction on i. For $i = 0$, the lemma holds trivially, because $D_0(P) = S_{\sigma_{\gamma(0)}}(v_0) = (-\infty, +\infty)$.

We assume $i = k$, $D_k(P) \subseteq S_{\sigma_{\gamma(k)}}(v_k)$. Because of the static Data-Expansion Lemma, we have $D_k(P) \subseteq S_{\sigma_{\gamma(k)}}(v_k) \subseteq S_{\sigma_{\gamma(k+1)}}(v_k)$.

For $i = k + 1$, in state $\sigma_{\gamma(k+1)}$, the link $l : u_k \to u_{k+1}$ is a backbone link. Crossing the link would put the same constraint on both dynamic search bound and static bound, for example, if link l is a right child pointer of u_k, then $D_{k+1}(P) = D_k(P) \cap (u_k.key, +\infty)$, and also $S_{\sigma_{\gamma(k+1)}}(v_k) = S_{\sigma_{\gamma(k+1)}}(v_{k+1}) \cap (u_k.key, +\infty)$. So we have $D_{k+1}(P) \subseteq S_{\sigma_{\gamma(k+1)}}(v_{k+1})$.

So the lemma holds for every $0 \leq i < m$. □

Corollary 5.2. *Suppose a search path $P = \{v_0, v_1, \cdots v_m\}$ is visited by a search operation in an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$ satisfying shape, data and mark invariants. For simplicity we assume the search operation invoked at state σ_0 and return at σ_n. We denote at state σ_i, the search operation is visiting node $v_{\phi(i)}$(namely pointer $n = v_{\phi(i)}$), then the dynamic search bound $D_{\phi(i)}(P)$ of node v is contained in the static bound $S_{\sigma_i}(v_{\phi(i)})$. Since the search key k always lies in the dynamic search bound, we have*

$$k \in D_{\phi(i)}(P) \subseteq S_{\sigma_i}(v_{\phi(i)})$$

Proof. A search path is a temporal node path from `Root` goes through a sequence of states $T_s = \{\sigma_{\tau(1)}, \cdots \sigma_{\tau(m)}\}$. So we have from above lemma:

$$D_{\phi(i)}(P) \subseteq S_{\sigma_{\tau(\phi(i))}}(v_{\phi(i)}) \subseteq S_{\sigma_i}(v_{\phi(i)})$$

This is because obviously we can make $\phi(\tau(k)) = k$ for $k \leq m$. □

5.4 Verification of Linearizability

We define an effectless operation as one of three types: `contains` operations, `remove` operations returning false, and `add` operations returning false. In these three cases, the linearizability points of these operations are non-fixed. Namely, the linearizability point of one thread running an effectless operation is sometimes in another thread. However, the linearizability of effectless operations can be directly deduced from the thread modular invariants, which simplifies our verification.

Lemma 5. *Effectless operations are linearizable with respect to their sequential specifications.*

Proof. All effectless operations invokes the `search` operation as sub-procedure. We assume the search path of one effectless procedure is $v_0 = \texttt{Root}, v_1, \cdots v_m$, v_m is a leaf node. The execution trace is $T = (\sigma_0, \sigma_1, \cdots \sigma_n)$. According to the temporal backbone lemma, we know that link $L_m = (v_{m-1}, v_m)$ was a backbone link in some past state. We denote σ_d the last state when the L_m is on backbone before the `search` operation crosses the link. (If it remains a backbone till the search crosses the link, we take σ_d to be the last state before it decides no further search is needed, line 5 in the algorithm below) We claim that σ_d is the right linearizability point.

We distinguish two cases: If a search operation actually "finds" a node with the search key , namely $v_m.key = k$, then in σ_d, v_m was on the backbone. If search operation finds $v_m.key \neq k$, then node with key k is not in the tree on σ_d. We can prove this as follows:

Without loss of generality, we assume $k > v_{m-1}.key$. The "$\leq$" case follows the same argument. From the Data-Expansion lemma, we know that $k \in S_{\sigma_d}(v_{m-1})$. Namely, if a leaf node k is presented in the tree, it should be found in the subtree rooted at node v_{m-1}, namely, on the right subtree of v_{m-1}.

If in σ_{d+1}, L_m is still a backbone link, then the computation step $s = (\sigma_d, \sigma_{d+1})$ is the link crossing of the `search` operation, the heap $h_{\sigma_d} = h_{\sigma_{d+1}}$. Then since $v_m.key \neq k$, so k is not in $Abs(\sigma_d)$.

If in σ_{d+1}, L_m is not a backbone link. The invariants δ_{sn} and δ_e guarantee that in σ_{d+1} and subsequent states, the right child of node v_{m-1} remains the same as in state σ_d. If node with key k exists in state σ_d, it should be on the right subtree of node v_{m-1}. However, the right child of v_{m-1} is a leaf node v_m with $v_m.key \neq k$. So we know that no leaf node with key k exists in state σ_d. □

The linearizability of effectful operations, which have fixed linearizability points, are not hard to prove.

Lemma 6. *The External BST algorithm implemented above is correct with respect to the sequential specification.*

Proof. It is easy to verify the invariants of Table 1 using separation logic [7] and Owicki-Gries logic [11]. This verification can be done in a purely thread modular way. The rest is to define the linearizability points of each operation. The linearizability point of a effectful operation is the state before the execution of the last atomic section. The linearizability points of effectless operations is defined above. The linearizability of effectless operations is implied by the thread-modular invariants, which we have already proved in the lemma above. Now we only have to prove the linearizability of effectful operations.

We consider the abstract set function on states, $Abs(\sigma)$. $Abs(\sigma)$ is the set of keys of all reachable (unmarked) leaf nodes in the tree. Its formal definition is included in Appendix B.

For effectful `add` operations, let $s = (\sigma_a, \sigma'_a)$ be the computation step of the execution of last atomic section. The validation condition ensures $l_1 : *p \to n$ is a backbone link in σ_a. Using this validation condition and the state invariants in Listing 1, and the definition of abstraction function, it is obvious to check the computation step modify the heap according to its specification: all leaf nodes reachable from `Root` in σ_a remains reachable in σ'_a, and a single new leaf node with key k become reachable.

The case for effectful `remove` operations is similar. let $s = (\sigma_r, \sigma'_r)$ be the computation step of the execution of last atomic section. The validation condition ensures $l_1 : *p \to n$ and $l_2 : *gp \to p$

are backbone links in σ_r. It is obvious to check the computation step modifies the heap according to its specification: all leaf nodes reachable from `Root` in σ_a remains reachable in σ'_a, except the leaf node pointed by n. $\square$

6 Generalized Hindsight Lemma

In this section, we generalize Hindsight Lemma to a very general form. The lemma plays an essential role in the verification of both linked list and trees, and interestingly, it is still valid on a large class of linked data structures. We use the concept of *search data structure* to express the lemma.

Definition 6.1. *A data node is a fixed-size dynamic-allocated heap object consisting of a boolean mark field, a data field and several successor pointers to other data nodes. A search data structure is a heap object consisting of several data nodes with a specific node H, called the entry node. A concurrent search structure is a concurrent object whose shared heap is a search data structure. We assume that the concurrent object also satisfies the thread-modular invariant that a node is marked if and only if it is unreachable from H.*

For a concurrent search structure T, we assume the object also satisfies the step invariant that when or after nodes are removed from reachable heap, they cannot become backbone again and their successor pointers remain unchanged. We call this assumption "Removed Unchanged Assumption (RUA)". On a concurrent search structure, we define link, backbone link, temporal backbone, temporal node path as we do in Section 4 and Section 5. We formalize all the conditions of the Generalized Hindsight Lemma and the Generalized Temporal Backbone Lemma in Appendix B.

Lemma 7. ***Generalized Hindsight Lemma***
For a concurrent search structure T_g, assume T_g satisfies RUA*. Consider an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$. For $0 \leq i \leq j \leq n$, if there is a backbone link $u \rightarrow_{\sigma_i} v$, and a link $v \rightarrow_{\sigma_j} w$ (u, v, w are different nodes), then there is $i \leq k \leq j$, such that $v \rightarrow_{\sigma_k} w$ is a backbone link.*

Proof. If in state σ_j, node v is a backbone node, then choose $k = j$ and we are done. If not, then v is not a backbone node in σ_j, let l be the largest index such that v is a backbone node in σ_l, so $l \geq i$. In σ_{l+1}, v is removed from backbone. But the link $v \rightarrow_{\sigma_j} w$ exists, according to δ_e and δ_{sn}, $v \rightarrow w$ exists from state σ_l to σ_j. But in state σ_l, v is a backbone node, so the link $v \rightarrow_{\sigma_k} w$ is a backbone link. $\square$

7 Extensions to Other Algorithms

7.1 Balanced Binary Search Trees

Balanced binary search trees are similar to the example tree we present above, with a special rebalancing operation performing a tree rotation. Careless rebalancing will mislead the tree traversal. Consider the scenario in Figure 2. The searching thread T_1 fails to find the existing node with key 6 due to a concurrent rotation by thread T_2. So T_1 usually have to backtrack and retry. In this way, T_1 makes the Data-Expansion Lemma hold through the search path.

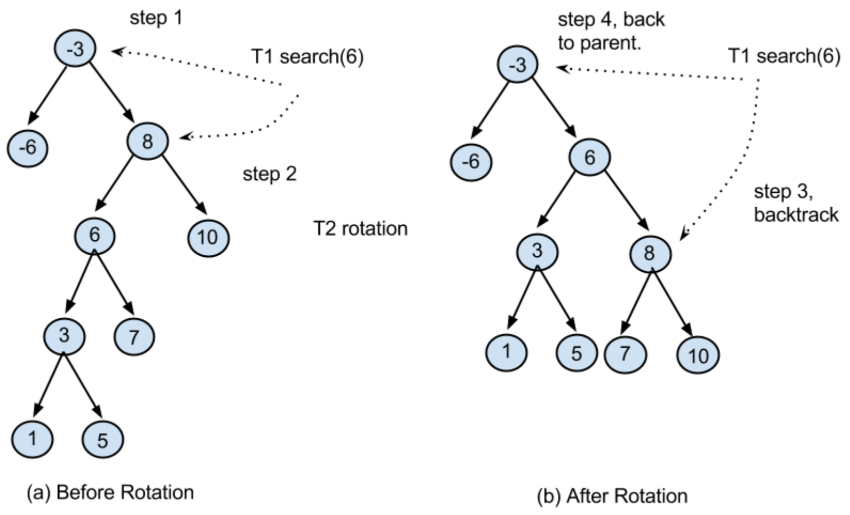

Figure 2: Concurrent BST with rotation

However, one can also perform rebalancing by addition and removal of nodes instead of in-place rotation. Performing a tree rotation is equivalent to performing the following procedure [2]: create two new nodes with key 6 and key 8, set their pointers to form the rotated subtree, and change the pointer of node with key -3 to the new node with key 6. In this way, the rebalancing procedure satisfies almost all the step and state invariants listed in Table 1, thus can be verified in this way.

Listing 2 is an example of balanced binary search tree implemented in [2]. We only list the `Delete` operation here for simplicity. Other operations are similar to this one, and they can be found in their paper.

First, we briefly review the LLX and SCX primitives [2]. To simplify our proof, we assume that the primitives are atomic. LLX and SCX operates on tree nodes. LLX(r) returns a snapshot of the mutable fields of the node r if r is on the tree. If r is removed from the tree, LLX(r) returns FINALIZED. $SCX(V, R, flr, new)$ modifies the data structure only when every node r in V has not been changed since LLX(r). The SCX atomically verifies the nodes in V remain unchanged, and then it sets the nodes in R as FINALIZED, at last it sets the pointer flr to new. The relaxed RB tree is an external binary search tree, with two sentinel nodes. Each node n in the tree has three fields, $n.w$ stands for weight, $n.k$ stands for the key, $n.v$ stands for the value corresponding to $n.k$, and $n.left$, $n.right$ stands for the children.

We can verify the tree-based map algorithm satisfies the invariants listed in Table 2.

```
Get(key)
  (-,-,l) := SEARCH(key);
  return (key = l.k)? l.v: null;

Search(key)
  n0 = null; n1 = Root; n2 := entry.left;
  while n2 is internal
    n0 := n1; n1 := n2;
    n2 := (key < n1.k)? n1.left: n1.right;
  return (n0,n1,n2);

Delete(key)
  do
```

```
    result := Trydelete(key);
  while result = FAIL;
  (value,violation) := result;
  if violation then Cleanup(key);
  return value;

Cleanup(key)
  while(true)
    n0 := null; n1 := null; n2:= Root; n3:= Root.left;
    while(true)
      if(n3.w > 1 || n2.w =0 && n3.w =0)
        Tryrebalance(n0,n1,n2,n3);
        break;
      else if n3 is a leaf
        return;
      if (key < n3.k)
        n0 := n1; n1 := n2; n2:= n3; n3:= n3.left;
      else n0 := n1; n1 := n2; n2:= n3; n3:= n3.right;

Trydelete(key)
  (n0,-,-) := Search(key);
  if (n0 = null) return (null, false)
  s0 =  LLX(n0);
  if (s0 = FAIL)
    n1 := (key <s0.left.k)? s0.left: s0.right;
  s1 = LLX(n1);
  if (s1 = FINALIZED) return FAIL;
  n2 := (key<s1.left.k) ? s1.left: s1.right;
  if(n2.k != key) return FALSE;
  s2 = LLX(n2);
  if (s2 = FINALIZED) return FAIL;
  n3 := (key<s1.left.k) ? s1.right: s1.left;
  s3 := LLX(n3);
  w := (n1.k = infty || n0.k = infty) ? 1: n1.w+n3.w
  V := (key<s1.left.k)? (n0,n1,n2,n3) :(n0,n1,n3,n2);
  R := (key<s1.left.k) ? (n1,n2,n3) :(n1,n3,n2);
  fld = (key<s0.left.k) & n0.left: & n0.right;
  if SCX(V,R,fld,new) return (n2.v, (w>1));
  else return FAIL;
```

Listing 2: Relaxed RB Tree

Using similar notations, we can prove the following Data-Expansion Lemma. The proof is almost the same as the one presented in Section 4. The General Hindsight Lemma also trivially applies here.

Lemma 8. ***Static Data-Expansion Lemma***

Given an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$ satisfying shape, data and mark invariants in Table 3, then for each $0 \leq i \leq j \leq n$, if internal node u exists from state σ_i, then we have

$$S_{\sigma_i}(u) \subseteq S_{\sigma_j}(u)$$

Proof. We only have to prove $S_{\sigma_i}(u) \subseteq S_{\sigma_{i+1}}(u)$ for each i. We distinguish 2 cases:

1. If u is a backbone node in both σ_i and σ_{i+1}. Let $A_k = \{v.key | v < u \text{ in } \sigma_k\}$. Then $A_{i+1} \subseteq A_i$, because from δ_o, each node key k such that $\exists u \bullet u.key = k \wedge u <_{\sigma_{i+1}} v$ satisfies also $\exists u \bullet u.key = k \wedge u <_{\sigma_i} v$.

   So $A_{i+1} \subseteq A_i$. Since the static bound is determined by the the set A_i, and the key of u remain the same, so the static bound of u is non-decreasing.

Shape	ϕ_R	`Root` node exists.
Shape	ϕ_{loop}	Shared heap does not contain any loop.
Shape	ϕ_{c2}	Every internal node has two children.
Data	ϕ_∞	`Root` node has key $-\infty$.
Data	$\phi_<$	Data preserves tree order, for any node u, the keys on the left subtree $\leq u.k <$ the keys on the right subtree.
Mark	ϕ_R	A node is finalized $\Leftrightarrow$ it is a removed node.
Shape	δ_e	Child fields of removed nodes never changes.
Shape	δ_o	For a computation step (σ_1, σ_2), v is a backbone node in σ_1 and σ_2. $\forall k$ such that $u <_{\sigma_2} v$ and $u.k = k$, we have $\exists u'$ such that $u'.k = k$ and $u <_{\sigma_1} v$.
Shape	δ_R	`Root` never changes.
Shape	δ_{sn}	If a computation step removes a backbone node, the successors of the node remains unchanged in the next state.
Shape	δ_{Re}	A finalized node can never become backbone again.
Data	δ_K	Key of any node can never change.

Table 2: Invariants of the relaxed RB tree.

2. If u is not a backbone node in σ_{i+1}, then the static bound is obviously the same in σ_i and σ_{i+1}.

□

Lemma 9. ***Data-Expansion Lemma***
Let T be an execution trace $\sigma_0, \sigma_1, \cdots \sigma_n$ satisfying shape, data and mark invariants. Let P be a temporal node path $P = \{v_0 = \texttt{Root}, v_1, \cdots v_m\}$ goes through subsequence $T_s = \{\sigma_{\tau(1)}, \cdots \sigma_{\tau(m)}\}$. For simplicity we assume v_m is a leaf node. Then the dynamic search bound $D_i(P)$ of the temporal node path is contained in the static bound $S_{\sigma_{\tau(i)}}(v_i)$. Namely, we have

$$D_i(P) \subseteq S_{\sigma_{\tau(i)}}(v_i), 0 \leq i < m$$

Following the same line, we define effectless operations are of three kinds: `Put` operations returning false, `Delete` operations returning false, and `Get` operations. It is easy to prove the following lemma.

Lemma 10. *Effectless operations are linearizable with respect to their sequential specifications.*

Note since the rebalancing operations do not change the value of *Abs* and the static bound, so this operation does not need to be taken into consideration explicitly in our verification. This is why our verification is much simpler than previous solutions.

7.2 Skip Lists

we can also easily apply the verification method in this paper to an optimistic variant of the lazy skip list algorithm [5]. Traversal in the skip list is wait-free, and several natural thread modular invariants can imply Generalized Hindsight Lemma and Data-Expansion Lemma on skip lists, thus can guarantee the correctness of the algorithm.

We list the optimistic version of the algorithm in Listing 3.

```
class Node {
  int key;
  int topLayer;
  Node *[MaxHeight] nexts;
  bool marked;
}

int findNode(int v, Node* preds[], Node * succ){
  int lFound = -1;
  Node* pred = Head;
  for(int layer = MaxHeight -1; layer>= 0; layer --){
    Node * curr = pred->nexts[layer];
    while(v>curr->key){
      pred = curr; curr = pred->nexts[layer];
    }
    if(lFound == -1 && v = curr->key)
      lFound = layer;
    preds[layer] = pred;
    succs[layer] = curr;
  }
  return lFound;
}

bool add(int v){
  int topLayer = randomLevel(MaxHeight);
  Node * preds[MaxHeight], succs[MaxHeight];
  while(true){
    int lFound = findNode(v,preds,succs);
    if(lFound != -1)
      return false;
  }
  Node * pred, * succ, * prevPred = null;
  bool valid = true;
  atomic{
  for(int layer =0; valid && layer<= toplayer; layer++){
    pred = preds[layer];
    succ = succs[layer];
    if(pred != prevPred)
      prevPred = pred;
    valid = !prev->marked && !succ->marked &&
      pred->nexts[layer] == succ&& valid;
  }
  if(!valid) continue;
  Node * newNode = new Node(v,topLayer);
  for(int layer =0;layer<= toplayer;layer++){
    newNode -> nexts[layer] = succs[layer];
    preds[layer]->nexts[layer] = newNode;
  }
  return true;
 }
}

bool remove(int v){
  Node *nodeToDelete = null;
  bool isMarked = false;
  int topLayer = -1;
  Node* preds[MaxHeight], succs[MaxHeight];
  while(true){
    int lFound = findNode(v,preds,succs);
    if(lFound == -1){
      return false;
```

Shape	ϕ_R	`Head` and `Tail` node exists.
Shape	ϕ_{loop}	Shared heap does not contain any loop.
Shape	ϕ_T	The tail node has no successor.
Shape	ϕ_{c2}	Every node other than the tail node has *node.topLayer* successors.
Data	$\phi_{H\infty}$	`Head` node has key $-\infty$.
Data	$\phi_{T\infty}$	`Tail` node has key $+\infty$.
Data	ϕ_{sort}	The keys of successors of a node are strictly sorted, nodes in the bottom layer have smallest key.
Data	$\phi_<$	Data preserves order, the key of every node is smaller than the keys of its successors.
Mark	ϕ_R	The tail node is reachable from every node.
Shape	δ_e	Successor fields of removed nodes never changes.
Shape	δ_H	The value of `Head` never changes.
Shape	δ_T	The value of `Tail` never changes.
Shape	δ_{sn}	If a computation step changes the successor pointer of one node, the node will remain on backbone in the following state.
Shape	δ_{Re}	A removed node can never become backbone again.
Data	δ_K	Key of any node can never change.

Table 3: Invariants of the optimistic skip list.

```
      }
      nodeToDelete = succs[lFound];
      atomic{
         for(int layer =0; valid && layer<= toplayer; layer++){
          pred = preds[layer];
          succ = succs[layer];
          valid = !pred->marked && pred->nexts[layer] = succ;
          for(int layer =toplayer ;layer>=0; layer--)
            preds[layer]->nexts[layer] = nodeToDelete->nexts[layer];
      }
      return true;
    }
  }}
```

Listing 3: Optimistic skip list

For the skip list algorithm above, we also figure out the set of invariants below in Table 3.

We can also define the concepts of the temporal backbone, temporal node paths on the optimistic skip list. Then we have the following version of Generalized Hindsight Lemma:

Lemma 11. ***Skip-List Version Hindsight Lemma***

Consider an execution trace satisfying the shape invariants in Table 3, $\sigma_0, \sigma_1, \cdots \sigma_n$. For $0 \leq i \leq j \leq n$, if there is a backbone link $u \rightarrow_{\sigma_i} v$, and a link $v \rightarrow_{\sigma_j} w$ (u, v, w are different nodes), then there is $i \leq k \leq j$, such that $v \rightarrow_{\sigma_k} w$ is a backbone link.

Proof. See proof of Lemma 7. □

Lemma 12. ***Skip-List Version Temporal Backbone Lemma***

Given an execution trace $T = (\sigma_0, \sigma_1, \cdots, \sigma_n)$ satisfying the shape invariants in Table 3 and a temporal node path $N = \{(u_0 = \texttt{Root}, u_1), (u_1, u_2), \cdots (u_{m-1}, u_m)\}$ going through $T_s = \{\sigma_{i_1}, \cdots \sigma_{i_m}\}$, Then there is another subsequence of execution trace $T'_s = \{\sigma_{j_1}, \cdots \sigma_{j_m}\}$ such that for all $1 \leq k \leq m-1$, $j_{k-1} \leq j_k \leq i_k$, and N is a temporal backbone going through T'_s.

Proof. Apply the Tree version Hindsight Lemma n times, and the theorem follows. □

We also have the following version of Data-Expansion Lemma.

Lemma 13. *Consider an execution trace satisfying the shape, data and mark invariants in Table 3, $\sigma_0, \sigma_1, \cdots \sigma_n$. Let N be a temporal node path $N = \{(u_0, u_1), (u_1, u_2), \cdots (u_{m-1}, u_m)\}$ goes through subsequence $T_s = \{\sigma_{i_1}, \cdots \sigma_{i_m}\}$. Assume u_{m-1}, u_m are on the bottom layer of the skip list. Then for each $l = (u_{m-1}, u_m)$, there is a state σ_{n_i}, such that $Abs(\sigma_{n_i}) \cap [u_i.key, u_{i+1}.key] = \{u_i.key, u_{i+1}.key\}$.*

Proof. The proof follows from the Generalized Hindsight Backbone Lemma, and the invariants that keys in the skip lists are strictly sorted. □

Following the same line, we define effectless operations are of three kinds: `add` operations returning false, `remove` operations returning false, and `find` operations. It is easy to prove the following lemma.

Lemma 14. *Effectless operations are linearizable with respect to their sequential specifications.*

8 Remarks

8.1 Related Works

The proof strategy used in this paper is essentially based on the idea of Herlichy and Wing [5]. In their fundamental paper, a proof of linearizability using data abstraction function is presented.

Our work is related to the recent advances [4, 1, 3] in concurrent binary search tree algorithms. The algorithm we set as our verification target is similar to [4], except that we use locks or atomic sections instead of non-blocking primitives. We find the idea of our verification may also be applicable to many of these algorithms. Our work also shares commonalities with the Hindsight Lemma paper [10]. We go one significant step forward by providing purely thread modular proofs for advanced concurrent algorithms such as trees. In fact, most tree algorithms are extremely complicated and hard to prove correct or verify rigorously. There are some recent proofs for tree algorithms [4], However, their proofs are purely mathematical (not formal) and do not use explicit thread modular arguments, making their proofs much longer than ours, and it is very hard (if not impossible) to refine their proofs into formal ones.

As the verification of the lazy linked list algorithm in [10], our verification of the invariants can also be viewed as taking place in simple Owicki-Gries logic [11], namely, we do not use complex mechanisms such as these used in rely-guarantee reasoning [8]. In order to express our verification in a clean way, we use small atomic sections instead of locks. This technical limitation, however, is by no means essential. In the price of more complicated proofs, we can actually allow the verification of lazy counterpart of these algorithms with some extra complexity.

In [2], a new abstraction to implement concurrent search trees is presented, together with several mathematical proofs of correctness. Also, in the correctness proof, the author proved a result similar to the Generalized Hindsight Lemma in the context of their implementation. However, their correctness results rely on specialized implementation technique and do not rely on explicit thread local invariants. So they cannot be used as a basis for thread modular formal verification.

8.2 Conclusions

Formal verification of shared memory concurrent algorithms is a hard but important problem in the multicore era. The main difficulty is to prove the correctness of the algorithms in the presence of complicated thread interferences. Existing methods such as [11] usually need to introduce many auxiliary states, which lead to over-complicated proofs. So they cannot be adapted to some advanced concurrent data structures, such as binary search trees. In [10], O'Hearn etc. have shown that for a special concurrent linked list algorithm, thread modular verification can be established. In this paper, we make a rather surprising observation that for some advanced concurrent data structures, such as binary search trees, thread modular proofs are also achievable, thus can greatly simplify formal verification of concurrent algorithms.

In [11], Owicki and Gries argued that using auxiliary states is sometimes a must, and many simple concurrent programs cannot admit purely thread modular proofs without auxiliary states. Although Owicki and Gries' work limits the use of thread modular proofs, it is interesting to see that many advanced highly concurrent data structures do not fall into this limitation. Thanks to the Data-Expansion lemma and the Generalized Hindsight Lemma, we can see that some advanced concurrent algorithms can admit purely thread modular formal verification. This observation makes the goal of formal verification of many advanced concurrent objects actually achievable.

On the bright side, the Generalized Hindsight Lemma is proved correct on a large class of data structures satisfying only the Removed Unchanged Assumption, which is easy to formalize and, hopefully, to automate. On the other side, the Data-Expansion Lemma is more data structure specific. It is shown in our running example to hold on the binary search trees. However, the lemma is very promising for generalization to other data structures.

The Data Expansion Lemma combined with the Generalized Hindsight Lemma eliminates the needs of constructing linearizability points in other threads before carrying out formal proofs. This is particularly important for advanced concurrent data structures, such as binary search trees, whose internal logic is highly complicated. These lemmas give an direct formal explanation of why the tree traversal can work without any synchronization. They may play an important rule in the design and verification of concurrent algorithms.

References

[1] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.

[2] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.

[3] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 343–356, New York, NY, USA, 2014. ACM.

[4] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

[5] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

[6] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[7] Samin S Ishtiaq and Peter W O'Hearn. Bi as an assertion language for mutable data structures. In *ACM SIGPLAN Notices*, volume 36, pages 14–26. ACM, 2001.

[8] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[9] Aravind Natarajan and Neeraj Mittal. Fast concurrent lockfree binary search trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*, 2014.

[10] Peter W O'Hearn, Noam Rinetzky, Martin T Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 85–94. ACM, 2010.

[11] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

A Programming Language

We define that a concurrent program P is comprised of a *init* atomic command and a set of concurrent operations. One thread runs the *init* command first, and then other threads are made ready to run.

Atomic commands. An atomic command is comprised of one or more program statements. It can be represented as an edge in the control flow graph, so each atomic command is a basic block. However, a basic block is not necessarily an atomic command. Intuitively, atomic commands are not interfered by other threads. Each atomic command may access global variables, thread-local variables or the shared heap. To simplify the programming language, we only allow the thread to refer to the global variables and the share heap by load/store statements.

Concurrent Operations. A concurrent operation is comprised of several atomic commands. A concurrent operation can be run by any thread, and the schedule of atomic commands between concurrently executing threads are completely undefined. We define the semantic domain of our programming language below. $A \rightharpoonup B$ represents the set of partial functions from A to B.

$$\begin{aligned}
&\text{Values} = \text{Integers} \cup \text{Bools} \cup \text{Locations}\\
&\text{Heaps} = \cup_{A \in_{\texttt{fin}} \text{Locations}}(A \rightharpoonup \text{Memory Allocated Objects})\\
&\text{Memory Allocated Objects} = \text{Field Names} \rightharpoonup \text{Values}\\
&\text{Stores}_L = V_L \rightharpoonup \text{Values}\\
&\text{Stores}_G = V_G \rightharpoonup \text{Values}
\end{aligned}$$

We can define the syntax of our programming language. e represents an expression over local stack variables, T represents a memory allocated object type, c is a field of a heap cell. $\{x, y, z \cdots\}$ are local stack variables in V_L, we write the local variables of thread t by $\{x_t, y_t \cdots\}$. $\{X, Y, Z \cdots\}$ are global stack variables in V_G, b is a boolean variable.

```
Stmt = Skip
| x = e | x = Y | X = y
| x = new T()| assume(b)
| x = y.c | x.c = y
```

Listing 4: Program Statements

Memory States. A memory state $\sigma = (s_l, s_g, h)$ is comprised of a local stack for every thread $s_l \in T \rightharpoonup \text{Stores}_L$, a global stack $s_g \in \text{Stores}_G$, and a heap $h \in$ Heaps. We write Σ the set of program states, and T the set of thread IDs. The small step semantics of the program statements can be defined in Table 2.

Program Semantics. The semantics of an atomic command is determined by the composition of relations of the small step semantics above. An atomic transition $tr_a = \sigma \rightarrow_t \sigma'$ means that executing sequentially the program statements in the semantics above may produce a new program state σ'. Its execution thread is t. The atomic transition is also called a computation step. A program execution is a sequence of transitions $\sigma_1, \sigma_2 \cdots$.

skip	$(s_l, s_g, h) \to (s_l, s_g, h)$	
x = e	$(s_l, s_g, h) \to (s_l[x_t \mapsto [\![e]\!](s_l)], s_g, h)$	
x = Y	$(s_l, s_g, h) \to (s_l[x_t \mapsto s_g(Y)], s_g, h)$	
X = y	$(s_l, s_g, h) \to (s_l, s_g[x \mapsto s_l(y_t)], h)$	.
x = new T()	$(s_l, s_g, h) \to (s_l[x_t \mapsto l], h[l \mapsto \lambda c.\bot])$	$l \notin dom(h)$
x = y.c	$(s_l, s_g, h) \to (s_l[x_t \mapsto h(y_t, c)], s_g, h)$	$s_l(y_t) \in dom(h)$
x = y.c	$(s_l, s_g, h) \to$ error	$s_l(y_t) \notin dom(h)$
x.c = y	$(s_l, s_g, h) \to (s_l, s_g, h[(s_l(x), c) \mapsto s_l(y_t)])$	$s_l(x_t) \in dom(h)$
x.c = y	$(s_l, s_g, h) \to$ error	$s_l(x_t) \notin dom(h)$
assume(b)	$(s_l, s_g, h) \to (s_l, s_g, h)$	$[\![b]\!](s_l)$

Table 4: Semantics of Program Statements by Thread t.

B Formal Definitions of Invariants and Abstraction Function

B.1 Invariants Used in the Proof of Concurrent BST

We formalize the shape and state invariants using separation logic. The invariants can then be used in thread modular formal verification.

First, we define some sub-formulas corresponding to four kinds of nodes on the tree:

$$N_1^I(x, k, p_1, p_2) = x \mapsto (\mathbf{false}, k, p_1, p_2)$$

$$N_1^L(x, k) = x \mapsto (\mathbf{false}, k)$$

$$N_2^L(x, k, p_1, p_2) = x \mapsto (\mathbf{true}, k, p_1, p_2)$$

$$N_2^L(x, k) = x \mapsto (\mathbf{true}, k)$$

$$N(x, k) = N_1^L(x, k) \vee N_2^L(x, k) \vee (\exists p_1, p_2 \bullet N_1^I(x, k, p_1, p_2)) \vee (\exists p_1, p_2 \bullet N_2^I(x, k, p_1, p_2))$$

Each unmarked leaf node on the tree may be the root of a removed inversed tree. So we define the sub-formulas:

$$T(x, k) = \mathbf{emp} \vee (\exists k_1', p_1', y \bullet k_1' > k \wedge N_2^I(y, k_1', p_1', x) * Tr_2(p_1', k_1'))$$
$$\vee(\exists k_2', p_2', y' \bullet k_2' \le k \wedge N_2^I(y', k_2', x, p_2') * Tr_3(p_2', k_2'))$$

$$Tr_2(x, k) = \exists k_1 \bullet (k_1 \le k \wedge N_2^L(x, k_1) * T(x, k_1))$$

$$Tr_3(x, k) = \exists k_2 \bullet (k_2 > k \wedge N_2^L(x, k_1) * T(x, k_1))$$

Then we can encode the invariants in Table 1 in a single separation logic formula:

$$Tr_1(x, k) = (N_1^L(x, k) * T(x, k))$$
$$\vee(\exists p_1, k_1, p_2, k_2 \bullet k_1 \le k \wedge k < k_2 \wedge N_1^I(x, k, p_1, p_2) * Tr_1(p_1, k_1) * Tr_1(p_2, k_2))$$

The overall shape invariant is defined as:

$$\phi(\texttt{Root}, -\infty) = Tr_1(\texttt{Root}, -\infty) \tag{1}$$

First, we have the following immutability invariants:

$$\phi_k(x, k) = N(x, k) \tag{2}$$

$$\phi_R(x) = \quad \texttt{Root} = x \wedge x \neq \text{null} \tag{3}$$

Second, we have the removed unchanged assumption invariant:

$$\phi_{rua}(x, k, p_1, p_2) = N_2^I(x, k, p_1, p_2) \tag{4}$$

We have also the following conditional order preserving invariant, it holds in any computation step σ_1, s, σ_2, when x, y both points to backbone nodes in pre and post states σ_1 and σ_2. First we write the following sub-formula:

$$p_r(x, y) = (\exists k, p_1 \bullet N_1^I(x, k, p_1, y)) \vee (\exists k, p_2 \bullet N_1^I(x, k, y, p_2))$$
$$\vee(\exists k, p_1, z \bullet N_1^I(x, k, p_1, z) * \phi_r(z, y)) \vee (\exists k, p_2, z \bullet N_1^I(x, k, z, p_2) * \phi_r(z, y))$$

Then the invariant can be written as, for any computation step (σ, σ'), we have:

$$\sigma \models p_r(x, y)*\text{true} \Leftrightarrow \sigma' \models (p_r(x, y)*\text{true}) \vee (\exists k, p_1, p_2 \bullet N_2^I(x, k, p_1, p_2)) \vee (\exists k, p_1, p_2 \bullet N_2^I(y, k, p_1, p_2))$$

The abstraction function is defined as:

$$Abs(\sigma) = \{k \in (-\infty, +\infty) | \sigma \models N_1^L(\_, k) * \text{true}\} \tag{5}$$

We have the following obvious formula, namely, a node is not removed is a node which is reachable from `Root`.

$$\phi(\texttt{Root}, -\infty) \Rightarrow (\forall k \bullet (k \in Abs(\sigma)) \Leftrightarrow \exists x \bullet p_r(\texttt{Root}, x) * N_1^L(x, k) * \text{true}) \tag{6}$$

B.2 Formalizing the Generalized Hindsight Lemma

We formalize the Generalized Hindsight Lemma, to make it available in future verification of concurrent algorithms. We present a few invariants, that can be easily checked by Hoare tuples written in separation logic. In this way, we can see why this lemma is promising to automate.

First, we assume the data nodes have type $T_1, T_2, \cdots T_n$. n is a constant. Nodes of type T_i have a mark field, a data field, and k_i pointers to other nodes. First , we define sub-formulas:

$$N_1^i(x, d, \mathbf{p_i}) = N_1^i(x, d, p_1, p_2, \cdots p_{k_i}) = x \mapsto (\mathbf{false}, d, p_1, p_2 \cdots p_{k_i}), \quad 1 \le i \le n \tag{7}$$

$$N_2^i(x, d, \mathbf{p_i}) = N_1^i(x, d, p_1, p_2, \cdots p_{k_i}) = x \mapsto (\mathbf{true}, d, p_1, p_2 \cdots p_{k_i}), \quad 1 \le i \le n \tag{8}$$

$$N^i(x, d, \mathbf{p_i}) = N_1^i(x, d, \mathbf{p_i}) \vee N_2^i(x, d, \mathbf{p_i}) \tag{9}$$

$$N_1(x) = \vee_{i=1}^n (\exists d, \mathbf{p_i} \bullet N_1^i(x, d, \mathbf{p_i})) \tag{10}$$

$$S(x, y) = \vee_{i=1}^n (\exists p_1, p_2 \cdots, p_{k_i - 1} \bullet \vee_{j=1}^{k_i} N^i(x, d, \mathbf{p_i^j(x)})) \tag{11}$$

where $\mathbf{p_i^j(x)} = (p_1, p_2 \cdots p_{j-1}, x, p_j \cdots p_{k_i - 1})$.

The Generalized Hindsight Lemma assumes first, there is a pointer H, H does not change, so we have the invariant:

$$\phi_h(x) = H = x \wedge \exists d, \mathbf{p_i} \bullet N_1^i(x, d, \mathbf{p_i}) \tag{12}$$

We define the sub-formula, the formula formalize the assertion that there is a heap path from x to y. Note the presence of dangling pointers is not a problem, our model allow loops in the heap.

$$p_r(x, y) = (x = y \wedge \mathbf{emp}) \vee (\exists z \bullet S(x, z) * p_r(z, y)) \tag{13}$$

The concurrent search data structure should also satisfy the following invariant:

$$\phi_{mark}(x) = p_r(H, x) \leftrightarrow N_1(x) \tag{14}$$

In practice the mark field can be omitted. It can be viewed as an auxiliary variable in order to make our formulas simpler.

The *RUA* assumption step invariant can be formalized as, for any computation step (σ, σ'):

$$\delta_{rua}(x, d, d') : \sigma \models N^i(x, d, \mathbf{p_i}) \wedge \sigma' \models N_2^i(x, d', \mathbf{p_i'}) \Rightarrow (\mathbf{p_i'} = \mathbf{p_i}) \tag{15}$$

The above step invariant ensures that once a node become marked, its successor pointers become immutable. It can easily translated to Hoare tuple used in thread modular verification.

The Generalized Hindsight Lemma can be formally stated as:

Lemma 15. *Given a concurrent search structure with nodes of types $T_1, T_2 \cdots T_n$. If it satisfies thread modular state invariants $\exists x \bullet \phi_h(x)$, $\forall x \bullet \phi_{mark}(x)$, and step invariant $\forall x, d, d' \bullet \delta_{rua}(x, d)$ in an execution trace $\sigma_1, \sigma_2, \cdots \sigma_m$. Then if $S(u, v) \wedge N_1(u)$ in σ_i, and $S(v, w)$ in σ_j, then there is $i \le k \le j$, such that $S(v, w) \wedge N_1(v)$.*

Lemma 16. ***Generalized Temporal Backbone Lemma***

Given a concurrent search structure with nodes of types $T_1, T_2 \cdots T_n$. If it satisfies thread modular state invariants $\exists x \bullet \phi_h(x)$, $\forall x \bullet \phi_{mark}(x)$, and step invariant δ_{rua} in an execution trace $T = (\sigma_1, \sigma_2, \cdots \sigma_n)$. Let N be a temporal node path $N = \{(u_0, u_1), (u_1, u_2), \cdots (u_{m-1}, u_m)\}$ goes through subsequence $T_s = \{\sigma_{i_1}, \cdots \sigma_{i_m}\}$, such that (u_0, u_1) is a backbone link in σ_{i_1}. Then there is another subsequence of execution trace $T_s' = \{\sigma_{j_1}, \cdots \sigma_{j_m}\}$ such that for all $1 \le k \le m - 1$, $j_{k-1} \le j_k \le i_k$, and N is a temporal backbone going through T_s'.

Proof. This lemma simply follows for applying the Generalized Hindsight Lemma m times. $\square$

C Definitions

Definition C.1. *A history H is an execution trace containing only invocations and responses. A sequential history is a history where for each invocation, follows by a corresponding response. A partial history H_t of thread t w.r.t history H is the subsequence of H which is invoked by thread t. A history H is called well-formed when for every thread t, H_t is sequential. A sequential specification S_p is a set of sequential histories.*

Definition C.2. *Suppose H is a well formed history, it is linearizable with sequential history H_S, if there is a map τ from operations in H to the same operations in H_S that preserves real time order(Namely, if two operations t_1, t_2 with the response of t_1 is before the invocation of t_2, then $\tau(t_1)$ before $\tau(t_2)$), then H is linearizable w.r.t H_S. If every execution of an algorithm is a linearizable history w.r.t a sequential history in its sequential specification S_p, the algorithm is said to be linearizable w.r.t. S_p.*

D Proofs

In this section, we prove the invariant $\phi(\texttt{Root}, -\infty)$.

We write the following formula:

$$\begin{aligned} Tr_0(x,k,b) = (\exists p_2, k_2 \bullet N_1^I(x,k,b,p_2) * Tr_1(p_2,k_2)) \\ \vee(\exists p_2, k_2 \bullet N_1^I(x,k,p_1,b) * Tr_1(p_1,k_1)) \\ \vee(\exists p_1,k_1,p_2,k_2 \bullet N_1^I(x,k,p_1,p_2) * Tr_0(p_1,k_1,b) * Tr_1(p_2,k_2)) \\ \vee(\exists p_1,k_1,p_2,k_2 \bullet N_1^I(x,k,p_1,p_2) * Tr_1(p_1,k_1) * Tr_1(p_2,k_2,b)) \end{aligned}$$

We have the following important lemmas:

Lemma 17.

$$(\exists k' \bullet Tr_0(x,k,b) * Tr_1(b,k')) \Rightarrow Tr_1(x,k)$$

Proof. We have

$$(\exists k', p_2, k_2 \bullet N_1^I(x,k,b,p_2) * Tr_1(p_2,k_2) * Tr_1(b,k')) \Rightarrow Tr_1(x,k)$$

By using induction we have:

$$(\exists k', p_1, k_1, p_2, k_2 \bullet N_1^I(x,k,p_1,p_2) * Tr_0(p_1,k_1,b) * Tr_1(p_2,k_2) * Tr_1(b,k')) \Rightarrow Tr_1(x,k)$$

$$(\exists k', p_1, k_1, p_2, k_2 \bullet N_1^I(x,k,p_1,p_2) * Tr_1(p_1,k_1) * Tr_1(p_2,k_2,b) * Tr_1(b,k')) \Rightarrow Tr_1(x,k)$$

So combine the cases above, we have

$$(\exists k' \bullet Tr_0(x,k,b) * Tr_1(b,k')) \Rightarrow Tr_1(x,k)$$

□

Lemma 18.

$$(Tr_1(x,k) \wedge (N_1^L(b,k') * \textit{\textbf{true}} \vee \exists p1, p2 \bullet N_1^I(b,k',p1,p2) * \textit{\textbf{true}})) \Rightarrow Tr_0(x,k,b)$$

Proof. Unroll the definition of Tr_1 and use induction. □

```
{emp ∧ Root == null}
Init(){
  {emp ∧ Root == null}
  Root = new Internal(-infty);
  {Root → (false, −∞, _, _)}
  Root.left = new Leaf(-infty);
  {Root → (false, −∞, x, _) ∗ $N^L_1(x, -\infty)$}
  Root.right = new Leaf(+infty);
  {Root → (false, −∞, x, y) ∗ $N^L_1(x, -\infty) * N^L_1(x, +\infty)$}
  {$N^I_1$(Root, −∞, l, r) ∗ $N^L_1(l, -\infty) * N^L_1(r, +\infty)$}
  {$\phi$(Root, −∞)}
}
```

Listing 5: Proof of `Init`

```
{$\phi(\mathtt{Root}, -\infty)$}
bool add1(Key k){
    {$\phi(\mathtt{Root}, -\infty)$}
    while(true){
        {$\phi(\mathtt{Root}, -\infty)$}
        -,p,n := search1(k);
        {$\phi(\mathtt{Root}, -\infty) \wedge (p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \mathtt{true}) \wedge k'_p < k'_n < k$}
        dir = k.compareTo(n.key);
        {$\phi(\mathtt{Root}, -\infty) \wedge (p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \mathtt{true}) \wedge k'_p < k'_n < k \wedge dir = 1$}
        if(dir == 0)
        {$\phi(\mathtt{Root}, -\infty) \wedge (p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \mathtt{true}) \wedge k'_p < k'_n < k \wedge dir = 1$}
          return false;
        {$\phi(\mathtt{Root}, -\infty) \wedge (p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \mathtt{true}) \wedge k'_p < k'_n < k \wedge dir = 1$}
        atomic {
          if(!p->marked && p.isParentOf(n)){
        {$\phi(\mathtt{Root}, -\infty) \wedge (N^I_1(p, k'_p, \_, n) * N^L_1(n, k'_n) * \mathtt{true}) \wedge k'_p < k'_n < k \wedge dir = 1$}
          na = new Leaf(k);
        {$(\exists k_m, m \bullet Tr_0(-\infty, \mathtt{Root}, p) * N^I_1(p, k'_p, m, n) * Tr_1(k_m, m) * Tr_1(n, k'_n) * N^L_1(na, k)) \wedge k'_p < k'_n < k \wedge dir = 1$}
          n1 = new Internal(min(n.key,k));
          {$(\exists k_m, m \bullet Tr_0(-\infty, \mathtt{Root}, p) * N^I_1(p, k'_p, m, n) * Tr_1(k_m, m) * Tr_1(n, k'_n) * N^L_1(na, k) * N^I_1(n1, k'_n, \_, \_))$
         $\wedge k'_p < k'_n < k \wedge dir = 1$}
          n1.setChild(n,na);
         {$(\exists k_m, m \bullet Tr_0(-\infty, \mathtt{Root}, p) * N^I_1(p, k'_p, m, n) * Tr_1(k_m, m) * Tr_1(n, k'_n) * N^L_1(na, k) * N^I_1(n1, k'_n, n, na))$
         $\wedge k'_p < k'_n < k \wedge dir = 1$}
          p.changeChild(n,n1);
        {$(\exists k_m, m \bullet Tr_0(-\infty, \mathtt{Root}, p) * N^I_1(p, k'_p, m, n1) * Tr_1(k_m, m) * Tr_1(n, k'_n) * N^L_1(na, k) * N^I_1(n1, k'_n, n, na))$
        $\wedge k'_p < k'_n < k \wedge dir = 1$}
        {$(\exists k_m, m \bullet Tr_0(-\infty, \mathtt{Root}, p) * Tr_1(p, k'_p)) \wedge k'_p < k'_n < k \wedge dir = 1$}
         return true;
          }
           {$(\exists k_m, m \bullet Tr_0(-\infty, \mathtt{Root}, p) * Tr_1(p, k'_p)) \wedge k'_p < k'_n < k \wedge dir = 1$}
           {$\phi(\mathtt{Root}, -\infty)$}
        }
    {$\phi(\mathtt{Root}, -\infty)$}
```

Listing 6: Proof of `add`

In order to make our formulas shorter and clearer, we separate the verification by the result of the `search` operation. We assume the special `search1` operation is guarded by an assume command which guarantees $k'_{gp} < k'_p < k'_n < k$, `search2` operation guarantees $k'_{gp} < k'_p < k'_n = k$. Other cases are very similar, we omit them. The formulas in the actually proof is the disjunction of formulas of all these cases.

```
${\phi(\texttt{Root}, -\infty)}$
 bool remove2(Key k){
  while(true){
${\phi(\texttt{Root}, -\infty)}$
    gp,p,n := search2(k);
$\{\phi(\texttt{Root}, -\infty) \wedge (gp \mapsto (\_, k'_{gp}, \_, \_) * p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \texttt{true}) \wedge k'_{gp} < k'_p < k'_n = k\}$
  dir = k.compareTo(n.key);
$\{\phi(\texttt{Root}, -\infty) \wedge (gp \mapsto (\_, k'_{gp}, \_, \_) * p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \texttt{true}) \wedge k'_{gp} < k'_p < k'_n < k \wedge dir = 0\}$
   if(dir != 0)
      return false;
$\{\phi(\texttt{Root}, -\infty) \wedge (gp \mapsto (\_, k'_{gp}, \_, \_) * p \mapsto (\_, k'_p, \_, \_) * n \mapsto (\_, k'_n) * \texttt{true}) \wedge k'_{gp} < k'_p < k'_n < k \wedge dir = 0\}$
atomic{
  if(gp.isParentOf(p)&& p.isParentOf(n)&& gp!=marked){
$\{\phi(\texttt{Root}, -\infty) \wedge (N_1^I(gp, k'_{gp}, \_, p) * N_1^I(p, k'_p, \_, n) * N_1^L(n, k) * \texttt{true}) \wedge k'_{gp} < k'_p < k'_n < k \wedge dir = 0\}$
        p.marked = true;
$\{(\exists k_{m1}, m1, k_{m2}, m2\bullet$
$Tr_0(-\infty, \texttt{Root}, gp) * N_1^I(gp, k'_{gp}, m1, p) * Tr_1(k_{m1}, m1) * N_2^I(p, k'_p, m2, n) * Tr_1(k_{m2}, m2) * Tr_1(n, k'_n))$
$\wedge k'_p < k'_n < k \wedge dir = 1\}$
        n.marked = true;
$\{(\exists k_{m1}, m1, k_{m2}, m2\bullet$
$Tr_0(-\infty, \texttt{Root}, gp) * N_1^I(gp, k'_{gp}, m1, p) * Tr_1(k_{m1}, m1) * N_2^I(p, k'_p, m2, n) * Tr_1(k_{m2}, m2) * N_2^L(n, k'_n) * T(n, k'_n))$
$\wedge k'_p < k'_n < k \wedge dir = 1\}$
        gp.changeChild(p,p.getOtherChild(n));
$\{(\exists k_{m1}, m1, k_{m2}, m2\bullet$
$Tr_0(-\infty, \texttt{Root}, gp) * N_1^I(gp, k'_{gp}, m1, m2) * Tr_1(k_{m1}, m1) * N_2^I(p, k'_p, m2, n) * Tr_1(k_{m2}, m2) * N_2^L(n, k'_n) * T(n, k'_n))$
$\wedge k'_p < k'_n < k \wedge dir = 1\}$
$\{(\exists k_{m1}, m1, k_{m2}, m2 \bullet Tr_0(-\infty, \texttt{Root}, gp) * N_1^I(gp, k'_{gp}, m1, m2) * Tr_1(k_{m1}, m1) * Tr_1(k_{m2}, m2))$
$\wedge k'_p < k'_n < k \wedge dir = 1\}$
${\phi(\texttt{Root}, -\infty)}$
        return true;
     }
    }
   }
  }
  ${\phi(\texttt{Root}, -\infty)}$
}
```

Listing 7: Proof of `Remove`