A Constructive Approach for Proving Data Structures' Linearizability

Kfir Lev-Ari, Gregory Chockler, Idit Keidar

A Constructive Approach for Proving Data Structures' Linearizability *

Kfir Lev-Ari[1], Gregory Chockler[2], and Idit Keidar[1]

[1] EE Department, Technion – Israel Institute of Technology, Haifa, Israel
[2] CS Department, Royal Holloway University of London, Egham, UK

Abstract. We present a comprehensive methodology for proving correctness of concurrent data structures. We exemplify our methodology by using it to give a roadmap for proving linearizability of the popular Lazy List implementation of the concurrent set abstraction. Correctness is based on our key theorem, which captures sufficient conditions for linearizability. In contrast to prior work, our conditions are derived directly from the properties of the data structure in sequential runs, without requiring the linearization points to be explicitly identified.

1 Introduction

While writing an efficient concurrent data structure is challenging, proving its correctness properties is usually even more challenging. Our goal is to simplify the task of proving correctness. We present a methodology that offers algorithm designers a constructive way to analyze their data structures, using the same principles that were used to design them in the first place. It is a generic approach for proving handcrafted concurrent data structures' correctness, which can be used for presenting intuitive proofs.

The methodology we present here generalizes our previous work on reads-write concurrency [10], and deals also with concurrency among write operations as well as with any number of update steps per operation (rather than a single update step per operation as in [10]). To do so, we define the new notions of *base point preserving steps*, *commutative steps*, and *critical sequence*. We demonstrate the methodology by proving linearizability of Lazy List [8], as opposed to toy examples in [10].

Our analysis consists of three stages. In the first stage we identify conditions, called *base conditions* [10], which are derived *entirely* by analysis of sequential behavior, i.e., we analyze the algorithm as if it is designed to implement the data structure correctly only in sequential executions. These conditions link states of the data structure with outcomes of operations running on the data structure

* This work was partially supported by the Israeli Science Foundation (ISF), the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by a Royal Society International Exchanges Grant IE130802, and by the Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research Program.

from these states. More precisely, base conditions tell us what needs to be satisfied by a state of the data structure in order for a sequential execution to reach a specific point in an operation from that state. For example, Lazy List's *contains(31)* operation returns *true* if 31 appears in the list. A possible base condition for returning *true* is "there is an element that is reachable from the head of the list and its value is 31". Every state of Lazy List that satisfies this base condition causes *contains(31)* to return *true*.

In the second stage of our analysis we prove the linearization of update operations, (i.e., operations that might modify shared memory). We state two conditions on update operations that together suffice for linearizability. The first is *commutativity* of steps taken by *concurrent updates*. The idea here is that if two operations' writes to shared memory are interleaved, then these operations must be independent. Such behavior is enforced by standard synchronization approaches, e.g., two-phase locking. The second condition requires that some state reached during the execution of the update operation satisfy base conditions of all the update operation's writes. For example, the update steps of an *add(7)* operation in Lazy List depend on the predecessor and successor of 7 in the list. Indeed, Lazy List's *add(7)* operation writes to shared memory only after locking these nodes, which prevent concurrent operations from changing the two nodes that satisfy the base conditions of *add(7)*'s steps.

In the third stage we consider the relationship between update operations and read-only operations. We first require each update operation to have at most one point in which it changes the state of the data structure in a way that "affects" read-only operations. We capture the meaning of "affecting" read-only operations using base conditions. Intuitively, if an update operation has a point in which it changes something that causes the state to satisfy a base condition of a read-only operation, then we know that this point defines the outcome of the read-only operation. For example, Lazy List's *remove(3)* operation first *marks* the node holding 3, and then detaches it from its predecessor. Since *contains* treats marked nodes as deleted, the second update step does not affect *contains*.

In addition, we require that each read-only operation has a state in the course of its execution that satisfies its base condition. In order to show that such a state exists, we need to examine how the steps that we have identified in the update operations affect the base conditions of the read-only operations. For example, in Lazy List, *contains(9)* relies on the fact that if a node holding 9 is reachable from the head of the list, then there was some concurrent state in which a node holding 9 was part of the list. We need to make sure that the update operations support this assumption.

The remainder of this paper is organized as follows: Section 2 provides formal preliminaries. We formally present and illustrate the analysis approach in Section 3. We state and prove our main theorem in Section 4. Then, we demonstrate how base point analysis can be used as a roadmap for proving linearizability of Lazy List in Section 5. Section 6 concludes the paper.

2 Preliminaries

We extend here the model and notions we defined in [10]. Generally speaking, we consider a standard shared memory model [1] with one refinement, which is differentiating between local and shared state, as needed for our discussion.

Each process performs a sequence of operations on shared data structures implemented using a set $X = \{x_1, x_2, ...\}$ of shared variables. The shared variables support atomic operations, such as read, write, CAS, etc. A *data structure implementation* (algorithm) is defined as follows:

- A set $\mathcal{S}$ of *shared states*, some of which are *initial*, where $\underline{s} \in \mathcal{S}$ is a mapping assigning a value to each shared variable.
- A set of operations representing methods and their parameters (e.g., *add*(7) is an operation). Each *operation op* is a state machine defined by: A set of local states $\mathcal{L}_{op}$, which are given as mappings l of values to local variables; and a deterministic transition function $\tau_{op}(\mathcal{L}_{op} \times \mathcal{S}) \rightarrow \mathit{Steps} \times \mathcal{L}_{op} \times \mathcal{S}$ where *Steps* are transition labels, such as *invoke*, *return(v)*, $a \leftarrow$ *read*(x_i), *write*(x_i,v), *CAS*(x_i,v_{old},v_{new}), etc.

Invoke and return steps interact with the application, while read and write steps interact with the shared memory and are defined for every shared state. In addition, the implementation may use synchronization primitives (locks, barriers), which constrain the scheduling of ensuing steps, i.e., they restrict the possible *executions*, as we shortly define.

For a transition $\tau(l, \underline{s}) = \langle \mathit{step}, l', \underline{s'} \rangle$, l determines the step. If *step* is an invoke or return, then l' is uniquely defined by l. Otherwise, l' is defined by l and potentially $\underline{s}$. For invoke, return, read and synchronization steps, $\underline{s} = \underline{s'}$. If any of the variables is assigned a different value in $\underline{s}$ than in $\underline{s'}$, then the step is called an *update step*.

A state consists of a local state l and a shared state $\underline{s}$. We omit either the shared or the local component of the state if its content is immaterial to the discussion. A *sequential execution of an operation* from a shared state $\underline{s_i} \in \mathcal{S}$ is a sequence of transitions of the form:

$$\underset{\underline{s_i}}{\overset{\perp}{}},\ \mathit{invoke},\ \underset{\underline{s_i}}{\overset{l_1}{}},\ \mathit{step}_1,\ \underset{\underline{s_{i+1}}}{\overset{l_2}{}},\ \mathit{step}_2,\ \dots,\ \underset{\underline{s_j}}{\overset{l_k}{}},\ \mathit{return}_k,\ \underset{\underline{s_j}}{\overset{\perp}{}},$$

where $\perp$ is the operation's initial local sate and $\tau(l_m,\ \underline{s_n}) = \langle \mathit{step}_m,\ l_{m+1},\ \underline{s_{n+1}} \rangle$. The first step is invoke and the last step is a return step.
A *sequential execution of a data structure* is a (finite or infinite) sequence μ:

$$\mu = \underset{\underline{s_1}}{\overset{\perp}{}},\ O_1,\ \underset{\underline{s_2}}{\overset{\perp}{}},\ O_2,\ \dots,$$

where $\underline{s_1} \in \mathcal{S}_0$ and every $\underset{\underline{s_j}}{\overset{\perp}{}},\ O_j,\ \underset{\underline{s_{j+1}}}{\overset{\perp}{}}$ in μ is a sequential execution of some operation. If μ is finite, it can end after an operation or during an operation. In

the latter case, we say that the last operation is *pending* in μ. Note that in a sequential execution there can be at most one pending operation.

A *concurrent execution fragment of a data structure* is a sequence of interleaved states and steps of different operations, where each state consists of a set of local states $\{l_i, ..., l_j\}$ and a shared state $\underline{s_k}$, where every l_i is a local state of a *pending* operation, which is an operation that has not returned yet. A *concurrent execution of a data structure* is a concurrent execution fragment that starts from an initial shared state and an empty set of local states. In order to simplify the discussion of initialization, we assume that every execution begins with a dummy (initializing) update operation that does not overlap any other operation. A state $\underline{s'}$ is *reachable from a state* $\underline{s}$ if there exists an execution fragment that starts at $\underline{s}$ and ends at $\underline{s'}$. A state is *reachable* if it is reachable from an initial state.

An operation for which there exists an execution in which it perform update steps is called *update operation*. Otherwise, it is called a *read-only operation*.

A data structure's correctness in sequential executions is defined using a *sequential specification*, which is a set of its allowed sequential executions. A *linearization* of execution μ is a sequential execution μ_l, such that:

- Every operation that is not invoked in μ is not invoked in μ_l.
- Every operation that returns in μ returns also in μ_l and with the same return value.
- μ_l belongs to the data structure's sequential specification.
- The order between non-interleaved operations in μ and μ_l is identical.

A data structure is *linearizable* [9] if each of its executions has a linearization.

3 Base Point Analysis

In this section we present key definitions for analyzing and proving correctness using what we call *base point analysis*. We illustrate the notions we define using Lazy List [8], whose pseudo code appears in Algorithm 1.

We start by defining *base conditions* [10]. A *base condition* establishes some link between the local state that an operation reaches and the shared variables the operation has read before reaching this state. It is given as a predicate Φ over shared variable assignments. Formally:

Definition 1 (Base Condition). Let l be a local state of an operation *op*. A predicate Φ over shared variables is a *base condition* for l if every sequential execution of *op* starting from a shared state $\underline{s}$ such that $\Phi(\underline{s})$ is true, reaches l.

For completeness, we define a base condition for $step_i$ in an execution μ to be a base condition of the local state that precedes $step_i$ in μ. For example, consider an execution of Lazy List's *contains(31)* operation that returns *true*. A possible base condition for that return step is ϕ : "there is an unmarked node in which $key = 31$, and that node is reachable from the head of the list". Every sequential

▷ $\Phi_{loc}(\underline{s}, n_1, n_2, e)$: $(\texttt{Head} \overset{*}{\Rightarrow} n_1) \wedge (n_1.next = n_2) \wedge \neg n_1.marked \wedge \neg n_2.marked \wedge (n_1.val < e) \wedge (e \leq n_2.val)$

```
1  Function contains(e)
2      c ← read(Head)
3      while read(c.val) < e
4          c ← read(c.next)
5      ▷ Φ_c : (Head ⇒* c) ∧ (c.val ≥ e)
         ∧ (∄n : (Head ⇒* n) ∧
         (e ≤ n.val < c.val))
6      if read(c.marked) ∨ read(c.val) ≠ e
7          ▷ Φ_c ∧ (c.marked ∨ c.val ≠ e)
8          return false
9      else
10         ▷ Φ_c ∧ (c.val = e)
11         return true

12 Function add(e)
13     ⟨n1, n2⟩ ← locate(e)
14     ▷ Φ_loc(s, n1, n2, e)
15     if read(n2.val) ≠ e
16         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val ≠ e)
17         write(n3, new Node(e, n2))
18         write(n1.next, n3)
19         unlock(n1)
20         unlock(n2)
21         return true
22     else
23         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val = e)
24         unlock(n1)
25         unlock(n2)
26         return false

27 Function locate(e)
28     while true
29         n1 ← read(Head)
30         n2 ← read(n1.next)
31         while read(n2.val) < e
32             n1 ← n2
33             n2 ← read(n2.next)
34         lock(n1)
35         lock(n2)
36         if read(n1.marked) = false ∧
37         read(n2.marked) = false ∧
38         read(n1.next) = n2
39             return ⟨n1, n2⟩
40         else
41             unlock(n1, n2)

42 Function remove(e)
43     ⟨n1, n2⟩ ← locate(e)
44     ▷ Φ_loc(s, n1, n2, e)
45     if read(n2.val) = e
46         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val = e)
47         write(n2.marked, true)
48         write(n1.next, n2.next)
49         unlock(n1)
50         unlock(n2)
51         return true
52     else
53         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val ≠ e)
54         unlock(n1)
55         unlock(n2)
56         return false
```

Algorithm 1: Lazy List. Base conditions are listed as comments, using Φ_{loc} defined above the functions.

execution of *contains(31)* from a shared state that satisfied ϕ reaches the same *return true* step. Base conditions for all of Lazy List's update and return steps are annotated in Algorithm 1, and are discussed in detail in Section 5.1 below.

For a given base condition, the notion of *base point* [10] links the local state that has base condition Φ to a shared state $\underline{s}$ where $\Phi(\underline{s})$ holds.

Definition 2 (Base Point). Let *op* be an operation in an execution μ, and let Φ_t be a base condition for the local state at point t in μ. An execution prefix of *op* in μ has a *base point* for point t with Φ_t, if there exists a shared state $\underline{s}$ in μ, called a *base point of* t, such that $\Phi_t(\underline{s})$ holds.

Note that together with Definition 1, the existence of a base point $\underline{s}$ for point t implies that the step or local state at point t in operation *op* is reachable from $\underline{s}$ in a sequential run of *op* starting from $\underline{s}$. In Figure 1 we depict two states of Lazy List: $\underline{s_1}$ is a base point for a *return true* step of *contains(7)*, whereas $\underline{s_2}$ is not.

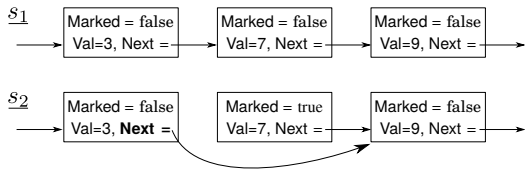

Figure 1: Two states of Lazy List (Algorithm 1): $\underline{s_1}$ is a base point for *contains(7)*'s *return true* step, as it satisfies the base condition "there is a node that is reachable from the head of the list, and its value is 7". The shared state $\underline{s_2}$ is not a base point of this step, since there is no sequential execution of *contains(7)* from $\underline{s_2}$ in which this step is reached.

Let $\underline{s_0}$ and $\underline{s_1}$ be two shared states, and let $\underline{s_0}$, st, $\underline{s_1}$ be an execution fragment. We call $\underline{s_0}$ the *pre-state* of step st, and $\underline{s_1}$ the *post-state* of st.

We now define *base point preserving* steps, which are steps under which base conditions are invariant.

Definition 3 (Base Point Preserving Step). A step st is base point preserving with respect to an operation *op* if for any update or return step b of *op*, for any concurrently reachable pre-state of st, st's pre-state is a base point of b if and only if st's post-state is a base point of b.

An example of a base point preserving step is illustrated in Figure 2. In this example, the second write step in Lazy List's *remove* operation is base point preserving for *contains*. Intuitively, since *contains* treats marked nodes as removed, the same return step is reached regardless whether the marked node is detached from the list or reachable from the head of the list.

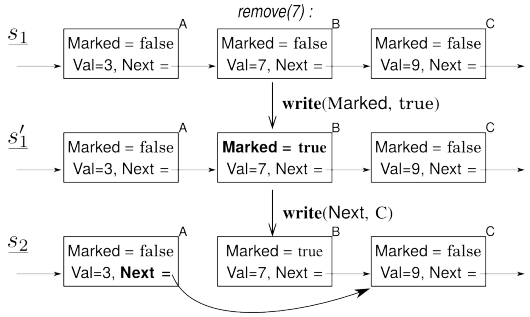

Figure 2: Operation *remove(7)* of Lazy List has two write steps. In the first, *marked* is set to *true*. In the second, the *next* field of the node holding 3 is set to point to the node holding 9. If a concurrent *contains(7)* operation sequentially executes from state $\underline{s_1}$, it returns true. If we execute *contains(7)* from $\underline{s'_1}$, i.e., after *remove(7)*'s first write, *contains* sees that 7 is marked, and therefore returns false. If we execute *contains* from state $\underline{s_2}$, after *remove(7)*'s second write, *contains* does not see B because it is no longer reachable from the head of the list, and also returns false. The second write does not affect the return step, since in both cases it returns false.

4 Linearizability using Base Point Analysis

We use the notions introduced in Section 3 to define sufficient conditions for linearizability. In Section 4.1 we define conditions for update operations, and in Section 4.2 we define an additional condition on read-only operations, and show that together, our conditions imply linearizability.

4.1 Update Operations

We begin by defining the *commutativity* of steps.

Definition 4 (Commutative Steps). Consider an execution μ of a data structure ds that includes the fragment $a, \underline{s_1}, b, \underline{s_2}$. We say that steps a and b *commute* if $a, \underline{s_1}, b, \underline{s_2}$ in μ can be replaced with $b, \underline{s'_1}, a, \underline{s_2}$, so that the resulting sequence μ' is a valid execution of ds.

We now observe that if two update steps commute, then their resulting shared state is identical for any ordering of these steps along with interleaved read steps.

Observation 1 *Let $\underline{s_0}, a, \underline{s_1}, b, \underline{s_2}$ be an execution fragment of two update steps a and b that commute, then $\underline{s_2}$ is the final shared state in any execution fragment that starts from $\underline{s_0}$ and consists of a, b and any number of read steps (for any possible ordering of steps).*

We are not interested in commutativity of all steps, but rather of "critical" steps that modify shared memory or determine return values. This is captured by the following notion:

Definition 5 (Critical Sequence). The *critical sequence* of an update operation op in execution μ is the subsequence of op's steps from its first to its last update step; if op takes no update steps in μ, then the critical sequence consists solely of its last read.

For example, if in Lazy List $op_1 = add(2)$ and $op_2 = add(47)$ concurrently add items in disjoint parts of the list, then all steps in op_1's critical sequence commute with all those in op_2's critical sequence. The same is not true for list traversal steps taken before the critical sequence, since op_2 may or may not traverse a node holding 2, depending on the interleaving of op_1 and op_2's steps. In general, Lazy List uses locks to ensure that the critical steps of two operations overlap only if these operations' respective steps commute. This is our first condition for linearizability of update operations.

Our second requirement from update operations is that each critical sequence begin its execution from a base point of all the operation's update and return steps. Together, we have:

Definition 6 (Linearizable Update Operations). A data structure ds has linearizable update operations if for every execution μ, for every update operation $uo_i \in \mu$:

1. $\forall uo_j \in \mu, i \neq j$, if the critical sequence of uo_j interleaves with the critical sequence of uo_i in μ, then all of uo_i's steps in its critical sequence commute with all of the steps in uo_j's critical sequence, and all the update steps of uo_i and uo_j are base point preserving for uo_j and uo_i respectively.
2. The pre-state of uo_i's critical sequence is a base point for all of uo_i's update and return steps, and moreover, if uo_i is complete in μ, then this state is not a base point for any other possible update step of uo_i.

To satisfy these conditions, before its critical sequence, an update operation takes actions to guarantee that the pre-state of its first update will be a base point for the operation's update and return steps, as depicted in Figure 3. For example, any algorithm that follows the two-phase locking protocol [2] satisfies these conditions: operations perform concurrent modifications only if they gain disjoint locks, which means that their steps commute. And in addition, once all locks are obtained by an operation, the shared state is a base point for all of its ensuing steps, i.e., for its critical sequence.

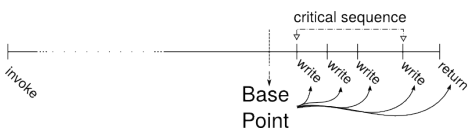

Figure 3: The structure of update operations. The steps before the critical sequence ensure that the pre-state of the first update step is a base point for all of the update and return steps.

We now show that every execution that has linearizable update operations and no read-only operations is linearizable.

Lemma 1. *Let μ be an execution consisting of update operations of some data structure that has linearizable update operations. Let μ' be a sequential execution of all the operations in μ starting from the same initial state as μ such that if some operation op_1's critical sequence ends before the critical sequence of another operation op_2 begins in μ, then op_1 precedes op_2 in μ'. Then μ' is a linearization of μ.*

Proof. By construction, μ' includes only invoke steps from μ, and every two operations that are not interleaved in μ occur in the same order in μ and μ'. It remains to show that every operation has the same return step in μ and μ'.

Denote by μ'_i the prefix of μ' consisting of i operations, and by μ_i the subsequence of μ consisting of the steps of the same i operations. Denote by op_i the i^{th} operation in μ'.

We prove by induction on i that μ'_i is a linearization of μ_i and both executions end in the same final state. As noted above, for linearizability, it suffices to show that all operations that return in both μ'_i and μ_i return the same value.

The first operation in both μ and μ' is a dummy initialization, which returns before all other operations are invoked. Hence, $\mu_1 = \mu'_1$, and their final states are identical.

Assume now that μ'_i is a linearization of μ_i and their final states are the same. The critical sequence of op_{i+1} in μ_{i+1} overlaps the critical sequences of the last zero or more operations in μ_i. We need to show that (1) the execution of op_{i+1} that overlaps these steps in μ_{i+1} yields the same return value and the same final state as a sequential execution of op_{i+1} from the final state of μ_i; and (2) the return values of the operations that op_{i+1} is interleaved with in μ_{i+1} are unaffected by the addition of op_{i+1}'s steps.

(1) By definition 6, the pre-state $\underline{p}$ of op_{i+1}'s critical sequence in μ_{i+1} is a base point for op_{i+1}'s update and return steps. Note that $\underline{p}$ occurs in μ_{i+1} before any update step of op_{i+1}, and thus it also occurs in μ_i. Thus, the same $\underline{p}$ occurs also in μ_i. All the update steps after $\underline{p}$ in μ_{i+1} belong to operations that have interleaved critical sequences with op_{i+1} in μ_{i+1}, and therefore by definition 6 their update steps are base point preserving for op_{i+1}. These are the update steps that occur after $\underline{p}$ in μ_i, and so the final state of μ_i is a base point for the update and return steps that op_{i+1} takes in μ_{i+1}.

By the induction hypothesis, the last states of μ_i and μ'_i are identical, and we conclude that op_{i+1} has the same update and return steps in μ_{i+1} and μ'_{i+1}.

In addition, the final states of μ_{i+1} and μ'_{i+1} occur at the end of execution fragments that consist of the same update steps, s.t. if two update steps have different orders in μ_{i+1} and in μ'_{i+1} then they are commute. By Observation 1 we conclude that the last states of μ_{i+1} and μ'_{i+1} are identical.

(2) If an update step of op_{i+1} occurs in μ_{i+1} before operation op_j's return step, then op_{i+1} has an interleaved critical sequence with op_j. This means that all of op_{i+1}'s update steps are base point preserving for op_j. Thus, the same

base points are reached before op_j's critical sequences in μ_i and in μ_{i+1}. By definition 6, op_j takes the same update and return steps in μ_i and μ_{i+1}. □

4.2 Read-Only Operations

We state two conditions that together ensure linearizability of read-only operations. First, each read-only operation ro should have a base point for its return step, which can be either a post-state of some step of operation that is concurrent to ro, or the pre-state of ro's invoke step. Second, update operations should have at most one step that is not base point preserving for read-only operations.

In Theorem 2 we present a sufficient condition for linearizability. Intuitively, we want the linearizable update operations to satisfy two conditions: (1) the read-only operations should see the update operations as a sequence of single steps that mutate the shared state. To express this relation we use the base point preserving property; and (2) the update operations should guarantee the correctness of the returned values of the read-only operation, as expressed by the return steps' base conditions.

Theorem 2. *Let ds be a data structure that has linearizable update operations. If ds satisfies the following conditions, it is linearizable:*

1. *Every update operation of ds has at most one step that is not base point preserving with respect to all read-only operations.*
2. *For every execution μ, for every complete read-only operation $ro \in \mu$, there exists in μ a shared state $\underline{s}$ between the pre-state of ro's invoke step and the pre-state of ro's return step (both inclusive) that is a base-point for ro's return step.*

Proof. For a given execution μ^-, let μ be an execution that is identical to μ^- with the addition that all pending operations in μ^- are allowed to complete. Note that μ also has linearizable update operations. We now show that μ has a linearization, and therefore μ^- has a linearization.

We build a sequential execution μ_{seq} as follows:

1. μ_{seq} starts from the same shared state as μ.
2. We sequentially execute all the update operations that takes steps of their critical sequence in μ in the order of their steps that are not base point preserving for read-only operations, (or the last read step in case all steps are base point preserving). We denote this sequence of steps by $\{ord_i\}$. The update operation that performs ord_i in μ is denoted uo_i.
3. Each read-only operation ro of μ is executed in μ_{seq} after an update operation uo_i such that the post-state of ord_i in μ is a base point for ro, and is either concurrent to ro or the latest step in $\{ord_i\}$ that precedes ro's invoke step. Such a step exists since (1) by our assumption, ro has a base point between its invoke step's pre-state and its return step's pre-state; and (2) every step that is not in $\{ord_i\}$ is base point preserving for ro.

4. The order in μ_{seq} between non-interleaved read-only operations that share the same base point follows their order in μ. The order between interleaved read-only operations that are executed in μ_{seq} from the same base point is arbitrary.

Now, by Lemma 1, the sequence of update operations in μ_{seq} is a linearization of the sequence of update operations in μ.

Therefore we only need to prove that the order between the read-only operations and other operations that are not interleaved in μ is identical in μ_{seq} and μ, and that each read-only operation has the same return step in both executions.

We observe that:

1. In μ and μ_{seq} the steps of $\{ord_i\}$ appear in the same order, and in both executions each read-only operation is either executed after the same ord_i in both, or is executed concurrently to ord_i in μ and immediately after uo_i in μ_{seq}.
2. Each shared state satisfies the same base conditions since the update steps that appear in a different order in μ and μ_{seq} commute.

Therefore each post-state of ord_i remains a base point in μ_{seq} for the same read-only operations that it was in μ, and thus each read-only operation reaches the same return step as in μ.

Assume towards contradiction that two read-only operations ro_1 and ro_2 have a different order in μ and μ_{seq}, and w.l.o.g. ro_1 precedes ro_2 in μ, and ro_2 precedes ro_1 in μ_{seq}.

Let uo_1 be the update operation that precedes ro_1 in μ_{seq}, and uo_2 be the update operation that precedes ro_2 in μ_{seq}. $uo_2 \neq uo_1$, otherwise ro_1 and ro_2 had the same base point and their execution order was identical to their order in μ. Since ro_2 precedes ro_1 in μ_{seq}, we conclude that ord_2 occurs before ord_1 in μ. ord_1 takes place in μ as last as one step before uo_1's return step. Therefore ord_2 must appear somewhere before ro_1's return step. But ro_1 precedes ro_2 in μ, meaning that ord_2 is not the latest steps of ord that precedes ro_2's invoke step, in contradiction. □

5 Roadmap for Proving Linearizability

We now prove that Lazy List (Algorithm 1) satisfies the requirements of Theorem 2, implying that it is linearizable. We demonstrate the three stages of our roadmap for proofing linearizability using base point analysis.

5.1 Stage I: Base Conditions

We begin by identifying base conditions for the operations' update and return steps. The base conditions are annotated in comments in Algorithm 1. To do so, we examine the possible sequential executions of each operation.

Add & Remove Let $Head \stackrel{*}{\Rightarrow} n$ denote that there is a set of shared variables $\{Head, x_1, ..., x_k\}$ such that $Head.next = x_1 \wedge x_1.next = x_2 \wedge ... \wedge x_k = n$, i.e., that there exists some path from the shared variable *Head* to n. Let $\Phi_{loc}(\underline{s}, n_1, n_2, e)$ be the predicate indicating that in the shared state $\underline{s}$, the place of the key e in the list is immediately after the node n_1, and at or just before the node n_2:

$\Phi_{loc}(\underline{s}, n_1, n_2, e) : Head \stackrel{*}{\Rightarrow} n_1 \wedge n_1.next = n_2 \wedge \neg n_1.marked \wedge \neg n_2.marked \wedge n_1.val < e \wedge \ e \leq n_2.val.$

Observation 2 *$\Phi_{loc}(\underline{s}, n_1, n_2, e)$ is a base condition for the local state of add(e) (remove(e)) after line 14 (resp., 44).*

Now, $\Phi_{loc}(\underline{s}, n_1, n_2, e) \wedge n_2.val \neq e$ is a base condition for *add*'s write and *return true* steps and *removes*'s *return false* step. And a base condition for *add*'s *return false* step and *remove*'s write and *return true* steps is $\Phi_{loc}(\underline{s}, n_1, n_2, e) \wedge n_2.val = e$.

Contains First, we define the following predicate:

$\Phi_c : \ Head \stackrel{*}{\Rightarrow} c \wedge \ c.val \ \geq \ e \ \wedge (\nexists \, n \ : \ Head \stackrel{*}{\Rightarrow} \ n \ \wedge e \leq n.val < c.val) \ .$

In a shared state satisfying Φ_c, c is the node with the smallest value greater than or equal to e in the list. The base condition for *contains*'s *return true* step is $\Phi_c \wedge c.val = e$, and the base condition for *return false* is the predicate $\Phi_c \wedge (c.marked \vee c.val \neq e)$.

These predicates are base conditions since every sequential execution from a shared state satisfying them reaches the same return step, i.e., if c is the node in the list with the smallest value that is greater than or equal to e and is reachable from the head of the list, then after traversing the list and reaching it, the return step is determined according to its value.

5.2 Stage II: Linearizability of Update Operations

We next prove that Lazy List has linearizable update operations. Using Definition 6, it suffices to show the following: (1) each update operation has a base point for its update and return steps, (2) each critical sequence commutes with interleaved critical sequences, and (3) the update steps are base point preserving for operations with interleaved critical sequences.

Base Points for Update and Return Steps

Proof Sketch First we claim that in every execution of an *add* (*remove*) operation, line 10 (37, respectively), is a base point for all the operation's update and return steps.

Claim 1. *Consider the shared state $\underline{s}$ immediately after line 14 (44) of an execution of add(e) (remove(e)). Then $\Phi(\underline{s}, n_1, n_2, e)$ is true.*

Claim 1 can be proven by induction on the steps of an execution. Intuitively, the idea is to show by induction that the list is sorted, and that in each *add* (*remove*) operation, *locate* locks the two nodes and verifies that they are unmarked, and so no other operation can change them and they remain reachable from the head of the list and connected to each other. Formal proofs of this claim were given in [11, 13].

Based on Claim 1 and the observation that after line 14 (44) of an execution of *add(e)* (*remove(e)*) the value of $n_2.val$ persists until n_2 is unlocked, we conclude that the shared state after *locate* returns is a base point for update operations' update and return steps. Since the locked nodes cannot be modified by concurrent operations, the pre-state of the first update step is also a base point for the same steps. In case the update operation has no update steps, the same holds for the last read step.

Commutative and Base Point Preserving Steps

Proof Sketch We now show that the steps of update operations that have interleaved updates are commutative, and that the update steps are base point preserving. Specifically, we examine the steps between the first update step and the last one (or just the last read step in case of an update operation that does no have update steps).

In order to add a key to the list, an update operation locks the predecessor and successor of the new node. For removing a node from the list, the update operation locks the node and it predecessor. This means that every update operation locks the nodes that it changes and the nodes that it relies upon before it verifies its steps' base point. Thus, update operations have concurrent critical sequence only if they access different nodes. Therefore their steps commute, and are base point preserving for one another.

5.3 Stage III: Linearizability of Read-Only Operations

The final stage in our proof is to show the conditions stated in Theorem 2 hold for each read-only operation.

Single Non-Preserving Step per Update Operation First we show that every update operation of Lazy List has at most one step that is not base point preserving for all read-only operations.

Proof Sketch We only need to consider update steps, since every other step in *add* and *remove* does not modify the shared memory, and therefore does not affect any base condition of *contains*. There are two update steps in an operation. In *add*, the first update step allocates a new (unreachable) node. Nodes that are not reachable from the head of the list do not affect any base condition. Therefore, only the second step, the one that changes the list, is not base point preserving for *contains*.

In *remove*, the first update step marks the removed node, and the second makes the node unreachable from the head of the list. Since marked nodes are treated in every base condition of *contains* as if they are already detached from the list, the second update step does not change the truth value of the base condition of contains. More precisely, if we compare the second update step's pre-state to its post-state, they both satisfy the same base conditions of *contains*'s return steps.

Concurrent Base Points Last, we show that in every execution of *contains*, the return step of *contains* has a base point, and that base point occurs between the pre-state of *contains*'s invoke step and the pre-state of *contains*'s return step.

Proof Sketch When *add* inserts a new value to the list, it locks the predecessor node n and the successor m, and verifies that n and m are not marked and that $n.next = m$.

Since n or m cannot be removed as long as they are locked, and since nodes are removed only when their predecessor is also locked, new nodes are not added to detached parts of the list. This means that every node encountered during a traversal of the list was reachable from the head at some point.

In addition, if *add* inserts a value e, it satisfies $n.val < e < m.val$, since n and m are locked, and no value other than e is inserted between them before e is added (this can be proven by induction on executions).

The execution of *contains(e)* reaches line 6 only after it traverses the list from its head and reaches the first node c whose value v satisfies $e \leq v$. Thus, there is some concurrent shared state $\underline{s}$ that occurs after the invocation of *contains(e)* in which c is unmarked and reachable from the head of the list. State $\underline{s}$ is a base-point of *contains(e)*'s return step.

6 Discussion

We introduced a constructive methodology for proving correctness of concurrent data structures and exemplified it with a popular data structure. Our methodology outlines a roadmap for proving correctness. While we have exemplified its use for writing semi-formal proofs, we believe it can be used at any level of formalism, from informal correctness arguments to formal verification. In particular, our framework has the potential to simplify the proof structure employed by existing formal methodologies for proving linearizability [3–7, 11, 12], thus making them more accessible to practitioners.

Currently, using our methodology involves manually identifying base conditions, commuting steps, and base point preserving steps. It would be interesting to create tools for suggesting a base condition for each local state, and identifying the interesting steps in update operations using either static or dynamic analysis.

Acknowledgements

We thank Naama Kraus, Noam Rinetzky and the anonymous reviewers for helpful comments and suggestions.

References

1. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons (2004)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1987)
3. Chockler, G., Lynch, N., Mitra, S., Tauber, J.: Proving atomicity: An assertional approach. In: Proceedings of the 19th International Conference on Distributed Computing. pp. 152–168. DISC'05, Springer-Verlag, Berlin, Heidelberg (2005)
4. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: In 18th CAV. pp. 475–488. Springer (2006)
5. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings. pp. 323–337 (2011)
6. Dongol, B., Derrick, J.: Proving linearisability via coarse-grained abstraction. CoRR abs/1212.5116 (2012)
7. Guerraoui, R., Vukolic, M.: A scalable and oblivious atomicity assertion. In: van Breugel, F., Chechik, M. (eds.) CONCUR. Lecture Notes in Computer Science, vol. 5201, pp. 52–66. Springer (2008)
8. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Proceedings of the 9th International Conference on Principles of Distributed Systems. pp. 3–16. OPODIS'05, Springer-Verlag, Berlin, Heidelberg (2006)
9. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
10. Lev-Ari, K., Chockler, G., Keidar, I.: On correctness of data structures under reads-write concurrency. In: Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings. pp. 273–287 (2014)
11. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. pp. 85–94. PODC '10, ACM, New York, NY, USA (2010)
12. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 129–136. PPoPP '06, ACM, New York, NY, USA (2006)
13. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: A safety proof of a lazy concurrent list-based set implementation. Tech. Rep. UCAM-CL-TR-659, University of Cambridge, Computer Laboratory (jan 2006)