A Formal Interface for Concurrent Search Structure Templates

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

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

Abstract

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 and William Mansky. 2025. A Formal Interface for Concurrent Search Structure Templates. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Search structure templates [3] were introduced by Krishna et al. as an approach to modular verification of concurrent data structures. The approach consists of a programming pattern and specification style that decompose concurrent data structures into *concurrency templates* (lock coupling, give-up) and *sequential data structures* (linked list, hash table). Later work in the same line extends the template approach to adding multicopy support (e.g., log-structure merging) [], or combining thread-safe node implementations into larger data structures []. 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.

However, while Krishna et al. [3] lay out the high-level ideas of the template approach and work through several examples, they do not formally define the approach. 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 theory, given m data structure components and n concurrency templates, we should immediately be able to obtain $m \times n$ verified concurrent data structures. The original paper by Krishna et al. 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, 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 a surprisingly challenging task: ideally, the data structure implementations should not explicitly leave space for concurrency metadata like locks, and the concurrency mechanism should not come between pieces of the data structure. We discuss several undesirable approaches and their relationships to object-orientation patterns, and show that the most desirable approach corresponds to *traits*, though it can also be implemented in C.
- We give a formal **interface** for data structure components and concurrency templates, in the form of separation logic specifications for each of the relevant functions. These specifications follow the style of Krishna et al. [3], 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 the 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 BST) and two concurrency templates (lock coupling and give-up) that satisfy our interface, and from them **freely obtain four verified concurrent data structures**, implemented in C and verified in VST.

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, including lookup, insert, etc. but also some helper functions. A template implements traversal and top-level data structure operations, interacting with the underlying nodes only via the specified functions. 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 p;
      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
```

Figure 1. The lock-coupling search structure template

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 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.

The verification mechanism for search structure mechanisms is concurrent separation logic, and the interface between data structures and templates relies on two key constructs: *flow interfaces* and *logical atomicity*. Flow interfaces [4] are a generic mechanism for capturing the relationship of a single node to a larger data structure, and can be implemented in separation logic using ghost state. The interface provided by a data structure implementation includes a predicate of the form $\mathsf{node}(n, I, C)$, where n is the node itself (i.e., a pointer to the node data structure), I is n's flow interface, and C is n's contribution to the state of the overall data structure, e.g., the set of keys contained in n. Data structure operations such as insert, lookup, and findNext are specified in terms of the node predicate.

Logical atomicity [2] is used to lift a sequential data structure specification to the concurrent setting. A logically atomic triple $\forall a.\ \langle \mathsf{P}_p(a) \rangle\ \mathsf{c}\ \langle \mathsf{Q}_p(a) \rangle$ asserts that the program c atomically updates the abstract data a from a state satisfying P_p to a state satisfying 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

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

saying that insert atomically updates the state of the data structure from $\mathsf{Abs}(m)$ to $\mathsf{Abs}(m[\mathtt{k} \mapsto \mathtt{v}])$, without mentioning the details of either the synchronization mechanism or the underlying data structure implementation. The traverse function for a given template 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 data structure operations, lifting the sequential specifications for insert, lookup, etc. to the concurrent setting.

Krishna et al., following Shasha and Goodman's original work on template algorithms [7], define three templates: lock coupling, give-up, and link. The lock coupling template acquires locks on both parent and child node 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

```
struct node{ int key; int value;
  node* next;
  lock l; }

```

(a) Linked-list node with lock

```
struct node{ int key; int value;
  node* next;
  lock l; int min; int max; }

```

(b) Linked-list node with lock and range

```
struct node{ int key; int value;
  node* left; node* right;
  lock l; }
```

(c) BST node with lock

```
struct node{ int key; int value;
  node* left; node* right;
  lock l; int min; int max; }

```

(d) BST node with lock and range

Figure 2. Four different node implementations

traverse function and use it to implement and verify the top-level data structure operations.

3 Coding Search Structure Templates

Suppose we have implemented four concurrent data structures in C: a linked list and a binary search tree, 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.

This modular approach stands in contrast to Krishna et al. [3], where functions like lockNode and inRange are assumed to be provided by the data structure. This is convenient from one perspective, since the data structure is entirely responsible for implementing and manipulating the node struct, and the template can interact with it only via interface functions; on the other hand, it runs counter to the idea that the data structure implementation is "concurrency-unaware", since it must include locks in its implementation. Furthermore, requiring the data structure to provide

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

```

(a) Linked-list node

```
struct md_entry{ lock l;
  int min; int max; }

```

(b) Give-up metadata

```
struct css{ node* root;
  md_entry* metadata[TABLE_SIZE]; }
```

(c) Top-level concurrent data structure

Figure 3. Modular definition of concurrent search structures

template-specific features like inRange rules out any possibility of modularity, since the template can then only be applied to data structures that implement its specific requirements (and a large enough set of templates may have mutually inconsistent requirements). Instead, we propose that each template should be responsible for providing whatever fields it requires, and each data structure should only implement the fields that make up the core sequential data structure (e.g., key, value, and next).

Writing code that composes in this way is quite difficult in most languages. The most direct analogue is Rust and Scala's traits... 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 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.

Once we have decomposed the data structure type, we can then implement the functions for each component. The data structure defines the local findNext, lookupOp, and insertOp functions as operations on nodes; the template defines traverse, as well as an insertHelper function to maintain metadata on updates, 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 the implementation of both `node` (the data structure) and `md_entry` (the template).

4 Specifying Data Structures and Templates as Interfaces

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

- The interface for data structures is an abstract node predicate, plus sequential Hoare triples for the data structure functions (`findNext`, `insertOp`, `lookupOp`) that characterize their behavior at the level of nodes.
- The interface for templates takes an arbitrary data structure as a parameter, and gives logically atomic triples for the template functions (`traverse`, `insertHelper`) in terms of their effects on an abstract CSS predicate.
- The top-level `insert` and `lookup` functions are specified and verified once and for all, taking both a data structure and a template as a parameter, and providing proofs that the top-level `insert` and `lookup` functions satisfy atomic specifications on CSS:

$$\forall m.\ \langle \mathrm{CSS}(m) \rangle\ \texttt{insert(r, k, v)}\ \langle \mathrm{CSS}(m[\mathrm{k} \mapsto \mathrm{v}]) \rangle$$

$$\forall m.\ \langle \mathrm{CSS}(m) \rangle\ \texttt{lookup(r, k)}\ \langle v.\ \mathrm{CSS}(m) \wedge m(k) = v \rangle$$

In this section, we present the precise 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. As is typical when defining interfaces, the key challenge is to define specifications for the data structure functions that give enough information to verify any template, but are general enough to be satisfied by any data structure. 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.

4.1 Data Structure Interface

The data structure interface is built around an abstract predicate node(n, I, C) representing a node in the data structure, where n is the concrete pointer to the node, I is the flow interface for the node (an abstraction of its position in the data structure), and C is the contents of the note as 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. We then specify the key data structure operations as follows:

`findNext` `insertOp` `lookupOp`

4.2 Template Interface

The template interface includes two abstract predicates: Abs(p, C), which describes the abstract state of the entire concurrent data structure at pointer p as a key-value map C, and md_node(n, I, C), which represents the combination of a node and the metadata attached to it in the template. Prior work does not clearly distinguish between node and md_node—for instance, Krishna et al. [3] assume that the node predicate includes the lock and, in the give-up template, the range for the node—but this distinction is necessary for truly modular specification: the data structure interface should not include concurrency metadata, since it is both concurrency-specific and different for each template instance. Thus, while data structure functions act on nodes, the template functions are defined entirely at the level of md_nodes.

The template operations are specified as follows:

`traverse` `insertHelper` `lookupHelper`

Each top-level operation will be implemented with a combination of `traverse`, which traverses the data structure to find the md_node where a key belongs, and one of the helper functions, which completes the chosen operation given the target md_node. The helper functions are responsible for both performing the actual data structure operation (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.

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. (implementations and proofs here)

In Rocq, these proofs are parameterized by instances of the DataStructure and Template classes. It is precisely this that gives us true modularity: 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. In other words, this guarantees that given m data structure implementations and n template implementations, we can immediately and with no additional effort obtain $m \times n$ verified concurrent data structures.

5 Data Structure Instances: Linked List and Binary Search Tree

6 Template Instances: Give-Up and Lock Coupling

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, allowing us to directly reuse the implementation of flow interfaces from Krishna et al. [3]. Our C code and our Rocq proofs follow a common structure that ensures modularity.

The files `data_structure.h` and `template.h` define the interfaces for data structures and templates respectively, and the corresponding VST files define the typeclasses for these interfaces, including the specification of each interface function. `bst.c` and `list.c` implement the data structure interface, and `give-up.c` and `coupling.c` implement the template interface. Finally, `template.c` implements the top-level operations, relying on the interfaces for data structures and templates but not on any details of their implementation (and the corresponding VST file verifies the operations with the interface typeclasses as parameters). Importantly, 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, simply by compiling `template.c` along with the chosen data structure and template file.

(LoC/LoP numbers for each part) (comparison of proof size to acsys proofs and/or CPP paper?)

8 Conclusion and Future Work

The framework we have presented fulfills the true promise 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 proved without any knowledge of the implementation of the data structure, 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 careful attention to which components belong to the data structure and which to the template, and by working to disentangle the two in both the C implementation and the Rocq specification and proof. Our interfaces are general enough that it should be easy to add more data structures and templates, including concurrency patterns that do not depend on locks (e.g., optimistic concurrency control with fine-grained atomics).

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] 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.

[3] 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

[4] 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

[5] William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 6 (Jan. 2024), 27 pages. https://doi.org/10.1145/3632848

[6] 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* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 60–74. https://doi.org/10.1145/3636501.3636940

[7] Dennis Shasha and Nathan Goodman. 1988. Concurrent Search Structure algorithms. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 53–90.