Verifying Concurrent Search Structure Templates

Siddharth Krishna
Microsoft Research
Cambridge, UK
siddharth@cs.nyu.edu

Nisarg Patel
New York University
USA
nisarg@nyu.edu

Dennis Shasha
New York University
USA
shasha@cims.nyu.edu

Thomas Wies
New York University
USA
wies@cs.nyu.edu

Abstract

Concurrent separation logics have had great success reasoning about concurrent data structures. This success stems from their application of modularity on multiple levels, leading to proofs that are decomposed according to program structure, program state, and individual threads. Despite these advances, it remains difficult to achieve proof reuse across different data structure implementations. For the large class of *search structures*, we demonstrate how one can achieve further proof modularity by decoupling the proof of thread safety from the proof of structural integrity. We base our work on the *template* algorithms of Shasha and Goodman that dictate how threads interact but abstract from the concrete layout of nodes in memory. Building on the recently proposed flow framework of compositional abstractions and the separation logic Iris, we show how to prove correctness of template algorithms, and how to instantiate them to obtain multiple verified implementations.

We demonstrate our approach by mechanizing the proofs of three concurrent search structure templates, based on link, give-up, and lock-coupling synchronization, and deriving verified implementations based on B-trees, hash tables, and linked lists. These case studies include algorithms used in real-world file systems and databases, which have been beyond the capability of prior automated or mechanized verification techniques. In addition, our approach reduces proof complexity and is able to achieve significant proof reuse.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Separation logic**; **Shared memory algorithms**.

Keywords: template-based verification, concurrent data structures, flow framework, separation logic

ACM Reference Format:
Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386029

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386029

1 Introduction

Modularity is as important in simplifying formal proofs as it has been for the design and maintenance of large systems. There are three main types of modular proof techniques: (i) Hoare logic [32] enables proofs to be compositional in terms of program structure; (ii) separation logic [48, 54] allows proofs of programs to be local in terms of the state they modify; and (iii) thread modular techniques [30, 33, 50] allow one to reason about each thread in isolation.

Concurrent separation logics [10, 11, 16, 18, 19, 21, 24, 27, 37, 46, 47, 58, 60] incorporate all of the above techniques and have led to great progress in the verification of practical concurrent data structures, including recent milestones such as a formal proof of the B-link tree [15]. Proofs of such real-world data structures, however, remain large, complex, paper-based, and verifiable only by hand.

An important reason why existing proofs, such as that of the B-link tree, are still so complicated is that they argue simultaneously about thread safety (i.e., how threads synchronize) and memory safety (i.e., how data is laid out in the heap). We contend that safety proofs should instead be decomposed so as to reason about these two aspects independently. When verifying thread safety we should abstract from the concrete heap structure used to represent the data and when verifying memory safety we should abstract from the concrete thread synchronization algorithm. Adding this form of abstraction as a fourth modular proof technique to our arsenal promises reusable proofs and simpler correctness arguments, which in turn aids proof automation.

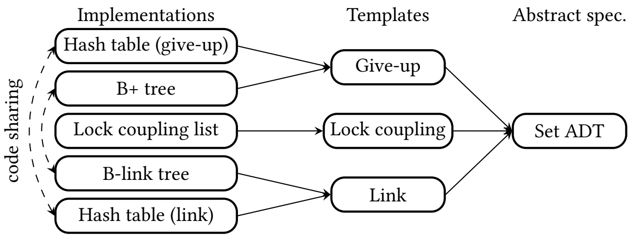

Figure 1. The structure of our proofs.

As an example, consider the B-link tree, which uses the link-based technique for thread synchronization. The following analogy [57] captures the essence of this technique. Bob wants to borrow book k from the library. He looks at the library's catalog to locate k and makes his way to the appropriate shelf n. Before arriving at n, Bob gets caught up in a conversation with a friend. Meanwhile, Alice, who works at the library, reorganizes shelf n and moves k as well as some other books to n'. She updates the library catalog and also leaves a sticky note at n indicating the new location of the moved books. Finally, Bob continues his way to n, reads the note, proceeds to n', and takes out k. The synchronization protocol of leaving a note (the *link*) when books are moved ensures that Bob can find k rather than thinking that k is nowhere in the library. However, when arguing the correctness of this protocol, we do not need to reason about how books are stored in shelves or how the catalog is organized.

The library patron corresponds to a thread searching for and performing an operation on the key k stored at some node n in the B-link tree and the librarian corresponds to a thread performing a split operation involving nodes n and n'. As in our library analogy, the synchronization technique of creating a forward pointer (the link) when nodes are split works independently of how data is stored within each node and how these are organized in memory (e.g. as a B-tree or hash table). Hence, it applies to vastly different concrete data structures. Our goal is to verify the correctness of *template algorithms* once and for all so that their proofs can be reused across different data structure implementations.

The challenge in achieving this *algorithmic proof modularity* is in reconciling the template abstractions with the proof technique of reasoning locally about modifications to the heap as in separation logic (SL), which is itself critical to obtaining simple proofs that are easy to mechanize. The proof of the link technique depends on certain invariants about the paths that a search for a key k follows in the data structure graph. However, with the standard heap abstractions used in separation logic (e.g. inductive predicates), it is hard to express these invariants independently of the invariants that capture how the data structure is represented in memory. Consequently, existing proofs such as the one of the B-link tree in [15] intertwine the synchronization invariants and the memory invariants, which makes the proof complex, hard to mechanize, and difficult to reuse on different structures.

Template Proof Methodology. This paper shows how to adapt and combine recent advances in compositional abstractions and separation logic in order to achieve the envisioned algorithmic proof modularity for the important class of *concurrent search data structures.*

We base our work on the template algorithms for concurrent search structures by Shasha and Goodman [57], who identified the key invariants needed for decoupling reasoning about synchronization and memory representation for such data structures. The second ingredient is the concurrent separation logic Iris [34, 35, 37, 39]. We show how to capture the high-level idea of [57] in terms of a new Iris resource algebra, yielding a general methodology for modular verification of concurrent search structures. This methodology independently verifies that (1) the template algorithm satisfies the (atomic) abstract specification of search structures assuming that node-level operations maintain certain shape-agnostic invariants and (2) the implementations of these operations for each concrete data structure maintains these invariants.

A key technical improvement over [57] is that our new resource algebra, in combination with Iris' notion of *atomic triples* [16, 36, 37], avoids explicit reasoning about execution histories and low-level programming language semantics. Moreover, it yields a local proof technique that eliminates the need to reason explicitly about the global abstract state of the data structure. The latter crucially relies on the recently proposed *flow framework* [40, 41], the final ingredient of our methodology. The flow framework provides an SL-based abstraction mechanism that allows one to reason about global inductive invariants of general graphs in a local manner. Using this framework, we can do SL-style reasoning about the correctness of a concurrent search structure template while abstracting from the specific low-level heap representation of the underlying data structure.

We note that our methodology generalizes to any data structure indexed by keys, including implementations of sets, maps, and multisets (but not, e.g., queues and stacks). Our approach of separating concurrency templates and heap implementations requires the data structure to have an abstract state (e.g. as mathematical set or map) with a certain algebraic structure: we need to be able to decompose the abstract state into local abstract states that are *disjoint* in some sense. Moreover, composition of abstract states needs to be associative, commutative, and homomorphic to composition of heap graphs. For instance, consider a binary search tree representing a mathematical map where each tree node stores a single key/value pair. If one arbitrarily splits the tree's heap graph into disjoint subgraphs, then these subgraphs represent disjoint mathematical maps whose union yields the map represented by the original composed heap graph. We conjecture that all search structure implementations follow these composition principles.

Case Studies. We demonstrate our methodology by mechanizing the correctness proofs of three template algorithms for concurrent search structures based on the link, the give-up, and the lock-coupling technique of synchronization (Fig. 1). For these, we derive concrete verified implementations based on B-trees, hash tables, and sorted linked lists, resulting in five different data structure implementations. §4 discusses the proof of the link template in detail. Section §5 presents a summary of the effort required by our verification approach.

A key advantage of our approach is that we can perform *sequential* reasoning when we verify that an implementation is a valid template instantiation. We therefore perform only the template proofs in Iris/Coq and verify the implementations using the automated deductive verification tool GRASShopper [51, 52]. The automation provided by GRASShopper enables us to bring the proofs of highly complicated implementations such as B-link trees within reach.

Our proofs include a mechanization of the meta-theory of the flow framework presented in [41], carried out independently in both GRASShopper and Iris/Coq. The verification efforts in the two systems are hence each fully self-contained. The template proofs done in Iris are parametric to any possible correct implementation of the node-level operations. The specifications assumed in Iris match those proved in GRASShopper. However, we note that there is no formal connection between the proofs done in the two systems. If one desires end-to-end certified implementations, one can perform both template and implementation proofs in Iris/-Coq (albeit with substantial additional effort). Performing the proofs completely in GRASShopper or a similar SMT-based verification tool would require additional tooling effort to support reasoning about Iris-style resource algebras.

The proofs we obtain are more modular, simpler, and more reusable than existing proofs of such concurrent data structures. Our experience is that adapting our technique to a new template algorithm and instantiating a template to a new data structure takes only a few hours of proof effort.

Summary. The contributions of this paper are:

- We propose a new methodology for verifying concurrent search structure templates that enables proofs to be compositional in terms of program structure and state, and exploit thread and algorithmic modularity. The technique applies to any data structure that is indexed by keys, including implementations of sets, maps, and multisets.
- We mechanically prove several complex real-world data structures such as the B-link tree that are beyond the capability of existing techniques for mechanized or automated formal proofs. The resulting proofs are relatively simple and reusable.
- We mechanize the meta-theory of the flow framework [41] within Coq and GRASShopper, and show how to use it to construct a general parametric resource algebra for flow-based proofs in Iris. The possible uses of this effort go beyond the specific application considered in this paper.

2 Overview

A search structure is a key-based store that implements three basic operations: search, insert, and delete. We refer to a thread seeking to search for, insert, or delete a key k as an operation on k, and to k as the *operation key*. For simplicity, the presentation here treats search structures as containing only keys (i.e. as implementations of mathematical sets), but all our proofs can be easily extended to consider search structures that store key-value pairs.

2.1 B-link Trees

The B-link tree (Fig. 2) is an implementation of a concurrent search structure based on the B-tree. A B-tree is a generalization of a binary search tree, in that a node can have more than two children. In a binary search tree, each node contains a key k_0 and up to two pointers y_l and y_r. An operation on k takes the left branch if $k < k_0$ and the right branch otherwise. A B-tree generalizes this by having l sorted keys $k_0, \ldots, k_{l-1}$ and $l + 1$ pointers $y_0, \ldots, y_l$ at each node, such that $B \leq l + 1 < 2B$ for some constant B. At internal nodes, an operation on k takes the branch y_i if $k_{i-1} \leq k < k_i$. In the most common implementations of B-trees (called B+ trees), the keys are stored only in leaf nodes; internal nodes contain "separator" keys for the purpose of routing only. When an operation arrives at a leaf node n, it proceeds to insert, delete, or search for its operation key in the keys of n. To avoid interference, each node has a lock that must be held by an operation before it reads from or writes to the node.

When a node n becomes full, a separate maintenance thread performs a split operation by transferring half its keys (and pointers, if it is an internal node) into a new node n', and adding a link to n' from the parent of n. A concurrent algorithm needs to ensure that this operation does not cause concurrent operations at n looking for a key k that was transferred to n' to conclude that k is not in the structure. The B-link tree solves this problem by linking n to n' and storing a key k' (the key in the gray box in the figure) that indicates to concurrent operations that the key k can be reached by following the link edge if $k > k'$. To reduce the time the parent node is locked, this split is performed in two steps: (i) a half-split step that locks n, transfers half the keys to n', and adds a link from n to n' and (ii) a complete-split performed by a separate thread that takes half-split nodes n, locks the parent of n, and adds a pointer to n'.

Fig. 2 shows the state of a B-link tree where node y_2 has been fully split, and its parent n has been half split. The full

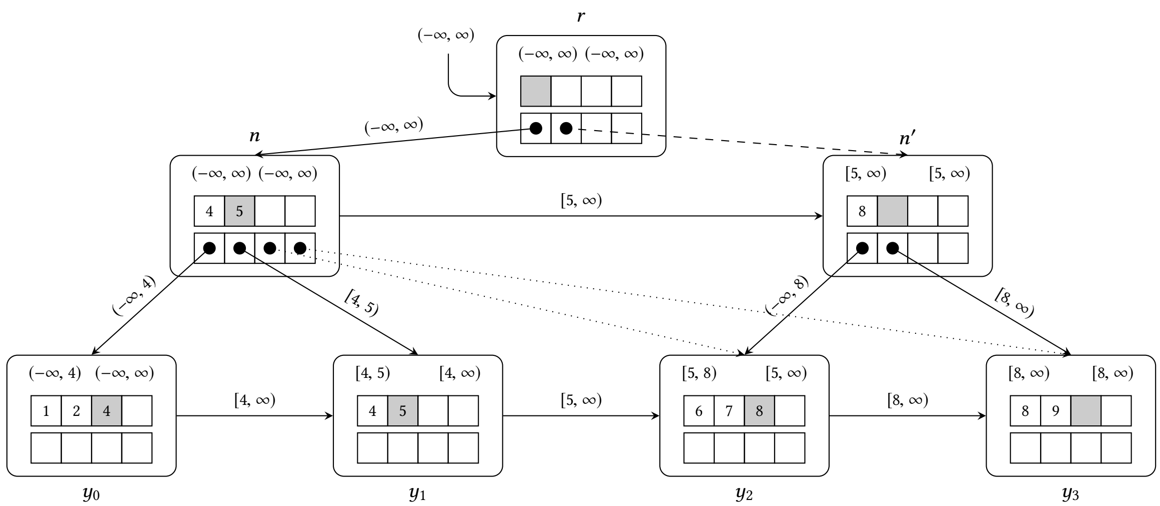

Figure 2. An example B-link tree state in the middle of of a split. Node n was full, and has been half-split and children y_2 and y_3 have been transferred to new node n' (old edges are shown with dotted lines), but the complete-split has yet to add n' to the parent r (the dashed edge). Each node contains an array of keys $k_0, \ldots, k_{l-1}$ in the middle, an array of pointers $y_0, \ldots, y_l$ in the bottom, the inset (see §4.3) in the top left and the inreach (§4.3) in the top right. (The key in the gray box is not considered part of the contents and determines when to take the link edge.) Each edge is labelled by its edgeset (§2.2), and the label with a curved arrow to the top-left of the root is its inflow (explained in §4.3).

split of y_2 moved keys $\{8, 9\}$ to a new node y_3, added a link edge, and added a pointer to y_3 in its (old) parent n. However, this caused n to become full, resulting in a half split that moved its children $\{y_2, y_3\}$ to a new node n' and added a link edge to n'. The key 5 in the gray box in n directs operations on keys $k \geq 5$ via the link edge to n'. The figure shows the state after this half split but before the complete-split when the pointer of n' will be added to r.

2.2 Abstracting Search Structures using Edgesets

The link technique is not restricted to B-trees: consider a hash table implemented as an array of pointers, where the ith entry refers to a bucket node that contains an array of keys $k_0, \ldots, k_l$ that all hash to i. When a node n gets full, it is locked, its keys are moved to a new node n' with twice the capacity, and n is linked to n'. Again, a separate operation locks the main array entry and updates it from n to n'.

While these two data structures look completely different, the main operations of search, insert, and delete follow the same abstract algorithm. In both, there is some local rule by which operations are routed from one node to the next, and both introduce link edges when keys are moved to ensure that no other operation loses its way.

To concretize this intuition, let the *edgeset* of an edge (n, n'), written $\mathrm{es}(n, n')$, be the set of operation keys for which an operation arriving at a node n traverses (n, n'). The B-link tree in Fig. 2 labels each edge with its edgeset; the edgeset of (n, y_1) is $[4, 5)$ and the edgeset of the link edge (y_0, y_1) is $[4, \infty)$. Note that 4 is in the edgeset of (y_0, y_1) even though an operation on 4 would not normally reach y_0. This

```
let rec traverse n k =
  lockNode n;
  match findNext n k with
  | None -> n
  | Some n' ->
     unlockNode n;
     traverse n' k

let rec cssOp ω r k =
  let n = traverse r k in
  match decisiveOp ω n k with
  | None -> unlockNode n;
     cssOp ω r k
  | Some res -> unlockNode n;
     res
```

Figure 3. The link template algorithm, which can be instantiated to the B-link tree algorithm by providing implementations of helper functions `findNext` and `decisiveOp`. `findNext n k` returns `Some n'` if $k \in \mathrm{es}(n, n')$ and `None` if there exists no such n'. `decisiveOp n k` performs the operation ω (either search, insert, or delete) on k at node n.

is deliberate. In order to make edgeset a local quantity, we say $k \in \mathrm{es}(n, n')$ if an operation on k would traverse (n, n') assuming it somehow found itself at n. In the hash table, assuming there exists a global root node, the edgeset from the root to the ith array entry is $\{k \mid hash(k) = i\}$. The edgeset from an array entry to the bucket node is the set of all keys KS, as is the edgeset from a deleted bucket node to its replacement.

2.3 The Link Template Algorithm

Fig. 3 lists the link template algorithm [57] that uses edgesets to describe the algorithm used by all core operations for both B-link trees and hash tables in a uniform manner. The algorithm assumes that an implementation provides certain primitives or helper functions, such as `findNext` that finds the next node to visit given a current node n and an operation key k, by looking for an edge (n, n') with $k \in \mathrm{es}(n, n')$. For the B-link tree, `findNext` does a binary search on the keys in a node to find the appropriate pointer to follow. For the hash table, when at the root `findNext` returns the edge to the array element indexed by the hash of the key, and at bucket nodes it follows the link edge if it exists. The function `cssOp` can be used to build implementations of all three search structure operations by implementing the helper function `decisiveOp` to perform the desired operation (read, add, or remove) of key k on the node n.

An operation on key k starts at the root r, and calls a function `traverse` on line 9 to find the node on which it should operate. `traverse` is a recursive function that works by following edges whose edgesets contain k (using the helper function `findNext` on line 3) until the operation reaches a node n with no outgoing edge having an edgeset containing k. Note that the operation locks a node only during the call to `findNext`, and holds no locks when moving between nodes. `traverse` terminates when `findNext` does not find any n' such that $k \in \mathrm{es}(n, n')$, which, in the B-link tree case means it has found the correct leaf to operate on. At this point, the thread performs the decisive operation on n (line 10). If the operation succeeds, then `decisiveOp` returns `Some res` and the algorithm unlocks n and returns `res`. In case of failure (say an insert operation encountered a full node), the algorithm unlocks n, gives up, and starts from the root again.

If we can verify this link template algorithm with a proof that is parameterized by the helper functions, then we can reuse the proof across diverse implementations. In the rest of this paper, we show how to do this using the flow framework in the Iris separation logic.

3 A Brief Introduction to Flows

This section describes the flow framework [40, 41], a separation logic based approach for specifying and reasoning about unbounded data structures. We give an informal description of the framework and demonstrate flow-based reasoning on a simple list example (for a more formal introduction, see [40, 41]). We use the fundamental flow framework [41] in this paper as it simplifies our proofs.

Separation logic is based on the powerful concept of *local reasoning*. However, many important properties of data structure graphs depend on non-local information. For instance, one cannot express the property that a graph is a tree by conjoining per-node invariants. The flow framework allows one to specify global graph properties in terms of node-local invariants by extending the graph with a *flow* – a function from nodes to values from some *flow domain*. These flow values are constrained to satisfy the *flow equation*, i.e. they must be a fixpoint of a set of algebraic equations induced by the entire graph (thereby allowing one to capture global constraints at the node level). When modifying a

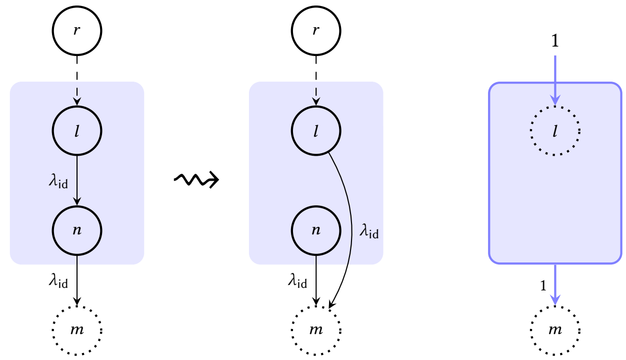

Figure 4. Unlinking a node n from a list by swinging the pointer from its predecessor l to its successor m. Edges are labeled with edge labels for path counting (λ_0 edges omitted). The interface of the blue region $\{l, n\}$ is shown on the right, and is preserved by this update.

graph, the framework allows one to perform a local proof that flow-based invariants are maintained via the notion of a *flow interface*. This is an abstraction of a graph region that specifies the flow values entering and exiting the region; if these are preserved then the flow values of the rest of the graph will be unchanged.

The rest of this section illustrates these concepts by considering some simple examples. Suppose we have a graph G on a set of nodes N and we want to express the property that it is a list rooted at some node r as a local condition on each node. To do this, we need to know some global information at each node: for instance, suppose there existed a function pc that mapped each node n to the number of paths from r to n.[1] If for every node n, $\mathrm{pc}(n) = 1$ and n has at most one outgoing edge (both node-local assertions) then we know that G must be a list rooted at r.

This path-counting function pc is an example of a flow because it can be defined as a solution to the flow equation:

$$\forall n \in N.\ \mathit{fl}(n) = \mathit{in}(n) + \sum_{n' \in N} e(n', n)(\mathit{fl}(n')) \qquad \text{(FlowEqn)}$$

This is a fixpoint equation on a function $\mathit{fl}\colon N \to M$, where M is a flow domain, *in* is an *inflow* that specifies the default-/initial flow value of each node, and e is a mapping from pairs of nodes to *edge functions* that determine how the flow of one node affects the flow of its neighbor. The flow framework works with directed partial graphs that are augmented with a flow, called flow graphs. A flow graph is a tuple $H = (N, e, \mathit{fl})$ consisting of a finite set of nodes $N \subseteq \mathfrak{N}$ ($\mathfrak{N}$ is potentially infinite), a mapping from pairs of nodes to edge functions $e\colon N \times \mathfrak{N} \to E$, and a function fl such that (FlowEqn) is satisfied for some inflow *in*. Flow graph composition $H_1 \odot H_2$ is a partial operator that is a disjoint union of the nodes, edges,

[1] We assume a definition of pc where $\mathrm{pc}(r) = 1$ even in acyclic graphs, this is because typically we are interested in the reachability of heap nodes from an external stack pointer.

and flow values and is defined only if the resulting graph continues to satisfy (FlowEqn).

In the case of the path-counting flow, the flow domain M is $\mathbb{N}$, the inflow is $in(n) := (n = r \ ?\ 1 : 0)$, and the edge function $e(n, n')$ is the identity function $\lambda_{\mathrm{id}} := (\lambda m.\ m)$ for all edges (n, n') in G and the zero function $\lambda_0 := (\lambda m.\ 0)$ otherwise. The flow equation then reduces to the familiar constraint that the number of paths from r to n, $\mathrm{pc}(n)$, equals 1 if $n = r$ else 0, plus the sum of the number of paths to all n' that have an edge to n.

The problem with assuming each node knows a flow value that satisfies some global constraint over the entire graph is that when a program modifies the graph, it can be hard to show that the flow-based invariants are maintained. In particular, when the program modifies a small part of the graph, say by modifying a single edge, we would ideally like to prove that the flow invariants are preserved by only reasoning about a small region around the modified edge. The flow framework enables such local proofs by means of an abstraction of flow (sub)graphs called flow interfaces.

Consider the simple example of a singly-linked list deletion procedure that unlinks[2] a given node n from the list (Fig. 4). The program swings the pointer from n's predecessor l to n's successor m. We use the path-counting flow and the flow-based local constraints described above to express the invariant that the graph is a list (we show how to formally express this later). For a flow graph H over the path-counting flow domain, modifying a single edge (n, n') can potentially change the flow (the path-count) of every node reachable from n. However, notice that the modification shown in Fig. 4 changes (l, n) to (l, m) where m is the successor of n. This preserves the flow of every node outside the modified subgraph $H_1 = H|_{\{l,n\}}$ (shown in blue in Fig. 4) because there was one path coming out of H_1 and going to m both before and after the modification.

Flow interfaces build on this intuition; the interface $I = (in, out)$ of a flow graph H with domain N is a tuple consisting of the inflow $in\colon N \to M$ (e.g., how many incoming paths each node in H has) and the outflow $out\colon (\mathfrak{N} \setminus N) \to M$ (e.g., how many outgoing paths H has to each external node). Formally, the inflow of $H = (N, e, \mathit{fl})$ is the in that satisfies (FlowEqn) (this is unique, see [41]) and the outflow is defined as $out(n) := \sum_{n' \in N} e(n', n)(\mathit{fl}(n'))$. For example, the flow interface of $\{l\}$ in the left of Fig. 4 is $(\{l \rightarrowtail 1\}, \lambda_0[n \rightarrowtail 1])$ because l has one incoming path from outside $\{l\}$ and the subgraph $\{l\}$ has one outgoing path to n. The interface of $\{l, n\}$ in the left and center of Fig. 4 is $(\{l \rightarrowtail 1, n \rightarrowtail 0\}, \lambda_0[m \rightarrowtail 1])$, which is depicted abstractly on the right. The flow framework tells us that if we have $H = H_1 \odot H_2$ and we modify H_1 to some H_1' with the same interface, then $H' = H_1' \odot H_2$ exists. This means that the flow of all nodes in H_2 is unchanged; thus it suffices to check that H_1' satisfies the flow-based invariant and has the same interface as H_1, which are both local checks.

Interfaces are also a convenient abstraction for expressing specifications. As we have seen, the flow framework requires expressing graph properties as a combination of global constraints (e.g., in path-counting the inflow of the entire graph determines the root node) and node-local constraints (e.g., the path-count of every node is 1). The global constraints can be expressed in terms of the global interface (of the entire graph or data structure), for instance in the list case:

$$\varphi(I) := I.in = (\lambda n.\ (n = r \ ?\ 1 : 0)) \wedge I.out = \lambda_0$$

We use $I.in$ and $I.out$ to denote, respectively, the inflow and outflow of an interface I. Note that, saying the outflow is uniformly zero makes it a closed list (no pointers leave the structure) as opposed to a list segment. The node-local constraints can be expressed on the singleton interfaces of each node; as the inflow of a node that does not have a self edge is equal to its flow, and most constraints on the edges of a node can also be expressed in terms of its outflow. For instance, to encode a list, one can say that each node and its singleton interface satisfy the following predicate: [3]

$$\nu_{\mathrm{ls}}(n, I_n) := I_n.in(n) = 1 \wedge (I_n.out = \lambda_0 \vee I_n.out = \lambda_0[\_ \rightarrowtail 1])$$

By instantiating the flow domain and specifying φ and ν appropriately, one can construct flows and flow interfaces that capture any graph property of interest [41]. Formally, flow interfaces form an algebra with a notion of interface composition $I_1 \oplus I_2$ that can be defined independently of flow graphs. The connection to flow graphs is needed only to interpret the specifications that we write in terms of flow interfaces. We can define an abstraction relation between flow graphs and interfaces and show that interfaces define a congruence relation on flow graphs. Additionally, flow interfaces form a separation algebra [12], which means they can be used in any abstract separation logic (and, as we show in this paper, in Iris).

In our proofs in §4, we specify data structure invariants in terms of constraints on singleton and global interfaces as described above. We then tie the concrete heap representation of each node to its singleton interface and say that the global interface is the composition of all singleton interfaces in the separation logic. The main proof obligation is showing that the program maintains the per-node condition ν in its footprint (i.e. the set of nodes it modifies), and that it preserves the interface of the footprint, which will imply that all other nodes have unchanged flows.

4 Verifying Search Structure Templates

This section shows how to tie together the edgeset framework and flow interfaces in Iris in order to verify template algorithms for concurrent search structures. We use the proof

[2] We assume a garbage-collected setting in this paper.

[3] Krishna et al. [41] use γ for this node-local predicate, but we use ν since the former is used for ghost locations by Iris.

$$\langle C.\ \mathsf{CSS}(r, C) \rangle\ \texttt{cssOp}\ \omega\ k\ \langle \mathsf{res}.\ \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res}) \rangle$$

$$\Psi_\omega(k, C, C', \mathsf{res}) := \begin{cases} C' = C \land (\mathsf{res} \Leftrightarrow k \in C) & \omega = \texttt{search} \\ C' = C \cup \{k\} \land (\mathsf{res} \Leftrightarrow k \notin C) & \omega = \texttt{insert} \\ C' = C \setminus \{k\} \land (\mathsf{res} \Leftrightarrow k \in C) & \omega = \texttt{delete} \end{cases}$$

Figure 5. Abstract specification of `cssOp`.

of the link template from §2 as an example. The other template algorithms we prove, as well as the implementations we consider, are described in the next section. For space reasons, we provide only the intuition for Iris' key logical constructs and reasoning steps as and when they are used; for a more detailed introduction see [35].

We specify the concurrent behavior of search structures using *atomic triples* [16, 36, 37]. A specification $R \twoheadrightarrow \langle P \rangle\ e\ \langle Q \rangle$ consists of a local precondition R, a shared precondition P and a postcondition Q. Such a triple means that the program e, despite executing in potentially many atomic steps, appears to operate atomically on the shared state and transform it from P to Q. Any thread-local resources that e needs are captured in R. Atomic triples are strongly related to the well-known *linearizability* [25] criterion for concurrent algorithms. Intuitively, there is a point in time, known as the *linearization point*, where e updates P to Q.

Our atomic specification of a search structure operation ω (either search, insert, or delete) in Fig. 5 uses an abstract predicate $\mathsf{CSS}(r, C)$ (for concurrent search structure) that represents a search structure with root r containing the set of keys C. The binder on C in the precondition is a special *pseudo-quantifier* that captures the fact that during the execution of ω, the value of C can change (e.g. by concurrent operations) but at the linearization point, ω on operation key k changes $\mathsf{CSS}(r, C)$ to $\mathsf{CSS}(r, C')$ in an atomic step. The new set of keys C', and the eventual return value res, satisfy the predicate $\Psi_\omega(k, C, C', \mathsf{res})$ – here C is bound to the contents *just before* the linearization point. The bottom line is that clients of the search structure can pretend that they are using an atomic implementation with specification Ψ_ω.

4.1 High-level Proof Idea

As our template algorithms are parameterized by concrete data structure implementations, their proofs cannot use any data-structure-specific invariants (such as that the array of keys in a B-tree is sorted). This also means that the specifications for helper functions like `findNext` and `decisiveOp` assumed by the templates must be data-structure-agnostic. Furthermore, if we are able to give local specifications to these helper functions then, since they operate on locked nodes, we will be able to verify their implementations using sequential reasoning. The key challenge is that we need to find a postcondition for `decisiveOp` that speaks only about the node n that it is called on, yet lets us prove that it also updates the global contents C appropriately.

Here is a first attempt at such a specification. Let us for the moment abstract from the data layout of the implementation and reason about mathematical graphs whose nodes are labelled with sets of keys (their contents). For example in the B-link tree in Fig. 2, the contents of y_0 are $\{1, 2\}$, while the contents of internal nodes like n are $\emptyset$. We could say that `decisiveOp` ω n k takes in a node n with contents C_n and returns n with updated contents C'_n such that $\Psi_\omega(k, C_n, C'_n, \mathsf{res})$ holds. The problem is in showing that this lifts to the entire structure, i.e. $\Psi_\omega(k, C, C', \mathsf{res})$. This is hard because the relation between C and C_n is that C is the union of C_n for all nodes n. Similarly, in the B-link tree in Fig. 2, if an operation seeking to delete 3 arrived at node y_0 and returned False because 3 was not present, then the proof must show that 3 is not present anywhere else in the structure.

Intuitively, we know that this is true because the rules defining a B-link tree ensure that y_0 is the only node where 3 can be present. Let the *keyset* of a node n be the set of keys $\mathsf{ks}(n)$ that, if present in the structure, must be in n. For example, the rules of a B-tree dictate that the keyset of node y_0 is $(-\infty, 4)$, and the keyset of y_2 is $[5, 8)$. Notice that every pair of distinct nodes have disjoint keysets; this means given any key k there is exactly one node where k could be present. If we have a data structure where all keysets are disjoint and the contents of each node n are a subset of the keyset of n, then we can show that it is sufficient for `decisiveOp` to ensure that Ψ_ω holds on the node n such that $k \in \mathsf{ks}(n)$. In our example, the delete operation was looking for 3 and called `decisiveOp` on y_0. As $3 \in \mathsf{ks}(y_0)$ and all keysets in the structure are disjoint, we know that if 3 is not in y_0 then 3 cannot be anywhere else in the structure.

To implement this high-level proof idea, we need to answer the following questions: (1) How do we formalize the proof argument in a separation logic? (2) How do we specify and reason locally about keysets (a quantity that depends on the entire graph)? (3) How do we show that the template algorithm finds the node n with k in its keyset? We solve (1) using a novel Iris resource algebra in §4.2 and use flows to encode keysets to solve (2) and (3) in §4.3.

4.2 Ghost State and Disjoint Keysets

Iris models both the knowledge of threads about the shared state (e.g. $k \in \mathsf{ks}(n)$) and protocols for modifying the shared state (e.g. only locked nodes can be modified) using the notion of *ghost state*. Ghost state, also known as logical or auxiliary state, is program state that helps with the proof but has no effect on run-time behavior. A proof about a program using ghost state transfers to a proof of the original program via "erasure" of the ghost state. Ghost state can be allocated by the prover at any time at unused *ghost names*, the analogue of memory addresses for concrete locations, and will contain values drawn from a user-specified resource algebra

(RA). A resource algebra is a generalization of the partial commutative monoid (PCM) algebra commonly used by separation logics. It consists of a set M, a *validity* predicate $\overline{\mathcal{V}}(-)$, a *core* function $|-|$ that maps elements to their core (a generalization of units), and a binary operation $(\cdot)\colon M \times M \to M$ (see [35] for formal definitions). Iris expresses ownership of ghost state by the proposition $\boxed{a}^{\gamma}$ which asserts that ownership of a piece $a \in M$ of the ghost location γ (analogous to the points-to predicate from standard separation logics). Ghost state can be split and combined according to the rules of the underlying RA: $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$. Furthermore, Iris maintains the invariant that the composition of all the pieces of ghost state at a particular location is valid (as given by $\overline{\mathcal{V}}$). To do this, Iris restricts updates to ghost locations to only *frame-preserving updates* $a \rightsquigarrow b$, i.e. those pairs such that b composes with any *frame* (other element) that a could have composed with.

For instance, given an RA M, the authoritative RA $\textsc{Auth}(M)$ (see [35] for the formal definition) can be used to model situations where one thread owns an *authoritative* element $a \in M$ and other parties are allowed to own fragments $b \in M$, with the invariant that all fragments $b \preccurlyeq a$ (shorthand for $\exists c.\ a = b \cdot c$). This can be used to model, for example, a shared heap, where there is a single authoritative heap a and each thread owns a fragment of it. The invariant that all fragments $b \preccurlyeq a$ implies that the fragments owned by all threads are consistent. We write $\bullet\, a$ for ownership of the authoritative element and $\circ b$ for fragmental ownership.

In order to talk about the keysets and contents of nodes, we use an authoritative RA of pairs of sets of keys (X, Y) such that $Y \subseteq X$ (a constraint we can enforce in the validity predicate $\overline{\mathcal{V}}$). We call this the *keyset RA* and define the RA operator to be component-wise disjoint union. We can then denote the abstract state of the search structure by $\boxed{\bullet(\mathsf{KS}, C)}^{\gamma}$ (where KS is the key space, or set of all keys, and C is the global contents), and denote the local abstract state (see §1) of each node by $\boxed{\circ(K_n, C_n)}^{\gamma}$ (where K_n and C_n are the keyset and contents, respectively, of n). By the definition of the authoritative RA, the assertion $\boxed{\bullet(\mathsf{KS}, C)}^{\gamma} * \mathop{\Huge *}_{n \in N} \boxed{\circ(K_n, C_n)}^{\gamma}$ expresses that the sets K_n for each $n \in N$ are disjoint and their union is included in KS . Moreover, $C_n \subseteq K_n$ and similarly the C_n sets are disjoint and are included in C. If we can tie each C_n and K_n to the contents and keyset, respectively, of n, then an assertion like the one above gives us the desired disjoint decomposition of abstract state into local states.

The keyset RA has frame-preserving updates such as:

$$\textsc{ks-del}\quad \frac{\overline{\mathcal{V}}((K, C)) \qquad \overline{\mathcal{V}}((K_n, C_n)) \qquad k \in K_n}{\bullet(K, C), \circ(K_n, C_n) \rightsquigarrow \bullet(K, C \setminus \{k\}), \circ(K_n, C_n \setminus \{k\})}$$

This rule says that if $\boxed{\bullet(K, C)}^{\gamma}$ and $\boxed{\circ(K_n, C_n)}^{\gamma}$ are valid resources such that $k \in K_n$ then we can update the fragment to $(K_n, C_n \setminus \{k\})$ (for instance when we remove k from the contents of a node n) and the authoritative resource to $(K, C \setminus \{k\})$ (meaning k is also removed from the global contents). Combining this with a similar rule for insertions, we get the following lemma:

$$\begin{array}{l}\textsc{ks-upd}\\ \boxed{\bullet(K, C)}^{\gamma} * \boxed{\circ(K_n, C_n)}^{\gamma} * k \in K_n * \Psi_\omega(k, C_n, C'_n, \mathrm{res}) \\ \quad \Rrightarrow \exists C'.\ \boxed{\bullet(K, C')}^{\gamma} * \boxed{\circ(K_n, C'_n)}^{\gamma} * \Psi_\omega(k, C, C', \mathrm{res})\end{array}$$

This lemma is expressed in terms of Iris' basic update modality $\Rrightarrow$. The intuitive meaning of $P \Rrightarrow Q$ is that if we have the resource P then we can do a ghost state update and get Q.

4.3 Encoding Keysets using Flows

We now turn to the question of how to tie the sets used in the keyset RA to the concrete nodes, and reason locally about graph updates and their effects on keysets using flows (§3).

To define keysets using flows, we build on the concept of edgesets. Recall that the edgeset $\mathsf{es}(n, n')$ is the set of keys for which an operation arriving at a node n traverses (n, n'). Let the *inset* of a node n, written $\mathsf{ins}(n)$, be defined by the following fixpoint equation

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

where $\mathit{in}(n) \coloneqq (n = r\ ?\ \mathsf{KS} : \emptyset)$. The inset of a node n is thus KS if n equals the root r, else the set of keys k that are in the inset of a predecessor n' such that $k \in \mathsf{es}(n', n)$. Intuitively, $\mathsf{ins}(n)$ is the set of keys for which operations could potentially arrive at n in a sequential setting. For example, in Fig. 2 insets are shown in the top-left of each node; $\mathsf{ins}(y_2) = [5, 8)$ and $\mathsf{ins}(n') = [5, \infty)$. Let the *outset* of n, $\mathsf{outs}(n)$, be the keys in the union of edgesets of edges leaving n. The keyset can then be defined as $\mathsf{ks}(n) = \mathsf{ins}(n) \setminus \mathsf{outs}(n)$.

If the equation defining the inset looks familiar, the reason is that it is just (FlowEqn) in disguise using sets and set operations, and edge functions that take the intersection with the appropriate edgeset. This means we can define a flow domain where the flow at each node is the inset of that node. This will allow us to talk about the keyset in node-local conditions: in particular, we can now give meaning to the ghost state storing the keysets that were described in §4.2.

Encoding the inset as a flow requires using multisets of keys[4] as the flow domain. We label each edge (n, n') in a graph G by the function $e_{\mathsf{es}(n,n')} \coloneqq (\lambda X.\ \mathsf{es}(n, n') \cap X)$. If the global inflow is $\mathit{in} = (\lambda n.\ (n = r\ ?\ \mathsf{KS} : \emptyset))$, which encodes the fact that operations on all keys k start at the root r, then the flow equation implies that $\mathit{fl}(n)$ is the inset of n.

How does the link template ensure that $k \in \mathsf{ks}(n)$ when `decisiveOp` is called? In the absence of concurrent operations (particularly concurrent split operations), this follows because we start off at the root r, where by definition $k \in \mathsf{ins}(r)$,

[4] We cannot use sets of keys because a flow domain is a cancellative commutative monoid [41], and set union is not cancellative.

and traverse an edge (n, n') only when $k \in \mathsf{es}(n, n')$, maintaining the invariant that $k \in \mathsf{ins}(n)$. When there does not exist an outgoing edge with k in the edgeset, we know by definition that $k \in \mathsf{ks}(n)$.

In the presence of concurrent split operations, the $k \in \mathsf{ins}(n)$ invariant no longer holds because the inset of a node n shrinks after a split. For example, when the split operation shown in Fig. 2 completes and r is linked to n', then the inset of n will reduce from $(-\infty, \infty)$ to $(-\infty, 5)$ as all keys larger than 5 will go from r directly to n'. This means that an operation looking for a key $k > 5$ which was on n before the split will now find itself at a node such that $k \notin \mathsf{ins}(n)$.

Fortunately, the operation is not lost: if it traverses the link edge, it will arrive at a node with k in its inset (namely, n'). This means that if we add k back to the inset of n, then we would not be changing the keyset of any node: k will not be in n's keyset as it is in the edgeset of the link edge, and k is already in the inset of n'. Because this quantity is no longer the inset (as k would not arrive at n in a sequential setting), we call this the *inreach* of n, written $\mathsf{inr}(n)$ (intuitively, this is the set of keys k that can start at n and reach the node containing k in its keyset). Fig. 2 shows the inreach of each node in its top-right corner; the inreach of y_2 is $[5, \infty)$ despite its inset's being only $[5, 8)$ because it can still reach nodes with keys in $[8, \infty)$ in their keyset via link edges.

Formally we define the inreach to be the solution to the following fixpoint equation

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

where in is any inflow such that $\mathit{in}(r) = \mathsf{KS}$. This may look identical to the definition of inset, but there is a subtle, but vital, difference: by not constraining the inflow of non-root nodes, we enable the split operation to add flow to nodes it has split to ensure that their inreach records the fact that they can still reach keys k that were moved to other nodes. For example, in Fig. 2 when the full-split adds the edge (r, n') and re-routes keys in $[5, \infty)$ to take (r, n') instead of (r, n), then n's inset reduces from $(-\infty, \infty)$ to $(-\infty, 5)$. However, the full-split can instead increase $\mathit{in}(n)$ from $\emptyset$ to $[5, \infty)$, thereby preserving its inreach of $(-\infty, \infty)$. As the newly added keys $[5, \infty)$ are propagated via the link edge to a node that has them in its inset (n'), this increase in inflow does not change any keysets.

We have one final issue to solve: as it stands, the full-split does not preserve the interface of $\{r, n, n'\}$ because the outflow to y_2 and y_3 has increased. The reason is that the flow domain is *multisets* of keys, and since we increased $\mathit{in}(n)$ by $[5, \infty)$ there are now two copies of these keys leaving n'. Our solution is to tweak the edge functions to $e_{\mathsf{es}(n,n')} :=$ $(\lambda X.\ \{k \rightarrowtail (k \in \mathsf{es}(n, n') \cap X\ ?\ 1 : 0)\})$, essentially projecting the multiset intersection back to a set, and preventing multiple copies of keys from being propagated.

$$\mathsf{inr}(I_n, n) := I_n\,.\mathit{in}(n) \qquad \mathsf{outs}(I_n) := \bigcup_{n' \notin \mathsf{dom}(I_n)} \mathsf{outs}(I_n, n')$$

$$\mathsf{outs}(I_n, n') := I_n\,.\mathit{out}(n') \qquad \mathsf{ks}(I_n, n) := \mathsf{inr}(I_n, n) \setminus \mathsf{outs}(I_n, n)$$

$$\mathsf{inFP}(n) := \exists N.\ \boxed{\circ N}^{\gamma_f} * n \in N$$

$$\mathsf{inInr}(k, n) := \exists R.\ \boxed{\circ R}^{\gamma_{i(n)}} * k \in R$$

$$\mathsf{N}(n, I_n, C_n) := \mathsf{node}(n, I_n, C_n) * \boxed{\tfrac{1}{2} I_n}^{\gamma_{h(n)}} * \boxed{\circ(\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \mathsf{inFP}(n) * \mathsf{dom}(I_n) = \{n\}$$

$$\varphi(r, I) := \overline{\mathcal{V}}(I) \wedge I.\mathit{in}(r) = \mathsf{KS} \wedge I.\mathit{out} = \lambda_0$$

$$\mathsf{CSS}(r, C) := \exists I.\ \boxed{\bullet I}^{\gamma_I} * \varphi(r, I) * \boxed{\bullet(\mathsf{KS}, C)}^{\gamma_k} * \boxed{\bullet\,\mathsf{dom}(I)}^{\gamma_f}$$
$$* \mathop{\Huge *}_{n \in \mathsf{dom}(I)} \Big(\exists b, I_n.\ \ell(n) \mapsto b * (b\ ?\ \mathsf{True} : \exists C_n.\ \mathsf{N}(n, I_n, C_n)) * \boxed{\circ I_n}^{\gamma_I} * \boxed{\tfrac{1}{2} I_n}^{\gamma_{h(n)}} * \boxed{\bullet\mathsf{inr}(I_n, n)}^{\gamma_{i(n)}} * \mathsf{dom}(I_n) = \{n\}\Big)$$

Figure 6. The invariant for the link template proof.

We now have an invariant for `traverse`: $k \in \mathsf{inr}(n)$. This is true at the root, because $\mathsf{KS} = \mathit{in}(r) \subseteq \mathsf{inr}(r)$, and it is preserved during traversal since `findNext` follows edges with k in the edgeset. We will ensure that no concurrent operations reduce the inreach of any node by adding an appropriate constraint to the search structure predicate CSS in §4.4. The keyset of each node n that is stored in the keyset RA is defined to be $\mathsf{inr}(n) \setminus \mathsf{outs}(n)$. This means that when `findNext` returns `None`, $k \in \mathsf{inr}(n)$ by the traversal invariant and $k \notin \mathsf{outs}(n)$ by the specification of `findNext`. Thus $k \in \mathsf{ks}(n)$, which by §4.2 is sufficient to ensure correctness of the decisive operation.

4.4 An Invariant for the Link Template

Fig. 6 contains our definition of the search structure predicate CSS that captures the link template invariant.[5] CSS is parameterized by a *heap representation* predicate $\mathsf{node}(n, I_n, C_n)$ whose definition is implementation-specific, and provided by the user for implementation proofs (more on this later). Our definition of CSS captures both the invariant maintained by the shared state as well as the protocol threads follow for modifying it:

- We use an authoritative RA of flow interfaces at location γ_I for the flow-based reasoning. Like the keyset RA from §4.2, CSS contains the assertion $\boxed{\bullet I}^{\gamma_I} * \mathop{\ast}_{n \in N} \boxed{\circ I_n}^{\gamma_I}$ which makes I the global interface, i.e. the composition of I_n for all fragments I_n. This allows threads to modify the structure as long as they preserve the flow interface of the modified region (see §3). We require that I satisfies $\varphi(r, I)$ (see Fig. 6), which says that the global interface is valid, the global inflow at the root r is the key space KS (as per the inreach equation from §4.3), and that the search

[5] The top part of Fig. 6 introduces some shorthand notation, which overload some symbols used before because they express the same quantities.

structure is closed. The former is used to prove that the traversal invariant $k \in \mathsf{inr}(n)$ holds initially, when $n = r$, and the latter is used to prove that operations do not leave the structure during traversal.

- We use the keyset RA described in §4.2 at ghost location γ_k. Note that the N predicate ties the fragments to each node's contents and keysets.
- We use an authoritative RA of sets of nodes at location γ_f to encode the footprint, i.e. the domain of the search structure's global interface. CSS owns the authoritative version $\boxed{\bullet\,\mathsf{dom}(I)}^{\gamma_f}$, and the following properties of authoritative sets allow threads to take snapshots of the footprint and assert locally that a given node is in the footprint:

AUTH-SET-UPD
$$\frac{X \subseteq Y}{\bullet X \rightsquigarrow \bullet Y}$$

AUTH-SET-SNAP
$$\bullet X \rightsquigarrow \bullet X \cdot \circ X$$

AUTH-SET-VALID
$$\frac{\mathcal{V}(\bullet X \cdot \circ Y)}{Y \subseteq X}$$

The inFP predicate in Fig. 6 uses this RA to express the fact that we have a pointer to a node in the footprint (e.g., to prove that `lockNode` is called on an allocated node).

- We assume that every node $n \in \mathsf{dom}(I)$ has a lock bit at location $\ell(n)$ that is set to True iff node n is locked. This lock protects the node predicate N, which can be removed from CSS by threads when locking the node (and hence, transfer the node into local state).
- We use fractional RAs at locations $\gamma_{h(n)}$ for each node n to store one half of the node's singleton interface I_n inside and outside N. Since fractional RAs can only be updated when both halves are together, this prohibits other threads from modifying the interface of n when one thread has locked n and removed $\mathsf{N}(n, I_n, C_n)$ from CSS.
- Finally, we use an authoritative RA of sets of keys, at locations $\gamma_{i(n)}$ for each node n, to encode the inreach of each node. This RA has similar rules as the authoritative RA of sets of nodes at location γ_f, hence threads can take snapshots of a node n's inreach and assert that a given key is in it even when they have not locked n (using the inInr predicate from Fig. 6).

4.5 Proof of the Link Template

Before we describe the link template proof, we start by presenting the assumptions it makes about its implementation (summarized in Fig. 7). Recall that we need local specifications for the helper functions `findNext` and `decisiveOp`. Our specifications say that `findNext` is given a node n satisfying $\mathsf{node}(n, I_n, C_n)$ and returns None if k is not in the outset of n else $\mathsf{Some}(n')$ such that k is in the outflow to n' (by our definition of edge functions, this means $k \in \mathsf{es}(n, n')$). Similarly, `decisiveOp` expects a node $\mathsf{node}(n, I_n, C_n)$ such that k is in the keyset of n. If `decisiveOp` returns None then it returns the node unchanged. On the other hand, if it returns $\mathsf{Some}(v')$ then the node is now $\mathsf{node}(n, I_n, C'_n)$, and the return value satisfies the search structure specification with respect to the old and new contents of the node n ($\Psi_\omega(k, C_n, C'_n, v')$).

$$\{\mathsf{node}(n, I_n, C_n) * k \in \mathsf{inr}(I_n, n)\}$$
`findNext n k`
$$\left\{\begin{array}{l} v.\ \mathsf{node}(n, I_n, C_n) * (v = \mathsf{None} * k \notin \mathsf{outs}(I_n) \\ \qquad\qquad \lor\ v = \mathsf{Some}(n') * k \in \mathsf{outs}(I_n, n')) \end{array}\right\}$$

$$\{\mathsf{node}(n, I_n, C_n) * k \in \mathsf{inr}(I_n, n) * k \notin \mathsf{outs}(I_n)\}$$
`decisiveOp ω n k`
$$\left\{\begin{array}{l} v.\ \mathsf{node}(n, I_n, C'_n) * (v = \mathsf{None} * C_n = C'_n \\ \qquad\qquad \lor\ v = \mathsf{Some}(v') * \Psi_\omega(k, C_n, C'_n, v')) \end{array}\right\}$$

$$\mathsf{node}(n, I_n, C_n) * \mathsf{node}(n, I'_n, C'_n) \twoheadrightarrow \mathsf{False}$$

Figure 7. Assumptions the link template proof makes on helper functions and implementation-specific predicates. These are defined and proved by implementations.

Note that these specifications use standard Hoare triples $\{P\}\ e\ \{Q\}$ instead of atomic triples $R \twoheadrightarrow \langle P \rangle\ e\ \langle Q \rangle$. This is because our definition of CSS and the use of node-level locks mean that they operate on *local* state that is not shared. Finally, we assume that the heap representation predicate $\mathsf{node}(n, I_n, C_n)$ implies that we have ownership of the heap location n; in particular, we need the property that it cannot be duplicated, hence owning two copies of it implies False.[6]

We now turn to the template proof: recall that our objective is to prove the atomic triple for `cssOp` from Fig. 5. Unlike standard Hoare triples, when proving a triple $R \twoheadrightarrow \langle P \rangle\ e\ \langle Q \rangle$, we cannot use P throughout the proof of program e. We can "peek" into the precondition P, but only for the duration of an atomic step, and after the step we must either "commit" and establish the postcondition Q (this will be at the linearization point) or "abort" and re-establish P.

Fig. 8 presents a proof outline of the link template algorithm, where the intermediate assertions in braces show the context of the proof (the premises that are currently available). All free variables in the intermediate assertions are implicitly existentially quantified. We use a standard lock module with specs given in lines 1 and 2. For our case studies, we used a simple spin lock and proved this specification, but note that we can swap it out with a more complex lock implementation if necessary. The specification of `traverse` is shown in the lines above and below the procedure. In the precondition, the assertions before the magic wand are resources that one needs in order to call `traverse` and use its atomic specification; these will be available in our proof context when `traverse` is called.

The `cssOp` operation begins with a call to `traverse` on line 20. To satisfy `traverse`'s precondition, we need to peek into CSS and take a snapshot of the global footprint (using AUTH-SET-SNAP and $\varphi(r, I) \Rightarrow r \in \mathsf{dom}(I)$), obtaining $\mathsf{inFP}(r)$. Also, $\varphi(r, I) \Rightarrow k \in \mathsf{inr}(I_r, r)$ so we also take a snapshot of

[6] The magic wand operator $\twoheadrightarrow$ is the SL analogue of implication.

```
1  inFP(n) -∗⟨ C. CSS(r, C) ⟩ lockNode n ⟨ CSS(r, C) ∗ N(n, I_n, C_n) ⟩
2  N(n, I_n, C_n) -∗⟨ C. CSS(r, C) ⟩ unlockNode n ⟨ CSS(r, C) ⟩
3
4  inFP(n) ∗ inInr(k, n) -∗⟨ C. CSS(r, C) ⟩
5  let rec traverse n k =
6    lockNode n;
7    {N(n, I_n, C_n) ∗ k ∈ inr(I_n, n)}
8    match findNext n k with
9    | None -> {N(n, I_n, C_n) ∗ k ∈ inr(I_n, n) ∗ k ∉ outs(I_n)}
10     n
11   | Some n' -> {N(n, I_n, C_n) ∗ k ∈ inr(I_n, n) ∗ k ∈ outs(I_n, n')}
12     {N(n, I_n, C_n) ∗ inFP(n') ∗ inInr(k, n')}
13     unlockNode n; {inFP(n') ∗ inInr(k, n')}
14     traverse n' k
15 ⟨ v. CSS(r, C) ∗ N(v, I_v, C_v) ∗ k ∈ inr(I_v, k) ∗ k ∉ outs(I_v) ⟩
16
17 ⟨ C. CSS(r, C) ⟩
18 let rec cssOp ω r k =
19   {inFP(r) ∗ inInr(k, r)}
20   let n = traverse r k in
21   {N(n, I_n, C_n) ∗ k ∈ inr(I_n, k) ∗ k ∉ outs(I_n)}
22   match decisiveOp ω n k with
23   | None -> {N(n, I_n, C_n)}
24     unlockNode n; {True}
25     cssOp ω r k
26   | Some res ->
27     {N(n, I_n, C'_n) ∗ Ψ_ω(k, C_n, C'_n, res) ∗ k ∈ ks(I_n, n)}
28     (* Linearization point: open CSS(r, C) *)
29     {[●(KS, C)]^{γ_k} ∗ [○(ks(I_n, n), C_n)]^{γ_k} ∗ Ψ_ω(k, C_n, C'_n, res)
        ∗ k ∈ ks(I_n, n) ∗ node(n, I_n, C'_n) ∗ [½I_n]^{γ_h(n)} ∗ ···}
30     {[●(KS, C')]^{γ_k} ∗ [○(ks(I_n, n), C'_n)]^{γ_k} ∗ Ψ_ω(k, C, C', res)
        ∗ node(n, I_n, C'_n) ∗ [½I_n]^{γ_h(n)} ∗ ···}
31     (* Close CSS(r, C'), prove postcondition *)
32     {N(n, I_n, C'_n)}
33     unlockNode n; {True}
34     res
35 ⟨ v. CSS(r, C') ∗ Ψ_ω(k, C, C', v) ⟩
```

Figure 8. The link template algorithm with a proof outline.

r's inreach at ghost location $\gamma_{i(r)}$ to add $\mathsf{inInr}(k, r)$ to our context. The resulting context is depicted in line 19.

To call `traverse` we also need $\mathsf{CSS}(r, C)$, so we need to peek into the precondition again. This is allowed because `traverse` has an atomic triple, it thus behaves atomically and we can peek into atomic preconditions around calls to it. After `traverse` returns, we add its postcondition in line 15 to our context (minus $\mathsf{CSS}(r, C)$, which needs to be given back to re-establish `cssOp`'s precondition since we do not commit here). The next step is the call to `decisiveOp`, for which we already have the precondition in our context.

We then look at the two possible outcomes of `decisiveOp`. In the case where it returns `None`, our context is unchanged, so we execute `unlockNode` using the $\mathsf{N}(n, I_n, C_n)$ in our context. We can use the specification of `cssOp` on the recursive call on line 25 to complete this branch of the proof.

On the other hand, if `decisiveOp` succeeds, we get back a modified node $\mathsf{node}(n, I_n, C'_n)$ with new contents C'_n that satisfies the search structure specification $\Psi_\omega(k, C_n, C'_n, \mathsf{res})$ locally (line 27). We now need to show that this modification results in `cssOp`'s postcondition; this is essentially the *linearization point* of this algorithm.

To do this, we again open the atomic precondition $\mathsf{CSS}(r, C)$. We now have the context in line 29 (we have also expanded $\mathsf{N}(n, I_n, C'_n)$), and now we can apply our ghost update KS-UPD to update the global contents and get the context in line 30. In particular, we have $\Psi_\omega(k, C, C', \mathsf{res})$ and $\mathsf{CSS}(r, C')$, which allows us to "commit" and establish the postcondition. We finally execute the call to `unlockNode` using the remaining $\mathsf{N}(n, I_n, C'_n)$ predicate[7], and complete the proof.

The proof of `traverse` follows a similar line-by-line reasoning using the appropriate specifications of helper functions; the intermediate contexts are shown in Fig. 8.

4.6 Proofs of Template Implementations

To obtain a verified implementation of the link template, one needs to specify the concrete representation of a node by defining the node predicate and provide code for the helper functions that satisfies the specifications in Fig. 7. As mentioned before, these specifications use sequential Hoare triples and have access only to the heap representation of the given node. Thus, if their implementations are sequential code, we can verify them using an off-the-shelf separation logic tool that can verify sequential heap-manipulating code.

5 Proof Mechanization and Automation

In addition to the link template presented in the previous section, we have also verified the give-up and lock-coupling template algorithms from [57], as depicted in Fig. 1. For the link and give-up templates, we have derived and verified implementations based on B-trees and hash tables. For the lock-coupling template we have considered a sorted linked list implementation. The lock-coupling template also captures the synchronization performed by maintenance operations on algorithms such as the split operation on B+ and B-link trees when they traverse the data structure.

The proofs of the template algorithms have been mechanized using the Coq proof assistant, building on the formalization of Iris [37]. These proofs parameterize over the implementation of the helper functions (e.g. `decisiveOp`, `findNext`, etc.) and the heap representation predicate node. The concrete implementations of these helper functions have been verified using the separation logic based deductive program verifier GRASShopper [52]. As the tool uses SMT solvers to

[7] Technically, we perform the linearization at the same time as executing `unlockNode`, but we omit the details here for space reasons.

Table 1. Summary of templates and instantiations verified in Iris/Coq and GRASShopper. For each algorithm or library, we show the number of lines of code, lines of proof annotation (including specification), total number of lines, and the proof-checking / verification time in seconds.

Module	Code	Proof	Total	Time
Templates (Iris/Coq)				
Flow library	0	2803	2803	114
Link template	14	487	501	55
Give-up template	18	390	408	49
Lock-coupling template	26	980	1006	238
Total	**58**	**4660**	**4718**	**456**
Implementations (GRASShopper)				
Flow library	0	721	721	9
Array library	143	320	463	9
B+ tree	63	99	162	21
B-link (core)	85	161	246	36
B-link (half split)	34	192	226	94
B-link (full split)	17	137	154	697
Hash table (link)	54	99	153	10
Hash table (give-up)	60	138	198	13
Lock-coupling list	59	300	359	51
Total	**515**	**2167**	**2682**	**940**

largely automate the verification process, this provided us with a substantial decrease in effort.

Our verification effort includes a mechanization of the meta-theory of flows [41] (i.e. that flow interfaces form an RA). Our formalization is parametric in the flow domain (i.e. the underlying cancellative, commutative monoid). We also provide instantiation of the meta-theory for the specific flow domains used in our proofs (e.g. multisets). We have duplicated this effort in Iris/Coq and GRASShopper in order to make the two parts of our verification self-contained. The formalization is available as two standalone libraries that can be reused for other flow-based proofs in these systems.

In addition to the helper functions of each data structure that are assumed by the templates, we have also verified the split operations for B-link trees. The B-link tree uses a two-part split operation: a half-split that creates a new node, transfers half the contents from a full node to this new node, and adds a link edge; and a full-split that completes the split by linking the original node's parent to the new node. For the split operations, we assume a harness template for a maintenance thread that traverses the data structure graph to identify nodes that are amenable to half splits. While we have not verified this harness, we note that it is a variation of our lock-coupling template where the abstract specification leaves the contents of the data structure unchanged. For the implementations of half and full splits, we verify that the operation preserves the flow interface of the modified region as well as its contents. The full development of our mechanization effort is available online[8].

Table 1 provides a summary of our development. Experiments have been conducted on a laptop with an Intel Core i7-5600U CPU and 16GB RAM. We split the table into one part for the templates (proved in Coq) and one part for the implementations (proved in GRASShopper). We note that for the B-link tree, B+ tree and hash table implementations, most of the work is done by the array library, which is shared between all these data structures. The size of the proof for the lock-coupling list and maintenance operations is relatively large. The reason is that these involve the calculation of a new flow interface for the region obtained after the modification. This requires the expansion of the definitions of functions related to flow interfaces, which are deeply nested quantified formulas. GRASShopper enforces strict rules that limit quantifier instantiation so as to remain within certain decidable logics [4, 51]. Most of the proof in this case involves auxiliary assertions that manually unfold definitions. The size of the proof could be significantly reduced with a few simple tactics for quantifier expansion.

It is difficult to assess the overall time effort spent on verifying the link template algorithm, which was the first algorithm that we considered. The reason is that we designed our verification methodology as we verified the template. However, with all the machinery now in place, our experience is that verifying a new template algorithm is a matter of a few hours of proof effort. In fact, adapting the link template proof to the give-up template was straightforward and required only minor changes. Our experience with adapting implementation proofs is similar.

We believe that our case studies are representative of real-world applications and that our methodology can be widely applied. The template algorithms that we have verified focus on lock-based techniques with fixed linearization points inside a decisive operation. In fact, many real-world applications perform better using lock-based algorithms instead of lock-free algorithms as the latter tend to copy data more[9]. On the other hand, our methodology does not require locking, and can be extended to prove lock-free algorithms such as the Bw-tree [42]. While our methodology can, in theory, be applied to any search structure implementation, there are implementations that use very specific concurrency techniques that cannot be used by other heap representations (e.g. Harris' list [29]). Our technique would give us a "single-use" template in such cases, but this would still structure the proof and make it simpler to construct and verify.

[8] https://github.com/nyu-acsys/template-proofs/tree/pldi_2020

[9] For instance, Apache's CouchDB uses a B+ tree with a global write lock; BerkeleyDB, which has hosted Google's account information, uses a B+ tree with page-level locks in order to trade-off concurrency for better recovery; and java.util.concurrent's hash tables lock the entire list in a bucket during writes, which is more coarse-grained than the one we verify.

6 Related Work

Our work builds on the search structure templates of [57], the Iris separation logic [35], and the flow framework [40, 41]. Our main technical contributions relative to these works are a new proof technique for verifying template algorithms of concurrent search structures that relies on the integration of the flow framework into Iris. The notion of edgesets and keysets are taken from [57] but we show how to reason locally about them using flows. Specifically, we capture the essence of the Keyset Theorem of [57] in terms of an Iris RA, thereby eliminating any dependencies on a specific programming language semantics, and allowing us to easily mechanize the proof in Iris. We also provide a full mechanization of the meta-theory of the flow framework presented in [41] in Coq/Iris and GRASShopper. We note that Krishna et al. [40] use the flow framework to verify a template algorithm based on the give-up technique. However, their proof is only on paper, still depends on a meta-level Keyset Theorem like [57] and uses a bespoke program logic that is difficult to mechanize due to limitations of the original flow framework (cf. [41]).

To our knowledge, we are the first to provide a mechanized proof of a concurrent B-link tree. Unlike the proof of da Rocha Pinto et al. [15], which is not mechanized, our proof does not assume node-level operations to be given as primitives. In particular, we also verify the challenging split operation. The only other comparable proof is that of a B+ tree in [44]. However, this work only considers a sequential B-tree implementation and the proof is considerably more complex than ours (encompassing more than 5000 lines of proof for roughly 500 lines of code). Moreover, much of our proof can be reused to verify other concurrent search structures that rely on linking, such as the concurrent hash table implementation that we consider.

Feldman et al. [23] show how to simplify linearizability proofs of concurrent data structures with unsynchronized searches by reasoning purely sequentially about the traversal performed by the search. Their contribution is orthogonal to ours as they do not aim to parameterize the concurrency proof by the heap representation of the data structure.

Iris does not support reasoning about deallocation. Therefore our proofs assume a garbage collected environment. However, Meyer and Wolff [45] demonstrate a similar proof modularity by decoupling the proof of data structure correctness from that of the underlying memory reclamation algorithm, allowing the correctness proof to be carried out under the assumption of garbage collection. An alternative approach to extending our proofs to deal with memory reclamation is to use Iron [5], a recent extension of Iris that allows proving absence of memory leaks. It is a promising direction of future work to integrate these approaches and our technique in order to obtain verified data structures where the user can mix-and-match the synchronization technique, memory layout, and the memory reclamation algorithm.

There exist many other program logics that help modularize the correctness proofs of concurrent systems [6, 16, 19, 24, 28, 31, 46, 53, 60, 61]. Like Iris, their main focus is on modularizing proofs along the interfaces of components of a system (e.g. between the client and implementation of a data structure) and accounting for differences in the concurrency semantics across different abstraction layers [28]. Instead, we focus on modularizing the proof of a single component (a concurrent search structure) so that the parts of the proof can be reused across many diverse implementations.

As discussed in §5, lock-free implementations of search structures often have non-fixed as well as external linearization points. Much work has been dedicated to addressing this challenge [7, 9, 14, 17, 20, 26, 38, 43, 49, 62]. However, we note that these papers do not aim to separate the proof of thread safety from the proof of structural integrity. In fact, we see our contributions as orthogonal to these works. For example, we can build on the recent work of supporting prophecy variables in Iris [36] to extend our methodology to non-blocking algorithms.

Note, our approach does not critically depend on the use of Iris. For example, our proof methodology can be replicated in other separation logics that support user-defined ghost state, such as FCSL [55], which would also be useful if one wanted to extend this work to non-linearizable data structures [56].

Fully automated proofs of linearizability by static analysis and model checking have been mostly confined to simple list-based data structures [1, 3, 8, 13, 22, 59]. Recent work by Abdulla et al. [2] shows how to automatically verify more complex structures such as concurrent skip lists that combine lists and arrays. However, it is difficult to devise fully automated techniques that work over a broad class of diverse heap representations. In particular, structures like the B-link tree considered here are still beyond the state of the art.

7 Conclusion

We have presented a proof technique for concurrent search structures that separates the reasoning about thread safety from memory safety. We have demonstrated our technique by formalizing and verifying three template algorithms, and showed how to derive verified implementations with significant proof reuse and automation. The result is fully mechanized and partially automated proofs of linearizability and memory safety for concurrent search structures.

Acknowledgments

This work is funded in parts by NYU WIRELESS and by the National Science Foundation under grants 1925605, CCF-1618059, and CCF-1815633. We thank Ketan Kanishka for his help on mechanizing the flow meta-theory in Coq. We also thank the Iris chat room for their patience and support, especially Ralf Jung, Robbert Krebbers, and Dan Frumin.

References

[1] Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. Springer, 324–338. https://doi.org/10.1007/978-3-642-36742-7_23

[2] Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2018. Fragment Abstraction for Concurrent Shape Analysis. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 442–471. https://doi.org/10.1007/978-3-319-89884-1_16

[3] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Werner Damm and Holger Hermanns (Eds.), Vol. 4590. Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49

[4] Kshitij Bansal, Andrew Reynolds, Tim King, Clark W. Barrett, and Thomas Wies. 2015. Deciding Local Theory Extensions via E-matching. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 87–105. https://doi.org/10.1007/978-3-319-21668-3_6

[5] Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: managing obligations in higher-order concurrent separation logic. *PACMPL* 3, POPL (2019), 65:1–65:30. https://doi.org/10.1145/3290378

[6] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. 2005. Variables as Resource in Separation Logic. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May 18-21, 2005 (Electronic Notes in Theoretical Computer Science)*, Martín Hötzel Escardó, Achim Jung, and Michael W. Mislove (Eds.), Vol. 155. Elsevier, 247–276. https://doi.org/10.1016/j.entcs.2005.11.059

[7] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

[8] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. On Reducing Linearizability to State Reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.), Vol. 9135. Springer, 95–107. https://doi.org/10.1007/978-3-662-47666-6_8

[9] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

[10] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

[11] Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65. https://dl.acm.org/citation.cfm?id=2984457

[12] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

[13] Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. 2010. Model Checking of Linearizability of Concurrent List Implementations. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 465–479. https://doi.org/10.1007/978-3-642-14295-6_41

[14] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:20)2015

[15] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark J. Wheelhouse. 2011. A simple abstraction for complex concurrent indexes. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 845–864. https://doi.org/10.1145/2048066.2048131

[16] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science)*, Richard E. Jones (Ed.), Vol. 8586. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[17] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

[18] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 287–300. https://doi.org/10.1145/2429069.2429104

[19] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.), Vol. 6183. Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

[20] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 233–246. https://doi.org/10.1145/2676726.2676963

[21] Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. 2016. Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic. *ACM Trans. Program. Lang. Syst.* 38, 2 (2016), 4:1–4:72. https://doi.org/10.1145/2818638

[22] Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. 2013. Automatic Linearizability Proofs of Concurrent Objects with Cooperating

Updates. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 174–190. https://doi.org/10.1007/978-3-642-39799-8_11

[23] Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018 (LIPIcs)*, Ulrich Schmid and Josef Widder (Eds.), Vol. 121. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:21. https://doi.org/10.4230/LIPIcs.DISC.2018.23

[24] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science)*, Rocco De Nicola (Ed.), Vol. 4421. Springer, 173–188. https://doi.org/10.1007/978-3-540-71316-6_13

[25] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer, 252–266. https://doi.org/10.1007/978-3-642-00590-9_19

[26] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

[27] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science)*, Paul Gastin and François Laroussinie (Eds.), Vol. 6269. Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

[28] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 646–661. https://doi.org/10.1145/3192366.3192381

[29] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings (Lecture Notes in Computer Science)*, Jennifer L. Welch (Ed.), Vol. 2180. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

[30] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[31] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.), Vol. 7737. Springer, 315–334. https://doi.org/10.1007/978-3-642-35873-9_20

[32] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[33] Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 321–332.

[34] 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, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

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

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

[37] 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, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[38] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

[39] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

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

[41] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020. Local Reasoning for Global Graph Properties. *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020* (2020). To appear.

[42] Justin J. Levandoski and Sudipta Sengupta. 2013. The BW-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage. *IEEE Data Eng. Bull.* 36, 2 (2013), 56–62. http://sites.computer.org/debull/A13june/bwtree1.pdf

[43] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. https://doi.org/10.1145/2491956.2462189

[44] J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM,

237–248. https://doi.org/10.1145/1706299.1706329

[45] Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *PACMPL* 3, POPL (2019), 58:1–58:31. https://doi.org/10.1145/3290371

[46] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[47] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

[48] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

[49] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, Andréa W. Richa and Rachid Guerraoui (Eds.). ACM, 85–94. https://doi.org/10.1145/1835698.1835722

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

[51] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

[52] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

[53] Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

[54] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[55] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 77–87. https://doi.org/10.1145/2737924.2737964

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

[57] Dennis E. Shasha and Nathan Goodman. 1988. Concurrent Search Structure Algorithms. *ACM Trans. Database Syst.* 13, 1 (1988), 53–90. https://doi.org/10.1145/42201.42204

[58] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[59] Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Neil D. Jones and Markus Müller-Olm (Eds.), Vol. 5403. Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27

[60] Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.), Vol. 4703. Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

[61] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 964–990. https://doi.org/10.1007/978-3-662-54434-1_36

[62] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1