Formal Specification and Verification of the Lazy JellyFish Skip List

Pedro Carrott
pedro.carrott@tecnico.ulisboa.pt
INESC-ID and IST, University of Lisbon
Lisbon, Portugal

Abstract

Concurrent append-only skip lists are widely used in data store applications, so as to maintain multiple versions of the same data with different timestamps, rather than delete outdated information. One such skip list implementation is JellyFish, which greatly mitigates the drop in performance witnessed in other skip lists induced by the append-only design. JellyFish accomplishes this feat by storing in each node a consistent timeline of values as a linked list, instead of inserting new nodes in the skip list.

In this work, we present a lock-based variant of JellyFish, using a lazy synchronization strategy, and formally verify its functional correctness. We further show that this data structure satisfies the specification of a concurrent map. To reason about concurrent updates on values, we define a novel resource algebra over timestamped domains. Using the argmax operator for this algebra, we prove that concurrent updates to the map always maintain the most recent values. We also show that updates to a node maintain its history of values consistent. Our proofs are mechanized in Coq using the concurrent separation logic of Iris.

Keywords: separation logic, specification, formal verification, concurrent data structures, Iris, Coq

1 Introduction

A map is an abstraction for a data structure which associates an identifying key to each value it stores. This abstraction has been vastly studied in computer science holding many real-world use cases. For instance, data store applications make use of *concurrent* maps to index data in a thread-safe environment. In fact, most of these applications maintain a history of values for each key in the map, so as to record different versions of the same data, instead of deleting outdated information. This append-only design, however, tends to hamper the performance of these concurrent maps, depending on how the old values are stored.

As an abstraction, maps can be implemented in several ways, with different assurances on their performance. While self-balancing trees are a classical approach, a more efficient implementation is the skip list. Structurally, skip lists are very similar to balanced trees but maintain their balance through a probabilistic strategy, rather than explicit rebalancing procedures. This difference makes the skip list preferable in concurrent environments, which is why it is the most widely used map implementation in data store applications. In particular, JellyFish [21] is a state-of-the-art skip list implementation whose performance surpasses that of other skip lists used in industry. As an append-only skip list, JellyFish efficiently supports version control by storing a list of values in every node. This list corresponds to the node's history of values with each value being assigned to a timestamp. To ensure that chronological order is maintained, new values are never added to the list if their assigned timestamps are less recent than any value already in the list.

Contributions. This implementation adopts a lazy synchronization strategy, which makes it easier to reason about its correctness. Our work proves that this skip list satisfies the specification of a concurrent map with timestamped values. Using the concurrent separation logic of Iris [7–9, 11], we reason about updates to the map by defining a novel resource algebra for the argmax operator. This operator abstracts the expected behaviour for map updates, since it always retains the value with the maximum argument, which in this context corresponds to the value with the most recent timestamp. Our proofs are mechanized in Coq and available at:

https://github.com/sr-lab/iris-jellyfish

To the best of our knowledge, this is the first effort to verify (a variant of) the JellyFish skip list and to reason about concurrent maps with version control through timestamped domains. The main contributions of this work can thus be summarized as follows: **(1)** the first verification effort of the JellyFish design for concurrent append-only skip lists; **(2)** a new concurrent map specification, which supports version control through the use of timestamps; **(3)** a mechanized proof that our skip list implementation satisfies the concurrent map specification; **(4)** a novel resource algebra in the concurrent separation logic of Iris for the argmax operation.

2 Background

We begin this section with an overview of skip lists, followed by a high-level description of the JellyFish design for concurrent append-only skip lists. We then introduce Iris, the Coq framework that we use to reason about our lazy variant of JellyFish.

2.1 Skip Lists

A skip list [15] is a list-based data structure, which can be considered a generalization of a sorted linked list. This data structure is composed of a maximum number of levels, where level 0 corresponds to the complete linked list and each level l is a sublist of level $l-1$. Since each level contains progressively fewer elements of the original list, maintaining all lists sorted allows searches in higher levels to skip elements that would otherwise be traversed in a standard linear search. We search in the top level, stop searching when we reach a value equal to or greater than the key, descend to the next level starting from the same element and repeat until we reach the bottom level. The search can end before reaching the bottom, if the key is found in one of the sublists.

The JellyFish Skip List. Several concurrent skip lists have been proposed, including append-only structures which are widely used in data store applications (*e.g.*, RocksDB[1] and LevelDB[2]). Rather than overwrite existing values, this design choice maintains a record of all values that have been stored during the data structure's lifetime. An efficient approach is to maintain a list of timestamped values per node, referred to as the node's *vertical list*. Updates to a key are then made by prepending a new value to its vertical list, maintaining the skip list itself with the same nodes. This approach corresponds to the design of the JellyFish skip list [21].

JellyFish is one of many lock-free concurrent skip list algorithms. However, while a non-blocking solution is desirable to reduce contention, this approach makes it harder to reason about the correctness of these algorithms. For instance, in the ConcurrentSkipListMap from the Java concurrency package[3], "*certain interleavings can cause the usual skip-list invariants to be violated*" [6]. For this reason, we consider a lock-based skip list algorithm, which follows the design of JellyFish and employs the lazy synchronization strategy of the lazy list of Heller *et al.* [5]. Inspired by the concurrent skip list of Herlihy *et al.* [6], the key idea behind the algorithm is to treat each level as an individual lazy list. The implementation for this lazy JellyFish skip list is discussed in detail in Section 3.

2.2 Iris

To verify the correctness of the lazy JellyFish skip list we have mechanized proofs for its methods in the Coq proof assistant using the Iris proof mode [10, 12]. Iris is a concurrent separation logic that allows reasoning about deep correctness properties for fined-grained concurrent programs written in higher-order imperative languages [8]. As a separation logic, Iris allows reasoning about ownership of *disjoint* resources (*e.g.*, heap addresses), which is useful for concurrent programs where threads handle separate parts of the heap. To reason about shared resources, however, Iris contains two key ingredients at its base: invariants and ghost state.

Invariants allow us to express assertions about a shared state between threads, which must always hold true, regardless of what changes are applied to that state. Access to this shared state can be obtained by opening the invariant, i.e., by claiming temporary ownership of the resources in the invariant. The invariant can only remain open for the duration of a single step of computation and must still hold true after this step is executed. The invariant can then be closed by revoking the thread's temporary ownership of the shared resources. By considering a single step, we can avoid reasoning about other threads interfering with its execution, assuring that the invariant will be preserved throughout all steps of the program. In Section 4, we discuss ghost state, which is what enables us to reason about altering shared resources, while preserving the invariant assertions.

3 A Lazy Variant of JellyFish

Lazy JellyFish is an append-only skip list implementation where the data is organized in memory following the JellyFish design [21], while concurrent updates to it employ a lock-based lazy synchronization strategy. Its implementation is shown in Figure 1, including the usual operations on maps: new, get and put. To match the JellyFish design, a node for this data structure is represented as a tuple (k, v, l, n) where:

- k is an integer for the node's key;
- v is a pointer for the node's vertical list;
- l is an array for the node's locks in each level;
- n is an array for the node's successors in each level.

In turn, the vertical list of a skip list node contains nodes represented as tuples (v, t, p) where:

- v is a value;
- t is the assigned timestamp;
- p is a pointer to the previous value's node.

Data Initialization. The skip list's construtor, new, initializes the data structure with two nodes, referred to as the left and right sentinels, with keys MIN and MAX, respectively; these values bound the range of valid keys for the map. Both arrays of the left sentinel are created with HMAX+1 entries using the **AllocN** primitive. All entries of the successor array point to the right sentinel, while initLocks creates a different lock for each entry of the lock array. Pointer arithmetic is used to index array entries: we write $a+i$ instead of the standard C-style notation $a[i]$. The right sentinel, on the other hand, has no successors or locks, since keys will always be lower than MAX, remaining constant throughout the skip list's lifetime. Since the sentinels do not correspond to actual entries of the map, no value is associated to these nodes, *i.e.*, these nodes have no vertical list. The constructor returns a pointer to the left sentinel.

[1]https://github.com/facebook/rocksdb
[2]https://github.com/google/leveldb
[3]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html

```
initLocks locks lvl ≜
  if $lvl = \mathrm{HMAX} + 1$ then ()
  else   locks + lvl ← newlock ();
         initLocks locks (lvl + 1)

new ≜
  let tail = (MAX, NULL, NULL, NULL) in
  let next = AllocN (HMAX + 1) (ref tail) in
  let locks = AllocN (HMAX + 1) () in
    initLocks locks 0;
    ref (MIN, NULL, locks, next)

find pred k lvl ≜
  let succ = !!(pred.next + lvl) in
    if $k \leq$ succ.key then (pred, succ)
    else find succ k lvl

findAll pred k lvl h ≜
  let (pred, succ) = find pred k lvl in
    if lvl = h then (pred, succ)
    else findAll pred k (lvl − 1) h

get p k ≜
  let (_, succ) = findAll !p k HMAX 0 in
    if $k \neq$ succ.key then None
    else let v = !(succ.val) in Some (v.val, v.ts)

link pred lvl n ≜
  let new = !n in
    new.next + lvl ← !(pred.next + lvl);
    new.locks + lvl ← newlock ();
    pred.next + lvl ← n

createAndLink pred k v t h ≜
  let next = AllocN (h + 1) () in
  let locks = AllocN (h + 1) () in
  let val = ref (v, t, NULL) in
  let n = ref (k, val, locks, next) in
    link pred 0 n;
    n

update node v t ≜
  let val = !(node.val) in
    if $t <$ val.ts then ()
    else node.val ← (v, t, ref val)

findLock pred k lvl ≜
  let (pred, _) = find pred k lvl in
  let lock = !(pred.locks + lvl) in
    acquire lock;
    let succ = !!(pred.next + lvl) in
      if $k \leq$ succ.key then (pred, succ)
      else   release lock;
             findLock succ k lvl

insert pred lvl n ≜
  let k = !n.key in
  let (pred, _) = findLock pred k lvl in
  let lock = !(pred.locks + lvl) in
    link pred lvl n;
    release lock

tryInsert pred k v t h ≜
  let (pred, succ) = findLock pred k 0 in
  let lock = !(pred.locks) in
    if k = succ.key
    then   update succ v t;
           release lock;
           None
    else   let n = createAndLink pred k v t h in
             release lock;
             Some n

insertAll curr k v t h lvl ≜
  if lvl = 0 then tryInsert curr k v t h
  else   let (pred, _) = find curr k (lvl − 1) in
         let opt = insertAll pred k v t h (lvl − 1) in
           match opt with
           | None     ⇒   None
           | Some n   ⇒   insert curr lvl n;
                          Some n

  putH p k v t h ≜
    let (pred, _) = findAll !p k HMAX h in
    insertAll pred k v t h h

  put p k v t ≜
    let h = randomLevel in
    let _ = putH p k v t h in ()
```

Figure 1. Pseudocode for the lazy JellyFish skip list. Its public methods are highlighted by a solid contour. A dashed contour highlights the private update procedure, which is responsible for updating a node's vertical list.

Data Retrieval. The get method performs a search for a key and returns its current value. Being parameterized by the left sentinel pointer p and the lookup key k, it first invokes findAll to traverse the skip list as it would in a sequential setting. The find procedure iterates over a single level, returning a pair of nodes with the greatest key lower than k and the lowest key equal to or greater than k within the current level. The pair of nodes from the bottom list is then returned by findAll, containing the last node in the skip list with key lower than k and the first node with key equal to or greater than k, following a case analysis for the latter's key. If the key differs from k, then k is not in the map and so there is no value associated to k. Otherwise, k belongs to the map's domain and its most recent value is stored in the head of its vertical list. We then retrieve the value and timestamp from this vertical list node.

Given that concurrent updates to the map might occur, the get method is optimistic in the sense that it might return outdated results. However, a correct result might still be invalidated by a concurrent write taking place immediately after the search's conclusion, regardless of the chosen implementation. It is thus preferable to opt for a non-blocking approach, since it reduces contention, making the findAll procedure suitable as a helper function for the put operation.

Data Updates. Concurrent writes on the map are done through the put method, which can insert new nodes or update existing ones. Updating a key k consists in associating it to a new value v and timestamp t. In case the key is not found, a new node must be created with a given height h chosen through randomLevel. The random height h is then passed to putH along with the remaining arguments to execute the map update. Since the levels above h will not contain the new node, findAll initially traverses the skip list until level h. The insertAll procedure is then called to either insert a new node in all remaining levels or update an existing node. To preserve the sublist relation between levels, the traversal continues until the bottom level, so that insertions may be executed bottom up. If a new node is inserted in the bottom list, then this node is propagated to the upper levels for insertion; otherwise, an existing node, already connected to all intended levels, has its value and timestamp updated upon reaching the bottom level.

To ensure that insertions and updates are done correctly, both insert and tryInsert first call findLock. Lazily, findLock tries to acquire the only lock it needs to perform the intended insertion or update. It first executes find to obtain the node with the greatest key lower than k in the current level and then acquires its lock. Since we only obtain exclusive ownership of the lock *after* acquiring it, we check if the node's successor still has a key equal to or greater than k. If not, the node no longer holds the greatest key lower than k; find is then continued from the successor to obtain a new node, repeating the process. Otherwise, we have acquired exclusive ownership of the lock and the successor's key is guaranteed to be equal to or greater than k.

If the keys are different, tryInsert will create a node for the new key-value pair. While the list insertion is performed through the link procedure at all levels, the new node is only created at the bottom level. Furthermore, we only validate the successor's key at the bottom level. Due to the sublist relation, if k is not in the bottom list, then it is not in any of its sublists. It is thus safe to invoke insert in the upper levels without validating the keys, as long as there is only one thread performing the insertions. This is guaranteed by having all threads reaching the bottom level first to claim the right for creating and inserting the new node. The threads which acquire the lock after the node has been inserted will see that the successor of the locked node has key k, so they will attempt to update its value.

The sequence of values in a node's vertical list should reflect a monotonic increase of the associated timestamps. Therefore, update fails when the node's timestamp is more recent than t, since it would yield an inconsistent timeline. On the other hand, a successful update occurs when t is more recent, as well as when the timestamps are equal. A new node containing v and t is then prepended to the head of the vertical list, extending the current history of values of the node.

4 Reasoning about Timestamped Domains

We now focus on reasoning about the correctness of the lazy JellyFish skip list. A key feature of Iris that allows us to reason about mutable shared state is its capability of defining, through *resource algebras*, an abstract or auxiliary state of the program, to which we call *ghost state*. In this section, we describe ghost state and present a novel resource algebra for reasoning about values with timestamps.

4.1 Ghost State in Iris

Ghost state provides a way of matching the physical state of shared data with an abstract state where certain properties must hold. For instance, if we choose to model this abstract state as a partial commutative monoid (PCM), we can ensure that operations on the shared state are commutative and associative. These two properties are useful when reasoning about concurrency, because they imply that, for any set of operations, the order of execution is irrelevant, eliminating the need for a combinatorial analysis.

Resource Algebras. Ghost state in Iris is defined as a resource algebra (RA), a broader algebraic definition than PCMs. The differences between both constructs are present in two other properties that define a PCM: partiality and the unit element. Partiality is a useful notion, since it allows us to express that a given operation might be undefined between certain elements; in RAs, operations are total, but some combinations of elements are deemed invalid, capturing a similar effect. Regarding the unit element, PCMs require a single unit for all elements, while RAs generalize this notion by requiring a unit (if any) for each element, defined by a core function. If the core is the same for all elements, then we obtain a unital RA, which only differs from a PCM in terms of partiality. The RA operator can thus be used to compose ghost state through the following rule:

$$\lceil a \rceil^{\gamma} * \lceil b \rceil^{\gamma} \dashv\vdash \lceil a \cdot b \rceil^{\gamma}$$

The dashed lines denote that we are dealing with a ghost variable (an auxiliary variable, rather than an in-memory program variable) associated to the ghost name γ, while the separating conjunction ($*$) is used to express the ownership of disjoint *ghost* resources, which can be owned separately by different threads. Through this rule, separate resources which correspond to the same ghost variable can be combined into a single resource using the operation of the underlying RA ($\cdot$). We can thus reason about changes to resources locally and the RA handles how those local changes can be combined to reflect the global state.

4.2 The argmax Resource Algebra

Our goal is to verify an implementation for a concurrent map. Since ghost state must represent an accurate abstraction of the data we are reasoning about, we require a suitable RA over maps. To define such a RA, we need to first understand how composition between maps should be applied. For maps with no keys in common, we can simply merge both maps into a single map with all key-value pairs, similarly to the set union. However, when both maps have some key in common, the associated value in each map might differ from one another. In this scenario, what value should the key possess? One way of handling this issue would be to invalidate such combinations. Another approach would be to combine both values, following the composition rule for singleton maps:

$$\{ k : x \} \cup \{ k : y \} = \{ k : x \cdot y \}$$

For this rule to be applicable, we need to make value composition possible, which means that the carrier of our map RA must map keys of a given type to values that belong to *another* RA. This leaves us with a new question: what value RA makes sense in the context of our problem? If two threads put different values for the same key, then what value should the key be mapped to? The answer to this question will depend on the timestamps of each insertion: the map should always store the value with the most recent timestamp. If both timestamps are equal, then both values will be prepended to the key's history, but their relative order will depend on the scheduler. Otherwise, the value with the most recent timestamp will become the new head of the key's vertical list, while the least recent insertion will only succeed if it gets scheduled first. In other words, combinations between values yield the value (or one of the values) with the *maximum* timestamp, which means that we will need to define a RA for the argmax operation.

To the best of our knowledge, our work is the first to formalize the argmax RA and use it in the verification of concurrent maps. We define the carrier for our RA as pairs between *sets of arguments* and *values*. We can then define the RA operator such that combining two pairs yields the pair with the maximum value. If the values are equal, then a new pair is returned, containing the same value and the union between both arguments. Finally, all combinations are defined as valid and a botZ element is added to the carrier to serve as unit. The argmax operator is thus defined such that for all sets of arguments (a and b) and values (i and j):

$$\begin{cases} (a, i) \cdot (b, j) = (b, j) \text{ if } i < j \\ (a, i) \cdot (b, i) = (a \cup b, i) \\ (a, i) \cdot \text{botZ} = (a, i) \end{cases}$$

In the context of maps with timestamped values, this novel RA will ensure that each key is associated to the timestamp at which its most recent update occurred, as well as to a set of values, which were all inserted with the key's associated timestamp. We require this set, rather than the actual value stored in memory, to keep track of every value that might be at the head of the key's vertical list. Since updates with the same timestamp will result in a non-deterministic ordering of operations, any of those updates can be the last one to be prepended to the list.

5 Specification for the Lazy JellyFish Skip List

Ghost state allows us to reason about concurrent maps in the abstract world of RAs. We now show how these abstractions can help us in reasoning about the physical state of a concurrent map and define a specification for its operations. We begin by describing the high-level specification and then present in detail its underlying definition.

5.1 Rules and Hoare Triples

We require a *representation predicate* to describe the known state of the map: due to the chosen implementation for this map, we refer to the representation predicate as IsSkipList. The first parameter of IsSkipList is a pointer to the head of the skip list. The second parameter is a map with partial knowledge of the map. Full knowledge of the map can be obtained by combining IsSkipList assertions through the SkipSep rule shown in Figure 2. The third parameter indicates whether we are in possession of the full view: it corresponds to a fraction ranging between 0 (exclusive) and 1 (inclusive). Full knowledge of the map's state corresponds to a fraction of 1, while splitting a view results in views with smaller fractions. The full view can only be obtained without sharing ownership, since excluding fractions of 0 ensures that it can only be split into fragments with a fraction lower than 1. The fourth parameter provides the required ghost names.

Our concurrent append-only map contains three public methods: new, get and put. In Figure 2, we show the Hoare triples for each operation using the IsSkipList predicate. The specification for new (SkipNew) is rather straightforward: no resources are needed as a precondition and creating the skip list returns the full fraction of an empty map. The other operations, however, possess more complex semantics.

Searches in our concurrent skip list are optimistic, meaning that, if some thread is searching for a key that another thread is inserting or updating, the searching thread might return an outdated result. For this reason, the specification for get (SkipGet) is only defined for the scenario where we have full ownership of the data structure. Since we know that no other thread is in possession of some map fragment, we can be certain that no concurrent write will interfere with the search. We can then prove that searching in the data structure for any key within the valid key range is equivalent to performing a lookup for the same key in the abstract map. Using the argmax RA for value composition, we can show that the key's set of values stored in the abstract map necessarily contains the value returned by the search.

The put method attempts to update a key with a given value and timestamp such that the key is within the valid key range. If the key has not yet been inserted into the map, then a new node with the given key, value and timestamp will be created and linked to all levels within its height range, updating the physical state of the map. To express this change

$$\textsc{SkipSep}\qquad \mathsf{IsSkipList}(p, M_1, q_1, \gamma) * \mathsf{IsSkipList}(p, M_2, q_2, \gamma) \dashv\vdash \mathsf{IsSkipList}(p, M_1 \cup M_2, q_1 + q_2, \gamma)$$

$$\textsc{SkipNew}\quad \frac{}{\{\,\mathsf{True}\,\}\quad \mathsf{new}\quad \{\,p.\ \exists\,\gamma.\ \mathsf{IsSkipList}(p, \emptyset, 1, \gamma)\,\}}$$

$$\textsc{SkipPut}\quad \frac{\mathsf{MIN} < k < \mathsf{MAX}}{\{\,\mathsf{IsSkipList}(p, M, q, \gamma)\,\}\quad \mathsf{put}\ p\ k\ v\ t\quad \{\,\mathsf{IsSkipList}(p, M \cup \{k : (\{v\}, t)\}, q, \gamma)\,\}}$$

$$\textsc{SkipGet}\quad \frac{\mathsf{MIN} < k < \mathsf{MAX}}{\{\,\mathsf{IsSkipList}(p, M, 1, \gamma)\,\}\quad \mathsf{get}\ p\ k\quad \left\{ v.\ \begin{array}{c} \mathsf{IsSkipList}(p, M, 1, \gamma) * ((v = \mathsf{None} * M[k] = \mathsf{None}) \vee \\ (\exists\, t, z, S.\ v = \mathsf{Some}(z, t) * M[k] = \mathsf{Some}(S, t) * z \in S)) \end{array} \right\}}$$

Figure 2. Specification for the lazy JellyFish skip list.

in the abstract state, the postcondition of SkipPut simply combines the view we have of the map with a singleton map that associates the key to a singleton set containing only the given value (*i.e.*, the only value that we know is associated to the key) and to the given timestamp. Since we know that the key is not in the map, adding a new key to our partial view is equivalent to adding it to the full view. While it is simpler to understand why this singleton composition works when we insert a new key, this update to the abstract map also works when we attempt to update an existing key.

Under the assumption that the key already exists in the map, we can infer that the abstract map results from a composition with some singleton map for that same key. Furthermore, while the key may not exist in our partial view, we know that updates to partial views are equivalent to updates to the full view. Thus, that singleton map, which we extract from the full view, will be combined with the singleton map from the update to our partial view. We can then apply the composition rule for singleton maps and obtain a new singleton map for the same key, with its value being the combination between the existing value and the new one. Using the argmax RA, this combination retains the values with the most recent timestamp, which is the intended behaviour for updates to a node's vertical list. Combining values in this manner ensures that the abstract map will always associate a key to every value that might be at the head of its vertical list, regardless of the update order.

5.2 Representation Predicate

To define the IsSkipList predicate, we first need to consider that operations on the skip list can be decomposed into local operations on the linked list of each level. This property allows us to reason about the correctness of the whole data structure by reasoning about each level independently. Thus, we can define a new predicate to describe the invariant resources for a given skip list level, while IsSkipList simply asserts ownership of the resources for all levels.

The IsSkipList predicate is thus defined using a unique invariant for each level, as shown in Figure 3. The invariants are represented by a solid border: the BotListInv predicate holds the shared resources for the bottom list, while the SublistInv predicate describes the upper levels. The parameter p should point to the left sentinel and the corresponding points-to assertion is made persistent ($\square$), since it will always point to the same node. The parameter γ is a list containing the names for the ghost variables of each level. The parameters M and q are used in the assertion within the dashed border to express the partial view of the map, as we will now see by describing the bottom list invariant.

5.3 Invariant for the Bottom List

Since the bottom list represents the full view of the abstract map, its shared state should reflect an equivalence between all entries of the abstract map and the existing physical nodes in memory. However, the IsSkipList predicate only expresses a partial view of the map, so we need relate *private* partial views to the *shared* full view. This can be accomplished with an authoritative RA, where the full view is an *authoritative* resource, while a partial view is a *fragment* resource.

Authoritative Ghost State. The relation between authoritative and fragment resources (preceded by $\bullet$ and $\circ$, respectively) is expressed through the following rule:

$$\boxed{\bullet\, a}^{\gamma} * \boxed{\circ\, f}^{\gamma} \vdash f \preccurlyeq a$$

If we consider a and f to be maps, then we can assert that f is a submap of a by owning a as the authoritative resource and f as one of its fragment resources. Thus, we can maintain inside the invariant an authoritative resource as the full view of the map, with each thread holding their own fragment resource as a partial view. To update a fragment resource, the same update must be applied to the authoritative resource, which can be accomplished by opening the invariant and obtaining that resource. In other words, a given thread's fragment resource can be seen as the combination of all its contributions to the map, meaning that combining all existing fragments should yield a resource containing the whole map. For this reason, the IsSkipList predicate asserts ownership of a fragment resource to reflect the partial view. To indicate whether we hold the full view, this partial view is associated to a fraction such that views can be combined using the rule:

$$\circ_{q_1} f_1 \cdot \circ_{q_2} f_2 = \circ_{q_1+q_2} (f_1 \cdot f_2)$$

Representation Predicate

$$\mathsf{IsSkipList}(p, M, q, \gamma) \triangleq \exists\, head.\; p \hookrightarrow_{\square} head * head.\mathsf{key} = \mathsf{MIN} *$$

$$\boxed{\circ_q\, M}^{\gamma_F^0} * \boxed{\mathsf{BotListInv}(head, \gamma^0)}^{\mathsf{levelN}(0)} * \mathop{\Huge *}_{i=1}^{\mathsf{HMAX}} \boxed{\mathsf{SublistInv}(i, head, \gamma^i, \gamma^{i-1})}^{\mathsf{levelN}(i)}$$

Invariants

$$\mathsf{BotListInv}(head, \gamma) \triangleq \exists\, M, S, L.$$

$$\boxed{\bullet\, M}^{\gamma_F} * \boxed{\bullet\, S}^{\gamma_A} * \boxed{\mathsf{KeyRange} \setminus S.\mathsf{keys}}^{\gamma_T} *$$

$$M.\mathsf{keys} = S.\mathsf{keys} * S \equiv_P L * \mathsf{Sorted}(L_{\mathsf{cat}}) *$$

$$\mathop{\Huge *}_{i=0}^{|L|} \begin{pmatrix} \mathsf{IsNext}(0, L_{\mathsf{cat}}[i], L_{\mathsf{cat}}[i+1]) * \\ \mathsf{HasLock}(0, L_{\mathsf{cat}}[i], \mathsf{InBotLock}) \end{pmatrix} *$$

$$\mathop{\Huge *}_{n \in S} \left(\exists\, v, vs.\; \begin{array}{l} n.\mathsf{val} \hookrightarrow_{1/2} v * v.\mathsf{val} \in vs * \\ M[n.\mathsf{key}] = \mathsf{Some}(vs, v.\mathsf{ts}) \end{array} \right)$$

$$\mathsf{SublistInv}(lvl, head, \Gamma, \gamma) \triangleq \exists\, S, L.$$

$$\boxed{\bullet\, S}^{\Gamma_A} * \boxed{\mathsf{KeyRange} \setminus S.\mathsf{keys}}^{\Gamma_T} *$$

$$S \equiv_P L * \mathsf{Sorted}(L_{\mathsf{cat}}) *$$

$$\mathop{\Huge *}_{i=0}^{|L|} \begin{pmatrix} \mathsf{IsNext}(lvl, L_{\mathsf{cat}}[i], L_{\mathsf{cat}}[i+1]) * \\ \mathsf{HasLock}(lvl, L_{\mathsf{cat}}[i], \mathsf{InSubLock}) \end{pmatrix} *$$

$$\mathop{\Huge *}_{n \in S} \left(\boxed{\circ\, \{n\}}^{\gamma_A} * \boxed{\{n.\mathsf{key}\}}^{\gamma_T} \right)$$

Lock Resources

$$\mathsf{InBotLock}(n, 0) \triangleq \exists\, s, succ.\; n.\mathsf{next}[0] \hookrightarrow_{1/2} s * s \hookrightarrow_{\square} succ * (succ = \mathsf{tail} \lor \exists\, v.\; succ.\mathsf{val} \hookrightarrow_{1/2} v)$$

$$\mathsf{InSubLock}(n, lvl) \triangleq \exists\, s.\; n.\mathsf{next}[lvl] \hookrightarrow_{1/2} s$$

where $\mathsf{levelN}(i)$ maps level i to its invariant namespace
$\mathsf{KeyRange} \triangleq \{k : \mathbb{Z} \mid \mathsf{MIN} < k < \mathsf{MAX}\}$
$\mathsf{tail} \triangleq (\mathsf{MAX}, \mathsf{NULL}, \mathsf{NULL}, \mathsf{NULL})$
$L_{\mathsf{cat}} \triangleq [head] +\!\!+\, L +\!\!+\, [\mathsf{tail}]$
$\mathsf{IsNext}(lvl, pred, succ) \triangleq \exists\, s.\; pred.\mathsf{next}[lvl] \hookrightarrow_{1/2} s * s \hookrightarrow_{\square} succ$
$\mathsf{HasLock}(lvl, node, R) \triangleq \exists\, \gamma, l.\; node.\mathsf{lock}[lvl] \hookrightarrow_{\square} l * \mathsf{IsLock}(\gamma, l, R(node, lvl))$
IsLock is the predicate for the lock invariant

Figure 3. Definition of the representation predicate and invariants.

Although fractions are necessary to express partial views accurately, the authoritative RA by itself is more adequate for dealing with set membership assertions. Ownership of a fragment containing a singleton set entails that the singleton's element belongs to the authoritative set, without requiring any knowledge about the rest of the set or what fraction of the set we currently hold. This property of the authoritative RA is useful for verifying concurrent traversals when we do not have full ownership of the data structure. Traversals are a recursive procedure where we loop over some (or all) nodes of the set until the desired node is found. For all node visits, the invariant is that the current node is one that belongs to the set. Thus, a set membership assertion for the visited node is necessary as a precondition to prove the correctness of a traversal that passes through that node. In other words, the ghost state for the bottom list should also contain an authoritative set of nodes, matching the physical nodes of the skip list.

Invariant Definition. The BotListInv predicate in Figure 3 shows the invariant for the bottom list. Since the shared resources can be updated, the invariant is existentially quantified by a map M, so as to not retain a constant inside the invariant. Combining all partial views will give us the full map, so no information is lost; we only require that such a map exists as an authoritative resource in the invariant.

The invariant also contains an authoritative resource S for the set of nodes, enforcing that the set must contain the same keys as the abstract map M. These keys are also used to assert ownership of a set of ghost tokens, discussed in Section 5.4. However, due to the unordered nature of sets, we are unable to express that the physical list (including the sentinels) must always remain sorted. For this reason, the invariant is also existentially quantified by a list L containing the same nodes as S. The chain of successor pointers created by the nodes should thus reflect the order of this list.

For each contiguous pair of nodes in L_{cat} (L concatenated with both sentinels), the IsNext predicate asserts that the first node of the pair should point to the second one. We require two distinct points-to assertions to relate a node with its successor: one for the node's array entry and another for the successor pointer it stores. Since each key can be present in more than one level, each node holds an array of successors, whose entries can be overwritten as new nodes are inserted into the data structure. Furthermore, it is possible for different nodes to have the same successor in different levels, meaning that each array entry should store a pointer instead of the node itself. To ensure that the successor pointer remains unchanged, we resort to the persistent points-to assertion, granting read-only access to its contents.

The value of each node should also reflect the key-value pairs from the abstract map M. However, M contains a set of possible values for each key, while a physical node can only store one actual value in memory. So, for each node we assert that a lookup for the node's key in M should return the node's timestamp, as well as a set of values containing the node's real value.

Lock Resources. Finally, we need to consider that both the value and the successor of every node are allowed to be updated. The InBotLock predicate, which describes the resources protected by a node's lock, contains a fraction of the points-to assertion for the node's array entry, as well as a fraction of the points-to assertion for its successor's value; the remaining fractions are kept in the bottom list invariant. The persistent assertion for the successor pointer is also stored inside the lock invariant so as to connect both assertions. The locks themselves are also stored in an array, but since they remain the same during the node's lifetime, the points-to assertion for each lock may also be made persistent.

Vertical List. While BotListInv ties every key to its current value, the previous values are never accounted for in the invariant. As such, the invariant alone does not guarantee the correctness of the version control mechanism employed by value updates on the data structure. We ensure that the history of values for an updated node is preserved by proving the Hoare triple shown in Figure 4. The head of the vertical list stored in *node*.val may differ from its initial value depending on the timestamps of the new and current values.

If the new timestamp is less recent than the current timestamp (**then** branch), no update should occur and the vertical list should remain unchanged. Otherwise (**else** branch), the new value and timestamp should be prepended to the vertical list's head. Being at the head of the vertical list, the new value should now point to the previous value, ensuring that the history of values is not forgotten. By making this points-to assertion persistent, we guarantee that the predecessor for a prepended value is immutable, yielding an immutable chain (or history) of values by construction. In other words, an update either returns the same immutable history of values or an immutable extension of it. Discriminating both cases using timestamps further ensures that timestamps within a given history grow monotonically, avoiding inconsistent timelines.

$$
\left\{ \begin{array}{c} \boxed{\circ_q M}^{\gamma_F} * \boxed{\mathsf{BotListInv}(head, \gamma)}^{\mathsf{levelN}(0)} * \\ \boxed{\circ \{node\}}^{\gamma_A} * node.\mathsf{val} \hookrightarrow_{1/2} val \end{array} \right\}
$$

$$
\text{update } node\ v\ t
$$

$$
\left\{ \begin{array}{c} \boxed{\circ_q M \cup \{node.\mathsf{key} : (\{v\}, t)\}}^{\gamma_F} * \\ \\ \textbf{if } t < val.\mathsf{ts} \textbf{ then } node.\mathsf{val} \hookrightarrow_{1/2} val \\ \textbf{else } \exists\, p.\ node.\mathsf{val} \hookrightarrow_{1/2} (v, t, p) * p \hookrightarrow_{\square} val \end{array} \right\}
$$

Figure 4. Specification for the update procedure.

5.4 Invariant for Sublists

We have seen how to reason about the skip list's underlying map using the invariant for the bottom list. We now turn our attention to its sublists, which serve to guide the search through the skip list towards the intended node in the bottom list. The relation between consecutive levels in the skip list is our main concern in defining the sublist invariant.

Sublist Relation. The key property behind the skip list design is that each level contains a sublist of the list contained in the level below it. In other words, when we stop the search in one level, we can continue the search in the next level from the same key without visiting its predecessors. Therefore, concluding a traversal in one level should provide the necessary context to verify the traversal in the next level.

As we discussed previously, to verify a traversal we require a set membership assertion for the node we are currently visiting, which can come in the form of a fragment resource from the authoritative set in the intended level. A node in the upper level should thus contain a fragment from the lower level, so that the succeeding traversal may be proven correct. Figure 5 captures this idea, associating the fragments of each node in level k+1 to the authoritative resource in level k.

The lower fragment can be obtained by updating the authoritative resource when inserting the node in the corresponding level. Since insertions are performed bottom-up, we obtain the fragment from the lower level before inserting in the upper level. The node can thus be inserted in the upper level, storing the fragment inside the level's invariant and returning a new fragment from the level's authoritative set.

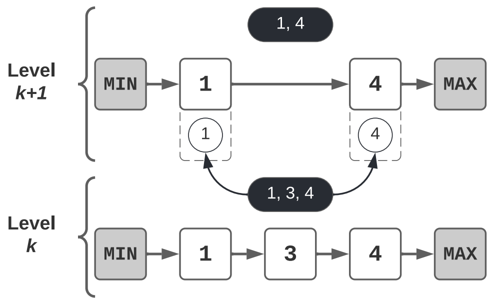

Figure 5. Preserving the sublist relation with fragments (white round boxes) from the authoritative resource (black round boxes).

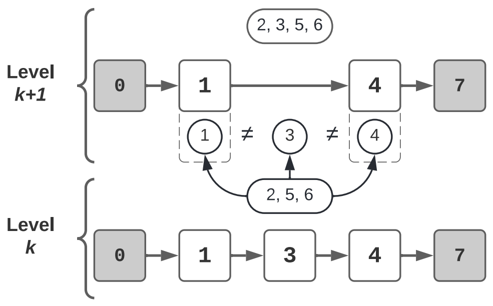

Figure 6. The available tokens in each level (round boxes) are all valid keys except the ones in the corresponding list. Having been inserted in level k, key 3 can be inserted in level k+1, since the token for key 4 is a different token.

Ghost Tokens. While the lower level fragments are required to express the sublist relation, verifying the insertion procedure within a given level requires additional information. When inserting in the bottom list, we check if the key from the successor node is equal to the key we want to insert; this verification step ensures that only one thread inserts the key in all intended levels. Since the absence of the key in the bottom level implies its absence in all levels, the inserting thread can insert in any upper level without checking the key of the successor in that level. To prove that the insertion is correct, however, we must still show that the new key and the successor's key are distinct, even though it is not made explicit in the code. We can take advantage of the fact that these insertions occur bottom-up to reason about this issue.

The idea is for every node to hold a token associated to its key, obtained from the lower level. Using the *disjoint* set union RA, each level contains KeyRange as the initial set of tokens. A token for a given key is merely a singleton set containing that key and can be obtained by extracting the key from the set of available tokens. Since we perform the insertion in the lower level before the upper level, we extract the token from the former before attempting to insert the node in the latter. As the RA requires sets to be disjoint, the token we obtain from inserting in the lower level must refer to a different key than the lower level token held by the successor at the upper level. Therefore, the keys are distinct and the node can be inserted in the upper level, without explicitly checking the keys in the code. The token is then stored in the invariant and a new token is extracted from the invariant to use in the next insertion. A visual representation of the token system can be seen in Figure 6.

Invariant Definition. The sublist invariant SublistInv can be seen in Figure 3, differing from BotListInv in some aspects. Map logic is no longer required for the upper levels, meaning that the authoritative map is removed and the map lookup assertions for each node are replaced with the associated fragment and token. Additionally, since values are only updated upon reaching the bottom level, acquiring upper level locks does not grant the required resources to overwrite a node's value, meaning that InSubLock only contains the node's array entry for the successor pointer.

6 Related Work

Iris has been used to reason about concurrent data structures, verifying (1) a contextual refinement of other simpler concurrent implementations [18, 19], (2) correctness under a weak memory model [14] and (3) template algorithms for search structures [13]. (2) and (3) focus on proving logical atomicity of operations on concurrent data structures. Logical atomicity allows clients to treat non-atomic operations as if they were atomic, verifying a stronger correctness criterion.

The only work in Iris which we are aware of reasoning about concurrent skip lists is that of Tassarotti and Harper [17]. They extended Iris to support probabilistic reasoning, proving correctness and temporal properties of a two-level concurrent append-only skip list, which is a probabilistic data structure. While their work focused more on probabilistic properties, we avoid reasoning about such aspects, proving correctness of the data structure independently of the height distribution. Our mechanization is deeply influenced by theirs, generalizing their arguments to an arbitrary number of levels and simplifying the required ghost state.

Considering other verification frameworks, we highlight: (1) the TSL theory for reasoning about skip lists with an arbitrary number of levels [16], (2) the verification in Agda of correctness properties of authenticated append-only skip lists [2] and (3) the skip list entry in the Archive of Formal Proofs [4]. These works, however, deal with sequential skip lists, so they tackle inherently different concerns from the ones we have discussed in this work. For concurrent skip lists, Abdulla *et al.* [1] present the only other work we are aware of providing mechanized proofs, automating their reasoning in a self-developed tool.

da Rocha Pinto *et. al.* [3] proposed a map specification based on each key-value pairing. Partial ownership of concrete entries of the map is achieved through fractions, while protocol tags serve to indicate how operations on shared resources should be composed. Xiong *et. al.* [20] expand on this work by defining an alternative key-value specification based on a logically atomic specification of the map. Protocols can be defined on top of this specification through suitable algebras. Although their work was not mechanized, we argue that our work could benefit from their approach.

7 Conclusion

We presented a lock-based variant of the JellyFish skip list and showed that its implementation satisfies a concurrent map specification. Conflicts on concurrent updates are handled through timestamps, which we reason about using a novel resource algebra for the argmax operation. We match the abstract state of the map with the physical state of the skip list by only applying map logic to the bottom level. Our proofs are generalized for any number of levels and height distribution by reasoning about each level independently, tying consecutive levels by ghost state. This work contributes to the understanding of complex list-based structures, providing a new approach for reasoning about concurrent maps.

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2018. Fragment Abstraction for Concurrent Shape Analysis. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, Cham, 442–471. https://doi.org/10.1007/978-3-319-89884-1_16

[2] Victor Cacciari Miraldo, Harold Carr, Mark Moir, Lisandra Silva, and Guy L. Steele Jr. 2021. Formal Verification of Authenticated, Append-Only Skip Lists in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 122–136. https://doi.org/10.1145/3437992.3439924

[3] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark Wheelhouse. 2011. A Simple Abstraction for Complex Concurrent Indexes. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 845–864. https://doi.org/10.1145/2048066.2048131

[4] Max W. Haslbeck and Manuel Eberl. 2020. Skip Lists. *Archive of Formal Proofs* (Jan. 2020). https://isa-afp.org/entries/Skip_Lists.html, Formal proof development.

[5] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems (Lecture Notes in Computer Science, Vol. 3974)*, James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer (Eds.). Springer, Berlin, Heidelberg, 3–16. https://doi.org/10.1007/11795490_3

[6] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *International Colloquium on Structural Information and Communication Complexity (Lecture Notes in Computer Science, Vol. 4474)*, Giuseppe Prencipe and Shmuel Zaks (Eds.). Springer, Berlin, Heidelberg, 124–138. https://doi.org/10.1007/978-3-540-72951-8_11

[7] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

[8] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (Nov. 2018), e20. https://doi.org/10.1017/S0956796818000151

[9] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

[10] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (July 2018), 30 pages. https://doi.org/10.1145/3236772

[11] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, Berlin, Heidelberg, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[12] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

[13] Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 181–196. https://doi.org/10.1145/3385412.3386029

[14] Glen Mével and Jacques-Henri Jourdan. 2021. Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 66 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473571

[15] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[16] Alejandro Sánchez and César Sánchez. 2014. Formal Verification of Skiplists with Arbitrary Many Levels. In *Automated Technology for Verification and Analysis (Lecture Notes in Computer Science, Vol. 8837)*, Franck Cassez and Jean-François Raskin (Eds.). Springer, Cham, 314–329. https://doi.org/10.1007/978-3-319-11936-6_23

[17] Joseph Tassarotti and Robert Harper. 2019. A Separation Logic for Concurrent Randomized Programs. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 64 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290377

[18] Simon Friis Vindum and Lars Birkedal. 2021. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 76–90. https://doi.org/10.1145/3437992.3439930

[19] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 100–115. https://doi.org/10.1145/3497775.3503689

[20] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, Berlin, Heidelberg, 964–990. https://doi.org/10.1007/978-3-662-54434-1_36

[21] Jeseong Yeon, Leeju Kim, Youil Han, Hyeon Gyu Lee, Eunji Lee, and Bryan S. Kim. 2020. JellyFish: A Fast Skip List with MVCC. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 134–148. https://doi.org/10.1145/3423211.3425672