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 Shared memory algorithms

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

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, 36, 41], skip lists [13, 21, 27], and lists/hash tables [23, 24, 35, 44].

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 [37, 32, 47, 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 the following 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 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`

[1] The reader should assume that these methods can be invoked at any time; the details of when the algorithm decides

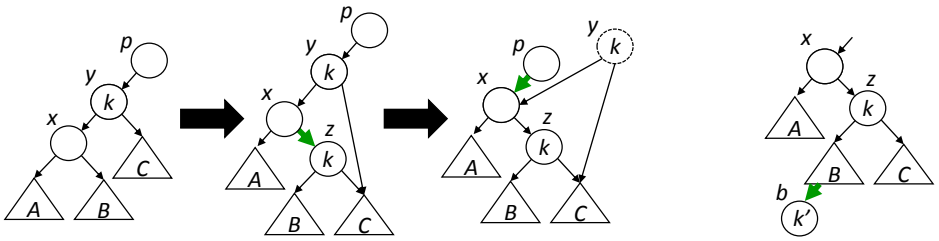
```
1  type N
2    int key
3    N left, right
4    bool del,rem

6  N root←new N(∞);

8  N×N locate(int k)
9    x,y←root
10   while (y≠null ∧ y.key≠k)
11     x←y
12     if (x.key<k)
13       y←x.right
14     else
15       y←x.left
16   {⟐(root ⇝k x) ∧ ⟐(root ⇝k y)
       ∧ x.key ≠ k ∧ y ≠ null ⟹ y.key = k}
17   return (x,y)

19 bool contains(int k)
20   (_,y)←locate(k)
21   if (y = null)
22     {⟐(root ⇝k null)}
23     return false
24   {⟐(root ⇝k y)}
25   if (y.del)
26     {⟐(root ⇝k y ∧ y.del) ∧ y.key = k}
27     return false
28   {⟐(root ⇝k y ∧ ¬y.del) ∧ y.key = k}
29   return true

30 bool delete(int k)
31   (_,y)←locate(k)
32   if (y = null)
33     {⟐(root ⇝k null)}
34     return false
35   lock(y)
36   if (y.rem) restart
37   ret ← ¬y.del
38   {root ⇝k y ∧ y.key = k ∧ ¬y.rem}
39   y.del←true
40   return ret

42 bool insert(int k)
43   (x,y)←locate(k)
44   {⟐(root ⇝k x) ∧ x.key ≠ k}
45   if (y≠null)
46     {⟐(root ⇝k y) ∧ y.key = k}
47     lock(y)
48     if (y.rem) restart
49     ret ← y.del
50     {root ⇝k y ∧ y.key = k ∧ ¬y.rem}
51     y.del←false
52     return ret
53   lock(x)
54   if (x.rem) restart
55   if (k < x.key ∧ x.left=null)
56     {root ⇝k x ∧ ¬x.rem
        ∧ k < x.key ∧ x.left = null}
57     x.left ← new N(k)
58   else if (x.right=null)
59     {root ⇝k x ∧ ¬x.rem
        ∧ k > x.key ∧ x.right = null}
60     x.right ← new N(k)
61   else
62     restart
63   return true

64 removeRight()
65   (z,_) ← locate(*)
66   lock(z)
67   y ← z.right
68   if(y=null ∨ z.rem)
69     return
70   lock(y)
71   if (y.del)
72     return
73   if (y.left=null)
74     z.right ← y.right
75   else if (y.right=null)
76     z.right ← y.left
77   else
78     return
79   y.rem ← true

81 rotateRightLeft()
82   (p,_) ← locate(*)
83   lock(p)
84   y ← p.left
85   if(y=null ∨ p.rem)
86     return
87   lock(y)
88   x ← y.left
89   if(x=null)
90     return
91   lock(x)
92   z ← duplicate(y)
93   z.left ← x.right
94   x.right ← z
95   p.left ← x
96   y.rem ← true
```

Figure 1 Running example. When a procedure terminates or **restart**s it releases all the locks it acquired. $*$ denotes an arbitrary key.

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].

▶ 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

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.

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 [47, 37].

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 $x = null$ 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:

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

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:

$$\mathcal{A}(H) = \{k \in \mathbb{N} \mid H \vDash \exists x.\, \texttt{root} \overset{k}{\rightsquigarrow} x \wedge x.key = k \wedge \neg x.del\}, \text{ where } H \vDash P \text{ means that } P \text{ is true in state } H.$$

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 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 38 (respectively, lines 50, 56, and 59) 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, since these are the only memory modifications performed by the set operations, it only remains to verify that no write performed by an auxiliary operation in state H modifies $\mathcal{A}(H)$. 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, `removeRight` removes a deleted node, and thus does not change $\mathcal{A}(H)$. Further, `rotateRightLeft` links z (y's replacement) to the tree before unlinking y, so the existence of a search path to $y.k = z.k$ is retained (although the actual path changes), leaving the contents of the abstract set unchanged because the del bit in z has the same value as in y.

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 $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y \wedge y.del)$ 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 $\neg x.rem$, 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 $x.rem$ before releasing the lock on x (this happens in lines 74, 76 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, 76 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 $\diamondsuit\mathbb{S}_{k,x}$ 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 the tree of 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 $\ell_1, \ldots, \ell_m$ is a k-search path in H and H' satisfies $H'(\ell_i) = H(\ell_i)$ for all $1 \leq i < m$, then $\ell_1, \ldots, \ell_m$ 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 $\ell_1, \ldots, \ell_m$ and $\ell_m, \ldots, \ell_{m+r}$ are both k-search paths in H, then so is $\ell_1, \ldots, \ell_m, \ldots, \ell_{m+r}$, i.e., search paths are closed under concatenation.
- If $\ell_1, \ldots, \ell_m$ is a k-search path in H then so is $\ell_i, \ldots, \ell_j$ for every $1 \leq i \leq j \leq m$, i.e., search paths are closed under truncation.

▶ Remark 2. It is simple to extend our framework to deduce properties of the form $\Diamond(\texttt{root} \overset{k}{\rightsquigarrow} x \wedge \varphi(x))$ 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 Appendix A.3.2 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 $\bar{r} = r_1, \ldots, r_d$ 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 $\diamond\mathbb{S}_{k,x}$ 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 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 $x.left$ and $x.right$ are ordered after the fields of x, and the fields of an object are ordered by $x.key < x.del < \{x.left, x.right\}$. 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*.

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 $1 \le m \le n$, if for some $0 \le i < m$, $H_c^{(i)} \vDash \mathbb{S}_{k,mod(w_m)}$ then $H_c^{(m-1)} \vDash \mathbb{S}_{k,mod(w_m)}$.

▶ **Example 7.** In our running example, preservation holds because w 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]. Every other write w' retains `root` $\overset{k}{\rightsquigarrow} x$ unchanged, unless it sets the field `rem` of x to *true* before releasing the lock on x. Preservation follows.

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)* $\le_H$ *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* $0 \le i \le n$ *s.t.* $\mathbb{S}_{k,x}(H_c^{(i)})$ *holds.*

In Appendix B 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 Appendix A.) 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 $\le_H$ (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).

---

[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.

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.

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 [43] (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

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 Appendix C for a full formal treatment). Step 3 concludes the proof as discussed in Sec. 2.2.

▶ Remark 3. 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

```
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   {⟐(root ⇝k x ∧ x.next = y)}
110   {x.key < k ∧ (y ≠ null ⟹ y.key ≥ k)}
111   return (x,y)

113 bool insert(int k)
114   (x,y)←locate(k)
115   if (y≠null ∧ y.key=k)
116     {⟐(root ⇝k y) ∧ y.key = k}
117     return false
118   lock(x)
119   lock(y)
120   if (x.mark ∨ x.next≠y)
121     restart
122   {¬x.mark ∧ x.next = y}
123   z←new N(k)
124   {y ≠ null ⟹ k > y.key}
125   z.next←y
126   {root ⇝k x ∧ x.next = y ∧ x.key < k ∧
        z.next = y ∧ ¬z.mark ∧ (y ≠ null ⟹ k > y.key)}
127   x.next←z
128   return true

129 bool contains(int k)
130   (_,y)←locate(k)
131   if (y=null)
132     {⟐(root ⇝k null)}
133     return false
134   if (y.key≠k)
135     {⟐(root ⇝k x ∧ x.next = y) ∧ k < x.key ∧ y.key > k}
136     return false
137   if (¬y.mark)
138     {root ⇝k y ∧ y.key = k ∧ ¬y.mark}
139     return true
140   {⟐(root ⇝k y) ∧ y.key = k ∧ y.mark}
141   return false // return true

143 bool delete(int k)
144   (x,y)←locate(k)
145   if (y=null)
146     {⟐(root ⇝k null)}
147     return false
148   if (y.key≠k)
149     {⟐(root ⇝k x ∧ x.next = y) ∧ x.key < k ∧ 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   {root ⇝k x ∧ x.next = y ∧ y.key = k ∧ ¬x.mark ∧ ¬y.mark}
157   y.mark←true
158   {root ⇝k x ∧ x.next = y ∧ y.key = k ∧ ¬x.mark ∧ y.mark}
159   x.next←y.next
160   return true
```

Figure 3 Lazy List [24]. The code is annotated with assertions written inside curely braces.

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 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 `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 `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:

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

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

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

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 `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 $\Diamond\!\!-$ assertion in line 109 for the predicate `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 `root` $\overset{k}{\rightsquigarrow} x$ property follows from the $\Diamond\!\!-$(`root` $\overset{k}{\rightsquigarrow} x$) deduced above by a local view argument using the same invariant in preservation above.[4] 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. The proofs appear in Appendices D and E.

[4] 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.

6 Related Work

Verifying linearizability of concurrent data structures has been studied extensively. Some techniques, e.g., [1, 2, 18, 50, 49], 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, 34, 25, 48, 52]. 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 [39] and Rely-Guarantee [30] logics, various extensions of Concurrent Separation Logic [4, 9, 38, 40] have been proposed in order to reason compositionally about different instances of fine-grained concurrency, e.g. [31, 33, 15, 42, 45, 46]. 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 [37] is based on establishing a *hindsight lemma* which states that every link traversed during an unsynchronized traversal was indeed reachable at some point in time. 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 proof in [37] is done in an ad hoc manner which is tailored to the lazy-list algorithm. In contrast, we present a fundamental technique which is applicable to list and tree-based data structures alike.

The proof methodology for proving linearizability of [32] 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 [37, 51] 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.

7 Conclusion and Future Work

This paper presents a novel approach for architecting 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 reasoning sequentially about 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.

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. URL: http://www.worldscientific.com/doi/abs/10.1142/S0129626416500158, arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129626416500158, 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 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.

33 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.

34 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.

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

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

37 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.

38 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.

39 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.

40 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.

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

42 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.

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

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

45 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.

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

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

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

49 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.

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

51 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.

52 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.

A Proof Method

In this section, we describe our main result, which allows to lift a property $\mathbb{P}$ that holds on the local view of a thread accumulated by performing a sequence of reads $\bar{r}$ to a property that held on the global, concrete, state *at some point* during the execution of $\bar{r}$.

A.0.0.1 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 threads share access to immutable global variables and to a mutable *global heap*. Thus, memory locations are used to stores the values of fields of objects. A *concurrent execution* (execution) π is a sequence of states produced by an interleaving of atomic actions issued by threads. A pair of states (H, H') is a *transition* in π if $\pi = \cdots H H' \cdots$. Without loss of generality, we assume that each transition results from either a *read* or *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.

A.1 Between the Local View and the Global State

For the rest of this section, we fix a sequence $\bar{r} = r_1, \ldots, r_d$ of reads performed by a thread, and the execution π from which it is taken. We denote the (global) state when the reading sequence $\bar{r}$ started its execution by $H_c^{(0)}$. We denote the sequence of writes performed concurrently with $\bar{r}$ in π by $\bar{w} = w_1, \ldots, w_n$. The sequence $\bar{w}$ produces intermediate *global states* after the execution of each write. We denote the global state after execution of $w_1 \ldots w_i$ by $H_c^{(i)}$, i.e., $H_c^{(i)} = w_1 \ldots w_i(H_c^{(0)})$.

Local views. The sequence of reads $\bar{r}$ induces a state H_{lv}, which directly corresponds to the values $\bar{r}$ observes in memory. This is constructed by assigning to location x the value read in the last read in $\bar{r}$ if x is read at all, i.e., When $\bar{r}$ starts its local view $H_{lv}^{(0)}$ is empty, and, assuming its ith read is (ℓ, v), 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.

Fabricated state. In order to relate between the local view and the global state, we consider a *fabricated state* H_f. The fabricated state is a state such that (i) $H_{lv} \subseteq H_f$, and (ii) H_f is constructed by a *subsequence* $\bar{w}_f = w_{i_1}, \ldots, w_{i_k}$ of $\bar{w}$, i.e., $H_f = w_{i_1} \ldots w_{i_k}(H_c^{(0)})$. Again, while the sequence of writes $\bar{w} = w_1, \ldots, w_n$ comes from an execution, the subsequence $w_{i_1}, \ldots, w_{i_k}$ may not be produced by any execution. and thus H_f may not occur in any execution of the algorithm. As before, we denote the intermediate fabricated state constructed after performing the first j writes in $\bar{w}_f$ by $H_f^{(j)}$, i.e., $H_f^{(j)} = w_{i_1} \ldots w_{i_j}(H_c^{(0)})$. The notion of a fabricated state is in line with the intuition that the reading sequence has been affected by some of the concurrent writes but also missed some. The specific construction of the fabricated state we use in our framework is provided in Appendix A.2.

Our proof approach (i) establishes a connection, that we call *simulation*, between the effect of the writes on the fabricated state to their effect in the concrete state, and (ii) uses a property called *upward absoluteness* to relate the local view to the fabricated state (based on the property that $H_{lv} \subseteq H_f$).

In this way, the fabricated state allows us to consider the memory (H_{lv}) that $\bar{r}$ observes as if $\bar{r}$ were operating sequentially on a state (H_f) that is closely related to the global memory state (H_c).

A.1.1 From the Fabricated State to the Global State Using Simulation

The fabricated state will be constructed such that it has the following connection to the concrete state: if a write turns $\mathbb{P}$ to true in an intermediate fabricated state $H_f^{(j)} = w_{i_1} \dots w_{i_j}(H_c^{(0)})$ obtained after executing $w_{i_1}, \dots, w_{i_j}$, then it also turns $\mathbb{P}$ to true in the intermediate global state $H_c^{(i_j)} = w_1 \dots w_{i_j}(H_c^{(0)})$ obtained after executing the prefix $w_1, \dots, w_{i_j}$ of the full sequence of writes. This is formalized in the following definition. We then show that if this connection is established, then $\mathbb{P}$ being true in the fabricated state transfers to it being true in *some* intermediate point in the *global* state.

▶ **Definition 12** (Simulation)**.** The subsequence of writes $w_{i_1} \dots w_{i_k}$ *simulates* the sequence $w_1 \dots w_n$ w.r.t. $\mathbb{P}$ if for every $1 \le j \le k$, if $\neg\mathbb{P}(H_f^{(j-1)})$ but $\mathbb{P}(w_{i_j}(H_f^{(j-1)}))$, then $\neg\mathbb{P}(H_c^{(i_j-1)}) \implies \mathbb{P}(w_{i_j}(H_c^{(i_j-1)}))$.

We say that $H_f = w_{i_1} \dots w_{i_k}(H_c^{(0)})$ *simulates* $H_c = w_1 \dots w_n(H_c^{(0)})$ w.r.t. $\mathbb{P}$ if $w_{i_1} \dots w_{i_k}$ simulates $w_1 \dots w_n$ w.r.t. $\mathbb{P}$.

▶ **Lemma 13.** *If H_f simulates H_c w.r.t. $\mathbb{P}$ and $\mathbb{P}(H_f)$ holds, then there exists some $0 \le i \le n$ s.t. $\mathbb{P}(H_c^{(i)})$.*

Proof. If $\mathbb{P}(H_c^{(0)})$, then $i = 0$ establishes the claim. Otherwise, let w_{i_j} be the first write to make $\mathbb{P}$ true in H_f, namely: let $1 \le j \le k$ be the minimal j such that $\neg\mathbb{P}(H_f^{(j-1)}$ but $\mathbb{P}(H_f^{(j)})$. If $\mathbb{P}(H_c^{(i_j-1)})$, take $i = i_j - 1$. Otherwise, $\neg\mathbb{P}(H_c^{(i_j-1)})$. So we have: $\neg\mathbb{P}(H_f^{(j-1)})$ but $\mathbb{P}(w_{i_j}(H_f^{(j-1)}))$ and $\neg\mathbb{P}(H_c^{(i_j-1)})$. From the premise that H_f simulates H_c it follows that $\mathbb{P}(w_{i_j}(H_c^{(i_j-1)}))$, and by taking $i = i_j$ the claim follows. ◀

▶ Remark (Information Between Predicates via Hindsight). In certain cases, it is useful to consider more than one predicate on the fabricated state. For example, when y is inserted as a child of x, the reachability of y is determined by the reachability of x before the write. To this end, we prove the translation from the fabricated state to the global state for all predicates of interest simultaneously. We use this approach in Appendix A.3.

A.1.2 From the Local View to the Fabricated State Using Upward-Absoluteness

Simulation lets us relate the fabricated state to the global state. Next, we relate the value of $\mathbb{P}$ on the local state to its value on the fabricated state. Recall that the fabricated state does not correspond exactly to the local view. Instead, $H_{lv} \subseteq H_f$. This gap is bridged when $\mathbb{P}$ is *upward-absolute* [43]:

▶ **Definition 14** (Upward-Absoluteness)**.** A predicate $\mathbb{P}$ is *upward-absolute* if for every pair of states H, H' such that $H \subseteq H'$, $\mathbb{P}(H) \implies \mathbb{P}(H')$.

Upward absoluteness is in line with the fact that an operation rarely observes all the contents of all memory locations. The procedure may have read only a partial view of memory. However, the decision based on $\mathbb{P}(H_{lv})$ must hold regardless of unobserved locations.

For example, $\texttt{root} \overset{k}{\rightsquigarrow} x$, for any location x and key k, is upward-absolute, because if a memory state contains a path then so does every extension of this state.

As we take H_f such that $H_{lv} \subseteq H_f$, we deduce from upward-absoluteness that if $\mathbb{P}(H_{lv})$ then $\mathbb{P}(H_f)$ holds, and, if H_f simulates H_c, then $\mathbb{P}(H_c^{(i)})$ holds for some i. This is summarized in the following theorem.

▶ **Theorem 15.** *If $\mathbb{P}$ is upward-absolute, $\mathbb{P}(H_{lv})$ holds, $H_{lv} \subseteq H_f$, and H_f simulates H_c w.r.t. $\mathbb{P}$, then there exists some $0 \le i \le n$ s.t. $\mathbb{P}(H_c^{(i)})$.*

A.2 The Fabricated State

In this section, we define the fabricated state H_f, obtained by a subsequence of writes $w_{i_1}, \ldots, w_{i_k}$. Our goal is to ensure that $H_{lv} \subseteq H_f$ and to relate H_f to H_c by simulation (Theorem 12).

In order to ensure that $H_{lv} \subseteq H_f$, it suffices to include in $w_{i_1}, \ldots, w_{i_k}$ all the writes that affected the reads that constructed the local view. However, ensuring simulation requires a more involved construction. Theorem 12 requires writes on the fabricated state to have a similar effect as in the global state. To achieve this, we choose the fabricated state to be as similar as possible to the global state from the perspective of the write and the predicate it might affect.

To this end, we develop a consistency condition of *forward-agreement*, based on a *partial order* on memory. The idea is that for each write w_{i_j} in $w_{i_1}, \ldots, w_{i_k}$, the write sees the same picture of the memory that comes forward in the data structure both in the fabricated state $H_f^{(j-1)}$ and in the corresponding global state $H_c^{(i_j-1)}$. We show how to construct the fabricated state so that this property holds. While this property alone does not suffice to imply simulation, in Appendix A.3 we show how together with an additional property (Theorem 24), forward agreement can be used to prove the simulation property w.r.t. predicates that track reachability along search paths.

Order on memory. Every state of the data structure induces a certain order on how operations read the different memory locations. To capture this, the user provides a mapping from a state H to a *partial order* that H induces on memory locations, denoted $\leq_H$.

▶ **Example 16.** In the running example of Fig. 1, the order $\leq_H$ on memory locations , i.e., fields of objects, is defined by following pointers from parent to children, i.e., all the fields of x.*left* and x.*right* are ordered after the fields of x, and the fields of an object are ordered by $x.key < x.del < \{x.left, x.right\}$.

We note that as $\leq_H$ depends on H, it changes with time (as the state changes). We make the following requirements on $\leq_H$, which the user also needs to establish (for Theorems 20 and 22 below).

Locality of the order. In order to ensure that the fabricated state is forward agreeing with the global state (Theorem 20), we require that the order $\leq_H$ is determined *locally* in the sense that 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 $\ell_1 \leq_{H'} \ell_2$. Note that the value in the target location does not affect the inclusion in the order; as an illustration, if $\leq_H$ is defined based on pointers, and ℓ_1 is a pointer to ℓ_2, making ℓ_2 an immediate successor of ℓ_1 in $\leq_H$, then this definition depends on the value in ℓ_1 but not on the value in ℓ_2.

Read in order. To enforce the fact that the order captures the order in which operations read the different memory locations, we require reads to the local view to respect the order. Formally, consider a read r in the sequence $\bar{r}$ reading the location ℓ from the global state $H_c^{(m)} = w_1 \ldots w_m(H_c^{(0)})$, and let $ReadSet$ be the set of locations read by earlier reads in $\bar{r}$. We require $ReadSet \leq^{\cup}_{w_1 \ldots w_m(H_c^{(0)})} \ell$. The read-in-order property holds if reads always read a location further in the order in the current global state. Namely, if ℓ' is the last location read into the local view, the next location ℓ read is such that $\ell' \leq_{H_c^{(m)}} \ell$.

Note that the read must respect the order of the global state but is oblivious to most of the global state; this is in line with the local nature of the order.

▶ **Example 17.** 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*.

Acyclicity of the accumulated order. 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 $\bigcup_{0 \leq s \leq m} \leq_{\hat{w}_1 \ldots \hat{w}_s(H_c^{(0)})}$. We require that the accumulated order $\leq^{\cup}_{w_1 \ldots w_n(H_c^{(0)})}$ is a partial order.

▶ **Example 18.** In the running example of Fig. 1, the accumulated order is constructed by collecting all pointer links created during an execution. As explained in Example 5 this relation is acyclic, and hence remains a partial order.

Construction of the fabricated state. We exploit the order to construct H_f so that it contains H_{lv}, but also satisfies the property of forward-agreement with the global state. Formally, we construct H_f in the following way: consider the reads that form H_{lv}. Each read r in $\bar{r}$ is of some location x_r in an intermediate memory state $H_c^{(m)} = w_1 \ldots w_m(H_c^{(0)})$. We take the writes that occurred *backwards* in time and modify locations *forward* in the accumulated order: $\mathit{precede\text{-}forward}(r) = \{w_j \mid j \leq m \wedge x_r \leq^{\cup}_{w_1 \ldots w_m(H_c^{(0)})} \mathit{mod}(w_j)\}$. The subsequence $\bar{w}_f = w_{i_1}, \ldots, w_{i_k}$ is taken to be the union of $\mathit{precede\text{-}forward}(r)$ for all r's in $\bar{r}$.

A.2.1 Forward-Agreement as a Step Toward Simulation

Next we formalize *forward-agreement* between the fabricated state and the global state. Forward-agreement requires that when a write in the subsequence $w_{i_1}, \ldots, w_{i_k}$ is performed on the intermediate fabricated state, the locations that come *forward* in the order — in the order induces by the before or after the writ — have *exactly* the same values as when the write is performed in global memory.

▶ **Definition 19** (Forward-Agreement). $H_f = w_{i_1} \ldots w_{i_k}(H_c^{(0)})$ is *forward-agreeing* with $H_c = w_1 \ldots w_n(H_c^{(0)})$ if for every $0 \leq j < k$,

$$\forall \ell.\ \left(\mathit{mod}(w_{i_j}) \leq_{H_f^{(j-1)}} \ell\right) \vee \left(\mathit{mod}(w_{i_j}) \leq_{H_f^{(j)}} \ell\right) \implies H_f^{(j-1)}(\ell) = H_c^{(i_j-1)}(\ell). \tag{1}$$

▶ **Lemma 20.** *If the order $\leq_H$ is determined locally, then H_f is forward-agreeing with H_c.*

For the proof of Theorem 20, we first show that, under the locality assumption, the accumulated order induced by a subsequence of writes is contained in the accumulated order of the entire sequence (this is important for relating the order in the fabricated and global state). We note that the accumulated order is trivially monotonic w.r.t. additional writes: $\leq^{\cup}_{w_1 \ldots w_i(H_c^{(0)})} \subseteq \leq^{\cup}_{w_1 \ldots w_{i+1}(H_c^{(0)})}$, but when considering a subsequence, the intermediate states do not coincide.

▶ **Lemma 21.** *If the order $\leq_H$ is determined locally, then for every sequence of writes $w_1, \ldots, w_n$ and every subsequence $w_{i_1}, \ldots, w_{i_k}$ operating on a state $H_c^{(0)}$,*

$$\leq^{\cup}_{w_{i_1} \ldots w_{i_k}(H_c^{(0)})} \subseteq \leq^{\cup}_{w_1 \ldots w_n(H_c^{(0)})}$$

Proof. Assume $\ell_1 \leq^{\cup}_{w_{i_1} \ldots w_{i_n}(H_c^{(0)})} \ell_2$, and let $x_1, \ldots, x_m$ be a sequence of locations such that $x_1 = \ell_1$, $x_m = \ell_2$ and for every $0 \leq i < m$, $x_i \leq_{w_{i_1} \ldots w_{i_s}(H_c^{(0)})} x_{i+1}$ for some $0 \leq s \leq k$. It suffices to show that $\leq_{w_{i_1} \ldots w_{i_s}(H_c^{(0)})} \subseteq \leq^{\cup}_{w_1 \ldots w_n(H_c^{(0)})}$, because this implies that $x_1, \ldots, x_m$ is an appropriate sequence for establishing that $\ell_1 \leq^{\cup}_{w_1 \ldots w_n(H_c^{(0)})} \ell_2$.

Assume therefore that $\ell_1 \leq_{w_{i_1} \ldots w_{i_k}(H_c^{(0)})} \ell_2$, and prove that $\ell_1 \leq^{\cup}_{w_1 \ldots w_n(H_c^{(0)})} \ell_2$. Since the order is discrete, there is a sequence $x_1, \ldots, x_m$ of locations such that $x_1 = \ell_1$, $x_m = \ell_2$ and x_{i+1}

is the immediate successor of x_i in $\leq_{w_{i_1}\dots w_{i_k}(H_c^{(0)})}$ for all $0 \leq i < m$. It suffices to show that $x_i \leq^{\cup}_{w_1\dots w_n(H_c^{(0)})} x_{i+1}$ for every $0 \leq i < m$.

Fix some i. Since $w_{i_1}, \dots, w_{i_k}$ is a subsequence of $w_1, \dots, w_n$, there is some intermediate state $w_1 \dots w_s(H_c^{(0)})$ that agrees on the value in the *single location* x_i, meaning $(w_{i_1} \dots w_{i_k}(H_c^{(0)}))(x_i) = (w_1 \dots w_s(H_c^{(0)}))(x_i)$. By the locality assumption, $x_i \leq_{w_1\dots w_s(H_c^{(0)})} x_{i+1}$. This implies that $x_i \leq^{\cup}_{w_1\dots w_n(H_c^{(0)})} x_{i+1}$. The claim follows. ◀

Proof of Theorem 20. Let w_{i_j} be a write in the subsequence. Let ℓ be a location s.t. $mod(w_{i_j}) \leq_{H_f^{(j-1)}} \ell$ or $mod(w_{i_j}) \leq_{H_f^{(j)}} \ell$. We need to show that $H_f^{(j-1)}(\ell) = H_c^{(i_j-1)}(\ell)$.

Let w_s the last write (in the global memory) to modify ℓ before w_{i_j}: take s to be maximal index such that $s < i_j$ and $mod(w_s) = \ell$. If there is no such s, $H_c^{(i_j-1)}(\ell) = H_f^{(j-1)}(\ell) = H_c^{(0)}(\ell)$, since ℓ is not modified by any preceding write. Otherwise, it suffices to show that w_s is also included in the subsequence, since w_s is performed on both H_f and H_c and there are no writes after w_s and before w_{i_j} to modify ℓ in the sequence of writes and therefore also in the subsequence.

By the construction of the subsequence, there is a read r of location x_r from the global state $H_c^{(m)}$ such that $i_j \leq m$ and $x_r \leq^{\cup}_{H_c^{(m)}} mod(w_{i_j})$. We have that $s < m$ and $x_r \leq^{\cup}_{H_c^{(m)}} mod(w_s)$ as $mod(w_s) = \ell$ and $mod(w_{i_j}) \leq^{\cup}_{H_c^{(m)}} \ell$ because $mod(w_{i_j}) \leq^{\cup}_{H_c^{(i_j)}} \ell$ (the accumulated order satisfies $\leq^{\cup}_{H_c^{(i_j)}} \subseteq \leq^{\cup}_{H_c^{(m)}}$ as $i_j \leq m$). Therefore, the construction includes w_s in the subsequence as well. ◀

In the next section we use forward agreement, together with an additional property, to show simulation for predicates defined by reachability along search paths.

A.2.2 Inclusion of the Local View in the Fabricated State

Next, we turn to showing that $H_{lv} \subseteq H_f$. For this purpose, we require that the definition of the order correctly captures the order of manipulation of the data structure. Namely, we require that the reads are performed in the order dictated by $\leq_H$, and that writes preserve the order in the sense that they do not introduce cycles in the accumulated order.

▶ **Lemma 22.** *If (1) the accumulated order $\leq^{\cup}_{\bar{w}(H_c^{(0)})}$ is a partial order, and (2) every sequence of reads satisfies the read-in-order property, then $H_{lv} \subseteq H_f$.*

Proof. Let x be a location in H_{lv}, and let r be the last read of x. Assume that r reads from the global state $H_c^{(m)}$. Let w_s be the write r *reads-from*, namely, w_s where s is the maximal index such that $s \leq m$ and $mod(w_s) = x_r$. By the construction, w_s is included in the subsequence, so let d be such that $i_d = s$. From this, $H_f^{(d)}(x) = H_{lv}(x)$. It remains to show that later writes in the subsequence do not modify this location, namely $mod(w_{i_j}) \neq \ell$ for all $d < j \leq k$.

Assume $mod(w_{i_j}) = x$ for some j. From the construction, w_{i_j} is included in the subsequence due to some read r' of location $x_{r'}$ from $H_c^{(m')}$ such that $i_j \leq m'$ and $x_{r'} \leq^{\cup}_{H_c^{(m')}} x$.

- If $r' \leq r$, $m' \leq m$ and thus $i_j \leq m$. Since $i_d = s$ was the maximal index so that $mod(x_s) = x$ and $mod(w_s) = x$, $i_j \leq i_d$.
- If $r' > r$, because reads respect the order, $x \leq^{\cup}_{H_c^{(m')}} x_{r'}$. Since the accumulated order is anti-symmetric, it follows that $x = x_{r'}$. But this is a contradiction to the fact that r is the *last* to read x_r.

The claim follows. ◀

A.3 Simulation w.r.t. Reachability by Search Paths

In this section we show how to prove the simulation property for predicates defined by *search paths*. We define a *preservation* property that complements forward-agreement to obtain a general proof of simulation for such predicates, showing that reachability by search paths transfers from the fabricated state to the global state. We then extend this result for predicates defined by reachability with checking another field, which is required for some operations.

Search paths. Intuitively, a k-search path in state H is a sequence of locations following $\leq_H$ that is traversed when searching for k. Formally, for a parameter k, a k-search path in a state H is a sequence of locations $\ell_1, \ldots, \ell_m$, with the following requirements:

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

We say that $\ell_1 \overset{k}{\rightsquigarrow} \ell_2$ holds in H if there exists a k-search path in state H that starts in ℓ_1 and ends in ℓ_2.

▶ **Example 23.** k-search paths in the tree of Fig. 1 are consists 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.

A.3.1 Simulation From Order and Preservation

Assuming a specific location serving as the entry point to the data structure, `root`, the predicate of reachability by a k-search path is $\mathbb{S}_{k,x} = \texttt{root} \overset{k}{\rightsquigarrow} x$.

▶ **Definition 24** (Preservation)**.** We say that $\bar{w}$ *ensures preservation of reachability by search paths* if for every $1 \leq m \leq n$, if for some $0 \leq i < m$, $H_c^{(i)} \vDash \mathbb{S}_{k,mod(w_m)}$ then $H_c^{(m-1)} \vDash \mathbb{S}_{k,mod(w_m)}$.

We now prove the simulation w.r.t. $\mathbb{S}_{k,x}$ between the fabricated and global state. We analyze *all* the reachability predicates for every parameter k of interest together: $\mathbb{S}_{k,x} \in \mathcal{PR}$ for every k and location x.

▶ **Lemma 25.** *If $\bar{w}$ ensures preservation of reachability by search paths, then the fabricated state H_f constructed in Appendix A.2 simulates H_c w.r.t. the predicate $\mathbb{S}_{k,x}$ for every k and location x.*

We prove the simulation property by induction on the index j of the write that changes the value of $\mathbb{S}_{k,x}$ on the fabricated state H_f. The proof idea is as follows. Given a write w that creates a k-search path to x in the fabricated state, we construct such a path in the corresponding global state. The idea is to consider the path that w creates in the intermediate fabricated state after w, divide it to two parts: the prefix until $mod(w)$, and the rest of the path. Relying on forward-agreement, the part of the path from $mod(w)$ to x is exactly the same in the corresponding global state, so we need only prove that there is also an appropriate prefix. But necessarily there has been a k-search path to $mod(w)$ in the fabricated state *before* w, so by the induction hypothesis (hindsight) applied on $mod(w)$, exploiting the fact that simulation up to $j-1$ implies that $\mathbb{S}_{k,x}(H_f^{(j-1)}) \implies \exists 0 \leq i \leq i_{j-1}.\ \mathbb{S}_{k,x}(H_c^{(i)})$

(see Theorem 13), there has been a k-search path to $mod(w)$ in some intermediate global state that occurred earlier than the time of w. Since w writes to $mod(w)$, the preservation property ensures that there is such a path to $mod(w)$ in the global state also at the time of the write, and the claim follows.

Proof. The proof is by induction on j, showing mutually simulation for w_{i_j} and that $\mathbb{S}_{k,x}(H_f^{(j)}) \implies \exists 0 \leq i \leq i_j.\ \mathbb{S}_{k,x}(H_c^{(i)})$ (see Theorem 13).

Let H_c such that $H_c \not\models \mathbb{P}$, and w a valid write on H_c. Let H_f be forward-agreeing with H_c, such that $H_f \not\models \mathbb{P}$ but $w(H_f) \models \mathbb{P}$. Our goal is to prove that $w(H_c) \models \mathbb{P}$.

Let π be the search path in $w(H_f)$. If π does not include $mod(w)$, it is also a search path in H_f (before the write), in contradiction to the premise that $H_f \not\models \mathbb{P}$. Let $\pi = \ell_1, \ldots, \ell_m$, and $\ell_p = mod(w)$.

Consider the prefix of π, $\ell_1, \ldots, \ell_{p-1}, \ell_p$. This is a k-search path in $w(H_f)$ (because π is, and by slicing), but the locations $\ell_1, \ldots, \ell_{p-1}$ are the same in H_f, so it is also a k-search path for in H_f, and so $H_f \models \texttt{root} \overset{k}{\rightsquigarrow} mod(w)$. Thus, there is some i such that $H_c^{(i)} \models \mathbb{P}$, and since w is valid on H_c, from preservation to $mod(w)$ it follows that $H_c \models \texttt{root} \overset{k}{\rightsquigarrow} mod(w)$. This is also true after the write: $w(H_c) \models \texttt{root} \overset{k}{\rightsquigarrow} mod(w)$ because the locations $\ell_1, \ldots, \ell_{p-1}$ are different from $mod(w)$ and thus remain the same after the write.

Now, $mod(w) = \ell_p, \ldots, \ell_m$ is a k-search path in $w(H_f)$. As that $mod(w) \leq_{w(H_f)} \ell_q$ for all $p \leq q$, from forward agreement we have that these locations have the same value also in $w(H_c)$. It follows that $w(H_c) \models mod(w) \overset{k}{\rightsquigarrow} x$.

Since $w(H_c) \models \texttt{root} \overset{k}{\rightsquigarrow} mod(w)$ and $w(H_c) \models mod(w) \overset{k}{\rightsquigarrow} x$, we have that $w(H_c) \models \texttt{root} \overset{k}{\rightsquigarrow} x$. ◀

We conclude:

▶ **Theorem 26.** *If $\leq_H$ is determined locally, every sequence of reads satisfies the read-in-order property, the accumulated order $\leq^{\cup}_{\bar{w}(H_c^{(0)})}$ is a partial order, and $\bar{w}$ ensures preservation of reachability by search paths, then for every k and location x, if $\mathbb{S}_{k,x}$ holds on $H_{\bar{w}}$, then there exists $0 \leq i \leq n$ s.t. $\mathbb{S}_{k,x}$ holds on $H_c^{(i)}$.*

A.3.2 Reachability with Another Field

We now extend the previous result to predicates involving not only the reachability of a location x by a search path but also a property of one of its fields. Let $\varphi(y)$ be a property of the value of field of an object y. Our goal is to establish properties of the form $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y) \wedge \varphi(y)$. As we show here, extending our results from plain reachability to reachability with an additional field is straightforward.

Such predicates are useful for the correctness of `contains`, which locates an element and checks without locks whether it is logically deleted, e.g., $\varphi(y) = y.del$ or $\varphi(y) = \neg y.del$. As another example, consider the predicate $\texttt{root} \overset{k}{\rightsquigarrow} x \wedge \varphi(x.next) = y$. The predicate says that there the *link* from x to y is reachable. Proving that exporting this from the local view of the traversal to the past of the concurrent execution is the key technical contribution of [37].

Our previous results allow to establish that $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y)$. Assume that $\bar{r}$ further reads the field of y and sees that $\varphi(y)$ holds. Since $\varphi(y)$ depends on a single location, this means that it holds *now*, at the time of the read. Our goal is to ensure that $\diamondminus(\texttt{root} \overset{k}{\rightsquigarrow} y \wedge \varphi(y))$ is also true, i.e., both $\texttt{root} \overset{k}{\rightsquigarrow} y$ and $\varphi(y)$ held at the same time in the past. The reasoning is as follows: Let $y.d$ be the field of on which $\varphi(y)$ depends. We have that at some point in the past $\texttt{root} \overset{k}{\rightsquigarrow} y$ holds. If $\varphi(y)$ also

held at that time, then we are done. Otherwise $\varphi(y)$ was false at the past but it is true now, when we read $y.d$. Therefore, a write w must have changed $y.d$. From preservation[5], $\texttt{root} \overset{k}{\rightsquigarrow} y$ holds also at the time of the write, so after w is performed it holds that $\texttt{root} \overset{k}{\rightsquigarrow} y \wedge \varphi(y)$.

[5] Technically, the important property of a field is that it is reachable iff the object it belongs to is reachable; $\texttt{root} \overset{k}{\rightsquigarrow} y.d \Rightarrow \texttt{root} \overset{k}{\rightsquigarrow} y$.

B Necessity of Conditions

In this section we illustrate how reasoning from unsynchronized traversals might be incorrect when the conditions of our framework are not satisfied, disabling its use to reason about concurrent executions from the sequential behavior of the local view. The examples are based on (artificial) modifications of the running example (Fig. 1).

Acyclicity. Consider a traversal of the tree in our running example searching for some key k, which starts from some arbitrary node y rather than from `root`. Assume that the traversal then reaches $null$. The traversal now starts from `root`, and happens to reach y. Based on the local view of the traversal, the operation declares that k is not present in the tree, since $\texttt{root} \overset{k}{\rightsquigarrow} null$ holds in the local view (as $y \overset{k}{\rightsquigarrow} null$ and $\texttt{root} \overset{k}{\rightsquigarrow} y$ were found).

However, this may not be true for any intermediate state of the concurrent execution: assume that the parent of y when the traversal begins is x and that $x.key = k$. The scenario above is possible if between the first and second phases of the traversal x, y are rotated (see Fig. 2a), although a node with key k is always present in the tree.

Note that since the modifying procedures are exactly as in the running example, preservation holds in this example. The search order in this example includes edges from $null$ to `root`, and this is of course not acyclic.

Preservation. Consider a binary search tree in which rotations are performed in-place, and a traversal exactly as `locate` in Fig. 1. Consider reachability to x when y, x are rotated (see Fig. 2a). If the traversal reaches y before the rotation, then the in-place rotation takes place, and the traversal continues, the traversal "misses" x, and could mistakenly declare that the key of x is not contained in the tree.

Note that the traversal by pointers property is maintained. The acyclicity conditions does not hold (in an insubstantial way, see Sec. 7) but this does not affect the view of the traversal depicted here. (It does affect the traversal if it reaches x instead of y when a rotation takes place.)

$$
\begin{array}{ll}
Root(H,H') & \stackrel{\text{def}}{=} \forall o_1.\ H \vDash \texttt{root} = o_1 \implies H' \vDash \texttt{root} = o_1 \\
Key(H,H') & \stackrel{\text{def}}{=} \forall o_1, k.\ H \vDash o_1.key = k \implies H' \vDash o_1.key = k \\
Rem(H,H') & \stackrel{\text{def}}{=} \forall o_1, k.\ H \vDash o_1.rem \implies H' \vDash o_1.rem \\
Acyclic(H,H') & \stackrel{\text{def}}{=} \forall o_1, o_2, o_3.\ ((H \vDash \texttt{root} \rightsquigarrow o_1 \wedge child(o_1,o_2)) \wedge (H' \vDash child(o_1,o_3) \wedge o_2 \neq o_3)) \\
 & \qquad \implies H \vDash o_2 \rightsquigarrow o_3 \vee H \vDash (new(o_3) \wedge \neg o_3 \rightsquigarrow o_1) \\
Preservation(H,H') & \stackrel{\text{def}}{=} \forall o_1.\ H \vDash \texttt{root} \overset{k}{\rightsquigarrow} o_1 \implies H' \vDash \texttt{root} \overset{k}{\rightsquigarrow} o_1 \vee H' \vDash o_1.rem \vee H' \vDash locked(o_1) \wedge \neg root \rightsquigarrow o_1 \\
 & \qquad H \vDash \neg root \rightsquigarrow o_1 \implies H' \vDash locked(o_1) \vee H' \vDash o_1.rem
\end{array}
$$

Figure 4 Transition invariants for the running example. The variables o_1, o_2, and o_3 are interpreted over allocated objects in the heap. The predicate $o_1 \rightsquigarrow o_2$ denotes the fact o_2 is reachable from o_1 in the heap (by some sequence of accesses to *left* or *right*). Also, $locked(o_1)$ means that the procedure executing the current step holds a lock on o_1, and $new(o_3)$ means that o_3 was allocated by the procedure executing the current step and never "made" reachable from the `root` (i.e., its address was never stored into some *left* or *right* field).

C Formally Justifying the Validity of Local View Arguments for the Running Example

We discuss a particular strategy for proving the conditions of Sec. 3.3 above which applies in particular our running example. The acyclicity condition refers to a partial order $\leq_H$ on memory locations in a heap H which in the case of the example is defined by

$$\forall o_1, o_2.\ child(o_1,o_2) \implies \forall f \in \{keo_2, left, right, del, rem\}.\ o_1.f \leq_H o_2.f$$
$$\forall o_1.\ o_1.key \leq_H \{o_1.rem, o_1.del\} \leq_H \{o_1.left, o_1.right\}$$

where $child(o_1,o_2)$ means that o_2 is a child of o_1. We may write $o_1 \leq_H o_2$ to say that all the fields of o_1 are before the fields of o_2 in $\leq_H$.

The acyclicity condition follows from the *transition invariant* $Acyclic(H,H')$ in Fig. 4, which describes a relation between the state H before and the state H' after executing any statement in the code of the algorithm (in any concurrent execution). According to this invariant, if some assignment changes the child of some object o_1, reachable from the root, from o_2 to o_3, then o_3 was either reachable from o_2 (which implies $o_1 \leq_H o_3$) or o_3 is a "new" object which was never reachable from the $root$ and o_1 is not reachable from o_3 (which implies $o_3 \not\leq_H o_1$). In both cases, adding the constraint o_1 is "smaller than" o_3 to the partial order $\leq_H$ will not introduce a cycle. Proving the validity of this invariant is rather easy. The only updates to the children of a node reachable from the `root` (when they are not `null`) are from `removeRight()` at Line 74 and Line 76, and from `rotateRightLeft()` at Line 94 and Line 95 (for all the other updates, the invariant holds vacuously). Notice that any property of fields of objects which are locked is true in concurrent executions as long as it holds in sequential executions. For instance, $y = z.right$ holds at line Line 74 and Line 76, and it implies on its own the invariant. A similar reasoning can be done at Line 95. For Line 94, the new child z of x is a newly allocated object whose children are not on a path to x.

The transition invariant $Preservation(H,H')$ in Fig. 4 implies a property which is even stronger than the preservation condition: for every execution e of the concurrent algorithm, and every update w in e to a heap object o, if $\texttt{root} \overset{k}{\rightsquigarrow} o$ became true at some moment before w, then it remains true until w gets executed (there is no requirement that w overlaps in time with a read-only code fragment). According to $Preservation(H,H')$, every other write w' that happens after the moment when $\texttt{root} \overset{k}{\rightsquigarrow} o$ became true will either maintain the validity of this predicate, or it will turn it to false, but then it will either hold a lock on o or set its field rem to *true* (and according to $Rem(H,H')$ the field rem never changes from *true* to *false*). The latter two cases are impossible since w is enabled only if

it holds a lock on o and the field rem is *false*. Concerning the proof of $Preservation(H, H')$, since the field key of every object, and the variable `root` are immutable (stated formally in $Key(H, H')$ and $Root(H, H')$), the only way to modify the validity of a predicate `root` $\overset{k}{\rightsquigarrow}$ o is by changing the pointer fields $left$ or $right$. The only such updates occur in the procedures `removeRight()` and `rotateRightLeft()`. These updates can modify search paths only by removing or inserting keys, the updates at Line 74, Line 76, and Line 95 remove the key of `y` while the update at Line 94 inserts the key of `y` (or `w`). The interesting case is when keys are removed from search paths: a predicate `root` $\overset{k}{\rightsquigarrow}$ o can become *false*, but only if o contains the removed key and o becomes unreachable from the `root`. Also, this can happen only if the procedure executing the current step holds a lock on o or if the field rem is set to *true* (an unlock validates $Preservation(H, H')$ since it can happen only when the field rem is already set).

D Example: Lock-Free List-Based Concurrent Set

In this section, we apply our approach to verify the lock-free list-based concurrent set algorithm shown in Fig. 5. The code of the algorithm is based on the algorithm of [28, Chapter 9.8], adapted to our language. The algorithm is explained in detail in [28]. Thus, we only describe the parts necessary to understand our linearizbility proof and the assertions, written inside curly braces, which annotate the code of the `contain` procedure.

The set algorithm uses an underlying sorted linked-list of dynamically-allocated objects of type `LFN`, which we refer to as *nodes*. Every node has three fields: an immutable integer field `key` storing the key of the node, a pointer field `ref` pointing to a successor node (or to a designated `null` value), and a boolean field `mark` indicating that the node was logically deleted from the list.

The `ref` and `mark` fields of a node can be accessed atomically: we encapsulate these fields inside a pair field `next` comprised of a reference and a boolean field which can be accessed atomically. We write `x.next.ref` and `x.next.mark` to denote accessing the `ref` and `mark` fields of the node pointed to by `x` separately. We use the notation $(\texttt{succ}, \texttt{marked}) \leftarrow \texttt{x.next}$, where `succ` is a pointer variable and `marked` a boolean variable, to denote an atomic assignment of `x.next.ref` and `x.next.mark` to `succ` and `marked`, respectively. Similarly, we write $\texttt{x.next} \leftarrow (\texttt{succ}, \texttt{marked})$ to denote an atomic assignment to the two components of the `next` pair of fields of the node pointed to by `x`. We write $\texttt{CAS}(\&\texttt{x.next}, (\texttt{currref}, \texttt{currmark}), (\texttt{newref}, \texttt{newmark}))$ to denote a *compare-and-set* operation which atomically sets the contents of the `next` field of `x` to $(\texttt{newref}, \texttt{newmark})$, provided that its current value is $(\texttt{newref}, \texttt{newmark})$. When the `mark`-component of the `next` field of a node is set, we say that the field, as well as the node itself, are *marked*, otherwise, we say that they are *unmarked*.

▶ Remark 4. The original code [28, Figures 9.24 to 9.27] is written in Java and keep the `ref` and `mark` fields of node using a *markable pointer* (see [28, Pragma 9.8.1]) to allow reading, writing, and applying CAS to the `ref` and `mark` fields simultaneously.

The list has designated sentinel *head* and *tail* nodes. The *head* node is always pointed to by the shared variable `head` and contains the value $-\infty$. The *tail* node is always pointed to by the shared variable `tail`, and contains the value ∞. The value $-\infty$ (resp. ∞) is smaller (resp. greater) than any possible value of a key. When the algorithm starts, it first sets *tail* to be the successor of *head* and sets *tail*'s successor to be `null`. The sentinel nodes remain unmarked throughout the execution.

The set algorithm is comprised of three interface procedures: `add`, `remove`, and `contains`. The first two use the internal `find` procedure to traverse the list and prune out marked nodes: when `find` is invoked to locate a key `key`, it traverses the list starting from the *head* node until it reaches an unmarked node with a key greater than `key`. During the traversal it removes marked nodes (Line 207). If an attempt to remove a node fails, the procedure restarts. In contrast, the `contains` method's traversal of the list is *optimistic*: it is done without any form of synchronization. As a result, while a thread is traversing the list, other threads might concurrently change the list's structure. When verifying this algorithm, our approach helps in proving that `contains` is linearizable, which is the most difficult part of the proof. Proving the linearizability of `add` and `remove` can be done using a rather standard invariant-based concurrent reasoning as discussed below.

Verifying assertions using invariants-based concurrent reasoning

Procedures `add`, `remove`, and `find` maintain several *state* invariants:[6]

(I_{rT}) the tail node is always reachable from the head node, where reachability between nodes is determined in this section by following `ref` fields;

[6] Recall that the `contains` does not modify the shared state.

```
161 type LFN
162   immutable int key
163   LFN×bool next=⟨ref,mark⟩

165 LFN tail←new(+∞,null)
166 LFN head←new(−∞,tail)


167 bool add(int key)
168   LFN newNode, pred, succ

170   (pred,succ)←find(key)
171   if (succ.key = k)
172     return false

174   newNode←new SNL(key,(succ,false))
175   bool added←CAS(&pred.next,(succ,false),(newNode,false))
176   if (¬added)
177     restart

179   return true


180 bool remove(int key)
181   LFN newNode, pred, succ

183   (pred,succ)←find(key)
184   if (succ.key ≠ k)
185     return false

187   LFN nodeToRemove←succ
188   (succ,bool)←nodeToRemove.next

190   while (true)
191     bool iMarkedIt←CAS(&nodeToRemove.next,(succ,false),(succ,true))
192     (succ,marked)←nodeToRemove.next
193     if (iMarkedIt)
194       find(key)
195       return true
196     if (marked)
197       return false


198 LFN×LFN find(int key)
199   bool find, snip, marked, cont←true
200   LFN pred←null, curr←null, succ←null
201   while (cont)
202     pred←head
203     curr←pred.next.ref
204     while (cont)
205       (succ,marked)←curr.next
206       while (marked)
207         snip←CAS(&pred.next,(curr,false),(succ,false))
208         if (¬snip)
209           restart
210         curr←succ
211         (succ,marked)←curr.next
212       if (curr.key < key)
213         pred←curr
214         curr←succ
215       else
216         cont←false
217   return (pred,curr)


218 bool contains(int key)
219   LFN curr, succ
220   bool marked←false

222   curr←head
223   succ←pred.next.ref
224   while (curr.key < key)
225     {⟐(head ⇝^key curr ∧ (curr.mark ⟺ marked))}
226     curr←succ
227     (succ,marked)←curr.next

229   {⟐(head ⇝^key curr ∧ (curr.mark ⟺ marked)) ∧ key ≤ curr.key}
230   return (curr.key=key ∧ ¬marked)
```

Figure 5 Lock-free concurrent list [28, Chapter 9.8]

(I_{UB}) all unmarked nodes are reachable from the *head* node; and
($I_<$) if node v is the `ref`-successor of node u then the key of v is strictly greater than that of u.
In addition, the procedures maintains the following *transition* invariants:
(δ_k) the `key` node is immutable;
(δ_{mn}) the `next` fields becomes immutable once it gets marked; and
(δ_{mr}) right after a node gets marked, it is reachable from the head.
Verifying the invariants hold is rather straightforward as it merely requires local reasoning about each mutation. For example, it is easy to see that invariant δ_{mn} holds: The modifications (Lines 175, 191 and 207) are done using a `CAS` operation which may succeed only if the modified `next` pair of fields is unmarked. Furthermore, a node u gets marked only in Line 191. Hence, u is reachable at that time. Note that the compare-and-set command cannot affect the `ref`-field of u's predecessor.

Verifying linearizability

We prove the linearizability of the algorithm using an *abstraction function* $\mathcal{A} : H \to \wp(\mathbb{N})$ that maps a concrete memory state of the list to the *abstract set* represented by this state, and showing that `add` and `delete` manipulate this abstraction according to their specification and that `find` does not modify it. We define $\mathcal{A}$ to map H to the set of keys of the unmarked nodes. Note that by invariants I_{rT}, I_{UB}, and $I_<$, these nodes are part of the sorted list segment connecting the *head* and *tail* nodes. We refer to this list segment as the *backbone list*.

Verifying linearizability of add and remove The proof that `add` and `remove` are linearizable follows directly from the invariants once we establish the following properties of `find`: (a) it does not change the abstract set represented by the list, and (b) the pointers (`pred`,`curr`) it returns point to nodes *pred* and `curr`, respectively, such that (i) the key of *pred* (resp. *curr*) is smaller than (resp. greater or equal to) `key`, (ii) *pred* was unmarked and the `ref`-predecessor of *curr* at some point during the traversal, and (iii) at some (perhaps different) time point during the traversal, *curr* was unmarked.

To verify property (a), we observe that `find` removes the marked node pointed to by `curr` by redirecting the `ref`-field of its predecessor (pointed to by `pred`) to point to `succ`—`curr`'s `ref`-successor: Using compare-and-set (Line 207) ensures that the removal succeeds only if `pred` is unmarked and its `ref`-field points to `curr`. The fact that `succ` is the `ref`-successor of `curr` is ensured by transition invariant δ_{mn} which prohibits the modification of marked node. Property (a) holds because cutting out marked nodes this way does not affect the reachability of unmarked nodes from the head.

Property (b.i) follows from the check made in Line 212 and the immutability of keys. To verify properties (b.ii) and (b.iii), it suffices to observe that after `find` traverses the `next`.`ref`-field of the node *curr* pointed to by `curr` (Lines 205 and 211) it ensures that *curr* is unmarked (Line 206) before it updates `pred` and `curr` (Lines 213 and 214).[7]

The linearizability of invocations of `add` and `remove` which return `true` follows from invariant I_{UB} which ensures that as they modify unmarked nodes, these nodes must be reachable from the head. This, together with property (a), shows that adding a new node or marking an existing unmarked one affects the represented set in the intended way.

The linearizability of invocations of `add` which return `false` follows from property (b.iii) and the check made in Line 170: The former ensures that the node pointed to by `curr` was unmarked

[7] Recall that the `ref` and `mark` fields are read in one atomic action.

during the traversal of `find` and the latter that the key of that node is the one the procedure attempts to add.

The linearizability of *unsuccessful* invocations of `remove`, i.e., ones which return `false`, can justified using two different reasons:

- `remove` returning `false` in Line 185 can be justified by properties (a) and (b.ii), which, together, ensure that there was some point during the execution of `find` in which the node *pred* pointed to by `pred` was reachable from the head and its successor was the node *curr* pointed to by `curr`. Hence, the latter was reachable too. As the key k the procedure tries to remove is bigger than the key of *pred* and smaller than the key of *curr*, the sortedness of the list ensures that at this time k was not the key of any unmarked node.
- `remove` returning `false` in Line 197 can be justified by property (a), the check made in Line 170, and property (b.iii): The first two ensure that the node pointed to by `curr` was unmarked during the traversal of `find` and the last one was that at a later point this node was marked. By transition invariant δ_{mr}, it follows that at some time point during the execution of `remove` a marked node with the key removes attempts to delete was reachable from the head, and as the list is sorted, that key was not part of the set the list represents.

Verifying linearizability of contains We use our framework to verify the linearizability of `contains` by defining the notion of an order over memory locations, the notion of *valid search path* for key k that starts at the *head* node, and proving that the code satisfies the acyclically and preservation conditions.

As in our running example, we define the order over memory locations based on reachability. We say that there is a valid search path to an object o_x for key k from the head of the list, denoted by $\texttt{head} \overset{k}{\rightsquigarrow} o_x$, if o_x is reachable from the head node to o_x and its key is smaller than k. Formally, search paths are defined as follows:

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

The acyclicity of the order stems from the immutability of keys and invariant $I_<$ which ensures that cycles are impossible.

To prove the preservation of search paths to locations of modification it suffices to note that as marked nodes are never modified, it suffices to show the property hold for unmarked ones. Note that neither marking a nor changing the successor of a node affects the search paths which go *through* it: Invariant $I_<$ ensures that adding a node v in between nodes u and w does not break any search paths which goes through w: These must be for keys greater than that of w, and hence of v. Removing a marked node may merely shorten a search path to an unmarked node. Marking a node has not effect of search paths.

To verify the linearizability of `contains` we establish the loop invariant

$$\texttt{head} \overset{key}{\rightsquigarrow} curr \wedge (curr.mark \iff marked)$$

in Line 225 using sequential reasoning. This is straightforward. We then lift it to concurrent executions (using its past form) by applying the extension of our framework discussed in Remark 2. This invariant, together with the definition of a search path ensures that as we get out of the loop only if we reached an unmarked node with the key that we look for, and in this case `marked` is false, or that during the traversal we never encountered that key that we look for or that this key was in a marked node. In either cases, the return value correspond to the contents of the abstract set at that time.

```
231 type SLN
232   immutable int key
233   immutable int topLevel
234   SLN × bool next[L]

236 SLN tail←new(+∞,L,(null,false),...,(null,false))
237 SLN head←new(−∞,L,(tail,false),...,(tail,false))
```

Figure 6 Lock-free concurrent skiplist: Type declaration and the `head` and `tail` global variables pointing to the first and last, respectively, sentential nodes of the list. (See [28, Fig 14.10]).

E Example: Lock-Free Skiplist-Based Concurrent Set

In this section, we apply our approach to verify the lock-free skiplist-based concurrent set algorithm shown in Fig. 6 to 10. The code of the algorithm is based on the algorithm of [28, Chapter 14.4], adapted to our language. The algorithm is explained in detail in [28]. Thus, we only describe the parts necessary to understand our linearizbility proof and the assertions, written inside curly braces, which annotate the code of the `contain` procedure (Fig. 10).

The set algorithm uses an underlying concurrent skiplist comprised of dynamically-allocated objects of type `SLN` (see Fig. 6), which we refer to as *nodes*. Every node has three fields: an immutable integer field `key` storing the key of the node, an array `next` with L entries, where each entry contains a pair comprised of a pointer field `ref` and a boolean field `mark` which allows to link every node in multiple levels, and an integer field `topLevel` which determines the number of populated entries. We refer to the list obtained by the links at the ith entry of the nodes array of links as the list at level i. Roughly speaking, every node u is part of lock-free lists (see Appendix D) at levels $0..u.\texttt{topLevel}$. The bottom list (`next`[0]) is the *main* list, and every list i, where $0 < i \leq u.\texttt{topLevel}$, serves as a shortcut which allows to bypass multiple nodes of the list at level $i-1$.

The skiplist has designated sentinel *head* and *tail* nodes. The *head* node is always pointed to by the shared variable `head` and contains the value $-\infty$. The *tail* node is always pointed to by the shared variable `tail`, and contains the value ∞. The value $-\infty$ (resp. ∞) is smaller (resp. greater) than any possible value of a key. When the algorithm starts, it first sets *tail* to be the successor of *head* in all L levels, and initializes all of *tail*'s `ref`-fields to be `null`. The sentinel nodes remain unmarked throughout the execution.

The set algorithm is comprised of three interface procedures: `add`, `remove`, and `contains`. The first two use the internal `find` procedure to traverse the list and prune out marked nodes. In contrast, the `contains` method's traversal of the list is *optimistic*: it is done without any form of synchronization. As a result, while a thread is traversing the list, other threads might concurrently change the list's structure. When verifying this algorithm, our approach helps in proving that `contains` is linearizable, which is, as in the case of the lock-free list, the most difficult parts of the proof. Proving the linearizability of `add` and `remove` follows rather easily using invariant-based concurrent reasoning as discussed below.

Verifying invariants using concurrent reasoning

Procedures `add`, `remove`, and `find` maintain several *state* invariants.[8] In particular, for every $i = 0..L$, the list at level i maintain all the state and transition invariants of the lock-free list-based set

[8] Recall that the `contains` does not modify the shared state.

algorithm (see Appendix D). In addition, the skip list maintain the following state invariants:
(I_{mu}) if the `next`$[i]$ field of a node u is marked then so are all the fields u.`next`$[j]$ for $i < j \leq L$.
(I_{sub}) if node u precedes node v at level i and both nodes are unmarked at level $i$at level i then u precedes v at level j for any $0 \leq j < i$.

Verifying the invariants hold is rather straightforward as it merely requires local reasoning about each mutation. For example, it is easy to see that the `next` field of a marked node is never modified: The modifications (Lines 252, 277, 286, 289, 311 and 315) are done using a `CAS` operation which may succeed only if the modified `next` pair of fields is unmarked. To verify invariant I_{mu}, we only need to observe that `remove` marks the entries of the `next` array from top to bottom (Line 315). Note that as a marked `next`-field never gets modified, invariant I_{mu} holds even if a thread tries to remove a node which is still being added to the list. To verify invariant I_{sub}, we first observe that `add` links a new node in (level-wise) a bottom up fashion (Line 289). Thus, a node v gets linked to the ith level, for any $0 \leq i < L$ before it gets linked in level $i + 1$. We then apply invariant I_{mu} to realize that if nodes u and v are unmarked at level i then they are unmarked at level $i + 1$. By invariant I_{UB} of the lock free list (see Appendix D), this means that the node is reachable at the lists at level $i + 1$ and at level i. By $I_{<}$, the lists at all levels are sorted. Thus, if u precedes v at level i it precedes is at level $i - 1$ too.

Verifying linearizability

We prove the linearizability of the algorithm using an *abstraction function* $\mathcal{A} : H \to \wp(\mathbb{N})$ that maps a concrete memory state of the list to the *abstract set* represented by this state, and showing that `add` and `delete` manipulate this abstraction according to their specification and that .`find` does not modify it. We define $\mathcal{A}$ to map H to the set of keys of the unmarked nodes of the main (bottom) list. Note that by invariants I_{rT}, I_{UB}, and $I_{<}$), these nodes are part of the *backbone list* of the main list—the sorted list connecting the *head* and *tail* nodes.

Verifying linearizability of add and remove The proof that `add` or `remove`, shown in Fig. 8 and 9, are linearizable follows almost immediately from the invariants once we establish the following properties of `find`, shown in Fig. 7: (A) `find` populates the pair of input arrays with pointers to predecessors and successors of the searched key at every level, and (B) it returns `true` if and only if it found an unmarked node at the bottom list containing the searched node. Furthermore, in this case, it sets `succs`$[0]$ to point to this node.

To prove property (A), we note that, roughly speaking, the `find` procedure of the skiplist traverses the lists at all the levels starting from the top lists and making its way down to the bottom list. It fills the ith entry of the `preds` and `succs` arrays, for every $0 \leq i \leq L$, at with pointers to the predecessor and successor nodes of the search key at the list at level i. As in the `find` procedure of the lock-free list, it prunes out marked nodes as it goes over the lists (Line 244). The most tricky aspect of `find` is that its traversal at level $i - 1$, for $0 < i \leq L$, does not start from the head of the list but from the predecessor node of the searched key at level i. `find` ensures that it does not miss a node u containing the desired key which is unmarked at the bottom level by switching the traversal from level i to level $i - 1$ in a node *pred* which, being a predecessor node, has a smaller key than the one `find` searches for and is unmarked at level $i - 1$ (Lines 245 and 246). By invariant I_{sub}, *pred* is unmarked at all levels $0..i - 1$, by invariant I_{UB}, u is on the backbone of the list at levels $0..i - 1$, and by invariant $I_{<}$, u appears after *pred* at these lists.

To prove property (B), we observe that `find` gets out of the `for`-loop only after it set `succs`$[0]$ to point to a node with a key greater or equal to the searched key (Lines 256 and 262) which was unmarked during the traversal (Line 251).

```
240 bool find(int key, SLN preds[L], SLN succs[L])
241   bool find, snip, marked, cont
242   SLN pred←null, curr←null, succ←null
243   pred←head
244   for (int level←L; 0 ≤ level; level−−)
245     (curr,marked)←pred.next[level]
246     if (marked)
247       restart
248     cont←true
249     while (cont)
250       (succ,marked)←curr.next[level]
251       while (marked)
252         snip←CAS(&pred.next[level],(curr,false),(succ,false))
253         if (¬snip)
254           restart
255         (succ,marked)←curr.next[level]
256       if (curr.key < key)
257         pred←curr
258         curr←succ
259       else
260         cont←false
261     preds[level]←pred.
262     succs[level]←succ
263   return curr.key=key
```

Figure 7 Lock-free concurrent skiplist algorithm: Procedure `find`. (See [28, Fig 14.13]).

▶ Remark. We modified the `add` procedure of [28, Fig 14.11] to update the successors of a newly added node using a compare-and-swap (Line 286) because the original version has a subtle race between concurrent `add` and `remove` of the same node. Also, following [**?**], we added to procedure `find` a check that the node `pred` in which the traversal switches to a lower level (Line 245) is unmarked at the new level (Line 246). This simplified the proof of `find`, which is done outside our framework. The `find` procedure of [28, Fig 14.13] does not make this check, which is indeed unnecessary: As `pred` was checked to be unmarked at the previous, higher, level, invariant I_{mu} ensures that at the time it was unmarked at the new, lower, level.

The proof of linearizability of `add` and `remove` is carried out essentially in the same way it is done for the lock-free list (see Appendix D) when applied to the main list. It is possible to do so because property (A) ensures that the pair of pointers $(\texttt{preds}[0], \texttt{succs}[0])$ returned by the skiplist's `find` fulfills the same conditions as the pair $(\texttt{pred}, \texttt{curr})$ returned by the lock-free list's `find`. (See property (b) in Appendix D).

Verifying linearizability of contains We use our framework to verify the linearizability of `contains`, shown in Fig. 10, by defining the notion of an order over memory locations, the notion of *valid search path* for key k that starts at the top level list entry of the *head* node, and proving that the code respect the acyclically and preservation conditions.

Recall that the order is defined for memory locations, i.e., at the granularity of fields. We define the order over memory locations containing entries of the `next` arrays of nodes in the following way: All locations pertaining to entries at level i are smaller than the ones at level j for any $i < j$. We define the order between locations pertaining to entries at the same level according to reachability, as we did in the case of the lock-free list (see Appendix D). The other fields are immutable, and thus their order is immaterial. In particular, it is easy to modify the code so that the key field of a node is read only once in a traversal when moving between levels; this is of course equivalent.

We say that there is a *valid search path* to a location to the ith entry of the `next` array of the node pointed to by x, denoted by $\texttt{head} \overset{k}{\rightsquigarrow} x.next_i$, if it is possible to reach from the top-level `next`-field

```
264 bool add(int key)
265   int topLevel←random(0..L)
266   SLN[] preds← new SLN[L]
267   SLN[] succs← new SLN[L]
268   SLN newNode, succ, newSucc

270   bool found←find(key,preds,succs)
271   if (found)
272     return false

274   newNode←new SNL(key,topLevel,null,...,null)
275   (succ,marked)←succs[0]
276   newNode.next[0]←(succ,marked)
277   bool setnext←CAS(&pred.next[0],(succ,false),(newNode,false))
278   if (¬setnext)
279     restart

281   for (int level←1; level ≤ L; level++)
282     bool linked←false
283     while (¬linked)
284       succ←succs[level].ref
285       newSucc←newNode.next[level].ref
286       setnext←CAS(&newNode.next[level],(newSucc,false),(succ,false))
287       if (¬setnext)
288         return true
289       linked←CAS(&preds[level].next,(newNode,false),(succ,false))
290       if (¬linked)
291         bool newMark←newNode.next[level].mark
292         if (newMark)
293           return true
294         find(key,pred,succ)
295    return true
```

Figure 8 Lock-free concurrent skiplist algorithm: Procedure `add`. (See [28, Fig 14.11]).

```
296 bool remove(int key)
297   SLN[] preds← new SLN[L]
298   SLN[] succs← new SLN[L]
299   SNL succ
300   bool marked, found
301   int level

303   found←find(key,preds,succs)
304   if (¬found)
305     return false

307   SLN nodeToRemove←succs[0]
308   for (level ←nodeToRemove.topLevel; 1 ≤ level; level−−)
309     (succ,marked)←nodeToRemove.next[level]
310     while (¬marked)
311       CAS(&nodeToRemove.next[level],(succ,false),(succ,true))
312       (succ,marked)←nodeToRemove.next[level]

314   while (true)
315     bool iMarkedIt←CAS(&nodeToRemove.next[level],(succ,false),(succ,true))
316     (succ,marked)←nodeToRemove.next[0]
317     if (iMarkedIt)
318       find(key,preds,succs)
319       return true
320     if (marked)
321       return false
```

Figure 9 Lock-free concurrent skiplist algorithm: Procedure `remove`. (See [28, Fig 14.12]).

of the node `head` to that entry by either (i) traversing over links at the same level if they originate from nodes which are either marked at that level or that their key is smaller than k or (ii) descend to a lower level entry in a node whose key is not greater than k and that it is unmarked at the current level. Formally, search paths are defined as follows:

$$o_r, i \overset{k}{\rightsquigarrow} o_x, j \overset{\text{def}}{=} \exists o_0, \iota_0, \ldots, o_m, \iota_m.\, o_0 = o_r \wedge \iota_0 = i \wedge o_m = o_x \wedge \iota_m = j \wedge$$
$$\forall i = 1..m.\ \mathrm{nextNode}(o_{i-1}, \iota_{i-1}, k, o_i, \iota_i)\ , \text{ and}$$
$$\mathrm{nextNode}(o_{i-1}, \iota_{i-1}, k, o_i, \iota_i) =$$
$$(o_{i-1}.next_{i-1} = o_i \wedge (o_{i-1}.mark \vee o_{i-1}.key < k)) \vee$$
$$(o_{i-1} = o_i \wedge (\neg o_{i-1}.mark \wedge o_{i-1}.key \leq k)\ .$$

Note that a search path to k might go at the bottom level through nodes with a key greater than k, but these nodes must be marked. Also note that if $\texttt{head} \overset{k}{\rightsquigarrow} x.next_0$ holds and $x.key > k$ this indicates that k is not in the abstract set represented by the list (because a valid search path to k does not continue past a node with key k and all the list elements are linked in the lowest level).

The acyclicity of the order stems from the immutability of keys, the fact that the order between entries at different levels never changes, and invariant $I_<$ of the lock-free list which ensures that cycles are impossible between entries at the same level.

To prove the preservation of search paths to locations of modification it suffices to note that as marked nodes are never modified, it suffices to show the property hold for unmarked ones. Note that neither adding a link nor removing one changes the search paths which goes *through* unmarked `next` fields: Only marked `next`-fields are removed (Lines 244, 311 and 315). As no search path goes from these nodes directly to a lower level entry, this change may only shorten same-level search paths. Adding a `next`-field into a list at level l (Lines 277 and 289) does not break a search path that used to go through its predecessor because: the predecessor field remains unmarked (thus there is no effect of search paths that goes down a level) and, as discuss in Appendix D, the list at level l remains sorted, and thus the key of the node is smaller than all the keys in the following nodes. Marking a node v adds search paths which go through the marked links, but does not remove any. However, it may remove search paths that used to switch at v to a lower level. Luckily, as the *head* node is never marked, invariant I_{sub} ensures that for any such search path which was removed there is another valid search path which goes through an unmarked level-l-predecessor u of v that gets to the $l-1$ `next` entry of v by going down a level at u and going through l-level links to get to the location of $u.\texttt{next}[l-1]$'s entry.

To verify the linearizability of `contains`, we establish three loop invariants. (Using sequential reasoning, establishing the present form of these invariants is straightforward.)

- The outer (`for`) loop invariant ensures that in Line 327, when the procedures starts traversing a new level, `pred` points to a node which at some point in time during its traversal was the target of a valid search path and was unmarked one level up. (Intuitively, this assertion justifies starting the new traversal at the middle of the list.)
- The former assertion allows to establish the loop invariant of the intermediate (`while(cont)`) loop which says that at Line 331 `curr` points to a node which at some point in time during the traversal was the target of a valid search path.
- The role of the loop invariant of the internal (`while(marked)`) loop is key. It says that whenever the loop is about to start (Line 334), not only was the $level$ entry of the `next` array of `curr` the target of a valid search path, but at that time its successor at level `level` was `succ` and that entry at that level was marked only if the `marked` variable is true.

We lift the invariants to concurrent executions by applying the extension of our framework discussed in Remark 2. From this point on, it is easy to prove that in Line 347, `curr` points to a node which at some point in time its bottom `next` entry was unmarked and the target of a search path. As the key

```
324 bool contains(int v)
325   SLN pred←head, curr←null, succ←null;
326   for (int level←L; 0 ≤ level; level−−)
327     {⟐(head ⇝^v pred.next_level ∧ ¬pred.next[level + 1].marked)) ∧ pred.key < v}
328     curr←pred.next[level].ref
329     bool cont←true
330     while (cont)
331       {⟐(head ⇝^v curr.next_level)}
332       (succ, marked)←curr.next[level]
333       while (marked)
334         {⟐(head ⇝^v curr.next_level ∧ curr.next[level] = (succ, marked))}
335         curr←succ
336         {⟐(head ⇝^v curr.next_level)}
337         (succ, marked)←curr.next[level]
338       {⟐(head ⇝^v curr.next_level ∧ curr.next[level] = (succ, marked))}
339       if (curr.key < v)
340         {⟐(head ⇝^v curr.next_level ∧ curr.next[level] = (succ, false)) ∧ curr.key < v}
341         pred←curr
342         curr←succ
343       else
344         {⟐(head ⇝^v curr.next_level ∧ curr.next[level] = (succ, false)) ∧ curr.key ≥ v}
345         cont←false

347     {⟐(head ⇝^v curr.next_0 ∧ ¬curr.next[0].marked)) ∧ curr.key ≥ v}
348     return curr.key = v
```

Figure 10 Lock-free concurrent skiplist algorithm: Procedure `contains`. (See [28, Fig 14.14]).

of the node is unmarked with a key equal or greater to the desired one. Linearizability follows.