A Formal Interface for Concurrent Search Structure Templates

Duc-Than Nguyen and William Mansky

University of Illinois Chicago, USA
{dnguye96,mansky1}@uic.edu

Abstract. Concurrent search structure templates simplify reasoning about concurrent data structures by dividing them into synchronization patterns and underlying sequential data structures, which can be designed and verified independently. Past work has used templates to decompose the verification of specific data structures and reuse some of their components, but has not given a general characterization of what makes a concurrency template or whether it can be applied to a given data structure. In this paper, we formally define the *interface* provided by search structure templates, and show that as long as a sequential data structure component and a concurrency template are proved to implement this interface, they can immediately compose into a verified concurrent data structure with no additional effort. Thus, from any m verified data structure components and n verified concurrency components, we obtain $m \times n$ verified data structures. We validate our interface by verifying two data structure instances (linked list and binary search tree) and three concurrency templates (coarse-grained locking, lock coupling, and give-up), which can be freely plugged into a top-level verified data structure whose correctness proof is parameterized by the interface.

Keywords: Concurrent data structures · Separation logic · Compositional verification.

1 Introduction

Concurrent search structure templates were introduced by Krishna et al. [3] as an approach to modular verification of concurrent data structures, based on earlier design work by Shasha and Goodman [12]. The approach consists of a programming pattern and specification style that decomposes concurrent data structures into *concurrency templates* (lock coupling, give-up) and *sequential data structures* (linked list, hash table), using *flow interfaces* [4] to describe the connections between local and global data structure constraints. Later work in the same line extends the template approach to adding multicopy support [8] (e.g., log-structure merging), or combining thread-safe node implementations into larger data structures [9]. However, these applications are qualitatively different from the original promise of the approach: that we can decompose both the implementation and the proof of a concurrent data structure into a *concurrency-unaware* data structure component and a *data-structure-unaware* concurrency component. This form of modularity is, to the best of our knowledge, unique in the literature, at least when the concurrent component is fine-grained (i.e., implemented with a lock per node or lock-free operations rather than one big lock). It also poses unique challenges in both implementation and specification/verification.

Krishna et al.'s presentation of templates explains the high-level idea of the approach and demonstrates several examples, but does not formally define templates or data structures. In particular, a modularization technique should have a well-defined *interface* for each of its components, so that we know that components written and verified according to the technique can be successfully combined. In a truly compositional approach, given m data structure components and n concurrency templates, we should immediately obtain $m \times n$ verified concurrent data structures. The original presentation of templates does not realize this (nor does it claim to): it presents three verified templates and five verified data structures, but each data structure is combined with only one template. In fact, close examination of the examples shows that different templates use different and incompatible specifications for the data structure component. In this paper, we rectify this by presenting a **formal interface for concurrent search structure templates**, implemented as a library in C and a collection of typeclasses in Rocq [13], that guarantees $m \times n$ verified data structures from $m + n$ components. More specifically:

- We describe an approach to **implementing** data structures and synchronization mechanisms so that they can be freely combined. This is harder than it might sound: we must define data structure implementations without explicitly leaving space for concurrency metadata like locks, but in a way that still composes with concurrency templates without any glue code. We propose an approach that can be implemented even in relatively non-modular languages like C.
- We give a formal **interface** for data structure components and concurrency templates, each of which consists of a set of functions with fixed separation logic specifications. These specifications follow the style of Krishna et al., but we clarify which specifications should be considered the official interface and which are derived or implementation-specific, as well as ensuring that data structure components are never assumed to include concurrent features and concurrency templates never rely on data structure details. We verify top-level concurrent data structure operations (`insert` and `lookup`) against these specifications, guaranteeing that they work correctly for any data structure component and template that satisfy the interface.
- We exhibit two data structures (linked list and binary search tree) and three concurrency templates (coarse-grained locking, lock coupling, and give-up) that satisfy our interface, and from them **freely obtain six verified concurrent data structures**, implemented in C and verified using the Verified Software Toolchain (VST) [1].

Related Work

Our work builds heavily on the presentation of templates by Krishna et al. [3], aiming to expand their approach into a formal and compositional framework. In Krishna et al.'s formulation, each template has its own specifications for data structure functions, and these specifications may be incompatible with each other: for instance, the lock-coupling template expects the data structure's insertion function `insertOp` to create a new node, while the give-up and link templates expect it to add keys to an existing node. Thus each template is composed with different data structure implementations, which must be designed to work with that particular template, and each template must implement and verify its own top-level functions like `lookup` and `insert`. Furthermore, data structure components are written in Grasshopper [10], while templates are written in Iris's HeapLang [2], so 1) there may be gaps between the properties expected by templates and those guaranteed by data structures, and 2) code components cannot be combined into a single executable data structure. In contrast, our framework standardizes the interfaces for data structures and templates, so that any data structure component in the framework is guaranteed to compose with any template: more precisely, we provide a single implementation of the concurrent `insert` and `lookup` functions (see Section 4.3) that calls functions from the template interface and is proved correct once and for all, quantified over an arbitrary data structure and template implementation. Furthermore, our implementation is entirely in C and VST, so the combination of any two components yields an executable C implementation of a concurrent data structure with an end-to-end correctness proof.

Our work is also closely related to Nguyen et al. [6]'s VST implementation of search structure templates without flow interfaces, which identified several obstacles to compositionality: inconsistent specifications for `insertOp`, ambiguous boundaries between data structures and templates, and the need to allocate synchronization metadata that may depend on data structure details. Our work overcomes all of these obstacles, and demonstrates that flow interfaces are a sufficient abstraction to communicate structural information between data structures and templates. Nguyen et al. also observed that the template approach implicitly involves a third style of operations, no-op maintenance operations like rotation in BSTs or node splitting in B-trees, which do not generally decompose into data structure and concurrency components. Our framework does not yet account for this third class of operations, but we believe they could be added into our top-level functions without breaking compositionality (but also without benefiting from it).

2 Background

2.1 Concurrent Search Structure Templates

Concurrent search structure templates [3] are an abstraction technique for decomposing the implementation/verification of a concurrent search structure (i.e., a data structure that supports lookup, insert, and possibly delete operations) into a concurrency part (the *template*) and a data structure part. A data structure implements a `node` type and core *local* operations on those nodes—e.g., looking up the value at a given node, or inserting a key at a specified point in the data structure—as well as some helper functions. A template implements top-level concurrent data structure operations like insertion and deletion, interacting with the underlying nodes only via the specified functions. The core of each template is a `traverse` function that moves through the data structure using the template's synchronization mechanism, ultimately finding the node at which a local data structure operation should be performed. In this way, any data structure that implements the appropriate functions can be plugged into the template to yield a concurrent data structure.

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

```
let insert r k =
  lockNode r;
  let n = traverse r r k in
  let res = insertOp n k in
  unlockNode n;
  res
```

Fig. 1: The lock-coupling search structure template

Figure 1 shows an example search structure template in ML-like pseudocode. This template's concurrency control mechanism is *lock coupling*, as can be seen in the `traverse` function, where we acquire the lock on the next node before releasing the lock on the current node. The node to travel to during traversal is selected by a black-box function `findNext` provided by the data structure; all the template needs to know is that it has some way of choosing a next node to examine. Once the appropriate node for the key has been found, the template returns it to a top-level function such as `insert` that calls out to the data structure to perform the actual insertion on the node. Thus, the `traverse` and `insert` functions can be written and verified without knowing anything about the target data structure other than its synchronization mechanism, as long as the data structure implements `findNext` and `insertOp` operations with the required semantics.

Krishna et al., following Shasha and Goodman's original work on template algorithms [12], define three templates: lock coupling, give-up, and link. The lock-coupling template acquires locks on both parent and child nodes when moving from one to the other, guaranteeing that the link followed is never out of date. The link template only acquires a lock on the current node, and its `traverse` may return an invalid node; in this case, the operation (insert, lookup, etc.) may fail, and if it does then the operation restarts from the root of the data structure. The give-up template stores an explicit range of expected keys in each node, and checks this range at each node it traverses, restarting the traversal if it ever reaches a node whose range does not include the target key. For each of these templates, they implement and verify a `traverse` function and use it to implement and verify the top-level data structure operations.

Templates and data structures are verified using concurrent separation logic, and the interface between data structures and templates relies on two key constructs, *flow interfaces* and *logical atomicity*, which we describe in more detail in the following sections. The interface provided by a data structure implementation includes a predicate of the form $\mathsf{node}\,(n, I_n, C_n)$, where n is the node itself (i.e., a pointer to the node data structure), I_n is a flow interface for n, and C_n is n's contribution to the state of the overall data structure, e.g., the set of keys contained in n. The flow interface I_n summarizes n's place in the data structure, allowing us to perform local modifications to n without invalidating the rest of the data structure. Data structure operations such as `insertOp`, `lookupOp`, and `findNext` are specified in terms of the `node` predicate. Template functions are specified using logical atomicity, guaranteeing that they provide thread-safe implementations of traversal, lookup, and insertion.

2.2 Flow interfaces

Flow interfaces [4] are a generic mechanism for capturing the relationship of a single node or collection of nodes to a larger data structure, and can be implemented in a separation logic (e.g., Iris) using ghost state. A flow interface I abstracts a subset of nodes in a data structure into the flow of some sort of information through that subset. A common example is the number of paths in and out of the subset in question; for search structure templates, the relevant flow is that of *keys*. A key k flows into a node n in a data structure if a search for k will always reach n; k flows out of n to n' if a search for k will pass through n and move on to n'. For example, in a binary search tree whose root node n contains key k and has left child n_l, the flow interface for n will have an inflow $I_n.in(n)$ equal to the set of all possible keys, and outflow $I_n.out(n_l)$ equal to the set of keys less than k. This flow captures the notion of the place where a key "belongs" in a data structure, whether or not it is currently present in the data structure.

Formally, given a set of nodes N and a set of possible keys K, a flow interface I is a pair of an inflow $I.in : N \rightharpoonup 2^K$ recording the set of keys that "flow into" each node in I, and an outflow $I.out : N \rightharpoonup 2^K$ recording the set of keys that "flow out to" each successor of I, where N is the set of nodes in the data structure and K is the set of keys. Given a node n with interface I_n[1], its *inset* $\mathsf{ins}(I_n, n) := I_n.in(n)$ is the set of keys for which a search will visit node n. Its *outset* to a node m, $\mathsf{outs}(I_n, m) := I_n.out(m)$, is the set of keys for which a search will leave n and proceed to m, with $\mathsf{outs}(I_n)$ defined as $\bigcup_m \mathsf{outs}(I_n, m)$. Then n's keyset $\mathsf{ks}(I_n, n) := \mathsf{ins}(I_n, n) \setminus \mathsf{outs}(I_n)$ is the set of keys that "belong" in n, in the sense that n is the only node where they could appear/be inserted in the data structure. The keysets of all nodes are a partition of the set of possible keys: each key has exactly one place in the data structure where it could be stored.

[1] In general, a flow interface may contain multiple nodes; here we use a single-node interface I_n as an illustrative example.

By convention, the domain of an interface is the domain of its inflow, i.e., $\mathrm{dom}(I) := \mathrm{dom}(I.in)$. The flow interface for a data structure is the composition of the interfaces for its individual nodes. Two flow interfaces I_1 and I_2 with disjoint domains can be composed to yield a flow interface $I_1 \oplus I_2$ that covers all the nodes in both of their domains. This is done by converting any outflow between the two interfaces into "internal" flow: outflow from I_1 to a node n in I_2 is subtracted from $I_2.in(n)$, and vice versa. For nodes $n \notin \mathrm{dom}(I_1) \cup \mathrm{dom}(I_2)$, the outflow $(I_1 \oplus I_2).out(n)$ is simply $I_1.out(n) \cup I_2.out(n)$. We can also add new nodes to a flow interface without affecting the existing flow: an interface I' is said to be a *contextual extension* of an interface I, written $I \precsim I'$, if it differs from I only by enlarging the domain with fresh nodes, whose outgoing edges do not affect the outflow of the existing region. Formally, $I \precsim I'$ when $\mathrm{dom}(I) \subseteq \mathrm{dom}(I')$, $I.in(n) = I'.in(n)$ for all $n \in \mathrm{dom}(I)$, and $I.out(n') = I'.out(n')$ for all $n' \notin \mathrm{dom}(I)$.

2.3 Logical atomicity

Logical atomicity [11] is used to lift a sequential data structure specification to the concurrent setting. A logically atomic triple has the form

$$\forall a.\ \langle \mathtt{P}_l \mid \mathtt{P}_p(a) \rangle\ \mathtt{c}\ \langle \mathtt{Q}_l \mid \mathtt{Q}_p(a) \rangle$$

where $\mathtt{P}_l$ and $\mathtt{Q}_l$ are local preconditions and postconditions, akin to a standard Hoare triple, while $\mathtt{P}_p$ and $\mathtt{Q}_p$ are public preconditions and postconditions, parameterized by a shared abstract value a. This asserts that the program `c` atomically updates the abstract data a from a state satisfying $\mathtt{P}_p$ to a state satisfying $\mathtt{Q}_p$, with no intermediate states visible to any other thread. For instance, the top-level specification for the `insert` operation on a (linearizable) concurrent data structure can be written as

$$\begin{gathered}\forall C.\ \langle \mathtt{Ref}(css) \mid \mathsf{CSS}(css, C) \rangle \\ \mathtt{insert(css,\ k,\ v)} \\ \langle \mathtt{Ref}(css) \mid \mathsf{CSS}(css, C\,[\mathtt{k} \leftarrow \mathtt{v}]) \rangle\end{gathered}$$

where `Ref` is a per-thread handle to the data structure at css and CSS is a shared assertion linking the data in memory at css to an abstract map C from keys to values. The triple says that `insert` atomically updates the state of the data structure from C to $C\,[\mathtt{k} \leftarrow \mathtt{v}]$, without mentioning the details of either the synchronization mechanism or the underlying data structure implementation. When the local pre- and postcondition are `emp`, we also write the triple as $\forall a.\ \langle P(a) \rangle\ \mathtt{c}\ \langle Q(a) \rangle$.

An atomic triple is proved by identifying a single *linearization point* for each execution of `insert` where the visible state of the data structure transitions from $\mathsf{CSS}(C)$ to $\mathsf{CSS}(C\,[\mathtt{k} \leftarrow \mathtt{v}])$. The `traverse` function for a template must satisfy a logically atomic specification that says roughly "this function finds the node where key `k` belongs". The `traverse` specification can then be used to prove atomic specifications for the data structure operations, lifting the sequential specifications for insert, lookup, etc. to the concurrent setting.

3 Compositional Implementation of Search Structure Templates

```c
typedef struct node {
  int key; void *value;
  node *next; lock_t l;
} node;
```

(a) Linked-list node with lock

```c
typedef struct node {
  int key; void *value;
  node *left, *right; lock_t l;
} node;
```

(b) BST node with lock

```c
typedef struct node {
  int key; void *value;
  node *next; lock_t l;
  int min, max;
} node;
```

(c) Linked-list node with lock and range

```c
typedef struct node {
  int key; void *value;
  node *left, *right; lock_t l;
  int min, max;
} node;
```

(d) BST node with lock and range

Fig. 2: Four different `node` implementations

Suppose we have implemented four concurrent data structures in C: a linked list and a binary search tree (BST), each with both the lock-coupling and give-up synchronization patterns. Figure 2 shows the definition of the `node` type for each of these implementations. All four node types include a `key` and `value` field; linked-list nodes have a `next` field, while BST nodes have a `left` and `right` child. Both kinds of nodes have a lock for synchronization; give-up nodes also have a range (`min` and `max`) indicating the range of keys allowed in this node and its children. In a modular approach, we should be able to define each component separately (linked list, BST, lock-coupling, give-up), and then freely combine them to yield these four node implementations. Furthermore, data structure functions should only access fields from the data structure component, and concurrent functions should only access fields from the synchronization component.

```c
typedef struct node {
  int key; void *value; node *next;
} node;
```

(a) Linked-list node

```c
typedef struct node {
  int key; void *value; node *left, *right;
} node;
```

(b) BST node

```c
typedef struct md_entry {
  lock_t l; int min, max;
} md_entry;
```

(c) Give-up metadata

```c
typedef struct css {
  node *root; md_table metadata;
} css;
```

(d) Top-level concurrent data structure

Fig. 3: Modular definition of concurrent search structures

Writing code that composes in this way is quite difficult in most languages. In languages with multiple inheritance like C++, we could define a `BST_giveup_node` that inherits fields and methods from `BST_node` and `giveup_node`, but we would still have to declare a class for each combination of data structure and template. Writing the template node as a wrapper around the data structure node (as done by Nguyen et al. [6]), or vice versa, entangles the two in a way that makes both programming and proving less modular. For our purposes, and working in C, we settle for a nonlocal but highly compositional approach: we store template and data structure fields separately, with the template maintaining a hash table that maps each data structure node to its associated template fields, as shown in Figure 3. Each data structure implements a `node` type, each template implements an `md_entry` type, and the top-level `css` type is defined once and for all.

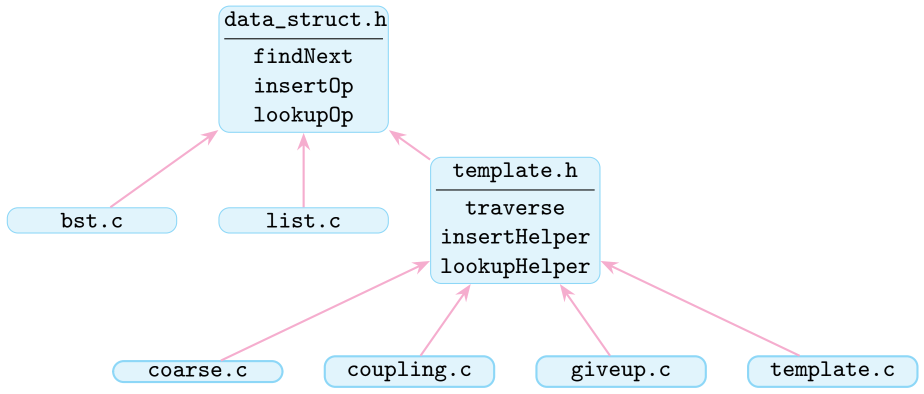


Fig. 4: Implementation layout

Once we have decomposed the data structure type, we can then implement the functions for each component. The data structure defines the local `findNext`, `insertOp`, and `lookupOp` functions as operations on `node`s; the template defines `traverse`, along with the `insertHelper` and `lookupHelper` functions, to maintain metadata during update or search operations, using the details of `md_entry` but treating the `node` type as a black box; and the top-level functions `insert`, `lookup`, etc. are defined once and for all on `css` by calling the template functions, generic in the implementation of both `node` (the data structure) and `md_entry` (the template). The dependency graph of our examples is shown in Figure 4, with data structures `bst` and `list`, and templates `coarse`, `coupling`, and `giveup`. The `.h` files define the interfaces for data structures and templates, while `template.c` uses those interfaces to implement the top-level operations `insert` and `lookup`, which are identical across all instances and simply combine calls to `traverse` with the corresponding template operations.

4 Specifying Data Structures and Templates as Interfaces

Our verification process for concurrent search structure templates follows the same modular architecture as the code:

- The interface for data structures is an abstract `node` predicate, plus sequential Hoare triples for the local data structure functions (`findNext`, `insertOp`, `lookupOp`) that characterize their behavior at the level of `node`s.
- The interface for templates takes an arbitrary data structure as a parameter and provides logically atomic triples for the template functions (`traverse`,

`insertHelper`, `lookupHelper`), which lift the data structure operations to thread-safe operations on a concurrent data structure represented by an abstract CSS predicate defined by the template.

- The top-level `insert` and `lookup` functions are implemented and verified once and for all, taking both a data structure and a template as a parameter. As shown in Figure 5, each template operation is implemented simply by calling `traverse` and a helper function from the template interface. Using the template interface's specifications, we can prove that these `insert` and `lookup` functions satisfy atomic specifications on CSS:

$$\forall C.\ \langle \mathsf{CSS}(css, C)\rangle\ \texttt{insert(css, x, v)}\ \langle \mathsf{CSS}(css, C\,[x \leftarrow v])\rangle$$
$$\forall C.\ \langle \mathsf{CSS}(css, C)\rangle\ \texttt{lookup(css, x)}\ \langle v.\ \mathsf{CSS}(css, C) \wedge C(x) = v\rangle$$

```
void insert(css *css, int x, void *v) {
  ... // initialize pn
  traverse(css, pn, x);
  insertHelper(css, pn->p, x, v);
  free(pn);
}
```

```
void *lookup(css *css, int x) {
  ... // initialize pn
  Status stt = traverse(css, pn, x);
  void *v = lookupHelper(css, pn->p, x, stt);
  free(pn);
  return v;
}
```

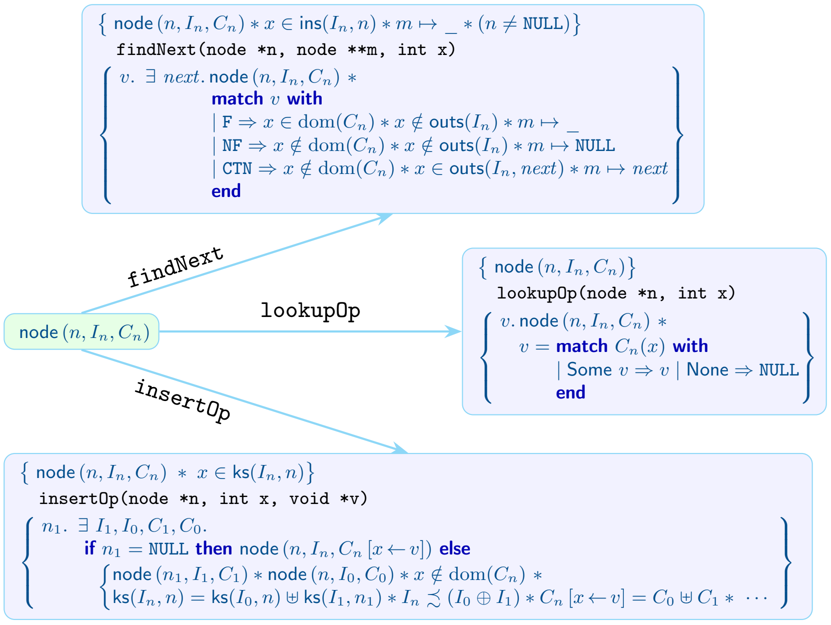
Fig. 5: Structure of the top-level operations

In this section, we present our specifications for the data structure and template functions. The structure of the specifications guarantees compositionality: if we have m data structures that satisfy the data structure interface, and n templates that satisfy the template interface, we can freely combine them to get $m \times n$ verified concurrent search structures. Each of our interfaces is implemented as a typeclass in Rocq, which makes it easy to do proofs over a generic instance of the interface.

A key feature of the separation between data structure and concurrency template is that data structure operations are purely *local*, taking effect on a single node and possibly its immediate children, while the template is entirely responsible for moving through the data structure. This means that the specifications for data structures must not talk about the state of the data structure as a whole, but only about local properties that can later be slotted into a global description at the template level. We follow Krishna et al. [3] in modeling this separation using *flow interfaces* [4] to characterize a node's contributions to the overall structure.

4.1 Data Structure Interface

Figure 6 illustrates the formal interface for data structure components, which perform sequential, local operations on pieces of a data structure. At the center is the node representation predicate $\mathsf{node}(n, I_n, C_n)$, where n is the concrete pointer to the node, I_n is the flow interface for the node, and C_n is the contents of the node (a map from keys to values). In practice, node will be implemented as a combination of concrete points-to predicates for the physical node in memory, and ghost state representing the node's contribution to the abstract state of the data structure. Surrounding it are the Hoare-style specifications of the three core operations, `findNext`, `lookupOp`, and `insertOp`, each of which acts on `node`s.



$\{\ \mathsf{node}\,(n, I_n, C_n) * x \in \mathsf{ins}(I_n, n) * m \mapsto \_ * (n \neq \mathtt{NULL})\}$

```
findNext(node *n, node **m, int x)
```

$$\left\{\begin{array}{l} v.\ \exists\, next.\, \mathsf{node}\,(n, I_n, C_n) * \\ \qquad \textbf{match } v \textbf{ with} \\ \qquad \mid \mathtt{F} \Rightarrow x \in \mathrm{dom}(C_n) * x \notin \mathsf{outs}(I_n) * m \mapsto \_ \\ \qquad \mid \mathtt{NF} \Rightarrow x \notin \mathrm{dom}(C_n) * x \notin \mathsf{outs}(I_n) * m \mapsto \mathtt{NULL} \\ \qquad \mid \mathtt{CTN} \Rightarrow x \notin \mathrm{dom}(C_n) * x \in \mathsf{outs}(I_n, next) * m \mapsto next \\ \qquad \textbf{end} \end{array}\right\}$$

$\{\ \mathsf{node}\,(n, I_n, C_n)\}$

```
lookupOp(node *n, int x)
```

$$\left\{\begin{array}{l} v.\, \mathsf{node}\,(n, I_n, C_n) * \\ \quad v = \textbf{match } C_n(x) \textbf{ with} \\ \qquad \mid \mathsf{Some}\ v \Rightarrow v \mid \mathsf{None} \Rightarrow \mathtt{NULL} \\ \qquad \textbf{end} \end{array}\right\}$$

$\{\ \mathsf{node}\,(n, I_n, C_n)\ *\ x \in \mathsf{ks}(I_n, n)\}$

```
insertOp(node *n, int x, void *v)
```

$$\left\{\begin{array}{l} n_1.\ \exists\, I_1, I_0, C_1, C_0. \\ \qquad \textbf{if } n_1 = \mathtt{NULL} \textbf{ then } \mathsf{node}\,(n, I_n, C_n\,[x \leftarrow v]) \textbf{ else} \\ \qquad\quad \left\{\begin{array}{l} \mathsf{node}\,(n_1, I_1, C_1) * \mathsf{node}\,(n, I_0, C_0) * x \notin \mathrm{dom}(C_n) * \\ \mathsf{ks}(I_n, n) = \mathsf{ks}(I_0, n) \uplus \mathsf{ks}(I_1, n_1) * I_n \precsim (I_0 \oplus I_1) * C_n\,[x \leftarrow v] = C_0 \uplus C_1 *\ \cdots \end{array}\right. \end{array}\right\}$$

Fig. 6: Data structure interface: predicates and specifications

The `findNext` function is used to traverse the data structure and find where a key belongs. It takes a node n and a key x and returns one of three possible results: *found* (`F`), *not found* (`NF`), or *continue* (`CTN`). `F` indicates that the key x is present in n. `NF` indicates that x is definitively not present in the data structure at n: it is not in n and is also guaranteed not to appear in n's successors. Finally, `CTN` means that x is not in n, but may be present in one of n's successors *next*, which is stored in the pointer m. We use flow interfaces to capture the idea that the key x "belongs" in n or its successors: x must initially be in the inset $\mathsf{ins}(I_n, n)$, and in the `CTN` case is also present in the outset $\mathsf{outs}(I_n, next)$ from n to *next*, while in the other cases it is not in any of n's outsets—thus implying that it is in the keyset $\mathsf{ks}(I_n, n)$ of n.

The `lookupOp` function simply looks up the key x in the node n, and returns its value if it is present, or `NULL` if it is not. Because it does not change the data structure, it does not need to interact with flow interfaces.

The `insertOp` function inserts the key x and value v at the given node n, which is valid when x belongs at n, i.e., it is in n's inset and not in any of its outsets. The function has two cases to consider. First, x may already be present in n; in this case, `insertOp` simply changes the value at x to v. Otherwise, it creates a new node n_1 containing x and v and adds it to the data structure, changing internal pointers and flow interfaces as necessary. The visible effect of this is to associate the original node n and the new node n_1 with new flow interfaces I_0, I_1 and key-value maps C_0, C_1, which between them contain both the original contents of n and the new mapping from x to v[2]. The keyset and flow of n in I_n are split across I_0, the new interface for n, and I_1, the interface for the newly created node n_1. The combined interface of the two nodes, $I_0 \oplus I_1$, must be a contextual extension of the original interface I_n (i.e., $I_n \precsim (I_0 \oplus I_1)$), ensuring that the remainder of the data structure is not affected by replacing n with the combination of n and n_1. The specific distribution of keys between n and n_1 is determined by the data structure; we might wish to allow it to be arbitrary, but we will see in Section 6.2 that some templates depend on specific constraints on this distribution, so we must include those constraints in our interface specification for `insertOp`. For the time being we write these constraints as $\cdots$, and explain them in detail in Section 6.2.

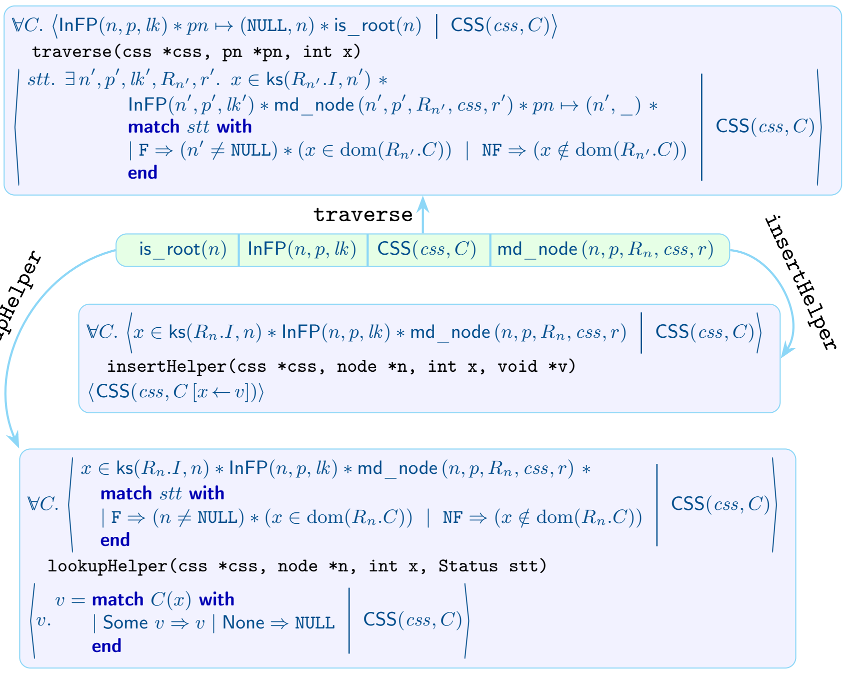
4.2 Template Interface

Figure 7 presents the concurrency template interface, which gives logically atomic specifications for three operations: `traverse`, `insertHelper`, and `lookupHelper`. The interface relies on four abstract predicates: $\mathsf{is\_root}(n)$, $\mathsf{InFP}(n, p, \mathit{lk})$, $\mathsf{CSS}(\mathit{css}, C)$, and $\mathsf{md\_node}\,(n, p, R_n, \mathit{css}, r)$. The predicate $\mathsf{is\_root}(n)$ identifies the root node of the structure. The predicate $\mathsf{InFP}(n, p, \mathit{lk})$ asserts that n belongs to the data structure, recording its metadata pointer p and lock lk. The predicate $\mathsf{CSS}(\mathit{css}, C)$ describes the abstract state of the entire concurrent data structure at pointer css as a key-value map C. Finally, $\mathsf{md\_node}\,(n, p, R_n, \mathit{css}, r)$ represents a node together with its concurrency metadata: n is the pointer to the node, p the pointer to its metadata, css the pointer to the top-level data structure, and r the root node of the data structure. At the template level, the abstract state of a node is a metadata record R_n containing a flow interface $R_n.I$ and a contents map $R_n.C$. To ensure modularity, all metadata (locks, ranges, etc.) must be stored in md_node rather than node, and templates must only interact with nodes via the data structure interface functions.

With the above predicates in place, we now turn to the specifications of the template operations. The `traverse` function is the core of a concurrency template: it takes a concurrent data structure css and a key x, and returns the node in the data structure where x belongs. Specifically, it uses a struct `pn` to track a pair of nodes: `typedef struct pn { struct node *p; struct node *n; } pn;` where the `n` field is initialized to the root of the data structure, and by the end of `traverse` the `p` field is set to a node n' such that $x \in \mathsf{ks}(I_{n'}, n')$. The function also returns an `enum` value that is either `F` if x is present in n' or `NF` if it is not. The specification of `traverse` is a logically atomic triple because it accesses the shared state of the data structure $\mathsf{CSS}(\mathit{css}, C)$, but it does not modify the data structure at all.

As shown in Figure 5, the top-level `insert` and `lookup` operations are realized by first invoking `traverse` to locate the appropriate node n', and then applying

[2] Our implementation also includes a special case for initializing an empty data structure, which we omit for readability here and in the following functions; its logic is in general a simpler form of that for the new-node case.



$\forall C.\ \langle \mathsf{InFP}(n,p,lk) * pn \mapsto (\mathtt{NULL}, n) * \mathsf{is\_root}(n) \mid \mathsf{CSS}(css, C)\rangle$

```
traverse(css *css, pn *pn, int x)
```

$\langle stt.\ \exists n', p', lk', R_{n'}, r'.\ x \in \mathsf{ks}(R_{n'}.I, n') * \mathsf{InFP}(n', p', lk') * \mathsf{md\_node}(n', p', R_{n'}, css, r') * pn \mapsto (n', \_) *$ **match** stt **with** $\mid \mathtt{F} \Rightarrow (n' \neq \mathtt{NULL}) * (x \in \mathrm{dom}(R_{n'}.C)) \mid \mathtt{NF} \Rightarrow (x \notin \mathrm{dom}(R_{n'}.C))$ **end** $\mid \mathsf{CSS}(css, C)\rangle$

$\mathsf{is\_root}(n)$ | $\mathsf{InFP}(n,p,lk)$ | $\mathsf{CSS}(css, C)$ | $\mathsf{md\_node}(n,p,R_n,css,r)$

$\forall C.\ \langle x \in \mathsf{ks}(R_n.I, n) * \mathsf{InFP}(n,p,lk) * \mathsf{md\_node}(n,p,R_n,css,r) \mid \mathsf{CSS}(css, C)\rangle$

```
insertHelper(css *css, node *n, int x, void *v)
```

$\langle \mathsf{CSS}(css, C[x \leftarrow v])\rangle$

$\forall C.\ \langle x \in \mathsf{ks}(R_n.I, n) * \mathsf{InFP}(n,p,lk) * \mathsf{md\_node}(n,p,R_n,css,r) *$ **match** stt **with** $\mid \mathtt{F} \Rightarrow (n \neq \mathtt{NULL}) * (x \in \mathrm{dom}(R_n.C)) \mid \mathtt{NF} \Rightarrow (x \notin \mathrm{dom}(R_n.C))$ **end** $\mid \mathsf{CSS}(css, C)\rangle$

```
lookupHelper(css *css, node *n, int x, Status stt)
```

$\langle v.\ v =$ **match** $C(x)$ **with** $\mid \mathsf{Some}\ v \Rightarrow v \mid \mathsf{None} \Rightarrow \mathtt{NULL}$ **end** $\mid \mathsf{CSS}(css, C)\rangle$

Fig. 7: Concurrency template interface: predicates and specifications

a helper function that performs the actual update (for insertion) or query (for lookup) at that node. The helper functions are responsible for modifying the underlying data structure (using `insertOp` or `lookupOp` from the data structure interface) and updating the metadata (creating new locks, modifying ranges, etc.) to create a consistent concurrent data structure for the new state. Both `insertHelper` and `lookupHelper` are specified with atomic triples that take the postcondition of `traverse` as their private precondition (although `insertHelper` ignores the `F`/`NF` distinction), and in their postcondition perform the desired operation. For `insertHelper` this means atomically adding the mapping of key x to value v to the data structure; for `lookupHelper` this means returning the value associated with x, if one exists.

4.3 Top-Level Operations

Given instances of the data structure and template interfaces, we can define and verify the top-level `insert` and `lookup` operations for concurrent search structures, shown in Figure 8.

Figure 8a shows the implementation of the `insert` function for the template algorithms. It first calls `traverse` to locate the node at which the key `x` should be inserted. Once the appropriate position is found, it invokes `insertHelper` to insert key `x` and value `v` at that node. The `lookup` function follows a similar procedure, except that it invokes `lookupHelper` to retrieve the value `v` associated with the key `x` and returns this value.

```
void insert(css *css, int x, void *v) {
  struct pn *pn = malloc(sizeof(*pn));
  pn->p = NULL;
  pn->n = get_root(css);
  Status stt = traverse(css, pn, x);
  insertHelper(css, pn->p, x, v);
  free(pn);
}
```

(a) The top-level `insert` function

```
void *lookup(css *css, int x) {
  struct pn *pn = malloc(sizeof(*pn));
  pn->p = NULL;
  pn->n = get_root(css);
  Status stt = traverse(css, pn, x);
  void *v = lookupHelper(css, pn->p, x, stt);
  free(pn);
  return v;
}
```

(b) The top-level `lookup` function

Fig. 8: Top-level concurrent data structure operations

$\langle \mathsf{CSS}(css, C) \rangle$

```
void insert (css *css, int x, void *v) {
   struct pn *pn = malloc(sizeof(*pn));
   pn->p = NULL;
   pn->n = get_root(css);
   {∃p, lk.  is_root(n) ∗ InFP(n, p, lk) ∗ pn ↦ (NULL, n)}
   Status status = traverse(css, pn, x);
   {∃n′, p′, lk′, R_n′, r′.  x ∈ ks(R_n′.I, n′) ∗ InFP(n′, p′, lk′) ∗ md_node (n′, p′, R_n′, css, r′) ∗ pn ↦ (n′, _)}
   insertHelper(css, pn->p, x, v); {pn ↦ (n′, _)}
   free(pn); {emp}
}
```

$\langle \mathsf{CSS}(css, C\,[x \leftarrow v]) \rangle$

Fig. 9: Proof outline of the top-level `insert` function

Figure 9 presents the proof outline for the top-level `insert` function. The proof follows directly from the specifications of `traverse` and `insertHelper`, since the postcondition of `traverse` matches the precondition of `insertHelper`. The call to `traverse` identifies the node n' where the key `x` should be inserted, acquires ownership of it via md_node, and stores it in `pn->p`. The call to `insertHelper` is the function's linearization point, atomically adding the mapping from key `x` to value `v` to the data structure. The proof for `lookup` is similar.

Importantly, these proofs rely only on the specifications of the template functions, and so are equally valid for any combination of data structure and template instance. In Rocq, the top-level proofs are parameterized by instances of DataStructure and Template typeclasses containing these specifications. This guarantees the compositionality we are aiming for: we know that for *any* data structure implementation meeting the data structure interface, and *any* template meeting the template interface, we can instantiate these parameterized proofs and obtain verified `insert` and `lookup` functions.

5 Data Structure Instances: Binary Search Tree and Linked List

An instance of the data structure interface consists of verified implementations of the `node` type and the `findNext`, `insertOp`, and `lookupOp` operations for the data structure. In this section, we demonstrate two such instances. As described in Section 4.1, we represent each `node` type with a predicate $\mathsf{node}\,(n, I_n, C_n)$ that connects the actual data in n to a flow interface I_n describing the keys that belong in n and its successors, and a key-value map C_n describing the keys and values present in n. We then prove that each function meets its specification in the interface.

5.1 Binary Search Tree

```
typedef struct node{ int key; void *value; node *left, *right; } node;
```

$$\mathsf{node}\,(n, I_n, C_n) :=$$

$$\begin{cases} I_n.\mathit{in} = \lambda_0 * I_n.\mathit{out} = \lambda_0 * C_n = \emptyset & \text{if } n = \texttt{NULL} \\ \exists\, x, v, \mathit{left}, \mathit{right}.\ n \mapsto (x, v, \mathit{left}, \mathit{right}) * \mathrm{dom}(I_n) = \{n\} * x \in \mathsf{ins}(I_n, n) * C_n = [x \leftarrow v] * & \text{if } n \neq \texttt{NULL} \\ \quad (I_n.\mathit{out} = (\mathit{left} = \texttt{NULL}\ ?\ \emptyset : [\mathit{left} \leftarrow \{\, y \in \mathsf{ins}(I_n, n) \mid y < x \,\}]) \cup & \\ \quad\quad (\mathit{right} = \texttt{NULL}\ ?\ \emptyset : [\mathit{right} \leftarrow \{\, y \in \mathsf{ins}(I_n, n) \mid y > x \,\}])\,) & \end{cases}$$

Fig. 10: The `node` definition for BST

The node implementation and corresponding predicate for a BST are shown in Figure 10. There is a special case for `NULL`, which represents a nonexistent node with no inflow or outflow (i.e., its flows are the zero function $\lambda_0 := (\lambda k.\ 0)$) and no contents. Otherwise, the `node` predicate for a node n includes physical ownership of the `node` struct ($n \mapsto (x, v, \mathit{left}, \mathit{right})$) along with information about the contents C_n and flow interface I_n of the node. The contents are simply $[x \leftarrow v]$, where x and v are the contents of the `key` and `value` field respectively—each BST node holds one key-value pair. The node's inset must include x, guaranteeing that traversal operations looking for x will reach n rather than going down another branch. Finally, the node's outsets must follow the BST structure: the left child (if it exists) receives all keys from n's inset that are less than x, and the right child (if it exists) receives all keys greater than x. These flow properties amount to a local characterization of a BST—if we know that each node in a data structure has these properties, then we are guaranteed that the data structure as a whole is indeed a BST.

```
Status findNext(node *n, node **m, int x) {
  if (x < n->key) {
    *m = n->left;
    if (!*m) return NF;
    else return CTN;
  }
  else if (x > n->key) {
    *m = n->right;
    if (!*m) return NF;
    else return CTN;
  }
  else return F;
}
```

(a) The `findNext` of the BST

```
node *insertOp(node *n, int x, void *v) {
  if (n->key == x) {
    n->value = v;
    return NULL;
  }
  node *nn = make_node(x, v);
  if (x < n->key) n->left = nn;
  else n->right = nn;
  return nn;
}
```

(b) The `insertOp` of the BST

Fig. 11: Operations of the data structure instances

The BST's `findNext` function (Figure 11a) determines whether the key `x` resides in the current node `n` or should be found in the left or right subtree.

As discussed in Section 4.1, the function returns one of three possible cases: `F` (found), `NF` (not found), or `CTN` (continue). For the BST, `x` is found if it is present in the node; otherwise, we compare it with `n`'s key and check the corresponding child, returning `NF` if the child is absent or `CTN` if it is present.

The BST's `insertOp` function (Figure 11b) handles two cases. First, if `n` already contains `x`, we simply set `n`'s value to `v`. Otherwise, we create a new node and insert it as either the left or right child of `n`, depending on whether `x` is less than or greater than `n`'s key. When we create a new node, we return a pointer to that node; when we update an existing node, we return `NULL` instead. The `lookupOp` function (not shown) simply returns the `value` field of the target node. Note that all of these functions are local to a single node; traversal will be left entirely to the concurrency template.

$$\{\mathsf{node}\,(n, I_n, C_n) \;*\; x \in \mathsf{ks}(I_n, n)\}$$

```
node *insertOp(node *n, int x, void *v) {
  //open node (n, I_n, C_n)
  {∃x_0, v_0, l, r. n ↦ (x_0, v_0, l, r) ∗ dom(I_n) = {n} ∗ x_0 ∈ ins(I_n, n) ∗ C_n = [x_0 ← v_0] ∗ ···}
  if (n->key == x) {
    n->value = v;
    {n ↦ (x, v, l, r) ∗ dom(I_n) = {n} ∗ x ∈ ins(I_n, n) ∗ C_n = [x ← v_0] ∗ ···} ⇛ {node (n, I_n, C_n [x ← v])}
    return NULL;
  }
  node *nn = make_node(x, v);
  {nn ↦ (x, v, NULL, NULL) ∗ n ↦ (x_0, v_0, l, r) ∗ ···}
  if (x < n->key) {
    n->left = nn;
    {(x < x_0) ∗ nn ↦ (x, v, NULL, NULL) ∗ n ↦ (x_0, v_0, nn, r) ∗ ···}
    //let (I_0, I_1) := send { x ∈ ins(I_n, n) | x < x_0 } from n to nn
    //close node (nn, I_1, [x ← v]), node (n, I_0, C_n)
  }
  else n->right = nn;
  return nn;
}
```

$$\left\{\begin{array}{l} n_1.\, \exists\, I_1, I_0, C_1, C_0. \\ \quad \textbf{if } n_1 = \mathtt{NULL} \textbf{ then } \mathsf{node}\,(n, I_n, C_n\,[x \leftarrow v]) \textbf{ else} \\ \quad \left\{\begin{array}{l} \mathsf{node}\,(n_1, I_1, C_1) * \mathsf{node}\,(n, I_0, C_0) * x \notin \mathrm{dom}(C_n) * \\ \mathsf{ks}(I_n, n) = \mathsf{ks}(I_0, n) \uplus \mathsf{ks}(I_1, n_1) * I_n \precsim (I_0 \oplus I_1) * C_n\,[x \leftarrow v] = C_0 \uplus C_1 \end{array}\right. \end{array}\right\}$$

Fig. 12: Proof outline of the BST `insertOp` function

Figure 12 illustrates the proof of the BST `insertOp` function. When the key x is already present in n, we unfold the predicate $\mathsf{node}\,(n, I_n, C_n)$ and replace its original value v_0 with v; this does not affect the flows and so the node predicate is restored with the new value in its contents. Otherwise, we add a new node nn with key x and value v as either the left or right child of n. In either case, the changes to values in memory are straightforward, but to establish the node predicates in the postcondition, we must also find new flow interfaces I_1, I_0 and content maps C_1, C_0 that capture the splitting of keys across n and nn. In a BST, as we saw in the definition of node, the flow from n to its left child must be precisely $\{\, x \in \mathsf{ins}(I_n, n) \mid x < x_0 \,\}$, the subset of n's inflow that is less than n's key x_0. Adding these keys to the outflow of n and the inflow of nn yields precisely the new flow interfaces that we need. Finally, we must show that this new configuration of flows $(I_0 \oplus I_1)$ contextually extends the original interface

I_n (i.e., $I_n \precsim (I_0 \oplus I_1)$); this is true because the inflow to n is unchanged and there are no new outflows to any previously existing nodes. This proof shows how flow interfaces abstract the details of the data structure into generic information about the distribution of keys, which can then be used by the template functions; the primary challenge in data-structure-side verification is then to come up with the right flow interfaces.

5.2 Linked List

```
typedef struct node{ int key; void *value; node *next; } node;
```

$$\mathsf{node}\,(n, I_n, C_n) :=$$

$$\begin{cases} I_n.in = \lambda_0 * I_n.out = \lambda_0 * C_n = \emptyset & \text{if } n = \texttt{NULL} \\ \exists\, x, v, next.\;\; n \mapsto (x, v, next) * \mathrm{dom}(I_n) = \{n\} * x \in \mathsf{ins}(I_n, n) * C_n = [x \leftarrow v] * & \text{if } n \neq \texttt{NULL} \\ \qquad\qquad I_n.out = (next = \texttt{NULL}\ ?\ \emptyset : [next \leftarrow \{\, y \in \mathsf{ins}(I_n, n) \mid y > x \,\}]) & \end{cases}$$

Fig. 13: The `node` definition for linked list

The second data structure we consider is a sorted linked list traversed via linear search. Its `node` predicate is similar to the BST's, but with a simpler outflow, with only one possible child that holds only larger keys. Its `findNext` function (Figure 14a) is likewise similar, but if we reach a node with a key greater than the target key `x`, we simply return `NF`: if we have passed the point where `x` would appear in the list, then it cannot be present in the list at all.

```
Status findNext(node *n, node **m, int x) {
  if (x > n->key) {
    *m = n->next;
    if (!*m)
     return NF;
    else
      return CTN;
  }
  else if (x < n->key)
    return NF;
  else
    return F;
}
```

(a) The `findNext` of the linked list

```
node *insertOp(node *n, int x, void *v) {
  if (n->key == x) {
    n->value = v;
    return NULL;
  }
  node *nn = make_node(x, v);
  if (x < n->key) {
    nn->next = n->next;
    n->next = nn;
    swap(n->key, nn->key);
    swap(n->value, nn->value);
  }
  else n->next = nn;
  return nn;
}
```

(b) The `insertOp` of the linked list

Fig. 14: Operations of the linked list instance

The `insertOp` function (Figure 14b) differs somewhat more from its BST analogue. The case for an already-existing key is the same, as is the case where `x > n->key`. However, when `x < n->key`, we need to insert the `x` *before* `n->key` to maintain sortedness. For convenience, we do this by inserting `nn` after `n` and then swapping their contents, so that the inflow to `n` remains unchanged. With this approach, the proof outline of the linked list's `insertOp` is very similar to that of the BST; the main effort is in showing that both insertion before and insertion after constitute contextual extensions of the flow interface.

6 Template Instances: Lock Coupling and Give-Up

As described in Section 4.2, a concurrency template implements a type for per-node metadata/synchronization, a `traverse` function for finding the node where a key "belongs", and functions `insertHelper` and `lookupHelper` that lift the sequential data structure operations to thread-safe concurrent operations. Our concurrent data structures are implemented as

```
typedef struct css { node *root; md_table metadata; } css;
```

where the `md_table` maps each `node*` pointer to an `md_entry`, a type defined differently by each template[3]. In both of our example templates, the per-node metadata includes a lock that protects the contents of the node. We write $\mathsf{inv\_for\_lock}\,(\ell, R)$ to indicate that lock ℓ is associated with a predicate R, called the lock invariant—this means that any thread acquiring ℓ gains resources satisfying R, and must restore R upon release.

6.1 Lock-coupling Template

The first template we consider is lock coupling (also called hand-over-hand locking), in which threads use the locks on each node to prevent interference from other threads during traversal. Each thread always holds at least one lock, and acquires the lock on the next node before releasing its current lock, ensuring that other threads cannot invalidate the ongoing search.

The lock-coupling template's metadata for each node is simply its lock:

```
typedef struct md_entry { lock_t lock; } md_entry;
```

The lock-coupling pattern can be seen on lines 15-16 of Figure 15a, where `traverse` acquires the next node's lock and then releases the current node's lock. The rest of the function simply repeatedly calls the `findNext` function from the data structure interface, which returns either found (`F`), not found (`NF`), or continue (`CTN`); in the first two cases `traverse` returns this status, while on `CTN` it continues to traverse the data structure.

Figure 15c shows the implementation of `insertHelper` for the lock-coupling template. The function is intended to be called after `traverse`, which identifies a suitable node `n` at which to insert the key `x` and acquires the lock on `n`. The function begins by calling `insertOp` to perform the insertion in the data structure at node `n`. If `insertOp` returns `NULL` (indicating that the key `x` was already present in `n`), `insertHelper` then immediately releases the lock on `n` and returns. Otherwise, a new node has been added, so `insertHelper` needs to create metadata for the new node by allocating a new `md_entry` and then initializing its `lock` field with a new lock `new_lock` (`makelock()` returns a new lock in the held state). Finally, we release both the new lock and the lock on `n`.

[3] The `md_table` type can be implemented with any kind of map, and does not need to be thread-safe, since each entry is initialized when the corresponding node is allocated and thereafter treated as read-only. For simplicity, our implementation uses a naïve hash table and assumes the absence of collisions, since writing a collision-free hash table is orthogonal to our contributions.

```
Status traverse(css *css, pn *pn, int x) {
  Status stt = NF;
  md_entry *md_n = lookup_md(css, pn->n);
  md_entry *md_p;
  acquire(md_n->lock);
  for ( ; ; ) {
    pn->p = pn->n;
    stt = findNext(pn->p, &pn->n, x);
    if (stt == F) break;
    else if (stt == NF) break;
    else {
      md_n = lookup_md(css, pn->n);
      md_p = lookup_md(css, pn->p);
      acquire(md_n->lock);
      release(md_p->lock);
    }
  }
  return stt;
}
```

(a) The `traverse` method of the lock-coupling template algorithm

```
Status traverse(css *css, pn *pn, int x) {
  Status stt = NF;
  node *p = pn->n;
  for ( ; ; ) {
    md_entry *md = lookup_md(css, pn->n);
    acquire(md->lock);
    pn->p = pn->n;
    if (inRange(md, x)) {
      stt = findNext(pn->p, &pn->n, x);
      if (stt == F) break;
      else if (stt == NF) break;
      else release(md->lock);
    }
    else {
      release(md->lock);
      pn->n = p;
    }
  }
  return stt;
}
```

(b) The `traverse` method of the give-up template algorithm

```
void insertHelper(css *css, node *n,
                  int x, void *v) {
  node *new_node = insertOp(n, x, v);
  md_entry *md = lookup_md(css, n);
  lock_t parent_lock = md->lock;
  if (!new_node) {
    release(parent_lock);
    return;
  }
  md_entry *new_md = malloc(sizeof(md_entry));
  lock_t new_lock = makelock();
  new_md->lock = new_lock;

  set_md(css, n, new_md);
  release(new_lock);
  release(parent_lock);
}
```

(c) The `insertHelper` method of the lock-coupling template algorithm

```
int inRange(md_entry *m, int x) {
  if (x > m->min && x < m->max) return 1;
  else return 0;
}
```

(d) The `inRange` method of the give-up template algorithm

```
void *lookupHelper(css *css, node *n,
                   int x, Status stt) {
  void *v;
  md_entry* md = lookup_md(css, n);
  lock_t parent_lock = md->lock;
  if(stt == F) v = lookupOp(n, x);
  else v = NULL;
  release(parent_lock);
  return v;
}
```

(e) The `lookupHelper` method for both template algorithms

Fig. 15: Operations of the concurrency templates

The `lookupHelper` function for the lock-coupling template is shown in Figure 15e. Its behavior depends on the status returned from `traverse`: if the status is `F` then we get the value in node `n` using `lookupOp`, while if it is `NF` the value is `NULL` (since the target key `x` was not found in the data structure). Either way, the function then releases the lock and returns the value found.

While the proofs of data structure instances only involve real memory and logical flow interfaces, the proofs of concurrency templates make extensive use of ghost state to track the relationship between individual nodes and the abstract data structure as a whole. Our representation predicate CSS, shown in Figure 16, largely follows that of Krishna et al. [3], but with metadata explicitly stored in a hash table h, and with some adaptations to our more realistic setting (we include

$$h : \; val \to val \qquad N : \; val \rightharpoonup (val \times val) \qquad R_n := \; (I_n, C_n)$$

$$\begin{aligned}
\varphi(r, I) &:= I.in\,(r) = \mathsf{KS} * I.out = \lambda_0 \\
\mathsf{InFP}(n, p, lk) &:= \exists N.\, N(n) = (p, lk) * \boxed{\circ\, N}^{\gamma_f} \\
\mathsf{own\_nodes}\,(\gamma_f, I, h) &:= \exists N.\, \mathrm{dom}(N) = \mathrm{dom}(I) * \boxed{\bullet\, N}^{\gamma_f} * \\
&\quad\; (\forall n, p, lk.\, N(n) = (p, lk) \Rightarrow h(n) = p) \\
\mathsf{md\_node}\,(n, p, R_n, css, r) &:= \mathsf{node}\,(n, I_n, C_n) * \boxed{\circ\, I_n}^{\gamma_I} * \boxed{\circ\, (\mathsf{ks}(I_n, n), \mathrm{dom}(C_n))}^{\gamma_k} * \boxed{\circ\, (\mathrm{Ex}\; C_n)}^{\gamma_m} \\
\mathsf{CSS}(css, C) &:= \exists I, h, r.\; \boxed{\bullet\, I}^{\gamma_I} * \boxed{\bullet\, (\mathsf{KS}, \mathrm{dom}(C))}^{\gamma_k} * \boxed{\bullet\, (\mathrm{Ex}\; C)}^{\gamma_m} * \\
&\quad\; \mathsf{own\_nodes}\,(\gamma_f, I, h) * \varphi(r, I) * \mathsf{hashtable}(h) * \\
&\quad\; \mathop{\Large *}_{n \in \mathrm{dom}(I)} \left(\begin{aligned} &\exists p, lk, R_n.\, \mathsf{InFP}(n, p, lk) * p.\mathtt{lock} \mapsto_{\Box} lk * \\ &\qquad\qquad \mathsf{inv\_for\_lock}\,(lk, \mathsf{md\_node}\,(n, p, R_n, css, r)) \end{aligned} \right)
\end{aligned}$$

Fig. 16: Definition of CSS and related predicates in the lock-coupling proof

values in nodes as well as keys, for instance, and our data is explicitly laid out in C structs). We make heavy use of the *authoritative* pattern of ghost state, where an element $\boxed{\bullet\, a}^{\gamma}$ characterizes the overall abstract state, and individual contributors hold *fragments* $\boxed{\circ\, b_n}^{\gamma}$ with the restriction that the combination of all fragments yields exactly the overall state. The specific kinds of ghost state we use are as follows:

- The node ghost state γ_f maps nodes in the data structure to their corresponding metadata and lock pointers. We use this to construct the membership predicate InFP, which asserts that a node n belongs to the data structure, and the own_nodes predicate, which holds the authoritative map of all nodes in the data structure and their corresponding locks and metadata pointers.
- The flow interface ghost state γ_I collects the flow interfaces for every node; the top-level interface I is the composition of all per-node flows I_n.
- The keyset ghost state γ_k holds pairs (K, k), where K is the set of keys that "belong" in a node (the keyset ks for an individual node, the full set of possible keys KS for the data structure as a whole) and k is the set of keys actually present in the data structure. This is an important distinction—if, e.g., the key 3 is not present in a data structure, then it will not be in the mapping C from keys to values, but it will still be present in both the top-level keyset KS and the keyset of some specific node where 3 could be inserted. The keysets of the individual nodes partition the space of possible keys, and the actual contents C are the combination of all the per-node contents C_n.
- The mapping ghost state γ_m holds the key-value mapping for each node and for the data structure as a whole. This is separate from the keyset ghost state because the keysets mention only keys, not values.

All of this ghost state is used in the definition of md_node (n, p, R_n, css, r), which connects a node with all its per-node ghost state. The predicate for the data structure as a whole, CSS, collects four pieces:

- The authoritative parts of each kind of ghost state, describing the top-level abstract state of the data structure.
- Global properties on the data structure: for instance, $\varphi(r, I)$ ensures that the inflow at the root node r is the full keyset KS and that no flow leaves the

$$\forall C.\ \langle \mathsf{InFP}(n, p, lk) * pn \mapsto (\texttt{NULL}, n) * \texttt{is\_root}(n)\ \mid\ \mathsf{CSS}(css, C)\rangle$$

```
1  int traverse(css *css, pn *pn, int x) {
2     Status stt = NF;
3     md_entry *md_n = lookup_md(css, pn->n);
4     md_entry *md_p;
5     acquire(md_n->lock);
6     $\{x \in \mathsf{ins}(R_n.I, n) * \mathsf{md\_node}\,(n, p, R_n, css, r) * \cdots\} \Rrightarrow \{\mathsf{traverse\_inv}\}$
7     for ( ; ; ) {           $\{\mathsf{traverse\_inv}\}$
8        pn->p = pn->n;
9        $\{x \in \mathsf{ins}(R_{n'}.I, n') * \mathsf{md\_node}\,(n', p', R_{n'}, css, r') * pn \mapsto (n', n') * \cdots\}$
10       stt = findNext(pn->p, &pn->n, x);
11       if (stt == F) break;
12       $\{x \in \mathsf{ks}(R_{n'}.I, n') * x \in \mathrm{dom}(R_{n'}.C) * \mathsf{md\_node}\,(n', p', R_{n'}, css, r') * pn \mapsto (n', n') * \cdots\}$
13       else if (stt == NF) break;
14       $\{x \in \mathsf{ks}(R_{n'}.I, n') * x \notin \mathrm{dom}(R_{n'}.C) * \mathsf{md\_node}\,(n', p', R_{n'}, css, r') * pn \mapsto (n', m) * \cdots\}$
15       else {
16       $\{x \in \mathsf{outs}(R_{n'}.I, m) * x \notin \mathrm{dom}(R_{n'}.C) * \mathsf{md\_node}\,(n', p', R_{n'}, css, r') * pn \mapsto (n', m) * \cdots\}$
17          md_n = lookup_md(css, pn->n);
18          md_p = lookup_md(css, pn->p);
19          acquire(md_n->lock);
20          $\{x \in \mathsf{ins}(R_m.I, m) * \mathsf{md\_node}\,(m, q, R_m, css, r_q) * \mathsf{md\_node}\,(n', p', R_{n'}, css, r') * \cdots\}$
21          release(md_p->lock);
22          $\{x \in \mathsf{ins}(R_m.I, m) * \mathsf{md\_node}\,(m, q, R_m, css, r_q) * \cdots\} \Rrightarrow \{\mathsf{traverse\_inv}\}$
23       }
24    }
25    return stt;
26 }
```

$$\left\langle \begin{array}{l} stt.\ \exists\, n', p', lk', R_{n'}, r'.\ x \in \mathsf{ks}(R_{n'}.I, n') * \\ \quad \mathsf{InFP}(n', p', lk') * \mathsf{md\_node}\,(n', p', R_{n'}, css, r') * pn \mapsto (n', \_) * \\ \quad \textbf{match}\ stt\ \textbf{with} \\ \quad \mid \texttt{F} \Rightarrow (n' \neq \texttt{NULL}) * (x \in \mathrm{dom}(R_{n'}.C))\ \mid\ \texttt{NF} \Rightarrow (x \notin \mathrm{dom}(R_{n'}.C)) \\ \quad \textbf{end} \end{array} \ \middle|\ \mathsf{CSS}(css, C) \right\rangle$$

Fig. 17: Proof outline of the lock-coupling `traverse` function

data structure, and own_nodes requires that each node's metadata pointer p is exactly the pointer associated with it in the hash table h.

- The representation predicate of the hash table h itself.
- A per-node assertion for each node in the top-level flow interface I. Specifically, each node's metadata must contain a lock lk that protects the md_node for that node. We use a persistent points-to assertion $p.\texttt{lock} \mapsto_{\square} lk$ to indicate that the lock is shared among all threads.

Taken together, these pieces connect the abstract key-value map C with a concrete pointer css, forming the top-level representation predicate that will be used in the specifications of the template functions.

The proof of the `traverse` function is outlined in Figure 17. The goal of the function is to find and acquire the lock on a node n' such that $x \in \mathsf{ks}(R_{n'}.I, n')$ (i.e., n' is the node where x belongs), returning `F` if x is present in n' and `NF` if it is not. We begin with the root node n in `pn->n`. First, we establish $x \in \mathsf{ins}(R_n.I, n)$, which is the key property we will maintain throughout `traverse`: we are in the right part of the data structure to find the key x. Since n is the root, this follows directly from the global property $\varphi(n, I)$ of CSS. We then acquire n's lock and obtain md_node for the current node `pn->n` (line 6). Now, we begin the main loop. The loop's invariant includes both ownership of the md_node assertion for `pn->n` and the property that x is in the inset of n:

```
        $\forall C.\ \langle x \in \mathsf{ks}(R_n.I, n) * \mathsf{InFP}(n, p, lk) * \mathsf{md\_node}\,(n, p, R_n, css, r) \mid \mathsf{CSS}(css, C)\rangle$
1  void insertHelper(css *css, node *n, int x, void *v) {
2    $\{\exists I_n, C_n.\, (I_n = R_n.I \land C_n = R_n.C) * \mathsf{node}\,(n, I_n, C_n) * \boxed{\circ\, I_n}^{\gamma_I} * \boxed{\circ\, (\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \boxed{\circ\, (\mathsf{Ex}\ C_n)}^{\gamma_m} * \cdots\}$
3    node *nn = insertOp(n, x, v);
4    $\{(\exists\, n_1, I_1, I_0, C_1, C_0.\ \textbf{if}\ n_1 = \mathsf{NULL}\ \textbf{then}\ \mathsf{node}\,(n, I_n, C_n\,[x \leftarrow v])\ \textbf{else}$
         $\mathsf{node}\,(n_1, I_1, C_1) * \mathsf{node}\,(n, I_0, C_0) * x \notin \mathrm{dom}(C_n) * I_n \precsim (I_0 \oplus I_1)) *$
      $\boxed{\circ\, I_n}^{\gamma_I} * \boxed{\circ\, (\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \boxed{\circ\, (\mathsf{Ex}\ C_n)}^{\gamma_m} * \cdots\}$
5    md_entry *md = lookup_md(css, n);
6    lock_t parent_lock = md->lock;
7    if (!nn) { release(parent_lock); return; }
8    md_entry *new_md = malloc(sizeof(md_entry));
9    lock_t new_lock = makelock();
10   new_md->lock = new_lock;
11   set_md(css, n, new_md);
12   //Linearization point, open $\mathsf{CSS}(css, C)$
13   $\{\mathsf{node}\,(nn, I_1, C_1) * \mathsf{node}\,(n, I_0, C_0) * \boxed{\bullet\, I}^{\gamma_I} * \boxed{\circ\, I_n}^{\gamma_I} * \boxed{\bullet\, (\mathsf{Ex}\ C)}^{\gamma_m} * \boxed{\circ\, (\mathsf{Ex}\ C_n)}^{\gamma_m} *$
       $\boxed{\bullet\, (\mathsf{KS}, \mathrm{dom}(C))}^{\gamma_k} * \boxed{\circ\, (\mathsf{ks}(I_n, n), \mathrm{dom}(C_n))}^{\gamma_k} * \cdots\} \Rrightarrow$
14   $\{\exists I'.\, \mathsf{node}\,(nn, I_1, C_1) * \mathsf{node}\,(n, I_0, C_0) * \boxed{\bullet\, I'}^{\gamma_I} * \boxed{\circ\, I_0}^{\gamma_I} * \boxed{\circ\, I_1}^{\gamma_I} *$
         $\boxed{\bullet\, (\mathsf{Ex}\ (C\,[x \leftarrow v]))}^{\gamma_m} * \boxed{\circ\, (\mathsf{Ex}\ C_n)}^{\gamma_m} * \boxed{\circ\, (\mathsf{Ex}\ ([x \leftarrow v]))}^{\gamma_m} *$
         $\boxed{\bullet\, (\mathsf{KS}, \mathrm{dom}(C\,[x \leftarrow v]))}^{\gamma_k} * \boxed{\circ\, (\mathsf{ks}(I_0, n), \mathrm{dom}(C_n))}^{\gamma_k} * \boxed{\circ\, (\mathsf{ks}(I_1, nn), \{x\})}^{\gamma_k} * \cdots\}$
15   //Close $\mathsf{CSS}(css, C\,[x \leftarrow v])$, prove postcondition
16   $\{\mathsf{md\_node}\,(n, p, R_{np}, css, r') * \mathsf{md\_node}\,(nn, nnp, R_{nn}, css, r')\}$
17   release(new_lock);
18   release(parent_lock); $\{\mathsf{emp}\}$
19 }
                          $\langle\mathsf{CSS}(css, C\,[x \leftarrow v])\rangle$
```

Fig. 18: Proof outline of the lock-coupling `insertHelper` function

$$\begin{aligned}&\mathsf{traverse\_inv}(css, pn, x) := \exists\, n, m, p, lk, R_n, r.\\ &\quad x \in \mathsf{ins}(R_n.I, n) * \mathsf{InFP}(n, p, lk) * \mathsf{md\_node}\,(n, p, R_n, css, r) * pn \mapsto (m, n) * p.\mathtt{lock} \mapsto_{\square} lk\end{aligned}$$

In each iteration, we update `pn->p` to the current `pn->n` (denoted n') and call `findNext` to locate the next node to traverse. If `findNext` returns `F` or `NF`, the traversal terminates, and we learn that x is either present in n' or not in the data structure at all; in either case, we know $x \notin \mathsf{outs}(R_{n'}.I)$, which combined with $x \in \mathsf{ins}(R_{n'}.I, n')$ yields $x \in \mathsf{ks}(R_{n'}.I, n')$—there is no successor of n' where we should look for x, so x belongs in n'. Otherwise, if `findNext` returns `CTN`, we learn that $x \in \mathsf{outs}(R_{n'}.I, m)$ for some successor m stored in `pn->n`. We then proceed to m by lock coupling. First, we acquire the lock on `pn->n` and obtain the $\mathsf{md\_node}$ for m. Since m is a successor of n', the outflow from n' is received as inflow to m, so $x \in \mathsf{ins}(R_m.I, m)$, as expressed in line 20. We then release the lock on the old node `pn->p`, relinquishing ownership of it while retaining ownership of the successor m, thereby completing the lock coupling (line 22). This lock coupling has ensured that the link between the nodes remains intact while we traverse it, and we can reestablish $\mathsf{traverse\_inv}$ for the new values in `pn`.

The proof of `insertHelper` is outlined in Figure 18. Given the $\mathsf{md\_node}$ predicate for a node n such that $x \in \mathsf{ks}(R_n.I, n)$, it atomically inserts the mapping from x to v into the data structure. We begin by calling the data structure's `insertOp`, whose postcondition is shown in line 4. This either modifies the existing value in n (when n's key is x), or inserts a new node nn with key x and value v. In the former case, nn is `NULL`, and all we need to do is update the mapping ghost state γ_m with the new key-value binding and release the lock on n. Otherwise, we need to allocate new metadata for nn and update the ghost state to reflect that the extended structure implements $C\,[x \leftarrow v]$, where C is the abstract state of the data structure immediately before the linearization point. At the linearization point, we open $\mathsf{CSS}(css, C)$ and perform frame-preserving updates on each pair of authoritative and fragment ghost state, shown in the proof annotations from line 13 to line 14: (i) interface update (γ_I)—$R_n.I$ is split into I_0 and I_1 with $R_n.I \precsim I_0 \oplus I_1$, reflecting the division of flow between the original node n and the new node nn; (ii) keyset update (γ_k)—the logical keyset $\mathsf{ks}(R_n.I, n)$ is split into $\mathsf{ks}(I_0, n)$ and $\mathsf{ks}(I_1, nn)$, while x is added to the actual keys present in nn and the global keyset; and (iii) map update (γ_m)—the binding $[x \leftarrow v]$ is added to the local contents of nn and the global map, extending C to $C\,[x \leftarrow v]$. Together these updates yield $\mathsf{CSS}(css, C\,[x \leftarrow v])$, establishing the atomic postcondition of `insertHelper`, and leaving us with updated `md_node` predicates for both n (with metadata $R_{np} := (I_0, C_n)$) and nn (with $R_{nn} := (I_1, [x \leftarrow v])$) at line 16. Finally, we release both locks and relinquish ownership of the nodes, completing the proof.

6.2 Give-up Template

We next consider the give-up template, which uses an optimistic concurrency control approach, acquiring fewer locks at the cost of sometimes having to recover from synchronization errors. Unlike the lock-coupling template, which maintains locks during traversal between nodes, the give-up template only acquires a lock just before operating on a node, and holds at most one lock at any time. This means that a conflicting operation may invalidate a traversal, for instance by moving the next node to another part of the data structure before we acquire its lock. To guard against this, the `traverse` function (Figure 15b) must explicitly check whether the target key is in the range of the current node using an `inRange` function (line 9). If a check fails, we give up and start the traversal over from the root node. The give-up template performs well in scenarios where operations generally do not conflict, either because they are on independent parts of the data structure or because they do not delete or relocate nodes.

To support the `inRange` function, the give-up template's metadata for each node includes a range:

```
typedef struct md_entry { lock_t lock; int min; int max; } md_entry;
```

As we will see, this means that the template can be applied to any data structure, but also gives the template the responsibility to maintain this metadata. Figure 15b shows the give-up template's implementation of `traverse`. Its main loop is similar to that of the lock-coupling template, but in each iteration, it calls `inRange` to check whether `x` is in the range of keys held in node `pn->n` and its successors. If `x` is outside the node's range (e.g., because the node has been relocated), the search is restarted from the root node `p`.

```
int key = getKey(n);
if (x < key) {
  new_md->min = md->min;
  new_md->max = key;
} else {
  new_md->min = key;
  new_md->max = md->max;
}
```

The `insertHelper` function, by contrast, is almost identical to the lock-coupling `insertHelper`, but it must also update the `min` and `max` metadata before releasing the locks (see the code snippet on the left).

This follows the logic of BST insertion: if the parent node `n` has range (a, b) and key k, then a node inserted on the left has range (a, k), and a node inserted on the right has range (k, b). As it happens, this logic also works for the sorted linked list: we always insert the new node after `n`, and the range of the new node is all keys greater than k (i.e., the `max` field of every node is always `INT_MAX`). As such, we use this code in the give-up `insertHelper` function, and place related conditions in the postcondition of `insertOp` (alluded to in Section 4.1): if the inserted key x is less than n's key k then the inset of the new node must be $\{a \in \mathsf{ins}(I_n, n) \mid a < k\}$, and if $x > k$ then the new node's inset must be $\{a \in \mathsf{ins}(I_n, n) \mid a > k\}$. This approach works well for our two example data structures, which are both fundamentally *ordered*, but a data structure where nodes are not globally ordered (but still have key ranges) would not necessarily satisfy this condition and thus would not fit into our data structure interface. Thus, there may be an expressiveness tradeoff between allowing more templates and allowing more data structures; studying more template instances will help clarify whether this is a general problem or one specific to the give-up template.

$$
\begin{aligned}
R_n &:= (I_n, C_n, min_n, max_n)\\
\mathsf{md\_node}\,(n, p, R_n, css, r) &:= \mathsf{node}\,(n, I_n, C_n) * \boxed{\circ\, I_n}^{\gamma_I} * \boxed{\circ\,(\mathsf{ks}(I_n, n), \mathrm{dom}(C_n))}^{\gamma_k} * \boxed{\circ\,(\mathrm{Ex}\ C_n)}^{\gamma_m} *\\
&\quad (p \neq \mathtt{NULL} \Rightarrow (\forall k.\ min_n < k < max_n \Rightarrow k \in \mathsf{ins}(I_n, n))) *\\
&\quad p.\mathsf{min} \mapsto min_n * p.\mathsf{max} \mapsto max_n
\end{aligned}
$$

Fig. 19: The definition of `md_node` for the give-up template

As shown in Figure 19, the `md_node` predicate for the give-up template must account for the additional metadata fields `min` and `max`. Each metadata record R_n includes this extra information and is defined as $R_n := (I_n, C_n, min_n, max_n)$. We ensure that these ranges are informative by requiring that all keys between min_n and max_n are in $\mathsf{ins}(I_n, n)$, i.e., the range stored in memory is an underapproximation of the logical inset of n. Thus, whenever a call to `inRange` succeeds, we have established that the target key is in the inset of n and we are in the correct section of the data structure. Otherwise, the representation predicate CSS is defined identically to the lock-coupling case.

The proofs of the give-up template functions are broadly similar to the lock-coupling proofs. In `traverse`, we do not hold any locks between iterations of the loop body, and we do not maintain the invariant that x is in the inset of the current node; this is instead checked explicitly using the `inRange` function. In the case where `inRange` fails, we can then reestablish the loop invariant at the root node. In `insertHelper`, we must show that the *min* and *max* values for the new node underapproximate its inset, using the additional guarantees from `insertOp` described above.

7 Implementation and Evaluation

The code for our concurrent data structures is written in C, and our specifications and proofs are written in the Verified Software Toolchain (VST) [1]. Using C forces us to confront the implementation challenges described in Section 3, which we might accidentally circumvent in a core calculus. VST allows us to prove separation logic specifications of C programs in Rocq, and newer versions of VST [5] are built on Iris [2], allowing us to directly reuse the implementation of flow-interface ghost state from Krishna et al. [3]. In addition to the data structures and templates described above, we prove a simple coarse-grained locking template, where a single lock protects the entire data structure. Note that there is no file in either C or Rocq that is specific to a combination of data structure and template, such as BST+give-up or list+coupling: we can freely combine any data structure and concurrency template and obtain a working concurrent data structure with no further effort, as desired.

Table 1: Lines of code and proof per module

Category	**Module**	**Code**	**Proof**	**Total**
Data Structures	BST	43	805	848
	Linked List	48	772	820
	Helper Proofs	0	392	392
Coarse-grained	Traverse	25	437	462
	Insert Helper	13	716	729
	Lookup Helper	11	365	376
	Helper Proofs	0	1273	1273
Lock-coupling	Traverse	38	505	543
	Insert Helper	25	960	985
	Lookup Helper	13	432	445
	Helper Proofs	0	1915	1915
Give-up	Traverse	41	480	521
	Insert Helper	38	1211	1249
	Lookup Helper	13	432	445
	InRange	6	31	37
	Helper Proofs	0	1845	1845
Top-Level	Insert	8	162	170
	Lookup	10	148	158
	Total	**332**	**12881**	**13213**

We have not tried to make our proofs concise, so LoC numbers may not be especially meaningful, but we do observe that the data structure proofs are about half the size of the template proofs. The complexity of the data structure proofs comes mostly from reasoning about flow interfaces, while the complexity in template proofs comes mostly from ghost state and logical atomicity reasoning. The former could likely be simplified by proving more general lemmas about flow interfaces (for instance, the operation of inserting a new node into a flow interface has roughly the same structure in both the BST and the linked list).

We have only implemented three templates and two data structures so far, but our results already illustrate some of the benefits of having clear interfaces for data structures and templates. In prior work, Nguyen et al. [6] verified BST instantiations of the lock-coupling and give-up templates; they also used VST, so we can reasonably compare proof sizes. Our verification of the data structure interface functions is much larger than theirs—800 lines compared to 250—because we must do complicated flow-interface reasoning to meet the interface's postcondition instead of returning an explicit description of the modified data structure. In contrast, our template proofs are roughly the same size. Furthermore, our proofs are much more reusable: Nguyen et al. state and prove different specifications for the BST operations for each template, and their template proofs are deeply entangled with the BST structure, while we have only one version of the BST proofs and our templates freely apply to the linked-list data structure as well. Of course, the benefits of modularity will increase further as more data structure and template components are verified using our framework.

We cannot directly compare our proof effort to the work of Krishna et al. [3] due to the difference in languages and tools, but we can make some observations about the tradeoffs involved in modularity. For instance, each of their templates relies on a different specification for `insertOp`; for lock coupling they use a specification similar to ours, while for give-up they use one where keys are always added to an existing node rather than creating a new one. This works because they only instantiate the give-up template with a hash table and a B+-tree, both of which can store multiple keys in a single node. On the data structure side, we could imagine defining simpler specifications for common data structure patterns like multi-key nodes, and proving that these simpler specifications imply our generic data structure interface, making it easier to instantiate the interface with data structures that fit that pattern. On the template side, relying on the simpler `insertOp` yields simpler proofs, but those proofs simply do not apply to data structures like the BST and linked list where insertion requires creating a new node—the higher effort required to verify the template against the more complicated specification leads directly to a more generally applicable template. Thus, with suitable infrastructural support for common cases, our framework should make proofs harder than less-modular approaches only when that difficulty is required for compositionality.

8 Conclusion and Future Work

We have demonstrated a highly compositional formalization of the concurrency template approach: each data structure instance is implemented and verified without any awareness of concurrency or metadata, each template instance is implemented and verified without any knowledge of the data layout, and we obtain verified concurrent data structures immediately by instantiating our top-level theorems with any combination of data structure and template instances. We accomplished this by carefully separating both implementations and specifications into data structure and template components, with flow interfaces as the abstraction between the two layers; flow-interface-based specifications for data structures provide enough information for templates to manage synchronization metadata, without overly restricting data structure implementations. Our next goal is to implement more data structures and, especially, more templates as instances of our interfaces, including lock-free concurrency patterns (e.g., optimistic concurrency control with fine-grained atomics), with the ultimate aim of decomposing and verifying real-world concurrent search structure implementations. Implementing more instances will also help answer the question of whether we should expect all templates to be compatible with all data structures, or whether templates should be allowed to place extra requirements on compatible data structures. We are also interested in tackling the problem of maintenance operations, which prior work [6] has identified as not clearly decomposable into concurrency and data-structure components; it should still be possible to prove correctness of these operations once per combination of data structure and template, and then combine them with compositionally verified data structure and template components to yield, e.g., a verified lock-coupling BST with rotation-based deletion.

Data Availability Statement

The artifact accompanying this paper is available online [7]. The latest version of the development is maintained on GitHub at https://github.com/UIC-verif-group/concurrency-templates.

Acknowledgements. We thank Nisarg Patel for detailed discussions on the original search structure template proofs, and the anonymous reviewers for their comprehensive and insightful feedback. This work was partially supported by the National Science Foundation under Award No. 2414582 (SHF: Small: *End-to-End Formal Verification of Concurrent C Programs*).

References

1. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press (2014)
2. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 637–650. ACM (2015). https://doi.org/10.1145/2676726.2676980
3. Krishna, S., Patel, N., Shasha, D., Wies, T.: Verifying Concurrent Search Structure Templates. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 181–196. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386029
4. Krishna, S., Shasha, D., Wies, T.: Go with the Flow: Compositional Abstractions for Concurrent Data Structures. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158125
5. Mansky, W., Du, K.: An Iris Instance for Verifying CompCert C Programs. Proc. ACM Program. Lang. **8**(POPL) (Jan 2024). https://doi.org/10.1145/3632848
6. Nguyen, D.T., Beringer, L., Mansky, W., Wang, S.: Compositional Verification of Concurrent C Programs with Search Structure Templates. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 60–74. CPP 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3636501.3636940
7. Nguyen, D.T., Mansky, W.: Artifact for "A Formal Interface for Concurrent Search Structure Templates" (2026). https://doi.org/10.5281/zenodo.18319035
8. Patel, N., Krishna, S., Shasha, D., Wies, T.: Verifying Concurrent Multicopy Search Structures. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). https://doi.org/10.1145/3485490
9. Patel, N., Shasha, D., Wies, T.: Verifying Lock-Free Search Structure Templates. In: Aldrich, J., Salvaneschi, G. (eds.) 38th European Conference on Object-Oriented Programming (ECOOP 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 313, pp. 30:1–30:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.ECOOP.2024.30, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.30
10. Piskac, R., Wies, T., Zufferey, D.: Grasshopper. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 124–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
11. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014 – Object-Oriented Programming. pp. 207–231. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
12. Shasha, D., Goodman, N.: Concurrent Search Structure algorithms. ACM Transactions on Database Systems (TODS) **13**(1), 53–90 (1988)
13. Team, T.R.D.: The Rocq Prover (Apr 2025). https://doi.org/10.5281/zenodo.15149629