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

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 ReLoC/Iris, we show how to prove contextual refinement for template algorithms, and how to instantiate them to obtain multiple verified implementations. We demonstrate our approach by formalizing three concurrent search structure templates, based on link, give-up, and lock-coupling synchronization, and deriving implementations based on B-trees, hash tables, and linked lists. Our proofs are mechanized and partially automated using the Coq proof assistant and the deductive verification tool GRASShopper. Not only does our approach reduce the proof complexity, we are also 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 [40, 46] allows proofs of programs to be local in terms of the state they modify; and (iii) thread modular techniques [27, 30, 42] allow one to reason about each thread in isolation.

Concurrent separation logics [9, 10, 15, 17, 18, 20, 22, 25, 33, 38, 39, 50, 52] 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 [14]. Such proofs remain large and complex, however, and are not yet mechanized.

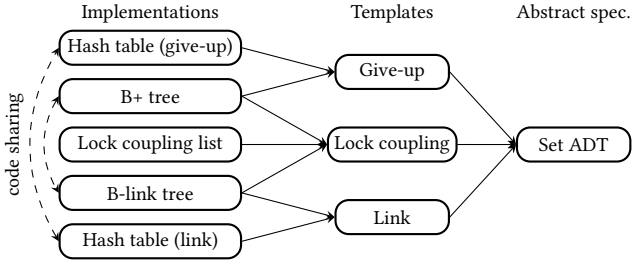

Figure 1. The structure of our proofs.

A *search structure* is a data structure that supports fast search, insert, and delete operations on a set of key-value pairs. Examples of search structures include sorted lists, binary search trees, hash tables, B-trees, and B-link trees. Existing proofs of concurrent search structures such as [14] intertwine the synchronization invariants and the memory invariants, consequently fusing together the reasoning about threads and memory. Not only does this make the proofs harder to understand, it also limits proof reuse. We propose that proofs of concurrent search structures be decomposed into two parts: (i) the reasoning about interference and (ii) the verification of the heap.

This paper shows how to adapt and combine recent advances in compositional abstractions, separation logic, and refinement proofs in order to verify *template algorithms* for concurrent search structures [49]. These template algorithms abstract from the concrete layout of nodes in the data structure (e.g. a tree or a hash table) and the organization of data within a node. We prove that these template algorithms refine the abstract specification of search structures, and show how they can in turn be refined to individual data structure implementations with significantly reduced proof effort. In particular, we are the first to obtain a mechanized proof of concurrent B-link trees. Moreover, unlike [14], the proof does not assume node-level operations to be given as primitives.

Our method builds on the abstraction of *flow interfaces* [36], an abstraction mechanism for unbounded heap regions in separation logic. Flow interfaces can express constraints on the contents of a data structure independently of its shape. This allows us to disentangle the invariants necessary to reason about thread synchronization from the structural invariants needed for memory safety. While the paper [36] presented a template algorithm based on the so-called give-up technique, the accompanying formal proof was only of

NYU TR2019-992, February, 2019
2019.

memory safety. Moreover, the formal proofs were not mechanized, because the original flow framework cannot be used with existing separation logic verification tools and proof assistants. By contrast, this paper shows how to adapt the flow framework to enable proof mechanization.

Specifically, we show in §4 that flow interfaces form a resource algebra, which allows us to use them as ghost state in the Iris higher-order separation logic framework [31–33, 35]. Iris is formalized in Coq [13], yielding mechanically checked proofs. We then show in §5 how to use the ReLoC relational logic [24] that builds on Iris to prove that our template algorithms contextually refine their sequential specifications while abstracting from the implementation.

We demonstrate our method by formalizing three template algorithms based on the link, the give-up, and the lock-coupling technique of synchronization (Fig. 1). For these, we derive concrete implementations based on B-trees, hash tables, and sorted linked lists, leading to five different data structures whose implementations we verify using the automated separation logic tool GRASShopper [43, 44]. We thus obtain fully mechanized and partially automated proofs of both linearizability and memory safety for a large family of concurrent search structures.

Our approach enjoys some advantages over the state-of-the-art. Encoding our flow-interface-based invariants in Iris allows us to perform *sequential* reasoning when we verify that a concrete implementation is a valid instantiation of a template. This means that we can use off-the-shelf automated separation logic tools for sequential programs, which decreases the manual proof effort. Further, the template-based modularity in our proofs allow us to mix and match synchronization protocols and heap representations with negligible additional proof effort. For example, most of the core operations on B-trees such as insertion and deletion of a key into a node are shared between the B-link (which uses the link technique) and the B+ tree (which uses the give-up technique). Hence, they need to be verified only once.

2 Overview

This section motivates and demonstrates our approach using the B-link tree implementation of a search structure, and the link template algorithm that generalizes it. 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 $l+1$ is between B and $2B$ for some constant B. At internal nodes, an operation on k takes the branch y_i if $k_{i-1} \leq k < k_i$. 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 (or pointers, if it is an internal node) into a new node n', and adding a link to n' from n's parent. In the concurrent setting, one 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 store a key k' (the key in the gray box in the figure) that indicates to concurrent operations that all keys $k > k'$ can be reached by following the link edge. 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 implemented as a separate thread that traverses the structure looking for half-split nodes n, locks the parent of n, and adds the 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_3 to 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' was added to r.

2.2 The Link Template Algorithm

The link technique is not restricted to B-trees: consider a hash table implemented as an array of pointers, where the ith 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'.

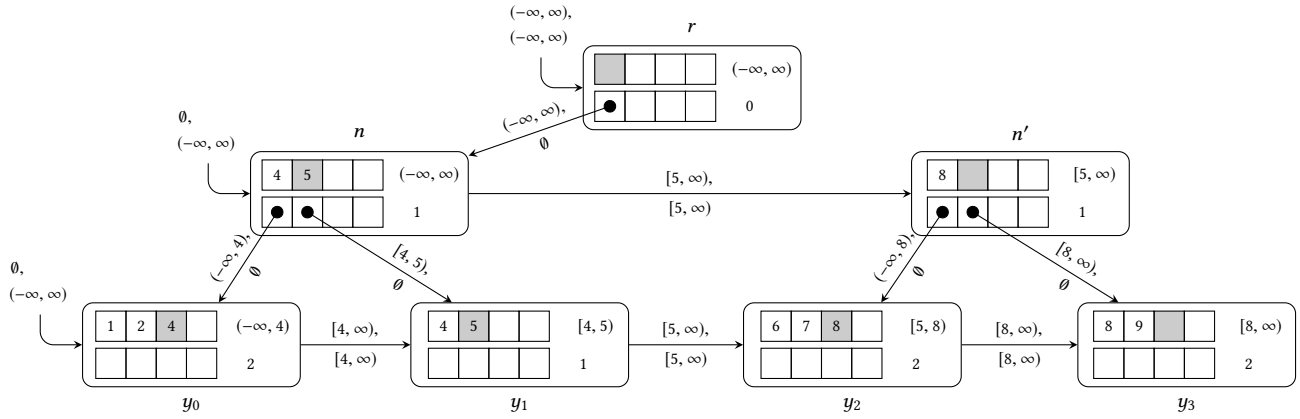

Figure 2. An example B-link tree. Each node shows the array of keys in the top left, the array of pointers in the bottom left, l (number of keys) in the bottom right, and its inflow in the top right. The key in the gray box is not considered part of the contents and determines the edgeset of the link edge. The edges are labelled with edgesets and linksets (see §5.1). The global inflow is shown as curved arrows on the top left of nodes, and is omitted when zero.

```
1 let rec traverse n k =
2   lockNode n;
3   match findNext n k with
4   | None -> n
5   | Some n' -> unlockNode n; traverse n' k
6
7 let rec searchStrOp dOp r k =
8   let n = traverse r k in
9   match decisiveOp dOp n k with
10  | None -> unlockNode n; searchStrOp dOp r k
11  | 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 like `findNext` and `decisiveOp`.

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 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 $\text{es}(n, n')$, 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 $[5, \infty)$. Note that 4 is in the edgeset of (y_0, y_1) even though an operation on 4 would not normally reach y_0; in general, $k \in \text{es}(n, n')$ 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.

Fig. 3 lists the link template algorithm [49], 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 a query key k, by looking for an edge (n, n') with $k \in \text{es}(n, n')$. For the B-link tree, `findNext` does a binary search on the keys to find the appropriate pointer to follow, while for the hash table, when at the root it 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 `searchStrOp` 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 helper function `traverse` on line 8 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 $k \in \text{es}(n, n')$, 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 9). 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 parametrized by the helper functions, then we can reuse the proof across diverse implementations.

2.3 The Edgeset Framework

As the link template algorithm is parametrized by the concrete data structure, its proof cannot use any data-structure-specific invariants (such as that the array of keys in a B-tree is sorted). The edgeset framework [49] provides a correctness condition for search structure algorithms in terms of reachability properties of sets of keys on a mathematical graph, abstracting from the data layout of the implementation.

Let the contents of a node be the set of keys that are stored at that node (for 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 let the state of a data structure be the graph whose edges are labelled with edgesets and nodes with their contents. The abstract state of a graph is then the union of the contents of all its nodes. Proving that the link template refines its abstract specification requires us to prove that the decisive operation updates the abstract state appropriately. In our B-link tree example, say an operation seeking to delete 3 arrived at node y_0 and returned because 3 was not present, then any 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.

To generalize this argument to arbitrary search structures, we build on the concept of edgesets. The *pathset* of a path between nodes n_1 and n_2 is defined as the intersection of edgesets of every edge on the path, and is thus the set of keys for which operations starting at n_1 would arrive at n_2 assuming neither the path nor the edgesets along that path change. For example, the pathset of the path between r and n' in Fig. 2 is $(-\infty, \infty) \cap [5, \infty) = [5, \infty)$. With this, we define the *inset* of a node n, written $\mathsf{ins}(n)$, as the union of the pathsets of all paths from the root node to n (B-link trees may have several paths from the root to a given leaf node). Let the *outset* of n, $\mathsf{outs}(n)$, be the keys in the union of edgesets of edges leaving n. If we take the inset of a node n, and subtract the outset, we get the *keyset* of n, $\mathsf{ks}(n)$. Intuitively, the keyset of a node n is the set of keys that if present in the structure, must be in n. Coming back to our example, the keyset of node y_0 is $(-\infty, 4) \setminus [4, \infty) = (-\infty, 4)$, and so it suffices for the delete operation to ensure that 3 is not present in y_0.

We enforce the above interpretation of the keyset using the following *good state* conditions:

(GS1) The contents of every node are a subset of the keyset of that node.

(GS2) The edgesets of two distinct edges leaving a node are disjoint.

For data structures with a single root, (GS2) ensures that the keysets of two distinct nodes are disjoint. This, along with (GS1), tells us that we can treat the keyset of n as the set of keys that n can potentially contain. In good states, k is in the inset of n if and only if operations on k pass through n, and k is in the keyset of n if and only if operations on k end up at n. Given a good state, if an operation looks for, inserts, or deletes k at a node n such that k is in the keyset of a node n, then the keyset theorem of Shasha and Goodman [49] shows that the operation modifies the abstract state correctly.

How does the link template ensure that $k \in \mathsf{ks}(n)$ when `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 $k \in \mathsf{ins}(r)$, 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 split operations, this invariant breaks down because the inset of a node n shrinks after a split, so that k might have been in the $\mathsf{ins}(n)$ before the split but not afterwards. Note, however, that if one traverses the link edge, one can get back to a node with k in its inset. The way to formalize a more general invariant is to define the *inreach* of a node n as

$$\mathsf{inr}(n) \coloneqq \mathsf{ins}(n) \cup \bigcup_{n'} \mathsf{es}(n, n') \cap \mathsf{inr}(n').$$

Intuitively, $\mathsf{inr}(n)$ is the set of keys k for which if we follow edges labelled with k from n then we will eventually reach a node n' with $k \in \mathsf{ins}(n')$. For example, in Fig. 2 the inreach of y_1 is $[4, \infty)$ even though its inset is only $[4, 5)$, for it can reach the nodes with k in their inset for all $k \geq 4$ by following link edges. The invariant of the traversal is then that $k \in \mathsf{inr}(n)$. This is true in the beginning, for the $\mathsf{inr}(r) = \mathsf{ins}(r) = \mathsf{KS}$, and it is preserved by the traversal by the definition of inreach. When `findNext` returns `None`, the definition of inreach implies that $k \in \mathsf{inr}(n) \setminus \mathsf{outs}(n) \subseteq \mathsf{ks}(n)$, which by the keyset theorem gives us correctness of the decisive operation.

The edgeset framework and keyset theorem thus give us abstract conditions under which a template algorithm is correct. However, reasoning about insets and inreach is still challenging, because they are global inductively-defined quantities of the data structure. If we can write local pre- and post-conditions for helper functions such as `decisiveOp`, then the proof of an implementation can reason only about the node that the helper function modifies. In the rest of the paper, we show how to verify template algorithms whose correctness relies on these global quantities using local reasoning.

3 A Brief Introduction to ReLoC

A formal introduction to the ReLoC logic and the underlying programming language semantics is, unfortunately, out of the scope of this paper. We provide intuition for the key logical constructs and reasoning steps in this section, and

introduce the others as and when they are used; for more details see [24].

Proving correctness of data structures generally involves showing memory safety and functional correctness. In this paper, we prove that template algorithms such as the link template above satisfy a specification that encapsulates correctness as well as safety: contextual refinement. An implementation program e_1 contextually refines a specification program e_2 if and only if, for every possible client, each behavior when using e_1 is a possible behavior of e_2. Note that as our specification (a mathematical set ADT) is atomic, we can infer that our implementation is linearizable [23].

ReLoC is a higher-order relational separation logic that can be used to prove contextual refinement between concurrent programs e_1 and e_2. As usual, it is very difficult to directly prove contextual refinement, as it involves reasoning about an arbitrary client. ReLoC provides a stronger notion of logical refinement $\Delta \mid \Gamma \vDash_{\mathcal{E}} e_1 \lesssim e_2 : \tau$, which is a first-class judgement[1] in the logic. ReLoC also provides inference rules for deductive reasoning about logical refinement.

ReLoC is an extension of the Iris higher-order separation logic framework [33], which provides the usual separating conjunction $*$ and implication $-\!\!*$ connectives, and the *invariant assertion* $\boxed{P}^{\mathcal{N}}$.

3.1 Ghost State and Resource Algebras

Ghost state is a standard way to maintain auxiliary information necessary for a proof. Iris is built on the notion of higher-order ghost state, a powerful concept that allows us to specify both ownership of shared state as well as protocols for manipulating shared state. The prover can pick the ghost state needed for a particular proof, as long as the ghost state is chosen from a *camera*, a generalization of the partial commutative monoids that are commonly used to represent state in separation logics. As we do not use higher-order ghost state (i.e. state which can embed propositions), we restrict our attention to *resource algebras* (RAs), a stronger, but simpler, structure. All RAs are cameras, so the subsequent structures and proofs fit into the Iris framework. In the following definition, *Prop* is the type of propositions of the meta-logic (e.g. Coq).

Definition 3.1. A *resource algebra* (RA) is a tuple $(M, \overline{\mathcal{V}} : M \to \mathit{Prop}, |-| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$

$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$

$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$

$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-CORE-IDEM)}$$

$$\forall a, b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$

$$\forall a, b.\ \overline{\mathcal{V}}(a \cdot b) \Rightarrow \overline{\mathcal{V}}(a) \qquad \text{(RA-VALID-OP)}$$

where $M^? := M \uplus \{\bot\}$ $\qquad a^? \cdot \bot := \bot \cdot a^? := a^?$

$a \preccurlyeq b := \exists c \in M.\ b = a \cdot c$

Iris allows the prover to update the ghost state as long as the invariant that the composite ghost state is valid is maintained. Such updates are called *frame-preserving updates*, and one can do a frame-preserving update from $a \in M$ to $B \subseteq M$, written $a \rightsquigarrow b$, if

$$\forall a_{\mathrm{f}}^? \in M^?.\ \overline{\mathcal{V}}(a \cdot a_f^?) \Rightarrow \overline{\mathcal{V}}(b \cdot a_f^?).$$

Intuitively, this condition says that every *frame* a_f that is compatible with a should also be compatible with b. Thus, changing one's fragment of the ghost state from a to some b will not violate the assumptions made by anyone else.

We next show that the flow framework yields a resource algebra that enables frame-preserving updates for reasoning locally about properties relying on quantities such as insets and inreach.

4 A Flow Interface Resource Algebra

The flow framework [36] is a separation logic based approach for specifying and reasoning about unbounded data structures. The framework represents the heap as an abstract labeled graph, rather like the edgeset framework. Data structure invariants are expressed as local conditions satisfied by each node in the graph. These conditions are allowed to depend on the *flow* of the node, a quantity computed inductively over the entire graph. Unbounded regions of the heap are then abstracted using *flow interfaces* that specify the relies and guarantees that the region imposes on the rest of the heap to maintain the local flow invariants at each of its nodes. Proving that a program preserves the data structure invariants is done by showing that the modified region satisfies an equivalent flow interface. The approach avoids several limitations of common solutions to data structure abstraction, and allows unrestricted sharing and arbitrary traversals of heap regions. Flow interfaces are also able to express constraints on the contents of a data structure independently of its shape. A key advantage of this approach is that its proof rules are data-structure-agnostic, which allows us to formally prove an abstract algorithm like the link template without needing to commit to a particular implementation.

However, there are some challenges in using the original flow framework in tools like Iris and GRASShopper. This

[1] Δ is an interpretation for type variables, Γ assigns types to program variables, τ is the type of e_1 and e_2, and $\mathcal{E}$ is an invariant mask that will be explained in §5.

section presents a revised version of the flow framework tailored to meet the demands of such tools. We postpone discussing the rationale for these modifications to §7.

4.1 Flows

The first step in using flows is to identify a flow domain which is the domain from which the edge labels of the graph are drawn. It is also the domain of the flow of each node.

A *ring* $(D, +, \cdot, 0, 1)$ is a set D equipped with binary operators $+$ and $\cdot$ that are maps from $D \times D$ to D. The operation $+$ is called addition, and the operation $\cdot$ multiplication. The two operators must satisfy the following properties: (1) $(D, +, 0)$ is an abelian group with identity 0; (2) $(D, \cdot, 1)$ is a monoid with identity 1; and (3) multiplication left and right distributes over addition. A partially ordered ring is a ring D along with a partial order $\sqsubseteq$ such that for all $d_1, d_2, d_3 \in D$, (1) $d_1 \sqsubseteq d_2$ implies $d_1 + d_3 \sqsubseteq d_2 + d_3$, and (2) $0 \sqsubseteq d_1$ and $0 \sqsubseteq d_2$ implies $0 \sqsubseteq d_1 \cdot d_2$.

Definition 4.1 (Flow Domain)**.** A *flow domain* $(D, +, \cdot, \sqsubseteq, 0, 1)$ is a partially ordered ring.

We identify a flow domain with its support set D. We write D^+ for the set $\{d \in D \mid 0 \sqsubseteq d\}$ of non-negative elements in the ring. In the following, we assume that D is a flow domain.

Example 4.2. The integers $(\mathbb{Z}, +, \cdot, \leq, 0, 1)$ form a flow domain. Moreover, given a flow domain D, the functions $X \to D$ for any nonempty set X form a flow domain where all operations on D are lifted point-wise.

We abstract heaps using directed partial graphs. The graphs are partial because they describe abstractions of heap regions rather than the whole heap. In particular, a graph may have edges to *sink nodes*, which are not themselves part of the graph. Such edges abstract pointers to locations outside of the described heap region. The nodes in these graph are labeled from a lattice, A, so as to encode pertinent information contained in each node (such as the node's contents, lock-related information, etc.).

Definition 4.3 (Graphs)**.** Given a (potentially infinite) set of nodes Node, a *(partial) graph* $G = (N, N^o, \lambda, \varepsilon)$ consists of a finite set of nodes $N \subseteq \mathsf{Node}$, a finite set of *sink nodes* $N^o \subseteq \mathsf{Node}$ disjoint from N, a node labeling function $\lambda \colon N \to A$, and an edge function $\varepsilon \colon N \times (N \cup N^o) \to D^+$.

Note that the edge function is total on $N \times N^o$. The absence of an edge between two nodes n, n' is indicated by $\varepsilon(n, n') = 0$. We let $\mathsf{dom}(G) = N$ and sometimes identify G and $\mathsf{dom}(G)$ to ease notational burden. The (unique) graph defined over the empty set of nodes and sinks is denoted by G_e.

A *flow* of G is a function $\mathsf{flow}(in, G) \colon N \to D^+$ that is calculated as a certain fixpoint over G's edge function starting from a given *inflow* $in \colon N \to D^+$ going into G. We refer to this fixpoint as the *capacity* $\mathsf{cap}(G) : N \times (N \cup N^o) \rightharpoonup D^+$ of G, which is defined as the least fixpoint of the following equation:

$$\mathsf{cap}(G)(n, n') = \mathsf{init}(n, n') + \sum_{n'' \in G} \varepsilon(n, n'') \cdot \mathsf{cap}(G)(n'', n')$$
$$\text{where } \mathsf{init}(n, n') \coloneqq \mathsf{ITE}(n = n', 1, 0).$$

$\mathsf{cap}(G)(n, n')$ is essentially the sum over all paths between n and n' of the product of edge labels along each path. The capacity is a partial function because if G contains a path from n to n' via a cycle such that the product of edge labels is positive, then $\mathsf{cap}(G)(n, n')$ will not converge.

Given a graph G and an inflow in, the flow of a node $n \in N$, denoted $\mathsf{flow}(in, G)(n)$, is then defined by

$$\mathsf{flow}(in, G)(n) \coloneqq \sum_{n' \in G} in(n') \cdot \mathsf{cap}(G)(n', n).$$

We call a pair (in, G) *effectively acyclic* if for all sequences of nodes $n_1, \ldots, n_l \in G$ and $k \leq l$, $in(n_1) \cdot \varepsilon(n_1, n_2) \cdots \varepsilon(n_{l-1}, n_l) \cdot \varepsilon(n_l, n_k) = 0$. If (in, G) is effectively acyclic, then $\mathsf{flow}(in, G)$ is a total function on $\mathsf{dom}(G)$.

Note that this restriction on cycles is not the same as requiring that there be no cycles of pointers on the heap. We can still reason about structures such as doubly-linked lists and the Harris list [26], for although they have cycles, the edge labels can be chosen to set all cycles to have zero product.

Example 4.4. For our encoding of the edgeset framework, we use the *key counting* flow domain $\mathfrak{K} \coloneqq (\mathsf{KS} \to \mathbb{Z}, +, \cdot, \leq, \mathbf{0}, \mathbf{1})$, where the operations and order are lifted pointwise from $\mathbb{Z}$ and $\mathbf{0} \coloneqq \lambda k.0$ and $\mathbf{1} \coloneqq \lambda k.1$. Here we label each edge (n, n') in graph G by the function $\lambda k.\, \mathsf{ITE}(k \in \mathsf{es}(n, n'), 1, 0)$, which encodes the edgeset $\mathsf{es}(n, n')$ of the edge. For $in = \lambda n\, k.\, \mathsf{ITE}(n = r, 1, 0)$, which demands that the searches for all keys k in the global graph start at the root r, the flow $kc = \mathsf{flow}(in, G)$ will then tell us for every node n and key k, how many paths there are to the node n that a search for k may follow. In particular, we have $kc(n)(k) > 0$ iff k is in the inset of n.

We express the good state condition (GS1) by saying that every key k in n's contents (which are stored in the node label), $kc(n)(k) > 0$, and there is no edge to n' with k in its edgeset (which is encoded in the edge label $\varepsilon(n, n')$). Similarly, we can express (GS2) by saying for any key k and other nodes $n_1 \neq n_2$, k is not in the edgeset of at least one of the two edges from n to n_1, n_2.

4.2 Flow Graphs

We define a disjoint union on graphs, $G_1 \uplus G_2$, in the expected way. We want to reason about graphs and their flows locally to prove that invariants expressed in terms of the flow are preserved under modifications of subgraphs. That is, for a graph $G = G_1 \uplus G_2$, if G_1 is modified to some G_1', we want to be able to show, by reasoning only about G_1 and

G_1', that the modification does not affect the flow in G_2, i.e., $\mathsf{flow}(in, G))|_{G_2} = \mathsf{flow}(in, G')|_{G_2}$ for $G' = G_1' \uplus G_2$.

Clearly, to enable this kind of reasoning, we need to know how much of the global inflow *in* flows into the subgraph G_1 via G_2. We thus consider not just graphs but *flow graphs*, which are pairs (in, G) of a graph and its associated local inflow. Formally, let

$$\mathsf{FG} ::= H \in \{(in, G) \mid (in, G) \text{ is effectively acyclic}\} \mid H_\lightning$$

and let $H_\mathrm{e} = (in_\mathrm{e}, G_\mathrm{e})$ be the *empty flow graph* consisting of the empty inflow in_e and the empty graph G_e. Then two flow graph H_1 and H_2 compose to a flow graph $H_1 \bullet H_2 = (in, G)$ if $H_1 = (in_1, G_1)$, $H_2 = (in_2, G_2)$, $G = G_1 \uplus G_2$, and $\mathsf{flow}(in_i, G_i) = \mathsf{flow}(in, G)|_{G_i}$ for $i \in \{1, 2\}$. In all other cases we define $H_1 \bullet H_2 = H_\lightning$. This composition is uniquely determined by the ring properties of D.

Lemma 4.5. *Composition of flow graphs $\bullet$ is a commutative monoid with identity H_e.*

4.3 Flow Interfaces

Finally, we need a mechanism that enables us to abstract from the internal structure of a graph G_1 while preserving enough information to reason about the flow in a composite graph $G = G_1 \uplus G_2$. Consider again a program that modifies G_1 to G_1'. The key idea is that the internal structure of G_1 is irrelevant for reasoning about the flow in G_2. What matters is how much flow G_1 routes between any of its source and sink nodes but not what paths this flow takes inside of G_1.

In our search structure example, take a key k and a path from n, a source of G_1, to n', a sink of G_1, with k is in its pathset. If for every such k, n, and n', G_1' also has some path from n to n' with k in its pathset, then it is not hard to see that the inset of every node in G_2 is preserved in $G' = G_1 \uplus G_2'$.

Formally, we define the *flow map* of a graph to be its capacity restricted to source-sink pairs:

$$\mathsf{flm}((in, G)) := \mathsf{cap}(G)|_{\{n \in N \mid in(n) \neq 0\} \times N^o}.$$

A graph is then abstracted by its *flow interface*:

Definition 4.6 (Flow Interface)**.** Given a flow graph $H \in \mathsf{FG}$, its *flow interface* is a tuple consisting of its inflow, the join of all its node labels, and its flow map:

$$\mathsf{int}(H) := (in, \sqcup_{n \in G} \lambda(n), \mathsf{flm}(in, G)) \quad \text{where } H = (in, G) \text{ and } G = (N, N^o, \lambda, \varepsilon).$$

The set of all flow interfaces is $\mathsf{FI} := \{\mathsf{int}(H) \mid H \in \mathsf{FG}\}$.

The following lemma allows us to lift flow graph composition to flow interfaces:

Lemma 4.7. $\mathsf{int}(H_1) = \mathsf{int}(H_1') = I_1 \wedge \mathsf{int}(H_2) = \mathsf{int}(H_2') = I_2 \Rightarrow \mathsf{int}(H_1 \bullet H_2) = \mathsf{int}(H_1' \bullet H_2')$.

As described earlier, we encode inductive properties of data structures using local conditions on flows, which we formalize using a *good condition* $\nu(n, I_n)$ that takes a node n and its interface. That is, $I_n = (in_n, a_n, f_n)$ where in_n is n's flow, a_n is n's node label, and f_n provides the labels of n's outgoing edges.

Given a good condition ν, we can filter the set of flow graphs to those that satisfy ν:

$$\mathsf{FG}_\nu := \{H \in \mathsf{FG} \mid \forall n \in H.\ \nu(n, (in_n, \lambda(n), \varepsilon_n))\} \mid H_\lightning$$

where $in_n := \{n \rightarrowtail \mathsf{flow}(H)(n)\}$ $\quad (in, (N, N^o, \lambda, \varepsilon)) := H$

and $\varepsilon_n := \{(n, n') \rightarrowtail \varepsilon(n, n') \mid n' \in N \cup N^o, \varepsilon(n, n') \neq 0\}$

The following lemma states that the condition ν is preserved under flow interface composition, which is the critical piece for enabling local reasoning about flow-based inductive properties of a data structure.

Lemma 4.8. *For all good conditions ν, if $H_1, H_2 \in \mathsf{FG}_\nu$ then $H_1 \bullet H_2 \in \mathsf{FG}_\nu$.*

We can now define the flow interface algebra, and show it is a resource algebra. Given a good condition ν, we define

$$\mathsf{FI}_\nu ::= I \in \{\mathsf{int}(H) \mid H \in \mathsf{FG}_\nu\} \mid I_\lightning \qquad \overline{\mathcal{V}}(a) := a \neq I_\lightning$$

$$|I| := I_e := \mathsf{int}(H_\mathrm{e}) \qquad |I_\lightning| := I_\lightning \qquad I_1 \oplus I_2 :=$$

$\mathsf{int}(H_1 \bullet H_2)$ for any H_1, H_2 s.t. $\mathsf{int}(H_1) = I_1 \wedge \mathsf{int}(H_2) = I_2$.

Theorem 4.9. *For every good condition ν, the flow interface algebra $(\mathsf{FI}_\nu, \overline{\mathcal{V}}, |-|, \oplus)$ is a resource algebra.*

When modifying a flow graph H to another flow graph H', requiring that H' satisfies the same interface $\mathsf{int}(H)$ is too strong a condition for verifying many data structure algorithms. Instead, we want to allow $\mathsf{int}(H')$ to differ from $\mathsf{int}(H)$ as long as it is *contextually equivalent* with respect to the flow. Since we only care about the flow map from source nodes that receive a non-zero inflow, we can weaken the requirement to the flow maps of the interfaces being equal only for such source nodes. We further generalize the definition to allow H' to have a larger inflow than H as long as this does not affect the outgoing flow.

Formally, we say interface (in, a, f) is *contextually extended* by (in', a', f'), written $(in, a, f) \precsim (in', a', f')$, if and only if $\mathsf{dom}_1(f) = \mathsf{dom}_1(f')$ and

$$\forall n \in \mathsf{dom}_1(f).\ in(n) \sqsubseteq in'(n) \text{ and}$$
$$\forall (n, n') \in \mathsf{dom}(f).\ (0 \sqsubset in(n) \Rightarrow f(n, n') = f'(n, n')) \wedge (in(n) \sqsubset in'(n) \Rightarrow in(n) \cdot f(n, n') = in'(n) \cdot f'(n, n'))$$

The following theorem states that contextual extension preserves composability and is itself preserved under interface composition.

Theorem 4.10 (Replacement)**.** *If $I = I_1 \oplus I_2$, and $I_1 \precsim I_1'$, then there exists $I' = I_1' \oplus I_2$ such that $I \precsim I'$.*

The Replacement Theorem enables frame-preserving updates of flow interfaces in Iris.

5 Verifying Search Structure Templates

We now tie together flow interfaces and ReLoC and show how to use them to verify template algorithms for concurrent search structures. We discuss the proof of the link template from §2 in depth and then provide a summary of the other case studies and our implementation in the next section.

5.1 Encoding the Edgeset Framework using Flows

We represent a search structure state abstractly as the set of keys C that it contains. The specification of the search structures operations is given below, in terms of the initial set of keys C, the resulting set of keys C', the query key k, and the value returned res:

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

Our first task is to provide an encoding of the edgeset framework using local conditions on flows that enables us to lift a proof that an operation correctly updated the contents of a single node in the search structure to a proof that it correctly updated the contents of the data structure as a whole.

Following Example 4.4, we use the key counting flow domain to encode the inset as a flow of each node. As mentioned in §2, it is hard to define the inreach as a flow, so we show here how to encode a sufficient under-approximation of the inreach. The 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 of only 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 key count domains as the flow domain, where the first component calculates the inset as described above. The roots of the second component are the first nodes on each level (as shown in Fig. 2), and the resulting flow at each node n is called the *linkset* of n, denoted $\mathsf{lnks}(n)$. 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 in the B-link tree, the linkset happens to be equal to the inreach. In general, we only require the following properties of the linkset, which we enforce in the local good condition on nodes: First, if $k \in \mathsf{lnks}(n) \setminus \mathsf{ins}(n)$, then the edge labeled k is the same in the inset and the linkset components. This is used to prove that the traversal using `findNext` maintains the invariant that the query key $k \in \mathsf{ins}(n) \cup \mathsf{lnks}(n)$. Second, if $k \in \mathsf{lnks}(n) \setminus \mathsf{outs}(n)$, then $k \in \mathsf{ins}(n) \Rightarrow k \in \mathsf{ks}(n)$, which implies that when `findNext` fails, we have found the right node n.

Before describing the good node condition, we introduce some shorthands for clarity:

$$\begin{aligned} \mathsf{ins}(I, n) &\coloneqq \{k \mid I^{in}(n)_{\mathsf{is}}(k) \geq 1\} \\ \mathsf{lnks}(I, n) &\coloneqq \{k \mid I^{in}(n)_{\mathsf{ls}}(k) \geq 1\} \\ \mathsf{outs}(I) &\coloneqq \{k \mid \exists n, n'.\ I^f(n, n')_{\mathsf{is}}(k) \geq 1\} \\ \mathsf{es}(I_n, n, n') &\coloneqq \{k \mid I^f(n, n')_{\mathsf{is}}(k) \geq 1\} \\ \mathsf{C}(I) &\coloneqq \textbf{let } (C, \_) = I^a \textbf{ in } C \\ \mathsf{inr}(I, n) &\coloneqq \textbf{let } (\_, \rho) = I^a \textbf{ in } \rho(n) \end{aligned}$$

where d_{is} and d_{ls} denote the inset and linkset component of the flow domain element d.

We obtain the desired interpretation of linkset and enforce the global good state conditions using the following local good condition on nodes:

$$\begin{aligned} &\nu(n, I) \coloneqq \mathsf{C}(I) \subseteq \mathsf{ins}(I, n) \setminus \mathsf{outs}(I) && (1) \\ &\quad \land (\forall n', n''.\ n' = n'' \lor \mathsf{es}(I, n, n') \cap \mathsf{es}(I, n, n'') = \emptyset) && (2) \\ &\quad \land (\forall k, n'.\ k \in \mathsf{inr}(I, n) \setminus \mathsf{ins}(I, n) \Rightarrow \\ &\qquad I^f(n, n')_{\mathsf{ls}}(k) = I^f(n, n')_{\mathsf{is}}(k)) && (3) \\ &\quad \land \mathsf{lnks}(I, n) \subseteq \mathsf{ins}(I, n) \cup \mathsf{outs}(I, n) && (4) \\ &\quad \land I^a = (\_, \{n \rightarrowtail \mathsf{ins}(I, n) \cup \mathsf{lnks}(I, n)\}). && (5) \end{aligned}$$

Here, conditions (1) and (2) encode the good state conditions (GS1) and (GS2). (3) and (4) are the two constraints on the linkset that we described earlier. Finally, (5) uses the node label to keep track of the inreach-approximation $\mathsf{lnks}(n) \cup \mathsf{ins}(n)$. We encode this set as partial functions from nodes to sets of keys: $(\text{Node} \rightharpoonup 2^{\mathsf{KS}} \uplus \{\top\}, \sqsubseteq)$, where

$$\rho_1 \sqsubseteq \rho_2 :\Leftrightarrow \rho_2 = \top \lor \rho_1 \neq \top \neq \rho_2 \land \rho_1 = \rho_2|_{\mathsf{dom}(\rho_1)}.$$

Note that this means that the join of two functions is $\top$ if they disagree on the inreach of any node. This gives us the property that

$$I = I_n \oplus \_ \land \mathsf{dom}(I_n) = \{n\} \land I^a \neq \top \Rightarrow I^a(n) = {I_n}^a(n), \quad (6)$$

which we will use in our proof.

We also require the following constraints on the global interface:

$$\varphi(I) \coloneqq (\forall n, k. I^{in}(n)_{\mathsf{is}}(k) = \mathsf{ITE}(n = r, 1, 0)) \land I^a_{\mathsf{ir}} \neq \top \land I^f = \epsilon$$

This says that in the inset flow domain component, the global inflow assigns a key count of 1 to the root r, and 0 for every other node, for all keys (i.e. all searches start at the root). It does not restrict the global inflow in the linkset component. We require the inreach function computed in the node label, I^a_{ir}, to not be $\top$, and finally we require that the global interface is closed (i.e. has no outgoing edges).

Putting all this together, we obtain the following lemma, which is the core of the Keyset Theorem from [49]. This lemma allows us to check that the search structure specification Ψ_ω is satisfied for the interface of a single node that has the query key in its keyset, and automatically lift it up to the interface of the entire data structure.

Lemma 5.1. *Given* $I, I', I_n, I'_n, I_2 \in \mathsf{FI}_v$, $k \in \mathsf{KS}$, $n \in \mathsf{Node}$, *and* res *such that*

- $\varphi(I)$ *holds and* $\mathsf{dom}(I_n) = \{n\}$,
- $k \in \mathsf{ins}(I_n, n)$ *and* $k \notin \mathsf{outs}(I_n)$, *and*
- $I = I_n \oplus I_2$ *and* $I' = I'_n \oplus I_2$,

$\Psi_\omega(k, \mathsf{C}(I_n), \mathsf{C}(I'_n), \mathsf{res}) \Rightarrow \Psi_\omega(k, \mathsf{C}(I), \mathsf{C}(I'), \mathsf{res})$.

5.2 An Invariant for the Link Template

We encode the invariants of the link template algorithm with the help of ghost state. Our ghost state uses a few standard resource algebras of Iris: the set RA and the authoritative RA. Given any set S, it is easy to see that we have an RA $(2^S, \mathsf{True}, \lambda X.\emptyset, \cup)$, which permits a frame-preserving update $X \rightsquigarrow Y$ for any $X, Y \subseteq S$. We will use a set RA in our proof to keep track of the global contents, which is the state of the specification program.

Given an RA M, the authoritative RA $\textsc{Auth}(M)$ (see [32] 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 $b \preccurlyeq a$. This can be used to model, for example, a 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. We use an authoritative RA of sets of keys in our proofs to keep track of the inreach-approximation of individual nodes, which satisfies the following properties:

$$\textsc{auth-set-upd}\ \frac{X \subseteq Y}{\bullet X \rightsquigarrow \bullet Y} \qquad \textsc{auth-set-snapshot}\ \bullet X \rightsquigarrow \bullet X \cdot \circ X \qquad \textsc{auth-set-valid}\ \frac{\mathcal{V}(\bullet X \cdot \circ Y)}{Y \subseteq X}$$

We also use an authoritative RA of flow interfaces $\textsc{Auth}(\mathsf{FI}_v)$ to keep track of the flow interface ghost state of our algorithms. Using Theorem 4.10, we can show that this RA permits the following non-deterministic frame-preserving update (which is a generalization of the standard frame-preserving update presented in §3, for details see [32]):

$$\textsc{auth-fi-upd}\ \frac{I_1 \precsim I'_1}{(\bullet\, I, \circ\, I_1) \rightsquigarrow \left\{(\bullet\, I', \circ\, I'_1) \;\middle|\; I \precsim I' \wedge \exists I_2.\ I = I_1 \oplus I_2 \wedge I' = I'_1 \oplus I_2\right\}}$$

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. The prover is allowed to allocate ghost state at any unused location and pick the RA from which the values stored are drawn. 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.

We can now put everything together and construct an invariant sufficient to prove correctness of the link template algorithm:

$$\mathsf{Inv} \coloneqq \exists I.\ \boxed{\bullet I}^{\gamma_I} * \varphi(I) * \boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f} * \ast_{n \in I} \boxed{\bullet \mathsf{inr}(I, n)}^{\gamma_{i(n)}}$$
$$* \ast_{n \in I} \exists b.\ \ell(n) \mapsto_i b * \mathsf{ITE}\,(b, \mathsf{True}, \exists I_n.\ \mathsf{Node}(n, I_n, I_n)) * \boxed{\mathsf{C}(I)}^{\gamma_c}$$
$$\mathsf{Node}(n, I_n, I'_n) \coloneqq \exists N.\ \boxed{\circ N}^{\gamma_f} * n \in N * \boxed{\circ I_n}^{\gamma_I} * \hbar(n, I'_n)$$

Our invariant uses a few different types of ghost states in order to capture the state of the link algorithm, the state of the specification, and the relation between them. First, we use the $\textsc{Auth}(\mathsf{FI}_v)$ RA at location γ_I to keep track of the flow graph abstraction. The invariant always owns the authoritative version $\boxed{\bullet I}^{\gamma_I}$, which is the interface of the global graph, and satisfies $\varphi(I)$. For every node $n \in I$, there is a heap location $\ell(n)$ that is set to True iff node n is locked. If n is unlocked, the invariant owns the fragment version of n's interface $\boxed{\circ I_n}^{\gamma_I}$ and n's heap representation $\hbar(n, I_n)$.

We use an authoritative RA of sets of nodes at location γ_f to encode the footprint of the global graph. The invariant owns the authoritative version $\boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f}$, which is the domain of the global interface. This allows threads to take snapshots of the footprint and assert locally that a given node is in the footprint.

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 allows threads to assert that a key is in the inreach of a given node when it is unlocked.

Finally, we use a sets of keys RA at location γ_c in order to encode the state of the specification program spec. The relation between the link algorithm and the specification is that the state of the specification must match the set of keys of the implementation, $\mathsf{C}(I)$. We specify spec in ReLoC for the search structure operation ω as:

$$\textsc{search-structure-spec}\ \frac{\boxed{C}^{\gamma_c}\ \forall v, C'.\ \boxed{C'}^{\gamma_c} * \Psi_\omega(k, C, C', v) \twoheadrightarrow (\Delta \mid \Gamma \vDash_{\mathcal{E}} e \precsim K[v] : \tau)}{\Delta \mid \Gamma \vDash_{\mathcal{E}} e \precsim K[\mathsf{spec}\ \ \omega\ k] : \tau}$$

5.3 Link Template Proof in ReLoC

Our aim is to prove (where 2 is the type of Booleans)

$$\boxed{\mathsf{Inv}}^{\mathcal{N}} \twoheadrightarrow \Delta \mid \emptyset \vDash (\mathtt{searchStrOp}\ \ \omega\ \ \mathtt{r}\ \ \mathtt{k}) \precsim (\mathtt{spec}\ \ \omega\ \ \mathtt{k}) : \mathtt{2}$$

which implies that the template algorithm contextually refines the abstract specification.

This refinement proof proceeds by unfolding the definition of the function in question, and then applying the ReLoC proof rules for each programming language construct in sequence, almost like symbolic execution. State is shared between threads by using the invariant assertion $\boxed{\mathsf{Inv}}^{\mathcal{N}}$. Invariants are tagged with *name spaces* (our invariant is

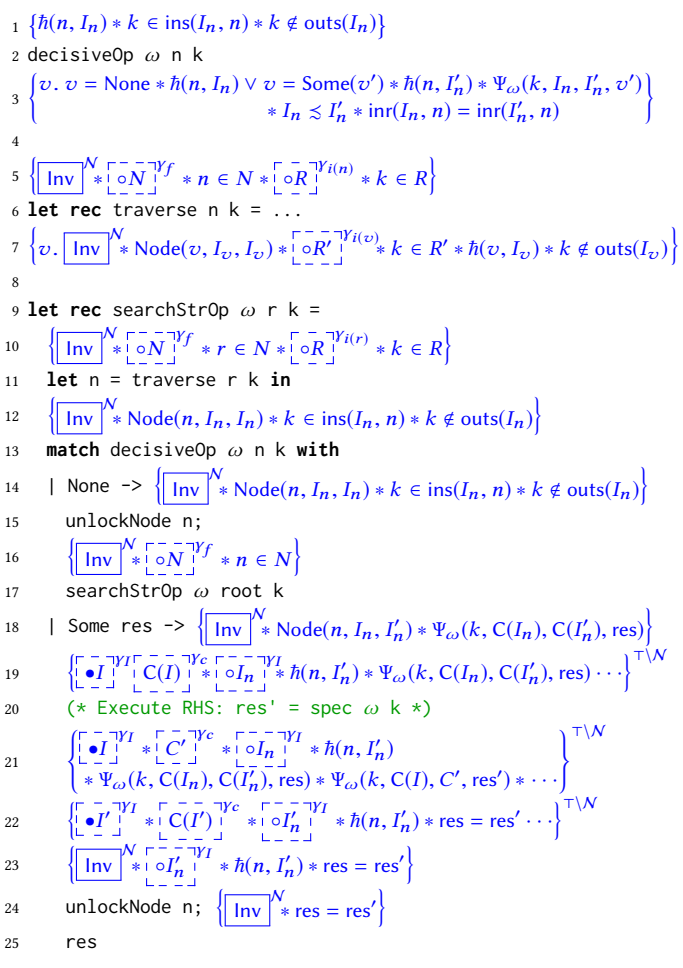

Figure 4. The link template algorithm with a proof outline.

tagged with the name space $\mathcal{N}$) so as to keep track of which invariants are currently open. This is to enforce that the invariant is reestablished in between atomic steps of the program. This is achieved by tagging the refinement judgement $\Delta \mid \Gamma \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ with a mask $\mathcal{E}$ of available invariants. E.g. the mask $\top \setminus \mathcal{N}$ encodes the fact that $\mathcal{N}$ has been opened and is no longer available. We represent this as superscripts on our Hoare-style assertions, and omit the superscript if the mask is $\top$, the set of all invariant names.

Our proof assumes that implementations provide certain helper functions that satisfy certain specifications, such as that of `decisiveOp` shown at the top of Fig. 4. Note that we have presented these specifications in the more familiar Hoare triple style $\{P\}\ e_1\ \{v.\ Q\}$, which in ReLoC is encoded as:

$$\frac{P \qquad (\forall v.\ Q \mathrel{-\!\!*} \Delta \mid \Gamma \vDash K[v] \precsim e_2 : \tau)}{\Delta \mid \Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

We also use a standard lock module, similar to the one in [24], adapted to a setting where each node n in our data structure has its own lock that protects the interface and heap representation of n, $\boxed{\circ I_n}^{\gamma_I} * \hbar(n, I_n)$.

Fig. 4 shows a Hoare-style proof outline, where the intermediate assertions in braces show the context of the proof (the premises that are currently available). The proof begins by applying the specification of `traverse` to symbolically execute the call on line 11. To satisfy the precondition, we need to open the invariant and use the fact that $\varphi(I) \Rightarrow r \in \mathrm{dom}(I) = N$. We can then take a snapshot of the domain of the global invariant using AUTH-SET-SNAPSHOT, and we add $\boxed{\circ N}^{\gamma_f} * r \in N$ to our context. We also use $r \in \mathrm{dom}(I)$ to unfold the iterated separating conjunction containing the inreach sets in the invariant, and take a snapshot of the inreach set of r using AUTH-SET-SNAPSHOT. Note that $\varphi(I) \Rightarrow I^{in}(r)_{\mathsf{is}}(k) = 1$ which by the invariant gives us $k \in \mathsf{inr}(I, r) = R$. The resulting context is depicted in line 10.

After symbolically executing `traverse`, we obtain its postcondition on line 7. The next step is to symbolically execute `decisiveOp`, for whose precondition we need to show that $k \in \mathsf{ins}(I_n, n)$. This follows from $k \in R \wedge k \notin \mathsf{outs}(I_n)$ by the definition of the invariant, the inreach node label property (6), and (4).

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 its specification. This involves giving up the fragment interface and the heap representation of n, $\boxed{\circ I_n}^{\gamma_I} * \hbar(n, I_n)$. We use the induction hypothesis on the recursive call to `searchStrOp` on line 17, completing this branch of the proof.

If `decisiveOp` succeeds, we get back a modified heap representation along with some constraints on the new interface I'_n (line 18). Since we have modified the search structure, we must now execute the specification program in order to maintain the invariant between the state of the implementation and that of the specification. This is essentially the *linearization point* of this algorithm.

To do this, we first open the invariant, allowing us to temporary access to its contents and setting the mask to $\mathcal{E} \setminus \mathcal{N}$ (line 19). We now have the resources to execute the specification program, using SEARCH-STRUCTURE-SPEC. This changes the set ghost state from $\mathsf{C}(I_n)$ to C' and tells us they are related by the predicate Ψ_ω.

We cannot yet close the invariant, for while the heap representation uses the interface I'_n, the ghost location γ_I still contains the old interface I_n. So we now perform a frame-preserving update to the authoritative flow interface ghost location, using AUTH-FI-UPD and the fact that I'_n contextually extends I_n according to the postcondition of `decisiveOp`. We also need to establish the relation between the implementation state and the specification state, namely $\mathsf{C}(I') = C'$. We do this with the help of Lemma 5.1, which gives us

$\Psi_\omega(k, \mathrm{C}(I), \mathrm{C}(I'), \mathrm{res})$. Note that Ψ_ω determines the new contents and the result uniquely, i.e.

$$\Psi_\omega(k, C, C'_1, \mathrm{res}_1) \wedge \Psi_\omega(k, C, C'_2, \mathrm{res}_2) \Rightarrow C'_1 = C'_2 \wedge \mathrm{res}_1 = \mathrm{res}_2.$$

This gives us the context in line 22, from which we can close the invariant.

We finally execute the call to `unlockNode` as above, and are left with the goal $\Delta \mid \Gamma \vDash \mathtt{res} \lesssim \mathtt{res'} : \tau$ which we can show using $\mathrm{res} = \mathrm{res}'$.

The proof of `traverse` is a similar symbolic execution of the implementation program using the respective specifications of the helper functions, except that it does not need to execute the specification program. We omit a description for space reasons.

5.4 Proofs of Template Implementations

To obtain a verified implementation of the link template, one needs to provide code for the helper functions that satisfies their specifications. As can be seen from the specification of `decisiveOp` in Fig. 4, these functions do not have access to the invariant and only have access to the heap representation of the given node. Thus, if their implementations are sequential code, one expects to be able to verify them using an off-the-shelf separation logic tool that can verify sequential heap-manipulating code.

We can formally justify such a sequential proof using the existing ReLoC rule:

LR-WP-ATOMIC-L

$$\frac{{}^{\top}\Rrightarrow^{\mathcal{E}} \mathrm{wp}_{\mathcal{E}}\, e\, \{v.\, \Delta \mid \Gamma \vDash_{\mathcal{E}} K[v] \lesssim e' : \tau\} \ \mathrm{atomic}(e) \ \ \mathrm{closed}(e)}{\Delta \mid \Gamma \vDash K[e] \lesssim e' : \tau}$$

where $\mathrm{wp}_{\mathcal{E}}\, e\, \{v.\, P\}$ is the weakest precondition of expression e returning a value v under a postcondition P and an invariant mask $\mathcal{E}$. If we use this rule with $\mathcal{E} = \top$, this tells us that we can symbolically execute the left hand side program of the refinement relation using standard separation logic weakest precondition rules, as long as the program only uses those resources that are not locked up in some invariant. Thus, we can take a proof produced by a standard SL tool and lift it to a ReLoC proof:

$$\frac{\{P\}\ e_1\ \{v.\, Q\} \qquad P \qquad (\forall v.\, Q \mathrel{-\!\!*} \Delta \mid \Gamma \vDash K[v] \lesssim e_2 : \tau)}{\Delta \mid \Gamma \vDash K[e_1] \lesssim e_2 : \tau}$$

6 Proof Mechanization and Automation

In addition to the link template presented in this paper, we have also verified the give-up and lock-coupling template algorithms from [49], as depicted in Fig. 1. For the link template and give-up template, 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 ReLoC [24]. The implementations of the helper functions for the concrete implementations that are assumed in the template algorithms (e.g. `decisiveOp`, `findNext`, etc.) have been verified using the separation logic based deductive program verifier GRASShopper [44]. 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 strong enough to support all the reasoning that we do in GRASShopper, but comes with significant additional manual effort. Furthermore, our automation of implementation proofs in GRASShopper shows that our revised flow framework is also suitable for reasoning in SMT-based automated tools.

The Coq formalization assumes that flow interfaces form an RA (Theorem 4.9) and that they enable frame preserving updates (Theorem 4.10) as well as some basic general lemmas about flow interfaces.The template proofs parameterize over the implementation of the helper functions, the heap representation predicate $\hbar$ as well as the actual flow domain, node label domain, and good condition ν.

All properties involving the specific flow domain elements and ν 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. The rationale for this, apart from increased proof automation, is that some maintenance operations on the concrete data structures require us to prove additional data structure specific invariants. For example, the split operation for B+ trees requires the data structure to be a tree. As the template proofs are parametric on the flow domains used, we can incorporate these invariants to the implementation proof using product flow domains without having to redo the proofs of the templates. To automate the implementation proofs and auxiliary lemmas in GRASShopper, we extended the tool with support for general maps and algebraic data types as they are needed to formalize flow interfaces.

Table 1 provides a summary of our development. 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 is relatively large for such a simple data structure. The reason is that the insertion operation, which adds a new node to the list, requires the calculation of a new flow interface for the region obtained

Table 1. Summary of templates and instantiations verified in Coq (above line) and GRASShopper (below). 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
Link template	29	518	547	70
Give-up template	35	406	441	25
Lock-coupling template	37	565	602	44
Total	**101**	**1489**	**1590**	**139**
Flow library	-	144	-	-
Array library	134	305	439	16
B+ tree	60	175	235	16
B-link tree	81	174	255	137
Hash table (link)	45	184	229	8
Hash table (give-up)	49	117	166	12
Lock-coupling list	69	211	280	153
Total	**438**	**1310**	**1748**	**342**

after the insertion. 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, 43]. 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. We note that the size of the proof could be significantly reduced with a few simple tactics for quantifier expansion. Nevertheless, one can see that the additional automation provided by the tool reduces the overall manual proof effort compared to an interactive proof assistant like Coq.

7 Related Work

Our work builds on the Iris separation logic [32], the ReLoC logic [24] for expressing refinement proofs in Iris, and the flow framework [36] for expressing compositional abstractions of complex data structures. 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/ReLoC, and a revision of the flow framework that is geared towards proof mechanization and automation. We also strengthen the replacement theorem of [36] to enable reasoning about a broader range of data structure updates.

There were two main obstacles to automating the original flow framework: (1) flow graphs could compose to more than one flow graph that has the same flow, and (2) reasoning about flow interfaces involved calculating solutions of a fixpoint equation that does not converge in finitely many steps. For example, (1) introduces an additional quantifier alternation when proving the premise of the replacement theorem for a modified heap region, while (2) means that establishing that a flow interface is preserved involves reasoning about fixpoints even if the modified region is bounded. We eliminated these obstacles by restricting flow domains to rings rather than semi-rings and by considering only effectively acyclic flow graphs. We found that, in practice, these restrictions make only small compromises on expressivity while greatly increasing the potential for automation. In particular, with these modifications we can now automatically prove that modified regions preserve their interfaces using an SMT solver, which we make use of when verifying template implementations in GRASShopper.

There exist many other concurrent separation logics that help modularize the correctness proofs of concurrent systems [5, 15, 18, 22, 28, 38, 45, 52, 53]. 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.

We verify correctness of concurrent data structures by showing that they refine a sequential specification. Most existing techniques instead focus on proving linearizability [27]. The connection between contextual refinement and linearizability was established in [23]. The template algorithms that we have verified focus on lock-based techniques with fixed linearization points inside a decisive operation. The give-up and link templates can be generalized to handle lock-free data structures. Though, many lock-free data structures have non-fixed linearization points, which ReLoC currently cannot reason about. Much work has been dedicated to handling non-fixed as well as external linearization points [6, 8, 12, 16, 19, 34, 37, 41, 54]. 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 as our approach does not critically depend on the use of ReLoC. Our proof methodology can be replicated in other separation logics that support user-defined ghost state, such as FCSL [47], which would also be useful if one wanted to extend this work to non-linearizable data structures [48].

Fully automated proofs of linearizability by static analysis and model checking have been mostly confined to simple list-based data structures [1, 3, 7, 11, 21, 51]. 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.

8 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 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

[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)*, 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)*, 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)*, 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)*, Vol. 9207. 87–105. https://doi.org/10.1007/978-3-319-21668-3_6

[5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

[6] 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)*, Vol. 7792. Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

[7] 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)*, Vol. 9135. Springer, 95–107. https://doi.org/10.1007/978-3-662-47666-6_8

[8] 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)*, Vol. 10427. Springer, 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

[9] 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

[10] Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

[11] 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)*, Vol. 6174. Springer, 465–479. https://doi.org/10.1007/978-3-642-14295-6_41

[12] 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

[13] The Coq Development Team. 2017. *The Coq Proof Assistant Reference Manual, version 8.7*. http://coq.inria.fr

[14] 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*. ACM, 845–864. https://doi.org/10.1145/2048066.2048131

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

[16] 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 fuer Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

[17] 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*. ACM, 287–300. https://doi.org/10.1145/2429069.2429104

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

[19] 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*. ACM, 233–246. https://doi.org/10.1145/2676726.2676963

[20] 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, Article 4 (Jan. 2016), 72 pages. https://doi.org/10.1145/2818638

[21] 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)*, Vol. 8044. Springer, 174–190. https://doi.org/10.1007/978-3-642-39799-8_11

[22] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *16th European Symposium on Programming, ESOP 2007 (Lecture Notes in Computer Science)*, Vol. 4421. Springer, 173–188. https://doi.org/10.1007/978-3-540-71316-6_13

[23] 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

[24] 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

[25] 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

[26] 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)*, Vol. 2180. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

[27] 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

[28] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013 (Lecture Notes in Computer Science)*, Vol. 7737. Springer, 315–334. https://doi.org/10.1007/978-3-642-35873-9_20

[29] 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

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

[31] 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

[32] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Submitted for publication* (2017).

[33] 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 '15)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

[34] 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)*, Vol. 10201. Springer, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

[35] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[36] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *Proc. ACM Program. Lang.* 2, POPL, Article 37 (Jan. 2018), 31 pages. https://doi.org/10.1145/3158125

[37] 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*. ACM, 459–470. https://doi.org/10.1145/2462156.2462189

[38] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *23rd European Symposium on Programming, ESOP 2014 (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[39] 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

[40] 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. http://dx.doi.org/10.1007/3-540-44802-0_1

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

[42] 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

[43] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *CAV (LNCS)*, Vol. 8044. Springer, 773–789.

[44] 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 (Lecture Notes in Computer Science)*, Vol. 8413. Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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)*, Vol. 5403. Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27

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

[53] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 964–990. https://doi.org/10.1007/978-3-662-54434-1_36

[54] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (2) (Lecture Notes in Computer Science)*, Vol. 9207. Springer, 3–19.