Compositional Verification of Concurrent C Programs with Search Structure Templates

Duc-Than Nguyen
dnguye96@uic.edu
University of Illinois at Chicago
Chicago, IL, USA

Lennart Beringer
eberinge@cs.princeton.edu
Princeton University
Princeton, NJ, USA

William Mansky
mansky1@uic.edu
University of Illinois at Chicago
Chicago, IL, USA

Shengyi Wang
shengyiw@princeton.edu
Princeton University
Princeton, NJ, USA

Abstract

Concurrent search structure templates are a technique for separating the verification of a concurrent data structure into concurrency-control and data-structure components, which can then be modularly combined with no additional proof effort. In this paper, we implement the template approach in the Verified Software Toolchain (VST), and use it to prove correctness of C implementations of fine-grained concurrent data structures. This involves translating code, specifications, and proofs to the idiom of C and VST, and gives us another look at the requirements and limitations of the template approach. We encounter several questions about the boundaries between template and data structure, as well as some common data structure operations that cannot naturally be decomposed into templates. Nonetheless, the approach appears promising for modular verification of real-world concurrent data structures.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: concurrent separation logic, fine-grained locking, Iris, logical atomicity, interactive theorem proving, Verified Software Toolchain

ACM Reference Format:
Duc-Than Nguyen, Lennart Beringer, William Mansky, and Shengyi Wang. 2024. Compositional Verification of Concurrent C Programs with Search Structure Templates. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), Jan 15–16, 2024, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

CPP '24, Jan 15–16, 2024, London, United Kingdom

https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Krishna et al. [10] proposed concurrent search structure templates as a method for proving the correctness of concurrent data structures *compositionally*, separating the proof of a concurrent access method (e.g., optimistic concurrency, hand-over-hand locking, forwarding via internal links) from the proof of the underlying data structure (e.g., linked list, hashtable, B-tree). The concurrency "templates" are verified parametrically over data structure operations, and the data structure operations are verified without any reference to concurrency. In theory, this could allow us to prove the correctness of n concurrency patterns and m (single-threaded) data structure implementations, and immediately obtain $n \times m$ verified concurrent data structures. In practice, the story is more complicated: certain patterns work only for specific data structures or require the data structures to store extra information, while some internal data structure operations may not fit the template model.

In this paper, we apply search structure templates to the problem of verifying C implementations of concurrent search structures. The template approach was originally implemented on top of flow interfaces [11], a framework for specifying and verifying graph-style data structures, in a combination of two verifiers: the templates were verified in the interactive Iris prover [8], while the data structure implementations were verified with the automated GRASShopper tool [16]. The target data structures were written in HeapLang, a simple functional programming language with shared-memory concurrency. We reimplement the approach in the Verified Software Toolchain (VST) [1], an interactive system for proving correctness of C programs based on a detailed semantics of the C language, and apply it to an existing data structure implemented in C. The template approach depends crucially on the idea of *logical atomicity* introduced in TaDA [3] and further developed in Iris [8], and our proofs make use of recent work integrating Iris-style logical atomicity into VST [13].

Our specific contributions are:

- We reimplement the template approach independently of flow interfaces, with a simple interface involving only the concept of "keys belonging in this node/substructure".
- We implement the template approach in VST, allowing us to apply it to C programs and obtain end-to-end correctness proofs in a single verification system.
- To the best of our knowledge, this is the first mechanized verification of a template approach to concurrent data structure implementations in a real-world programming language, and its first application to data structures not written specifically as case studies.
- We give a precise description of search structure templates, and identify places where it is difficult in practice to preserve the boundary between template and data structure.

Related Work

Recent years have seen major advances in concurrent separation logics (CSLs) for verifying fine-grained concurrent programs, including Iris [8], VST [1, 13], FCSL [17], TaDA [3], and VeriFast [7]. The innovations of Iris (custom ghost state) and TaDA (logical atomicity) were particularly essential for the formalization of the search structure template approach. Some of the more complex data structures verified include Java's `ConcurrentSkipListMap` [22] and a multi-producer multi-consumer concurrent queue from the Folly library [20]. Notably, VST is the only one of these systems that is all three of 1) mechanized (i.e., implemented in a theorem prover), 2) foundational (i.e., connected to a formal semantics of the target language), and 3) targeting an implementation language (C) rather than a toy language or algorithmic representation.

Separating concurrency reasoning from data structure reasoning has long been an appealing target. Linearizability [6], the most common correctness condition for concurrent data structures, was a first step in this direction, describing the conditions under which a concurrent implementation can replace a sequential one in all possible contexts. Logical atomicity has been shown to be a compositional analogue to linearizability [2], where the proof that each operation implements its sequential counterpart can be carried out independently. Concurrency templates can be seen as the next step towards compositionality, allowing us to prove linearizability/atomicity of concurrency patterns independently of specific data structures.

It is also worth mentioning recent work that extends the reach of template-style reasoning. Later work on search structure templates by Patel et al. [15] applies the template approach to multicopy search structures, where there may be more than one node containing the target key. Feldman et al. [5] take an approach similar to templates to verify search structures with highly optimistic concurrency patterns, where the data structure may be restructured by other threads during traversal. Their technique could potentially be applied to prove correctness of much more complex `traverse` operations than those we describe.

2 Background

2.1 Concurrent Search Structure Templates

A search structure is a data structure designed to efficiently store and retrieve data based on specific search criteria. In the abstract, a search structure implements a map from keys to values, providing operations such as search, insertion, deletion, and traversal on that map. For efficiency, many search structures allow concurrent access and modification, often employing fine-grained or lock-free concurrency to allow as many threads as possible to operate on separate parts of the data structure. Designing these search structures presents significant challenges, including ensuring correctness and consistency under concurrent access, achieving scalability by minimizing contention and maximizing parallelism, and maintaining performance and efficiency while managing synchronization and memory, including cache behavior [14]. The complex interplay between concurrency and data structure design in concurrent search structures makes them challenging targets for formal verification.

The Concurrent Search Structure Template approach of Krishna et al. [10] aims to make the problem tractable by separating verification of concurrency control patterns from verification of the underlying data structure. Addressing the proof of each component separately makes the individual proofs easier, and also offers the possibility of proof reuse: each verified concurrency pattern ("template") can be applied to many different data structures, and each verified sequential data structure can be outfitted with many different templates. In theory, verifying n templates and m data structures might yield $n \times m$ verified concurrent data structures; in practice, as we will see, both templates and data structures make assumptions that may invalidate certain combinations. As a side note, the example templates verified by Krishna et al. and the ones we present here focus on synchronization via locks, but the approach also applies to lock-free synchronization (as long as it is sequentially consistent; weak-memory template reasoning has not yet been investigated).

Figure 1 shows an example search structure template. The core of the template is the `traverse` function, which uses a specific concurrency control mechanism to travel through a data structure in search of the requested key. The mechanism in this example is *lock coupling*, where we acquire the lock on the next node before releasing the lock on the current node. The node to travel to 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

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

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

Figure 1. The lock-coupling search structure template

the actual insertion on the node. Thus, the `traverse` and `insert` functions can be 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. The concurrency functions in the template and the sequential `findNext` and `insertOp` functions provided by the data structure combine into a fully operational concurrent data structure. Krishna et al. specify templates in terms of *flow interfaces* [11], a framework for reasoning about graph-structured data structures (i.e., those in which a node may be reachable along multiple paths), but many common search structures do not require flow-style reasoning; in this paper, we present a version of templates that is independent of flow interfaces.

2.1.1 Logical atomicity.

We specify the concurrent behavior of search structures using logical atomicity [3, 8, 13], a separation logic technique that concisely defines the behavior of concurrent operations. A logically atomic triple has the form $\forall a.\ \langle \mathsf{P}_l \mid \mathsf{P}_p(a) \rangle\ \mathsf{c}\ \langle \mathsf{Q}_l \mid \mathsf{Q}_p(a) \rangle$, where P_l and Q_l are *local* preconditions and postconditions, akin to a standard Hoare triple, while P_p and Q_p are *public* preconditions and postconditions, parameterized by an abstract value a of the shared data structure. Intuitively, this says that c is an operation on an abstract object (i.e., data structure) a, and its effect is to *atomically* transform a from a state satisfying P_p to a state satisfying Q_p, with no intermediate states visible to any other thread. More precisely, the triple asserts that if P_l holds true before a call to c and P_p is true for some value of a in a shared state, then P_p will continue to be true for some (possibly different) value of a until the *linearization point* of c, at which point Q_p will become true atomically for the same value a (and Q_l will be true after c ends).

As an example, a sequential stack push operation could conventionally be specified as

$$\{\mathsf{stack}\ s\ p\}\ \mathsf{push}(p, v)\ \{\mathsf{stack}\ (v :: s)\ p\}$$

Its concurrent counterpart could be specified as

$$\forall s.\ \langle \mathsf{is\_stack}_g\ s \mid \mathsf{stack}_g\ vs \rangle$$
$$\mathsf{push}(s, v)$$
$$\langle \mathsf{is\_stack}_g\ s \mid \mathsf{stack}_g\ (v :: vs) \rangle$$

indicating that the push operation of a concurrent stack correctly implements the behavior of a sequential push, atomically transforming the stack from vs to $v :: vs$ at some point during its execution. The stack itself is a shared resource, and can only be accessed and modified atomically by threads holding the corresponding `is_stack` assertion. The local and shared assertions are connected by an arbitrary identifier g, which we will generally omit when clear from context. Logical atomicity can be used to prove linearizability—if all of a data structure's operations satisfy logically atomic triples derived from the corresponding sequential operations, the data structure is linearizable [2]—but can also be used to specify more complex behavior of nonlinearizable data structures [4].

Logical atomicity is key to the template approach: each template's `traverse` function is proved to 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 individual data structure operations, lifting the sequential specifications for insert, lookup, etc. to the concurrent setting.

2.2 Iris and VST

The search structure template approach uses concurrent separation logic to specify and prove pre- and postconditions for the template and data structure functions. Krishna et al. implemented their framework in Iris [8], a language-independent CSL framework built in the Coq proof assistant [19], with flexible support for ghost state, invariants, and atomic specifications. This makes it easy to describe the effects of concurrency control functions independently of the underlying data structure, e.g., as "this function atomically finds a node that contains the target key." Algorithms are verified in Iris's HeapLang, a simple functional programming language with shared-memory concurrency.

To apply the template approach to real-world code, we instead use the Verified Software Toolchain (VST) [1], a separation-logic-based verifier for C programs. VST is also built in Coq and is connected to the CompCert verified C compiler [12], allowing it to guarantee that proved properties will hold on compiled code. Recent work on VST [13] extended it to support most of the advanced concurrency features of Iris, including ghost state, invariants, and atomic specifications. In this paper, we use these extensions to reconstruct the template approach in VST and apply it to real C programs.

3 Search Structure Templates in VST

3.1 What is a Search Structure Template?

Krishna et al. [10] described the search structure template approach and used it to verify (parts of) several concurrent data structures. Their examples strongly suggest a systematic approach to separating sequential data structure logic from concurrent synchronization patterns. However, the approach is not formally defined, and on closer inspection its inputs and outputs are not identical across the example templates. In this section, we attempt to precisely describe the pieces of the search structure template approach, and how we can know whether we have successfully verified a data structure given a collection of verified sequential and concurrent functions.

A concurrent search structure is a data structure that supports three operations: `insert`, `lookup`, and `delete`. It is intended to implement a map from keys to values, and to behave correctly when accessed simultaneously by any number of threads. In separation logic, we can prove correctness of a search structure by showing that its operations satisfy the following logically atomic specifications, where Ref is a per-thread handle to the data structure and Abs is a shared assertion linking the values in memory to an abstract map m from keys to values:

$$\forall m.\ \langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m) \rangle$$
$$\texttt{insert(r, k, v)}$$
$$\langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m[\mathsf{k} \mapsto \mathsf{v}]) \rangle$$

$$\forall m.\ \langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m) \rangle$$
$$\texttt{lookup(r, k)}$$
$$\langle \mathsf{v}.\ \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m) \wedge m(\mathsf{k}) = \mathsf{v} \rangle$$

$$\forall m.\ \langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m) \rangle$$
$$\texttt{delete(r, k)}$$
$$\langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m[\mathsf{k} \mapsto \_]) \rangle$$

In other words, each operation takes effect atomically on the abstract map m, reading and/or updating it as appropriate to the operation. Krishna et al. use flow interfaces [11] to implement the Abs assertion connecting the data structure implementation to the abstract map, but this is not fundamental to the approach: any definition of Ref and Abs that can be used to prove the specifications (and is nontrivial, i.e., holds on the initial state of the data structure) will yield a correct data structure.

A search structure template is an implementation of these three functions, written in a particularly restricted way. Aside from some top-level management code (e.g., acquiring and releasing an initial lock), all that each function should do is call two functions: a function called `traverse`, which is specific to the template and implements its synchronization mechanism, and a concurrency-unaware function (e.g. `insertOp`) provided by the target data structure. (In fact, Krishna et al. implemented all three operations as a single function `decisiveOp`, parameterized by a function that takes the target operation type as input and performs the appropriate data structure operation; for ease of adaptation to C, we write three different functions, but their structures are identical.) A template consists of an implementation of the top-level operations and the `traverse` function used to define them; a data structure consists of an implementation of `insertOp`, etc., as well as a function `findNext` that is used to implement `traverse`. The interface between them is a set of standard specifications for `insertOp`, etc., and `findNext`: if a template's functions can be verified assuming these standard specifications, and a data structure's implementations can be shown to meet those specifications, then the template and the data structure can be combined to yield a verified concurrent search structure.

This is the theory behind the template approach to verification. Unfortunately, it is not straightforward to realize in practice. In the Coq development of Krishna et al., different templates use slightly different specifications for data structure operations and `findNext`; the specifications proved for data structure implementations do not always match those used by templates (which is possible because they use separate provers for the sequential and concurrent proofs); and some data structures rely on other functions that do not appear in the templates (e.g., maintenance operations that split full nodes in B-trees). In the following presentation, we follow the template plan as far as possible, noting the places where we must deviate from the uniform approach; we discuss maintenance operations in further detail in Section 4.2.

3.2 Defining a Template

The core of a search structure template is a `traverse` function that navigates a data structure using the chosen synchronization mechanism. A template also includes wrapper functions that use `traverse` to perform data structure operations (insert, lookup, etc.), and may include helper functions as well. The defining feature of all these functions is that they abstract away from the data structure being traversed: they operate on a generic `node` type and take data-structure-specific functions like `findNext` (which chooses the next node to search) as parameters. We consider lock-based synchronization in this paper, and so our generic node type can be defined in C as:

```
typedef struct node_t {
  node *t;
  lock_t *lock;
} node_t;
```

There are two things worth noting about this data structure. First, each node has its own lock: all of our templates will use *fine-grained locking*, holding locks on as few nodes as possible to allow other parts of the data structure to be modified concurrently. Second, the `node` struct itself is defined by

the target data structure: for instance, a binary search tree might define it as

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

while a linked list might use

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

This is the first place where we decompose our program into a generic concurrent component and a sequential data-structure component. Next, we use `node_t` to define template functions for specific concurrency patterns, and then prove (assuming appropriate specifications for data-structure-specific functions) that they correctly implement the data structure operations.

3.3 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. Figure 2 shows the lock-coupling `traverse` function as presented by Krishna et al. (Figure 2a) and our corresponding C implementation (Figure 2b). The C implementation uses a struct

```
typedef struct pn {
  struct node_t *p;
  struct node_t *n;
} pn;
```

to mimic the pair of nodes `(p, n)` returned by the functional implementation, where `n` is the current node and `p` is its parent. The template relies on one function provided by the underlying data structure, namely `findNext`, which is used to determine the next node `n'` to be visited based on the current node `n` and the key `k`. In the functional version, `findNext` returns an `option node`; in C, it instead returns a Boolean and, if a next node is found, modifies `pn->n`.

Our implementation of `traverse` translates the functional implementation into idiomatic C code. The lock-coupling pattern can be seen on lines 12-13, where `traverse` acquires the next node's lock and then releases the current node's lock. The function stops when it reaches an empty node (`pn->p->t == NULL`), or when `findNext` returns 0, indicating that the target key `k` is in the current node. The `traverse` function returns 1 when we reach an empty node and 0 when we find `k` in an existing node; the behavior of operations that call `traverse` can vary depending on this return value. For instance, an `insert` operation may create a new node when `traverse` returns 1 and modify an existing node when `traverse` returns 0, while a `lookup` operation may fail on 1 and return the value in the current node on 0.

Figure 2c shows the implementation of `insert` for the lock-coupling template. It uses `traverse` to find the node at which to insert the key `k`. If `traverse` returns 0, it has reached a node containing key `k`, and we only have to change the node's value to the target `value` (lines 5-6 in Figure 2c). Otherwise, `traverse` has reached an empty node that can hold key `k`, so it calls the underlying data structure's `insertOp` function to allocate a new node with key `k` and value `v` (line 9 in Figure 2c).

3.3.1 Verifying the Lock-Coupling Template.

We prove correctness of the template by showing that `traverse`, `insert`, etc. meet logically atomic specifications describing their effects on the data structure. These specifications are defined in terms of assertions Ref, representing a client thread's handle to the data structure, and Abs, representing the data structure's abstract state. We also have an assertion $\mathsf{In}(n)$ that serves as a reference to an individual node; Ref should include, at minimum, the In assertion for the root node. Holding $\mathsf{In}(n)$ for a node n should allow us to acquire the lock on n, which in turn gives us access to the contents of the node. Formally, we need to know that the following triples hold:

$$\langle \mathsf{In}\,(n) \mid \mathsf{Abs}\,(m)\rangle \quad \texttt{acquire}\,(n\texttt{->lock}) \quad \langle \mathsf{In}(n) * \mathsf{R}(n) \mid \mathsf{Abs}\,(m)\rangle$$

$$\langle \mathsf{In}\,(n) * \mathsf{R}(n) \mid \mathsf{Abs}\,(m)\rangle \quad \texttt{release}\,(n\texttt{->lock}) \quad \langle \mathsf{In}(n) \mid \mathsf{Abs}\,(m)\rangle$$

In other words, $\mathsf{In}(n)$ is sufficient to guarantee that node n is in the abstract state of the data structure and its lock protects associated resources $\mathsf{R}(n)$, the *lock invariant* for the node.

The `traverse` function can then be specified as follows:

$$\forall\, m.\ \left\langle \mathsf{pn} \mapsto (p, n) \ * \ \mathsf{In}(n) * \ \mathsf{R}(n) \ \middle|\ \mathsf{Abs}\,(m) \right\rangle$$

$$\texttt{traverse(pn, k)}$$

$$\left\langle \begin{array}{l} \mathit{res}.\exists\, n', v, \mathsf{range}. \\ \quad \mathsf{pn} \mapsto (n', n') * \mathsf{In}\,(n') * \ \mathsf{k} \in \mathsf{range} * \\ \quad \text{if } \mathit{res} \text{ then } \mathsf{node\_contents}(n', \cdot, \mathsf{range}) \\ \qquad \text{else } \mathsf{node\_contents}(n', (\mathsf{k}, v), \mathsf{range}) \end{array} \middle|\ \mathsf{Abs}\,(m) \right\rangle$$

The local precondition of `traverse` includes both the node handle $\mathsf{In}(n)$ and its contents $\mathsf{R}(n)$, indicating that a thread should already hold n's lock before calling `traverse`. The output of `traverse` is a new node n' in the data structure such that the key `k` falls within the range of n'. The local postcondition then includes the handle of the new node $\mathsf{In}(n')$, its contents $\mathsf{R}(n')$, and a Boolean variable *res* indicating whether `traverse` found an empty node or a node with key `k`.

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

(a) The `traverse` method of the lock-coupling template algorithm written in an ML-like language [9]

```
void insert (node_t **r, int k, void *v){
  struct pn *pn = (struct pn*)surely_malloc(sizeof
     *pn);
  pn->n = *r;
  acquire(pn->n->lock);
  if (traverse(pn, k) == 0){
    pn->p->t->value = v;
  }
  else{
    insertOp(pn, k, v);
  }
  release(pn->n->lock);
  free(pn);
}
```

(c) The `insert` method of the lock-coupling template algorithm written in C

```
int traverse(pn *pn, int k){
  for( ; ; ){
    pn->p = pn->n;
    if (pn->p->t == NULL)
      return 1;
    else{
      int b = findNext(pn, k);
      if (b == 0){
        return 0;
      }
      else{
        acquire(pn->n->lock);
        release(pn->p->lock);
      }
    }
  }
}
```

(b) The `traverse` method of the lock-coupling template algorithm written in C

Figure 2. The `traverse` method of the lock-coupling template algorithms

The resources R contained in a node depend on the specific data structure, but always include a piece of *ghost state* describing the current state of the node (its key, value, and key range) shared between the invariant R and the abstract state Abs, ensuring that the lock and the abstract state agree on the contents of the node. Formally, the lock invariant is defined by

$$\begin{aligned}&\mathsf{node\_contents}(n, c, \mathsf{range}) \triangleq \\ &\qquad \mathsf{ghost\_node}(n, c, \mathsf{range}) * \mathsf{node\_data}(n, c) \\ &\mathsf{R}(n) \triangleq \exists\, c, \mathsf{range}.\ \mathsf{node\_contents}(n, c, \mathsf{range})\end{aligned}$$

where the definition of node_data is supplied by the target data structure. The contents c of a node can be either a key-value pair (k, v), or the empty contents $\cdot$ (used for nodes that have been allocated but not yet assigned keys). Then Abs (m) is defined as a collection of ghost_nodes that form a tree containing all the key-value pairs in m.

The key to the correctness of the `traverse` function is the loop invariant for the top-level loop, which expresses that in each iteration, `traverse` holds the lock on a node that has k in its range:

$$\begin{aligned}&\mathsf{traverse\_inv}(\mathsf{pn}, \mathsf{k}) \triangleq \\ &\qquad \exists\, p, n, c, \mathsf{range}.\ \mathsf{pn} \mapsto (p, n) * \mathsf{k} \in \mathsf{range} * \\ &\qquad\qquad \mathsf{In}(n) * \mathsf{node\_contents}(n, c, \mathsf{range})\end{aligned}$$

Figure 3 shows the proof outline of the `traverse` function. We begin by checking whether the current node is null (line 5 of Figure 3); if it is, we have found the empty node where k belongs, and can prove the postcondition with *res* = true. Otherwise, we pass the node_data from the lock invariant to `findNext` (line 11), which returns a new node n'' to visit, stored in `pn->n`. If `findNext` returns 0, we have found a node containing k, and can prove the postcondition with *res* = false. Otherwise, we acquire n`->lock`, gaining access to the resources $\mathsf{R}(n)$, and then release the lock of the current node p and return its resources. By acquiring the lock for the next node n before releasing the lock for the current node p, we ensure that the connection between the two nodes remains intact and unaltered while we traverse it, and re-establish traverse_inv for the new values in pn. The two **`return`**s are also the two possible linearization points of the function.

$$\forall\, m.\, \langle \mathsf{pn} \mapsto (p, n) \;*\; \mathsf{In}(n) \;*\; \mathsf{R}(n) \mid \mathsf{Abs}\ (m) \rangle$$

```
1  int traverse(pn *pn, int k){
2    { pn ↦ (p,n) * In(n) * R(n) } ⇛ { traverse_inv }
3    for( ; ; ){                    { traverse_inv }
4      pn->p = pn->n;
5      { pn ↦ (n',n') * In(n') * k ∈ range * node_contents(n',c,range) }
6      if (pn->p->t == NULL)
7        return 1;
8        { pn ↦ (n',n') * In(n') * k ∈ range * node_contents(n',·,range) }
9      else{
10       { pn ↦ (n',n') * In(n') * k ∈ range * node_contents(n',(k',v'),range) }
11       int b = findNext(pn, k);
12       { ∃ n''. pn ↦ (n',n'') * In(n') * k ∈ range * node_contents(n',(k',v'),range) *
         { ((b = 0 * k' = k * n'' = n') ∨ (b = 1 * k ∈ range(n'') * In(n'')))
13       if (b == 0){
14         { pn ↦ (n',n') * k' = k * ··· }
15         return 0;
16         { pn ↦ (n',n') * In(n') * k ∈ range' * node_contents(n',(k,v'),range') }
17       }
18       else{
19         { pn ↦ (n',n'') * In(n') * k ∈ range(n'') * R(n') * In(n'') }
20         acquire(pn->n->lock);
21         { pn ↦ (n',n'') * In(n') * k ∈ range(n'') * R(n') * In(n'') * R(n'') }
22         release(pn->p->lock);
23         { pn ↦ (n',n'') * k ∈ range(n'') * In(n'') * R(n'') }
24       }
25     }
26   }
27 }
```

$$\left\langle \begin{array}{r} \mathsf{pn} \mapsto (n', n') \;*\; \mathsf{In}(n') \;*\; \mathsf{k} \in \mathsf{range} * \\ \mathit{res}.\ \exists\, n', v.\ \ \text{if } \mathit{res} \text{ then } \mathsf{node\_contents}(n', \cdot, \mathsf{range}) \\ \text{else } \mathsf{node\_contents}(n', (\mathsf{k}, v), \mathsf{range}) \end{array} \middle| \mathsf{Abs}\ (m) \right\rangle$$

Figure 3. Proof outline of the lock-coupling `traverse` function

The `insert` function uses this specification of `traverse` to update the state of the data structure. The desired specification of `insert` is:

$$\forall m.\, \langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m) \rangle$$
$$\texttt{insert(r, k, v)}$$
$$\langle \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m[\mathsf{k} \mapsto \mathsf{v}]) \rangle$$

where m and $m[\mathsf{k} \mapsto \mathsf{v}]$ denote the tree's abstract states before and after the function's execution, respectively. The proof of `insert` is outlined in Appendix A, but informally it is quite simple. We begin by initializing the pn struct with the root node and acquiring its lock, allowing us to satisfy the precondition of `traverse`. If `traverse` returns 0, we have found a node with key k, and all we need to do is update that node's value to v (line 6 in Figure 2c). Otherwise, `traverse` has reached an empty node with k in its range. We then call the data-structure-specific `insertOp` function, which inserts the new key-value pair at the empty node. In either case, before we release the node's lock, we must show that we have altered the tree from its current abstract state m to $m[\mathsf{k} \mapsto \mathsf{v}]$, thereby satisfying the public postcondition of `insert`. We do this by demonstrating that in both cases, the update to the concrete data structure corresponds to setting k to v in the abstract key-value map of the data structure. Finally, we release the lock acquired by `traverse` and deallocate the pn structure.

3.4 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 must explicitly check whether the target key is in the range of the current node. 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.

Figure 4a shows the give-up template algorithm as originally presented. In addition to `findNext`, the `traverse` function uses a helper function called `inRange`, which determines whether the key value k falls within the range of keys held in node n and its successors. Logically, this range is the same as the range in the lock-coupling template, but that range was a ghost-state construct that only appeared in the proofs; in the give-up template, the range must be stored in memory and checked in the code. If k is outside the node's range (e.g. because the node has been relocated), the search is restarted from the root node r. As in the previous section, we implement this in C with a loop: in each iteration, we acquire the lock on the current node, check that k is in range, and then use `findNext` as above, releasing the lock before we move to the next node. If the `inRange` call fails, we release our current lock and return to the root node by setting the current node to the root pointer p (lines 21-22 in Figure 4b). The give-up version of the `insert` operation (see Figure 4c) is almost identical to the lock-coupling version, except that it does not acquire a lock before calling `traverse`.

The `inRange` function raises an interesting question about the template approach: is `inRange` part of the give-up template, or the underlying data structure? Some data structures may already track the range of keys expected in the current node, and so might define `inRange` even in sequential implementations. However, in most sequential settings ranges can be computed from the data structure (e.g., in a binary search tree, the left subtree of a node with key k holds keys less than k), and there is no reason to explicitly store a node's range in the node itself. The give-up template of Krishna et al. implicitly assumes that the underlying data structure supports `inRange`; we prefer to consider `inRange` part of the template, so that the template can be applied to data structures without modifying them. Accordingly, in this template we add two new fields, `min` and `max`, to the type `node_t`:

```
typedef struct node_t {
  node *t;
  lock_t *lock;
  int min, max;
} node_t;
```

These fields store lower and upper bounds on the keys reachable from the current node. We can then define `inRange` as a helper function in the template, rather than requiring data structures to provide it. Our specification of `inRange` is:

$$\{\ \texttt{pn->n->min} \mapsto n_1 * \texttt{pn->n->max} \mapsto n_2\ \}$$
$$\texttt{inRange(pn, k)}$$
$$\left\{ res.\ \begin{array}{l} \texttt{pn->n->min} \mapsto n_1 * \texttt{pn->n->max} \mapsto n_2 * \\ \text{if } res \text{ then } (n_1 < \texttt{k} < n_2) \text{ else } (\texttt{k} \le n_1 \vee \texttt{k} \ge n_2) \end{array} \right\}$$

It simply computes whether the input key k is in the range (n_1, n_2) associated with the node n, and returns 0 or 1 accordingly.

3.4.1 Verifying the Give-Up Template. The give-up template's `traverse` specification is almost the same as in the lock-coupling template, except that the caller does not need to hold any locks before calling it, so the invariant R does not appear in the precondition:

$$\forall\, m.\ \left\langle\ \texttt{pn} \mapsto (p, n) * \mathsf{In}(n)\ \middle|\ \mathsf{Abs}\ (m)\right\rangle$$
$$\texttt{traverse(pn, k)}$$
$$\left\langle \begin{array}{l} res. \exists\, n', v, \text{range}. \\ \quad \texttt{pn} \mapsto (n', n') * \mathsf{In}\,(n') * \texttt{k} \in \text{range} * \\ \quad \text{if } res \text{ then node\_contents}(n', \cdot, \text{range}) \\ \qquad \text{else node\_contents}(n', (\texttt{k}, v), \text{range}) \end{array} \middle|\ \mathsf{Abs}\ (m) \right\rangle$$

As before, the `traverse` function will navigate the structure and return a node n' whose range includes k, as well as acquiring n''s lock and returning its contents. Also as before, the Boolean *res* indicates whether n' is empty or contains the key k. Our definition of Abs for the give-up template closely follows that of Krishna et al.

Next, we define the loop invariant for the give-up template's `traverse` function:

$$\mathsf{traverse\_inv}(\texttt{pn}) \triangleq$$
$$\exists\, p, r, n.\ \texttt{pn} \mapsto (p, n) * \mathsf{In}(r) * \mathsf{In}(n)$$

Unlike the lock-coupling case, we do not hold any locks between iterations of the loop body, and we do not maintain the fact that k is in the range of the current node (we must check this later with `inRange`). Instead, we keep a reference $\mathsf{In}(r)$ to the root node, so that we can return to it if an `inRange` check fails.

With this invariant in hand, the proof of `traverse` proceeds as follows. We begin by using the In predicate to acquire the lock on `pn->n` (line 4 in Figure 4b). We then call `inRange` to check whether we are still on the path to a node that can hold k. If the check fails (lines 18-20), we immediately release the lock and start over from the root node r. Otherwise, we proceed as in the lock-coupling template: if the current node's contents are `NULL`, we have found the empty node where k belongs (and can satisfy the postcondition); if `findNext` returns 0, we have found the node containing k (and can satisfy the postcondition); otherwise, we release the lock and re-establish traverse_inv for the new node indicated by `findNext`. In this case, we once again hold no locks, and have only the In assertion indicating that the new node is in the data structure.

The specification of the `insert` function is the same as for the lock-coupling template, and its proof is quite similar as well. The fact that the give-up version of `insert` does not acquire a lock before calling `traverse` is reflected in the difference in the precondition of `traverse` between the two

```
let rec traverse r n k =
  lockNode n;
  if inRange n k then
    match findNext n k with
    | None -> n
    | Some n' -> unlockNode n;
        traverse n' k
  else
    unlockNode n;
    traverse r r k
```

(a) The `traverse` method of the give-up template algorithm written in an ML-like language

```
void insert (node_t **r, int k, void *v){
  struct pn *pn = (struct pn *) surely_malloc (
    sizeof *pn);
  pn->n = *r;
  if (traverse(pn, k) == 0){
    pn->p->t->value = v;
  }
  else{
    insertOp(pn, k, v);
  }
  release(pn->n->lock);
  free(pn);
}
```

(c) The `insert` method of the give-up template algorithm written in C

```
int traverse(pn *pn, int k){
  node_t *r = (pn->n);
  for( ; ; ){
    acquire(pn->n->lock);
    pn->p = pn->n;
    if (inRange(pn, k) == 1){
      if (pn->p->t == NULL)
        return 1;
      else{
        int b = findNext(pn, k);
        if (b == 0){
          return 0;
        }
        else
          release(pn->p->lock);
      }
    }
    else{
      release(pn->p->lock);
      pn->n = r;
    }
  }
}
```

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

Figure 4. The `traverse` and `insert` methods of the give-up template algorithms

templates. Proof outlines for the give-up template functions can be found in Appendix A.

4 Verifying Binary Search Trees using Templates

In this section, we demonstrate how to instantiate the templates with specific data structures, giving us verified C implementations of concurrent data structures. As we saw in the previous section, the top-level specifications of data structure operations have already been verified at the template level; all we need to do now is provide implementations of data-structure-specific functions (e.g. `findNext`) that satisfy the right specifications to instantiate the template proofs. Once we have chosen a template and a data structure, we should not need to do any further reasoning about specific concurrency patterns.

Our target data structure is a binary search tree (BST) implemented in C. A lock-coupling BST was already written verified as a demonstration of VST's concurrency capabilities [18]. We refactored it into `findNext`, `traverse`, `insertOp`, etc. as required by the template approach, and then re-verified it using the template; we then replaced `traverse` with the give-up version and verified the resulting give-up BST as well. We were able to obtain verified `insert` and `lookup` operations using this technique; for `delete`, we found a mismatch between the template approach and BST deletion, which we discuss in Section 4.2.

4.1 Instantiating `findNext` and `insertOp`

We instantiate the templates with the binary search tree by defining and verifying the helper functions `findNext` (used in `traverse`) and `insertOp` (used in `insert`). As expected, their implementations are strictly sequential, their specifications are ordinary non-atomic Hoare triples (see Figures 5 and 6), and their proofs do not require any concurrency reasoning.

The contents of a BST node are defined in C as:

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

The BST's `findNext` function (Figure 7) indicates whether the target key is in the current node, its left subtree, or its

$$\left\{\ \texttt{pn->n->t} \mapsto (k', v', l, r)\ \right\}$$
$$\texttt{findNext(pn, k)}$$
$$\left\{ res.\ \exists\ n'.\ \begin{array}{c} \texttt{pn->n->t} \mapsto (k', v', l, r) * \\ \text{if } res \text{ then } (l = n' \wedge \texttt{k} < k') \vee (r = n' \wedge \texttt{k} > k') \text{ else } (n = n' \wedge \texttt{k} = k') \end{array} \right\}$$

Figure 5. Specification of `findNext` for the binary search tree (all templates)

$$\left\{\ \texttt{pn->n} \mapsto \texttt{NULL}\ \right\}$$
$$\texttt{insertOp(pn, k, v)}$$
$$\left\{ \exists\ l\ r\ lk_1\ lk_2.\ \begin{array}{c} \texttt{pn->n->t} \mapsto (\texttt{k}, \texttt{v}, l, r) * \\ l \mapsto (\texttt{NULL}, lk_1) * r \mapsto (\texttt{NULL}, lk_2) \end{array} \right\}$$

(a) Specification of `insertOp` for the lock-coupling template

$$\left\{\ \texttt{pn->n} \mapsto \texttt{NULL} * \texttt{pn->n->min} \mapsto n_1 * \texttt{pn->n->max} \mapsto n_2\ \right\}$$
$$\texttt{insertOp(pn, k, v)}$$
$$\left\{ \exists\ l\ r\ lk_1\ lk_2.\ \begin{array}{c} \texttt{pn->n->t} \mapsto (\texttt{k}, \texttt{v}, l, r) * \texttt{pn->n->min} \mapsto n_1 * \texttt{pn->n->max} \mapsto n_2 * \\ l \mapsto (\texttt{NULL}, lk_1, (n_1, \texttt{k})) * r \mapsto (\texttt{NULL}, lk_2, (\texttt{k}, n_2)) \end{array} \right\}$$

(b) Specification of `insertOp` for the give-up template

Figure 6. Specification of `insertOp` for the lock-coupling and give-up templates

right subtree. The precondition of `findNext` (Figure 5) states that the input has a non-null `t` field pointing to a node struct with key k', value v', and pointers to the left l and right r child nodes. The postcondition describes three possible outcomes: either the provided key `k` is less than the node's key k', and the next node n' is the left child l; `k` is greater than k' and n' is the right child r; or `k` is exactly k' and n' is the original node n, in which case the return value is 0. The proof is straightforward.

```
1  int findNext (pn *pn, int k){
2    int y = pn->p->t->key;
3    if (k < y){
4      pn->n = pn->p->t->left;
5      return 1;
6    }
7    else if (k > y){
8      pn->n = pn->p->t->right;
9      return 1;
10   }
11   else
12     return 0;
13 }
```

Figure 7. C implementation of `findNext` for the binary search tree (all templates)

The BST's `insertOp` function allocates a new node struct at an empty leaf, as well as two new empty leaves to serve as its children. The `insertOp` specification (Figure 6a) takes as input a node whose `t` field is `NULL`, i.e., an empty leaf node. It inserts a new node with the specified key `k` and value `v` at that leaf, complete with two new empty child nodes with corresponding locks, $(\texttt{NULL}, lk_1)$ and $(\texttt{NULL}, lk_2)$, which are pointed to by the l and r of the current node.

Unfortunately, while this specification suffices for the lock-coupling template, it cannot be used as is for the give-up template: because we allocate new empty leaf nodes, we must allocate `node_t` structures, whose implementation in the give-up template has extra fields that do not appear in the lock-coupling template. As shown in Figure 6b, a give-up template node (even an empty one!) has additional `min` and `max` fields, and these fields are needed to compute the bounds of the new child nodes. In a BST, if the current node's range is (`min, max`) and its key is `k`, its left child will have range (`min, k`) and its right child will have range (`k, max`). It may be possible to remedy this discrepancy by adding an "allocate empty node" function to the templates, which is then called by `insertOp`, but this would still complicate the approach: originally, we expected that templates would be parameterized by data structure implementations but not vice versa. However, this two-way dependency seems fundamental when the layout of the node structure depends on the concurrency control mechanism we choose. For now, we break the abstraction boundary and verify two different versions of `insertOp`, one for the lock-coupling template and one for the give-up template

4.2 Templates and Internal Reorganization

The `delete` operation on binary search trees highlights a limitation of the template approach as we have described it. The `insert` and `lookup` operations can be factored into a data-structure-agnostic concurrent step (`traverse`) and a concurrency-unaware data structure step (`insertOp` or `lookupOp`). In most of the prior template examples, deletion can be decomposed similarly, by traversing to the node to be deleted, acquiring its lock and possibly its parent's lock, and then performing a local, sequential data-structure-specific operation. When removing a node from a BST, on the other hand, we usually restructure the tree with a `pushdown_left` operation, rotating three-node sections of the tree until the node to be deleted is at a leaf. This operation is simultaneously concurrency-aware (we must lock the nodes involved to avoid race conditions) and data-structure-specific (we need to know precisely which nodes to target and how to rearrange them).

Krishna et al. also discuss an operation of this sort, the `split` operation in B-link trees. Their approach is to assume the existence of a separate maintenance thread that constantly crawls the tree searching for full nodes and splitting them; any operation that cannot take effect on full nodes can simply retry until the target node is split. The split itself is a no-op on the abstract state of the data structure: it makes internal structural changes but does not change the set of keys stored in the tree. Maintenance operations of this sort appear in almost every reasonably complex concurrent data structure, and so the template approach must account for their existence. In other words, in general the template approach must divide the operations of a target data structure into three categories: concurrent, data-structure-agnostic template operations, which can be verified once and applied to multiple data structures; sequential data structure operations, which can be verified once and plugged into multiple templates; and no-op maintenance operations, which are both concurrency-aware and data-structure-specific, but whose specifications leave the abstract state unchanged. For instance, rotation in the BST can be seen as a maintenance operation, and the deletion of a node that has already been rotated to a leaf position is a sequential, data-structure-specific operation. Rebalancing operations in AVL and red-black trees would also be considered maintenance operations. The existence of this third category means that we cannot completely separate concurrency reasoning from data structure reasoning, but because maintenance operations are logical no-ops, they should be easier to verify in a non-decomposed style than most concurrent data structure operations.

5 Proof Mechanization

All the proofs described above have been mechanized in VST, using its extensions for logical atomicity [13]. Statistics on the verification effort are shown in Table 1.

Lock coupling	**LoC**	**Give up**	**LoC**
Specifications	190	Specifications	138
`findNext` proof	27	`findNext` proof	27
`insertOp` proof	32	`insertOp` proof	31
		`inRange` proof	30
`traverse` proof	360	`traverse` proof	336
`insert` proof	450	`insert` proof	284
`lookup` proof	235	`lookup` proof	304
`pushdown_left` proof	282		
`delete` proof	182		
Supporting proofs	1909	Supporting proofs	1876
Total	**3667**		**3026**

Table 1. Size of Coq definitions and proofs, by topic.

The total lines of code required were about the same as for the original VST verification of the BST. The vast majority of the proof effort was in the templates, in both `traverse` and the top-level data structure operations. This is fairly encouraging, since this proof effort should be reusable for other data structures that use the same templates. For lock-coupling deletion, we retained the original (non-templated) proofs; we did not verify give-up deletion, since optimistic deletion is considerably harder to reason about and did not stand to benefit from the template approach. Proving the correctness of e.g. a lock-coupling linked list should be as simple as defining a new `node` struct, and then implementing and verifying `findNext` and `insertOp` (plus the appropriate functions for deletion).

Compared to prior template proofs in Iris and GRASShopper, our proof development is significantly larger (although direct comparison is difficult, since GRASShopper is an automatic prover). The biggest structural difference is that we need to directly state loop invariants for our imperative code, while the Iris proofs handle recursion with a simpler Löb-induction approach. Coming up with the loop invariants was the hardest part of the proofs. Our proofs also deal with the details of real C code: proving absence of integer overflows, accurately representing struct-and-pointer-based data structures instead of functional-programming-style structured data, etc. In exchange for this extra effort, the code we verify constitutes complete C programs that can be compiled and executed. The full development of our mechanization effort is available online at https://zenodo.org/record/8337004.

6 Conclusion and Future Work

Concurrent search structure templates are a promising approach for proving the correctness of concurrency patterns and data structures separately, and then combining them to obtain verified concurrent data structures. We have translated the approach to apply to C programs using VST, with appropriate imperative versions of the key `traverse` functions that define the templates, and used it to prove the

correctness of an (appropriately refactored) existing concurrent data structure. However, the translation also exposed some ambiguities and limitations in the approach: for instance, the `inRange` function of the give-up template must either be assumed to exist in the data structure or incorporated into the template, and the `delete` operation of binary search trees does not decompose naturally into a concurrent part and a data-structure-specific part. The next step is to implement templates as a truly generic framework, with proofs of `traverse`, etc. that can be freely combined with verified sequential data structures via a clear interface that defines the functions a data structure must provide and the assumptions the template makes on them (e.g., by verifying template functions with respect to a data-structure typeclass). Ultimately, we hope to use the template approach to verify data structures such as Masstree [14] and Wormhole [21], complicated real-world search structures that combine multiple concurrency patterns for maximum performance on multicore architectures.

Acknowledgments

We thank Roshan Sharma, Alex Oey, and Anastasiia Evdokimova for extensive work on the original binary search tree implementation and verification, and the anonymous reviewers for providing comprehensive and insightful reviews. This research was supported, by ...

References

[1] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press.

[2] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (aug 2021), 29 pages. https://doi.org/10.1145/3473586

[3] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231.

[4] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 792–808. https://doi.org/10.1145/3519939.3523451

[5] Yotam MY Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving Highly-Concurrent Traversals Correct. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

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

[7] Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/1926385.1926417

[8] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[9] Siddharth Krishna. 2019. *Compositional Abstractions for Verifying Concurrent Data Structures.* Ph. D. Dissertation. New York University.

[10] Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 181–196. https://doi.org/10.1145/3385412.3386029

[11] Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2017. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *Proc. ACM Program. Lang.* 2, POPL, Article 37 (dec 2017), 31 pages. https://doi.org/10.1145/3158125

[12] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[13] William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. *CoRR* abs/2207.06574 (2022). https://doi.org/10.48550/ARXIV.2207.06574

[14] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[15] Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying Concurrent Multicopy Search Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 113 (oct 2021), 32 pages. https://doi.org/10.1145/3485490

[16] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–139.

[17] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

[18] Roshan Sharma, Shengyi Wang, Alexander Oey, Anastasiia Evdokimova, Lennart Beringer, and William Mansky. 2022. Proving Logical Atomicity using Lock Invariants. (2022). Presented at Advances in Separation Logic (ASL 2022).

[19] The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.5846982

[20] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 100–115. https://doi.org/10.1145/3497775.3503689

[21] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16. https://doi.org/10.1145/3302424.3303955

[22] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *Proceedings*

of the 26th European Symposium on Programming (ESOP'17) (Lecture Notes in Computer Science, Vol. 10201), Hongseok Yang (Ed.). Springer, 964–990. https://doi.org/10.1007/978-3-662-54434-1_36

A Proof Outlines for Templates (Sections 3.3.1 and 3.4.1)

Below are proof outlines for the `traverse` function for the give-up template (Figure 8) as well as the `insert` function for both templates. The complete proofs can be found in our Coq mechanization (see Section 5).

$$\forall\, m.\, \langle\ \mathsf{pn} \mapsto (p, n) * \mathsf{ln}(n) \mid \mathsf{Abs}\ (m)\ \rangle$$

```
1  int traverse(pn *pn, int k){
2    node_t *r = (pn->n);   { pn ↦ (_, r) * ln(r) } ⇛ { traverse_inv }
3    for( ; ; ){            { traverse_inv } ≜ { ∃ n'. pn ↦ (_, n') * ln(n') * ln(r) }
4      acquire(pn->n->lock);
5      { pn ↦ (_, n') * ln(n') * R(n') * ln(r) }
6      pn->p = pn->n;  { pn ↦ (n', n') * ln(n') * R(n') * ln(r) }
7      if (inRange(pn, k) == 1){
8        { k ∈ range * pn ↦ (n', n') * ln(n') * node_contents(n', c, range) * ln(r) }
9        if (pn->p->t == NULL)
10         return 1;
11         { k ∈ range * pn ↦ (n', n') * ln(n') * node_contents(n', ·, range) }
12       else{
13         { k ∈ range * pn ↦ (n', n') * ln(n') * node_contents(n', (k', v'), range) * ln(r) }
14         int b = findNext(pn, k);
15         { ∃ n''. pn ↦ (n', n'') * ln(n') * k ∈ range * node_contents(n', (k', v'), range) *
           { ln(r) * ((b = 0 * k' = k * n'' = n') ∨ (b = 1 * ln(n''))) }
16         if (b == 0){
17           { pn ↦ (n', n') * k' = k * ··· }
18           return 0;
19           { pn ↦ (n', n') * ln(n') * k ∈ range' * node_contents(n', (k, v'), range) }
20         }
21         else{
22           { pn ↦ (n', n'') * ln(n') * k ∈ range * node_contents(n', (k', v'), range) *
             { ln(n'') * ln(r) }
23           release(pn->p->lock);
24           { pn ↦ (n', n'') * ln(n') * k ∈ range * ln(n'') * ln(r) }
25         }
26       }
27     }
28     else{
29       { pn ↦ (n', n') * ln(n') * R(n') * ln(r) }
30       release(pn->p->lock);
31       { pn ↦ (n', n') * ln(n') * ln(r) }
32       pn->n = r;
33       { pn ↦ (n', r) * ln(r) }
34     }
35   }
36 }
```

$$\left\langle \begin{array}{r} \mathsf{pn} \mapsto (n', n') * \mathsf{ln}(n') \ *\ \mathsf{k} \in \mathsf{range} * \\ res.\ \exists\, n', v.\ \ (\text{if } res \text{ then node\_contents}(n', \cdot, \mathsf{range}) \\ \text{else node\_contents}(n', (\mathsf{k}, v), \mathsf{range})) \end{array} \middle|\ \mathsf{Abs}\ (m) \right\rangle$$

Figure 8. Proof outline of the give-up `traverse` function

$$\forall\, m.\, \langle\ \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m)\ \rangle$$

```
1  void insert (node_t **r, int k, void *v){
2    struct pn *pn = (struct pn *)surely_malloc(sizeof *pn);
3    pn->n = *r;
4    { pn ↦ (_, r) ∗ Ref(r) }
5    acquire(pn->n->lock);
6    { pn ↦ (_, r) ∗ Ref(r) ∗ R(r) }
7    if (traverse(pn, k) == 0){
8      { ∃ n'. pn ↦ (n', n') ∗ Ref(n') ∗ node_contents(n', (k, v'), range) }
9      pn->p->t->value = v;
10     { pn ↦ (n', n') ∗ Ref(n') ∗ node_contents(n', (k, v), range) }
11     //Linearization point
12   }
13   else{
14     { ∃ n'. pn ↦ (n', n') ∗ n'->t = NULL ∗ Ref(n') ∗ node_contents(n', ·, range) ∗ k ∈ range }
15     insertOp(pn, k, v);
16     { pn ↦ (n', n') ∗ Ref(n') ∗ node_contents(n', ·, range) ∗ k ∈ range ∗
       { n'->t ↦ (k, v, l, r') ∗ l ↦ NULL ∗ r' ↦ NULL }
17     //Linearization point
18     { pn ↦ (n', n') ∗ Ref(n') ∗ node_contents(n', (k, v), range) ∗ k ∈ range }
19   }
20   { pn ↦ (n', n') ∗ Ref(n') ∗ R(n') }
21   release(pn->n->lock);
22   { pn ↦ (n', n') ∗ Ref(n') }
23   free(pn);
24   { Ref(n') }
25 }
```

$$\langle\ \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m[\mathsf{k} \mapsto \mathsf{v}])\ \rangle$$

(a) Proof outline of the lock-coupling `insert` function

$$\forall\, m.\, \langle\ \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m)\ \rangle$$

```
1  void insert (node_t **r, int k, void *v){
2    struct pn *pn = (struct pn *)surely_malloc(sizeof *pn);
3    pn->n = *r;
4    { pn ↦ (_, r) ∗ In(r) }
5    if (traverse(pn, k) == 0){
6      { ∃ n'. pn ↦ (n', n') ∗ In(n') ∗ node_contents(n', (k, v'), range) }
7      pn->p->t->value = v;
8      { pn ↦ (n', n') ∗ Ref(n') ∗ node_contents(n', (k, v), range) }
9      //Linearization point
10   }
11   else{
12     { ∃ n'. pn ↦ (n', n') ∗ n'->t = NULL ∗ In(n') ∗ node_contents(n', ·, range) ∗ k ∈ range }
13     insertOp(pn, k, v);
14     { pn ↦ (n', n') ∗ In(n') ∗ node_contents(n', ·, range) ∗ k ∈ range ∗
       { n'->t ↦ (k, v, l, r') ∗ l ↦ NULL ∗ r' ↦ NULL }
15     //Linearization point
16     { pn ↦ (n', n') ∗ In(n') ∗ node_contents(n', (k, v), range) ∗ k ∈ range }
17   }
18   { pn ↦ (n', n') ∗ In(n') ∗ R(n') }
19   release(pn->n->lock);
20   { pn ↦ (n', n') ∗ In(n') }
21   free(pn);
22   { In(n') }
23 }
```

$$\langle\ \mathsf{Ref}(\mathsf{r}) \mid \mathsf{Abs}\ (m[\mathsf{k} \mapsto \mathsf{v}])\ \rangle$$

(b) Proof outline of the give-up `insert` function

Figure 9. Proof outlines for the `insert` function