Order out of Chaos: Proving Linearizability Using Local Views

Yotam M. Y. Feldman
Tel Aviv University, Israel

Constantin Enea
IRIF, Univ. Paris Diderot & CNRS, France

Adam Morrison
Tel Aviv University, Israel

Noam Rinetzky
Tel Aviv University, Israel

Sharon Shoham
Tel Aviv University, Israel

Abstract

Proving the linearizability of highly concurrent data structures, such as those using optimistic concurrency control, is a challenging task. The main difficulty is in reasoning about the view of the memory obtained by the threads, because as they execute, threads observe different fragments of memory from different points in time. Until today, every linearizability proof has tackled this challenge from scratch.

We present a unifying proof argument for the correctness of unsynchronized traversals, and apply it to prove the linearizability of several highly concurrent search data structures, including an optimistic self-balancing binary search tree, the Lazy List and a lock-free skip list. Our framework harnesses *sequential reasoning* about the view of a thread, considering the thread as if it traverses the data structure without interference from other operations. Our key contribution is showing that properties of reachability along search paths can be deduced for concurrent traversals from such interference-free traversals, when certain intuitive conditions are met. Basing the correctness of traversals on such *local view arguments* greatly simplifies linearizability proofs. At the heart of our result lies a notion of *order on the memory*, corresponding to the order in which locations in memory are read by the threads, which guarantees a certain notion of consistency between the view of the thread and the actual memory.

To apply our framework, the user proves that the data structure satisfies two conditions: (1) acyclicity of the order on memory, even when it is considered across intermediate memory states, and (2) preservation of search paths to locations modified by interfering writes. Establishing the conditions, as well as the full linearizability proof utilizing our proof argument, reduces to simple concurrent reasoning. The result is a clear and comprehensible correctness proof, and elucidates common patterns underlying several existing data structures.

2012 ACM Subject Classification Computing methodologies → Shared memory algorithms, Program reasoning → Program verification

Keywords and phrases concurrency and synchronization, concurrent data structures, lineariazability, optimistic concurrency control, verification and formal methods

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.23

Related Version An extended version appears in https://arxiv.org/abs/1805.03992.

1 Introduction

Concurrent data structures must minimize synchronization to obtain high performance [16, 28]. Many concurrent search data structures therefore use *optimistic* designs, which search the data structure without locking or otherwise writing to memory, and write to shared memory only when modifying the data structure. Thus, in these designs, operations that do not modify the same nodes do not synchronize with each other; in particular, searches can run in parallel, allowing for high performance and scalability. Optimistic designs are now common in concurrent search trees [3, 10, 11, 14, 17, 19, 29, 37, 42], skip lists [13, 21, 27], and lists/hash tables [23, 24, 36, 46].

A major challenge in developing an optimistic search data structure is proving *linearizability* [26], i.e., that every operation appears to take effect atomically at some point in time during its execution. Usually, the key difficulty is proving properties of unsynchronized searches [38, 33, 49, 28], as they can observe an *inconsistent* state of the data structure—for example, due to observing only some of the writes performed by an update operation, or only some update operations but not others. Arguing about such searches requires tricky *concurrent reasoning* about the possible interleaving of reads and writes of the operations. Today, every new linearizability proof tackles these problems from scratch, leading to long and complex proofs.

Our approach: local view arguments. This paper presents a unifying proof argument for proving linearizability of concurrent data structures with unsynchronized searches that replaces the difficult concurrent reasoning described above with *sequential reasoning* about a search, which does not consider interference from other operations. Our main contribution is a framework for establishing properties of an unsynchronized search in a concurrent execution by reasoning *only* about its *local view*—the (potentially inconsistent) picture of memory it observes as it traverses the data structure. We refer to such proofs as *local view arguments*. We show that under two (widely-applicable) conditions listed below, the existence of a path to the searched node in the local view, deduced with sequential reasoning, also holds at some point during the *actual* (concurrent) execution of the traversal. (This includes the case of non-existence of a key indicated by a path to `null`.) Such *reachability* properties are typically key to the linearizability proofs of many prominent concurrent search data structures with unsynchronized searches [16]. Once these properties are established, the rest of the linearizability proof requires only simple concurrent reasoning.

Applying a local view argument requires establishing two conditions: (i) *temporal acyclicity*, which states that the search follows an *order* on the memory that is *acyclic* across intermediate states throughout the concurrent execution; and (ii) *preservation*, which states that whenever a node x is changed, if it was on a search path for some key k in the past, then it is also on such a search path at the time of the change. Although these conditions refer to concurrent executions, proving them for the data structures we consider is straightforward.

More generally, these conditions can be established with inductive proofs that are simplified by relying on the very same traversal properties obtained with the local view argument. This seemingly circular reasoning holds because our framework is also proven inductively, and so the case of executions of length $N + 1$ in both the proof that (1) the data structure satisfies the conditions and (2) the traversal properties follow from the local view argument can rely on the correctness of the other proof's N case.

Simplifying linearizability proofs with local view arguments. To harness local view arguments, our approach uses *assertions* in the code as a way to divide the proof between

(1) the linearizability proof that *relies on the assertions*, and (2) the proof of the assertions, where the challenge of establishing properties of unsynchronized searches in concurrent executions is overcome by local view arguments.

Overall, our proof argument yields clear and comprehensible linearizability proofs, whose whole is (in some sense) greater than the sum of the parts, since each of the parts requires a simpler form of reasoning compared to contemporary linearizability proofs. We use local view arguments to devise simple linearizability proofs of a variant of the contention-friendly tree [14] (a self-balancing search tree), lists with lazy [24] or non-blocking [28] synchronization, and a lock-free skip list.

Our framework's *acyclicity* and *preservation* conditions can provide insight on algorithm design, in that their proofs can reveal unnecessary protections against interference. Indeed, our proof attempts exposed (small) parts of the search tree algorithm that were not needed to guarantee linearizability, leading us to consider a simpler variant of its search operation (see Remark 1).

Contributions. To summarize, we make the following contributions:

1. We provide a set of conditions under which reachability properties of local views, established using sequential reasoning, hold also for concurrent executions,
2. We show that these conditions hold for non-trivial concurrent data structures that use unsynchronized searches, and
3. We demonstrate that the properties established using local view arguments enable simple linearizability proofs, alleviating the need to consider interleavings of reads and writes during searches.

2 Motivating Example

As a motivating example we consider a self-balancing binary search tree with optimistic, read-only searches. This is an example of a concurrent data structure for which it is challenging to prove linearizability "from scratch." The algorithm is based on the contention-friendly (CF) tree [12, 14]. It is a fine-grained lock-based implementation of a set object with the standard `insert`(k), `delete`(k), and `contains`(k) operations. The algorithm maintains an *internal* binary tree that stores a key in every node. Similarly to the lazy list [24], the algorithm distinguishes between the *logical deletion* of a key, which removes it from the set represented by the tree, and the *physical removal* that unlinks the node containing the key from the tree.

We use this algorithm as a running example to illustrate how our framework allows to lift sequential reasoning into assertions about concurrent executions, which are in turn used to prove linearizability. In this section, we present the algorithm and explain the linearizability proof based on the assertions, highlighting the significant role of local view arguments in the proof.

Fig. 1 shows the code of the algorithm. (The code is annotated with assertions written inside curly braces, which the reader should ignore for now; we explain them in Sec. 2.1.) Nodes contain two boolean fields, *del* and *rem*, which indicate whether the node is logically deleted and physically removed, respectively. Modifications of a node in the tree are synchronized with the node's lock. Every operation starts with a call to `locate(k)`, which performs a standard binary tree search—without acquiring any locks—to locate the node with the target key k. This method returns the last link it traverses, (x, y). Thus, if k is found, $y.key = k$; if k is not found, $y = null$ and x is the node that would be k's parent if k

```
type N
  int key
  N left, right
  bool del,rem

N root←new N(∞);

N×N locate(int k)
  x,y←root
  while (y≠null ∧ y.key≠k)
    x←y
    if (x.key<k)
      y←x.right
    else
      y←x.left
  {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x) \wedge \diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y)$
    $\wedge\ x.key \neq k \wedge y \neq null$
      $\implies y.key = k$}
  return (x,y)

bool contains(int k)
  (_,y)←locate(k)
  if (y = null)
    {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} null)$}
    return false
  {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y)$}
  if (y.del)
    {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y \wedge y.del) \wedge y.key = k$}
    return false
  {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y \wedge \neg y.del) \wedge y.key = k$}
  return true
```

```
bool delete(int k)
  (_,y)←locate(k)
  if (y = null)
    {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} null)$}
    return false
  lock(y)
  if (y.rem) restart
  ret ← ¬y.del
  {$\texttt{root} \overset{k}{\rightsquigarrow} y \wedge y.key = k \wedge \neg y.rem$}
  y.del←true
  return ret

bool insert(int k)
  (x,y)←locate(k)
  {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x) \wedge x.key \neq k$}
  if (y≠null)
    {$\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y) \wedge y.key = k$}
    lock(y)
    if (y.rem) restart
    ret ← y.del
    {$\texttt{root} \overset{k}{\rightsquigarrow} y \wedge y.key = k \wedge \neg y.rem$}
    y.del←false
    return ret
  lock(x)
  if (x.rem) restart
  if (k < x.key ∧ x.left=null)
     {$\texttt{root} \overset{k}{\rightsquigarrow} x \wedge \neg x.rem$
       $\wedge\ k < x.key \wedge x.left = \texttt{null}$}
    x.left ← new N(k)
  else if (x.right=null)
     {$\texttt{root} \overset{k}{\rightsquigarrow} x \wedge \neg x.rem$
       $\wedge\ k > x.key \wedge x.right = \texttt{null}$}
    x.right ← new N(k)
  else
    restart
  return true
```

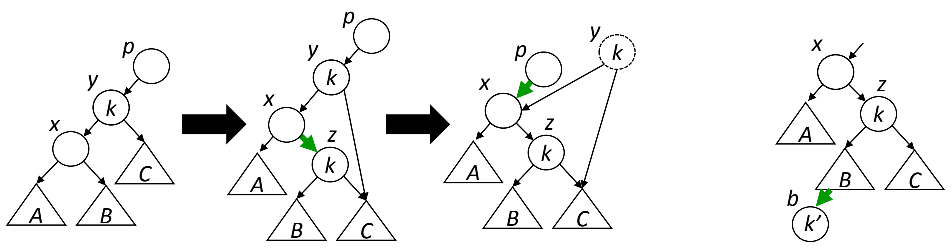
```
removeRight()
  (z,_) ← locate(*)
  lock(z)
  y ← z.right
  if(y=null ∨ z.rem)
    return
  lock(y)
  if (y.del)
    return
  if (y.left=null)
     z.right ← y.right
  else
    if (y.right=null)
     z.right ← y.left
    else return
  y.rem ← true

rotateRightLeft()
  (p,_) ← locate(*)
  lock(p)
  y ← p.left
  if(y=null ∨ p.rem)
    return
  lock(y)
  x ← y.left
  if(x=null)
    return
  lock(x)
  z ← duplicate(y)
  z.left ← x.right
  x.right ← z
  p.left ← x
  y.rem ← true
```

Figure 1 Running example. For brevity, **unlock** operations are omitted; a procedure releases all the locks it acquired when it terminates or **restart**s. $*$ denotes an arbitrary key.

were inserted. A `delete(k)` logically deletes y after verifying that y remained linked to the tree after its lock was acquired. An `insert(k)` either revives a logically deleted node or, if k was not found, links a new node to the tree. A `contains(k)` returns *true* if it locates a node with key k that is not logically deleted, and *false* otherwise.

Physical removal of nodes and balancing of the tree's height are performed using auxiliary methods.[1] The algorithm physically removes only nodes with at most one child. The `removeRight` method unlinks such a node that is a right child, and sets its *rem* field to notify threads that have reached the node of its removal. (We omit the symmetric `removeLeft`.) Balancing is done using rotations. Fig. 2a depicts the operation of `rotateRightLeft`, which needs to rotate node y (with key k) down. (We omit the symmetric operations.) It creates a new node z with the same key and *del* bit as y to take y's place, leaving y unchanged except for having its *rem* bit set. A similar technique for rotations is used in lock-free search trees [10].

[1] The reader should assume that these methods can be invoked at any time; the details of when the algorithm decides to invoke them are not material for correctness. For example, in [12, 14], these methods are invoked by a dedicated restructuring thread.



(a) Right rotation of y. (The bold green link is the one written in each step. The node with a dashed border has its *rem* bit set.)

(b) Node b is added after the right rotation of y, when y is no longer in the tree.

Figure 2 A right rotation, and how it can lead a search to observe an inconsistent state of the tree. If b is added after the rotation, a search for k' that starts before the rotation and pauses at x during the rotation will traverse the path $p, y, x, z, \ldots, b$, although y and b never exist simultaneously in the tree.

▶ Remark 1. The example of Fig. 1 differs from the original contention-friendly tree [12, 14] in a few points. The most notable difference is that our traversals do not consult the `rem` flag, and in particular we do not need to distinguish between a left and right rotate, making the traversals' logic simpler. Checking the `rem` flag is in fact unnecessary for obtaining linearizability, but it allows proving linearizability with a *fixed* linearization point, whereas proving the correctness of the algorithm without this check requires an *unfixed* linearization point. For our framework, the necessity to use an unfixed linearization point incurs no additional complexity. In fact, the simplicity of our proof method allowed us to spot this "optimization." In addition, the original algorithm performs backtracking by setting pointers from child to parent when nodes are removed. Instead, we restart the operation; see Sec. 7 for a discussion of backtracking. Lastly, we fix a minor omission in the description of [14], where the `del` field was not copied from a rotated node.

2.1 Proving Linearizability

Proving linearizability of an algorithm like ours is challenging because searches are performed with no synchronization. This means that, due to interference from concurrent updates, searches may observe an inconsistent state of the tree that has not existed at any point in time. (See Fig. 2.) In our example, while it is easy to see that `locate` in Fig. 1 constructs a search path to a node in sequential executions, what this implies for concurrent traversals is not immediately apparent. Proving properties of the traversal—in particular, that a node reached in the traversal truly lies on a search path for key k—is instrumental for the linearizability proof [49, 38].

Generally, our linearizability proofs consist of two parts: (1) proving a set of *assertions* in the code of the concurrent data structure, and (2) a proof of linearizability based on those assertions. The most difficult part and the main focus of our paper is proving the assertions using local view arguments, discussed in Sec. 2.2. In the remaining of this section we demonstrate that having assertions about the actual state during the concurrent execution makes it a straightforward exercise to verify that the algorithm in Fig. 1 is a linearizable implementation of a set, *assuming these assertions.*

Consider the assertions in Fig. 1. An assertion $\{\mathbb{P}\}$ means that $\mathbb{P}$ holds now (i.e., in any state in which the next line of code executes). An assertion of the form $\{\diamondminus\mathbb{P}\}$ means that $\mathbb{P}$ was true at some point between the invocation of the operation and now. The assertions contain predicates about the state of locked nodes, immutable fields, and predicates of the form $\texttt{root} \overset{k}{\rightsquigarrow} x$, which means that x resides on a *valid search path* for key k that starts at `root`; if $x = \mathit{null}$ this indicates that k is not in the tree (because a valid search path to k does not continue past a node with key k). Formally, search paths between objects (representing nodes in the tree) are defined as follows:

$$o_r \overset{k}{\rightsquigarrow} o_x \overset{\text{def}}{=} \exists o_0, \dots, o_m.\, o_0 = o_r \wedge o_m = o_x \wedge \forall i = 1..m.\ \text{nextChild}(o_{i-1}, k, o_i)\ ,\ \text{and}$$
$$\text{nextChild}(o_{i-1}, k, o_i) = (o_{i-1}.\mathit{key} > k \wedge o_{i-1}.\mathit{left} = o_i) \vee (o_{i-1}.\mathit{key} < k \wedge o_{i-1}.\mathit{right} = o_i)\ .$$

One can prove linearizability from these assertions by, for example, using an *abstraction function* $\mathcal{A} : H \to \wp(\mathbb{N})$ that maps a concrete memory state[2] of the tree, H, to the *abstract set* represented by this state, and showing that `contains`, `insert`, and `delete` manipulate this abstraction according to their specification. We define $\mathcal{A}$ to map H to the set of keys of the nodes that are on a valid search path for their key and are not logically deleted in H: $\mathcal{A}(H) = \{k \in \mathbb{N} \mid H \vDash \exists x. \texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.\mathit{key} = k \wedge \neg x.\mathit{del}\}$. ($H \vDash P$ means that P is true in H.)

The assertions almost immediately imply that for every operation invocation *op*, there exists a state H during *op*'s execution for which the abstract state $\mathcal{A}(H)$ agrees with *op*'s return value, and so *op* can be linearized at H. We provide a more detailed discussion in the extended version [20].

2.2 Proving the Assertions

To complete the linearizability proof, it remains to prove the validity of the assertions in concurrent executions. The most challenging assertions to prove are those concerning properties of unsynchronized traversals, which we target in this paper. In Sec. 3 we present our framework, which allows to deduce assertions of the form of $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x)$ at the end of (concurrent) traversals by considering only interference-free executions. We apply our framework to establish the assertions $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x)$ and $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y)$ in line 16. In fact, our framework allows to deduce slightly stronger properties, namely, of the form $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x \wedge \varphi(x))$, where $\varphi(x)$ is a property of a single field of x (see Remark 2). This is used to prove the assertions $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y \wedge y.\mathit{del})$ in line 26 and similarly in line 28. For completeness, we now show how the proof of the remaining assertions in Fig. 1 is attained, when assuming the assertions deduced by the framework. This concludes the linearizablity proof.

Reachability related assertions. In line 24 the fact that $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y)$ is true follows from line 16.

The writes in `insert` and `delete` (lines 38, 50, 56 and 59) require that a path exists *now*. This follows from the $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x)$ (known from the local view argument) and the fact that $\neg x.\mathit{rem}$, using an invariant similar to preservation (see Example 7): For every location x and key k, if $\texttt{root} \overset{k}{\rightsquigarrow} x$, then every write retains this unless it sets $x.\mathit{rem}$ before releasing the lock on x (this happens in lines 74, 77 and 95). Thus, when `insert` and `remove` lock x and see that it is not marked as removed, $\texttt{root} \overset{k}{\rightsquigarrow} x$ follows from $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x)$. Note that the fact that writes other than lines 74, 77 and 95 do not invalidate $\texttt{root} \overset{k}{\rightsquigarrow} x$ follows easily from their annotations.

[2] We use standard modeling of the memory state (the *heap*) as a function H from locations to values; see Sec. 3.

Additional assertions. The invariant that keys are immutable justifies assertions referring to keys of objects that are read earlier, e.g. in line 50 and the rest of the assertion in line 28 ($y.key$ is read earlier in `locate`). The rest of the assertions can be attributed to reading a location under the protection of a lock. An example of this is the assertion that $\neg y.rem$ in line 38.

3 The Framework: Correctness of Traversals Using Local Views

In this section we present the key technical contribution of our framework, which targets proving properties of traversals. We address properties of *reachability along search paths* (formally defined in Sec. 3.1). Roughly speaking, our approach considers the traversal in concurrent executions as operating without interference on a *local view*: the thread's potentially inconsistent picture of memory obtained by performing reads concurrently with writes by other threads. For a property $\mathbb{S}_{k,x} = \texttt{root} \overset{k}{\rightsquigarrow} x$ of reachability along a search path, we introduce conditions under which one can deduce that $\diamondminus\mathbb{S}_{k,x}$ holds in the actual global state of the concurrent data structure out of the fact that $\mathbb{S}_{k,x}$ holds in the local view of a single thread, where the latter is established using sequential reasoning (see Sec. 3.2). This alleviates the need to reason about intermediate states of the traversal in the concurrent proof.

This section is organized as follows: We start with some preliminary definitions. Sec. 3.1 defines the abstract, general notion of search paths our framework treats. Sec. 3.2 defines the notion of a local view which is at the basis of local view arguments. Sec. 3.3 formally defines the conditions under which local view arguments hold, and states our main technical result. In Sec. 3.4 we sketch the ideas behind the proof of this result.

Programming model. A *global state* (state) is a mapping between *memory locations* (locations) and *values*. A value is either a natural number, a location, or *null*. Without loss of generality, we assume that threads share access to a global state. Thus, memory locations are used to store the values of fields of objects. A *concurrent execution* (execution) is a sequence of states produced by an interleaving of atomic actions issued by threads. We assume that each atomic action is either a *read* or a *write* operation. (We treat synchronization actions, e.g., *lock* and *unlock*, as writes.) A *read* r consists of a value v and a location $read(r)$ with the meaning that r reads v from $read(r)$. Similarly, a write w consists of a value v and a location $mod(w)$ with the meaning that w sets $mod(w)$ to v. We denote by $w(H)$ the state resulting from the execution of w on state H.

3.1 Reachability Along Search Paths

The properties we consider are given by predicates of the form $\mathbb{S}_{k,x} = \texttt{root} \overset{k}{\rightsquigarrow} x$, denoting reachability of x by a k-search path, where `root` is the entry point to the data structure. A k-search path in state H is a sequence of locations that is traversed when searching for a certain element, parametrized by k, in the data structure. Reachability of an *object* x along a k-search path from `root` is understood as the existence of a k-search path between designated locations of x, e.g. the key field, and `root`.

Search paths may be defined differently in different data structures (e.g., list, tree or array). For example, k-search paths in Fig. 1 consist of sequences $\langle x.key, x.left, y.key\rangle$ where $y.key$ is the address pointed to by $x.left$ (meaning, the location that is the value stored in $x.left$) and $x.key > k$, or $\langle x.key, x.right, y.key\rangle$ where $y.key$ is the address pointed to by $x.right$ and $x.key < k$. This definition of k-search paths reproduces the definition of reachability along search paths from Sec. 2.1.

Our framework is oblivious to the specific definition of search paths, and only assumes the following properties of search paths (which are satisfied, for example, by the definition above):

- If $\ell_1, \dots, \ell_m$ is a k-search path in H and H' satisfies $H'(\ell_i) = H(\ell_i)$ for all $1 \le i < m$, then $\ell_1, \dots, \ell_m$ is a k-search path in H' as well, i.e., the search path depends on the values of locations in H only for the locations along the sequence itself (but the last).
- If $\ell_1, \dots, \ell_m$ and $\ell_m, \dots, \ell_{m+r}$ are both k-search paths in H, then so is $\ell_1, \dots, \ell_m, \dots, \ell_{m+r}$, i.e., search paths are closed under concatenation.
- If $\ell_1, \dots, \ell_m$ is a k-search path in H then so is $\ell_i, \dots, \ell_j$ for every $1 \le i \le j \le m$, i.e., search paths are closed under truncation.

▶ Remark 2. It is simple to extend our framework to deduce properties of the form $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x \wedge \varphi(x))$ where $\varphi(x)$ is a property of a single field of x. For example, $\varphi(x) = x.del$ states that the field del of x is true. As another example, the predicate $\texttt{root} \overset{k}{\rightsquigarrow} x \wedge (x.next = y)$ says that the *link* from x to y is reachable. See the extended version [20] for details.

3.2 Local Views and Their Properties

We now formalize the notion of *local view* and explain how properties of local views can be established using sequential reasoning.

Local view. Let $\bar{r} = r_1, \dots, r_d$ be a sequence of read actions executed by some thread. As opposed to the global state, the *local view* of the reading thread refers to the inconsistent picture of the memory state that the thread obtains after issuing $\bar{r}$ (concurrently with writes). Formally, the sequence of reads $\bar{r}$ induces a state H_{lv}, which is constructed by assigning to every location x which $\bar{r}$ reads the last value $\bar{r}$ reads in x. Namely, when $\bar{r}$ starts, its local view $H_{lv}^{(0)}$ is empty, and, assuming its ith read of value v from location ℓ, the produced local view is $H_{lv}^{(i)} = H_{lv}^{(i-1)}[\ell \mapsto v]$. We refer to $H_{lv} = H_{lv}^{(d)}$ as the *local view* produced by $\bar{r}$ (*local view* for short). We emphasize that while technically H_{lv} is a state, it is *not* necessarily an actual intermediate global state, and may have never existed in memory during the execution.

Sequential reasoning for establishing properties of local views. Properties of the local view H_{lv}, which are the starting point for applying our framework, are established using *sequential* reasoning. Namely, proving that a predicate such as $\texttt{root} \overset{k}{\rightsquigarrow} x$ holds in the local view at the end of the traversal amounts to proving that it holds in *any sequential* execution of the traversal, i.e., an execution without interference which starts at an *arbitrary* program state. This is because the concurrent traversal constructing the local view can be understood as a sequential execution that starts with the local view as the program state.

▶ **Example 1.** In the running example, straightforward sequential reasoning shows that indeed $\texttt{root} \overset{k}{\rightsquigarrow} x$ holds at line 16 in sequential executions of `locate(k)` (i.e., executions without interference), no matter at which program state the execution starts. This ensures that it holds, in particular, in the local view.

3.3 Local View Argument: Conditions & Guarantees

The main theorem underlying our framework bridges the discrepancy between the *local view* of a thread as it performs a sequence of read actions, and the actual *global state* during the traversal.

In the sequel, we fix a sequence of read actions $\bar{r} = r_1, \ldots, r_d$ executed by some thread, and denote the sequence of write actions executed concurrently with $\bar{r}$ by $\bar{w} = w_1, \ldots, w_n$. We denote the global state when $\bar{r}$ starts its execution by $H_c^{(0)}$, and the intermediate global states obtained after each prefix of these writes in $\bar{w}$ by $H_c^{(i)} = w_1 \ldots w_i(H_c^{(0)})$.

Using the above terminology, our framework devises conditions for showing for a reachability property $\mathbb{S}_{k,x}$ that if $\mathbb{S}_{k,x}(H_{lv})$ holds, then there exists $0 \leq i \leq n$ such that $\mathbb{S}_{k,x}(H_c^{(i)})$ holds, which means that $\diamondminus\mathbb{S}_{k,x}$ holds in the actual global state reached at the end of the traversal. We formalize these conditions below.

3.3.1 Condition I: Temporal Acyclicity

The first requirement of our framework concerns the order on the memory locations representing the data structure, according to which readers perform their traversals. We require that writers maintain this order *acyclic across intermediate states* of the execution. For example, when the order is based on following pointers in the heap, then, if it is possible to reach location y from location x by following a path in which every pointer was present at *some* point in time (not necessarily the same point), then it is not possible to reach x from y in the same manner. This requirement is needed in order to ensure that the order is robust even from the perspective of a concurrent reading operation, whose local view is obtained from a fusion of fractions of states.

We begin formalizing this requirement with the notion of search order on memory.

Search order. The acyclicity requirement is based on a mapping from a state H to a *partial order* that H induces on memory locations, denoted $\leq_H$, that captures the order in which operations read the different memory locations. Formally, $\leq_H$ is a *search order*:

▶ **Definition 2** (Search order). $\leq_H$ is a *search order* if it satisfies the following conditions:

(i) It is *locally determined*: if ℓ_2 is an immediate successor of ℓ_1 in $\leq_H$, then for every H' such that $H'(\ell_1) = H(\ell_1)$ it holds that $\ell_1 \leq_{H'} \ell_2$.

(ii) Search paths follow the order: if there is a k-search path between ℓ_1 and ℓ_2 in H, then $\ell_1 \leq_H \ell_2$.

(iii) Readers follow the order: reads in $\bar{r}$ always read a location further in the order in the current global state. Namely, if ℓ' is the last location read, the next read r reads a location ℓ from the state $H_c^{(m)}$ such that $\ell' \leq_{H_c^{(m)}} \ell$.

Note that the locality of the order is helpful for the ability of readers to follow the order: the next location can be known to come forward in the order solely from the last value the thread reads.

▶ **Example 3.** In the example of Fig. 1, the order $\leq_H$ is defined by following pointers from parent to children, i.e., all the fields of $x.\mathit{left}$ and $x.\mathit{right}$ are ordered after the fields of x, and the fields of an object are ordered by $x.\mathit{key} < x.\mathit{del} < \{x.\mathit{left}, x.\mathit{right}\}$. It is easy to see that this is a search order. Locality follows immediately, and so does the property that search paths follow the order. The fact that the read-in-order property holds for all the methods in Fig. 1 follows from a very simple syntactic analysis, e.g., in the case of `locate(k)`, children are always read after their parents and the field *key* is always accessed before *left* or *right*.

▶ Remark 3. Different search orders may be used for different traversals and different k's when establishing $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x)$ at the end of the traversal. In Definition 2, condition (iii) considers (just) the reads performed by the traversal of interest, and condition (ii) considers the possible search paths it constructs in the local view (just) for the k of interest.

Accumulated order and acyclicity. The accumulated order captures the order as it may be observed by concurrent traversals across different intermediate states. Formally, we define the *accumulated order* w.r.t. a sequence of writes $\hat{w}_1, \ldots, \hat{w}_m$, denoted $\leq^{\cup}_{\hat{w}_1 \ldots \hat{w}_m (H_c^{(0)})}$, as the transitive closure of $\bigcup_{0 \leq s \leq m} \leq_{\hat{w}_1 \ldots \hat{w}_s (H_c^{(0)})}$. In our example, the accumulated order consists of all parent-children links created during an execution. We require:

▶ **Definition 4** (Acyclicity). We say that $\leq_H$ satisfies *acyclicity of accumulated order* w.r.t. a sequence $\bar{w} = w_1, \ldots, w_n$ of writes if the accumulated order $\leq^{\cup}_{w_1 \ldots w_n (H_c^{(0)})}$ is a partial order.

▶ **Example 5.** In our running example, acyclicity holds because `insert`, `remove`, and `rotate` modify the pointers from a node only to point to new nodes, or to nodes that have already been reachable from that node. Modifications to other fields have no effect on the order. Note that `rotate` does not perform the rotation in place, but allocates a new object. Therefore, the accumulated order, which consists of all parent-children links created during an execution, is acyclic, and hence remains a partial order.

3.3.2 Condition II: Preservation of Search Paths

The second requirement of our framework is that for every write action w which happens concurrently with the sequence of reads $\bar{r}$ and modifies location $mod(w)$, if $mod(w)$ was k-reachable (i.e., $\mathbb{S}_{k,mod(w)}$ was true) at some point in time after $\bar{r}$ started and before w occurred, then it also holds right before w is performed. We note that this must hold in the presence of all possible interferences, including writes that operate on behalf of other keys (e.g. `insert`(k')). Formally, we require:

▶ **Definition 6** (Preservation). We say that $\bar{w}$ *ensures preservation of k-reachability by search paths* if for every $1 \leq m \leq n$, if for some $0 \leq i < m$, $H_c^{(i)} \models \mathbb{S}_{k,mod(w_m)}$ then $H_c^{(m-1)} \models \mathbb{S}_{k,mod(w_m)}$.

Note that $H_c^{(m-1)} \models \mathbb{S}_{k,mod(w_m)}$ iff $H_c^{(m)} \models \mathbb{S}_{k,mod(w_m)}$ since the search path to $mod(w_m)$ is not affected by w_m (by the basic properties of $\mathbb{S}_{k,mod(w_m)}$, see Sec. 3.1).

▶ **Example 7.** In our running example, preservation holds because w_m either modifies a location that has never been reachable (such as line 93), in which case preservation holds vacuously, or holds the lock on x when $\neg x.rem$ (without modifying its predecessor earlier under this lock).[3] In the latter case preservation holds because every previous write w' retains $\texttt{root} \overset{k}{\rightsquigarrow} mod(w_m)$ unchanged unless it sets the field `rem` of x to *true* before releasing the lock on x. Therefore, $\texttt{root} \overset{k}{\rightsquigarrow} mod(w_m)$ is retained still when w_m is performed. Preservation follows.

We emphasize that the preservation condition only requires that k-reachability is retained to modified locations ℓ and only at the point of time when the write w to ℓ is performed;

[3] In line 94, because x is a child of y which is a child of p and $\neg p.rem$, it follows that $\neg x.rem$ because a node marked with `rem` loses its single parent beforehand.

k-reachability *may* be lost at later points in time. In particular, locations whose reachability has been reduced may be accessed, as long as they are not modified after the reachability loss. For example, consider a rotation as in Fig. 2a. The rotation breaks the k-reachability of y: $\texttt{root} \overset{k}{\rightsquigarrow} y$ holds before the rotation but not afterwards. Indeed, our framework does not establish $\texttt{root} \overset{k}{\rightsquigarrow} y$, but infers $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y)$, which does hold. In this example, the preservation condition requires that the left and right pointers of y are not modified after this rotation is performed.[4] On the other hand, concurrent traversals may *access* y. In the example, this happens when (1) the traversal continues beyond y in the search for $k' \neq k$, and when (2) the traversal searches for k and terminates in y.

3.3.3 Local View Arguments' Guarantee

We are now ready to formalize our main theorem, relating reachability in the local view (Sec. 3.2) to reachability in the global state, provided that the conditions from Definitions 4 and 6 are satisfied.

▶ **Theorem 8.** *If (i) $\leq_H$ is a search order satisfying the accumulated acyclicity property w.r.t. $\bar{w}$, and (ii) $\bar{w}$ ensures preservation of k-reachability by search paths, then for every k and location x, if $\mathbb{S}_{k,x}(H_{lv})$ holds, then there exists $0 \leq i \leq n$ s.t. $\mathbb{S}_{k,x}(H_c^{(i)})$ holds.*

In the extended version [20] we illustrate how violating these conditions could lead to incorrectness of traversals. Sec. 3.4 discusses the main ideas behind the proof.

3.4 Proof Idea

We now sketch the correctness proof of Theorem 8. (The full details appear in the extended version [20].) The theorem transfers $\mathbb{S}_{k,x}$ from the local view to the global state. Recall that the local view is a fusion of the fractions of states observed by the thread at different times. To relate the two, we study the local view from the lens of a fabricated state: a state resulting from a subsequence of the interfering writes, which includes the observed local view. We exploit the cooperation between the readers and the writers that is guaranteed by the order $\leq_H$ (which readers and writers maintain) to construct a fabricated state which is closely related to the global state, in the sense that it *simulates* the global state (Definition 9); simulation depends both on the acyclicity requirement and on the preservation requirement (Lemma 11). Deducing the existence of a search path in an intermediate global state out of its existence in the local view is a corollary of this connection (Lemma 10).

Fabricated state. The fabricated state provides a means of analyzing the local view and its relation to the global (true) state. A *fabricated state* is a state consistent with the local view (i.e. it agrees with the value of every location present in the local view) that is constructed by a subsequence $\bar{w}_f = w_{i_1}, \ldots, w_{i_k}$ of the writes $\bar{w}$. One possible choice for $\bar{w}_f$ is the subsequence of writes whose effect was observed by $\bar{r}$ (i.e. $\bar{r}$ read-from). For relating the local view to the global state, which is constructed from the entire $\bar{w}$, it is beneficiary to include in $\bar{w}_f$ additional writes except for those directly observed by $\bar{r}$. In what follows, we choose the subsequence $\bar{w}_f$ so that the fabricated state satisfies a consistency property of *forward-agreement* with the global state. This means that although not all writes are included in $\bar{w}_f$ (as the thread misses some), the writes that are included have the same picture of the "continuation" of the data structure as it really was in the global state.

[4] Modification of $y.\texttt{rem}$ is allowed because this field does not affect search paths (see Sec. 3.1).

Construction of fabricated state based on order. Our construction of the fabricated state includes in $\bar{w}_f$ all the writes that occurred *backward* in time and wrote to locations *forward* in the order than the current location read, for every location read. (In particular, it includes all the writes that $\bar{r}$ reads from directly). Formally, let $mod(w)$ denote the location modified by write w. Then for every read r in $\bar{r}$ that reads location ℓ_r from global state $H_c^{(m)}$, we include in $\bar{w}_f$ all the writes $\{w_j \mid j \leq m \wedge \ell_r \leq^{\cup}_{w_1 \ldots w_m(H_c^{(0)})} mod(w_j)\}$ (ordered as in $\bar{w}$). We use the notation $H_f^{(j)} = w_{i_1} \ldots w_{i_j}(H_c^{(0)})$ for intermediate fabricated states. This choice of $\bar{w}_f$ ensures *forward-agreement* between the fabricated state and the global state: every write w_{i_j} in $\bar{w}_f$, the states on which it is applied, $H_c^{(i_j-1)}$ and $H_f^{(j-1)}$ agree on all locations ℓ such that $mod(w_{i_j}) \leq_{H_f^{(j-1)}} \ell$.

In what follows, we fix the fabricated state to be the state resulting at the end of this particular choice of $\bar{w}_f$. It satisfies forward-agreement by construction, and is an extension of the local view, relying on the *acyclicity* requirement.

Simulation. As we show next, the construction of $\bar{w}_f$ ensures that the effect of every write in $\bar{w}_f$ on $\mathbb{S}_{k,x}$ is guaranteed to concur with its effect on the real state with respect to changing $\mathbb{S}_{k,x}$ from false to true. We refer to this property as *simulation.*

▶ **Definition 9** (Simulation). For a predicate $\mathbb{P}$, we say that the subsequence of writes $w_{i_1} \ldots w_{i_k}$ $\mathbb{P}$-*simulates* the sequence $w_1 \ldots w_n$ if for every $1 \leq j \leq k$, if $\neg\mathbb{P}(H_f^{(j-1)})$ but $\mathbb{P}(w_{i_j}(H_f^{(j-1)}))$, then $\neg\mathbb{P}(H_c^{(i_j-1)}) \implies \mathbb{P}(w_{i_j}(H_c^{(i_j-1)}))$.

Simulation implies that the write w_{i_j} in $\bar{w}_f$ that changed $\mathbb{S}_{k,x}$ to true on the local view, would also change it on the corresponding global state $H_c^{(i_j)}$ (unless it was already true in $H_c^{(i_j-1)}$). This provides us with the desired global state where $\mathbb{S}_{k,x}$ holds. Using also the fact that $\mathbb{S}_{k,x}$ is upward-absolute [45] (namely, preserved under extensions of the state), we obtain:

▶ **Lemma 10.** *Let $\bar{w}_f$ be the subsequence of $\bar{w} = w_1, \ldots, w_n$ defined above. If $\mathbb{S}_{k,x}(H_{lv})$ holds and $\bar{w}_f$ $\mathbb{S}_{k,x}$-simulates $\bar{w}$, then there exists some $0 \leq i \leq n$ s.t. $\mathbb{S}_{k,x}(H_c^{(i)})$.*

Finally, we show that the fabricated state satisfies the simulation property. Owing to the specific construction of $\bar{w}_f$, the proof needs to relate the effect of writes on states which have a rather strong similarity: they agree on the contents of locations which come forward of the modified location. Preservation complements this by guaranteeing the existence of a path to the modified location:

▶ **Lemma 11.** *If $\bar{w}$ satisfies preservation of $\mathbb{S}_{k,mod(w)}$ for all w, then $\bar{w}_f$ $\mathbb{S}_{k,x}$-simulates $\bar{w}$ for all x.*

To prove the lemma, we show that preservation, together with forward agreement, implies the simulation property, which in turn implies that $\mathbb{S}_{k,x}(H_f^{(j-1)}) \implies \exists 0 \leq i \leq i_{j-1}\ \mathbb{S}_{k,x}(H_c^{(i)})$ (see Lemma 10). To show simulation, consider a write w_{i_j} that creates a k-search path ζ to x in $H_f^{(j)}$. We construct such a path in the corresponding global state. The idea is to divide ζ to two parts: the prefix until $mod(w_{i_j})$, and the rest of the path. Relying on forward agreement, the latter is exactly the same in the corresponding global state, and preservation lets us prove that there is also an appropriate prefix: necessarily there has been a k-search path to $mod(w_{i_j})$ in the fabricated state *before* w_{i_j}, so by induction, exploiting the fact that simulation up to $j-1$ implies that $\mathbb{S}_{k,x}(H_f^{(j-1)}) \implies \exists 0 \leq i \leq i_{j-1}.\ \mathbb{S}_{k,x}(H_c^{(i)})$, there has been a k-search path to $mod(w_{i_j})$ in some intermediate global state that occurred earlier than the time of w_{i_j}. Since w_{i_j} writes to $mod(w_{i_j})$, the preservation property ensures that there is a k-search path to $mod(w_{i_j})$ in the global state also at the time of the write w_{i_j}, and the claim follows.

4 Putting It All Together: Proving Linearizability Using Local Views

Recall that our overarching objective in developing the local view argument (Sec. 3) is to prove the correctness of assertions used in linearizability proofs (e.g., in Sec. 2.1). We now summarize the steps in the proof of the assertions. Overall, it is composed of the following steps:

1. Establishing properties of traversals on the local view using sequential reasoning,
2. Establishing the acyclicity and preservation conditions by simple concurrent reasoning, and
3. Proving the assertions when relying on local view arguments, augmented with some concurrent reasoning.

For the running example, step 1 is presented in Example 1, and step 2 consists of Examples 5 and 7 (see the extended version [20] for a full formal treatment). Step 3 concludes the proof as discussed in Sec. 2.2.

▶ Remark 4. While the local view argument, relying in particular on step 2, was developed to simplify the proofs of the assertions in 3, this goes also in the other direction. Namely, the concurrent reasoning required for proving the conditions of the framework (e.g., preservation) can be greatly simplified by relying on the correctness of the assertions (as they constrain possible interfering writes). Indeed, the proofs may mutually rely on each other. This is justified by a proof by induction: we prove that the current write satisfies the condition in the assertion, assuming that all previous writes did. This is also allowed in proofs of the conditions in Sec. 3.3, because they refer to the effect of *interfering* writes, that are known to conform to their respective assertions from the induction hypothesis. Hence, carrying these proofs together avoids circular reasoning and ensures validity of the proof.

5 Additional Case Studies

5.1 Lazy and Optimistic Lists

We successfully applied our framework to prove the linearizability of sorted-list-based concurrent set implementations with unsynchronized reads. Our framework is capable of verifying various versions of the algorithm in which `insert` and `delete` validate that the nodes they locked are reachable using a boolean field, as done in the lazy list algorithm [24], or by rescanning the list, as done in the optimistic list algorithm [28, Chap 9.8]. Our framework is also applicable for verifying implementations of the lazy list algorithm in which the logical deletion and the physical removal are done by the same operation or by different ones. We give a taste of these proofs here.

Fig. 3 shows an annotated pseudo-code of the lazy list algorithm. Every operation starts with a call to `locate(k)`, which performs a standard search in a sorted list—without acquiring any locks—to locate the node with the target key k. This method returns the last link it traverses, (x, y). Fig. 3 includes two variants of `contains`(k): In one variant, it returns `true` only if it finds a node with key k that is not logically deleted (line 139), while in the second variant it returns `true` even if that node is logically deleted (the commented `return` at line 141). Interestingly, the same annotations allow to verify both variants, and

```
97  type N
98    int key
99    N next
100   bool mark

102 N root←new N(−∞);

104 N×N locate(int k)
105   x,y←root
106   while (y≠null ∧ y.key<k)
107     x←y
108     y←x.next
109   {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y)$}
110   {$x.key < k \wedge (y \neq null \implies y.key \geq k)$}
111   return (x,y)

113 bool insert(int k)
114   (x,y)←locate(k)
115   if (y≠null ∧ y.key=k)
116     {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} y) \wedge y.key = k$}
117     return false
118   lock(x)
119   lock(y)
120   if (x.mark ∨ x.next≠y)
121     restart
122   {$\neg x.mark \wedge x.next = y$}
123   z←new N(k)
124   {$y \neq null \implies k > y.key$}
125   z.next←y
126   {$\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y \wedge x.key < k \wedge$
        $z.next = y \wedge \neg z.mark \wedge$
        $(y \neq null \implies k > y.key)$}
127   x.next←z
128   return true

129 bool contains(int k)
130   (_,y)←locate(k)
131   if (y=null)
132     {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} null)$}
133     return false
134   if (y.key≠k)
135     {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y) \wedge k < x.key \wedge y.key > k$}
136     return false
137   if (¬y.mark)
138     {$\texttt{root} \overset{k}{\rightsquigarrow} y \wedge y.key = k \wedge \neg y.mark$}
139     return true
140   {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} y) \wedge y.key = k \wedge y.mark$}
141   return false // return true

143 bool delete(int k)
144   (x,y)←locate(k)
145   if (y=null)
146     {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} null)$}
147     return false
148   if (y.key≠k)
149     {$⟠(\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y) \wedge x.key < k \wedge y.key > k$}
150     return false
151   {$y.key = k$}
152   lock(x)
153   lock(y)
154   if (x.mark ∨ y.mark ∨ x.next≠y)
155     restart
156   {$\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y \wedge y.key = k \wedge \neg x.mark \wedge \neg y.mark$}
157   y.mark←true
158   {$\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y \wedge y.key = k \wedge \neg x.mark \wedge y.mark$}
159   x.next←y.next
160   return true
```

Figure 3 Lazy List [24]. The code is annotated with assertions written inside curly braces. For brevity, **unlock** operations are omitted; a procedure releases all the locks it acquired when it terminates or **restart**s.

the proof differs only in the abstraction function mapping states of the list to abstract sets. Modifications of a node in the list are synchronized with the node's lock. An `insert(k)` operation calls `locate`, and then links a new node to the list if k was not found. `delete(k)` logically deletes y (after validating that y remained linked to the list after its lock was acquired), and then physically removes it.

As in Sec. 2, the assertions contain predicates of the form $\texttt{root} \overset{k}{\rightsquigarrow} x$, which means that x resides on a *valid search path* for key k that starts at `root`; the formal definition of a search path in the lazy list appears below. Note that $\texttt{root} \overset{k}{\rightsquigarrow} null$ indicates that k is not in the list.

$$o_r \overset{k}{\rightsquigarrow} o_x \overset{\text{def}}{=} \exists o_0, \ldots, o_m.\ o_0 = o_r \wedge o_m = o_x \wedge \forall i = 1..m.\ o_{i-1}.key < k \wedge o_{i-1}.next = o_i$$

We prove the linearizability of the algorithm using an *abstraction function*. One abstraction function we may use maps H to the set of keys of the nodes that are on a valid search path for their key and are not logically deleted in H:

$$\mathcal{A}^{logical}(H) = \{k \in \mathbb{N} \mid H \models \exists x. \texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.key = k \wedge \neg x.mark\}.$$

Another possibility is to define the abstract set to be the keys of all the reachable nodes:

$$\mathcal{A}^{physical}(H) = \{k \in \mathbb{N} \mid H \models \exists x. \texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.key = k\}.$$

We note that $\mathcal{A}^{logical}(H)$ can be used to verify the code of `contains` as written, while $\mathcal{A}^{physical}(H)$ allows to change the algorithm to return `true` in line 141. In both cases, the proof of linearizability is carried out using the same assertions currently annotating the code. In the rest of this section, we discuss the verification of the code in Fig. 3 as written, and thus use $\mathcal{A}(H) = \mathcal{A}^{logical}$ as the abstraction function. The assertions almost immediately imply that for every operation invocation *op*, there exists a state H during *op*'s execution for which the abstract state $\mathcal{A}(H)$ agrees with *op*'s return value, and so *op* can be linearized at H; we need only make the following observations. First, `contains()` and a failed `delete()` or `insert()` do not modify the memory, and so can be linearized at the point in time in which the assertions before their `return` statements hold. Second, in the state H in which a successful `delete(k)` (respectively, `insert(k)`) performs a write, the assertions on line 156 (respectively, line 126) imply that $k \in \mathcal{A}(H)$ (respectively, $k \notin \mathcal{A}(H)$). Therefore, these writes change the abstract set, making it agree with the operation's return value of *true*. Finally, it only remains to verify that the physical removal performed by `delete`(k) in state H does not modify $\mathcal{A}(H)$. Indeed, as an operation modifies a field of node v only when it has v locked, it is easy to see that for any node x and key k, if $\texttt{root} \overset{k}{\rightsquigarrow} x$ held before the write, then it also holds afterwards with the exception of the removed node y. However, `delete`(k) removes a deleted node, and thus does not change $\mathcal{A}(H)$.

The proof of the assertions in Fig. 3 utilizes a local view argument for the $\diamondminus$ assertion in line 109 for the predicate $\texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.next = y$, using the extension with a single field discussed in Remark 2. The conditions of the local view argument are easy to prove: The acyclicity requirement is evident, as writes modify the pointers from a node only to point to new nodes, or to nodes that have already been reachable from that node. Preservation holds because a write either (i) marks a node, which does not affect the search paths; (ii) modifies a location that has never been reachable (such as line 125), in which case preservation holds vacuously; (iii) removes a marked node y (line 159) which removes all the search paths that go through it. However, as y is marked, its fields are not going to be modified later on, and thus y cannot be the cause of violating preservation. Furthermore, all search paths that reach y's successor before the removal are retained and merely get shorter; or (iv) adds a reachable node z in between two reachable nodes x and y (line 127). However, as z's key is smaller than y's, the insertion preserves any search paths which goes *through* y's `next` pointer.

As for the rest of the assertions, when `insert` and `delete` lock x and see that it is not marked, the $\texttt{root} \overset{k}{\rightsquigarrow} x$ property follows from the $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} x)$ deduced above by a local view argument using the same invariant in preservation above.[5] The remainder assertions are attributed to reading a location under the protection of a lock, e.g. $\neg x.mark$ in line 122.

5.2 Lock-free List and Skip-List

We used our framework to prove the linearizability a sorted lock-free list-based concurrent set algorithm [28, Chapter 9.8] and of a lock-free skip-list-based concurrent set algorithm [28, Chapter 14.4]. In these proofs we use local view arguments to prove the concurrent traversals of the `contains` method, which is the most difficult part of the proofs: `add` and `remove` use the internal `find` which traverses the list and also prunes out marked nodes, and thus their correctness follows easily from an invariant ensuring the reachability of unmarked nodes.

[5] As in Sec. 5.2, these assertions could also be deduced directly from a slightly stronger invariant that unmarked nodes are reachable and that the list is sorted. This is not the case in the optimistic list of [28, Chap 9.8] which rescans instead of using a marked bit. In both cases `contains` requires a local view argument.

The proofs appear in [20].

6 Related Work

Verifying linearizability of concurrent data structures has been studied extensively. Some techniques, e.g., [1, 2, 18, 52, 51], apply to a restricted set of algorithms where the linearization point of every invocation is *fixed* to a particular statement in the code. While these works provide more automation, they are not able to deal with the algorithms considered in our paper where for instance, the linearization point of `contains(k)` invocations is not fixed. Generic reductions of verifying linearizability to checking a set of assertions in the code have been defined in [5, 6, 7, 35, 25, 50, 54]. These works apply to algorithms with non-fixed linearization points, but they do not provide a systematic methodology for proving the assertions, which is the main focus of our paper.

Verifying linearizability has also been addressed in the context of defining program logics for *compositional* reasoning about concurrent programs. In this context, the goal is to define a proof methodology that allows composing proofs of program's components to get a proof for the entire program, which can also be reused in every valid context of using that program. Improving on the classical Owicki-Gries [40] and Rely-Guarantee [30] logics, various extensions of Concurrent Separation Logic [4, 9, 39, 41] have been proposed in order to reason compositionally about different instances of fine-grained concurrency, e.g. [31, 34, 15, 43, 47, 48]. However, they focus on the reusability of a proof of a component in a larger context (when composed with other components) while our work focuses on simplifying the proof goals that guarantee linearizability. The concurrent reasoning needed for our framework could be carried out using one of these logics.

The proof of linearizability of the lazy-list algorithm given in [38] is based on establishing the conditions required by the *hindsight lemma* [38, Lemma 5.2]. The lemma states that every link traversed during an unsynchronized traversal was indeed reachable at some point in time between the beginning of the traversal and the moment the link was crossed. This enables verifying the correctness of the `contains` method using, effectively, sequential reasoning. The hindsight lemma is a specific instance of the extension discussed in Remark 2, and its assumptions narrows its application to concurrent set algorithms implemented using sorted singly-linked lists. In contrast, we present a fundamental technique which is based on far more generic properties which is applicable to list and tree-based data structures alike.

The proof methodology for proving linearizability of [33] relies on properties of the data structure in sequential executions. The methodology assumes the existence of *base points*, which are points in time during the concurrent execution of a search in which some predicate holds over the shared state. For instance, when applying the methodology to the lazy list, they prove the existence of base points using prior techniques [38, 53] that employ tricky concurrent reasoning. Our work is thus complementary to theirs: our proof argument is meant to replace the latter kind of reasoning, and can thus simplify proofs of the existence of base points.

The *Edgeset* framework of Shasha and Goodman [44], which has recently been formalized using concurrent separation logic [32], provides conditions for the linearizability of concurrent search data structures. It relies on a precondition that for any operation on key k, $\texttt{root} \overset{k}{\leadsto} x$ holds when the operation looks for, inserts, or deletes k at x. However, the optimistic data structures that we consider often do not satisfy this precondition, making the Edgeset framework inapplicable. (Example 7 describes how this precondition does not hold in our search tree example, and a similar issue exists in the lazy-list.) Moreover, the

Edgeset precondition implies that the linearization point of an operation occurs at one of its own atomic steps. Our framework does not have this requirement. Shasha and Goodman also describe three algorithm templates and prove, using concurrent reasoning, that these templates satisfy the preconditions of the Edgeset framework. In contrast, our argument uses sequential reasoning for traversals, and our concurrent proofs consider only the effects of interleaving writes—not both reads and writes.

7 Conclusions and Future Work

This paper presents a novel approach for constructing linearizability proofs of concurrent search data structures. We present a general proof argument that is applicable to many existing algorithms, uncovering fundamental structure—the acyclicity and preservation conditions—shared by them. We have instantiated our framework for a self-balancing binary search tree, lists with lazy [24] or non-blocking [28] synchronization, and a lock-free skip list. To the best of our knowledge, our work is the first to prove linearizability of a self-balancing binary search tree using a unified proof argument.

An important direction for future work is the mechanism of backtracking. Some algorithms, including the original CF tree [12, 14], backtrack instead of restarting when their optimistic validation fails. In the CF tree, backtracking is implemented by directing pointers from child to parent, breaking our acyclicity requirement. A similar situation arises in the in-place rotation of [8]. Handling these scenarios in our proof argument is an interesting direction for future work.

An additional direction to explore is validations performed during traversals. For example, the SnapTree algorithm [8] performs in-place rotations which violate preservation. The algorithm overcomes this by performing hand-over-hand validation during a lock-free traversal. This validation, consisting of re-reading previous locations and ensuring version numbers have not changed, does not fit our approach of sequential reasoning on traversals.

The preservation of reachability to location of modification arises naturally out of the correctness of traversals in modifying operations, ensuring that the conclusion of the traversal—the existence of a path—holds not only in some point in the past, but also holds at the time of the modification. We show that, surprisingly, preservation, when it is combined with the order, suffices to reason about the traversal by a local view argument. We base the correctness of read-only operations on the same predicates, and so rely on the same property. It would be interesting to explore different criteria which ensure the simulation of the fabricated state constructed based on the accumulated order.

Finding ways to extend the framework in these directions is an interesting open problem. This notwithstanding, we believe that our framework captures important principles underlying modern highly concurrent data structures that could prove useful both for structuring linearizability proofs and elucidating the correctness principles behind new concurrent data structures.

Acknowledgments. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreements No [759102-SVIS] and [678177]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259, and the Israeli Science Foundation (ISF) grant No. 2005/17. We thank the anonymous reviewers whose comments helped improve the paper.

References

1 Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.

2 Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07*, volume 4590 of *LNCS*, pages 477–490, 2007.

3 Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2611462.2611471`, `doi:10.1145/2611462.2611471`.

4 Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005. URL: `http://doi.acm.org/10.1145/1040305.1040327`, `doi:10.1145/1040305.1040327`.

5 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.

6 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 95–107, 2015.

7 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017. URL: `https://doi.org/10.1007/978-3-319-63390-9_28`, `doi:10.1007/978-3-319-63390-9_28`.

8 Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268, 2010.

9 Stephen D. Brookes. A semantics for concurrent separation logic. In Gardner and Yoshida [22], pages 16–34. URL: `https://doi.org/10.1007/978-3-540-28644-8_2`, `doi:10.1007/978-3-540-28644-8_2`.

10 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *PPoPP*, 2014.

11 Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, 2012.

12 Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 229–240, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

13 Tyler Crain, Vincent Gramoli, and Michel Raynal. No Hot Spot Non-blocking Skip List. In *ICDCS*, 2013.

14 Tyler Crain, Vincent Gramoli, and Michel Raynal. A fast contention-friendly binary search tree. *Parallel Processing Letters*, 26(03):1650015, 2016. `doi:10.1142/S0129626416500158`.

15 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, 2014. URL: `https://doi.org/10.1007/978-3-662-44202-9_9`, `doi:10.1007/978-3-662-44202-9_9`.

16 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, 2015.

17 Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. In *PPoPP*, 2014.

18 Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV '13*, volume 8044 of *LNCS*, pages 174–190. Springer.

19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *PODC*, 2010.

20 Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of chaos: Proving linearizability using local views. *CoRR*, abs/1805.03992, 2018. URL: `http://arxiv.org/abs/1805.03992`, `arXiv:1805.03992`.

21 Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory, February 2004.

22 Philippa Gardner and Nobuko Yoshida, editors. *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*. Springer, 2004. URL: `https://doi.org/10.1007/b100113`, `doi:10.1007/b100113`.

23 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, 2001.

24 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Bill Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.

25 Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, pages 242–256, 2013.

26 M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

27 Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *SIROCCO*, 2007.

28 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

29 Shane V. Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *SPAA*, 2012.

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

31 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. URL: `http://doi.acm.org/10.1145/2676726.2676980`, `doi:10.1145/2676726.2676980`.

32 Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL*, 2(POPL):37:1–37:31, 2018. URL: `http://doi.acm.org/10.1145/3158125`, `doi:10.1145/3158125`.

33 Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015. URL: `https://doi.org/10.1007/978-3-662-48653-5_24`, `doi:10.1007/978-3-662-48653-5_24`.

34 Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 561–574. ACM, 2013. URL: `http://doi.acm.org/10.1145/2429069.2429134`, `doi:10.1145/2429069.2429134`.

35 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470, 2013.

36 Maged M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *SPAA*, 2002.

37 Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP*, 2014.

38 P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 85–94, 2010.

39 Peter W. O'Hearn. Resources, concurrency and local reasoning. In Gardner and Yoshida [22], pages 49–67. URL: `https://doi.org/10.1007/978-3-540-28644-8_4`, `doi:10.1007/978-3-540-28644-8_4`.

40 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. URL: `http://doi.acm.org/10.1145/360051.360224`, `doi:10.1145/360051.360224`.

41 Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 297–302. ACM, 2007. URL: `http://doi.acm.org/10.1145/1190216.1190261`, `doi:10.1145/1190216.1190261`.

42 Arunmoezhi Ramachandran and Neeraj Mittal. A Fast Lock-Free Internal Binary Search Tree. In *ICDCN*, 2015.

43 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 2015. URL: `https://doi.org/10.1007/978-3-662-46669-8_14`, `doi:10.1007/978-3-662-46669-8_14`.

44 Dennis E. Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90, 1988. URL: `http://doi.acm.org/10.1145/42201.42204`, `doi:10.1145/42201.42204`.

45 Joseph R Shoenfield. The problem of predicativity. In *Mathematical Logic In The 20th Century*, pages 427–434. World Scientific, 2003.

46 Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *USENIX ATC*, 2011.

47 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13,*

Boston, MA, USA - September 25 - 27, 2013, pages 377–390. ACM, 2013. URL: http://doi.acm.org/10.1145/2500365.2500600, doi:10.1145/2500365.2500600.

48 V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2008.

49 V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.

50 Viktor Vafeiadis. Automatically proving linearizability. In *CAV '10*, volume 6174 of *LNCS*, pages 450–464.

51 Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI '09: Proc. 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.

52 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP '06*, pages 129–136. ACM.

53 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. A safety proof of a lazy concurrent list-based set implementation. Technical Report UCAM-CL-TR-659, University of Cambridge, Computer Laboratory, 2006.

54 He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 3–19, 2015.