Verifying Concurrent Search Structure Templates

Siddharth Krishna
New York University
USA
siddharth@cs.nyu.edu

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

Nisarg Patel
New York University
USA
nisarg@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 two concurrent search structure templates, based on link and give-up synchronization, and deriving verified implementations based on B-trees and hash tables. These case studies include algorithms used in real-world file systems and databases, which have so far been beyond the capability of automated or mechanized state-of-the-art verification techniques. In addition, our approach reduces proof complexity and is able to achieve significant proof reuse.

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 [29] enables proofs to be compositional in terms of program structure; (ii) separation logic [47, 54] allows proofs of programs to be local in terms of the state they modify; and (iii) thread modular techniques [27, 30, 49] allow one to reason about each thread in isolation.

Concurrent separation logics [10, 11, 16, 18, 19, 21, 23, 26, 34, 45, 46, 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]. Such proofs, 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 such 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 more reusable proofs and simpler correctness arguments, which in turn aids proof automation.

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 runs into a friend and gets caught up in a conversation. 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, note that 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 the nodes are organized in memory (e.g. whether they form a B-tree or a 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 involved abstractions with the proof technique of reasoning locally about modifications to the heap as in separation logic (SL), which is itself critical for 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.

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 [31, 32, 34, 37]. Specifically, we show how to capture the high-level idea of [57] in terms of Iris’ resource algebras, 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. The decoupling of these two subtasks crucially relies on the recently proposed flow framework [38, 40], 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 demonstrate our methodology by mechanizing the correctness proofs of two template algorithms for concurrent search structures based on the link and the give-up technique of synchronization (Fig. 1). For these, we derive concrete verified implementations based on B-trees and hash tables, resulting in four 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 a concrete implementation is a valid instantiation of a template. We therefore perform only the template proofs in Iris/Coq and verify the implementations using the automated deductive verification tool GRASShopper [50, 51]. The additional automation provided by GRASShopper enables us to bring the proofs of highly complicated implementations such as B-link trees within reach.

The proofs performed in GRASShopper are fully self-contained and include a mechanization of the meta-theory of the flow framework presented in [40]. The template proofs done in Iris assume the meta-theory of the flow framework proved in GRASShopper and are parametric in any possible correct implementation of the node-level operations. The facts proved in GRASShopper match those assumed in Iris. 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).

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 only takes a few hours of proof effort.

To summarize, 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.

• 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 obtained proofs are much simpler and more reusable than prior (pencil-and-paper) proofs of comparable structures.
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’s query 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 \( k_{i-1} < k < k_i \). In the most common implementations of B-trees (called B+ trees), only the keys stored in leaf nodes are considered the contents of a B-tree; 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 query 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 \( n \) gets 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 \( n \)’s parent. 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 \( k > k' \). For efficiency, 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 split of \( y_2 \) moved keys \( \{8, 9\} \) to a new node \( y_3 \), added a link edge, and added a pointer to \( y_1 \), 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 \( i \)th entry points 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 \( es(n, n') \), be the set of query keys for which an operation arriving at a node \( n \) traverses \( (n, n') \). For the B-link tree in Fig. 2, 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 is deliberate. In order to make edgeset a local quantity, we say \( k \in es(n, n') \) 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 \( i \)th 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 \( \text{findNext} \) that finds the next node to visit given a current node \( n \) and a query key \( k \), by looking for an edge \( (n, n') \) with \( k \in es(n, n') \). For the B-link tree, \( \text{findNext} \) does a binary search on the keys 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...
Figure 2. An example B-link tree state in the middle of a split. Node \( n \) was full, and has been half-split and children \( y_2 \) and \( y_5 \) have been transferred to new node \( n' \) (old edges are shown as 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, an array of pointers, and \( l \) (number of keys) in the top right. The first set labelling each edge is its edgeset (§2.2), and the key in the gray box is not considered part of the contents and determines the edgeset of the link edge. All other annotations will be explained in §4.1.

\[
\text{let rec traverse } n \ k =
\begin{align*}
\text{let rec cssOp } \omega \ r \ k = & \\
\text{lockNode } n; & \\
\text{match findNext } n \ k \text{ with } & \\
\text{| None } & \Rightarrow \text{decisiveOp } \omega \ n & \text{k with } \\
\text{| Some } n' & \Rightarrow \text{cssOp } \omega \ r \ k & \\
\text{unlockNode } n; & \\
\text{unlockNode } n; & \\
\text{traverse } n' \ k & \\
\end{align*}
\]

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

operations by implementing the helper function \text{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 \text{traverse} on line 9 to find the node on which it should operate. \text{traverse} is a recursive function that works by following edges whose edgesets contain \( k \) (using the helper function \text{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 \text{findNext}, and holds no locks when moving between nodes. \text{traverse} terminates when \text{findNext} does not find any \( n' \) such that \( k \in es(n, n') \), 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 \text{decisiveOp} returns \text{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 parametrized 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 [39, 40], 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 [39, 40]). We use the fundamental flow framework [40] 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.
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 (\( \lambda_b \) edges omitted).

The interface of the blue region \( \{l, n\} \) is shown on the right, and is preserved by this update.

induced by the entire graph (thereby allowing one to capture global constraints at the node level). When modifying a 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 \( n \). If for every node \( n \), \( pc(n) = 1 \) and \( n \) has at most one outgoing edge (both node-local announcements) 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:

\[
\forall n \in N. \text{flw}(n) = \text{in}(n) + \sum_{n' \in N} e(n', n)(\text{flw}(n')) \quad \text{(FlowEqn)}
\]

This is a fixpoint equation on a function \( \text{flw}: N \rightarrow M \), where \( M \) is a flow domain, \( \text{in} \) 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 neighbors. The flow framework works with directed partial graphs that are augmented with a flow, called flow graphs. A flow graph is a tuple \( H = (N, e, \text{flw}) \) consisting of a finite set of nodes \( N \subseteq \mathcal{R} \) (\( \mathcal{R} \) is potentially infinite), a mapping from pairs of nodes to edge functions \( e: N \times \mathcal{R} \rightarrow E \), and a function \( \text{flw} \) such that (FlowEqn) is satisfied for some inflow \( \text{in} \). Flow graph composition \( H_1 \bullet H_2 \) is a partial operator that is a disjoint union of the nodes, edges, and flow values and is only defined 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 \( \text{in}(n) := (n=r ? 1 : 0) \), and the edge function \( e(n, n') \) is the identity function \( \lambda_0 := (\lambda m. m) \) 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 \( n \), \( pc(n) \), 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 singly 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\(^1\) a given node \( n \) from the list (Fig. 4). The program finds the predecessor \( l \) and successor \( m \) of \( n \) and then swings the pointer from \( l \) to \( m \). 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 \( in: N \rightarrow M \) (e.g., how many incoming paths each node in \( H \) has) and the outflow \( out: \mathcal{R} \setminus N \rightarrow M \) (e.g., how many outgoing paths \( H \) has to each external node). Formally, the inflow of \( H = (N, e, \text{flw}) \) is the in that satisfies (FlowEqn) (this is unique, see [40]) and the outflow is defined as \( \text{out}(n) := \sum_{n' \in N} e(n', n)(\text{flw}(n')) \).

Here, for the flow interface of \( l \) in the left of Fig. 4 is \((\{l \mapsto 1\}, (n \mapsto 1))\) and the interface of \( \{l, n\} \) on both sides is \((\{l \mapsto 1, n \mapsto 0\}, (r \mapsto 1))\). The flow framework tells us that if we have \( H = H_1 \bullet H_2 \) and we modify \( H_1 \) to some \( H'_1 \) with the same interface, then \( H' = H'_1 \bullet H_2 \) exists. This means that the flow of all nodes in \( H_2 \) is unchanged; thus it suffices

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

\(^2\)We assume a garbage-collected setting in this paper.
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:

$$\varphi(I) := I^{in} = (\lambda n. \ (n = r \ ? \ 1 : 0)) \land I^{out} = (\lambda n. \ 0)$$

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 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:

$$\gamma_h(n, I_n) := I^{in}(n) = 1 \land (I^{out}(n) \lor \exists n'. \ I^{out} = \{n' \rightarrow 1\})$$

By instantiating the flow domain and specifying $\varphi$ and $\gamma$ appropriately, one can construct flows and flow interfaces that capture any graph property of interest [40]. Formally, flow interfaces are first-class citizens that come 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 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 $\gamma$ 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 edgest set framework and flow interfaces in Iris in order to verify template algorithms for concurrent search structures. We do this using the proof 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 only provide intuition for Iris’ key logical constructs and reasoning steps as and when they are used; for a more detailed introduction see [32].

We specify the concurrent behavior of search structures using atomic triples [16, 35, 53]. The meaning of a triple $(P, e, Q)$ is that the program $e$, despite executing in potentially many atomic steps, appears to operate atomically on the precondition $P$ and transforms it to $Q$. This is strongly related to the well-known linearizability 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 $\omega$ (either search, insert, or delete) in Fig. 5 uses an abstract predicate 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 $\omega$, the value of $C$ can change (e.g. by concurrent operations) but at the linearization point, $\omega$ on query key $k$ changes CSS$(r, C)$ to CSS$(r, C')$ in an atomic step. The new set of keys $C'$, and the eventual return value $res$, satisfy the predicate $\Psi_{\omega}(k, C, C', res)$ – 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 $\Psi_{\omega}$.

4.1 High-level Proof Idea

As our template algorithms are parametrized 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 these specifications should let us prove that if the call to decisiveOp $\omega \ n \ k$ updated $n$ correctly, then 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 

\[
\text{decisiveOp } \omega \ k \ n \ k \text{ takes in a node } n \text{ with contents } C_n \text{ and returns } n \text{ with updated contents } C'_n \text{ such that } \Psi_n(k, C_n, C'_n, \text{res}) \text{ holds.}
\]

The problem is in showing that this lifts to the entire structure, i.e. \( \Psi_n(k, C, C', \text{res}) \). This is hard because the relation between \( C \) and \( C_n \) is that \( C \) is the union of \( C_n \) for all nodes \( n \). In the B-link tree in Fig. 2, say 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 \( ks(n) \) 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 \( (−∞, 4) \), and the keyset of \( y_2 \) is \( [5, 8) \). Notice that every pair of distinct nodes have disjoint keysets; another way of saying this is given any key \( k \) there is exactly one node in a data structure 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 \( \Psi_n \) holds on the node \( n \) such that \( k \in ks(n) \). In our example, the delete operation was looking for 3 and called decisiveOp on \( y_0 \). As \( 3 \in ks(y_0) \) 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.

We now have the following challenges in implementing this high-level proof idea in a concurrent separation logic:

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 show how to solve (1) using Iris’ flexible notion of ghost state in §4.2 and show how using flows to capture keysets solves (2) and (3) simultaneously in §4.3.

### 4.2 Ghost State and Disjoint Keysets

Iris models both the knowledge of threads about the shared state (e.g. \( k \in ks(n) \)) 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 a type of primitive resource (analogous to the points-to predicate from standard separation logics) that helps with the proof but has no effect on run-time behavior. 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 \( \overline{V}(-) \), a core function \( |·| \) that maps elements to their core (a generalization of units), and a binary operation \( (·) : M \times M \rightarrow M \) (see [32] for formal definitions). Iris expresses ownership of ghost state by the proposition \( \overline{a}^{γ} \) which asserts that ownership of a piece \( a \in M \) of the ghost location \( γ \). Ghost state can be split and combined according to the rules of the underlying RA:

\[
\overline{\alpha a}^{γ} + \overline{\beta b}^{γ} \rightarrow \overline{αa + βb}^{γ}. \quad \overline{α a}^{γ} - \overline{β b}^{γ} \rightarrow \overline{α a - β b}^{γ}.
\]

Furthermore, Iris maintains the invariant that the composition of all the pieces of ghost state at a particular location is valid (as given by a validity predicate \( \overline{V} \)). To do this, Iris restricts updates to ghost locations to only frame-preserving updates \( a \triangleright b \), 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 \( \text{Auth}(M) \) (see [33] for the formal definition) can be used to model situations where one party owns the authoritative element \( a \in M \) and other parties are allowed to own fragments \( b \in M \), with the invariant that all fragments \( b \lessdot a \) (shorthand for \( ∃c. \ a = b \cdot c \)). 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 \( b \lessdot a \) implies that the fragments owned by all threads are consistent. We write \( a \triangleright b \) for ownership of the authoritative element and \( b \triangleright a \) 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 \( Y \subseteq X \) (captured by the validity predicate \( \overline{V} \)). We call this the keyset RA and define the RA operator to be component-wise disjoint union. By the definition of the authoritative RA, the assertion

\[
\bullet (K_n, C_n)^{γ} \ast \bigotimes_{n \in N} \circ (K_n, C_n)^{γ}
\]

expresses that the sets \( K_n \) for each \( n \in N \) are disjoint and their union is \( K \) (the key space, or set of all keys). Moreover, \( C_n \subseteq K_n \) and similarly the \( C_n \) sets are disjoint and compose to \( C \). If we can tie each \( C_n \) to \( K_n \) to the contents and keyset of the node \( n \) respectively, then an assertion like the one above will give us the properties we want of contents and keysets. We can also prove that this RA has the following frame-preserving updates:

**KS-INS**

\[
\overline{\Psi(V((K, C)))} \rightarrow \overline{\Psi((K_n, C_n))} \quad k \in K_n
\]

\[
\bullet (K, C), \circ (K_n, C_n) \triangleright \bullet (K, C \cup \{k\}), \circ (K_n, C_n \cup \{k\})
\]

**KS-DEL**

\[
\overline{\Psi(V((K, C)))} \rightarrow \overline{\Psi((K_n, C_n))} \quad k \in K_n
\]

\[
\bullet (K, C), \circ (K_n, C_n) \triangleright \bullet (K, C \setminus \{k\}), \circ (K_n, C_n \setminus \{k\})
\]
For instance, \( \text{ks-del} \) says that if we have valid resources 
\( (K(C_1) \setminus k) \) and \( \text{ins}(n, n') \) such that \( k \in K_1 \) then we can update the fragment to \( (K(C_1 \setminus \{k\}) \) (for instance when we remove \( k \) from the contents of a node \( n \)) and the authoritative resource to \( (K(C_1 \setminus \{k\}) \) (meaning \( k \) is also removed from the global contents). This is captured by the following lemma that follows from the two rules above:

\[
\text{ks-upd} = \{ (K(C_1) \setminus k) \text{ins}(n, n') \} \quad \text{such that } k \in K_1 \text{ then we can update the fragment to } (K(C_1 \setminus \{k\}) \}
\]

This lemma is expressed in terms of Iris’ basic update modality \( \triangleright= \). The intuitive meaning of \( P \triangleright= Q \) 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 we can express and reason about keysets locally using flows (§3).

To define keysets using flows, we build on the concept of edgesets. Recall that the edgeset \( \text{es}(n, n') \) 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 \( \text{ins}(n) \), be defined by the following fixpoint equation:

\[
\forall n \in N. \text{ins}(n) = \text{in}(n) \cup \bigcup_{n' \in N} \text{es}(n', n) \cap \text{ins}(n')
\]

Here, \( \text{in}(n) := (n = r \land \text{ks} : \emptyset) \), so the inset of a node \( n \) is KS if \( n = r \), or else the set of keys in the inset of a predecessor \( n' \) such that \( k \in \text{es}(n', n) \). Intuitively, \( \text{ins}(n) \) is the set keys for which operations could potentially arrive at \( n \) in a sequential setting. For example, the inset of \( y_1 \) in Fig. 2 is \( \{4, 5\} \) and \( \text{ins}(n') = [5, \infty) \). Let the **outset of** \( n \), \( \text{outs}(n) \), be the keys in the union of edgesets of edges leaving \( n \). The keyset can then be defined as \( \text{ks}(n) = \text{ins}(n) \setminus \text{outs}(n) \).

If the equation defining the inset looks familiar, it is because 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 keysets 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\(^3\) as the flow domain. We label each edge \( (n, n') \) in graph \( G \) by the function \( \text{es}(n, n') \) with the following format: \( (AX \cdot X \cap \text{es}(n, n')) \). If the global inflow is \( \text{in} = (\lambda n. (n = r \land \text{ks} : \emptyset)) \), which encodes the fact that operations on all keys \( k \) start at the root \( r \), then the flow equation implies that \( \text{flow}(n) \) is the inset of \( n \).

How does the link template ensure that \( k \in \text{ks}(n) \) when \( \text{decisiveOp} \) is called? In the absence of split operations and link edges, this follows because we start off at the root \( r \), where by definition \( k \in \text{ins}(r) \), and traverse an edge \( (n, n') \) only when \( k \in \text{es}(n, n') \), maintaining the invariant that \( k \in \text{ins}(n) \). When there does not exist an outgoing edge with \( k \) in the edgetset, we know by definition that \( k \notin \text{ks}(n) \).

In the presence of split operations, this invariant breaks down 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 \( k \notin \text{ins}(n) \). Note, however, that if it traverses the link edge, it can get back to a node with \( k \) in its inset (namely, \( n' \)).

Our key idea is to view the graph as an overlay of two structures: a standard structure where the flow computes the inset, and a link structure consisting only of the link edges. For the B-link tree, the main structure consists of the tree edges from nodes to their children, while the link structure is composed of one list per level. This is modeled in the flow framework by using the product of two multiset domains as the flow domain, where the first component calculates the inset as described above. Fig. 2 shows this by labelling the inflows and edgesets with tuples. The roots of the second component are the first (leftmost) nodes on each level (as shown in Fig. 2), and the resulting flow at each node \( n \) is called the **linkset of** \( n \), denoted \( \text{lnks}(n) \). The linkset of \( y_0 \) is \((\infty, \infty)\) as it is the first leaf, and the linkset of \( y_2 \) is \([5, \infty)\). One can think of the linkset component as describing how keys are routed when they traverse link edges. Note that when a node is split, although its inset may reduce, its linkset never decreases.

With these notions, we can define the invariant of Traverse to be that \( k \in \text{ins}(n) \cup \text{lnks}(n) \). This is true at the root, because \( \text{lnks}(r) = \text{ins}(r) = \text{KS} \), and it is preserved during the traversal even with concurrent splits. Another important property of linksets is that \( \text{lnks}(n) \setminus \text{outs}(n) \subseteq \text{ks}(n) \) (this is something we will enforce at the node level). This means that when findNext returns None, \( k \in \text{lnks}(n) \setminus \text{outs}(n) \subseteq \text{ks}(n) \), which by §4.2 is sufficient to ensure correctness of the decisive operation.

Before describing our specifications in terms of a global constraint \( \varphi \) and a node local constraint \( \gamma \), we introduce some shorthand notation for clarity (these overload some symbols used before because they express the same quantities):

\[
\text{ins}(I_n, n) := I_n^{\text{ins}}(n)_{ls} \quad \text{lnks}(I_n, n) := I_n^{\text{lnk}}(n)_{ls}
\]

\[
\text{out}(I_n, n') := I_n^{\text{out}}(n')_{ls} \cup I_n^{\text{out}}(n')_{ls}
\]

\[
\text{outs}(I_n) := \bigcup_{n' \notin I_n} \text{out}(I_n, n')
\]

\[
\text{ks}(I_n, n) := \text{ins}(I_n, n) \setminus \text{outs}(I_n, n)
\]
Verifying Concurrent Search Structure Templates

where \(m_{is}\) and \(m_{ls}\) denote the inset and linkset component of the flow value \(m\). The property we want from linksets is enforced by this node-local constraint \(\gamma\):

\[
y(n, I_n, C_n) := \text{inks}(I_n, n) \setminus \text{outs}(I_n, n) \subseteq \text{ins}(I_n, n)
\]

\[
\wedge \text{dom}(I_n) = \{n\} \wedge C_n \subseteq \text{ks}(I_n, n)
\]

We also require the following constraints on the global interface:

\[
\varphi(r, I) := (\forall n. \text{dom}(n) = (n = r ? \text{KS} : \emptyset)) \wedge \text{ins}(n, I) = \lambda_0
\]

This says that in the inset flow domain component, the global inflow assigns the entire key space \(\text{KS}\) to the root \(r\), and \(\emptyset\) for every other node (i.e. all searches start at the root). It does not restrict the global inflow in the linkset component. We also require that the search structure is closed, encoded by saying the global interface has no outgoing flow.

### 4.4 Proof of the Link Template

We define the search structure predicate as follows:

\[
\text{CSS}(r, C) := \exists I. \left( \sum_{n \in I} \gamma(n) \ast \text{dom}(I) \wedge \text{KS}(C) \ast \varphi(r, I) \right)
\]

\[
\ast \sum_{b \in I, \ell(n) \mapsto b} \ast (b \ast \text{true} : \exists C_n. \text{N}(n, I, C_n))
\]

\[
\ast \sum_{n \in I} \gamma(n) \ast \text{ins}(n, I) \ast \text{dom}(n) \ast \text{ks}(n, C)
\]

\[
\text{N}(n, I, C_n) := \text{node}(n, I, C_n) \ast \sum_{n \in I} \gamma(n) \ast \text{ins}(n, I) \ast \text{dom}(n) \ast \text{ks}(n, C)
\]

\[
\text{CSS} \text{ is parameterized by a heap representation predicate node}(n, I, C_n) \text{ whose definition is implementation-specific, and provided by the user for implementation proofs (more on this later). We assume that every node } n \in I \text{ has a lock bit at location } \ell(n) \text{ that is set to True iff node } n \text{ is locked. This lock protects the node predicate } \text{N}, \text{ which can be removed from CSS by threads when locking the node (and hence, transfer the node into local state). CSS uses a few different types of ghost states to capture the shared state and the protocol for modifying it:}
\]

- **We use an authoritative RA of flow interfaces at location \(\gamma_f\) to implement the flow-based reasoning. Like with the keyset RA from §4.2, CSS contains the assertion \(\sum_{n \in I} \gamma_f \ast \ast \), which makes \(I\) the global interface, i.e. the composition of \(I_n\) for all \(n\). We require that \(I\) satisfies \(\varphi(r, I)\) from §4.3.
- **We use an authoritative RA of sets of nodes at location \(\gamma_f\) to encode the footprint, i.e. the domain of the search structure. CSS owns the authoritative version \(\sum_{n \in I} \gamma_f \ast \text{dom}(n) \ast \gamma_f \), 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:

\[
\begin{align*}
\text{AUTH-SET-UPD} & : X \subseteq Y \quad \text{AUTH-SET-SNAP} & : X \leftrightarrow \bullet Y \\
\bullet X & \leftrightarrow \bullet Y \quad \text{AUTH-SET-VALID} & : \overline{\text{P}(\text{x} \cdot Y)} \\
\text{Y} & \subseteq X
\end{align*}
\]

- **We use a non-deterministic RA \(\gamma_f\) to implement the flow-based reasoning. Like with the keyset RA from §4.2, CSS contains the assertion \(\sum_{n \in I} \gamma_f \ast \ast \), which makes \(I\) the global interface, i.e. the composition of \(I_n\) for all \(n\). We require that \(I\) satisfies \(\varphi(r, I)\) from §4.3.

\[
\begin{align*}
\text{AUTH-SET-UPD} & : X \subseteq Y \quad \text{AUTH-SET-SNAP} & : X \leftrightarrow \bullet Y \\
\bullet X & \leftrightarrow \bullet X \circ \text{Y} \quad \text{AUTH-SET-VALID} & : \overline{\text{P}(\text{x} \cdot Y)} \\
\text{Y} & \subseteq X
\end{align*}
\]

This is used, for example, to express the fact that lockNode is called on a node in the search structure.

- **We use the keyset RA described in §4.2 at ghost location \(\gamma_k\). Note that the N predicate ties the fragments to each node’s contents and keysets.
- **We use fractional RAs at locations \(\gamma_l(n)\) for each node \(n\) to store the node’s singleton interface \(I_n\). This is so that when a node is locked and a thread removes \(N(n, I_n, C_n)\) from CSS, other threads are prohibited from modifying the interface of \(n\), but can still read and reason about it.
- **Finally, we use an authoritative RA of sets of keys, at locations \(\gamma_l(n)\) for each node \(n\), to encode the linkset of each node. This allows threads to assert that a key is in the linkset of a given node even when it is unlocked, as described in §4.3.

The link template proof makes certain assumptions on an implementation that are summarized in Fig. 6. For instance, it assumes that the heap representation predicate \(\text{node}(n, I_n, C_n)\) implies that we have ownership of the heap location \(n\) and that the node-level invariant \(\gamma\) from §4.3 are satisfied. We are also able to give local specifications for the helper functions. For instance, \(\text{findNext}\) is given a node \(n\) satisfying \(\text{node}(n, I_n, C_n)\) and returns None if \(k\) is not in the set of \(n\) else some \(n’\) such that \(k\) is in the edge of \(n, n’\) (captured by the \text{out} predicate). Similarly, the specification of \text{decisiveOp} expects a node \text{node}(n, I_n, C_n)\) such that \(k\) is in the keyset of \(n\). If \text{decisiveOp} returns None then it returns the node unchanged. On the other hand, if it returns some \((u’)\) then the node is now \text{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 \(\Psi(u)(k, C_n, C_n’, u’)\).

Note that these specifications use standard Hoare triples \(\{ P \} e \{ Q \}\) instead of atomic triples \(\{ P \} e \{ Q \}\). 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.

We now turn to the template proof: recall that our objective is to prove the atomic triple for \text{cssOp} in Fig. 5. Unlike

\[
\begin{align*}
\text{node}(n, I_n, C_n) \ast (k \in \text{ins}(I_n, n) \vee k \in \text{inks}(I_n, n))
\end{align*}
\]

\[
\begin{align*}
\text{findNext} n k
\end{align*}
\]

\[
\begin{align*}
\{ u. \text{node}(n, I_n, C_n) \ast (u = \text{None} \wedge k \not\in \text{outs}(I_n))
\end{align*}
\]

\[
\begin{align*}
\{ \wedge u = \text{Some}(n’) \wedge k \in \text{outs}(I_n) \}
\end{align*}
\]

\[
\begin{align*}
\{ \text{decisiveOp} \wedge n k
\end{align*}
\]

\[
\begin{align*}
\{ u. u = \text{None} \wedge \text{node}(n, I_n, C_n)
\end{align*}
\]

\[
\begin{align*}
\{ \wedge u = \text{Some}(u’) \wedge \text{node}(n, I_n, C_n’ \ast \Psi(u)(k, C_n, C_n’, u’))
\end{align*}
\]
Let us now step through the proof of \texttt{cssOp}. The code begins with a call to \texttt{traverse} on line 22. To satisfy \texttt{traverse}'s precondition, we need to peek into \texttt{CSS} and take a snapshot of the global footprint (using \texttt{AUTH-SET-\texttt{SNAP}} and \( q(r, l) \Rightarrow r \in \text{dom}(I) \) obtaining \texttt{inFP}(r)). Also, \( q(r, l) \Rightarrow k \in \text{ins}(I, r) \) so we also take a snapshot of \( r \)'s linkset at ghost location \( \gamma(r) \) to add \texttt{inln}(n, k) to our context. The resulting context is depicted in line 21.

To call \texttt{traverse} we also need \texttt{CSS}(r, C), so we need to peek into the precondition again. This is allowed because \texttt{traverse} has an atomic triple, it thus behaves atomically and we can peek into atomic preconditions around calls to it. After \texttt{traverse} returns, we add its postcondition in line 17 to our context (minus \texttt{CSS}(r, C), which needs to be given back since we don't commit here). The next step is the call to \texttt{decisiveOp}, for which we need to show that \( k \in \text{ins}(I, n) \). This follows from \texttt{inln}(n, k) and \( k \notin \text{outs}(I_b) \) by \( \gamma \).

We then look at the two possible outcomes of \texttt{decisiveOp}. In the case where it returns \texttt{None}, our context is unchanged, so we execute \texttt{unlockNode} using the \texttt{N}(n, I_b, C_n) in our context. We can use the specification of \texttt{cssOp} on the recursive call on line 27 to complete this branch of the proof.

On the other hand, if \texttt{decisiveOp} succeeds, we get back a modified node \texttt{node}(n, I_b, C_n) with new contents \( C' \) that satisfies the search structure specification \( \Psi_{\text{\texttt{d}}}(k, C_n, C'_n, \text{res}) \) locally (line 29). We now need to show that this modification results in the appropriate postcondition; this is essentially the \textit{linearization point} of this algorithm.

To do this, we again open the atomic precondition \texttt{CSS}(r, C). We now have the context in line 31, and now we can apply our ghost update \texttt{ks-upd} to update the global contents and get the context in line 32. In particular, we have \( \Psi_{\text{\texttt{d}}}(k, C, C', \text{res}) \) and \texttt{CSS}(r, C'), which allows us to "commit" and establish the postcondition.

We finally execute the call to \texttt{unlockNode} using the remaining \( \texttt{N}(n, I_b, C_n) \) predicate, and complete the proof. The proof of \texttt{traverse} follows a similar line-by-line reasoning using the appropriate specifications of helper functions; the intermediate contexts are shown in Fig. 7.
helper functions that satisfies the specifications in Fig. 6. As mentioned before, these specifications use sequential Hoare triples and only have access 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 template algorithm from [57], as depicted in Fig. 1. For these templates, we have derived and verified implementations based on B trees and hash tables.

The proofs of the template algorithms have been mechanized using the Coq proof assistant, building on the formalization of Iris [34]. The implementations of the helper functions for the concrete implementations that are assumed in the template algorithms (e.g. \texttt{decisiveOp}, \texttt{findNext}, etc.) have been verified using the separation logic based deductive program verifier GRASShopper [51]. This provided us with a substantial decrease in verification effort, as the tool infers intermediate assertions whose validity is proved automatically using SMT solvers. While we do not have, as of now, a formal proof for the transfer of proofs between Iris and GRASShopper, note that Iris is expressive enough to support all the reasoning that we do in GRASShopper, but comes with significant additional manual effort.

The Coq formalization assumes the meta-theory of flow interfaces (i.e. that flow interfaces form an RA) and that they enable frame preserving updates as well as some basic general lemmas about flow interfaces. All these facts are proved in our GRASShopper formalization of flow interfaces. The template proofs parameterize over the implementation of the helper functions, the heap representation predicate node as well as the actual flow domain.

All properties involving the specific flow domain elements and $y$ needed in the template proofs are factored out into a few lemmas. These are assumed in Coq and proved in GRASShopper as they can be easily discharged using an SMT solver.

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 parent node of the original node to the newly created 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 simple variation of our existing templates 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 in the supplementary materials.

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 proofs for the maintenance operations is relatively large. The reason is that these require 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, 50]. Most of the proof in this case involves auxiliary assertions that manually unfold definitions. The actual calculation of the interface is performed by the SMT solver. 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. This is because 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-plate proof to the give-up template was straightforward and only required minor changes. Our experience with adapting implementation proofs is similar.

6 Related Work

Our work builds on the search structure templates of [57], the Iris separation logic [32], and the flow framework [39, 40]. 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 key-sets 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 [40] in GRASShopper. We note that Krishna et al. [39] 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. [40]).

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 [43]. 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.

Iris does not support reasoning about deallocation. Therefore our proofs assume a garbage collected environment. However, Meyer and Wolff [44] 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 the 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 concurrent separation logics that help modularize the correctness proofs of concurrent systems [6, 16, 19, 23, 28, 45, 52, 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). 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.

The template algorithms that we have verified focus on lock-based techniques with fixed linearization points inside a decisive operation. This is representative of real-world applications, many of which prefer lock-based over lock-free algorithms as the latter tend to copy data more. On the other hand, our methodology does not require locking, and can be extended to prove lock-free algorithms such as the Bw-tree [41]. Such algorithms have often have non-fixed as well as external linearization points. Much work has been dedicated to addressing this challenge [7, 9, 14, 17, 20, 25, 36, 42, 48, 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 [53] to extend our methodology to non-blocking algorithms. Also, 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 scope of 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 two template algorithms, and

---

[4]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.
show 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 a large class of concurrent search structures.

References


Verifying Concurrent Search Structure Templates

PL’18, January 01–03, 2018, New York, NY, USA