Proving Highly-Concurrent Traversals Correct

YOTAM M. Y. FELDMAN, Tel Aviv University, Israel
ARTEM KHYZHA, Tel Aviv University, Israel
CONSTANTIN ENEA, IRIF, Université de Paris, France
ADAM MORRISON, Tel Aviv University, Israel
ALEKSANDAR NANEVSKI, IMDEA Software Institute, Spain
NOAM RINETZKY, Tel Aviv University, Israel
SHARON SHOHAM, Tel Aviv University, Israel

Modern highly-concurrent search data structures, such as search trees, obtain multi-core scalability and performance by having operations traverse the data structure without any synchronization. As a result, however, these algorithms are notoriously difficult to prove linearizable, which requires identifying a point in time in which the traversal's result is correct. The problem is that traversing the data structure as it undergoes modifications leads to complex behaviors, necessitating intricate reasoning about all interleavings of reads by traversals and writes mutating the data structure.

In this paper, we present a general proof technique for proving unsynchronized traversals correct in a significantly simpler manner, compared to typical concurrent reasoning and prior proof techniques. Our framework relies only on *sequential properties* of traversals and on a conceptually simple and widely-applicable condition about the ways an algorithm's writes mutate the data structure. Establishing that a target data structure satisfies our condition requires only simple concurrent reasoning, without considering interactions of writes and reads. This reasoning can be further simplified by using our framework.

To demonstrate our technique, we apply it to prove several interesting and challenging concurrent binary search trees: the logical-ordering AVL tree, the Citrus tree, and the full contention-friendly tree. Both the logical-ordering tree and the full contention-friendly tree are beyond the reach of previous approaches targeted at simplifying linearizability proofs.

CCS Concepts: • **Theory of computation → Program reasoning**; **Concurrent algorithms**;

Additional Key Words and Phrases: concurrent data structures, traversal, traversal correctness, proof framework, linearizability

ACM Reference Format:
Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving Highly-Concurrent Traversals Correct. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 128 (November 2020), 30 pages. https://doi.org/10.1145/3428196

1 INTRODUCTION

A *search data structure* provides a mutable, searchable set or dictionary (e.g., a binary search tree or B+tree). Many important systems, such as databases [Mao et al. 2012; Tu et al. 2013] and operating

Authors' addresses: Yotam M. Y. Feldman, Tel Aviv University, Israel, yotam.feldman@gmail.com; Artem Khyzha, Tel Aviv University, Israel, artkhyzha@mail.tau.ac.il; Constantin Enea, IRIF, Université de Paris, France, cenea@irif.fr; Adam Morrison, Tel Aviv University, Israel, mad@cs.tau.ac.il; Aleksandar Nanevski, IMDEA Software Institute, Spain, aleks.nanevski@imdea.org; Noam Rinetzky, Tel Aviv University, Israel, maon@cs.tau.ac.il; Sharon Shoham, Tel Aviv University, Israel, sharon.shoham@gmail.com.

2475-1421/2020/11-ART128
https://doi.org/10.1145/3428196

systems [Clements et al. 2012], rely on highly-concurrent search data structures for multi-core scalability and performance. In a *highly-concurrent* algorithm, operations synchronize only when modifying the same node. In particular, traversals searching for a key simply navigate the data structure, without performing synchronization, which enables them to run completely in parallel on multiple cores. This design principle is key to search data structure performance [David et al. 2015; Gramoli 2015] and underpins modern concurrent search trees [Arbel and Attiya 2014; Brown et al. 2014; Clements et al. 2012; Crain et al. 2016; Drachsler et al. 2014; Ellen et al. 2010; Howley and Jones 2012; Natarajan and Mittal 2014; Ramachandran and Mittal 2015], skip lists [Crain et al. 2013b; Fraser 2004; Herlihy et al. 2007], and lists/hash tables [Harris 2001; Heller et al. 2005; Michael 2002; Triplett et al. 2011].

Highly-concurrent algorithms are notoriously difficult to prove correct [Feldman et al. 2018; Lev-Ari et al. 2015a; O'Hearn et al. 2010; Vafeiadis 2008]. The standard desired correctness condition is *linearizability* [Herlihy and Wing 1990], which requires that every operation appears to take effect atomically at some point during its execution. It is hard, however, to identify a point in which a highly-concurrent traversal's result holds, because traversing the data structure while it is undergoing modifications can lead to following a path whose links did not exist simultaneously in memory, navigating from a node after it becomes unreachable, and similar complex behaviors.

Accordingly, an emerging research thrust is to design proof techniques that enable using *sequential* reasoning to simplify proving the correctness of highly-concurrent algorithms [Feldman et al. 2018; Lev-Ari et al. 2015a]. The vision is for correctness proofs to follow from a meta-theorem about properties of the algorithm's sequential code, i.e., when running without interference. The user's job then reduces to proving that these sequential properties hold, which does not involve difficult *concurrent* reasoning about interleaved steps of concurrent operations.

While existing work does not yet fully remove concurrent reasoning on the user's part, the amount of concurrent reasoning required is decreasing. Whereas base points [Lev-Ari et al. 2015a] require the user to prove properties of concurrent traversals, local view arguments [Feldman et al. 2018] only rely on sequential properties of traversals, applying under certain conditions, which must be proven with concurrent reasoning. Unfortunately, the local view framework's preconditions are complex and restrictive, and are not satisfied by several data structures such as those by Crain et al. [2016]; Drachsler et al. [2014].

In this paper, we present a proof technique based on a *conceptually simpler and widely-applicable condition*, which enables tackling data structures beyond the reach of previous approaches in addition to simplifying proofs of the data structures supported by them.

Our technique targets proving *traversal correctness*, a proof goal that was implicit in previous works [Feldman et al. 2018; O'Hearn et al. 2010], which is defined to mean that a traversal searching for key k reaching a node n implies that at some point during the traversal's execution so far, n satisfied a *reachability predicate* P_k over the memory state—e.g., that n is on the search path for k. Proving traversal correctness is typically the crux of the data structure's linearizability proof, which then becomes straightforward to complete by the user.

Our key theorem establishes traversal correctness *without having to reason about how a traversal is affected by concurrent writes*, by reasoning solely about how the algorithm's writes modify the memory state. Specifically, writes should satisfy a *forepassed* condition, which (informally) states that they do not reduce the reachability of any memory location unless that location is either not modified later, or modified to point only to locations that have already been reachable. The use of the forepassed condition *alleviates* the need to reason about *how reads in the traversal interleave with interfering writes*; the condition only depends on how different *writes* interleave. Proving that the forepassed condition holds can be done by relying (inductively) on traversal correctness, which

greatly simplifies the proof. This apparent circularity is valid because our theorem is also proven inductively, so both inductions can be combined (§7).

The *only* requirement for applying our technique is that the reachability predicates P_k be compatible with the traversals, which (informally) means that if a traversal searching for k navigates from node n to node n' and $P_k(n)$ holds, then so does $P_k(n')$. Compatibility is a property of the traversal's code on a static memory state, and so can be established with sequential reasoning. In fact, it typically holds trivially, when the predicates are defined based on how traversals navigate the data structure.

Overall, our technique facilitates clear and simple linearizability proofs, applicable to concurrent search data structures with optimistic traversals implementing sets/maps in sequentially-consistent shared-memory.[1] We demonstrate our technique on several sophisticated binary search tress (BSTs): the logical ordering AVL tree [Drachsler et al. 2014] and the full contention-friendly tree [Crain et al. 2016], which cannot be reasoned about with prior proof techniques, and the Citrus tree [Arbel and Attiya 2014], which is a complex algorithm that has not been proven using the prior techniques.

A framework [Shasha and Goodman 1988] simplifying proofs of concurrent data structures has recently been pivotal in several works on proof simplification using concurrent separation logic [Krishna et al. 2020, 2018], but this framework is inapplicable to optimistic traversals (see §9). Our work provides a new proof framework, suitable for highly-concurrent data structures, in which we use sequential reasoning to tackle one of the Gordian knots of this domain. We believe that the theory underlying our result sheds a light on the fundamental reasons for the correctness of these algorithms. We further hope that the theory can be useful for the design of new algorithms, which would explicitly target satisfying the forepassed condition.

Contributions. To summarize, this paper makes the following contributions:

(1) We formally define traversal correctness and present a new general proof argument for the correctness of highly-concurrent traversals.
(2) We provide a simple condition on interfering writes and prove that it establishes traversal correctness.
(3) We apply our framework to prove several interesting and challenging concurrent data structures, including the logical ordering AVL tree [Drachsler et al. 2014], the Citrus tree [Arbel and Attiya 2014], and the full contention-friendly tree [Crain et al. 2016].

2 BACKGROUND: PROOFS OF LINEARIZABILITY

In this section we provide some background on proofs of linearizability [Herlihy and Wing 1990], which is the standard correctness criterion for many highly-concurrent data structures.[2] Throughout this paper we assume a sequentially consistent shared-memory system, i.e., in which the execution is a sequence of interleaved memory operations performed by the threads.

Linearizability requires that every individual method invocation appears to take place instantaneously at some point between its invocation and its return. A classic approach to proving linearizability is to show that the concurrent data structure is *simulated* by a reference implementation where methods execute in a *single* step and according to the expected sequential semantics of the data type. Concretely, this corresponds to defining an *abstraction function* (or simulation relation) that relates states of the concurrent data structure with states of a reference sequential

[1] Our framework is also applicable to algorithms whose traversals use stronger synchronization (e.g. hand-over-hand-locking), but this stronger synchronization satisfies stronger properties allowing simpler proofs with alternative methods; see e.g. [Attiya et al. 2010].

[2] Our technique may also be applicable to proofs of non-linearizable objects [Sergey et al. 2016], a direction which we plan to pursue in future work.

implementation of the data type, such that any step in the concurrent data structure is mapped by the abstraction function to a step of the reference implementation (modulo stuttering).

We focus the presentation on *set* data structures, as are the examples in our paper. Set data structures implement the standard methods `insert`(k) and `delete`(k) for adding or removing an element from the set, respectively, and `contains`(k) which checks membership of an element. For set data structures, the abstraction function $\mathcal{A} : \Sigma \to \mathcal{P}(\mathbb{N})$ maps every concrete memory state $\sigma \in \Sigma$ of the concurrent data structure (a *state* is a mapping from memory locations to values) to an abstract (mathematical) set. For simplicity, we assume that set elements are natural numbers. Establishing linearizability then amounts to showing that in every execution, and for every method invocation in the execution (also called an *operation*), there is a point during its execution interval where the operation "takes effect" on the abstract set. In our context, this means that in every execution π (an execution is a finite sequence of states $\sigma_0, \sigma_1, \ldots$):

- For every invocation of `contains`(k) that returns true, resp., false, there is a state $\sigma \in \pi$ during the invocation such that $k \in \mathcal{A}(\sigma)$, resp., $k \notin \mathcal{A}(\sigma)$. (Note that both such states may exist, in which case `contains`(k) may return either true or false.)
- For every unsuccessful invocation of `insert`(k), resp., `delete`(k), (i.e., an invocation returning false), there is a state $\sigma \in \pi$ during the invocation such that $k \in \mathcal{A}(\sigma)$, resp., $k \notin \mathcal{A}(\sigma)$.
- For every successful invocation of `insert`(k) (returning true), there is an insert-*decisive transition*: a consecutive pair of states $\sigma_i, \sigma_{i+1} \in \pi$ during the invocation such that $\mathcal{A}(\sigma_{i+1}) = \mathcal{A}(\sigma_i) \cup \{k\}$ and $\mathcal{A}(\sigma_i) \neq \mathcal{A}(\sigma_{i+1})$.
- For every successful `delete`(k) (returning true), there is a delete-*decisive transition*: a consecutive pair of states $\sigma_i, \sigma_{i+1} \in \pi$ during the invocation such that $\mathcal{A}(\sigma_{i+1}) = \mathcal{A}(\sigma_i) \setminus \{k\}$ and $\mathcal{A}(\sigma_i) \neq \mathcal{A}(\sigma_{i+1})$.

Further, there is only one point of modification associated with each successful modification: there is a one-to-one mapping between a pair of states $(\sigma_i, \sigma_{i+1}) \in \pi$ where $\mathcal{A}(\sigma_i) \neq \mathcal{A}(\sigma_{i+1})$ to a method invocation for which (σ_i, σ_{i+1}) is a decisive transition (this corresponds to the fact that the methods of the reference implementation execute in a single step).

The above conditions guarantee that all concurrent executions are linearizable. In the algorithms we consider, the decisive transition of a modifying invocation (successful `insert` or `delete`) can be identified statically to correspond to one fixed write in the method's code, constituting a *fixed linearization point*. The case of `contains` and unsuccessful `insert`, `delete` is different. For such invocations, it is impossible to identify a fixed linearization point, i.e., *statically* identify the state where the abstract set contains an element or not as the state reached when the method executes some *fixed* instruction.

3 OVERVIEW

We use the logical-ordering tree [Drachsler et al. 2014] as a running example for the challenge of proving linearizability of a highly-concurrent search data structure and how our framework simplifies such proofs.

3.1 Example: The Logical Ordering Tree

The Logical Ordering (LO) tree is a self-balancing binary search tree (BST), in which keys are stored in both internal and leaf nodes. To maintain a small height, a self-balancing tree modifies its structure in response to insertions and deletions. These modifications are performed by *tree rotations*, which mutate the tree's structure without changing the order of the keys. Figure 1 shows an example, in which node y is rotated right so that the length of the path to the subtree A decreases.

The main challenge for concurrent self-balancing trees is how to avoid having a rotation throw a concurrent traversal "off track." Figure 1 shows an example, in which a traversal headed towards A that is located at y before the rotation would instead reach C, if the rotation happens before the traversal reads y's left child pointer. The LO tree solves this problem using a *logical ordering* technique. In addition to the usual *left*/*right* pointers that induce the binary tree structure, each node also has *pred*/*succ* pointers, which link the node to its predecessor/successor, respectively, according to the logical ordering of the keys. The idea is that a traversal can follow the *pred*/*succ* pointers to find its target node, should some rotation throw it off track.

Conceptually, the data structure consists of a doubly linked list sorted according to logical key order, with a balanced binary tree superimposed on the list's nodes. The binary tree structure is used to speed up the set `insert`/`delete`/`contains` operations, but it is ultimately a key's presence or absence in the list that determines an operation's result. Figure 2 presents the code of the algorithm. (The code is annotated with assertions in curly braces, which should be ignored at this point.) For brevity, we omit the rebalancing logic, which decides when to perform rotations, as well as other auxiliary code functions.

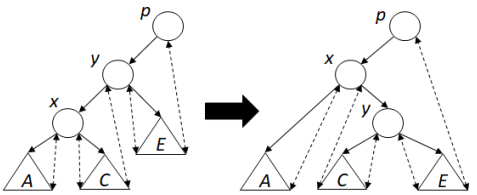


Fig. 1. Logical ordering tree. Following a right rotation of y, a traversal about to navigate from y towards A (before) would normally reach C instead (after). However, it can follow *pred* pointers (dashed) to reach A from C.

The algorithm uses fine-grained locking. Each node has two locks, `treeLock` and `succLock`, for protecting tree and list manipulations, respectively. Every operation begins by traversing the tree, without acquiring any locks, until reaching a leaf. A `contains` operation then traverses the linked list, searching for the target key, and returns whether it was found. First it follows *pred* pointers until it finds a node with a key not greater than the target, and if the target key was not yet found it continues to follow *succ* pointers (see Figure 3). (The distinction between *pred* and *succ* pointers is important because they are not modified together atomically. As we shall see, the decisive view is of the list defined by *succ* pointers.) The `insert` and `delete` operations acquire locks, verify that they have landed at the correct location in the list, and then perform their respective operation on both the list and the tree. The *update* operations, `insert` and `delete`, are *successful* if they insert/delete the key from the data structure, and *failed* otherwise.

Deletions are performed in two steps. The node is first *logically* deleted, removing its key from the set represented by the tree, by setting its boolean `rem` field. The node is then *physically* removed from the tree and list. Physical removal of a leaf or a node with a single child simply splices the node from the tree. Physical removal of a node with two children is more subtle (see Figure 4). It is performed by replacing the deleted node with its successor, which is obtained from the list. Similarly to a rotation, moving the successor up the tree in this way[3] can cause concurrent tree traversals searching for the successor to miss it (in Figure 4, `s` is unreachable in the tree after modification (a) and before (c)). For this reason, other concurrent BSTs [Arbel and Attiya 2014; Bronson et al. 2010; Crain et al. 2016] implement internal node deletion differently. The LO tree solves the problem by relying on the underlying list structure for correctness. In insertion, `chooseParent` returns the location in the tree to which the new node should be linked. We omit its code, because (as we shall see in the proof below), the correctness of tree operations stems from the list structure. A node's tree location only determines the efficiency of a traversal locating the node.

[3]In the tree, the successor is the leftmost leaf of the node's right subtree.

```
1  type N
2    int key
3    bool rem
4    N left, right, parent
5    Lock treeLock
6    N succ, pred
7    Lock succLock
8
9  N min←new N(−∞);
10 N max←new N(∞);
11 min.succ←max
12 max.pred←min
13 root←max
14
15 private N tree-locate(int k)
16   x,y←root
17   while (y≠null ∧ y.key≠k)
18     x←y
19     if (x.key<k)
20       y←x.right
21     else
22       y←x.left
23   {⇳ ({−∞} ⇝ x)
       ∧ ⇳ ({−∞} ⇝ y)}
24   return (y=null? x : y)
25
26 bool contains(int k)
27   x←tree-locate(k)
28   while x.key > k
29     {⇳ ({−∞} ⇝ x)}
30     x←x.pred
31   while x.key < k
32     {⇳ ({−∞} ⇝^k x)}
33     x←x.succ
34   if (x.key ≠ k)
35     {⇳ ({−∞} ⇝^k x) ∧ x.key > k}
36     return false
37   {⇳ ({−∞} ⇝^k x) ∧ x.key = k}
38   if (x.rem)
39     {⇳ ({−∞} ⇝^k x ∧ x.rem)
         ∧ x.key = k}
40     return false
41   {⇳ ({−∞} ⇝^k x ∧ ¬x.rem)
       ∧ x.key = k}
42   return true
43 bool delete(int k)
44   x←tree-locate(k)
45   p←(x.key>k ? x.pred : x)
46   lock(p.succLock)
47   s←p.succ
48   if k∉(p.key,s.key] ∨ p.rem
49     restart
50   {{−∞} ⇝^k p ∧ k ∈ (p.key, s.key]
       ∧ ¬p.rem ∧ p.succ = s }
51   if s.key≠k
52     {{−∞} ⇝^k s ∧ s.key > k}
53     return false
54   lock(s.succLock)
55   {{−∞} ⇝^k s ∧ s.key = k ∧ ¬s.rem}
56   s.rem←true
57   removeFromTree(s)
58   y←s.succ
59   y.pred←p
60   {{−∞} ⇝^k p ∧ p.succ = s ∧ s.succ = y
       ∧ s.key = k ∧ p.key < k < y.key
       ∧ ¬p.rem ∧ s.rem}
61   p.succ←y
62   return true
63
64 bool insert(int k)
65   x←tree-locate(k)
66   p←(x.key>k ? x.pred : x)
67   lock(p.succLock)
68   s←p.succ
69   if k∉(p.key,s.key] ∨ p.rem
70     restart
71   {{−∞} ⇝^k p ∧ k ∈ (p.key, s.key]
       ∧ ¬p.rem ∧ p.succ = s }
72   if s.key=k
73     {{−∞} ⇝^k s ∧ s.key = k ∧ ¬s.rem}
74     return false
75   n←new N(k)
76   n.succ←s
77   n.pred←p
78   z←chooseParent(p,s,n)
79   n.parent←z
80   lock(z.treeLock)
81   if (z.key<k)
82     z.left←n
83   else
84     z.right←n
85   s.pred←n
86   {{−∞} ⇝^k p ∧ ¬p.rem
       ∧ p.succ = s ∧ k ∈ (p.key, s.key)
       ∧ n.key = k ∧ ¬n.rem ∧ n.succ = s}
87   p.succ←n
88   return true
89 private removeFromTree(n)
90   lock(n.treeLock)
91   if (n.left = null)
92     updateChild(n.parent,
93                 n,
94                 n.right)
95     return
96   if (n.right = null)
97     updateChild(n.parent,
98                 n,
99                 n.left)
100    return
101  s←n.succ
102  lock(s.treeLock)
103  // temporarily unlink s
104  updateChild(s.parent,
105              s,
106              s.right)
107  // s takes n's location
108  s.left←n.left
109  s.right←n.right
110  n.left.parent←s
111  if (n.right ≠ null)
112    n.right.parent←s
113  updateChild(n.parent,n,s)
114
115 private updateChild(p,n,c)
116  // pre: n locked
117  // pre: n.parent = p
118  // pre: c = n's only child
119  lock(p.treeLock)
120  if (p.left = n)
121    p.left = c
122  else
123    p.right = c
124  if (c ≠ null)
125    c.parent←p
126
127 rotateRightLeft()
128  p←tree-locate(∗)
129  lock(p.treeLock)
130  y←p.left
131  if (y=null)
132    return
133  lock(y.treeLock)
134  x←y.left
135  if(x=null)
136    return
137  lock(x.treeLock)
138  p.left←x
139  p.parent←y
140  p.left←x
141  y.left←x.right
142  x.right←y
```

Fig. 2. Logical-ordering tree [Drachsler et al. 2014]. For brevity, **unlock** operations are omitted; a procedure releases all the locks it acquired when it terminates or **restart**s. ∗ denotes an arbitrary key.

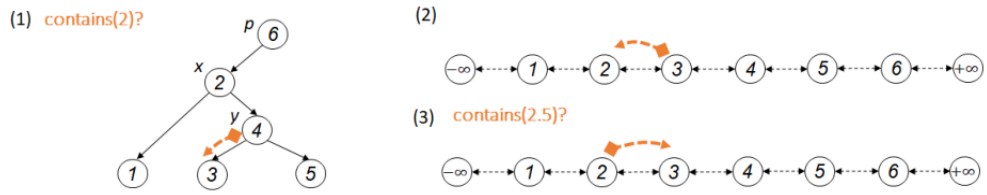


Fig. 3. Traversals in the LO tree. A traversal looking for key 2 has reached node y due to a concurrent rotation in the tree (see Figure 1). In (1), it continues to perform a binary search in the tree and does not find 2 (lines 17 to 22 in Figure 2). Nevertheless, in (2), the traversal continues by reading *pred* pointers and finds the key, allowing it to return `contains(2)` = *true* (line 29). Another traversal, this time looking for key 2.5, encountering the same scenario in (1)–(2) performs an extra step of reading *succ* pointers until it reaches a node with a larger key, allowing it to return `contains(2.5)` = *false* (line 32).

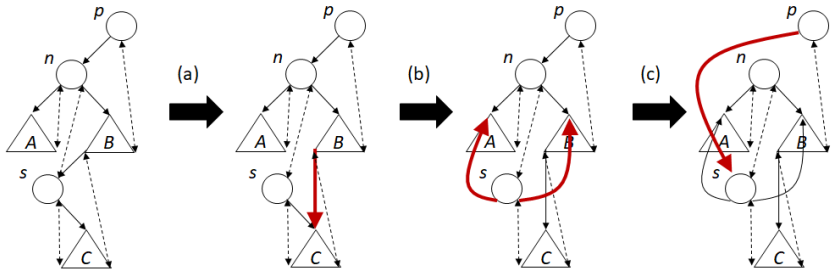


Fig. 4. Removing a node n with with two children from the tree structure of the LO tree (`removeFromTree` in Figure 2). s is the successor of n, found using the list layout. (a) s is temporarily removed from the tree structure (line 104). (b) the children of n are copied to s (lines 108 to 109). (c) n's parent is modified to point to s instead; thus s takes the location of n in the tree (line 113). Note that *pred*, *succ* are not modified yet, and are thus inconsistent with the tree layout (the list layout is updated afterwards). The updates to the *parent* field are not presented.

Overall, successful update operations mutate both the tree and list structure, which involves writing to multiple memory locations. Concurrent traversals can observe a subset of these writes, effectively observing the data structure in mid-update. Reasoning about all possible interleavings of writes and traversal reads is incredibly hard. Indeed, we find that the LO tree's original linearizability proof [Drachsler et al. 2014] is flawed.[4] The proof claims that an `insert`(k) takes effect when k's node gets pointed to by its predecessor's *succ* pointer. However, a `contains` searching for k could find it before this update by following *pred* pointers. Consequently, the code we present in Figure 2 uses a different order of pointer updates than the original code [Drachsler et al. 2014].

3.2 Linearizability of the Logical Ordering Tree

In this section, we give an overview of a standard linearizability proof for the LO tree. We show that the proof boils down to reasoning about traversal correctness, and explain why proving traversal correctness is non-trivial.

The proof uses an abstraction function $\mathcal{A}$ mapping each concrete memory state to the (abstract, mathematical) set it represents, and showing that every operation "takes effect" at some point during its execution (see §2.) To define the abstraction function $\mathcal{A}$, we conceptually break the LO

[4]This has been confirmed with Drachsler et al.

tree's doubly linked list into the *successor* and *predecessor* lists, induced by following *succ* pointers from the sentinel node `min` (denoted $\{-\infty\}$) and by following *pred* pointers from the sentinel node `max` (denoted $\{+\infty\}$), respectively. We define $\mathcal{A}$ based on reachability in the successors list. $\mathcal{A}$ maps a state σ to the set of the keys of the nodes on the successor list that are not logically deleted. Technically, for every key k, we define a state-predicate over memory locations ℓ, $\{-\infty\} \overset{k}{\rightsquigarrow} \ell$. This predicate holds in state σ if, when following the *succ* pointers from $\{-\infty\}$ in σ and stopping when k or a greater key is found, we encounter ℓ. We then say that ℓ is *k-reachable*. The abstraction function is then defined as follows:

$$\mathcal{A}(\sigma) = \{k \in \mathbb{N} \mid \sigma \models \exists x.\, \{-\infty\} \overset{k}{\rightsquigarrow} x \wedge x.key = k \wedge \neg x.rem\},$$
$$\text{where } \sigma \models P \text{ means that } P \text{ is true in } \sigma.$$

The challenge now is to identify when an operation's execution takes effect on the abstract state by establishing properties of the data structure and the algorithm. These are displayed as *assertions* annotating the code in Figure 2. The assertion notation should be interpreted as follows. An assertion $\{P\}$ means that in every execution, P holds in any state in which the next line of code executes, which can be thought of as "now" (with respect to the executing operation's perspective). An assertion $\{\diamondsuit\,(P)\}$ says that P was true at some point between the invocation of the operation and "now." Note that assertions use both $\{-\infty\} \overset{k}{\rightsquigarrow} \ell$ as well as another reachability predicate, $\{-\infty\} \rightsquigarrow \ell$, which holds in a state whenever ℓ is reachable from $\{-\infty\}$ by following *succ* pointers (irrespective of any key).[5]

Assuming that these assertions hold, it is a rudimentary exercise to show that any operation takes effect on the abstract set (see §2) during some point of its execution; see the extended version [Feldman et al. 2020] for an elaboration. The important point to notice is that reachability assertions are inherent to showing the necessary $k \in \mathcal{A}(\sigma)$ or $k \notin \mathcal{A}(\sigma)$ during the operation's execution. For example, that a node with key k was previously reachable when `contains` returns true (line 41) is crucial for showing that at some point σ during the operation's execution indeed $k \in \mathcal{A}(\sigma)$.

Proving the assertions. The proof's main goal is thus to prove the assertions in Figure 2, which would conclude the linearizablity proof. Which assertions are hard to prove? The assertions that do not concern traversals and $\{\diamondsuit\,(P)\}$ properties are straightforward, and rely on simple inductive invariants about the data structure, properties such as the immutability of keys, and reasoning about interleavings of the lock-protected critical sections. (See the extended version [Feldman et al. 2020] for an elaboration.)

The crux of the linearizability proof, therefore, is to prove the assertions that concern the unsynchronized traversals (lines 23, 29, 32, 35, 37, 39 and 41). These assertions state that nodes were reachable *at some point in time* during an operation's execution. We formalize the problem of proving such properties as that of establishing *traversal correctness*.

3.3 The Need for Traversal Correctness Assertions

The focus of our work is $\{\diamondsuit\,(P)\}$ properties, for predicates P that have to do with reachability. We use such properties to capture traversal correctness. An example is the assertion $\diamondsuit\,(\{-\infty\} \overset{k}{\rightsquigarrow} x)$ in line 32, which states that when the traversal on successor nodes in lines 31–33 executes, every node the traversal encounters has been k-reachable at some point between the method's invocation and now.

[5] For brevity, we omit from the assertions facts about locks, which in this algorithm can always be inferred from the scope.

Traversal correctness guarantees the reachability of encountered nodes at *some* time in the past. (We formally define traversals and traversal correctness in §4.) Such a property seems less intuitive than the more natural claim that these nodes are reachable *when they are encountered*. That easier claim would indeed have been nice to have, only that it does not hold, and $\{-\infty\} \overset{k}{\rightsquigarrow} x$ would be an incorrect assertion in line 32—a traversal can arrive at a node *after it is no longer reachable*. For example, a traversal can arrive at node x', but before it continues to $x = x'.succ$, x' and then x are removed. Thus, when the traversal reads $x'.succ$ and arrives at x, the node x is no longer reachable.

Consequently, the crux of the linearizability argument follows from traversal correctness properties. Unfortunately, traversal correctness is fundamentally harder than proving the other assertions in the code, as explained next.

3.4 The Challenge of Proving Traversal Correctness

Proving traversal correctness requires showing that a node was reachable at *some point in time* during the traversal's execution. While it is immediate that if x is reachable now and x points to y now, then y is reachable now, analogous reasoning is not straightforward for reachability "in the past." Consider, for example, a list traversal that reads a pointer $x.succ$ which points to node y. Knowing that $\diamondsuit(\{-\infty\} \rightsquigarrow x)$ does not directly imply the reachability of y—neither now nor in the past—because $\{-\infty\} \rightsquigarrow x$ might *not* hold when $x.succ$ is read. Imagine what could happen in between: For example, as explained above, x could be removed, and subsequently y could be removed, and still $x.succ = y$, even though y is no longer reachable. Although there is a link $x.succ = y$ and it is easy to see (from the assertions) that y was reachable when this link was written, this is not evidence enough for $\diamondsuit(\{-\infty\} \rightsquigarrow y)$, because this write could have taken place before the beginning of the traversal (and before the invocation of the current method). Another tricky scenario to consider is that although y can be removed, its key could be inserted elsewhere in a new node z, and the traversal may not be aware of this and miss z. It is therefore possible for traversals to reach a key that has already been removed, "miss" insertions, etc. Furthermore, the path that the traversal follows is built of pointers from many points in time, and may never have existed in memory in its entirety. Surprisingly, it turns out that deciding existence/absence of a key based on such a traversal is correct. That is, for a list traversal that starts at $\{-\infty\}$, it can indeed be established that $\diamondsuit(\{-\infty\} \rightsquigarrow y)$ holds in this example, but this involves intricate reasoning about how interfering writes interleave with the traversal's reads. Simplifying this has been the goal of previous works [Feldman et al. 2018; O'Hearn et al. 2010].

The traversal in the LO tree is even more complex. While it is the reachability in the doubly linked list that determines the result of a traversal, its first part, `tree-locate`, traverses the nodes over the binary tree links. The traversal over the list *does not start from the head of the list*, but from where the tree traversal landed, so it is necessary to prove properties of the tree-traversal as well. Reasoning about the tree traversal is even more challenging than the traversal over the list, for two reasons:

(1) Having to reason about lists while traversing a tree makes "off-the-shelf" approaches not suitable for proving the reachability assertions during traversals (see §9). Intuitively, `tree-locate` accesses nodes in the list in almost a random-access way, so it is unclear why it cannot go wrong.
(2) Moreover, examining the tree traversal is very challenging due to complex interference patterns: in between this traversal's reads, in-place tree rotations can take place, nodes can be unlinked and linked back, etc. Previous approaches fall short, and cannot be applied to liberate the user from considering the interleavings of writes in between the traversal's reads (see §9).

3.5 Our Framework: Proving Traversal Correctness

Our proof framework provides a *general* method for proving traversal correctness while reasoning only about the effect writes have on memory, *without* resorting to complex concurrent reasoning about how reads in the traversal interleave with interfering writes (in the like of the corner cases above). In the remainder of this section we illustrate the ideas behind our framework by (informally) proving one of the traversal correctness assertions, $\Diamond (\{-\infty\} \overset{k}{\rightsquigarrow} x)$ in line 32 for the segment of the traversal over successor links (we prove all the remaining traversal correctness assertions in §6, after formally presenting the framework).

Consider the traversal in lines 31 to 33. For us, a traversal is simply a sequence of read operations of locations $\ell_0, \ell_1, \ldots$ (§4), and here this sequence traverses successor pointers until it reaches a node with a key greater or equal than k.

Suppose that we have already established the assertion $\Diamond (\{-\infty\} \overset{k}{\rightsquigarrow} x)$ before this traversal, that is, at the entry to the loop in line 31. Our goal is to prove that, in spite of possible interference, $\Diamond (\{-\infty\} \overset{k}{\rightsquigarrow} x)$ holds also for the new x's that the traversal reaches by following successor links. Our framework achieves such a proof by showing that two properties hold. The first, *single-step compatibility*, connects the reachability predicate $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ to the way the traversal navigates from one node to the next. The second, *forepassed*, constrains the effect over time of interfering writes on the reachability predicate.

(1) Single-step compatibility. The traversal chooses the next location to read based on the last read location and the key; for example, the traversal reads $\ell_i = o.key$ and decides to proceed by reading $\ell_{i+1} = o.left$ according to the binary search. This is called a *step* of the traversal. The next step in this case is to read $\ell_{i+2} = o'.key$ where o' is the object to which $o.left$ points to (dereferencing the pointer), from which the traversal proceeds by taking steps in a similar fashion.

The requirement of *single-step compatibility* (Definition 5.1) focuses on a single step of the traversal at each time. Consider the traversal advancing from ℓ_i to ℓ_{i+1} by reading the single value at the location ℓ_i from the current memory state σ_t. Then single-step compatibility requires that if $\sigma_t \models \{-\infty\} \overset{k}{\rightsquigarrow} \ell_i$, then also $\sigma_t \models \{-\infty\} \overset{k}{\rightsquigarrow} \ell_{i+1}$ holds. Single-step compatibility obviously holds for the traversal over successor links and this reachability predicate (see Example 5.2). Note that both $\{-\infty\} \overset{k}{\rightsquigarrow} \ell_i$ and $\{-\infty\} \overset{k}{\rightsquigarrow} \ell_{i+1}$ here are evaluated in the same memory state σ_t (without interference). Importantly, this means that establishing this condition relies on purely *sequential* reasoning: the scope of this condition is a single read operation, and interference is irrelevant.

(2) Forepassed interference. This condition tracks writes that *reduce k-reachability*: the k-reachability of a location ℓ is *reduced* by the write w if before w it holds that $\{-\infty\} \overset{k}{\rightsquigarrow} \ell$ but not after w. For example, removing a node (by modifying the pointer of its parent) reduces its reachability. Intuitively, such an interfering write is "dangerous" because a traversal can reach ℓ and be unaware that ℓ is no longer k-reachable. The *forepassed* condition (Definition 5.3) requires that a location ℓ whose reachability is reduced by w at time t either

- cannot be later modified (we call this *strong forepassed*), or,
- otherwise, if ℓ is modified by a later write w', then w' writes a value that points to a location ℓ' that is known to have been k-reachable at some intermediate moment, between the time immediately before w was performed and the time immediately after w' was performed.

In the traversal over successor links in the LO tree, whenever the reachability of a node is reduced it is first *marked*, inhibiting later writes to this location (see Example 5.5), which guarantees the forepassed condition.

The insight behind our approach and the forepassed condition is that accessing a memory location ℓ whose reachability has been reduced is not itself problematic; ℓ still *has been* k-reachable. The challenge is to prove that locations reached through ℓ have also been k-reachable. The main idea of the forepassed condition is that to achieve that, we must limit the ways ℓ can be modified after its reachability is reduced. Consider the next location the traversal visits, ℓ', which is pointed to by ℓ. In any point in time where ℓ was k-reachable and contained the same value, ℓ' was also k-reachable. Thus, if ℓ was not later modified prior to reading its value and visiting ℓ' (this is the case of strong forepassed), then ℓ' has also been k-reachable. The forepassed condition still allows some writes to ℓ after its reachability was reduced, as long as they retain the property that the next location in the traversal ℓ'—which is dictated by the values these writes put in ℓ—also has been k-reachable. Either way, the forepassed condition guarantees that the traversal continues to locations that have been k-reachable, and the traversal is not "led astray".

In almost all our examples, the interference satisfies the *strong forepassed* condition, which is simpler to reason about, but the more general condition is required e.g. for an implementation of backtracking (see §8.2).

Deducing traversal correctness. The core of our framework is the theorem that if the traversal is single-step compatible and writes satisfy forepassed interference, both w.r.t. the reachability predicate, then the traversal is correct—every location it reaches has been reachable at some point (Theorem 5.7). Thus, in the traversal over successor links of the LO tree, we deduce that every x the traversal reaches satisfies $\Diamond (\{-\infty\} \overset{k}{\rightsquigarrow} x)$, finally proving our much sought traversal correctness assertion.

Remark 3.1. *Traversal correctness, single-step compatibility, and the forepassed condition all depend on the choice of reachability predicate. In the example we have considered here, the reachability predicate was* $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$*, and it arose directly from the definition of the abstraction function, which is usually the case. Occasionally, choosing the right reachability predicate demands more care. In the LO tree, the traversal over the tree and the predecessor links (lines 16–24,28–30) is correct w.r.t.* $\{-\infty\} \rightsquigarrow \cdot$ *irrespective of the key (which makes sense because it performs "random accesses" to the successors list). This is discussed in §6. A more intricate scenario is the Citrus tree [Arbel and Attiya 2014], discussed in §8.3. There, traversal correctness requires a weakening of the reachability predicate that captures (using ghost state) the potential of paths to go off track in certain ways.*

Scope and limitations. We are not aware of algorithms in our domain where our proof argument is inherently inapplicable, although such examples are possible. The Citrus tree (§8.3) comes close, in the sense that our framework does not apply with a straightforward choice of the reachability predicate (in fact, the traversal is not correct in the sense of Definition 4.3), but we are nonetheless able to apply our technique using a modified reachability predicate and ghost code.

In this paper we assume sequential consistency (SC), following most concurrent data structures and many papers on their verification. In many relaxed memory models, data-race free programs have SC semantics, making our techniques applicable (in C/C++, for example, the shared node fields would be accessed with SC atomics). Proving the correctness of algorithms that exploit weaker consistency models is an interesting future direction.

Outline. We formally define traversals and traversal correctness in §4, and develop the framework's conditions and main theorem in §5. In §6, we apply it to prove all the traversal correctness assertions of the LO proof, thereby completing the linearizability proof here. In §7, we discuss how traversal correctness assertions can rely on the very same assertions that they help prove, in an inductive

manner, making such proofs simple. We apply the framework to additional challenging examples in §8. Related work is discussed in §9, and §10 concludes.

4 TRAVERSALS AND THEIR CORRECTNESS CRITERION

Our framework targets traversals. In this section we describe our model of traversals as a sequence of reads determined by a sequence of local steps (§4.2), a view which is essential to phrase our single-step compatibility condition below (in §5.1). We then formally define traversal correctness w.r.t. a reachability predicate (§4.3) which is the central proof goal of our framework.[6] As a running example, we use the same traversal whose correctness we described informally in §3.5: the traversal in lines 31 to 33 of `contains` of Figure 2 and its correctness with respect to the reachability predicate $\{-\infty\} \overset{k}{\rightsquigarrow} x$. We begin with some preliminary definitions.

4.1 States, Locations, Executions, and Writes

A *state*, denoted σ, is a mapping from memory locations to values. We use an indexing of memory locations by pairs, $(o, \mathbf{f})$, where o is an object identifier, and $\mathbf{f}$ is a field name. The value in a location can be another location (a "pointer"—e.g. to (o, key)). A *write* is a pair (ℓ, v) of a location ℓ and the value v being written to it. We use discrete timestamps $\ldots, t-1, t, t+1, \ldots$ to model the order in which writes occur. An *execution* is a sequence of (atomic) writes (performed by different threads) at increasing timestamps performed by the algorithm—this corresponds to recording just the write operations in a run of the algorithm. Given an execution, we denote by σ_t the state of the algorithm at time t, and by $\sigma_t(\ell)$ the value in location ℓ in σ_t. States are modified by writes: if a write (ℓ, v) occurs at time t, then $\sigma_t(\ell') = \sigma_{t-1}(\ell')$ for every $\ell' \neq \ell$, and $\sigma_t(\ell) = v$ (that is, σ_t is the state after the write is performed). When we consider an execution in the *timespan* $[t_\star, t^\star]$, then the first state in the execution is $\sigma_{t_\star}$, and the execution consists of writes with timestamps in $(t_\star, t^\star]$. (The first timestamp may be an arbitrary point in the algorithm's run, not necessarily the beginning of any operation.) A *read* is performed to a memory location ℓ from a memory state σ_t, observing the value $\sigma_t(\ell)$. For our purposes, we shall not need to model the exact time when a read occurs, only the state from which it reads the value (when ordering the reads and writes together the read occurs after write t and before $t+1$).

Remark 4.1. *In our formalization, reads and writes occur in specific memory states, to/from specific memory locations. There is a subtle gap between this and the* code *of the algorithm (such as the code in Figure 2); it is necessary to translate the read and write program instructions to the actual read and write operations performed when the code executes, which is the level of abstraction our formalization uses. Bridging such a gap in a formal proof is usually the role of a program logic; in this paper, when we apply the framework to prove algorithms presented in code, this connection is straightforward.*

4.2 Traversals

To formally define a traversal, we use a relation $\text{extend}_p(\ell, v, \ell')$, which encodes that after reading location ℓ and seeing value v, the location ℓ' is the next to be read. This relation captures the sequence of reads a traversal (e.g. lines 31 to 33 of `contains` in the Figure 2) performs. The specialization by p denotes the dependence of the traversal's logic on certain parameters, such as a key in our running example. For a given p, the relation extend_p only depends on ℓ, v, ℓ'. The intuition is that the next location ℓ' to be read is chosen based on the last location read ℓ and the value read v (but nothing else). The definition of the relation reflects the code that implements the traversal.

[6] A similar notion was implicit in [Feldman et al. 2018].

Given an execution, a *traversal* is a sequence $(\ell_1, t_1), \ldots, (\ell_n, t_n), \ell_{n+1}$, where ℓ_i's are locations, t_i's are timestamps, and every consecutive pair satisfies extend_p, namely, $\forall i = 1, \ldots, n.\ \text{extend}_p(\ell_i, v_i, \ell_{i+1})$ where $v_i = \sigma_{t_i}(\ell_i)$. The pair (ℓ_i, t_i) indicates that the traversal reads ℓ_i from the state σ_{t_i} (observing the value $\sigma_{t_i}(\ell_i)$). When the traversal performs the read of location ℓ_i, we say that the traversal *reaches* ℓ_{i+1}. Note that at this point, ℓ_{i+1} itself is not (yet) read—as an illustration, ℓ_{i+1} may be reached by "following the pointer" in ℓ_i, which amounts to reading ℓ_i.

Example 4.1. The `contains` operation of the LO tree visits a sequence of nodes that can be split into two sequences: (1) from the beginning of `contains` until line 30, the operation visits nodes by following `left`, `right` and `pred` links; (2) afterwards, at lines 31 to 33, the operation visits nodes by following `succ` links. We use the latter in our illustrations of the framework throughout §4 and §5, and give full details for the LO tree in §6.

Consider the traversals over successor links that a `contains` operation performs searching for a key k at lines 31 to 33 in Figure 2. To analyze them, we instantiate extend_p with a relation extend_k, which is parametrized by the key k and is true only for the following cases (for every objects o and o', and value m):

$$\text{extend}_k((o, \mathit{key}), m, (o, \mathit{succ})) \qquad \text{if } m < k$$
$$\text{extend}_k((o, \mathit{succ}), (o', \mathit{key}), (o', \mathit{key}))$$

Informally, this definition states that (i) from a *key* field of an object, the traversal is extended to the *succ* field of the same object, in case the value in the *key* field is less than k, and (ii) from a *succ* field the traversal is extended to the location holding the *key* field of the object to which *succ* points (following the pointer, in short). (In the assertions in Figure 2, when we write $\{-\infty\} \overset{k}{\rightsquigarrow} x$, reachability to the object x is a shorthand to reachability to (x, key).) It is immediate that extend_k correctly summarizes the code performing the traversal in lines 31 to 33 of `contains`, namely, in the sequence of reads performed by this operation executing this code section, every consecutive pair of locations satisfies extend_k with the corresponding value read (see also Remark 4.1).

4.3 Traversal Correctness

Roughly, traversal correctness requires that every location reached by the traversal has been "reachable" within a certain preceding timespan. The notion of reachability and the timespan are formalized next.

Reachability. Reachability is defined by a reachability predicate, chosen to be useful in the overall correctness argument, such as $\{-\infty\} \overset{k}{\rightsquigarrow} x$ in §3.2 (see also Remark 3.1). Formally, a *reachability predicate* is a unary state-predicate over locations that can be parameterized, e.g. by a (fixed) root and by a key of interest. We denote this predicate $\mathit{reach}(\cdot)$. For a location ℓ, we use $\models_t \mathit{reach}(\ell)$ to denote that the predicate $\mathit{reach}(\ell)$ holds in the state σ_t (the state at time t). We also say that ℓ satisfies the reachability predicate at time t.

Example 4.2. In the running example, the reachability predicate of interest for the traversal in lines 31 to 33 searching for some key k is $\{-\infty\} \overset{k}{\rightsquigarrow} x$, called k-reachability. The k-reachability predicate is defined by the existence of a sequence of locations that follows extend_k: $\{-\infty\} \overset{k}{\rightsquigarrow} x$ holds in state σ if there is a sequence of locations $\ell_0, \ldots, \ell_n$ s.t. $\ell_0 = (\{-\infty\}, \mathit{key})$, $\ell_n = x$, and $\forall i < n.\ \text{extend}_k(\ell_i, \sigma(\ell_i), \ell_{i+1})$.

Our framework can accommodate versatile reachability predicates, including reachability and k-reachability in a list (§6), k-reachability by binary search in a tree (§8.2), and even sophisticated reachability using ghost state (§8.3).

Base time. In addition to the reachability predicate, traversal correctness is also relative to a *base* time $t_\star \le t_1$ within the timespan of the current operation (that is, the state $\sigma_{t_\star}$ was present concurrently with the operation), such that the first location ℓ_1 is reachable at base time $t_\star$. The base time, similarly to the reachability predicate, is chosen as part of the proof. Most often, $t_\star = t_1$ and ℓ_1 is the root that is always reachable, but a base $t_\star \le t_1$ is needed e.g. in the running example (see §6). Below, for convenience, we consider executions whose timespan begins at the base time $t_\star$.

Traversal correctness. Given the reachability predicate and the base, *traversal correctness* requires that every location ℓ_{i+1} reached by the traversal at time t_i as defined in §4.2 has satisfied the reachability predicate at some point (state) in the execution between $t_\star$ and t_i (inclusive). Let $\diamondsuit_t^{t'}(reach(\ell))$ be a shorthand for $\exists t''.\ t \le t'' \le t' \wedge \models_{t''} reach(\ell)$. Then we can define traversal correctness as follows.

Definition 4.3. A traversal $(\ell_1, t_1), \ldots, (\ell_n, t_n), \ell_{n+1}$ is *correct* w.r.t. $reach(\cdot)$ and base $t_\star$ if

$$\overset{t_i}{\underset{t_\star}{\diamondsuit}}(reach(\ell_{i+1}))$$

holds for every $i = 0, \ldots, n$, with $t_0 = t_\star$.

Note that traversal correctness requires that ℓ_1 is reachable at the base time $t_\star$.

In our applications of the framework to prove different algorithms we always use timestamps that are concurrent with the current operation, and so use $\diamondsuit$ without time bounds to indicate that $t_\star$ is the beginning of the operation and $t^\star$ is the current time. Most often we are interested in the reachability of the last location reached in the traversal, and deduce the assertion $\diamondsuit\,(reach(\ell_{i+1}))$ in the code performing the traversal.

The goal of our framework is to prove traversal correctness using simple concurrent reasoning, as we describe next.

5 THE FRAMEWORK: PROVING THE CORRECTNESS OF TRAVERSALS

In this section we describe how our framework proves traversal correctness. We first explain how the reachability predicate needs to be tied to the traversal itself via single-step compatibility (§5.1), and explain the forepassed condition about writes (§5.2) that guarantees traversal correctness in spite of interference (§5.3). Lastly, we extend the framework to deduce reachability together with a property of a single field (§5.4).

Throughout this section, we fix an execution of the algorithm in timespan $[t_\star, t^\star]$.

5.1 Single-Step Compatibility

Our framework is applicable to prove traversal correctness w.r.t. reachability predicates that are compatible with the extend_p relation underlying the traversal in the following way:

Definition 5.1. We say that a reachability predicate $reach(\cdot)$ is *single-step compatible* with an extend_p relation if for every state σ in the execution and every pair of locations ℓ, ℓ' it holds that $\sigma \models reach(\ell) \wedge \text{extend}_p(\ell, \sigma(\ell), \ell') \implies \sigma \models reach(\ell')$.

Example 5.2. In the LO tree running example, single-step compatibility of $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ (Example 4.2) with extend_k (Example 4.1) holds by construction—and this is usually the case—since this reachability predicate is formally defined by the existence of a sequence of locations that follows extend_k (see Example 4.2). At times, reachability is defined not through sequences of locations following extend_p, in which case the compatibility relies on a different argument. This scenario arises when we analyze the traversal in lines 16 to 24 and lines 28 to 30 (see §6).

5.2 The Condition on Interference: Forepassed

We now describe our main condition about how the writes in the execution, potentially interfering with traversals, affect the reachability predicate. In essence, the idea is that if a write w reduces the reachability of a memory location ℓ, then afterwards writes to ℓ are not allowed, unless they modify ℓ in a very specific way: by pointing only to locations that have already been reachable at some point in between these writes.

Definition 5.3 (Forepassed). A write w at time $t > t_\star$ in the execution satisfies the *forepassed* condition if for every location ℓ, either

(1) $\models_{t-1} \mathit{reach}(\ell) \implies \models_t \mathit{reach}(\ell)$ (that is, ℓ's reachability is not reduced by w); or
(2) for every write w' to ℓ in time $t' \in [t, t^\star]$, if w' writes a value v to ℓ and $\mathrm{extend}_p(\ell, v, \ell')$ for some ℓ', then $\Diamond^{t'}_{t-1}(\mathit{reach}(\ell'))$ holds (that is, every subsequent write to ℓ, including the current write w (if it writes to ℓ), points to a location that has been reachable at some point from just before w to just after w'.)

In the algorithms we consider, most interfering writes satisfy a stronger, and simpler, condition: that if w reduces the reachability of ℓ, then ℓ is not modified by any write, including w:

Definition 5.4 (Strong Forepassed). A write w at time t in the execution satisfies *strong forepassed* if for every location ℓ, either

(1) $\models_{t-1} \mathit{reach}(\ell) \implies \models_t \mathit{reach}(\ell)$ (that is, ℓ's reachability is not reduced by w, as in Definition 5.3); or
(3) no write in time $[t, t^\star]$ modifies ℓ (that is, ℓ is "immutable" from this time on; this includes the current write w, so w cannot write to ℓ itself).

Note that condition (3) is a special case of (2) in the presence of single-step compatibility, but it is often conceptually simpler, so we often allude to it in our applications of the framework to different data structures.

When all the writes satisfy the strong forepassed condition, this corresponds to the preservation of reachability to locations of modification from Feldman et al. [2018].

Example 5.5. In the LO tree running example, the (strong) forepassed condition holds w.r.t. $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$. The idea is that only deletions reduce the k-reachability of mutable locations, these locations become immutable afterwards, thanks to marking them, thereby satisfying the strong forepassed condition. In more detail, we need only consider modifications to *succ* fields, since *key* is immutable, and other fields are not involved in the definition of $\{-\infty\} \overset{k}{\rightsquigarrow} x$, and thus cannot reduce the reachability of any location (hence satisfying condition 1). These writes are: (1) line 76, which modifies an unreachable (newly allocated) node, hence does not reduce the reachability of any location; (2) line 87, which reduces the k-reachability of $\mathtt{z}.\mathit{key}$ for k that is the inserted key. However, the key field would not be modified later (it is immutable); (3) line 61, which reduces the k-reachability of $\mathtt{s}$. However, $\mathtt{s}$ is marked before the locks are released, and thus future operations would refrain from modifying it (see the assertions in lines 60 and 86). Note that this argument considers how writes affect reachability and the possibility of later writes, and need not resort to complex reasoning about how writes interleave with the traversal's reads (as typical linearizability proofs require). For an illustration of case 2 of the forepassed condition, see §8.2.

The forepassed condition is a property of the writes in the execution, so establishing it requires concurrent reasoning on interleavings of writes (but not on how reads interleave with writes). As illustrated by the example, the necessary reasoning is often very simple, as it does not require the correctness of traversals for the sake of proving the forepassed condition. In other cases, establishing

the forepassed condition can benefit from the properties of writes which are themselves proved based on the correctness of preceding traversals. This is in fact possible, using a proof by induction, as we explain in §7. Overall this leads to simple proofs of the forepassed condition and, as a result, of traversal correctness.

5.3 Main Theorem

We are now ready to state our main theorem that establishes traversal correctness from the ingredients above.

We begin with a lemma that captures the important effect of the forepassed condition: that from whenever a location becomes reachable and onwards, the value it holds directs traversals only to locations that themselves have been reachable (although not necessarily at the same time).

Lemma 5.6. *Consider an execution in timespan* $[t_\star, t^\star]$. *If* $reach(\cdot)$ *is single-step compatible with* $extend_k$ *(§5.1) and all writes satisfy the forepassed condition (Definition 5.3), then if a location is reachable, it will always afterwards point to a location that was reachable:*

$$\forall t \in [t_\star, t^\star].\ \forall t' \in [t, t^\star].\ \forall \ell, \ell'.\ \models_t reach(\ell) \wedge extend_p(\ell, \sigma_{t'}(\ell), \ell') \implies \diamondsuit_{t_\star}^{t'}(reach(\ell'))$$

Proof. Let t, t', ℓ, ℓ' be as in the premise of the lemma, and denote $v = \sigma_{t'}(\ell)$. Our goal is to find $\tilde{t} \in [t_\star, t']$ such that $\models_{\tilde{t}} reach(\ell')$.

We consider two cases depending on whether $\sigma_t(\ell) = \sigma_{t'}(\ell) = v$ holds. Let us first assume that it does. Note that $\models_t reach(\ell)$ and $\text{extend}_p(\ell, v, \ell')$ both hold. Thus, by single-step compatibility, we get $\models_t reach(\ell')$. Thus, letting $\tilde{t} = t$ concludes the lemma for this case.

We now consider the case when $\sigma_t(\ell) \neq \sigma_{t'}(\ell) = v$. There must exist a write w in time $t_w \in (t, t']$ that modifies ℓ to v (that is, $\sigma_{t_w}(\ell) = v$ holds). Recall that all writes satisfy the forepassed condition. Let us first assume that they all satisfy Definition 5.3.(1) on ℓ: i.e., no write in $[t_\star, t^\star]$ reduces the reachability of ℓ. Therefore, neither do writes in $(t, t_w]$. In that case, knowing that $\models_t reach(\ell)$ holds, we get that so does $\models_{t_w} reach(\ell)$. From the premise we have that $\text{extend}_p(\ell, v, \ell')$ and since $\sigma_{t_w}(\ell) = v$ by single-step compatibility we get $\models_{t_w} reach(\ell')$. Overall, in this case, taking $\tilde{t} = t_w$ yields the desired.

Let us now consider the case when there is at least one write $w^\dagger$ at time $t^\dagger \in (t, t^\star]$ reducing the reachability of ℓ. When $t^\dagger \in (t_w, t^\star]$, we establish the lemma analogously to the previous case. Let $t^\dagger \in (t, t_w]$ hold. Definition 5.3.(2) holds of $w^\dagger$. Hence, for w, which occurs at $t_w > t^\dagger$ and writes a value v satisfying $\text{extend}_p(\ell, v, \ell')$, we get that there exists $\tilde{t} \in [t^\dagger - 1, t'] \subseteq [t_\star, t']$ such that $\models_{\tilde{t}} reach(\ell')$, which concludes the proof. □

Theorem 5.7. *Consider an execution in timespan* $[t_\star, t^\star]$ *and a traversal* $\tau = (\ell_1, t_1), \ldots, (\ell_n, t_n), \ell_{n+1}$ *defined through* $extend_p$, *such that* $[t_1, t_n] \subseteq [t_\star, t^\star]$ *and* $\models_{t_\star} reach(\ell_1)$ *hold. If* $reach(\cdot)$ *is single-step compatible with* $extend_p$, *and all writes in the execution satisfy the forepassed condition (Definition 5.3), then* τ *is a correct traversal w.r.t.* $reach(\cdot)$ *and* $t_\star$.

Proof. We do a proof by induction on the length of the traversal. By Definition 4.3, we need to show that $\models_{t_\star} reach(\ell_1)$ holds, and that for every i, $1 \leq i \leq n$, it holds that $\diamondsuit_{t_\star}^{t_i}(reach(\ell_{i+1}))$. The former is the base case of induction and holds trivially as a premise of the theorem. For the latter, we let $t_0 = t_\star$ for convenience of notation, and prove the induction step: assuming the induction hypothesis $\diamondsuit_{t_\star}^{t_i}(reach(\ell_{i+1}))$, we show $\diamondsuit_{t_\star}^{t_{i+1}}(reach(\ell_{i+2}))$.

From the induction hypothesis we have that there there exists $\tilde{t}_i \in [t_\star, t_i]$ such that $\models_{\tilde{t}_i} \mathit{reach}(\ell_{i+1})$. Hence, since $t_{i+1} \geq t_i \geq \tilde{t}_i$, we apply Lemma 5.6 to obtain

$$\models_{\tilde{t}_i} \mathit{reach}(\ell_{i+1}) \wedge \mathrm{extend}_p(\ell_{i+1}, \sigma_{t_{i+1}}(\ell), \ell_{i+2}) \implies \diamondsuit_{t_\star}^{t_{i+1}}(\mathit{reach}(\ell_{i+2})).$$

According to the definition of traversals, we have $\mathrm{extend}_p(\ell_{i+1}, \sigma_{t_{i+1}}(\ell), \ell_{i+2})$, and so the premise of the equation above holds. We conclude $\diamondsuit_{t_\star}^{t_{i+1}}(\mathit{reach}(\ell_{i+2}))$, and with it the induction step. □

Example 5.8. We now apply our main theorem to deduce traversal correctness for traversals over successor links. We showed in Example 5.2 that $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ is single-step compatible with extend_k, and in Example 5.5 that interfering writes satisfy the forepassed condition Definition 5.3. Therefore, any traversal with a base $t_\star$ s.t. $\models_{t_\star} \mathit{reach}(\ell_1)$ starting from ℓ_1 is a correct traversal. In §6 we use this fact to prove properties of the entire traversal in Figure 2 (which starts from tree- and predecessor-links before it traverses successor links).

5.4 Reachability with Another Field

The assertions in lines 39 and 41 are about some point in time in which a location was reachable *and at the same time* a certain field (*rem*) had a certain value (true or false). Our proof technique extends to properties involving reachability of an object *and* the value of a single field in the following way: Consider a location ℓ and a location f (intuitively, f is a field of an object that resides in ℓ). We require the following condition, akin to the strong forepassed condition (Definition 5.4), but that focuses only on ℓ, f:

Definition 5.9. A write w at time t in the execution satisfies the *forepassed* condition w.r.t. to locations ℓ, f if either

(1) $\models_{t-1} \mathit{reach}(\ell) \implies \models_t \mathit{reach}(\ell)$ (that is, ℓ's reachability is not reduced by w); or
(2) no write in time $t' \in [t, t^\star]$ modifies f (that is, f is "immutable" from this time on; this includes the current write w).

Theorem 5.10. *Consider an execution in timespan $[t_\star, t^\star]$. If t and t' are such that $t_\star \leq t \leq t' \leq t^\star$, $\models_t \mathit{reach}(\ell)$ and $\sigma_{t'}(f) = v$ hold, and all writes satisfy Definition 5.9 in the interval $[t_\star, t^\star]$, then it is the case that $\diamondsuit_t^{t'}(\mathit{reach}(\ell) \wedge f = v)$ holds.*

Proof. Let t, t', ℓ, ℓ' be as in the premise. When $\sigma_t(f) = \sigma_{t'}(f)$ holds, we have $\models_t (\mathit{reach}(\ell) \wedge f = v)$, and the theorem holds trivially. In the following, we consider the case when $\sigma_t(f) \neq \sigma_{t'}(f) = v$. There must exist a write w in time $t_w \in (t, t^\star]$ that modifies f to v (and then $\sigma_{t_w}(f) = v$ holds). Recall that all writes satisfy the forepassed condition. Let us assume that there is a write w' at $t'_w \in (t, t_w]$ reducing reachability of ℓ. By Definition 5.9.(2), no later write in $[t'_w, t_w]$ modifies f. Since w modifies f at time t_w, we arrive to a contradiction. Thus, all writes in $(t, t_w]$ satisfy Definition 5.9.(1): no write in $(t, t_w]$ reduces the reachability of ℓ. Knowing that $\models_t \mathit{reach}(\ell)$ holds, we get that so does $\models_{t_w} \mathit{reach}(\ell)$. Finally, when $\sigma_{t_w}(f) = v$, we get $\models_{t_w} (\mathit{reach}(\ell') \wedge f = v)$. □

Example 5.11. In the LO tree, we use this extension to deduce properties such as $\diamondsuit (\{-\infty\} \overset{k}{\rightsquigarrow} x \wedge x.\mathit{rem})$ in line 39 and similarly in line 41. The condition holds for the reachability of x the field and $x.\mathit{rem}$ because when the reachability of x is reduced, it is marked (see Example 5.5), so future writes refrain from modifying $x.\mathit{rem}$ (line 55).

6 LOGICAL ORDERING TRAVERSAL, THE FULL STORY

In this section, we give an overview of how our framework applies in proving the challenging past-reachability assertions, whose proof completes the linearizability proof of the LO tree (§3.2).

To this end, we consider an execution from when the operation begins, t_{begin}, that at time t reaches the following assertions appearing in the proof outline for `contains` in Figure 2:

(i) $\diamondsuit^{t}_{t_{\text{begin}}}(\{-\infty\} \rightsquigarrow x)$ at the traversal over tree- and predecessor-links (lines 16–24,28–30);

(ii) $\diamondsuit^{t}_{t_{\text{begin}}}(\{-\infty\} \overset{k}{\rightsquigarrow} x)$ afterwards, at the traversal over successor links (lines 31–33);

(iii) $\diamondsuit^{t}_{t_{\text{begin}}}(\{-\infty\} \overset{k}{\rightsquigarrow} x \wedge x.rem)$ and $\diamondsuit^{t}_{t_{\text{begin}}}(\{-\infty\} \overset{k}{\rightsquigarrow} x \wedge \neg x.rem)$ afterwards, after further reading the *rem* field (lines 39 and 41).

We establish the first two assertions by applying Theorem 5.7, proving traversal correctness (Definition 4.3) w.r.t. an appropriate reachability predicate and the base time t_{begin}, with an extend$_p$ relation capturing the traversal. We establish the last assertions using Theorem 5.10, our extension for reachability with the value of a single field. The proof of each assertion relies on the preceding ones; we now describe how these proofs progress using our framework.

Case (i). We capture this traversal, using the following **extend** relation:

$$\begin{array}{l}\text{extend}((o, key), \cdot, (o, \mathbf{f}))\\ \text{extend}((o, \mathbf{f}), (o', key), (o', key))\end{array} \qquad \mathbf{f} \in \{left, right, pred\}.$$

For the **reachability predicate**, we take $\{-\infty\} \rightsquigarrow x$, which holds of a state σ iff there is a sequence of locations $\ell_1, \ldots, \ell_{n+1}$ starting from $\ell_1 = (\{-\infty\}, key)$, ending in $\ell_{n+1} = (x, key)$, which is connected via the successors list: for every $i \in [0, n]$, if $\ell_i = (o, key)$ then $\ell_{i+1} = (o, succ)$, and if $\ell_i = (o, succ)$ with $\sigma(\ell_i) = (o', key)$ then $\ell_{i+1} = (o', key)$.

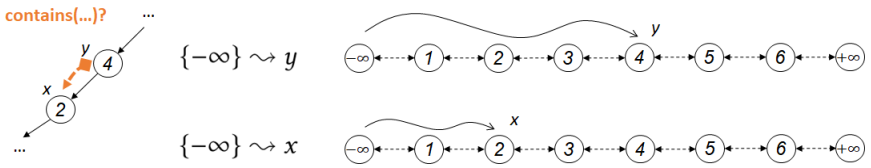


Fig. 5. Single-step compatibility of the tree traversal w.r.t. $\{-\infty\} \rightsquigarrow \cdot$.

single-step compatible with the relation extend. This is because it is an invariant that in $\{-\infty\} \rightsquigarrow$-reachable nodes, *left*-, *right*- and *pred*-links point to other $\{-\infty\} \rightsquigarrow$-reachable nodes: $\forall x, y.\ \{-\infty\} \rightsquigarrow x \wedge x.\mathbf{f} = y \implies \{-\infty\} \rightsquigarrow y$, for $\mathbf{f} \in \{left, right, pred\}$. Figure 5 illustrates how a traversal moving across *left*/*right* pointers remain on nodes that are reachable in the successors list. This invariant holds since `insert` first links a node to the successors list, and `remove` unlinks the successors list only after it unlinks from the tree[7] and the predecessors list; other operations may unlink a node from the tree but not from the successors list.

To be able to apply the framework, we need to prove that interfering writes satisfy the **forepassed** condition. Indeed, the writes in LO satisfy case 1 or case 3 of the forepassed condition, with a gist similar to Example 5.5: the only write that reduces the reachability of a location is the removal of a node in line 61, reducing the reachability of y, but this node is marked removed and no further writes to it will occur, satisfying case 3.

[7]This relies, in line 113, on the invariant that the sole parent of x is $x.parent$, when x is unlocked. Note that modifications to the *parent* field are protected by the parent's `treeLock` (in other words, a linked node's parent field is written to only when the parent's treeLock is held—which occurs during removal and rotations), and thus `c`, `n.left`, and `n.right` do not have to be locked in `removeFromTree` since only their *parent* field is modified.

From these ingredients, Theorem 5.7 **yields the desired assertion**. Formally, we consider any traversal $\tau = (\ell_1, t_1), \ldots, (\ell_n, t_n), \ell_{n+1}$ w.r.t. extend occurring within an execution in timespan $[t_{\text{begin}}, t]$, so that $\ell_1 = (\{-\infty\}, key)$, ending in $\ell_{n+1} = (x, key)$. By Theorem 5.7, τ is correct w.r.t. $\{-\infty\} \rightsquigarrow \cdot$ and t_{begin}, which concludes the case (i).

Between (i) and (ii). The choice of the reachability predicate in assertion (i) is with the aim of proving assertion (ii), which concerns $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ in the successors list. In assertion (i) we used a different reachability predicate, of plain-reachability through successor links, without considering any specific key. The reason is that the traversal over tree- and predecessor-links visits nodes in an order that is, in some sense, "random access" into the successors list, and does not respect the search for k in the successors list. However, assertion (i) is important for proving assertion (ii). The necessary "glue" is the following observation of what holds in between them, before the first iteration in line 31:

Let z be the value of x just after the loop at lines 28 to 30. From this loop's condition, necessarily $z.key \leq k$. From assertion (i), there is some timestamp $t' \geq t_{\text{begin}}$ when $\{-\infty\} \rightsquigarrow z$ holds. The successors list is sorted and contains unique values, an invariant that can be established from the assertions as (see §3.2). Hence, it also holds that $\{-\infty\} \overset{k}{\rightsquigarrow} z$ at time t'. Thus assertion (ii) holds just before the loop in lines 31 to 33. Our goal now is to prove this assertion also when this loop executes and `contains` traverses successor links.

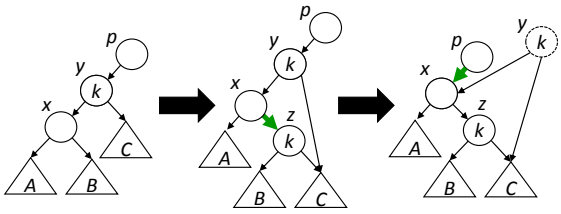
Case (ii). To prove this assertion we consider a traversal over (only) successor links that starts from z—where the traversal over tree- and predecessor-links left off—strictly after that traversal: $\tau' = (\ell'_1, t'_1), (\ell'_2, t'_2), \ldots, (\ell'_m, t'_m), \ell'_{m+1}$, where $\ell'_1 = (z, key)$ and $t'_1 \geq t'$ (as t' occurred sometime during the previous traversal).

As a traversal over successor links, it visits locations connected by the **extend**$_k$ relation from Example 4.1. We study its correctness w.r.t. the **reachability predicate** $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ from Example 4.2. As we have shown in Example 5.2, $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ is **single-step compatible** with extend$_k$. We also proved in Example 5.5 that the LO tree's writes satisfy the **forepassed** condition. At time t', the first location is k-reachable: $\{-\infty\} \overset{k}{\rightsquigarrow} (z, key)$. From these, by Theorem 5.7 we get that τ' is a correct traversal w.r.t. $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ and the base time t'. By Definition 4.3, $\diamondsuit^t_{t'}(\{-\infty\} \overset{k}{\rightsquigarrow} x)$ holds, and in particular $\diamondsuit^t_{t_{\text{begin}}}(\{-\infty\} \overset{k}{\rightsquigarrow} x)$, which concludes the case (ii).

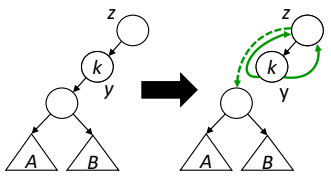
Case (iii). We apply the extended framework of §5.4 in our proofs of $\diamondsuit(\{-\infty\} \overset{k}{\rightsquigarrow} x \wedge x.rem)$ and $\diamondsuit(\{-\infty\} \overset{k}{\rightsquigarrow} x \wedge \neg x.rem)$. From assertion (ii), there is a point in time $\tilde{t} \in [t_{\text{begin}}, t]$ where $\{-\infty\} \overset{k}{\rightsquigarrow} x$ holds. We consider any possible execution with timespan $[\tilde{t}, t]$. Let v be the value of $x.\texttt{rem}$ returned by the read at line 38, i.e. $v = \sigma_{t'}(x.\texttt{rem})$. As we have shown in Example 5.11, the LO tree's writes satisfy the **forepassed** condition w.r.t. ℓ and $x.\texttt{rem}$. By Theorem 5.10, $\diamondsuit(\{-\infty\} \overset{k}{\rightsquigarrow} x \wedge x.rem = v)$ holds, which concludes the case (iii).

7 DISCUSSION: ON PROVING THE FOREPASSED CONDITION

The forepassed condition (Definition 5.3) is the key requirement to algorithm implementations in our framework. While it is simple to establish in the LO tree (§6), in general this requires reasoning about concurrent executions. However, this task can be simplified by relying on assertions showing the correctness of writes. Since proofs of traversal correctness with our framework are carried out by induction on the length of concurrent executions, inductive arguments for the forepassed



(a) Right rotation of y in the Contention-Friendly Tree, from [Feldman et al. 2018]. (The bold green link is the one written in each step. The node with a dashed border has its *rem* bit set.)



(b) Bypassing y and backtracking from it by pointing from y to its parent z.

Fig. 6. Operations in the contention-friendly tree.

condition can be integrated into the proof. This introduces a curious circularity: the forepassed condition on the prefix of an execution is used to conclude correctness of the corresponding traversal, which in turn can be leveraged in justifying the forepassed condition on a longer prefix of the execution. The integration is possible because justifying traversal correctness after a prefix of an execution requires the forepassed condition to hold only on that prefix. A similar approach has been previously proposed by Feldman et al. [2018].

8 ADDITIONAL CASE STUDIES

8.1 List-Based Structures

Our method can prove all the list-based structures handled by the local view framework [Feldman et al. 2018]: Lazy List [Heller et al. 2005], lock-free list [Herlihy and Shavit 2008, Chapter 9.8], and and lock-free skiplist [Herlihy and Shavit 2008, Chapter 14.4]. This is because the traversals in all these examples are single-step compatible with the reachability predicate (similar to Example 5.2), and the preservation condition of Feldman et al. [2018] is a special case of forepassed interference (see Definition 5.4).

8.2 Contention-Friendly Tree with Backtracking

The contention-friendly tree [Crain et al. 2013a, 2016] is a self-balancing binary search tree, in which traversals operate without synchronization, and rotations are performed by allocating a copy of the rotated node (see Figure 6a). The linearizability proof of this tree uses the k-reachability predicate $\texttt{root} \overset{k}{\rightsquigarrow} x$, meaning that a node x is on a path from the `root` in a standard tree binary search for key k [Feldman et al. 2018]. Feldman et al. [2018] proved traversal correctness of a variant of this algorithm, but their proof cannot handle *backtracking*. In the original version [Crain et al. 2013a, 2016], when a node x is physically removed, its *left*/*right* pointers are modified to point to its parent (see Figure 6b). In this way, a traversal reaching a physically removed node backtracks until it can continue from a node still linked to the tree [Crain et al. 2016], without requiring traversals to perform explicit synchronization or validation steps [Bronson et al. 2010]. This backtracking-like operation is inherently problematic for the framework of Feldman et al. [2018] because it breaks their temporal acyclicity requirement: what was once a child of a node is now its parent.

We apply our framework to prove traversal correctness even in the presence of backtracking. For a key k, we capture the traversal searching for k using the following extend_k relation, defined

to be true iff it is one of the following cases:

$$\begin{aligned}
&\mathrm{extend}_k((o, key), m, (o, right)) && \text{if } m < k\\
&\mathrm{extend}_k((o, key), m, (o, left)) && \text{if } m > k\\
&\mathrm{extend}_k((o, left), (o', key), (o', key))\\
&\mathrm{extend}_k((o, right), (o', key), (o', key))
\end{aligned}$$

We define the k-reachability predicate through sequences of locations that follow extend: $\texttt{root} \overset{k}{\rightsquigarrow} x$ holds in state σ if there is a sequence of locations $\ell_0, \ldots, \ell_n$ s.t. $\ell_0 = (\texttt{root}, key)$, $\ell_n = x$, and $\forall i < n.\ \mathrm{extend}_k(\ell_i, \sigma(\ell_i), \ell_{i+1})$. These definitions exactly follow a binary search in the tree, that is: if k is greater (smaller) than the current key, the path continues through the right (left) child respectively. Note that the path does not continue after finding the target key. Since $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ is defined using extend_k, their **single-step compatibility** is evident.

The **forepassed** condition holds because in this algorithm, when the k-reachability of a node is reduced, the node is marked, so future operations do not modify it (similar to Example 5.5, *except* the backtracking modifications—which do modify the node's pointers *after* it is no longer reachable. However, these modifications satisfy case 2 of the forepassed condition, because they point to the parent, which has been k-reachable itself at the time that its child's reachability was reduced. It is important to note that rotations, which could reduce the k-reachability of the node rotated downwards (because binary searches now encounter the node rotated upwards first and can continue in the other direction), satisfy the forepassed condition in the CF tree, because rotations in the CF tree use a *newly allocated* node to represent the node rotated down (see Figure 6a). See the extended version [Feldman et al. 2020] for the code and a detailed discussion of the traversal correctness proof of this algorithm.

8.3 Citrus Tree

The Citrus tree [Arbel and Attiya 2014] is a concurrent binary tree implementing a key-value map with the standard operations `insert`(k,d), `delete`(k), and `contains`(k) operations. The tree's nodes include a `rem` boolean field indicating logical removal (like in the LO tree), and a `tag` integer field, used to prevent an ABA problem due to multiple nullifications of the `left` field upon insertion. (Nodes also contain `key`, `data`, `left`, and `right` fields.) The operations of this concurrent map performs lock-free binary search in the tree. In the following, we discuss the use of our framework in proving their correctness (more details are given in the extended version [Feldman et al. 2020]).

The most intricate part of this algorithm is the physical removal of a node with two children as part of `delete`. (Indeed, many concurrent tree algorithms with optimistic traversals refrain from physically removing nodes until they have only one child [e.g. Bronson et al. 2010; Crain et al. 2016].) In the Citrus tree, this is done as pictured in Figure 7. (The assignments to the field `key` should be ignored for now.) Let y be a node with two children. The operation finds cs, the successor of y in the tree, and creates a copy w of cs. It then performs the following mutations (see the labels in Figure 7): (1) setting w's left and right children to be the same as in y; (2) linking w to the tree as the left child of y (at this point y is unlinked from the tree); and (3) finally, unlinking cs from the tree. Nodes are first marked logically deleted before being unlinked from the tree. Every write to a node is protected by a lock associated to that node. Also, crucially, unlinking cs from the tree is guarded by an RCU lock [Desnoyers et al. 2012; McKenney 2004; McKenney and Slingwine 1998] that synchronizes this write with traversals executing in parallel: it blocks this write until all the traversals that have already started finish.

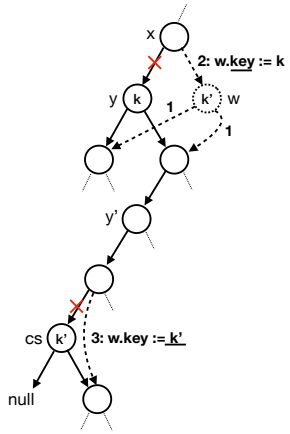


Fig. 7. Removing a node y with two children by creating a copy w of the node cs containing the smallest key k' bigger than the key k of y (the *successor*). Dashed edges represent changes of pointers made during this removal. They are labeled with integers that show their order and updates of the ghost field $\underline{\text{key}}$.

Proving that removals of nodes with two children do not hinder the results of concurrent lock-free traversals is quite tricky. We show that the lock-free traversals in Citrus are *not* correct with respect to the standard k-reachability predicate $\texttt{root} \overset{k}{\rightsquigarrow} \cdot$ (§8.3.1), leading us to defined a weaker version of the reachability predicate (§8.3.2), to which the framework applies (§8.3.3). Correctness with respect to this weaker k-reachability predicate is however enough to prove linearizability (§8.3.4).

8.3.1 Warmup Attempt: Traversals Fail Standard Reachability. We first show that our framework (Theorem 5.7) fails to prove traversal correctness w.r.t. $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ (which follows a binary search in the tree like in the previous case studies), and that this is due to the fact that the traversals are not correct w.r.t. this reachability predicate.

The traversal is single-step compatible with $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$. However, the forepassed condition w.r.t. $\{-\infty\} \overset{k}{\rightsquigarrow} \cdot$ does not hold. When a node with two children is removed as in Figure 7, the write that links x to w reduces reachability in the subtree of w. Specifically, it no longer holds that $\{-\infty\} \overset{\tilde{k}}{\rightsquigarrow} \alpha.\mathit{left}$ for every node α in the right subtree of w and for every $\tilde{k}$ in the interval $(k, k']$ (upon this modification, $\tilde{k}$-paths go to the left of w or stop at w, rather than going to the right as before). However, these fields *can* be modified afterwards, in a way that increases reachability, thereby *violating the forepassed condition*. This occurs when a new node β is inserted as the left child of some α, thus modifying $\alpha.\mathit{left}$ to point to β which is *not* $\tilde{k}$-reachable at the point of insertion and of course also not before. (Such a node β can be inserted when the left child of α is deleted and it has two children, in which case β serves as a copy of the successor of the left child of α. Note that this successor is not cs, so this additional removal operation is possible concurrently.) This constitutes a violation the forepassed condition, and Theorem 5.7 does not apply.

In fact, it is not only that our main theorem cannot prove traversal correctness, but traversal correctness *does not hold* with respect to the standard k-reachability predicate. In the same scenario, a $\tilde{k}$-traversal that happens to reside at y when the modification linking w occurs and continues from there may reach the new node β from above although β was never $\tilde{k}$-reachable.

We use our framework to show that the traversals in this algorithm are correct w.r.t. $\texttt{root} \overset{k}{\dashrightarrow} \cdot$, a *weaker* reachability predicate. Intuitively, $\texttt{root} \overset{k}{\dashrightarrow} x$ allows some searches to "take a wrong turn"—in a way we make accurate hereafter—due to a concurrent removal of a node with two children. The assertion $\diamondsuit(\texttt{root} \overset{k}{\dashrightarrow} x)$ implied by traversal correctness makes it possible to infer $\diamondsuit(\texttt{root} \overset{k}{\rightsquigarrow} x)$ (the form of reachability that underlies the abstraction function in the linearizability proof) in special cases, such as when x is the endpoint of the traversal. We first explain how $\texttt{root} \overset{k}{\dashrightarrow} \cdot$ is defined, then why our framework applies to this reachability predicate, and then sketch how $\texttt{root} \overset{k}{\rightsquigarrow} \cdot$ can be derived from it as necessary.

8.3.2 Weak Reachability Predicate Using Ghost State. The predicate $\texttt{root} \overset{k}{\dashrightarrow} \cdot$ is defined on top of an instrumentation of the algorithm with *ghost code* that statically captures the ways traversals looking for a key k can deviate from standard k-search paths (defined by the predicate $\texttt{root} \overset{k}{\rightsquigarrow} \cdot$). Looking at Figure 7, we refer to the deviation that occurs after linking x to w and define this predicate such that $\texttt{root} \overset{\tilde{k}}{\dashrightarrow} \ell$ holds for keys $\tilde{k} \in (k, k']$ and locations ℓ in the right sub-tree of w even after x is linked to w (the problematic case mentioned above). Technically, we introduce a *ghost* field $\underline{\texttt{key}}$ for every node in the tree, which does not change the actual behavior of the algorithm. The field $\underline{\texttt{key}}$ equals $\texttt{key}$ unless the node is a copy w introduced during the physical removal of a node y with two children (cf. Figure 7): $\underline{\texttt{key}}$ is set to k atomically with linking x to w, and it is set to k' atomically with unlinking cs from the tree (thus becoming equal to $\texttt{key}$). We refer to the last operation as "collapsing" the ghost interval; this indicates the end of this operation—from this point on, traversals are *not* permitted to steer off course because of this operation. (Collapsing the ghost interval is necessary to be able to infer interesting properties of $\texttt{root} \overset{k}{\rightsquigarrow} \cdot$ out of weak reachability.) This ghost state is defined *per traversal*: a traversal starts with a copy of the state in which there is no ghost state ($\underline{\texttt{key}} = \texttt{key}$ in all nodes), and a write modifies the ghost state of all the traversals that have already started and did not terminate.

The predicate $\texttt{root} \overset{k}{\dashrightarrow} x$ holds in a state σ if and only if there is a sequence of locations $\ell_1, \ldots, \ell_{n+1}$ starting from $\ell_1 = (\texttt{root}, key)$, ending in $\ell_{n+1} = (x, key)$, and connected via the $\texttt{left}$ and $\texttt{right}$ fields in the following manner: for every $i = 1, \ldots, n$,

$$\begin{aligned}
&\text{if } \ell_i = (o, \texttt{left}) \text{ or } \ell_i = (o, \texttt{right}) \text{ with } \sigma(\ell_i) = (o', key), \text{ then} \\
&\qquad\qquad \ell_{i+1} = (o', key), \text{ and} \\
&\text{if } \ell_i = (o, \texttt{key}),\ \sigma(o, \texttt{key}) = m,\ \sigma((o, \underline{\texttt{key}})) = \underline{m}, \text{ then} \\
&\qquad\qquad \ell_{i+1} = (o, \texttt{right}) \text{ if } (k > \underline{m} \wedge k \neq m) \vee (k = m \wedge \underline{m} \neq m), \text{ and} \\
&\qquad\qquad \ell_{i+1} = (o, \texttt{left}) \text{ if } k < m.
\end{aligned} \tag{1}$$

Note that standard k-reachability $\texttt{root} \overset{k}{\rightsquigarrow} \ell$ implies that $\texttt{root} \overset{k}{\dashrightarrow} \ell$. However, this predicate is weaker than $\texttt{root} \overset{k}{\rightsquigarrow} \ell$ because: (1) the search for $k \in (\underline{m}, m)$ can go either left or right, and (2) the search for m can continue (to the right) after finding the key ($k = m$) in case the ghost key and the real key are different. The updates on the ghost field $\underline{\texttt{key}}$ are visible only to the traversals executing in parallel (this is related to the use of the RCU lock). In more detail, each operation has its own instance of $\underline{\texttt{key}}$ for every node, and an update to this field should be read as modifying all the instances of the traversals executing in parallel atomically in one shot.

8.3.3 Applying the Framework. First, the predicate $\texttt{root} \overset{k}{\dashrightarrow} \cdot$ is **single-step compatible** with the standard extend_k relation corresponding to binary search tree traversals (formally defined in the extended version [Feldman et al. 2020]). Essentially, extending a sequence of locations satisfying the relationship of Equation (1) with a another location chosen according to binary search continues to satisfy Equation (1).

Second, showing that all the writes in this algorithm satisfy the strong **forepassed** condition w.r.t. $\texttt{root} \overset{k}{\dashrightarrow} \cdot$ is relatively straightforward: Writes made within an `insert` operation only increase the reachability of locations. Almost all the writes in `delete` satisfy forepassed for the usual reason (cf. Example 5.5): they reduce the reachability only of nodes that are marked logically deleted, which would never be modified. The only tricky write is the last step in the removal of a node with two children, unlinking the node *cs* in Figure 7, because it "collapses" the ghost field $\underline{\texttt{key}}$ k' to the actual key k. This reduces $\texttt{root} \overset{\tilde{k}}{\dashrightarrow} \cdot$ for all the nodes in the right sub-tree of w and $\tilde{k} \in (k, k']$, which, when $k \neq k'$, is problematic because they can later be modified (as explained above in why $\texttt{root} \overset{k}{\rightsquigarrow} \cdot$ does not satisfy forepassed). This is where the RCU synchronization comes into play, together with our definition of the ghost code: if $\underline{\texttt{key}} \neq \texttt{key}$, then concurrently with the traversal there was a write introducing this ghost. But then this traversal must terminate *before* the collapsing of the ghost interval, because the RCU synchronization waits for existing traversals to terminate before performing this write. Thus, this *reduction* of reachability—from $\underline{\texttt{key}} \neq \texttt{key}$ to $\underline{\texttt{key}} = \texttt{key}$—cannot occur. Note that this argument uses the fact that the ghost state is per traversal and introduced by a write only in the existing, concurrent, traversals.

We can now apply Theorem 5.7 to deduce that traversals in the Citrus tree are correct w.r.t. $\texttt{root} \overset{k}{\dashrightarrow} \cdot$.

8.3.4 Proving Linearizability. So far we have established that every location x the traversal reached satisfies $\diamondminus\, (\texttt{root} \overset{k}{\dashrightarrow} x)$. To prove linearizability, we use this fact in order to infer properties of the standard reachability predicate. For example, in `contains`, $\diamondminus\, (\texttt{root} \overset{k}{\rightsquigarrow} x)$ holds when x is the *endpoint* of the traversal (the last node it reached). This can be inferred from $\texttt{root} \overset{k}{\dashrightarrow} x$ as follows. This implication is immediate for traversals that do not pass through copies of nodes w like in Figure 7 because the ghost field $\underline{\texttt{key}}$ can differ from `key` only for such nodes. Moreover, a traversal passing through such a node can deviate from a standard search path only when looking for a key $\tilde{k} \in (k, k']$. If it ends in *null*, then $\tilde{k} \in (k, k')$ and $\texttt{root} \overset{\tilde{k}}{\rightsquigarrow} \mathit{null}$ was indeed true in the past (the algorithm ensures that k' is the smallest key bigger than k). Otherwise, if it ends in a non-null node x, then $\tilde{k} = k'$ and x is the node *cs* in Figure 7. Again, this node was k-reachable in the past, concluding that indeed $\diamondminus\, (\texttt{root} \overset{k}{\rightsquigarrow} x)$, as desired. Slightly different properties are required for the linearizability of `insert` and `delete`. For instance, inserting to a left child uses `tag` validation to infer the that the modified node is reachable *now* w.r.t. the *standard* reachability predicate (from $\diamondminus\, (\texttt{root} \overset{k}{\dashrightarrow} x)$). These are discussed in the extended version [Feldman et al. 2020]. The correctness of the traversal that searches for the successor node (as part of the operation that removes a node with two children), which is subject to different reachability patterns from the one in `contains` (and serves a different purpose), is also discussed in the extended version [Feldman et al. 2020].

9 RELATED WORK

A couple of prior methods [Feldman et al. 2018; O'Hearn et al. 2010] also prove traversal correctness by reasoning strictly about how the algorithm's writes modify the memory state (i.e., by considering

executions as sequences of interleaving writes, rather than interleavings of writes with traversals' reads) plus static *sequential* properties of the traversal code. Our method is more general: the hindsight lemma [O'Hearn et al. 2010] is specific to linked lists; in particular, they do not clearly divide between the invariants used for traversal correctness and the rest of the proof. Feldman et al. [2018] can handle algorithms beyond the list, but require that pointers always form an acyclic structure. In this work we relieve traversal correctness from the acyclicity condition, and instead build on single-step compatibility (§5.1). This allows us to prove examples that Feldman et al. [2018] cannot handle, including the contention-friendly tree with backtracking and the Logical-Ordering tree (which includes in-place rotations). Our framework also has the benefit of significantly simpler theory behind it. The proof of our main theorem (Theorem 5.7) is more similar to the hindsight lemma, by induction over the traversals' reads and case-splitting on whether the link was modified, rather than induction over interleaving writes as in Feldman et al. [2018].

Other works aim to simplify elements of the linearizability other than traversal correctness. Lev-Ari et al. [2015b] harness properties of sequential executions in the linearizability proof. Their methodology relies on *base points*, points during the concurrent execution where certain predicates hold, thus necessitating concurrent reasoning. When applying their framework to the lazy list, they rely on the tricky concurrent reasoning from previous works [O'Hearn et al. 2010; Vafeiadis et al. 2006] to establish base points. Our work is thus complementary, aiming to simplify concurrent reasoning of the sort that could also be used to establish base points.

The Edgeset framework [Shasha and Goodman 1988] proves the linearizability of concurrent search algorithms based on the notion of *keyset*, which is reminiscent of k-reachability. This framework has recently been the algorithmic basis for the mechanizations by Krishna et al. [2020, 2018], where the main technical contribution is showing how to obtain a mechanized proof of the Edgeset arguments in the Iris separation logic [Jung et al. 2018] using the novel notion of flows (cf. [Krishna et al. 2020, §7]). This development is orthogonal to our work, where we focus on the algorithmic essence of correctness, independent of a particular assertion language or program logic. Shasha and Goodman [1988] provide three algorithmic templates, and conditions for when these templates guarantee linearizability. Of the three templates, the one closest to optimistic traversals is the link template. The Edgeset condition for the link template requires that a node gets *accessed* for key k *only when* it is k-reachable or has an outgoing path leading to a k-reachable node. This condition does not hold in our examples, where a node reached by `contains`(k) may no longer be reachable, nor lead to a k-reachable node afterwards.[8] In contrast, the forepassed condition allows such accesses, but only constrains later modifications to such locations. The fundamental difference between the forepassed condition and the Edgeset conditions is that their conditions guarantee the existence of a path *now*, whereas the forepassed condition applies to algorithms where it can only be shown that a path existed *at some point*. Furthermore, proving the link condition involves reasoning about interleavings of both reads and writes, whereas our forepassed condition involves interleavings of writes only. Indeed, Krishna et al. [2020] use a strengthening of the Edgeset condition for the link technique, requiring that the set of nodes that is k-reachable or leads to a k-reachable node is never reduced by interleaving writes. This simplifies the reasoning, but is also too strong for optimistic traversals.

[8] For example, consider the lazy list [Heller et al. 2005] (which is essentially the list traversal in the LO tree—also see §8). Suppose the list is $A \mapsto B \mapsto C \mapsto D$ and a read-only `contains`(D) has read the pointer $A \mapsto B$. If now B, C, and D are removed—in this order—by another thread, the `contains`(D) will traverse the path $B \mapsto C \mapsto D$. Such a traversal is prohibited by the link template, because the traversed nodes are not reachable nor lead to a reachable node, and so the `contains`(D) is not allowed to access B, and similarly for C (cf. Shasha and Goodman [1988, §4.5]). Still, C and D *have been* reachable, and so such accesses are allowed by our framework.

The designers of certain data structures proved correctness in ways that share some of the structure of the argument in our framework. Arbel and Attiya [2014] prove that every node accessed in a traversal has been reachable at some point (Lemma 1), but this is shown for plain reachability rather than k-reachability, which is essential in binary search trees (e.g. for the correctness of insertions). The proof is specific to the Citrus tree, and does not seem to use a condition similar to forepassed. Brown et al. [2014] prove a lock-free self-adjusting binary tree, establishing traversal correctness (Lemma 18) based on the fact that nodes that are not in the abstract set are finalized. This is a specific case of our condition (see Definition 5.4). Our work distills the ingredients needed for a general proof technique, which can prove traversal correctness for multiple different algorithms.

Program logics for compositional reasoning about concurrent programs and data structures have been studied extensively. 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 [Owicki and Gries 1976] and Rely-Guarantee [Jones 1983] logics, various extensions of Concurrent Separation Logic [Bornat et al. 2005; Brookes 2004; O'Hearn 2004; Parkinson et al. 2007] have been proposed in order to reason compositionally about different instances of fine-grained concurrency, e.g. [da Rocha Pinto et al. 2014; Dragoi et al. 2013; Jung et al. 2018, 2020; Krishna et al. 2018; Ley-Wild and Nanevski 2013; Nanevski et al. 2019; Raad et al. 2015; Sergey et al. 2015; Turon et al. 2013; Vafeiadis 2008, 2009]. 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. It is interesting to note that the lazy list has played an important case study in several of these works [Vafeiadis 2008; Vafeiadis et al. 2016], and recently some works [Krishna et al. 2020, 2018] have used the Edgeset framework [Shasha and Goodman 1988] for abstracting some of the reasoning.

Some works [e.g. Abdulla et al. 2013; Amit et al. 2007] attempt at more automatic verification of concurrent data structures. However, they apply in cases where the linearization point of every invocation is *fixed* to a particular statement in the code. This is not the case in the algorithms considered in this paper where for instance, the linearization point of `contains(k)` invocations is not fixed. Generic reductions of linearizability to assertion checking [e.g. Bouajjani et al. 2013, 2015, 2017; Henzinger et al. 2013; Liang and Feng 2013; Vafeiadis 2010; Zhu et al. 2015] apply also 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.

10 CONCLUSION

In this paper we have presented a simple and effective method for proving traversal correctness and showed its applicability to several complex concurrent search data structures. Our main observation is that proving traversal correctness is possible by analyzing the effect of writes on (static) reachability, guaranteeing a consistent extension of the traversal in each point, while relying on the local nature of the decisions made by traversals in each step. In a sense, this result demonstrates, surprisingly, that extremely sophisticated concurrency techniques can be tamed using general, comprehensible principles. We hope that this can direct exploration of new algorithms in the design space. Moving forward to even more intricate data structures, it would be useful to explore general concepts for elements of correctness beyond traversals, such as synchronization patterns in lock-free algorithms.

We have demonstrated the applicability of our proof framework in simplifying "pen and paper" proofs. Leveraging our proof argument in a mechanized proof as important future work. The technique of Krishna et al. [2020] seems a promising starting point. It does not currently handle

unfixed linearization points, which are required in the algorithms we consider. It will also be interesting to see how our proof technique and its decomposition to traversal correctness will interact with new ideas in concurrent separation logics, such as the use prophecy variables in Hoare-style proofs [Jung et al. 2020], to successfully mechanize the proofs of the challenging algorithms we consider. Such mechanization would also need to tackle the need to reason about reachability invariants. Our experience is that the specific forms of reasoning required by applications of our framework usually benefit from the local nature of the modifications. For example, calculating which locations suffered a k-reachability reduction is typically obtained from the premise that the location of modification is k-reachable, sometimes employing a few relatively simple invariants (e.g. that a list is sorted). Translating these arguments into a mechanized proof will be interesting future work.

ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful comments. This research was partially supported by the European Union's Horizon 2020 research and innovation program (grant agreement No. 678177, 724464, and 759102), the Spanish MICINN project BOSCO (PGC2018-102210-B-I00), the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, the Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, the Pazy Foundation, and the Israel Science Foundation (ISF) grant No. 1996/18, 2005/17, and 1810/18.

REFERENCES

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *TACAS*. 324–338.

Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *CAV '07 (LNCS)*, Vol. 4590. 477–490.

Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *PODC 2014*.

Hagit Attiya, G. Ramalingam, and Noam Rinetzky. 2010. Sequential verification of serializability. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 31–42. https://doi.org/10.1145/1706299.1706305

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

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *ESOP '13 (LNCS)*, Vol. 7792. Springer, 290–309.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. 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*. 95–107.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *PPoPP 2010*.

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *PPoPP 2014*.

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

Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013a. A Contention-Friendly Binary Search Tree. In *Euro-Par 2013*.

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

Tyler Crain, Vincent Gramoli, and Michel Raynal. 2016. A Fast Contention-Friendly Binary Search Tree. *Parallel Processing Letters* 26, 03 (2016).

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

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

Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Trans. Parallel Distrib. Syst.* 23, 2 (2012), 375–382.

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

Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. 2013. Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates. In *CAV '13 (LNCS)*, Vol. 8044. Springer, 174–190.

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

Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC 2018*.

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving Highly-Concurrent Traversals Correct. *CoRR* (2020). https://arxiv.org/abs/2010.00911

Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, Computer Laboratory.

Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP 2015*.

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

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Bill Scherer, and Nir Shavit. 2005. A Lazy Concurrent List-based Set Algorithm. In *OPODIS 2005*.

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR*. 242–256.

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

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

M. P. Herlihy and J. M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. 12, 3 (1990).

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

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *IFIP Congress*. 321–332.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Siddharth Krishna, Nisarg Patel, Dennis Shasha, , and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*. ACM.

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

Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. 2015a. A Constructive Approach for Proving Data Structures' Linearizability. In *DISC 205*.

Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. 2015b. A Constructive Approach for Proving Data Structures' Linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*. 356–370. https://doi.org/10.1007/978-3-662-48653-5_24

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

Hongjin Liang and Xinyu Feng. 2013. 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*. 459–470.

Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys 2012*.

Paul McKenney. 2004. *Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels*. Ph.D. Dissertation. OGI.

Paul E. McKenney and John D. Slingwine. 1998. Read-copy update: using execution history to solve concurrency problems. In *PDCS*.

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

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying concurrent programs in separation logic: morphisms and simulations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 161:1–161:30. https://doi.org/10.1145/3360587

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

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. 2010. Verifying Linearizability with Hindsight. In *PODC 2010*.

Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976), 279–285. https://doi.org/10.1145/360051.360224

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

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *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*. 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

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

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *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. 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 92–110. https://doi.org/10.1145/2983990.2983999

Dennis E. Shasha and Nathan Goodman. 1988. Concurrent Search Structure Algorithms. *ACM Trans. Database Syst.* 13, 1 (1988), 53–90.

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

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *SOSP 2013*.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390. https://doi.org/10.1145/2500365.2500600

V. Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Ph.D. Dissertation. University of Cambridge.

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *VMCAI '09: Proc. 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 5403. Springer, 335–348.

Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

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

Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. 2016. Proving correctness of highly-concurrent linearisable objects. In *PPOPP '06*. ACM, 129–136.

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. 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*. 3–19.