Verifying Lock-free Search Structure Templates

Nisarg Patel
New York University

Dennis Shasha
New York University

Thomas Wies
New York University

Abstract

We present and verify template algorithms for lock-free concurrent search structures that cover a broad range of existing implementations based on lists and skiplists. Our linearizability proofs are fully mechanized in the concurrent separation logic Iris. The proofs are modular and cover the broader design space of the underlying algorithms by parameterizing the verification over aspects such as the low-level representation of nodes and the style of data structure maintenance. As a further technical contribution, we present a mechanization of a recently proposed method for reasoning about future-dependent linearization points using hindsight arguments. The mechanization builds on Iris' support for prophecy reasoning and user-defined ghost resources. We demonstrate that the method can help to reduce the proof effort compared to direct prophecy-based proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Separation logic; Theory of computation → Shared memory algorithms

Keywords and phrases skiplists, lock-free, separation logic, linearizability, future-dependent linearization points, hindsight reasoning

Related Version *Extended Version*: `https://arxiv.org/abs/2405.13271`

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.11051385`

Funding This work is funded in parts by NYU Wireless and by the United States National Science Foundation under grants CCF-2304758, 1840761, 2304758, and 25-74100-F1202. Further funding came from an Amazon Research Award Fall 2021. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of Amazon.

Acknowledgements We thank Sebastian Wolff for many insightful discussions and his suggestions to improve the presentation of the paper.

1 Introduction

A search structure is a key-based store that implements a mutable map of keys to values (or a mutable set of keys). It provides five basic operations: (i) create an empty structure, (ii) insert a key-value pair, (iii) search for a key and return its value, (iv) delete the entry associated with a key, and (v) update the value associated with a particular key. Because of their general usefulness, search structures are ubiquitous in data-intensive workloads.

Earlier works [20, 35, 19] developed a framework to verify a wide range of lock-based implementations of concurrent search structures. Specifically, they proved that these implementations are linearizable [12].

A core ingredient of the framework is the idea of template algorithms [38]. A template algorithm dictates how threads interact but abstracts away from the concrete layout of nodes in memory. Once the template algorithm is verified, its proof can be instantiated on a variety of search structures.

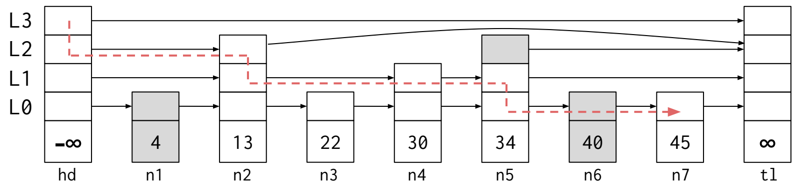

Figure 1 Skiplist with four levels. A node that is marked (logically deleted) at a level is shaded gray at that level. The red line indicates the path taken by a traversal searching for key 42.

The template algorithms of [20, 35, 19] use locks as a synchronization technique. Locks ensure non-interference on portions of memory to guarantee that certain needed constraints hold in spite of concurrency.

The disadvantage of locks is that if a thread holding a lock on some portion of memory p stops, then no other thread can get a conflicting lock on p. For that reason, some practical implementations such as Java's `ConcurrentSkipListMap` [34] use lock-free algorithms.

This paper shows how to capture multiple variants of concurrent lock-free skiplists and linked lists in the form of template algorithms. Thus, proving the correctness of such a template algorithm results in a proof that is applicable to many variants at once. Our template algorithms are parametric in the skiplist height and allow variations along the following three dimensions: (i) maintenance style (eager vs lazy) (ii) node implementations and (iii) the order of maintenance operations on the higher levels of the skiplists.

By instantiating our template algorithm with appropriate maintenance operations and node implementations we obtain verified versions of existing (skip)list algorithms from the literature such as the Herlihy-Shavit skiplist algorithm [11, § 14], the Michael set [32], and the Harris list algorithm [10]. We also obtain a new concurrent skiplist algorithm that has not been considered before. The new algorithm is correct by construction thanks to our modular verification framework.

We mechanize our development in the concurrent separation logic Iris [15, 17]. One technical contribution of our work is a formalization of *hindsight reasoning* [33, 23, 7, 8, 27, 28] in Iris. Hindsight reasoning has shown its usefulness in dealing with future-dependent and external linearization points, a challenge that commonly arises in lock-free data structures.

Specifically, we build on the hindsight theory developed in [28], providing a mechanism in Iris where one can establish that a linearization point has passed by inferring knowledge about past states using a form of temporal interpolation.

To our knowledge, our development is the first formalization of hindsight theory in a foundational program logic. The usefulness of the developed theory extends beyond our lock-free template algorithms. In fact, we demonstrate that it can help to reduce the proof effort compared to alternative proof techniques in Iris. To this end, we reverify the multicopy template algorithms of [35] using our formalization of hindsight as opposed to our previous tailor-made proof argument for dealing with future-dependent linearization points. The new approach reduces the proof effort by 53%.

To summarize, our contributions are (i) template algorithms for a wide variety of lock-free search structure algorithms, (ii) mechanized proofs of linearizability based on hindsight reasoning in Iris. The result is, to our knowledge, the first formal verification of fully-functional lock-free algorithms for skiplists of unbounded height.

2 The Skiplist Template Algorithm

A *skiplist* is a search structure over a totally ordered set of keys $\mathbb{K}$. We focus our discussion on skiplists that implement mutable sets rather than maps. The extension of the presented algorithms to mutable maps is straightforward. The data structure is composed of sorted lists at multiple levels, with the base list determining the actual contents of the structure, while higher level lists are used to speed up the search. An example is shown in Figure 1. A skiplist node contains a key and has a height, determining how many higher level lists this node is a part of. Each node has a next pointer for each of its levels. Two sentinel nodes signify the head (hd with key $-\infty$) and the tail (tl with key ∞) of the skiplist. Lock-free linked lists often use the technique of logical deletion by *marking* a node before it is physically unlinked from the list. This involves storing a mark bit together with the next pointer, so as to allow reading and updating them together in a single (logically) atomic step. Lock-free skiplist implementations also use this technique. Since a skiplist node can be part of multiple lists, it has one mark bit per level.

The traversal for a key not only goes left to right as usual, but also top to bottom. The red line in Figure 1 depicts a traversal searching for key 42. The traversal begins at the highest level of the head node. At each non-base level, the traversal continues till it reaches a node with a key greater than or equal to the search key. Thereafter, the traversal drops down a level, and continues at the lower levels until it terminates on the bottom level at the first node whose key is greater than or equal to the search key.

The traversals in a concurrent skiplist perform *maintenance* in the form of physically unlinking encountered marked nodes. In Figure 1, node n_5 has been unlinked at level 2, thus the traversal does not visit it at that level. Operations that mark and change the next pointers at the higher levels do not affect the actual contents of the structure. We therefore consider them to be part of the maintenance.

Many variants of lock-free skiplist algorithms have been proposed in the literature and implemented in practice. These variants differ in (i) their node implementations, (ii) the styles of maintenance operations and/or (iii) the orders in which they perform maintenance operations with regard to other operations.

For example, node implementations in low-level languages often use bit-stealing [11] (or an equivalent of Java's `AtomicMarkableReference`) so that both the next pointer and mark bit can be atomically read or updated. Other implementations use more complex solutions. For instance, the skiplists in [9] use nodes with back links to reduce traversal restarts due to marked nodes. Java's `ConcurrentSkipListMap` [34] implements each node as a list of simpler nodes, one per level. The higher level nodes have both right pointers and down pointers, while the base nodes only have right pointers. Java's implementation also uses *marker nodes* for marking, instead of bit-stealing.

In terms of style of maintenance, the traversal in the Michael Set [32] and Herlihy-Shavit lock-free skiplist [11, § 14] unlinks one marked node at a time. By contrast, the traversal in the Harris List [10] unlinks the entire sequence of marked nodes in one shot with a single CAS operation. The variants also differ in the order of marking of a node at higher levels. In the Herlihy-Shavit skiplist, the marking of a node goes from top level to the bottom level. This differs from skiplists in [34] and [9], whose marking goes from bottom to top.

Despite the differences in the skiplist algorithms described above (and others to be invented in the future), the bulk of their correctness reasoning remains the same. A goal of this paper is to show how to exploit that fact.

Template algorithm. Our template algorithm for skiplists abstracts away from node-level implementation details and the way in which traversals perform maintenance. As we shall see, the particular details regarding how the data is stored internal to the node does not affect the correctness of the core operations - `search`, `insert` and `delete`. Nor is the correctness affected by whether the traversal unlinks one marked node at a time or an entire sequence of marked nodes. We also show that the order in which maintenance operations are performed on the higher levels of the list does not matter for correctness. In summary, the template algorithm we present abstracts from: (i) node-level details; (ii) the style of unlinking marked nodes and (iii) the order of maintenance operations on higher levels.

The template algorithm is assumed to be operating on a set of nodes N that contains the two sentinel nodes head hd and tail tl. Let the maximum allowed height of a skiplist node be L (> 1). Each node n is associated with (i) its key $\mathsf{key}(n) \in \mathbb{K} = \mathbb{N} \cup \{-\infty, \infty\}$, (ii) its height $\mathsf{height}(n) \in [1, L)$, (iii) the next pointers $\mathsf{next}(n, i) \in N$ for each i from 0 to $\mathsf{height}(n) - 1$, and (iv) its mark bits per level $\mathsf{mark}(n, i) \in \{\mathit{true}, \mathit{false}\}$ for each i from 0 to $\mathsf{height}(n) - 1$. When discussing $\mathsf{next}(n, i)$ or $\mathsf{mark}(n, i)$, we implicitly assume that i lies between 0 and $\mathsf{height}(n) - 1$. We sometimes say a node n is unmarked to mean that it is unmarked at the base level, i.e., $\mathsf{mark}(n, 0) = \mathit{false}$. The structural invariant maintains the following facts: $\mathsf{key}(hd) = -\infty$, $\mathsf{key}(tl) = \infty$, $\mathsf{height}(hd) = \mathsf{height}(tl) = \mathsf{L}$, $\mathsf{next}(tl, i) = tl$ for all i, $\mathsf{next}(hd, \mathsf{L} - 1) = tl$, $\mathsf{mark}(hd, i) = \mathsf{mark}(tl, i) = \mathit{false}$ for all i.

The core operations of the skiplist template are expressed using *helper functions* such as `findNext` and `markNode` that abstract from the details of the node implementation. We describe the behavior of these helper functions as and when we encounter them. The template is instantiated by implementing these functions. The helper functions are assumed to be *logically atomic*, i.e., appear to take effect in a single step during its execution.

Figure 2 shows the core operations of the skiplist template algorithm. (We omit the code for the data structure initialization as it is straightforward.) All three operations begin by allocating two arrays ps and cs via `allocArr`, each of size L and values initialized to hd and tl respectively. These arrays are then populated by the `traverse` operation as it computes the predecessor-successor pair for operation key k at each level. Intuitively, these pairs indicate where k would be inserted at each level. The template algorithm here abstracts away from the concrete `traverse` implementation. We later consider two implementations of `traverse` that differ in the way that maintenance is performed, as discussed earlier.

As far as the core operations are concerned, they rely on `traverse` to satisfy the following specification. First, it returns a triple (p, c, res) where p and c are nodes and res a Boolean such that $p = ps[0]$, $c = cs[0]$ and res is true iff k is contained in c. Second, the node c must have been unmarked at some point during the traversal; and third, for each $0 \leqslant i < \mathsf{L}$, the traversal observes that $\mathsf{key}(ps[i]) < k \leqslant \mathsf{key}(cs[i])$.

Let us now describe the core operations, starting with the `search` operation. The `search` operation simply invokes the `traverse` function, whose result establishes whether k was in the structure. The `delete` operation starts similarly by invoking `traverse` and checking if the key is present in the structure. If it is, then `delete` invokes the maintenance operation `maintainanceOp_del`, which attempts to mark c at the higher levels (i.e. all levels except 0). We provide the implementation of `maintainanceOp_del` in a moment. Once `maintainanceOp_del` terminates, `delete` finally attempts to mark c via `markNode` at the base level. If marking succeeds, it terminates by invoking `traverse` (which performs the task of physically unlinking marked nodes at all levels) and returning *true*. Otherwise, a concurrent thread must have already marked c, in which case `delete` returns *false*.

The `insert` operation also begins with `traverse`. If the traversal returns *true*, then the key must already have been present. Hence, `insert` returns *false* in this case. Otherwise, a

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false

18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

Figure 2 The template algorithm for lock-free skiplists. The template can be instantiated by providing implementations of `traverse` and the helper functions `markNode`, `createNode` and `changeNext`. The `markNode` i c attempts to mark node c at level i atomically, and fails if c has been marked already. `createNode` k h cs creates a new node e of height h containing k, and whose next pointers are set to nodes in array cs. Finally, `changeNext` i p c cn is a CAS operation attempting to change the next pointer of p from c to cn. `changeNext` i p c cn succeeds only if $\mathsf{mark}(p, i) = \mathit{false}$ and $\mathsf{next}(p, i) = c$. Other functions used here include `randomNum` to generate a random number and maintenance operations associated with `insert` and `delete`. `maintainanceOp_del` marks node c at the higher levels, while `maintainanceOp_ins` inserts a new node e at the higher levels.

new node e is created using `createNode`. The node's height is determined randomly using `randomNum`, which generates a random number h such that $0 < h < L$. After creating a new node, the algorithm attempts to insert it into the list by calling `changeNext` at the base level (line 27). If the attempt succeeds, `insert` proceeds by invoking the maintenance operation `maintainanceOp_ins`, which also inserts the new node into the list at all higher levels. The `insert` then returns with *true*. If the `changeNext` operation fails, then the entire operation is restarted.

We now describe the maintenance operations for `insert` and `delete`, shown in Figure 3. The maintenance operations here differ from those in traditional skiplist implementations in regards to the order in which maintenance is performed at higher levels. In traditional implementations, the marking of a node goes from top to bottom, while insertion of a new node goes from bottom to top. The skiplist template presented here makes sure that the base level gets marked at the end and the insertion first happens at the base level, but it imposes no order on how it proceeds at higher levels. That is, when marking a node, a `delete` thread could for instance first mark odd levels, then even levels and finally the base level 0. The maintenance operations in the skiplist template captures all such permutations. As our proof shows later, the order of maintenance at higher levels has no bearing on the correctness of the algorithm.

The `maintainanceOp_del` marks node c from levels 1 to $\mathsf{height}(c)$. It begins by reading the height of c as h, and generating a permutation of $[1 \ldots (h-1)]$ stored in array pm via the `permute` function. The `maintainanceOp_del_rec` then recursively marks c in the order prescribed by pm. Note that the maintenance continues regardless of whether `markNode` succeeds or fails, because c will be marked at the end regardless.

The `maintainanceOp_ins` begins in the same way by reading the height, generating the permutation and invoking `maintainanceOp_ins_rec`. The `maintainanceOp_ins_rec` first collects the predecessor-successor pair at the current level from arrays ps and cs,

```
let maintainanceOp_del_rec i h pm c =
  if i < h-1 then
    let idx = pm[i] in
    markNode idx c;
    maintainanceOp_del_rec (i+1) h pm c
  else
    ()

let maintainanceOp_del c =
  let h = getHeight c in
  let pm = permute h in
  maintainanceOp_del 0 h pm c
```

```
let maintainanceOp_ins_rec i h pm ps cs e =
  if i < h-1 then
    let idx = pm[i] in
    let p = ps[idx] in
    let c = cs[idx] in
    match changeNext idx p c e with
    | Success ->
      maintainanceOp_ins_rec (i+1) h pm ps cs e
    | Failure ->
      traverse ps cs k;
      maintainanceOp_ins_rec i h pm ps cs e
  else
    ()

let maintainanceOp_ins k ps cs e =
  let h = getHeight e in
  let pm = permute h in
  maintainanceOp_ins 0 h pm ps cs e
```

Figure 3 The maintenance operations for the skiplist. The `getHeight` c helper function returns $\mathsf{height}(c)$. The `permute` function generates a permutation of $[1 \ldots (h-1)]$ as an array.

respectively. Then it tries to insert the new node e using `changeNext` on predecessor node p. If `changeNext` succeeds, then the recursive operation continues. Otherwise, it recomputes the predecessor-successor pairs using `traverse`. After the recomputation, the insertion is retried at the same level.

We can now finally turn to the implementations of `traverse`. We consider two implementations that differ in their treatment of marked nodes. The *eager* traversal attempts to unlink every marked node it encounters, while the *lazy* traversal simply walks over the marked nodes till it reaches an unmarked node. The traversal then attempts to unlink the entire marked segment at once. The two implementations are similar in other aspects, so we discuss only the eager traversal in detail here.

The eager traversal is shown in Figure 4. The `traverse` function is implemented using mutually-recursive functions `eager_rec` and `eager_i`[1]. The function `eager_rec` populates the arrays ps and cs with the predecessor-successor pair at level i computed by `eager_i`. The `eager_i` performs a traversal at level i by first reading the mark bit and next pointer of c using `findNext`. If c is found to be marked, then `eager_i` attempts to physically unlink the node using `changeNext`. In the case that `changeNext` fails (because either p is marked or it does not point to c anymore), `eager_i` simply restarts the `traverse` function. In the case of `Success` of `changeNext`, the traversal continues. If c is unmarked, then `traverse_i` proceeds by comparing k to $\mathsf{key}(c)$. For $\mathsf{key}(c) < k$, the traversal continues with c and cn. Otherwise, `eager_i` ends at c, returning (p, c, true) if $\mathsf{key}(c) = k$ and (p, c, false) otherwise. As mentioned before, `eager_i` attempts to unlink immediately whenever a marked node is encountered.

[1] For ease of exposition, the implementation of the eager traversal shown in Figure 4 differs slightly from the version we have verified in Iris. The Iris version uses option return types instead of mutually-recursive functions in order to obtain a more modular proof of the eager traversal. We use the mutually recursive implementation here for clarity of exposition.

```
let eager_i i k p c =
  match findNext i c with
  | cn, true ->
    match changeNext i p c cn with
    | Success -> eager_i i k p cn
    | Failure -> traverse ps cs k
  | cn, false ->
    let kc = getKey c in
    if kc < k then
      eager_i i k c cn
    else
      let res = (kc = k ? true : false) in
      (p, c, res)

let eager_rec i ps cs k =
  let p = ps[i+1] in
  let c, _ = findNext i p in
  let p', c', res = eager_i i k p c in
  ps[i] <- p';
  cs[i] <- c';
  if i = 0 then
    (p', c', res)
  else
  eager_rec (i-1) ps cs k

let traverse ps cs k =
  eager_rec (L - 2) ps cs k
```

Figure 4 The eager traversal for the skiplist template. `findNext` $i\,k\,c$ returns a pair $(\mathsf{next}(c,i), \mathsf{mark}(c,i))$. The `getKey` c helper function returns $\mathsf{key}(c)$.

3 Proof Intuition

Our goal is to show that the skiplist template is linearizable. That is, we must prove that each of the core operations take effect in a single atomic step during its execution, the *linearization point*, and satisfies the sequential specification shown in Figure 5. For the skiplist template, we define the abstract state $C(N)$ to be the union of the *logical contents* $C(n)$ of all nodes in N, where $C(n) \coloneqq (\mathsf{mark}(n,0)\ ?\ \emptyset : \{\mathsf{key}(n)\})$. In other words, the abstract state of the structure is a collection of keys contained in unmarked nodes at the base level. There are existing techniques from the literature that help us analyze the skiplist template. The two main techniques that we rely on are the *Edgeset Framework* [38] and *Hindsight Reasoning* [33, 23, 7, 8, 27, 28]. We begin by giving a brief overview of the two techniques, proceeded by the analysis of the skiplist template using these techniques.

$$\Psi_{\mathtt{op}}(k, C, C', res) \coloneqq \begin{cases} C' = C \land (res \Leftrightarrow k \in C) & \mathtt{op} = \mathtt{search} \\ C' = C \cup \{k\} \land (res \Leftrightarrow k \notin C) & \mathtt{op} = \mathtt{insert} \\ C' = C \setminus \{k\} \land (res \Leftrightarrow k \in C) & \mathtt{op} = \mathtt{delete} \end{cases}$$

Figure 5 Sequential specification of a search structure. k refers to the operation key, C and C' to the abstract state before and after operation `op`, respectively, and res is the return value of `op`.

3.1 The Edgeset Framework

The Edgeset Framework provides a common terminology to capture how search operations navigate in a variety of search structures. We view each search structure as a mathematical graph whose edges are associated with an *edgeset*, a label that is a set of keys. We denote the edgeset from n to n' by $\mathsf{es}(n, n')$, and $k \in \mathsf{es}(n, n')$ signifies that a search for key k will proceed from node n to n'. In the context of the skiplist template, we define the edgeset leaving n to be all values greater than the key in n if n is unmarked. If node n is marked, then the edgeset leaving n is the entire keyspace. Formally: $\mathsf{es}(n, n') \coloneqq (n' = \mathsf{next}(n, 0) \land \mathsf{mark}(n, 0) = \mathit{false}\ ?\ (\mathsf{key}(n), \infty) : \mathbb{K})$. Note that, our definition of edgesets in the skiplist template depends only on the base list, and not on higher level mark bits and next pointers.

A notion defined in terms of edgesets is the *inset* of a node, denoted by $\mathsf{inset}(n)$, signifying a set of keys for which a search will arrive at n. In order to understand the concept of inset intuitively, consider Figure 6. The inset of node n_4 is $(2, \infty)$, because, for all keys greater than 2, the search will enter n_4. We say node n_1 is the *logical predecessor* of n_4 if it is the first unmarked predecessor of n_4. The inset of the root is $\mathbb{K}$ and the inset of n is the intersection of $\mathbb{K}$ with the edgesets of all nodes between the root and n. For sorted linked lists in general, a more local notion gives the same result: the inset of an unmarked node n is $(\mathsf{key}(n'), \infty)$, where n' is the logical predecessor of n.

In contrast to inset, we define the *outset* as the union of all its outgoing edgesets: $\mathsf{outset}(n) := \bigcup_{n' \in N} \mathsf{es}(n, n')$.

We can now define the *keyset* of a node n as $\mathsf{keyset}(n) := \mathsf{inset}(n) \setminus \mathsf{outset}(n)$, i.e. intuitively, the set of keys for which a search enters n but never leaves. The importance of keysets is that if k is in $\mathsf{keyset}(n)$, then k is either in the contents of n or is nowhere in the structure. In Figure 6, the keyset of n_4 is $(2, 9]$ and in general, the keyset of an unmarked node n is $(\mathsf{keyset}(n'), \mathsf{key}(n)]$ where n' is its logical predecessor. The keyset of a marked node is $\emptyset$ because its outset is the set of all keys $\mathbb{K}$.

The technical definition of inset relies on the global data structure graph, defined as a solution to the following fixpoint equation

$$\forall n \in N.\ \mathsf{inset}(n) = \mathit{in}(n) \cup \bigcup_{n' \in N} \mathsf{es}(n', n) \cap \mathsf{inset}(n')$$

where $\mathit{in}(n) := (n = \mathit{hd}\ ?\ \mathbb{K} : \emptyset)$. Thus, the inset is a global quantity and hence difficult to reason about. Fortunately, this is where the Flow Framework [21, 22, 29] comes in handy. It allows us to reason about quantities that can be expressed as a solution to a fixpoint equation (like inset) in a local manner by attaching *flow* values to the node. The framework then provides tools to track changes to the flow values that are induced by changes to the underlying graph. Our approach to encoding keysets in Iris using the Flow Framework is borrowed from [19]. We defer further details on this matter to the later sections.

As mentioned above, $\mathsf{keyset}(n)$ intuitively is the set of all keys that n is responsible for. Consider Figure 6 again, a thread executing `search(6)` without any interference will reach node n_4 and terminate, concluding that 6 is not present in the structure. In this sense, we say n_4 is responsible for key 6 and therefore 6 is part of n_4's keyset. The keysets of all nodes partition the set of all keys and provide the crucial *Keyset Property*:

$$\forall\ n \in N, k \in \mathsf{K}.\ k \in \mathsf{keyset}(n) \Rightarrow (k \in C(N) \Leftrightarrow k \in C(n)) \qquad (\mathsf{KeysetPr})$$

This property enables one to lift a proof of the specification at the node level to a proof of the sequential specification $\Psi_{\mathtt{op}}$. A particular situation where (KeysetPr) proves indispensable is when `search` fails to find the search key. Note that `search` observes only the nodes it visited, and hence has only a partial view of the structure. When `search` fails to find the key, the proof has to reconcile this partial view of the structure with the global view. In essence, if a concurrent invocation of `search` on key k fails to find the key, can we conclude that there was a point in time during its execution when k was in fact not present in the structure? Here, the property (KeysetPr) helps us reconcile facts gathered by search with the global state of the structure. Specifically, if `search` can determine a node n such that $k \in \mathsf{keyset}(n)$ and $k \notin C(n)$, then we can immediately infer that k was not present in the structure at that point in time.

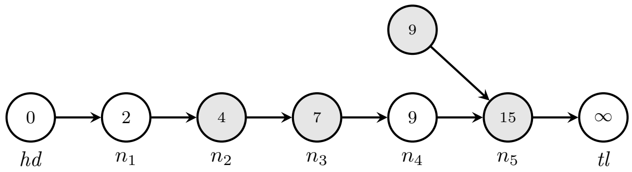

Figure 6 Possible state of the base list in the skiplist template. Nodes are labeled with the value of their `key` field. Edges indicate `next` pointers. Marked (logically deleted) nodes are shaded gray. $\mathsf{keyset}(hd) = \{0\}$, $\mathsf{keyset}(n_1) = (0, 2]$, $\mathsf{keyset}(n_4) = (2, 9]$ and $\mathsf{keyset}(tl) = (9, \infty)$. The keyset of a marked node is always $\emptyset$.

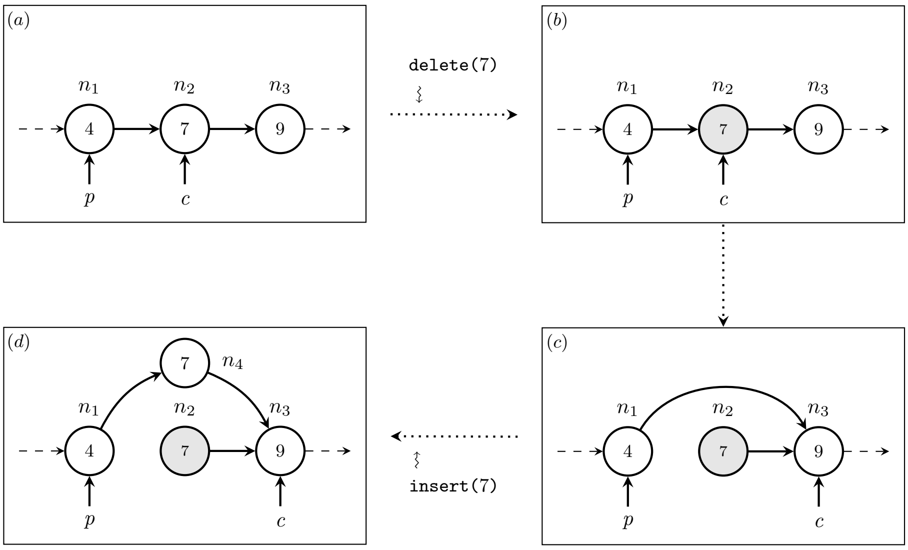

Figure 7 Possible states of `search(7)` on the base level in presence of interference from concurrent `delete(7) and insert(7)`.

3.2 Hindsight Reasoning

Lock-free structures often exhibit future-dependent linearization points. That is, the linearization point of an operation cannot be determined at any fixed moment, but only at the end of the execution, once any interference of other concurrent operations has been accounted for. To understand the interference issue, consider the `search` operation. Since, `search` returns the result of `traverse`, let us look at the eager traversal implementation. To simplify the explanation further, let us assume that the maximum height allowed for every non-sentinel node is one. Then, we can ignore the `eager_rec` function and focus on `eager_i` called at the base level.

Let there be a thread T executing `search(7)`. Concurrently, there is a thread T_d executing `delete(7)` and a thread T_i executing `insert(7)`. Figure 7 shows interesting scenarios that thread T might potentially observe. Box (a) captures the state of the structure at the beginning of the `eager_i` call processing n_2. Let **Scenario 1** be the situation when thread T faces no interference from T_d and T_i. Here, thread T finds the key 7 in n_2 and `eager_i` returns *true*. The point when `eager_i` finds n_2 to be unmarked becomes the linearization point for this scenario.

Now consider **Scenario 2** to be the situation where thread T_d marks n_2 before `eager_i` processes it, as shown in Box (b). Thread T will attempt to unlink n_2, and assuming no further interference, the unlink will result in the structure in Box (c). Thread T will process n_3 next, finding n_3 to be unmarked with key greater than 7, and will terminate with result *false*. So when is the linearization point in this scenario? It cannot be when T finds n_3 unmarked when processing it. Because there could be further interference from thread T_i which inserts key 7 in a new node as shown in Box (d). The new node could be added right before T reads the mark bit of n_3. Thus, when `eager_i` finds n_3 unmarked and returns *false*, key 7 could actually be present in the structure at that point in time.

The linearization point is actually the point in time shown in Box (c), i.e., right after n_2 is unlinked. However, thread T cannot confirm this when n_2 is unlinked because `eager_i` may not terminate at n_3 with *false* as the result. The reason is that by the time T processes n_3, it could get marked in a manner similar to n_2 in Box (b), resulting in the unlinking of n_3 and potentially a restart. That Box (c) is the linearization point is confirmed when T has found n_3 to be unmarked later. The structure maintains the invariant that once a node is marked, it remains marked. Using this invariant, an analysis of thread T's history concludes that n_3 must have been unmarked at the point when n_2 was unlinked. Once `eager_i` terminates at n_3 with *false*, an analysis can *establish in hindsight* that Box (c) indeed was the linearization point.

Hindsight reasoning as formalized in [27, 28] is designed to deal with situations like the `search` in Figure 7. It enables temporal reasoning about computations using a *past predicate* $\diamondsuit q$, which expresses that proposition q held true at some prior state in the computation (up to the current state). For instance, $\diamondsuit(\mathsf{next}(n_1, 0) = n_2)$ holds in Box (c) even though $\mathsf{next}(n_1, 0) = n_3$ at that point. The reason is that $\mathsf{next}(n_1, 0) = n_2$ was true at an earlier point in time, namely in Box (b). Note that the past operator $\diamondsuit$ abstracts away the exact time point when the predicate held true. Note also that a past predicate is not affected by concurrent interferences, as it merely records some fact about a past state.

There are two ways to establish a past predicate that are relevant for our proofs. The first is to establish the predicate in the current state directly. That is, $\diamondsuit q$ holds if q holds in the current state. As an example, we obtain $(\mathsf{next}(n_1, 0) = n_2)$ when `findNext` on n_1 returned n_2 in Box (a). Thus, for all subsequent states including Box (b) and (c), we get $\diamondsuit(\mathsf{next}(n_1, 0) = n_2)$. The second way to establish a past predicate is through the use of *temporal interpolation* [28]. That is, one proves a lemma of the form: if there existed a past state that satisfied property q and the current state satisfies r, then there must have existed an intermediate state that satisfied o. Such lemmas can then be applied, e.g., to prove that if thread T finds n_3 to be unmarked in Scenario 2, then it must have been unmarked when n_2 was unlinked in Box (c).

Equipped with the Edgeset Framework and hindsight reasoning, we are now ready to analyze the core operations of the skiplist template.

3.3 Proof Outline for Core Operations

We refer to a linearization point as *modifying* if the operation changes the abstract state of the data structure (like in the case of a succeeding `delete` and `insert`) and otherwise refer to it as *unmodifying* (like `search` and in the case of a failing `delete` or `insert`). The modifying linearization points of the skiplist template are easier to reason about because they are not future-dependent. For `delete`, the linearization point occurs when `markNode` succeeds, and similarly, for `insert` the linearization point occurs when the call to `changeNext` on line 27 succeeds. The proof strategy for unmodifying linearization points is to combine (KeysetPr) with the $\diamondsuit$ operator from hindsight reasoning. Let us expand on this proof strategy in detail and show why the skiplist template is linearizable.

We begin by describing the specification for `traverse` that is assumed for analyzing the core operations of the template. Then, we analyze each of the operations in detail. Finally, we show how the eager implementations of `traverse` satisfies the specification that was assumed in the beginning. Along the way, we introduce (as and when necessary) invariants maintained by the skiplist template that are crucial for proving linearizability.

Specification of `traverse`. The function `traverse` $ps\ cs\ k$ updates arrays ps and cs with predecessor-successor pairs for each level and returns a triple (p, c, res) that satisfies the following past predicate regarding node c: $\Diamond(k \in \mathsf{keyset}(c) \wedge (res \Leftrightarrow k \in C(c)))$. Recall that our definition of edgesets in Section 3.1 implies the following invariant:

Invariant 1 For all nodes n, if $\mathsf{mark}(n, 0)$ is set to *true* then $\mathsf{keyset}(n) = \emptyset$.

Using Invariant 1, we can establish that c is unmarked at the base level at the time point when $k \in \mathsf{keyset}(c)$ holds. Note that `traverse` may physically unlink marked nodes. However, this step does not change the abstract state of the structure. Hence, the specification for `traverse` involves no change of the abstract state.

We now consider each of the core operations in detail.

Proof of `search`. Function `search` returns res out of the triple (p, c, res) returned by `traverse`. The specification of `traverse` says $res \Leftrightarrow k \in C(c)$ at some point, say t, during its execution. The specification additionally guarantees $k \in \mathsf{keyset}(c)$ at time t. These two facts, combined with the (KeysetPr) at time point t, allow us to immediately infer that res is true iff k was in the structure at that point. Hence, we can establish that $(res \Leftrightarrow k \in C(c))$ was true at some point during the execution of `search`.

Proof of `delete`. We analyze `delete` by case analysis on the value res returned by `traverse`. If res is *false*, then again we can establish that k was not in the structure at some point during `traverse`'s execution by the same reasoning used in the proof of `search`. So let us consider the case that res is *true*. By the specification of `traverse`, we can establish a time point when c was unmarked and contained k. The `delete` operation then calls `maintainanceOp_del` which marks c at all the higher levels. Finally, the `markNode` on Line 15 attempts to mark c at the base level. If `markNode` succeeds, then this step becomes the linearization point of `delete` and k can be considered to be deleted from the structure. But if `markNode` fails, then we gain the knowledge that $\mathsf{mark}(c, 0) = \mathit{true}$. Hindsight reasoning allows us to infer that c was marked at the base level by a concurrent thread between the end of `traverse` and the invocation of `markNode`. The point right after c was marked by a concurrent thread becomes the linearization point of `delete` in this case, as we can determine that k was not present in the structure at that point.

This hindsight reasoning relies on two facts: first, the key of a node never changes and second, once a mark bit is set to *true* by a successful `markNode` operation (at line 15 in `delete` or line 4 in `maintainanceOp_del`), no other operation will set it back to *false*. In fact, these two facts are invariant for the skiplist template:

Invariant 2 For all nodes n and level i, once $\mathsf{mark}(n, i)$ is set to *true*, it remains *true*.

Invariant 3 For all nodes n, $\mathsf{key}(n)$ remains constant.

Proof of `insert`. Similar to `delete`, we begin by case analysis on res returned by `traverse`. If res is true, then we can establish that k was already present in the structure at some point. Otherwise, res is *false* and `insert` creates a new node e with key k. Using `changeNext`, an attempt is made to insert node e between nodes p and c. If the attempt succeeds, then k is now part of the structure and this becomes the linearization point. The following `maintainanceOp_ins` operation does not change the abstract state of the structure, and thus, has no effect in terms of linearizability. If the `changeNext` fails, then `insert` simply restarts.

As is evident with the proof outline for the core operations, the specification assumed for `traverse` plays a critical role in case the operation exhibits an unmodifying linearization point. Let us now turn to `traverse` and show how its specification can be proved. We analyze the eager traversal in detail in the following section. The proof argument for the lazy version is similar.

3.4 Proof Outline for Eager Traversal

As stated earlier, `traverse` returns (p, c, res) such that $\diamond(k \in \mathsf{keyset}(c) \wedge (res \Leftrightarrow k \in C(c)))$. Since the returned triple is the result of a call to `eager_i` at the base level, let us begin by analyzing the behavior of this call.

In the sequential setting, the traversal in a search structure maintains the invariant that the search key is always in the inset of the current node. This invariant holds by the design of the Edgeset Framework. Unfortunately, this invariant no longer holds for the skiplist template in the concurrent setting as evidenced by Box (c) in Figure 7. However, we argue first that `eager_i` does maintain the invariant that the search key was in the inset of the current node c between the start of the traversal and the point at which the `eager_i` accesses c. We call this the *inset in hindsight* invariant.

We prove this invariant inductively. We make use of the following locally maintained invariants: (i) At all times, there is one list, denoted the *reachable list*, from the head node that includes all unmarked and some marked nodes. (This list is characterized by the set of nodes with non-empty inset, see Figure 6 for intuition). (ii) The keys in the reachable list are sorted. A consequence of these two invariants is that if a node n is in the reachable list (whether n is marked or not) and has a key less than k, then k is in the inset of n.

To prove that inset in hindsight is an invariant, we have to show that (a) it is an invariant when `eager_i` takes a step (Line 2) when traversing the base level, and (b) that we can establish inset in hindsight when `eager_rec` initiates `eager_i` (Line 17) at the base level.

To show (a), observe that if a node n becomes unlinked from the reachable list, then it will never again be part of the reachable list. Hence, if n is not in the reachable list when `eager_i` begins executing at the base list, then `eager_i` will never visit n. The contrapositive of this statement allows us to say that if `eager_i` reaches some node c, then it must have been part of the reachable list at some point during the execution of `eager_i`. Additionally, `eager_i` proceeds to the node following c only when $\mathsf{key}(c) < k$. With the help of invariants (i) and (ii) above, we can thus establish that k was in the inset of n at some point.

To show (b), we must do a case analysis on whether node p (Line 16) is marked. If it is unmarked, then it is straightforward to establish that k is in the inset of c currently. However, if p is marked, then we require temporal interpolation based on the following invariant:

Invariant 4 For all nodes n and level i, once $\mathsf{mark}(n, i)$ is set to *true*, $\mathsf{next}(n, i)$ does not change.

This invariant tells us that if p was known to be unmarked in the past, and it is marked currently, then p must have been pointing to c right before it got marked. At that point in time, we can establish that k must have been in the inset of c.

This completes the inductive proof that inset in hindsight is indeed an invariant maintained by the traversal. The inset in hindsight invariant is sufficient to prove the `traverse` specification by the following simple argument. If the traverse encounters k in an unmarked node n, then `traverse` will return *true* as it should. If, by contrast, `traverse` encounters an unmarked node n such that $\mathsf{key}(n) > k$, then by the inset in hindsight invariant, k must have been in the inset of n at some point t in the past and k cannot be in the outset of n (because $\mathsf{key}(n) > k$ and n is unmarked), so therefore k must have been in the keyset of n at time t.

4 Hindsight Reasoning in Iris

Linearizability in Iris is defined via *(logically) atomic triples* [5, 17]. Intuitively, an atomic triple $\langle x.\, P \rangle\ e\ \langle v.\, Q \rangle$ says that at some point during the execution of e, the resources described by the precondition P will be updated to satisfy the postcondition Q for return value v in one atomic step. The variable x can be thought of as the abstract state of the data structure before the update at the linearization point.

Linearizability of a search structure operation op can be expressed by an atomic triple of the form

$$\boxed{\mathsf{Inv}(r)} \mathrel{-\!\!*} \langle C.\, \mathsf{CSS}(r, C) \rangle\ \mathsf{op}\, r\, k\ \langle res.\, \exists\, C'.\, \mathsf{CSS}(r, C') * \Psi_{\mathsf{op}}(k, C, C', res) \rangle. \quad (\mathsf{ClientSpec})$$

Here, r is the pointer to the head of the data structure. The predicate $\mathsf{CSS}(r, C)$ is the *representation predicate* that relates the head pointer with the contents C of the structure. The predicate $\mathsf{Inv}(r)$ is the shared data structure invariant. It can be thought of as a thread-local precondition of the atomic triple, which we express using separating implication. The invariant ties $\mathsf{CSS}(r, C)$ to the data structure's physical representation and may contain other resources necessary for proving the atomic triple. The predicate $\Psi_{\mathsf{op}}(k, C, C', res)$ captures the sequential specification of the structure. The specification essentially says there is a single atomic step in op where the abstract state changes from C to C' according to the sequential specification $\Psi_{\mathsf{op}}(k, C, C', res)$ (Figure 5). This step is op's linearization point. We call (ClientSpec) the *client-level* atomic specification for the data structure under proof.

Proving atomic triples. The proof of establishing an atomic triple involves a *linearizability obligation* that must be discharged directly at the linearization point. However, it can be challenging to determine the linearization point precisely and to discharge the linearizability obligation exactly at that point. When the program execution reaches a potential linearization point that depends on future interferences by other threads, then the proof will fail if it is unable to determine whether the linearizability obligation should be discharged now or later. In Iris, this challenge is overcome using *prophecy variables* [16], which enable the proof to reason about the remainder of the computation that has not yet been executed.

Another challenge is that the linearization point of an operation may be an atomic step of another operation that is executed by a different thread (like in Scenario 2 discussed in Section 3.2). Data structures that demonstrate such behavior are said to deploy *helping*. This behavior complicates thread modular reasoning. The conventional solution to this challenge in Iris is to use a *helping protocol* [16, 35, 14]. The helping protocol is specified as part of the shared data structure invariant and consists of a registry that tracks which threads are expected to be linearized by other threads as well as conditional logic that governs the correct transfer and discharge of the associated linearizability obligations.

Both the use of prophecy variables and the helping protocol need to be tailored to the specific data structure at hand, which adds considerable overhead to the proof. To reduce this overhead, we present an alternative proof method that enables linearizability proofs based on hindsight arguments in Iris. Rather than identifying the linearization point precisely, the proof can establish linearizability in hindsight using temporal interpolation in the style of the intuitive proof argument for the skiplist template presented in Section 3.2.

Hindsight specification. Our proof method offers an intermediate specification, a Hoare triple specification, which in essence expresses that linearizability has been established in hindsight. In our Iris formalization, we show that any data structure whose operations satisfy the hindsight specification also satisfy the client-level atomic specification. This proof relates the two specifications via prophecy variables and a helping protocol. However, the helping protocol is data structure agnostic, making our proof method applicable to a broad class of structures exhibiting future-dependent unmodifying linearization points.

From the perspective of a proof author using our method to prove linearizability of some structure, one has to only establish the hindsight specification to obtain the proof of the client-level atomic specification. To this end, our method provides further guidance to the proof author.

In order to use hindsight reasoning, one has to have the history of computation at hand. Here, we offer a shared state invariant with a mechanism to store the history. The shared state invariant has three main components: a mechanism to store the history, the helping protocol, and finally, an abstract predicate that can be instantiated with invariants specific to the structure at hand. The first two components are data structure agnostic. The proof author only needs to specify the data structure-specific invariant and what information about the data structure state should be tracked by the history.

In the rest of this section, we discuss our method in detail. We begin with the hindsight specification, followed by a discussion of the shared state invariant and how to use it.

4.1 Linearizability in Hindsight

We motivate the hindsight specification using the challenges we face when proving the client-level atomic specification for the `delete` operation of the skiplist template. Let us recall the intuitive proof argument for `delete` from Section 3.3. As per the observation regarding the modifying and unmodifying linearization points, a `delete` thread with modifying linearization point can fulfill the obligation at the point when the structure is modified. However, a `delete` thread with an unmodifying linearization point requires helping.

Prophecy reasoning. An important detail of our proof method is how it determines whether a thread requires helping. In the following, we refer to the operation that a thread performs at its linearization point as its *decisive operation*. In `delete`, the traversal observes node c to be unmarked at some point during execution. In the case where c is marked by the time that the thread calls its decisive operation `markNode` (in Line 15), the thread requires helping from the thread that marks c.

In order to determine in advance whether a thread requires helping, our proof method attaches a prophecy to each thread. A prophecy in Iris can predict a sequence of values and is treated as a resource that can be owned by a thread. Ownership of a prophecy p is captured by the predicate $\mathsf{Proph}(p, pvs)$, where pvs is the list of predicted values. The predicate signifies the right to resolve p when the thread makes a physical step that produces some result value v. The resolution of p establishes equality between v and the head of the list pvs (i.e., the next value predicted by p). The resolution step yields the updated predicate $\mathsf{Proph}(p, pvs')$ where pvs' is the tail of pvs. This mechanism enables the proof to do a case analysis on the predicted values pvs before these values have been observed in the program execution[2].

[2] For further details on prophecies in Iris, we refer to [16].

The prophecy attached to a thread predicts the results of the thread's decisive operation. In case of `delete`, the decisive operation is the call to `markNode` in the base list, while for `insert`, it is the call to `changeNext` in the base list. Note that a thread may restart if its decisive operation fails (like in the case of `insert`). Therefore, the prophecy needs to predict a sequence of result values, one for each attempted call to the thread's decisive operation.

For the purpose of this discussion, we assume that the prophecy predicts a sequence of `Success` or `Failure` values. If the sequence contains a `Success` value, then the decisive operation will succeed and the thread will modify the structure. Otherwise, the thread's linearization point is unmodifying. Let predicate $\mathsf{Upd}(pvs)$ hold when pvs contains at least one `Success` value.

The proof author only needs to identify the decisive operations that potentially change the abstract state of the structure (like `markNode` as discussed above) by resolving the prophecy around these decisive calls.

Hindsight specification. Before we can present the hindsight specification, we need to provide necessary details regarding the atomic triples in Iris. An atomic triple $\langle x.\, P \rangle\ e\ \langle v.\, Q \rangle$ is defined in terms of standard Hoare triples of the form $\forall\, \Phi.\ \{\mathsf{AU}_{x.P,Q}(\Phi)\}\ e\ \{v.\, \Phi(v)\}$. The predicate $\mathsf{AU}_{x.P,Q}(\Phi)$ is the *atomic update token* and represents the linearizability obligation of the atomic triple. At the beginning of each atomic step that the thread takes up to its linearization point, the token offers the resources in P and the token itself transforms into a choice. That is, at the end of the atomic step, the prover has to chose to either *commit* the linearization or *abort*. When committing, the prover has to show that the thread's atomic step transforms the resources in P to those in Q, receiving $\Phi(v)$ from the update token in return, which serves as the receipt of linearization of the atomic triple. In case of an abort, the prover needs to show that the thread's atomic step reestablishes P.

We also need to introduce two more auxiliary predicates:

- $\mathsf{Thread}(tid, t_0)$: this predicate is used to *register* the thread with identifier tid in the shared invariant. The argument t_0 denotes the time when thread tid began its execution.
- $\mathsf{PastLin}(\mathsf{op}, k, res, t_0)$: this predicate holds if there was a past state in the history between time t_0 and the point when this predicate is evaluated for which the sequential specification Ψ_{op} held with result res. It essentially captures whether the sequential specification was true for any point after time t_0.

We now have all the ingredients to present the hindsight specification:

$$\begin{array}{l}\forall\ tid\ t_0\ pvs.\ \boxed{\mathsf{Inv}(r)} -\!\!*\ \mathsf{Thread}(tid, t_0) -\!\!* \\ \{\mathsf{Proph}(p, pvs)\ *\ (\mathsf{Upd}(pvs) -\!\!*\ \mathsf{AU}_{\mathsf{op}}(\Phi))\}\ \mathsf{op}\ r\ k \\ \left\{\begin{array}{r} res.\ \exists pvs'.\ \mathsf{Proph}(p, pvs')\ *\ pvs = (\_\ @\ pvs') \\ *\ (\mathsf{Upd}(pvs) -\!\!*\ \Phi(res)) \\ *\ (\neg\mathsf{Upd}(pvs) -\!\!*\ \mathsf{PastLin}(\mathsf{op}, k, res, t_0)) \end{array}\right\}\end{array} \qquad (\mathsf{HindSpec})$$

We explain it piece by piece. The local precondition $\mathsf{Thread}(tid, t_0)$ ties the thread to its identifier tid and provides knowledge that tid begins executing at time t_0. The Hoare triple can be best understood by observing how prophecy resources are allowed to change (highlighted in brown) and what are the obligations when $\mathsf{Upd}(pvs)$ holds (in teal) versus when it does not hold (in magenta). Let us look at each of these in detail. First, the prophecy resource $\mathsf{Proph}(p, pvs)$ in the precondition changes to $\mathsf{Proph}(p, pvs')$ in the postcondition where pvs' is a suffix of pvs. It basically says that operation op is allowed to resolve the prophecy p as many times as it needs and then return the remaining resource at the end.

Now let us consider the case when $\mathsf{Upd}(pvs)$ holds. The precondition here provides the atomic update token $\mathsf{AU}_{\mathtt{op}}(\Phi)$ to `op`, expecting the receipt of linearization $\Phi(res)$ in return. Thus, the responsibility of linearization is delegated to `op` when $\mathsf{Upd}(pvs)$ holds. We can gain better insight by relating this situation to the `delete` operation from the skiplist template as before. This case corresponds to when `markNode` (from line 15) succeeds as $\mathsf{Upd}(pvs)$ holds here. The point when `markNode` succeeds becomes the linearization point and so the thread does not require help from other threads to linearize. The hindsight specification simply asks for the receipt from linearization $\Phi(res)$ at the end.

Finally, let us consider the case when $\mathsf{Upd}(pvs)$ does not hold. The precondition provides no additional resources here, while the postcondition requires the predicate $\mathsf{PastLin}(\mathtt{op}, k, res, t_0)$. In simple terms, this means that if $\mathsf{Upd}(pvs)$ is not true, i.e., the prophecy says the thread is not going to modify the structure, then the hindsight specification allows exhibiting a past state from history when the sequential specification was true. Relating again to `delete`, if the `markNode` fails, then the thread can look at the history of the structure and exhibit precisely the point when the decisive node got marked.

The proof argument for establishing the hindsight specification is significantly simpler than if one were to attempt a direct proof of the client-level atomic specification. In particular, the proof author does not need to reason about helping and atomic update tokens in last case discussed above. Instead, they only need to reason about the structure-specific history invariant.

Soundness of the hindsight specification. Our proof that relates the hindsight specification for `op` to the atomic triple specification involves a helping protocol. The details of the helping protocol and the soundness proof for the hindsight specification are similar to those of the proofs presented in [16, 35]. We therefore provide only a brief summary here. Additional details regarding the proof and the helping protocol can be found in [36].

Before `op` begins executing, the proof creates the prophecy resource $\mathsf{Proph}(p, pvs)$ assumed in the precondition of the hindsight specification. If the prophecy determines that the thread requires helping, then its client-level atomic triple is registered to a predicate which encodes the helping protocol as part of the shared state invariant $\mathsf{Inv}(r)$. The registered atomic triple serves as an obligation for the helping thread to commit the atomic triple. This obligation will be discharged by the appropriate concurrent operation determined by the `op`'s sequential specification $\Psi_{\mathtt{op}}$. The proof then uses the hindsight specification to conclude that it can collect the committed triple from the shared predicate. The committed triple serves as a receipt that the obligation to linearize has been fulfilled.

To govern the transfer of linearizability obligations and fulfillment receipts between threads via the shared invariant, the helping protocol tracks a *registry* of thread IDs with unmodifying linearization points that require helping from other concurrent threads. Each thread registered for helping is in either *pending* state or *done* state, depending on whether the thread has already been linearized. A thread registered for helping must be able to determine its current protocol state in order to be able to extract its committed atomic triple from the registry. For this purpose, the helping protocol includes a *linearization condition* that holds iff a registered thread *tid* has linearized (and is, hence, in *done* state).

From the point of view of a thread which *does* the helping, the linearization condition forces its proof to scan over the pool of uncommitted triples registered in the helping protocol and identify those that need to be linearized at its linearization point, changing their protocol state from *pending* to *done*. This step involves a proof obligation for the helping thread to show that the sequential specification of *tid*'s operation is indeed satisfied at the linearization point.

One crucial innovation in our helping protocol is that we have formulated a linearization condition that is parametric in the sequential specification of the data structure operations, making the soundness proof for the hindsight specification applicable to many structures at once. In particular, we deal with the aspect of scanning and updating the registry in the proof of the helping thread, the proof author simply invokes a lemma provided by our method at the identified linearization points. Therefore, the helping protocol mechanism remains fully opaque to the proof author.

4.2 Invariant for Hindsight Reasoning

Hindsight arguments involve reasoning about past program states. Our encoding therefore tracks information about past states using *computation histories.* We define computation histories as finite partial maps from *timestamps*, $\mathbb{N}$, to *snapshots*, $\mathbb{S}$. A snapshot describes an abstract view of a program state. It is a parameter of our method. For instance, a snapshot may capture the physical memory representation of the data structure under proof, while abstracting from the remainder of the program state. Another parameter is a function $|\cdot|$ that computes the abstract state of the data structure from a given snapshot.

$$
\begin{aligned}
\mathsf{Inv}(r) &\coloneqq \exists\, H\, T\, C.\ \overline{\mathsf{CSS}}(r, C) * |H(T)| = C \\
&\qquad * \mathsf{Hist}(H, T) * \mathsf{Inv}_{help}(H, T) * \mathsf{Inv}_{tpl}(r, H, T) \\
\mathsf{Inv}_{tpl}(r, H, T) &\coloneqq \mathsf{resources}(r, H(T)) \\
&\qquad * (\forall t,\ 0 \leqslant t \leqslant T \Rightarrow \mathsf{per\_snapshot}(H(t))) \\
&\qquad * (\forall t,\ 0 \leqslant t < T \Rightarrow \mathsf{transition\_inv}(H(t), H(t+1)))
\end{aligned}
$$

Figure 8 Definition of the shared state invariant encoding the hindsight reasoning. Variable H represents the history, T the current timestamp in use and C the abstract state of the structure.

Figure 8 shows a simplified definition of the invariant that encodes the hindsight reasoning. For sake of brevity, we provide only a high-level overview of the predicates used in the invariant. The predicate $\mathsf{Hist}(H, T)$ contains the mechanism to track the history of snapshots. That is, H denotes the history that has been observed so far and T is the current time stamp. Using appropriate ghost resources, it ensures that the timestamps are non-decreasing and past states recorded in H are preserved by future updates to the history. This allows us to define a *past predicate* $\Diamond_{s,t_0}(q)$ with the intuitive meaning that the history contains state s recorded after (or at) time t_0 for which proposition q holds true. The exact definition of the past predicate uses the ghost resources used to preserve the past states. The predicate $\mathsf{Hist}(H, T)$ also guarantees that $\mathrm{dom}(H) = \{0 \ldots T\}$, ensuring that there are no gaps in the history.

The conjunct $|H(T)| = C$ and the predicate $\overline{\mathsf{CSS}}(r, C)$ together tie the abstract state C of the data structure to the latest snapshot in the history. The predicate $\overline{\mathsf{CSS}}(r, C)$ is the dual of the representation predicate $\mathsf{CSS}(r, C)$ used in the client-level atomic specification. Both represent one half of an ownership over the abstract state of the structure, keeping the abstract state defined by $\mathsf{Inv}(r)$ synchronized with the representation predicate $\mathsf{CSS}(r, C)$.

The helping protocol predicate $\mathsf{Inv}_{help}(M, T)$ contains a *registry* of thread IDs with unmodifying linearization points that require helping from other concurrent threads. For each thread ID *tid* in the registry, the protocol stores information such as the start time of the thread, whether it has been linearized or not, etc.

The predicate $\mathsf{Inv}_{tpl}(r, H, T)$ captures invariants particular to the data structure under proof. It is further composed of three abstract predicates that are meant to be instantiated with the structure specific invariants. The three predicates serve the following purpose. The first predicate $\mathsf{resources}(r, H(T))$ ties the current snapshot to the physical representation of the structure. The predicate $\mathsf{Hist}(H, T)$ contains a conjunct $(\forall t, t < T \Rightarrow H(t) \neq H(t + 1))$. Together with the predicate resources, this conjunct forces a thread to update the history whenever the structure is modified.

The predicate $\mathsf{per\_snapshot}(H(T))$ captures the structural invariants that hold for any given snapshot. For instance, when proving the skiplist template, this predicate holds facts about the nodes *hd* and *tl* having maximum height, etc. The predicate $\mathsf{transition\_inv}(s, s')$ captures a transition invariant on snapshots observed in the history. That is, it constrains how certain quantities evolve over time. Again as an example from the skiplist template proof, the fact that a node marked in s remains marked in s' is included here. Crucially, the facts in $\mathsf{transition\_inv}(s, s')$ allow temporal interpolation required to establish facts about past states in the history (like in Section 3.2).

To summarize, the proof author defines the snapshot of the structure, the function $|\cdot|$, and instantiates the three abstract predicates in Inv_{tpl} appropriately. The resulting shared state invariant then tracks the history and handles the helping protocol without requiring further fine-tuning to the data structure at hand.

5 Verifying the Skiplist Template

We relate the intuitive proof argument from Section 3 to the development on hindsight reasoning in Iris in Section 4 to obtain a complete proof of the skiplist template. To achieve this, we must perform three tasks required by the proof method in Section 4. The first task is to determine the decisive operations that potentially alter the structure, and resolve the prophecy around those operations. As discussed previously, the decisive operations are `markNode` for `delete` and `changeNext` for `insert`. The `search` operation does not modify the abstract state and hence, it has no decisive operation.

The second task is to define a snapshot in the context of the skiplist template and instantiate Inv_{tpl} appropriately. This includes the predicate resources that ties the concrete state of the structure to the latest snapshot, as well as invariants that allow temporal interpolation. The third and the final task is to prove the hindsight specification for the core operations.

In this section we focus on the second task of defining the snapshot and providing invariants necessary to formalize the intuitive proof argument. Once, we have set up the right invariants, the formalized proof follows the intuitive proof very closely. We explain this with `delete` as an example.

5.1 Snapshot and the Skiplist Template Invariant

Recall that the notion of keysets are central to the intuitive proof argument for the core operations of the skiplist template. Hence, a snapshot of the structure must contain information about the keysets. For encoding keysets in Iris, we borrow heavily from [19], especially the *keyset camera* and the representation of keysets via the Flow Framework.

We define the snapshot of the skiplist template as a tuple containing the following components:

- the set of nodes N comprising the structure (also referred to as the *footprint* below)

$$
\begin{aligned}
\mathsf{Inv}_{tpl}(r, H, T) &:= \mathsf{resources}(r, H(T)) \\
&\quad * (\forall t,\ 0 \leqslant t \leqslant T \Rightarrow \mathsf{per\_snapshot}(H(t))) \\
&\quad * (\forall t,\ 0 \leqslant t < T \Rightarrow \mathsf{transition\_inv}(H(t), H(t+1))) \\
\mathsf{resources}(s) &:= \mathop{\Huge *}_{n \in \mathsf{FP}(s)} \mathsf{Node}(n, \mathsf{mark}(s,n), \mathsf{next}(s,n), \mathsf{key}(s,n), \mathsf{height}(s,n)) \\
&\quad * \mathsf{resources\_keyset}(s) \\
\mathsf{transition\_inv}(s, s') &:= (\mathsf{FP}(s) \subseteq \mathsf{FP}(s')) \\
&\quad * (\forall n,\ \mathsf{key}(s', n) = \mathsf{key}(s, n) \wedge \mathsf{height}(s', n) = \mathsf{height}(s, n)) \\
&\quad * (\forall n\ i,\ \mathsf{mark}(s, n, i) = \mathit{true} \Rightarrow \mathsf{mark}(s', n, i) = \mathit{true}) \\
&\quad * (\forall n\ i,\ \mathsf{mark}(s, n, i) = \mathit{true} \Rightarrow \mathsf{next}(s', n, i) = \mathsf{next}(s, n, i))
\end{aligned}
$$

Figure 9 Instantiating Inv_{tpl} with invariants of the skiplist template.

- the abstract state of the structure (a set of keys)
- the mark bits (a map from N to $\mathbb{N} \to \mathsf{Bool}$, i.e., a Boolean per level)
- the next pointers (a map from N to $\mathbb{N} \to N$)
- the keys (a map from N to K)
- the height of nodes (a map from N to $\mathbb{N}$)
- the representation of flow values

We reparameterize the $\mathsf{mark}(n, i)$ function introduced earlier to take the snapshot as an argument. Thus, we use $\mathsf{mark}(s, n, i)$ to mean the mark bit of node n at level i in snapshot s. We redefine $\mathsf{next}(\cdot)$, $\mathsf{key}(\cdot)$, $\mathsf{keyset}(\cdot)$ and other such functions similarly by adding the snapshot s as an additional parameter. We also use $\mathsf{FP}(s)$ to represent the footprint of the snapshot s.

We now present the skiplist template invariant in Figure 9. The resources predicate ties the snapshot to the concrete state through an intermediary node-level predicate $\mathsf{Node}(n, k, h, mk, nx)$. This predicate actually ties the physical representation of a node in the heap to the abstract quantities ($\mathsf{key}(\cdot)$, $\mathsf{height}(\cdot)$, $\mathsf{mark}(\cdot)$ and $\mathsf{next}(\cdot)$, respectively) that the skiplist template relies on. The Node predicate also owns all the resources needed to execute the helper functions. The skiplist template proof is parametric in the definition of Node. Thus, we achieve proof reuse across skiplist variants that follow the same high-level skiplist algorithm, but implement the node differently. We provide more details on this matter later. We discuss some concrete node implementations in Section 6.

The predicate $\mathsf{resources\_keyset}(s)$ capture the ownership resources required for keyset reasoning. Using the ghost resources in Iris and the keyset camera from [19], it ensures that the keysets and the logical contents of nodes in s satisfy (KeysetPr).

The predicate per_snapshot captures structural invariants that hold for all snapshots recorded in the history. This includes invariants of three kinds: first, invariants to ensure that each component of the snapshot is of the correct type and the maps (from nodes to mark bits, next pointers, etc.) are defined for all nodes in the footprint; second, the node-level invariants relating the node's inset, outset, mark bit, etc (like Invariant 1); and third, invariants about the hd and tl nodes, such as $\mathsf{key}(s, hd) = -\infty$, $\mathsf{height}(tl) = L$, etc.

The predicate $\mathsf{transition\_inv}(s, s')$ captures invariants about how certain quantities evolve over time, such as that mark bits once set to true remain true. The invariants 2, 3, and

$$\left\langle k\, h\, mk\, nx.\ \mathsf{Node}(n, k, h, mk, nx) \right\rangle\ \texttt{getKey}\ n\ \left\langle k.\ \mathsf{Node}(n, k, h, mk, nx) \right\rangle$$

$$\left\langle k\, h\, mk\, nx.\ \mathsf{Node}(n, k, h, mk, nx) \right\rangle\ \texttt{getHeight}\ n\ \left\langle h.\ \mathsf{Node}(n, k, h, mk, nx) \right\rangle$$

$$\left\langle k\, h\, mk\, nx.\ \mathsf{Node}(n, k, h, mk, nx) * (i < h) \right\rangle\ \texttt{findNext}\ i\ n\ \left\langle n'.\ \mathsf{Node}(n, k, h, mk, nx) * (nx(i) = n') \right\rangle$$

$$\left\langle k\, h\, mk\, nx.\ \mathsf{Node}(n, k, h, mk, nx) * (i < h) \right\rangle\ \texttt{markNode}\ i\ n$$
$$\left\langle \begin{array}{l} x.\ \mathsf{Node}(n, k, h, mk', nx) * (mk(i) = \mathit{true} \Rightarrow x = \mathsf{Failure} * mk' = mk) \\ \qquad * (mk(i) = \mathit{false} \Rightarrow x = \mathsf{Success} * mk' = mk[i \rightarrowtail \mathit{true}]) \end{array} \right\rangle$$

$$\left\langle k\, h\, mk\, nx.\ \mathsf{Node}(n, k, h, mk, nx) * (i < h) \right\rangle\ \texttt{changeNext}\ i\ n\ n'\ e$$
$$\left\langle \begin{array}{l} x.\ \mathsf{Node}(n, k, h, mk, nx') * ((mk(i) = \mathit{true} \lor nx(i) \neq n') \Rightarrow x = \mathsf{Failure} * nx' = nx) \\ \qquad * ((mk(i) = \mathit{false} \land nx(i) = n') \Rightarrow x = \mathsf{Success} * nx' = nx[i \rightarrowtail e]) \end{array} \right\rangle$$

Figure 10 Specifications of the helper functions used by the skiplist template.

4 presented in Section 3 are part of this predicate. These invariants form the crux of the hindsight reasoning, as they enable temporal interpolation.

Before we go into the formal proof argument for `delete`, we must discuss how to reason about the node-level helper functions. Figure 10 shows the specification for the helper functions assumed by the skiplist template. The specifications are logically atomic, i.e., they behave like a single atomic step in the template. The preconditions for all of the functions rely solely on the predicate Node. The functions `getKey`, `getHeight` and `findNext` read various components of the node. Note that `findNext` reads both the mark bit and the next pointer together.

The specification for functions `markNode` and `changeNext` is slightly more complex because they potentially change the structure. Let us explain them briefly. For `markNode` on node n at level i, the return value (`Success` or `Failure`) is determined by whether n is already marked at i. If it is, then the function returns `Failure` without modifying the node. If it is unmarked, then `markNode` successfully marks it, and updates the node accordingly. The specification for `changeNext` can be interpreted similarly. Here, the return value hinges upon the mark bit being false and the next pointer of n pointing to n' at i.

5.2 Proof of `delete`

We now have all the ingredients to show that `delete` satisfies (HindSpec). We provide only a high-level summary of the proof here. Please see [36] for more details.

The precondition provides access to the invariant $\mathsf{Inv}(r)$ and knowledge that the thread ID is tid with start time t_0. Additionally, the thread has the right to resolve prophecy p around the decisive operations, and if the thread observes a successful decisive operation, then the atomic update $\mathsf{AU}(\Phi)$ is available to help with the linearization. The `delete` operation begins with `traverse`. Using the $\diamondsuit$ operator defined in Section 4.2, we express the postcondition of `traverse` as

$$\diamondsuit_{s,t_0}(k \in \mathsf{keyset}(s, c) \land (\mathit{res} \Leftrightarrow k \in C(s, c))).$$

Intuitively, this assertion captures that there is a past state s in the history (after time point t_0) in which k is in the keyset of c and res is true iff k is in the logical contents of c.

The argument here proceeds by case analysis on res. Let us first consider the case that res is false. The `delete` operation also terminates with false. Since the thread terminates without any calls to the decisive operations, this case corresponds to the $\neg\mathsf{Upd}(\mathit{pvs})$ case in the postcondition of (HindSpec). The postcondition requires `delete` to establish the predicate $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$. In this context, establishing this predicate amounts to identifying a witness past state in which k was not part of the abstract state. Clearly, this is witnessed by state s from the specification of `traverse`. Applying (KeysetPr) in state s, we can establish the predicate $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$.

Now, let us consider the case that *res* is *true*. The `maintainanceOp_del` marks node c at the higher level, but the interesting part of the proof is when the decisive operation `markNode` is called at the base level (Line 15). Again there are two cases to consider, depending on whether `markNode` succeeds. If `markNode` succeeds, then we can establish $\mathsf{Upd}(\mathit{pvs})$ as we see a `Success` value being resolved. In this case, the precondition of (HindSpec) provides the atomic update $\mathsf{AU}(\Phi)$. Since, the thread has modified the abstract state, this becomes the linearization point. The thread can linearize with $\mathsf{AU}(\Phi)$ to obtain the receipt Φ and satisfy its postcondition. The proof also has to update the history with the new snapshot of the structure, as c goes from being unmarked to marked.

The final (and most interesting) case is when `markNode` fails. Here again, we must establish $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$ to complete the proof of (HindSpec). Two facts are useful: (i) in the past state s referred to in the `traverse` spec, we can establish that $\mathsf{mark}(s, c) = \mathit{false}$; and (ii) since the `markNode` has failed, in the current state say s_0, $\mathsf{mark}(s_0, c) = \mathit{true}$. Hence, by using the second conjunct of transition_inv in Figure 9 and temporal interpolation on the two facts above, we can infer the existence of two consecutive states s_1 and s_2, such that $\mathsf{mark}(s_1, c) = \mathit{false}$ and $\mathsf{mark}(s_2, c) = \mathit{true}$. Clearly, a concurrent `delete` thread marked c in state s_2. Hence, this state becomes the witness to establish the predicate $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$. This completes the proof that `delete` satisfies (HindSpec).

6 Proof Mechanization and Evaluation

We now shed light on the mechanization of the hindsight methodology, as well as its application to the skiplist template. We additionally reverify the multicopy template from [35] using our new hindsight specification to modularize the proof effort. Although the multicopy algorithms are lock-based, hindsight reasoning is helpful in their verification. The case study demonstrates a substantial reduction in proof size due to the encoding of hindsight reasoning in Iris, illustrating the generality of our contribution. Our development is available as a VM and docker image on Zenodo [3].

All of the proofs we discuss below are mechanized in Iris/Coq. The templates, traversals and the node implementations are written in Iris's default programming language HeapLang. In order to correctly capture the dependence between different layers of the proofs (such as hindsight specification and the templates, the templates and the `traverse`/node implementations), we heavily make use of Coq's module system.

The organization of our proofs is shown in Figure 11. Going from left to right, the first column relates to the formalization of hindsight reasoning in Iris. The box "Hindsight" captures the assumptions regarding the hindsight specification from Section 4. These assumptions not only include the hindsight specification itself but also the relevant definitions of snapshots, histories, etc. The module "Client-level Spec" relates the client-level specification expressed in terms of atomic triples to the hindsight specification used for the template-level proofs. The corresponding proof involves the reasoning about prophecies and the helping protocol, which is done once and for all and applicable to all data structures that fulfill the assumptions made in the "Hindsight" module.

The middle column consists of modules for the two verified templates (skiplist and multicopy) and the associated proofs verifying the template operations against the hindsight

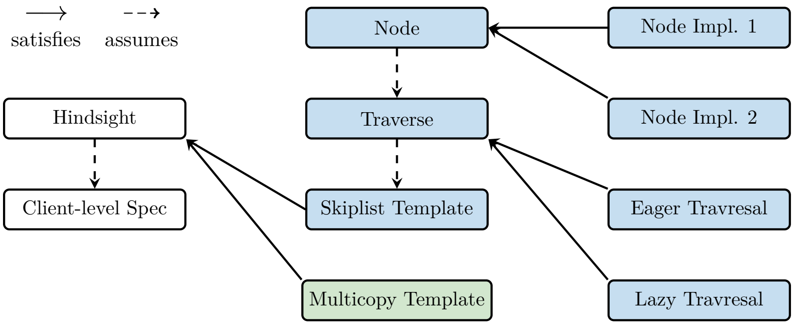

Figure 11 The structure of our proofs. Each box represents a collection of modules relevant to the label. The dashed arrows represent module dependence, i.e., assumption of specifications. The normal arrows represent implementation of the target module (fulfillment of the assumptions).

specification. We discuss them in turn.

Skiplist template case study. The skiplist template, as described in Figure 2, abstracts from the concrete implementations of nodes and the `traverse` operation. Hence, we package their specifications into separate modules. To ensure that the specified data structure invariant for the skiplist template is not vacuous, we also verified an `init` routine that initializes the data structure and establishes the invariant.

The final column shows modules for the two node implementations of the skiplist template, as well as the eager and lazy traversal discussed in Section 2. The helper functions `markNode` and `changeNext` are implemented using an atomic CAS operation in both of the node implementations. The crux of the node implementation for the skiplist template is to determine a memory representation of the mark bit and the next pointer (at some level) such that both values can be read or written together with one atomic CAS operation. The first node implementation does this by using a sum type. The second node implementation is conceptually similar but uses more low-level data types instead of a sum type.

The traversal and node implementations above correspond to several existing lock-free (skip)list algorithms from the literature. The Herlihy-Shavit skiplist algorithm [11, § 14] is obtained by instantiating our template with the eager traversal, the node implementation 2, and maintenance operations that link higher-level nodes in increasing order of level and unlink nodes in the opposite order. The Michael set [32] is obtained as a degenerate case of the Herlihy-Shavit template instantiation where the skiplist is restricted to $\mathsf{L} = 2$ (For $\mathsf{L} = 2$, Level 1 consists of only a fixed single edge between the sentinel nodes. So, conceptually, Level 1 can be ignored in this case.)

We obtain a novel variant of a skiplist by replacing the eager traversal in the Herlihy-Shavit instantiation with the lazy traversal. The lazy traversal is inspired by the Harris list algorithm [10], which is obtained as a degenerate case of this new lazy skiplist algorithm by restricting it to $\mathsf{L} = 2$.

We present a summary of the proof effort for the skiplist template in Table 1. The proof-checking time was measured on the Docker image running on an Apple M1 Pro chip with 16GB RAM. The flow library contains the Iris formalization of the Flow Framework developed in [19, 35]. As a minor contribution, we extend this library with general lemmas for reasoning about graph updates that have an affect on an unbounded number of nodes. These lemmas are useful for the proofs of `insert`, `delete` and lazy `traverse`. The unbounded updates, as well as the maintenance operations, are the reason for the relatively high number of proof lines for the `insert` and `delete` operations.

Skiplist Template (Iris/Coq)

Module	Code	Proof	Total	Time
Flow Library	0	5330	5330	33
Hindsight	0	950	950	11
Client-level Spec	9	329	338	18
Skiplist	12	1693	1705	26
Skiplist Init(∗)	6	319	325	15
Skiplist Search(∗)	7	62	69	6
Skiplist Insert(∗)	37	3457	3494	111
Skiplist Delete(∗)	28	2401	2429	72
Node Impl. 1	118	908	1026	35
Node Impl. 2	106	836	942	35
Eager Traversal	38	1165	1203	96
Lazy Traversal	47	2063	2110	145
Total	**408**	**19513**	**19921**	**603**
Herlihy-Shavit	243	11212	11455	390

Table 1 Summary of the proof effort. For each module, we show the number of lines of program code, lines of proof, total number of lines, and the proof-checking time in seconds. The code for the initialization and the core operations of the skiplist (entries with (∗)) is technically defined in the "Skiplist" module, however here we present them separately for each operation to provide a better picture. The count for Herlihy-Shavit is the summation of rows "Hindsight", "Client-level Spec", all "Skiplist" modules, "Node Impl. 2" and "Eager Traversal".

Multicopy template case study. The multicopy template from [35] generalizes search structures such as the lock-based Log-Structured Merge (LSM) tree used widely in modern database systems. It satisfies the Map ADT specification, with `search` and `upsert` (for insert/update) as its core operations. To deal with the complexity of future-dependent external linearization points, the original proof relies on an intermediate template-level specification based on the concept of *search recency*.

Table 2 presents a detailed comparison of the multicopy template proofs from [35] versus the new proof based on the hindsight framework. The original proof consists of a total of 2779 lines. By contrast, the definitions ("Defs") and "Client-level Spec" proofs can be factored out of the total cost of the hindsight-based proof, because it is part of the hindsight library itself. Hence, the new proof based on hindsight reasoning consists of only 1310 lines, which is a reduction of 53%. To summarize, the improvement stems from the fact that the original proof relies on an intermediate specification and a helping protocol that is tailored to multicopy structures, while our new proof uses a helping protocol that is shared among all proofs that build on the new hindsight proof method.

While the majority of the reduction in the proof size stems from the elimination of structure-specific specifications and helping protocol proofs, we also saw a minor reduction in the size of the remainder of the proof. One outlier is the proof of `upsert`. Here, the increase is attributed to the fact that the proof has to construct a fresh snapshot when the operation succeeds. However, this construction is conceptually simple and could be factored out into more abstract lemmas that are provided directly by the hindsight library.

Multicopy Template (Iris/Coq)		
Module	**Original**	**Hindsight**
Defs	866	(950)
Client-level Spec	434	(338)
LSM	741	540
Search	411	399
Upsert	327	371
Total	**2779**	**1310**

Table 2 Comparison of multicopy template proofs. The column "Original" shows the number of lines from the proofs in [35], while "Hindsight" shows them for our new proof effort. Module "Defs" represents definitions required for proving the client-level specification (helping invariant, history predicate, etc). Module "Client-level Spec" contains the proof relating the intermediate specification (Search Recency Specification from [35] and Hindsight Specification in our paper) to the client specification. Module "LSM" contains definitions required to instantiate the frameworks for LSM trees. Modules "Search" and "Upsert" refer to the proofs for the search and upsert operations, respectively. Entries in '()' for the 'Hindsight' column are not included in the total due to being part of the hindsight library.

7 Related Work

The formal verification of linearizability has received much attention in recent years. We refer to [6] for a survey of relevant techniques and focus our discussion to the most closely related works.

Our work builds on the idea of template algorithms for lock-based concurrent search structures of [20, 35, 19], which we extend to the setting of lock-free implementations. A common challenge when verifying linearizability of lock-free data structures is the prevalence of future-dependent and external linearization points. Hindsight theory [33, 23, 7, 8, 27, 28] has emerged as a suitable technique to address this challenge in the context of concurrent search structures. To our knowledge, we are the first to formalize hindsight reasoning within a foundational program logic. Tools like Poling [39], plankton [27, 28], and nekton [26] automate hindsight reasoning at the expense of an increased trusted code base. However, these tools currently cannot handle complex data structures with unbounded outdegree like skiplists. Also, they do not aim to characterize the design space of related concurrent data structures like our template algorithms do.

Other techniques for dealing with future-dependent linearization points include arguments based on forward simulation (e.g., by tracking all possible linearizations of ongoing operations [13], tracking a partial order [18], or using commit points [4]) and backward simulation (e.g., using prophecy variables [1, 24, 16]). Our encoding of hindsight reasoning in Iris combines forward reasoning (by tracking the history of the data structure state) and backward reasoning (by using prophecies). However, the details of this encoding are for the most part hidden from the proof engineer by providing a higher-level reasoning interface based on past predicates and temporal interpolation as proposed in [28]. Our comparison with a prior proof of multicopy structure templates [35] suggests that this abstraction helps to reduce the proof complexity.

Several works propose techniques for automatically verifying concurrent skiplists. Abdulla et al. [2] propose a technique for verifying linearizability of lock-free list-based data structures using forest automata. The evaluation considers bounded skiplists with up to 3 levels. However, the implementation does not scale to larger bounds and the unbounded case is outside the scope of the technique. We note that the height of the skiplist is tied to the expected runtime of the skiplist operations. To guarantee the expected worst-case runtime bounds, the skiplist's height must be of order $O(\log(n))$ where n is the expected maximal number of entries in the list. For this reason, real-world skiplist implementations are also parametric in the height. Heights up to 63 levels are feasible in deployed skiplists [25], so the restriction to height 3 in [2] is unrealistic. By contrast, our proofs cover skiplists of arbitrary height.

Sánchez and Sánchez [37] present an SMT-based approach towards an automated verification of concurrent lock-based skiplists. The approach is based on a decidable theory of unbounded skiplists. However, it does not consider lock-free implementations and focuses on establishing *shape invariants* preserved by the structure instead of proving linearizability.

Unlike these automated tools, our approach does not rely on data-structure specific decidable theories for reasoning about inductive properties of heap graphs. Instead, we build on the Flow Framework [21, 22, 29], which enables local reasoning about such properties over general graphs in separation logic. As a minor contribution, we extend the mechanization of the Flow Framework from [20] with lemmas to reason about graph updates that affect properties of an unbounded number of nodes.

There are some skiplist algorithms that are not immediately covered by our template algorithm. For example, skiplists based on the algorithm presented in [9] such as Java's `ConcurrentSkipListMap` [34] use *backlinks* to avoid restarts when a traversal fails. However, we believe that our template algorithm can be extended to subsume such algorithms by abstracting from the restart policy, similarly to how the present template abstracts from the maintenance policy.

In this paper, we assume a programming language with a garbage collected semantics. The rationale for this assumption is that issues arising from manual memory reclamation can be addressed by orthogonal means. For instance, [30, 31] propose a technique that decouples the proof of data structure correctness from that of the underlying memory reclamation algorithm, allowing the correctness proof of the data structure to be carried out under the assumption of garbage collection. Recent work also showed how to carry out such modular proofs in program logics like Iris [14].

8 Conclusions and Future Work

This paper shows how to verify some of the most challenging concurrent data structure algorithms in existence. The accompanying proofs are fully mechanized in the foundational program logic Iris. The proofs are modular and cover the broader design space of the underlying algorithms by parameterizing the verification over aspects such as the low-level representation of nodes and the style of data structure maintenance.

Besides being the first work to verify unbounded lock-free skiplists, the work has developed technologies for Iris, particularly hindsight reasoning, that can be useful in many applications.

Our proofs guarantee safety but not liveness. This limitation is shared by the algorithms they verify: in any highly concurrent (minimal or no locking) setting, a thread t may never complete because of other threads that overtake it. Fortunately, this never happens in practice where threads all advance more or less at the same pace. Verifying liveness under such fairness assumptions remains an interesting direction for future work.

Another area of future work is to verify algorithms that mix locking parts with lock-free parts both for single copy and multicopy search structures. We believe that the present framework will be a good basis for that effort.

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

2 Parosh Aziz Abdulla, Lukás Holík, Bengt Jonsson, Ondrej Lengál, Cong Quy Trinh, and Tomás Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2013.

3 Anonymous. *Artifact for "Verifying Lock-free Search Structure Templates"*, April 2024. URL: `https://doi.org/10.5281/zenodo.11051385`.

4 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017.

5 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *ECOOP*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, 2014.

6 Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19:1–19:43, 2015.

7 Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of chaos: Proving linearizability using local views. In *DISC*, volume 121 of *LIPIcs*, pages 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

8 Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.*, 4(OOPSLA):128:1–128:29, 2020.

9 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59. ACM, 2004.

10 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.

11 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

12 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

13 Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *PACMPL*, 8(POPL), jan 2024. `doi:10.1145/3632924`.

14 Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2):828–856, 2023.

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

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

17 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

18 Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. Proving linearizability using partial orders. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 639–667. Springer, 2017.

19 Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. Verifying concurrent search structure templates. In *PLDI*, pages 181–196. ACM, 2020.

20 Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. *Automated Verification of Concurrent Search Structures*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2021.

21 Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.*, 2(POPL):37:1–37:31, 2018.

22 Siddharth Krishna, Alexander J. Summers, and Thomas Wies. Local reasoning for global graph properties. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 308–335. Springer, 2020.

23 Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015.

24 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.

25 Meta. Facebook Open Source Library: ConcurrentSkipList. `https://github.com/facebook/folly/blob/main/folly/ConcurrentSkipList.h`. Last accessed: Apr 2024.

26 Roland Meyer, Anton Opaterny, Thomas Wies, and Sebastian Wolff. nekton: A linearizability proof checker. In *CAV (1)*, volume 13964 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2023.

27 Roland Meyer, Thomas Wies, and Sebastian Wolff. A concurrent program logic with a future and history. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1378–1407, 2022.

28 Roland Meyer, Thomas Wies, and Sebastian Wolff. Embedding hindsight reasoning in separation logic. *Proc. ACM Program. Lang.*, 7(PLDI):1848–1871, 2023.

29 Roland Meyer, Thomas Wies, and Sebastian Wolff. Make flows small again: Revisiting the flow framework. In *TACAS (1)*, volume 13993 of *Lecture Notes in Computer Science*, pages 628–646. Springer, 2023.

30 Roland Meyer and Sebastian Wolff. Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.*, 3(POPL):58:1–58:31, 2019.

31 Roland Meyer and Sebastian Wolff. Pointer life cycle types for lock-free data structures with memory reclamation. *Proc. ACM Program. Lang.*, 4(POPL):68:1–68:36, 2020.

32 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82. ACM, 2002.

33 Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *PODC*, pages 85–94. ACM, 2010.

34 Oracle. Java concurrent skiplist set. `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/ConcurrentSkipListSet.html`. Last accessed: Jan 2024.

35 Nisarg Patel, Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Verifying concurrent multicopy search structures. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021.

36 Nisarg Patel, Dennis Shasha, and Thomas Wies. Verifying lock-free search structure templates. *CoRR*, abs/2405.13271, 2024. URL: `http://arxiv.org/abs/2405.13271`, `arXiv:2405.13271`.

37 Alejandro Sánchez and César Sánchez. Formal verification of skiplists with arbitrary many levels. In *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 314–329. Springer, 2014.

38 Dennis E. Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90, 1988.

39 He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015.