Verifying Concurrent Search Structure Templates

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

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 formalizing three concurrent search structure templates, based on link, give-up, and lock-coupling synchronization, and deriving verified implementations based on B-trees, hash tables, and linked lists. These case studies represent 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. Our verification is split between the Coq proof assistant and the deductive verification tool GRASShopper in order to demonstrate that our proof technique and framework can be applied both in fully mechanized proof assistants as well as automated program verifiers. 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 [Hoare 1969] enables proofs to be compositional in terms of program structure; (ii) separation logic [O’Hearn et al. 2001; Reynolds 2002] allows proofs of programs to be local in terms of the state they modify; and (iii) thread modular techniques [Herlihy and Wing 1990; Jones 1983; Owicki and Gries 1976] allow one to reason about each thread in isolation.

Concurrent separation logics [Brookes 2007; Brookes and O’Hearn 2016; da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2013, 2010; Dodds et al. 2016; Feng et al. 2007; Fu et al. 2010; Jung et al. 2015; Nanevski et al. 2014; O’Hearn 2007; Svendsen and Birkedal 2014; Vafeiadis and Parkinson 2007] 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 [da Rocha Pinto et al. 2011]. Such proofs, however, remain large, complex, and on paper; verified 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 [Shasha and Goodman 1988] 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 concrete data structure is represented in memory. Consequently, existing proofs such as the one of the B-link tree in [da Rocha Pinto et al. 2011] intertwine the synchronization invariants and the memory invariants, which makes the proof complex, hard to mechanize, and difficult to reuse.

This paper shows how to adapt and combine recent advances in compositional abstractions, separation logic, and refinement proofs in order to achieve the envisioned algorithmic proof modularity for an important class of concurrent data structures: search structures.

A search structure is a data structure that supports fast search, insert, and delete operations on a set of key-value pairs (e.g. sorted lists, binary search trees, hash tables, and B-trees). We present a methodology for specifying and verifying template algorithms for concurrent search structures that abstract from the concrete low-level representation of the data structure in memory. The methodology independently verifies that (1) the template algorithm satisfies the 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 verification uses SL-style reasoning for both subtasks and the methodology is designed to be usable within off-the-shelf SL-based verification tools so that the proofs can be mechanically checked and automated. Moreover, 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 base our work on the template algorithms for concurrent search structures by Shasha and Goodman [1988], who identified the key invariants needed for decoupling reasoning about
Verifying Concurrent Search Structure Templates

![Diagram of proof structure](image)

**Fig. 1.** The structure of our proofs.

synchronization and memory representation for such data structures. The second ingredient is the recently proposed flow framework [Krishna et al. 2018], an SL-based abstraction mechanism that allows one to reason about global inductive invariants of general graphs in a local manner. The flow framework enables us to formalize the correctness argument of Shasha and Goodman [1988] within separation logic. Krishna et al. [2018] use the flow framework to verify a template algorithm based on the so-called give-up technique. However, the proof is only on paper and it depends on a bespoke program logic. This logic requires new reasoning primitives that are not supported by the logics implemented in existing SL-based tools. In addition, the underlying semantic model of the logic makes it hard to use equational reasoning, which is a critical prerequisite for enabling proof automation.

To enable tool support for our proposed methodology, we therefore adapt and simplify the flow framework so that it fits within the confines of existing SL-based verification tools and engineer solutions for specific challenges that arise in the formalization of the complex invariants that the template algorithms depend on. Specifically, our new flow framework presented in §3 simplifies the meta theory of the original flow framework and can now be embedded directly into existing separation logic tools like GRASShopper [Piskac et al. 2013, 2014] and the Iris higher-order separation logic framework [Jung et al. 2016, 2017, 2015; Krebbers et al. 2017]. The former enables proof automation via SMT solvers and the latter can be used to mechanically check proofs using Coq [Coq Development Team 2017]. This showcases the flexibility of the new framework, as it allows the verifier to choose either highly-trusted but labor-intensive tools or tools that provide more automation but have a larger trusted code base, depending on the needs at hand.

We demonstrate our methodology by formalizing three template algorithms for concurrent search structures based on the link, the give-up, and the lock-coupling technique of synchronization (Fig. 1). For these, we derive concrete implementations based on B-trees, hash tables, and sorted linked lists, resulting in five different data structure implementations. §4 discusses the proof of the link template in detail. A summary of our verification effort is provided in §5. We perform the template proofs in Iris/Coq and verify the implementations in GRASShopper, in order to bring the proofs of highly complicated implementations such as B-link trees within reach.

We note that there is no formal connection between the proofs done in Coq and GRASShopper. If one desires end-to-end certified proofs, one can parametrize Iris by the programming language used in GRASShopper and use GRASShopper as an oracle for implementation proofs, or even perform both template and implementation proofs in Iris/Coq (albeit with substantial manual effort). The only other trusted component is the meta theory of our flow framework. These proofs are simple facts about graphs and graph properties that need only be proven once, apply to every data structure proof in our evaluation and beyond, and are proved in detail in §A. By contrast, all the proofs pertaining to our individual case studies are performed and verified by either Coq or GRASShopper.

All the algorithms we have considered use fine-grained node-level locking. 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\(^1\). On the other hand, our methodology does not require locking, and can be extended to prove lock-free algorithms such as the Bw-tree [Levandoski and Sengupta 2013].

The proofs we obtain are both more modular and simpler than existing proofs of such concurrent data structures. In fact, we are the first to obtain a mechanically verified proof of concurrent B-link trees. Unlike the proof of da Rocha Pinto et al. [2011], 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 [Malecha et al. 2010]. However, this work only considers a sequential implementation of B-trees and the proof is considerably more complex than ours (encompassing more than 5000 lines of proof for roughly 500 lines of code).

The contributions of this paper can be summarized as follows:

• We propose a methodology for verifying concurrent search structure templates that enables proofs to be compositional in terms of program structure and state, and exploit both thread and algorithmic modularity.

• We present an improved and simplified flow framework for reasoning compositionally about the complex invariants needed for the template and implementation proofs. The framework is designed so that it can be used with off-the-shelf separation logic verification tools.

• 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.

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 \( B \leq l + 1 < 2B \) for some constant \( B \). At internal nodes, an operation on \( k \) takes the branch \( y_i \) if \( k_{i-1} \leq k < k_i \).

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

\(^1\)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.

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 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 \( y_3 \) 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 \( n' \) 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 \( n' \).
Fig. 3. The link template algorithm, which can be instantiated to the B-link tree algorithm by providing implementations of helper functions \( \text{findNext} \) and \( \text{decisiveOp} \). \( \text{findNext} \) \( n \ k \) returns \( \text{Some } n' \) if \( k \in \text{es}(n, n') \) and \( \text{None} \) if there exists no such \( n' \). \( \text{decisiveOp} \ n \ k \) performs the operation \( \omega \) (either search, insert, or delete) on \( k \) at node \( n \).

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 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 \emph{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 order to make edgeset a local quantity, we say \( k \in \text{es}(n, n') \) 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 \( i \)th array entry is \( \{ k \mid \text{hash}(k) = i \} \). The edgeset from an array entry to the bucket node is the set of all keys \( \text{KS} \), 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 [Shasha and Goodman 1988] 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 \( k \in \text{es}(n, n') \). For the B-link tree, \( \text{findNext} \) 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 \( \text{searchStrOp} \) can be used to build implementations of all three search structure 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 7 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 \( k \in \text{es}(n, n') \), 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 8). If the operation succeeds, then \( \text{decisiveOp} \) returns \( \text{Some } \text{res} \) and the algorithm unlocks \( n \) and returns \( \text{res} \). 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.
2.4 A Proof Strategy for Template Search Structures

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 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 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.

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 $\text{inset}(n)$, 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 $n$, $\text{outset}(n)$, 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 $n$, $\text{keyset}(n)$. 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:

1. (GS1) The contents of every node are a subset of the keyset of that node.
2. (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 [1988] shows that the operation modifies the abstract state correctly.

How does the link template ensure that $k \in \text{keyset}(n)$ 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 $k \in \text{inset}(r)$, and traverse an edge $(n, n')$ only when $k \in \text{es}(n, n')$, maintaining the invariant that $k \in \text{inset}(n)$. When there does not exist an outgoing edge with $k$ in the edgeset, we know by definition that $k \in \text{keyset}(n)$.

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 inset($n$) 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.

(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 [1988] shows that the operation modifies the abstract state correctly.

How does the link template ensure that $k \in \text{keyset}(n)$ 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 $k \in \text{inset}(r)$, and traverse an edge $(n, n')$ only when $k \in \text{es}(n, n')$, maintaining the invariant that $k \in \text{inset}(n)$. When there does not exist an outgoing edge with $k$ in the edgeset, we know by definition that $k \in \text{keyset}(n)$.

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 inset($n$) 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 \textit{inreach} of a node \( n \) as

\[
inreach(n) := \text{inset}(n) \cup \bigcup_{n'} \text{es}(n, n') \cap \text{inreach}(n').
\]

Intuitively, \( \text{inreach}(n) \) 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 \( k \in \text{inset}(n') \). For example, in Fig. 2 the \( \text{inreach} \) of \( y_1 \) is \([4, \infty)\) even though its \( \text{inset} \) 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 \( k \in \text{inreach}(n) \). This is true at the root, because \( \text{inreach}(r) = \text{inset}(r) = KS \), and it is preserved during the traversal even with concurrent splits. When \( \text{findNext} \) returns \( \text{None} \), the definition of \( \text{inreach} \) implies that \( k \in \text{inreach}(n) \setminus \text{outset}(n) \subseteq \text{keyset}(n) \), 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 \( \text{inreach} \) 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 \texttt{decisiveOp}, then the proof of an implementation can reason only about the node that the helper function modifies. In the next section, we show how to reason about the correctness conditions for template algorithms that rely on global quantities using local reasoning.

3 \hspace{1em} A FLOW INTERFACE RESOURCE ALGEBRA

The flow framework [Krishna et al. 2018] is a separation logic based approach for specifying and reasoning about unbounded data structures. The framework represents the heap as an abstract labeled graph. Data structure invariants are expressed as local conditions satisfied by each node in the graph. These conditions are allowed to depend on the \textit{flow} of the node, a quantity computed inductively over the entire graph. Unbounded regions of the heap are then abstracted using \textit{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. Flows interfaces are powerful enough to express the edgeset framework’s good conditions and prove that programs maintain them, despite their dependence on global quantities such as the inset.

The flow framework 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, using the original flow framework in existing separation logic tools is challenging. This section describes these challenges and presents a revised version of the flow framework tailored to meet the demands of such tools.

3.1 Motivation for a New Framework

There are two main obstacles in using the original flow framework in existing separation logics and automated tools: (1) flow graphs can compose to more than one flow graph that has the same flow, and (2) reasoning about flow interfaces involves calculating solutions of a fixpoint equation that does not converge in finitely many steps.
The first problem stems from the fact that the original framework defined flows over a flow domain having a semiring structure. Because of this, two flow graphs can compose to more than one flow graph that has the same flow. However, most separation logics, including Iris, require user-defined resources to have an algebraic structure (for example, Iris uses resource algebras, §3.2) that requires that composition be a function. To solve this issue, the original framework defines an equivalence on inflows based on the resulting flow, and uses inflow equivalence classes to define composition as a function.

However, reasoning about inflow equivalence classes (a set of functions) is both tedious and difficult to automate. For example, it introduces an additional quantifier alternation when proving the premise of the replacement theorem for a modified heap region. Krishna et al. [2018] suggest switching to a relational semantics for flow interface composition in order to avoid equivalence classes. However, some critical proof rules for manipulating flow interfaces no longer hold true in this model. Moreover, Iris assumes a functional semantics for the composition operator and with a relational semantics we also lose the ability to do simple equational reasoning which automated tools such as GRASShopper rely on.

The second problem is that when proving that modifications preserve the interfaces of the region they modify, one must compute or reason about the flow map (a function summarizing how flow is routed through a region) of the modified region. Calculating the flow map requires reasoning about the solution of a fixpoint equation. This means that automatic reasoning, say by unrolling the definition of the flow map, may not be feasible in all cases.

The flow framework presented in this section eliminates the above obstacles by restricting flow domains to rings rather than semi-rings and by considering only effectively acyclic flow graphs (see §3.3). We found that, in practice, these restrictions make only small compromises on expressivity while greatly increasing the potential for automation. Our modifications yield a resource algebra, which means we can use flow interfaces as ghost state in Iris, and allow us to automatically prove that modified regions preserve their interfaces in tools such as GRASShopper.

### 3.2 Resource Algebras and Ghost State

We first describe resource algebras (RAs), the structure underlying user-defined resources in Iris². To use flow-based reasoning in Iris, we need a notion of flow interfaces that form an RA. RAs are a generalization of partial commutative monoids or separation algebras, the standard algebraic structure underlying resources in most separation logics. Thus, showing that flow interfaces form an RA is useful even if one wants to use flows in a separation logic other than Iris, for example FCSL [Sergey et al. 2015].

The definition of a resource algebra is given in Fig. 4, where Prop is the type of propositions of the meta-logic (e.g. Coq). Readers familiar with separation logic will notice that the composition operator is not partial as is the case in standard separation algebras. Instead, RAs use the validity predicate \( \mathcal{V} \) to identify valid elements of the domain; cases where composition used to be undefined can be encoded by sending them to an invalid element. Another difference is that RAs do not have a single unit element, and instead the partial function \( \lvert - \rvert \) assigns to an element \( a \) a core \( \lvert a \rvert \) (which can be thought of as \( a \)'s own unit). These features allow Iris to express higher-order state and build more expressive encodings.

²Iris actually uses cameras as the structure underlying resources, but as we do not use higher-order resources (i.e. state which can embed propositions) in this paper we restrict our attention to RAs, a stronger, but simpler, structure.
A resource algebra is a tuple \((M, \overline{V} : M \to \text{Prop}, \lnot: M \to M^2, (\cdot): M \times M \to M)\) satisfying:

\[
\forall a, b, c. \ (a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \text{(ra-assoc)}
\]

\[
\forall a, b. \ a \cdot b = b \cdot a \quad \text{(ra-comm)}
\]

\[
\forall a. \ |a| \in M \Rightarrow |a| \cdot a = a
\]

\[
\forall a. \ |a| \in M \Rightarrow |a| = |a|
\]

\[
\forall a, b. \ |a| \in M \land |a| \leq b \Rightarrow |b| \in M \land |a| \leq |b|
\]

\[
\forall a, b. \ \overline{V}(a \cdot b) = \overline{V}(a) \quad \text{(ra-valid-op)}
\]

where \(M^2 := M \cup \{|\perp\}\)

\[
a^\perp \cdot \perp := \perp \cdot a^\perp := a^\perp
\]

\[
a \ll b := \exists c \in M. \ b = a \cdot c
\]

Fig. 4. The definition of a resource algebra (RA).

For example, heaplets, the standard separation algebra used to encode the heap, can be viewed as an RA:

\[
M := \{h: \text{Loc} \to \text{Val}\} \cup \{|\perp\}\quad \overline{V}(a) := a \neq \perp \quad |a| := \epsilon
\]

\[
a \cdot b := \begin{cases} a \cup b & a \neq \perp \neq b \land \text{dom}(a) \cap \text{dom}(b) = \emptyset \\ \perp & \text{otherwise} \end{cases}
\]

where \text{Loc} is a set of memory addresses or locations, \text{Val} is a set of values that includes \text{Loc}, and \(\epsilon\) is the empty heaplet. As in standard separation logic, the composition operator allows composing only those heaplets whose domains are on disjoint sets of locations. This permits splitting the heap and sharing it, for instance among threads.

However, Iris restricts the kind of updates one can make on an RA to only those that preserve the invariant that the composition of all RA elements is valid. Such updates are called frame-preserving updates, and one can do a frame-preserving update from \(a \in M\) to \(B \subseteq M\), written \(a \leadsto b\), if

\[
\forall a_i^\perp \in M^2. \ \overline{V}(a \cdot a_i^\perp) \Rightarrow \overline{V}(b \cdot a_i^\perp),
\]

where \(M^2\) is the option type of \(M\) defined in Fig. 4. Intuitively, this condition says that every frame \(a_i^\perp\) that is compatible with \(a\) should also be compatible with \(b\). Thus, changing one’s fragment of the resource from \(a\) to some \(b\) will not violate the assumptions made by anyone else.

In our heaplet example, we can easily check that we have the following frame-preserving update, which, as expected, allows us to change the value of a location in the domain:

\[
\forall \ell \in \text{Loc}, \ u \in \text{Val}. \ \ell \in \text{dom}(h) \Rightarrow h \leadsto h[\ell \mapsto u].
\]

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.

### 3.3 Flows

A flow is an inductively-defined quantity computed over a labelled graph. The first step in using flows is to identify a flow domain: 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
Verifying Concurrent Search Structure Templates 1:11

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 3.1 (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 3.2. 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.

Example 3.3. Let \((D_1, +_1, \cdot_1, \sqsubseteq_1, 0_1, 1_1)\) and \((D_2, +_2, \cdot_2, \sqsubseteq_2, 0_2, 1_2)\) be flow domains. The product \((D_1 \times D_2, +_1+_2, \cdot_1 \cdot_2, \sqsubseteq_1 \sqsubseteq_2, (0_1, 0_2), (1_1, 1_2))\), where \( \sqsubseteq, +, \cdot \) operate on each component respectively, is a flow domain. Moreover, the flow on the product domain is the product of the flows on each component. This construction is very useful when reasoning about overlaid structures (see §4.1).

We abstract heaps using directed partial graphs. The graphs are partial because they describe abstractions of heaplets 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 graphs 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 3.4 (Graphs). Given a (potentially infinite) set of nodes \( \text{Node} \), a (partial) graph \( G = (N, N^0, \lambda, \varepsilon) \) consists of a finite set of nodes \( N \subseteq \text{Node} \), a finite set of sink nodes \( N^0 \subseteq \text{Node} \) disjoint from \( N \), a node labeling function \( \lambda : N \to A \), and an edge function \( \varepsilon : N \times (N \cup N^0) \to D^+ \).

Note that the edge function is total on \( N \times (N \cup N^0) \). The absence of an edge between two nodes \( n, n' \) is indicated by \( \varepsilon(n, n') = 0 \). We let \( \text{dom}(G) = N \) and sometimes identify \( G \) and \( \text{dom}(G) \) 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 \( \text{flow}(in, G) : N \to D^+ \) that is calculated in terms of a certain fixpoint over \( G \)’s edge function starting from a given inflow \( in : N \to D^+ \) going into \( G \). We refer to this fixpoint as the capacity \( \text{cap}(G) : N \times (N \cup N^0) \to D^+ \) of \( G \), which is defined as the least fixpoint of the following equation:

\[
\text{cap}(G)(n, n') = \text{init}(n, n') + \sum_{n'' \in G} \varepsilon(n, n'') \cdot \text{cap}(G)(n'', n')
\]

where \( \text{init}(n, n') := (n = n' \land 1 : 0) \).

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 \( \text{cap}(G)(n, n') \) will not converge.

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

\[
\text{flow}(in, G)(n) := \sum_{n' \in G} in(n') \cdot \text{cap}(G)(n', n).
\]

We call a pair \((in, G)\) effectively acyclic if for all sequences of nodes \( n_1, \ldots, n_k \in G \) 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 \( \text{flow}(in, G) \) is a total function on \( \text{dom}(G) \).
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 [Harris 2001], for although they have cycles, the edge labels can be chosen to set all cycles to have zero product.

Example 3.5. For our encoding of the edgeset framework, we cannot use sets of keys as the flow domain because sets do not form a ring. Instead, we encode key sets using the key counting flow domain $\mathcal{H} := (\mathcal{K} \times \mathbb{Z}^+) \cup \{\cdot\} \subseteq \mathbb{Z}^+$, where the operations and order are lifted pointwise from $\mathbb{Z}$ and $\mathbf{0} := \lambda k. 0$ and $1 := \lambda k. 1$. Here we label each edge $(n, n')$ in graph $G$ by the function $\lambda k. (k \in \text{es}(n, n') ? 1 : 0)$, which encodes the edgeset $\text{es}(n, n')$ of the edge. For $\text{in} = \lambda n k. (n = r \uplus 0 : 1)$, which demands that the searches for all keys $k$ in the global graph start at the root $r$, the flow $kc = \text{flow}(\text{in}, G)$ 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 $e(n, n')$). 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 $n_1, n_2$.

3.4 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. the restriction of the flow to $G_2$, $\text{flow}(\text{in}, G)|_{G_2} = \text{flow}(\text{in}, G')|_{G_2}$ 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 $(\text{in}, G)$ of a graph and its associated local inflow. Formally, let

$$\text{FG} := H \in \{(\text{in}, G) \mid \text{in}: N \to D^+ \land (\text{in}, G) \text{ is effectively acyclic}\} \ | \ H_1$$

and let $H_e = (\text{in}_e, G_e)$ be the empty flow graph consisting of the empty inflow $\text{in}_e$ and the empty graph $G_e$. Then two flow graph $H_1$ and $H_2$ compose to a flow graph $H_1 \bullet H_2 = (\text{in}, G)$ if $H_1 = (\text{in}_1, G_1)$, $H_2 = (\text{in}_2, G_2)$, $G = G_1 \uplus G_2$, and $\text{flow}(\text{in}_1, G_1) = \text{flow}(\text{in}_2, G_2)$ for $i \in \{1, 2\}$. In all other cases we define $H_1 \bullet H_2 = H_1$. This composition is uniquely determined by the ring properties of $D$.

Lemma 3.6. Composition of flow graphs $\bullet$ is a commutative monoid with identity $H_e$.

3.5 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:

$$\text{flm}((\text{in}, G)) := \text{cap}(G)|_{N \times N^o}.$$ 

A graph is then abstracted by its flow interface:
**Definition 3.7 (Flow Interface).** Given a flow graph $H \in \text{FG}$, its flow interface is a tuple consisting of its inflow, the join of all its node labels, and its flow map:

$$\text{int}(H) := (\text{in}, \bigsqcup_{n \in G} \lambda(n), \text{flm}(\text{in}, G))$$

where $H = (\text{in}, G)$ and $G = (N, N^o, \lambda, \varepsilon)$.

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

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

**Lemma 3.8.** $\text{int}(H_1) = \text{int}(H_1') = I_1 \land \text{int}(H_2) = \text{int}(H_2') = I_2 \Rightarrow \text{int}(H_1 \bullet H_2) = \text{int}(H_1' \bullet H_2')$.

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, $I_n = (\text{in}_n, a_n, f_n)$ where $\text{in}_n$ 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 $\nu$, we can filter the set of flow graphs to those that satisfy $\nu$:

$$\text{FG}_\nu := \{H \in \text{FG} \mid \forall n \in H. \nu(n, (\text{in}_n, \lambda(n), \varepsilon(n)))\} | H \forall n$$

where $\text{in}_n := \{n \mapsto \text{flow}(H)(n)\}$

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

The following lemma states that the condition $\nu$ 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 3.9.** For all good conditions $\nu$, if $H_1, H_2 \in \text{FG}_\nu$ then $H_1 \bullet H_2 \in \text{FG}_\nu$.

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

$$\text{FI}_\nu := I \in \{\text{int}(H) \mid H \in \text{FG}_\nu\} | I \quad \bar{\nu}(a) := a \neq I \quad |I| := I_e := \text{int}(H_e) \quad |I| := I$$

$$I_1 \oplus I_2 := \text{int}(H_1 \bullet H_2) \text{ for any } H_1, H_2 \text{ s.t. } \text{int}(H_1) = I_1 \land \text{int}(H_2) = I_2.$$ 

**Theorem 3.10.** For every good condition $\nu$, the flow interface algebra $(\text{FI}_\nu, \bar{\nu}, |\cdot|, \oplus)$ is a resource algebra.

When modifying a flow graph $H$ to another flow graph $H'$, requiring that $H'$ satisfies the same interface $\text{int}(H)$ is too strong a condition for verifying many data structure algorithms. Instead, we want to allow $\text{int}(H')$ to differ from $\text{int}(H)$ 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 allocating new nodes, as long as the new nodes are fresh and have no outgoing edges.

Formally, we say an interface $(\text{in}, a, f)$ is contextually extended by $(\text{in}', a', f')$, written $(\text{in}, a, f) \preceq (\text{in}', a', f')$, if and only if

- $\text{dom}(\text{in}) \subseteq \text{dom}(\text{in}')$,
- $\forall n \in \text{dom}(\text{in}). \text{in}(n) = \text{in}'(n)$,
- $\forall n \in \text{dom}(\text{in}), n'. 0 \subset \text{in}(n) \Rightarrow f(n, n') = f'(n, n')$, and
- $\forall n \in \text{dom}(\text{in}') \setminus \text{dom}(\text{in}), n'. f'(n, n') = 0$.

The following theorem states that contextual extension preserves composability and is itself preserved under interface composition.
We define the specification program $e$.

This section shows how to tie together the edgeset framework and flow interfaces in Iris in order to prove that a delete search, insert, or delete operation on query key $k$ is correct, we additionally need to show that $k$ is not present in the contents of any other node in the structure.

\[ \Psi_\omega(k, C, C', \text{res}) := \begin{cases} C' = C \land (\text{res} \iff k \in C) & \omega = \text{search} \\ C' = C \cup \{k\} \land (\text{res} \iff k \notin C) & \omega = \text{insert} \\ C' = C \setminus \{k\} \land (\text{res} \iff k \in C) & \omega = \text{delete} \end{cases} \]

Fig. 5. Abstract specification of search structure operations.

Theorem 3.11 (Replacement). If $I = I_1 \oplus I_2$, and $I_1 \preceq I'_1$ are all valid interfaces such that $I'_1 \cap I_2 = \emptyset$, and $\forall n_1 \in I'_1 \setminus I_1, n_2 \in I_2, I'_2(n_2, n_1) = 0$, then there exists a valid $I' = I'_1 \oplus I_2$ such that $I \preceq I'$.

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

4 VERIFYING SEARCH STRUCTURE TEMPLATES

This section shows how to tie together the edgeset framework and flow interfaces in Iris in order to verify template algorithms for concurrent search structures. We 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. A formal introduction to Iris and the underlying programming language semantics is, unfortunately, beyond the scope of this paper. We provide intuition for the key logical constructs and reasoning steps as and when they are used; for a more detailed introduction to Iris see [Jung et al. 2017].

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 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 $e_2$.

To prove contextual refinement, we use ReLoC [Frumin et al. 2018], an extension of Iris with first-class support for reasoning about refinement. However, to simplify the presentation, we show only the Hoare-style proof of the invariant needed for refinement in this section and indicate at the appropriate points what extra proof obligations are needed for the full refinement proof. All free variables in the intermediate assertions are implicitly existentially quantified.

Let us abstractly represent the state of a search structure as the set of keys $C$ that it contains. We define the specification program $\text{searchStrSpec}$ for search structure operation $\omega$ (either search, insert, or delete) on query key $k$ as an atomic step that modifies the set $C$ to a new set $C'$, and returns the value res, such that the predicate $\Psi_\omega(k, C, C', \text{res})$ defined in Fig. 5 holds. Note that as our specification (a mathematical set ADT) is atomic, we can infer that our implementation is linearizable [Filipovic et al. 2009].

The high-level idea behind the refinement proof is to relate the state of the implementation to the state of the specification so that we can show that the implementation modifies the state in a manner consistent with the specification. We will do this in Iris using an invariant $\text{Inv}$ which is a formula in Iris’ logic that we formally define in §4.3 but describe intuitively here. Since efficient concurrent implementations usually distribute the state over a set of nodes, a natural first step is to associate each node with the set of keys it contains and use the invariant $\text{Inv}$ to express that the union of contents of all nodes is equal to the abstract state $C$. However, as mentioned in §2.4, this is not enough if one wants to do local reasoning. For instance, if we want to prove that a delete operation on $k$ that removed $k$ from node $n$ is correct, we additionally need to show that $k$ is not present in the contents of any other node in the structure.

The essential property we require is that the contents of distinct nodes be disjoint, but disjointness is not a local condition either.

The edgeset framework comes to our rescue here: if the implementation state satisfies the good state conditions of §2.4 and $k$ is in the keyset of $n$, then we know that $k$ cannot be present in any other node. Thus, $\text{Inv}$ will also enforce that the state is a good state. As we have seen, to reason about the good state conditions locally, we encode them using flows. We will use ghost state (see §4.2) to keep track of the flow interface of each node, and use the node-local constraint $\nu$ to enforce that each node satisfies the local version of the good state conditions from Example 3.5. Our invariant $\text{Inv}$ will tie the flow interface of each node to the actual heap representation of the node. The flow interfaces also keep track of each node’s contents using the node labels, and $\text{Inv}$ will further enforce that the node label of the global interface should be equal to the abstract contents $C$.

Apart from showing that all threads maintain the invariant, we must additionally show that any assumptions made by a thread is not violated by the actions of other threads. For example, as discussed in §2.4, $\text{traverse}$ relies on the fact that $k \in \text{inreach}(n)$ when it is called in order to guarantee that $k \in \text{keyset}(n)$ when it returns. So we must ensure that no other operation modifies the state in a way that violates $k \in \text{inreach}(n)$, which we can do by proving that no operation decreases the inreach of any node. One can think of this as being the protocol obeyed by each thread in the link technique.

The rest of this section explains how to implement this high-level proof structure in Iris. First, §4.1 explains how we use flow interfaces to enforce the edgeset framework’s good state conditions and lift a proof that an operation correctly updated a node to a proof that the operation correctly updated the entire data structure. §4.2 then describes the resource algebras that we use to encode both flow interfaces and other ghost information needed for our proof, as well as to enforce the protocol by which the shared state is updated. Finally, in §4.3 we define the invariant $\text{Inv}$ in Iris and prove that the link template algorithm maintains $\text{Inv}$. We also describe how $\text{Inv}$ is strong enough to extend this proof to show refinement.

### 4.1 Encoding the Edgeset Framework using Flows

Our first task is to provide an encoding of the edgeset framework using 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 3.5, we use the key counting flow domain to encode the inset as a flow of each node. It is hard to define the inreach directly as a flow [Krishna et al. 2018], so we encode an under-approximation of inreach that is sufficient for correctness. 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 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 key count domains (see Example 3.3) as the flow domain, where the first component calculates the inset as in Example 3.5. 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 $\text{linkset}(n)$. 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 require only that the linkset approximate the inreach in such a way that it has the following properties: First, if $k \in \text{linkset}(n) \setminus \text{inset}(n)$ then for every edge $(n, n')$, $k$ is in the edge label of the inset component (i.e. the edgeset) of $(n, n')$ if and only if $k$ is in the linkset component of the edge label of $(n, n')$. This is used to prove that if $k \in \text{inset}(n) \cup \text{linkset}(n)$ and $\text{findNext n k}$ returns Some $n'$, then $k \in \text{inset}(n') \cup \text{linkset}(n')$ (needed by the recursive call to $\text{traverse}$).
Second, if \( k \in \text{lnkset}(n) \setminus \text{outset}(n) \), then \( k \in \text{inset}(n) \Rightarrow k \in \text{keyset}(n) \), which implies that when \( \text{findNext} \) fails, we have found the right node \( n \) (\( k \) is in \( n \) or nowhere in the structure). These two properties mean that instead of the inreach it is sufficient to work with the inreach-approximation \( \text{inset}(n) \cup \text{lnkset}(n) \), which for simplicity we shall call inreach in the following. We enforce these properties in \( v \), the local good condition on nodes.

We use a product of two node domains for the node labels of flow interfaces to keep track of two kinds of information. First, as mentioned in the high-level proof idea, we need the set of contents so that we can relate the state of the implementation to the state of the specification. Second, we need to keep track of the inreach of each node so that we can enforce the protocol that the inreach only increases. We encode this using a node domain consisting of partial functions from nodes to sets of keys: \((\text{Node} \rightarrow 2^\text{KS} \cup \{\top\}, \sqsubseteq)\), where

\[
\rho_1 \sqsubseteq \rho_2 :\iff \rho_2 = \top \lor \rho_1 \neq \top \land \rho_2 = \rho_2[\text{dom}(\rho_1)].
\]

Note that this means that the join of two functions is \( \top \) if they disagree on the inreach of any node. This gives us a property that we will use in our proof:

\[
I = I_n \oplus \_ \land \text{dom}(I_n) = \{n\} \land I_{ir}^a \neq \top \Rightarrow I_{ir}^a(n) = I_{ir}^a(n),
\]

where \( a_{ir} \) denotes the inreach component of the node domain element \( a \).

Before describing \( v \), we introduce some shorthand notation for clarity (these overload the symbols used when describing the edgset framework because they express the same quantities):

\[
\text{inset}(I, n) := \{ k \mid I_{in}^a(n)_{is}(k) \geq 1 \} \quad \text{lnkset}(I, n) := \{ k \mid I_{ln}^a(n)_{is}(k) \geq 1 \}
\]

\[
\text{outset}(I) := \{ k \mid \exists n'. I_f(n, n')_{is}(k) \geq 1 \} \quad \text{es}(I_n, n, n') := \{ k \mid I_f(n, n')_{is}(k) \geq 1 \}
\]

\[
C(I) := \text{let } (C, \_ ) = I^a \text{ in } C \quad \text{inreach}(I, n) := \text{let } (_, \rho ) = I^a \text{ in } \rho(n)
\]

where for an interface \( I = (in, a, f) \), \( I_{in}^a, I_{ln}^a \), and \( I_f \) denote \( in, a, \) and \( f \) respectively, and \( d_{is} \) 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:

\[
v(n, I) := C(I) \subseteq \text{inset}(I, n) \setminus \text{outset}(I)
\]

\[
\land (\forall n', n''. n' = n'' \lor \text{es}(I, n, n') \land \text{es}(I, n, n'') = \emptyset)
\]

\[
\land (\forall k, n'. k \in \text{inreach}(I, n) \setminus \text{inset}(I, n) \Rightarrow I_f(n, n')_{is}(k) = I_f(n, n')_{ls}(k))
\]

\[
\land \text{lnkset}(I, n) \subseteq \text{inset}(I, n) \cup \text{outset}(I, n)
\]

\[
\land I^a = (_) \{ n \mapsto \text{inset}(I, n) \cup \text{lnkset}(I, n) \}.
\]

Here, conditions (2) and (3) encode the good state conditions (GS1) and (GS2). (4) and (5) are the two constraints on the linkset that we described earlier. Finally, (6) uses the node label to keep track of the inreach \( \text{inset}(n) \cup \text{lnkset}(n) \).

We also require the following constraints on the global interface:

\[
\varphi(I) := (\forall n, k. I_{in}^a(n)_{is}(k) = (n = r \ ? 1 : 0)) \land I_{ir}^a \neq \top \land I_f = \epsilon
\]

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 node label, \( I_{ir}^a \), to be unequal to \( \top \), and finally we require that the global interface is closed (i.e. has no outgoing edges).

The good condition \( v \) and the global interface constraint \( \varphi \) together result in a flow that computes the inset and the linkset of each node. They also enforce the good state conditions of the edgset framework. Finally, we can use these quantities and constraints to formulate a version of the Keyset
Theorem from [Shasha and Goodman 1988], which reduces the problem of proving the search structure specification $\Psi_\omega$ for the global interface to proving it for the interface of the single node modified.

**Lemma 4.1.** Given $I, I', I_n, I'_n, I_2 \in \mathcal{F}_V$, $k \in \mathcal{K}_S$, $n \in \text{Node}$, and res such that

- $\varphi(I)$ holds and $\text{dom}(I_n) = \{n\}$,
- $k \in \text{inset}(I_n, n)$ and $k \notin \text{outset}(I_n)$, and
- $I = I_n \oplus I_2$ and $I' = I'_n \oplus I_2$,

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

### 4.2 Resource Algebras and Ghost State

Iris models both the knowledge of threads about the shared state (e.g. $k \in \text{keyset}(n)$) and protocols for modifying the shared state (e.g. inreach cannot decrease) 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 run-time behavior. Ghost state can be split and combined according to the rules of the underlying RA: $\overline{a} \overline{y} \circ \overline{b} \overline{y} \iff \overline{a} \overline{b} \overline{y}$. Furthermore, Iris maintains the invariant that the composition of all the pieces of ghost state at a particular location is valid (in terms of the $\overline{V}$ predicate from Fig. 4).

For instance, consider the authoritative RA that we will use to keep track of the inreach of each node. Given an RA $M$, the authoritative RA $\text{AUTH}(M)$ (see [Jung et al. 2017] 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 $b \preceq a$. 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 $b \preceq a$ 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, at locations $\gamma_i(n)$ for each node $n$, to encode the inreach of each node. From the definition, one can show that this RA satisfies the following properties:

\[
\begin{align*}
\text{AUTH-SET-UPD} & \quad X \subseteq Y \\
\bullet X \rightsquigarrow \bullet Y \\
\text{AUTH-SET-VALID} & \quad \overline{V} (\bullet X \cdot \circ Y) \\
\text{AUTH-SET-VALID} & \quad \overline{V} (\bullet X \cdot \circ Y) \\
\end{align*}
\]

We store the inreach of a node $n$ with interface $I_n$ in the shared state as $\bullet \text{inreach}(I_n, n)$, $\circ \text{inreach}(I_n, n)$, and $\circ \text{inreach}(I_n, n)$ to their local state. This allows them to make assertions such as $\exists X, \overline{a} \circ X \overline{y} (i(n)) \circ k \in X$ for some key $k$, which in conjunction with $\text{AUTH-SET-VALID}$ encodes the knowledge that $k \in \text{inreach}(I_n, n)$. This knowledge is stable under interference by other threads, for the only frame-preserving update permitted by this RA is $\text{AUTH-SET-UPD}$, which allows threads to increase the inreach of a node.

We also use an authoritative RA of flow interfaces $\text{AUTH}(\mathcal{F}_V)$ to keep track of the flow interface ghost state of our algorithms. Using Theorem 3.11, we can show that this RA permits the following non-deterministic frame-preserving update (which is a generalization of the standard...
\[
\{ h(n, I_n) \}
\]
\[
\text{findFirst } n \ k \ \\
\{ v. h(n, I_n) * (v = \text{None} * k \not\in \text{outset}(I_n)) \ \\
\quad \vee v = \text{Some}(n') * k \in \text{es}(I_n, n, n') \}
\]
\[
\{ h(n, I_n) * k \in \text{inset}(I_n, n) * k \not\in \text{outset}(I_n) \}
\]
\[
\text{ decideOp } \omega \ n \ k \ \\
\{ v. v = \text{None} * h(n, I_n) \ \\
\quad \vee v = \text{Some}(v') * h(n, I_n) * \Psi_\omega(k, I_n, I'_n, v') \ \\
\quad * I_n \leq I'_n * \text{inreach}(I_n, n) = \text{inreach}(I'_n, n) \}
\]

Fig. 6. Specifications of helper functions that are defined by implementations.

frame-preserving update presented in §3.2, for details see [Jung et al. 2017]):

\[
\text{AUTH-FI-UPD} \quad I_1 \preceq I'_1 \ \\
(\bullet I, o I_1) \leadsto \{ (\bullet I', o I'_1) \mid I \preceq I' \land \exists I_2. I = I_1 \oplus I_2 \land I' = I'_1 \oplus I_2 \}
\]

Finally, given any set \( S \), it is easy to see that we have an RA \((2^S, \text{True}, \lambda X . \emptyset, \cup)\), which permits a frame-preserving update \( X \leadsto Y \) 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.

### 4.3 Proof of the Link Template

Fig. 7 presents a proof outline of the link template algorithm, where the intermediate assertions in braces show the context of the proof (the premises that are currently available). Our proof assumes that the implementation-specific helper functions satisfy the specifications shown in Fig. 6. We also use a standard lock module, similar to the one in [Frumin et al. 2018], adapted to a setting that the implementation-specific helper functions satisfy the specifications shown in Fig. 6. We assume the specification of the lock and unlock methods in Fig. 7, but note that our Iris proofs prove this specification.

The Hoare-style specification for searchStrOp is simply \( \{ \text{Inv} \} \text{searchStrOp } \omega \ r \ k \{ \text{Inv} \} \). However, the refinement proof will also require us to show that if we execute the specification program searchStrSpec at the linearization point, then searchStrSpec returns the same value as will be returned by searchStrOp. In order to do this, we construct an Iris invariant that relates the implementation state and the specification state and is sufficient to prove correctness:

\[
\text{Inv} := \exists I. \left[ \begin{array}{c} I \upharpoonright \ Omega_f \ast \varphi(I) \ast \bullet \text{dom}(I) \upharpoonright f \ast \bigotimes_{n \in I} \bullet \text{inreach}(I, n) \upharpoonright \Psi(n) \ast \bigodot_{c} \Pi(I) \upharpoonright c \end{array} \right] \ast \ast \ni \end{array} \right] \exists b. \ (\ell(n) \mapsto b \ast (b = ? \text{ True : } \exists I_n. \text{Node}(n, I_n, I_n))
\]

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 Auth(1_f) RA at location \( \gamma_f \) to keep track of the flow graph abstraction. The invariant always owns the authoritative version \( \left[ \begin{array}{c} I \upharpoonright \ Omega_f \ast \varphi(I) \ast \bullet \text{dom}(I) \upharpoonright f \end{array} \right] \), which is the interface of the global graph and that satisfies \( \varphi(I) \). We assume that every node \( n \in I \) has a lock bit at 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 \( \left[ \begin{array}{c} I \upharpoonright \ Omega_f \ast \varphi(I) \ast \bullet \text{dom}(I) \upharpoonright f \end{array} \right] \) and \( n \)'s heap representation \( h(n, I_n) \) (denoted Node\((n, I_n, I_n)\)).

We use an authoritative RA of sets of keys, at locations \( \Omega(n) \) 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 even when it is unlocked, as described in §4.2. We also use an authoritative RA of sets of nodes at location \( \gamma_f \) to encode the footprint of the global graph. The invariant owns the authoritative version...
Verifying Concurrent Search Structure Templates

(*) Let \( \text{inFP}(n) \coloneqq \exists N. \left[ \begin{array}{c} \text{lockNode} n \rightarrow \text{unlockNode n} \\ \text{unlockNode n} \rightarrow \text{lockNode n} \end{array} \right] \) \( \Rightarrow n \in N \), \( \text{inInr}(n, k) \coloneqq \exists R. \left[ \begin{array}{c} \text{unlockNode n} \\ \text{lockNode n} \rightarrow \text{unlockNode n} \rightarrow \text{lockNode n} \rightarrow \text{unlockNode n} \end{array} \right] \) \( \Rightarrow k \in R \). (*)

\[
\begin{aligned}
\text{lockNode n;} & \\
\text{unlockNode n;}
\end{aligned}
\]

\[
\begin{aligned}
\text{let rec} & \quad \text{traverse n k =} \\
\text{lockNode n;} & \\
\text{unlockNode n;}
\end{aligned}
\]

\[
\begin{aligned}
\text{match findNext n k with} & \\
| \text{None} \rightarrow & \\
| \text{Some} n' \rightarrow & \\
\text{traverse n' k}
\end{aligned}
\]

\[
\begin{aligned}
\text{let rec} & \quad \text{searchStrOp } \omega r k = \\
\text{let n =} & \\
\text{match decisiveOp } \omega n k with & \\
| \text{None} \rightarrow & \\
| \text{Some res' } \rightarrow & \\
\text{unlockNode n; } & \\
\text{searchStrOp } \omega r k
\end{aligned}
\]

\[
\begin{aligned}
\text{unlockNode n; } & \\
\end{aligned}
\]

\[
\begin{aligned}
\text{res}
\end{aligned}
\]

Fig. 7. The link template algorithm with a proof outline.

\( \bullet \text{dom}(I) \cup N \) \( \text{,} \) 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 (used for example, in the precondition of lockNode).
Finally, we use a sets of keys $\{R_A\}$ at location $\gamma_c$ in order to encode the state of the specification program $\text{searchStrSpec}$. 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, $C(I)$.

In Iris, the invariant assertion $\text{Inv}^N$ denotes state that can be shared between threads that always satisfies the invariant $\text{Inv}$. Invariants are tagged with name spaces (our invariant is tagged with the name space $N$) so as to enforce that the invariant is reestablished in between atomic steps of the program. This is achieved in Iris by tagging the proof goal with a mask $E$ of available invariants, and allowing the proof to consider an atomic action only when all invariants opened by the previous action have been closed. E.g. the mask $\mathcal{T} \setminus \mathcal{N}$ encodes the fact that $\mathcal{N}$ has been opened and needs to be reestablished before proceeding. We represent this as superscripts on our Hoare-style assertions, and omit the superscript if the mask is $\mathcal{T}$, the set of all invariant names.

Let us now step through the proof of $\text{searchStrOp}$. The code begins with a call to $\text{traverse}$ on line 21. To satisfy $\text{traverse}$’s precondition, we need to open the invariant and use the fact that $\phi(I) \Rightarrow r \in \text{dom}(I)$. We can then take a snapshot of the domain of the global invariant using $\text{AUTH-SET-SNAPSHOT}$, and we add $\text{inFP}(r)$ to our context. Note that $\phi(I) \Rightarrow I_{\Omega}(r)_{\Omega}(k) = 1$ which by the invariant and (1) gives us $k \in \text{inreach}(I, r)$. So we also use $r \in \text{dom}(I)$ to unfold the iterated separating conjunction containing the inreach sets in the invariant, and take a snapshot of the inreach set of $r$ using $\text{AUTH-SET-SNAPSHOT}$ to add $\text{inlnr}(r, k)$ to our context. The resulting context is depicted in line 20.

After the call to $\text{traverse}$, we can add its postcondition (line 17) to our context. The next step is the call to $\text{decisiveOp}$, for whose precondition we need to show that $k \in \text{inset}(I_n, n)$. This follows from $\text{inlnr}(n, k) \land k \notin \text{outset}(I_n)$ by the definition of the invariant, the inreach node label property (1), and (5).

We then look at the two possible outcomes of $\text{decisiveOp}$. In the case where it returns None, our context is unchanged, so we execute $\text{unlockNode}$ using the Node$(n, I_n, I_n)$ in our context. We can use the specification of $\text{searchStrOp}$ on the recursive call on line 26 to complete this branch of the proof.

On the other hand, if $\text{decisiveOp}$ succeeds, we get back a modified node Node$(n, I_n, I_n')$ with a new interface $I_n'$ that satisfies the search structure specification $\Psi_{\omega}(k, C(I_n), C(I_n'), \text{res})$ locally (line 27). Since we have modified the search structure, we now have an extra proof obligation for the refinement proof: we need to show that the result value of the specification program if executed on line 21. To satisfy $\text{traverse}$’s precondition, we can add its postcondition (line 17) to our context. The next step is the call to $\text{decisiveOp}$, for whose precondition we need to show that $k \in \text{inset}(I_n, n)$. This follows from $\text{inlnr}(n, k) \land k \notin \text{outset}(I_n)$ by the definition of the invariant, the inreach node label property (1), and (5).

We then look at the two possible outcomes of $\text{decisiveOp}$. In the case where it returns None, our context is unchanged, so we execute $\text{unlockNode}$ using the Node$(n, I_n, I_n)$ in our context. We can use the specification of $\text{searchStrOp}$ on the recursive call on line 26 to complete this branch of the proof.

On the other hand, if $\text{decisiveOp}$ succeeds, we get back a modified node Node$(n, I_n, I_n')$ with a new interface $I_n'$ that satisfies the search structure specification $\Psi_{\omega}(k, C(I_n), C(I_n'), \text{res})$ locally (line 27). Since we have modified the search structure, we now have an extra proof obligation for the refinement proof: we need to show that the result value of the specification program if executed at this point is the same as what $\text{searchStrOp}$ will return. This is essentially the linearization point of this algorithm.

To do this, we first open the invariant to get temporary access to its contents, which sets the mask to $\mathcal{T} \setminus \mathcal{N}$ (line 28). We now have the resources to execute the specification program, which changes the specification state from $C(I_n)$ to $C'$ and tells us they satisfy $\Psi_{\omega}(k, C(I), C', \text{res}')$.

On the concrete side, we only know $\Psi_{\omega}(k, C(I_n), C(I_n'), \text{res})$. If we can lift this knowledge to the whole graph, we can use the fact that $\Psi_{\omega}$ determines the result uniquely to obtain $\text{res} = \text{res}'$:

$$\Psi_{\omega}(k, C, C_1', \text{res}_1) \land \Psi_{\omega}(k, C, C_2', \text{res}_2) \Rightarrow C_1' = C_2' \land \text{res}_1 = \text{res}_2.$$
Table 1. Summary of templates and instantiations verified in Iris/Coq and GRASShopper. For each algorithm or library, we show the number of lines of code, lines of proof annotation (including specification), total number of lines, and the proof-checking / verification time in seconds.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Module</th>
<th>Code</th>
<th>Proof</th>
<th>Total</th>
<th>Checker Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iris/Coq</td>
<td>Flow library</td>
<td>-</td>
<td>99</td>
<td>99</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Link template</td>
<td>29</td>
<td>518</td>
<td>547</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>Give-up template</td>
<td>35</td>
<td>406</td>
<td>441</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Lock-coupling template</td>
<td>37</td>
<td>565</td>
<td>602</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td>101</td>
<td>1588</td>
<td>1689</td>
<td>139</td>
</tr>
<tr>
<td>GRASShopper</td>
<td>Flow library</td>
<td>-</td>
<td>127</td>
<td>127</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Array library</td>
<td>132</td>
<td>300</td>
<td>432</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>B+ tree</td>
<td>73</td>
<td>161</td>
<td>234</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>B-link tree (core)</td>
<td>105</td>
<td>312</td>
<td>417</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>B-link tree (half split)</td>
<td>83</td>
<td>207</td>
<td>290</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td>Hash table (link)</td>
<td>58</td>
<td>176</td>
<td>234</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Hash table (give-up)</td>
<td>64</td>
<td>141</td>
<td>205</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Lock-coupling list</td>
<td>83</td>
<td>231</td>
<td>314</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td>598</td>
<td>1655</td>
<td>2253</td>
<td>401</td>
</tr>
</tbody>
</table>

We now have the context in line 31, from which we can close the invariant. We finally execute the call to unlockNode as above, and complete the proof. The proof of traverse follows a similar line-by-line reasoning using the appropriate specifications of helper functions, except that it does not need to execute the specification program. We omit a description for space reasons but the intermediate contexts are shown in Fig. 7.

4.4 Proofs of Template Implementations
To obtain a verified implementation of the link template, one needs to provide code for the helper functions in Fig. 6 that satisfies the given specifications. As can be seen from the specifications, these functions do not have access to the invariant, and hence the shared state, 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. Indeed, using some of the core rules of ReLoC, we can show that we can reason about the implementation program using standard separation logic 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 an Iris proof.

5 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 [Shasha and Goodman 1988], 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 [Frumin et al. 2018]. 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 [Piskac et al. 2014]. 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. 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 3.10) and that they enable frame preserving updates (Theorem 3.11) as well as some basic general lemmas about flow interfaces. The proofs of these properties are provided in Appendix A. The template proofs parameterize over the implementation of the helper functions, the heap representation predicate \( h \) as well as the actual flow domain, node label domain, and good condition \( v \).

All properties involving the specific flow domain elements and \( v \) 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. 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.

In addition to the helper functions of each data structure that are assumed by the templates we have also verified the half split operation for B-link trees. We have focused on the half split because it is the more complex part of the two-part split operation. The half split is also more critical for the correctness of the data structure as it is needed to ensure progress of insert operations. For the half split, 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 lock-coupling template where the abstract specification leaves the contents of the data structure unchanged. For the implementation of half split, we verify that the operation preserves the flow interface of the modified region as well as its contents.

Table 1 provides a summary of our development. Experiments have been conducted on an laptop with an Intel Core i7-5600U CPU and 16GB RAM. We split the table into one part for the templates (proved in Coq) and one part for the implementations (proved in GRASShopper). We note that for the B-link tree, B+ tree and hash table implementations, most of the work is done by the array library, which is shared between all these data structures. The size of the proof for the lock-coupling list 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 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 [Bansal et al. 2015; Piskac et al. 2013]. 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.

All the proof scripts and implementation files are available as part of the supplementary materials.

6 RELATED WORK

Our work builds on the Iris separation logic [Jung et al. 2017], the ReLoC logic [Frumin et al. 2018] for expressing refinement proofs in Iris, and the flow framework [Krishna et al. 2018] of...
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 [Krishna et al. 2018] to enable reasoning about a broader range of data structure updates. See §3 for more details on the obstacles to using the original flow framework in tools such as Iris or GRASShopper. Our changes to the framework enable automation and simplify the meta theory, without any noticeable loss in expressiveness. This also enables us to use flows in existing logics without the need to redefine the logic’s semantics as done by Krishna et al. [2018].

A recent paper [Meyer and Wolff 2019] demonstrates a similar proof modularity by decoupling the proof of data structures from that of the underlying manual memory management algorithm. Note that as our proofs are done in Iris, which does not support reasoning about deallocation, our proofs assume a garbage collected environment. However, by using Iron [Bizjak et al. 2019], a recent extension of Iris that allows proving absence of memory leaks, we can extend our proofs to the manual memory setting as well. It is a promising direction of future work to integrate this approach and our technique in order to obtain verified data structures where the user can mix-and-match the synchronization technique, memory layout, as well as the memory management system.

There exist many other concurrent separation logics that help modularize the correctness proofs of concurrent systems [Bornat et al. 2005; da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Feng et al. 2007; Heule et al. 2013; Nanevski et al. 2014; Raad et al. 2015; Vafeiadis and Parkinson 2007; Xiong et al. 2017]. 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 [Herlihy and Wing 1990]. The connection between contextual refinement and linearizability was established in [Filipovic et al. 2009]. 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 [Bouajjani et al. 2013, 2017; Chakraborty et al. 2015; Delbianco et al. 2017; Dodds et al. 2015; Khyzha et al. 2017; Liang and Feng 2013; O’Hearn et al. 2010; Zhu et al. 2015]. 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/Iris. Our proof methodology can be replicated in other separation logics that support user-defined ghost state, such as FCSL [Sergey et al. 2015], which would also be useful if one wanted to extend this work to non-linearizable data structures [Sergey et al. 2016].

Fully automated proofs of linearizability by static analysis and model checking have been mostly confined to simple list-based data structures [Abdulla et al. 2013; Amit et al. 2007; Bouajjani et al. 2015; Cerný et al. 2010; Dragoi et al. 2013; Vafeiadis 2009]. Recent work by Abdulla et al. [2018] 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 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


A PROOFS

We first prove that for every good condition $v$, the flow interface algebra $(\mathcal{FI}_v, \overline{V}, [\cdot], \oplus)$ is a resource algebra.

**Proof of Theorem 3.10.** We prove the conditions in turn:

- **RA-ASSOC:** By definition of composition and Lemma 4.5.
- **RA-COMM:** By definition of composition and Lemma 4.5.
- **RA-CORE-ID:** By definition of $[\cdot]$ and $I_e$.
- **RA-CORE-DEM:** By definition of $[\cdot]$.
- **RA-CORE-MONO:** By definition of $[\cdot]$.
- **RA-VALID-OP:** Because if $a$ is not valid then $a = I_i$, but $I_i \oplus I = I_i$ for all $I$.

\[ \Box \]

**Lemma A.1.** The flow interface algebra $\mathcal{FI}_v$ is a discrete, total, and unital camera.

**Proof.** All RAs are discrete cameras, and by Theorem 4.9, $\mathcal{FI}_v$ is an RA. It is easy to see that $I_e$ is a unit element, as it is valid, is a left identity of composition, and is its own core. Thus $\mathcal{FI}_v$ is unital, and by Lemma 1 of the Iris Documentation, it is a total camera as well.

Given an interface $I = (\text{in}, a, f)$, we define its canonical flow graph to be $H_I := (\text{in}, (N, N'^0, \lambda, \epsilon))$ where $N = \text{dom}(\text{in}), N'^0 = \{n' \mid \exists n \in \text{dom}(\text{in}). 0 \subseteq f(n, n')\}, \lambda = (\lambda n.a_e)$, and $\epsilon = f$. Note that because the capacity, and hence the flow map, is defined as a fixpoint, the flow map of $H_I$ is still $f$. Hence, $\text{int}(H_I) = I$. This construction will be useful when proving properties of interfaces.

We repeat the definition of contextual extension of interfaces here for clarity: we say an interface $(\text{in}, a, f)$ is contextually extended by $(\text{in}', a', f')$, written $(\text{in}, a, f) \preceq (\text{in}', a', f')$, if and only if

\[
\begin{align*}
\text{dom}(\text{in}) &\subseteq \text{dom}(\text{in}'), \\
\forall n \in \text{dom}(\text{in}). \text{in}(n) &= \text{in}'(n), \\
\forall n \in \text{dom}(\text{in}), n'. 0 \subseteq \text{in}(n) \Rightarrow f(n, n') &= f'(n, n'), \text{ and} \\
\forall n \in \text{dom}(\text{in}') \setminus \text{dom}(\text{in}), n'. f'(n, n') &= 0.
\end{align*}
\]

We identify the domain of an interface with the domain of the graphs it abstracts, i.e. $\text{dom}(I) = \text{dom}(\text{in})$ for $I = (\text{in}, a, f)$ and write $I$ for $\text{dom}(I)$ when it is clear from the context.
We first claim a useful property of this new inflow:
\[ \in' \in n \in_f \]
where we assume that old values of \( n \)

To see this, given \( n \)

and by the premise that \( I_2 \) sends no flow to the new nodes in \( I_1' \) we know that the sum of flow from \( I_2 \) is 0, so \( 0 \subseteq in'(n) \). On the other hand if \( n \in I_2' \subseteq I \),
\[
in'(n) = \in_2'(n) - \sum_{n' \in I_2} \in_2'(n') \cdot f_2'(n', n)
\]

This completes the proof of (11).

Second, we prove a useful property relating paths before and after the modification. Let \( n_1, \ldots, n_k \in I' \) such that \( n_1 \in I_1', n_2 \in I_2' \) for \( i \neq j \in \{1, 2\} \). If \( 0 \not\subseteq \in_1'(n_1) \), then we claim that
\[
f_1'(n_1, n_2) \cdot f_1'(n_2, n_3) \cdots f_1'(n_{k-1}, n_k) = f_1(n_1, n_2) \cdot f_j(n_2, n_3) \cdots f_j(n_{k-1}, n_k),
\]
where we assume that old values of \( f(n, n') = 0 \) when \( (n, n') \notin \text{dom}(f) \).

To see this, first assume \( n_1 \in I_1' \setminus I \). Then, \( I_2' = I_2 \) means that \( n \in I_2' \setminus I_1 \), which means \( f_1(n_1, n_2) = 0 \) and by (10) that \( f_1'(n_1, n_2) = 0 \). Thus, both old and new path products are identically zero. On the other hand, if \( n_1 \in I \) then by (8) we know that \( 0 \not\subseteq \in_1(n_1) \), which by \( I = I_1 \oplus I_2 \) means \( 0 \not\subseteq \in_2'(n_1) \). By \( I_1 \not\subseteq I_1' \) and (9) we have \( f_1(n_1, n_2) = f_1'(n_1, n_2) \). Now, since \( 0 \not\subseteq \in_1(n_1) \) \( f_1(n_1, n_2) \subseteq \in_1(n_2) \) (by definition of composition) we can inducively use (12) on the path \( n_2, n_3, \ldots, n_k \) and conclude the proof.

We now prove that \( I_1' \oplus I_2 \) exists by showing that \( H' := (in', G_1' \uplus G_2) = H_1' \cdot H_2 \neq H_\varepsilon \).

- First, the graph composition \( G_1' \uplus G_2 \) exists by the premise that \( I_1' \cap I_2 = \emptyset \).
- \( 0 \not\subseteq \in'(n) \) follows directly from (11).
- To show that \( H' \) is effectively acyclic, suppose \( n_1, \ldots, n_k \in I' \) where \( 1 \leq l \leq k \) is some path in \( H' \), then we show that if the product \( P = \in'(n_1) \cdot f_1'(n_1, n_2) \cdots f_1'(n_k, n_l) \neq 0 \) then a
correspondance path is nonzero in $H$. Since $P \neq 0$ we have $0 \subseteq \text{in}'(n_1) \subseteq \text{in}'(n_1)$, so by (12), we have that there is a nonzero cycle in the original interface, contradicting validity of $I$.

This shows that $H'$ is valid, hence let $I' := \text{int}(H')$ and we know by Lemma 3.8 that $I' = I'_1 \oplus I_2$.

We now show that $I \trianglelefteq I'$:

- (7): This follows from $I' = I'_1 \oplus I_2$ and (7).
- (8): This follows directly from (11).
- (9): Given any $n \in \text{dom}(\text{in})$ and $n'$ such that $0 \subseteq \text{in}(n)$, consider any path $f'_j(n, n_1) \cdots f'_k(n_k, n')$ in $I'$. By (12) (since $\text{in}(n) \subseteq \text{in}_i(n)$), we know that this path product is the same as $f_i(n, n_1) \cdots f_i(n_k, n')$ the one in $I$. Thus, $f'_i(n, n') = f(n, n')$.
- (10): Given any $n \in \text{dom}(\text{in}') \setminus \text{dom}(\text{in})$ and $n'$, since $I'_2 = I_2$ we know that $n \in I'_1 \setminus I_1$. So if we look at any path $f'_i(n, n_1) \cdots f'_k(n_k, n')$, the first edge on this path $f'_i(n, n_1) = 0$ by (10) on $I_1 \leq I'_1$. Hence, $f(n, n') = 0$.

□

Before we prove AUTH-FI-UPD, let us give the definition of non-deterministic frame-preserving updates: It is possible to do a frame-preserving update from $a \in M$ to $B \subseteq M$, written $a \rightsquigarrow B$, if

\[
\forall a_i^j \in M^j. \overline{\text{V}}(a \cdot a_i^j) \Rightarrow \exists b \in B. \overline{\text{V}}(b \cdot a_i^j)
\]

PROOF OF AUTH-FI-UPD. Let $I, I_1, I'_1$ be valid interfaces such that $I \trianglelefteq I'_1$. Let $B$ be the set

\[
\{(\bullet I', \circ I'_1) \mid I \trianglelefteq I' \land \exists I_2. I = I_1 \oplus I_2 \land I' = I'_1 \oplus I_2\}.
\]

For any $a \in M^j$ such that $\overline{\text{V}}((\bullet I, \circ I_1) \cdot a)$:

- If $a = \bot$: $\overline{\text{V}}((\bullet I, \circ I_1))$ means that there exists $I_2$ such that $I = I_1 \oplus I_2$. By Theorem 3.11 (whose other premises are true because we assume the GC only allocates fresh addresses), there exists $I' = I'_1 \oplus I_2$ such that $\overline{\text{V}}(I')$. This means that $\overline{\text{V}}((\bullet I', \circ I'_1) \cdot (\bullet I, \circ I_1))$ is in $B$.
- If $a = (\bullet I_3, b)$: then by definition of AUTH(F1v) we know that $(\bullet I_3, b) \cdot (\bullet I, \circ I_1)$ is not valid, so this branch is complete.
- If $a = (\circ I_3)$: then by $\overline{\text{V}}((\bullet I, \circ I_1) \cdot \circ I_3)$ we know that exists $I_4$ such that $I = I_1 \oplus I_3 \oplus I_4$. Let $I_2 = I_3 \oplus I_4$, and by Theorem 3.11 again there exists $I' = I'_1 \oplus I_2 = I'_1 \oplus I_3 \oplus I_4$ such that $\overline{\text{V}}(I')$. This means that $\overline{\text{V}}((\bullet I', \circ I'_1) \cdot \circ I_3)$ and $(\bullet I', \circ I'_1) \in B$.

□

A.1 Flow Interface Lemmas

LEMMA A.2 (flowint_comp_fp). Given interfaces such that $I = I_1 \oplus I_2$, $\text{dom}(I) = \text{dom}(I_1) \cup \text{dom}(I_2)$.

PROOF. Follows from the fact that the definition of interface composition is disjoint graph union of the witness flow graphs.

□

The following lemmas are essentially the definition of interface composition, restricted to the case of composing singleton graphs.

LEMMA A.3 (int_comp_fold). If $I_x, I_y$ are valid singleton interfaces on nodes $x \neq y$ respectively, and $I_y^f(y, x) = 0 \land I_x^\text{in}(x) \cdot I_x^f(x, y) = I_y^\text{in}(y)$, then there exists a valid interface $I = I_x \oplus I_y$. 

LEMMA A.4 (int_comp_unfold). If \( I = I_x \oplus I_y \) is a valid interface, for singleton interfaces \( I_x, I_y \) on nodes \( x \neq y \) respectively, and \( I_y f(x, y) = 0 \), then
\[
I_x^{in}(x) = I^{in}(x)
\]
\[
I_y^{in}(y) = I^{in}(y) + I_x^{in}(x) \cdot I_x f(x, y)
\]
\[
\forall n. I f(x, n) = I x f(x, n) + I_x f(x, y) \cdot I_y f(y, n)
\]
\[
\forall n. I f(y, n) = I_y f(y, n).
\]

LEMMA A.5 (int_comp_fold_3). If \( I_x, I_y, I_z \) are valid singleton interfaces on nodes \( x \neq y \neq z(\neq x) \) respectively, and
\[
I_y f(y, x) = 0 \land I_z f(z, x) = 0 \land I_z f(z, y) = 0
\]
\[
\land I_x^{in}(x) \cdot I_x f(x, y) \leq I_y^{in}(y)
\]
\[
\land I_x^{in}(x) \cdot I_x f(x, y) + I_y^{in}(y) \cdot I_y f(y, z) \leq I_z^{in}(z),
\]
then there exists a valid interface \( I = I_x \oplus I_y \oplus I_z \).

LEMMA A.6 (int_comp_unfold_3). If \( I = I_x \oplus I_y \oplus I_z \) is a valid interface, for singleton interfaces \( I_x, I_y, I_z \) on nodes \( x,y,z \) respectively, and \( I_y f(y, x) = I_z f(z, x) = I_z f(z, y) = 0 \), then
\[
I_x^{in}(x) = I^{in}(x)
\]
\[
I_y^{in}(y) = I^{in}(y) + I_x^{in}(x) \cdot I_x f(x, y)
\]
\[
I_z^{in}(z) = I^{in}(z) + I_x^{in}(x) \cdot I_x f(x, z) + I_y^{in}(y) \cdot I_y f(y, z)
\]
\[
\forall n. I f(x, n) = I x f(x, n) + I_x f(x, y) \cdot I_y f(y, n) + I_x f(x, z) \cdot I_z f(z, n)
\]
\[
\forall n. I f(y, n) = I y f(y, n) + I_y f(y, z) \cdot I_z f(z, n)
\]
\[
\forall n. I f(z, n) = I_z f(z, n).
\]

LEMMA A.7 (contextualLeq_impl_fp). If \( I \preceq I' \) then dom(I) = dom(I').

Proof. This is one of the conditions of the definition of contextual extension \( \preceq \). □

LEMMA A.8 (acyclic_2). If \( I_x, I_y \) are singleton interfaces on nodes \( x \neq y \) respectively such that \( \overline{V}(I_x \oplus I_y) \) and \( I_x^{in}(x) \cdot I_y^{in}(y) \neq 0 \), then \( I_y f(y, x) = 0 \).

Proof. By contradiction: if not, then \( I_x^{in}(x) \cdot I_y^{in}(y) \cdot I_y f(y, x) \neq 0 \), which contradicts the effectively acyclic assumption. □

LEMMA A.9 (proj). If \( I = I_x \oplus I_y \) and \( x \in \text{dom}(I_x) \), then \( I^{in}(x) \subseteq I_x^{in}(x) \).

Proof. By definition of interface composition, \( I_x^{in}(x) \) is the sum of \( I^{in}(x) \) and another term, and all flow graph values are non-negative. □

LEMMA A.10 (step). If \( I = I_1 \oplus I_2 \) with \( x \in \text{dom}(I_1), y \in \text{dom}_2(I_1 f) \), and \( I f = \epsilon \), then \( y \in \text{dom}(I_2) \) and \( I_1^{in}(x) \oplus I_1 f(x, y) \subseteq I_y^{in}(y) \).

Proof. If \( y \notin \text{dom}(I_2) \), then \((x,y)\) would be an outgoing edge of \( I \), contradicting \( I f = \epsilon \). The inequality follows easily from the definition of composition and the fact that all flow graph values are non-negative. □
A.2 Edgeset Framework

We prove the lemma that corresponds to the Keyset Theorem in our encoding of the edgeset framework.

**Proof of Lemma 4.1.** This proof proceeds by case splitting on $\omega$ and is fairly obvious for insert, successful delete, and successful search calls. For the other cases, one needs to show that $k \in (\text{inset}(I_n, n) \setminus \text{outset}(I_n))$ implies that $k \not\in C(I_2)$. If this were not true, by (GS1) there would be a path to some $n' \in I_2$ with $k$ in its pathset as well as the path to $n$, which contradicts (GS2). □

A.3 Link Template Encoding

Recall the constraints on the global interface used in the link template proof:

$$\phi(I) := (\forall n. k. I_{in}(n)|_{is}(k) = (n = r \ ? 1 : 0)) \land I_{ir}^a \neq \top \land I_f^e = \epsilon$$

It is easy to see that $\phi$ satisfies the following properties:

- $\phi(I) \Rightarrow r \in \text{dom}(I)$
- $\phi(I) \Rightarrow \text{dom}_2(I_f^e) = \emptyset$
- $\phi(I) \Rightarrow \forall k. k \in \text{inreach}(I, r)$

$$\phi(I) \land I \leq I' \land I_{ir}^a \neq \top \Rightarrow \phi(I')$$

The encoding of inreach in the node label also satisfies the following properties:

**Lemma A.11.** If a valid interface $I = I_n \oplus I'$ such that $\text{dom}(I_n) = \{n\}$ and $\phi(I)$, then $I_{ir}^a(n) = I_{ir}^a(n)$.

**Lemma A.12.** If valid interfaces $I = I_n \oplus I_o$ and $I' = I'_n \oplus I_o$ such that $\text{dom}(I_n) = \text{dom}(I'_n) = \{n\}$, $\phi(I)$, and $I_{ir}^a(n) = I_{ir}^a(n)$, then for all $n' \in \text{dom}(I')$. $I_{ir}^a(n') = I_{ir}^a(n')$.

A.4 Give-up and Lock-coupling Templates

Since these templates do not use the inreach, they use a simpler good condition on nodes:

$$\nu(n, I)C(I) \subseteq \text{inset}(I, n) \setminus \text{outset}(I) \land (\forall n', n''. \ n' = n'' \lor \text{es}(I, n, n') \cap \text{es}(I, n, n'') = \emptyset)$$

They use a similar global interface constraint:

$$\phi(I) := (\forall n. k. I_{in}(n)|_{is}(k) = (n = r \ ? 1 : 0)) \land I_f^e = \epsilon$$

Which satisfies the same properties:

- $\phi(I) \Rightarrow r \in \text{dom}(I)$
- $\phi(I) \Rightarrow \text{dom}_2(I_f^e) = \emptyset$
- $\phi(I) \Rightarrow \forall k. k \in \text{inreach}(I, r)$