nekton: a linearizability proof checker

Roland Meyer[1], Anton Opaterny[1(✉)], Thomas Wies[2], and Sebastian Wolff[2]

[1] TU Braunschweig, Braunschweig, Germany
{anton.opaterny,roland.meyer}@tu-bs.de
[2] New York University, New York, USA
{wies,sebastian.wolff}@cs.nyu.edu

Abstract nekton is a new tool for checking linearizability proofs of highly complex concurrent search structures. The tool's unique features are its parametric heap abstraction based on separation logic and the flow framework, and its support for hindsight arguments about future-dependent linearization points. We describe the tool, present a case study, and discuss implementation details.

Keywords: separation logic · proof checker · linearizability · flow framework

1 Introduction

We present nekton, a mostly automated deductive program verifier based on separation logic (SL) [23,27]. The tool is designed to aid the construction of linearizability proofs for complex concurrent search structures. Similar to many other SL-based tools [2,8,14,22,33,33], nekton uses an SMT solver to automate basic SL reasoning. Similar to the original implementation of CIVL [7], it uses non-interference reasoning à la Owicki-Gries [25] to automate thread modularity. What makes nekton stand out among these relatives is its inbuilt support for expressing complex inductive heap invariants using the flow framework [12,13,20] and the ability to (partially) automate complex linearizability arguments that require hindsight reasoning [4,5,15,18,19,24]. Together, these features enable nekton to verify challenging concurrent data structures such as the FEMRS tree [4] with little user guidance.

nekton [17] is derived from the tool plankton [18,19], which shares the same overall goals and features as nekton but strives for full proof automation at the expense of generality. In terms of the trade-off between automation and expressivity, nekton aims to occupy a sweet spot between plankton and general purpose program verifiers. In the following, we discuss nekton's unique features in more detail and explain how it deviates from plankton's design.

The flow framework can be used to express global properties of graph structures in a node-local manner, aiding compositional verification of recursive data structures. The framework is parametric in a *flow domain* which determines what global information about the graph is provided at each node. Various flow domains have been proposed that have shown to be useful in concurrency proofs [11,26]. To simplify proof automation, plankton uses a fixed flow domain that is geared towards verifying functional correctness of search structures. In contrast, nekton is parametric in the flow domain. For instance, it supports custom domains for reasoning about overlayed structures and

other data-structure-specific invariants. This design choice significantly increases the expressivity of the tool at the cost of a mild increase in the annotation burden for the user. For instance, the FEMRS tree case study that we present in this paper relies on a flow domain that is beyond the scope of plankton. In fact, the flow domain is also beyond state-of-the-art abstract interpretation-based verification tools checking linearizability [1]. However, computing relative to a given flow domain is considerably more difficult than computing with a hard-coded one: it requires parametric versions for (1) computing post images, (2) checking entailment, and (3) checking non-interference. Yet, it allows for sufficient automation compared to general user-defined (recursive) predicates as accepted by, e.g., Viper [22] and VeriFast [9].

The second key feature of nekton is its support for *hindsight reasoning*. Intuitively, hindsight arguments rely on statements of the form “if q holds in the current state and p held in some past state, then r must have held in some intermediate state”. Such arguments can greatly simplify the reasoning about complex concurrent algorithms that involve future-dependent linearization points. At a technical level, hindsight reasoning is realized by lifting a state-based separation logic to one defined over computation histories [18,19]. nekton’s support for this style of reasoning goes beyond the simple hindsight rule in [18] but does not yet implement the general *temporal interpolation* rule introduced more recently in [19], which is already supported by plankton.

These features set nekton apart from its competitors. First, it offers more expressivity compared to tools with a higher degree of automation like plankton [18,19], Cave [29–31], and Poling [34]. Second, it’s proofs require less annotation effort than more flexible refinement-proofs for fine-grained concurrency, like those of CIVL [7,10] and Armada [16]. Last, it integrates techniques for proving linearizability, which are missing in industrial grade tools like Anchor [6].

In the remainder of this paper, we provide a high-level overview of the tool (Sect. 2), present a case study (Sect. 3), and discuss implementation details some of which also concern plankton and have not yet been reported on before (Sect. 4).

2 Input

nekton checks the correctness of proof outlines for the linearizability of concurrent data structures. Its distinguishing feature compared to its ancestor plankton is that the heap abstraction is not hard-coded inside the tool, but taken as an input parameter. That is, nekton’s input is a *heap abstraction* and a set of *proof outlines*, one for each function manipulating the data structure state. The heap abstraction defines how the data structure’s heap representation is mapped onto a labeled graph that captures the properties of interest and that can then be reasoned about in separation logic. It also embeds the mechanism for checking linearizability.

nekton works with the recent flow graphs proposed by Krishna et al. [12,13], in their latest formulation due to [18]. Flow graphs augment heap graphs with ghost state. The ghost state can be understood as a certificate formulating global properties of heap graphs in a node-local manner. It takes the form of a so-called flow value that has been propagated through the heap graph and, therefore, brings global information with it. The propagation is like in static analysis, except that we work over heap graphs rather than

control-flow graphs. To give an example, assume we want to express the global property that the heap graph is a tree. A helpful certificate would be the path count, the number of paths from a distinguished root node to the node of interest. It allows us to formulate the tree property node-locally, by saying that the path count is always at most one.

Our first input is a flow domain (M, gen). The parameter $(M, +, 0)$ is a commutative monoid from which we draw the flow values. The propagation needs standard fixed point theory: the natural ordering $a \leq a + b$ for $a, b \in M$ on the monoid should form an ω-complete partial order. We expect the user to specify both $+$ and $\leq$ to avoid the quantifier over the offset in the definition of $\leq$. The parameter gen generates the transfer functions labeling the edges in the heap graph. Transfer functions transform flow values to record information about the global shape. The generator has the type

$$gen : \mathsf{PointerFld} \to (\mathsf{DataFld} \to Data) \to \mathsf{Mon}(M \to M) \ .$$

We assume flow graphs distinguish between pointer fields (PointerFld) and fields that hold data values (DataFld). Flow values are propagated along every pointer field, in a way that depends on the current data values but that does not depend on the target of the field. To see that the data values are important, imagine a node has already been deleted logically but not yet physically from a data structure, as is often the case in lock-free processing. Then the logical deletion would be indicated by a raised flag (a distinguished data field), and we would not forward the current path count. To reason about flow values with SMT solvers, we restrict the allowed types of flow values to

$$M \quad ::= \quad \mathbb{B} \ | \ \mathbb{N} \ | \ \mathbb{P}(\mathbb{B}) \ | \ \mathbb{P}(\mathbb{N}) \ | \ M \times M \ .$$

Flow values are (sets of) Booleans or integers, or products over these base types. When defining a product type, the user has to label each component with a selector allowing to project a tuple onto this component. Importantly, the user can define the addition operation $+$ for the flow monoid freely over the chosen type as long as the definition is expressible within the underlying SMT theory (e.g., for $\mathbb{N}$ one may choose as $+$ the usual addition or the maximum). The tool likewise inherits the assertion language for integers and Booleans that is supported by the SMT solver. There are two more user-defined inputs that are tightly linked to the heap representation.

Linearizability. We establish the linearizability of functions manipulting a data structure with the help of the keyset framework [11,28], which we encode using flows. A crucial problem when proving linearizability are membership queries: we have to determine whether a given key has been in the data structure at some point in time while the function was running. The keyset framework localizes these membership queries from the overall data structure to single nodes. It assigns to each node n a set of keys for which n is responsible, in the sense that n has to answer the membership queries for these keys. This set of keys is n's *keyset*. Imagine we have a singly linked list

$$\texttt{Head} \xrightarrow{(-\infty,\infty)} (n_1, 5) \xrightarrow{[6,\infty)} (n_2, 7)^{\dagger} \xrightarrow{[6,\infty)} (n_3, 10) \xrightarrow{[11,\infty)} \bot \ .$$

The shared pointer `Head` propagates the keys in the interval $(-\infty, \infty)$ as a flow value to node n_1 holding key 5. This set is called n_1's *inset*. The inset of a node n contains all keys k for which a search will reach n. If $k > 5$, the search will proceed to n_2, otherwise

it will stay at n_1. Thus, the keyset of n_1 is $(-\infty, 5]$. That is, if $k \in (-\infty, 5]$, the answer to the membership query is determined by the test $k = 5$. Node n_1 forwards $[6, \infty)$ to the successor node n_2 with key 7. Since n_2 has been logically deleted, indicated by the tombstone †, it cannot answer membership queries: the keyset is empty. Instead, the node forwards its entire inset $[6, \infty)$ to node n_3, which is now responsible for the keyset $[6, 10]$. We speak of a framework because whether a given key k belongs to a node's keyset or whether it is propagated to one of the node's successors is specific to each data structure, but the way in which the linearizability argument for membership queries is localized to individual flow graph nodes is always the same.

In nekton, the user can define $\mathbb{P}(\mathbb{N})$ for sets of keys as (a component in) the flow domain of interest. With parameter gen, they can implement the propagation. We also provide flexibility in the definition of the keyset and membership queries in the form of two predicates rsp (responsible) resp. $cnts$ (contains). To give an example, we would define

$$rsp(x, k) \triangleq k \in x\!\rightarrow\!\mathtt{flow.is} * k \leq x\!\rightarrow\!\mathtt{key} * \neg x\!\rightarrow\!\mathtt{marked}\ .$$

With $x\!\rightarrow\!\mathtt{flow}$, we denote x's flow value. The flow domain is a product, and we refer to the component called is. With $x\!\rightarrow\!\mathtt{key}$ and $x\!\rightarrow\!\mathtt{marked}$ we denote the x's key and marked fields. Formally, the dereference notation is a naming convention for logical variables that refer to values of resources defined in the node-local invariant explained below. Reconsider the example and let $k = 6$. The key belongs to the inset $[6, \infty)$ that n_2 receives from n_1. We discussed that the node's keyset is empty, and indeed $rsp(n_2, 6)$ is false. For n_3, we have $rsp(n_3, 6)$ true. With the predicate rsp in place, we can also refer to $n.\mathtt{keyset}$ in assertions.

For verifying functions with non-fixed linearization points, nekton implements the hindsight principle [24]. Reasoning with that principle goes as follows. We record information about bygone states of the data structure in past predicates $\diamondsuit\, a$. For example, $\diamondsuit(k \in x\!\rightarrow\!\mathtt{flow.is})$ says that the key of interest was in the node's inset at some point while the function was running. Moreover, the assertion about the current state may tell us that the key is smaller than the key held by the node and that the node is not marked now, $k \leq x\!\rightarrow\!\mathtt{key} * \neg n\!\rightarrow\!\mathtt{marked}$. Then the hindsight principle will guarantee that there has been a state in between the two moments where the node still had the key in its inset, the inequality held true, and the node was unmarked. This is $\diamondsuit\, rsp(n, k)$ as defined above. To draw this conclusion, the hindsight principle inspects the interferences the data structure state may experience from concurrently executed functions. In the example, no interferene can unmark a node or change a key. So the predicates encountered in the current state must have held already in the past state when $k \in x\!\rightarrow\!\mathtt{flow.is}$ was true. This form of hindsight reasoning is stronger than the one in [18] but not yet as elaborate as the one in [19]. From a program logic point of view, hindsight reasoning relies on a lifting of state-based to computation-based separation algebras [18].

Implications. Reasoning about automatically generated transfer functions is difficult, in particular when they relate different components in a product flow domain. Consider $\mathbb{N} \times \mathbb{P}(\mathbb{N})$ with the first component the path count at a node and the second component the keyset. The transfer functions will never forget to count a path, and so the following implication will be valid over all heap graphs:

$$(x\!\rightarrow\!\mathtt{flow.pcount}) = 0 \quad\Longrightarrow\quad (x\!\rightarrow\!\mathtt{flow.keyset}) = \varnothing\ . \tag{1}$$

Despite the help of an SMT solver, nekton will fail to establish the validity of such an implication. Therefore, the user may input a set of such formulas that the tool will then take for being valid without further checks. Correctness of a proof is always relative to this set of implications.

2.1 Proof Outlines

A concurrent data structure consists of a set of structs defining the heap elements and a set of functions for manipulating the data structure state. nekton expects as input a proof outline for each such function. The program logic implemented by nekton is an Owicki-Gries system that, besides partial correctness, requires interference freedom of the given proof outlines. The user is expected to give the interferences as input.

The proof outlines accepted by nekton take the form $\{\, pre \,\}\; po \;\{\, post \,\}$ with

$$po \;::=\; \mathtt{com} \;\mid\; \{\, a \,\} \;\mid\; po \,;\, po \;\mid\; (po + po)\,\{\, a \,\} \;\mid\; \{\, a \,\}\, po^{*}\, \{\, a \,\} \;\mid\; \mathtt{atomic}\; po\;.$$

The proof outlines are partial in that intermediary assertions, say in $\mathtt{com}_1 \,;\, \mathtt{com}_2$, may be omitted. nekton will automatically generate the missing information using strongest postconditions. What has to be given are loop invariants and unifying assertions for the different branches of if-then-else statements. Consecutive assertions $\{\, a \,\} \,;\, \{\, b \,\}$ are interpreted as a weaking of a to b.

Programs are given in a dialect of C. Commands are assignments to/from variables and memory locations, allocations, assumptions, and acquires/releases of locks

$$\begin{aligned} \mathtt{com} \;::=\;& \mathtt{p} \coloneqq \mathtt{q} \;\mid\; \mathtt{p{\to}fld} \coloneqq \mathtt{q} \;\mid\; \mathtt{p} \coloneqq \mathtt{q{\to}fld} \;\mid\; \mathtt{p} \coloneqq \mathtt{malloc} \\ &\mid\; \mathtt{assume(cond)} \;\mid\; \mathtt{acquire(p{\to}fld)} \;\mid\; \mathtt{release(p{\to}fld)}\;. \end{aligned}$$

Here, p, q are program variables, fld is a field name, and dereferences are denoted by an arrow. The language is strictly typed with base types void, bool, and int. The latter represents the mathematical integers, i.e., has an infinite domain. We admit the usual conditions over the base types. Using the struct keyword users can specify their own types. In addition, nekton supports syntactic sugar like if-then-else, (do-)while loops, non-recursive macros, break and return statements, assertions, simultaneous assignments, and compare-and-swaps. These can be expressed in terms of the core language in the expected way.

The assertion language is a standard separation logic defined over the base types, heap graphs, and the given flow domain. It has the separating conjunction and classical implication (no magic wand). Our heap model is divided into a local and a shared heap, and we use the box operator $\boxed{a}$ to indicate assertions over the shared state. The shared state is represented by an iterated separating conjunction. Since this conjunction refers to a set of nodes and we want to reason first-order, we handle it implicitly. We let each assertion a in a proof outline stand for $\exists \boldsymbol{x}.\; a * \mathop{\Huge\ast}_{n \in \mathbb{N} \setminus \mathit{Nodes}(a)} \mathit{NInv}(n)$. The iterated separating conjunction is over all nodes that do not occur in a, and asserts a node-local invariant for each of them. The existential quantifier is over all logical variables in the assertion. Keeping it implicit makes the assertions more concise and aids automation.

Node Invariants. nekton expects the node-local invariant $\mathit{NInv}(n)$ as another input. The role of this invariant is to make use of the flow framework and state global

properties of the data structure in a local way. The invariant would say, for instance, that sentinel nodes are never marked. Compared to the implication list, the node-local invariant has the advantage that its claims are actually checked. Technically, the node-local invariant is a separation logic formula that is only allowed to refer to the given node n and its fields. It will often define logical variables like $n \rightarrow \texttt{flow}$ that refer to the entry of the flow field and can be used outside the node-local invariant. These variables are quantified away by $\exists \boldsymbol{x}$ above.

Interferences. Interferences are RGSep actions [32] restricted to the format

$$NInv(x).\ \{\ a\ \} \rightsquigarrow [\texttt{fld}_1, \ldots, \texttt{fld}_n]\{\ b\ \}\ . \tag{2}$$

To give an example, we formulate that a concurrently executed function may mark a node using the action $NInv(x).\ \{\ \neg(x \rightarrow \texttt{marked})\ \} \rightsquigarrow [\texttt{marked}]\{\ x \rightarrow \texttt{marked}\ \}$. An action refers to a single node in the heap graph as described by the above node-local invariant. The action applies if the assertion a evaluates to true, and modifies the node in a way that satisfies b. Like the invariant, the assertions a and b have to be node-local and only refer to the values of x's fields. The assertions may introduce logical variables that are implicitly existentially quantified and whose scope extends over a and b. Such variables allow us to relate the pre- and post-state of the interference. The fields given in the brackets are the ones that may change under the action. If assertion b does not refer to the value of a field that is given in the list, the field may receive arbitrary values. If a field is not named, it is guaranteed to stay unchanged.

3 Case Study

We present a linearizability proof of the FEMRS tree [4] conducted with `nekton`. We omit the data structure's maintenance operation because it leads to flow updates that neither `nekton` nor another state-of-the-art technique aimed at automation can handle. Each node in the tree stores one key and points to up to two child nodes `left` and `right`, storing keys with lower and higher values, respectively. In addition, each node contains two Boolean fields `del` and `rem` for the removal of nodes. This is because the tree distinguishes the logical removal, indicated by the `del` flag, from the physical unlinking of a node, indicated by the `rem` flag. As long as a logically removed node has not been unlinked, it can become part of the tree again. The idea is to save the creation of new nodes for keys that are physically but no longer logically part of the tree. Lastly, every node can be locked.

Figure 1 depicts a possible state of the FEMRS tree. Each node is labeled with its key. Dashed nodes have been logically removed. To prove linearizability, we rely on the keyset framework. The inset flow is used to define the keysets, as explained earlier. The edges in the figure are labeled with the flow they propagate. The transfer functions leading to this propagation stem from

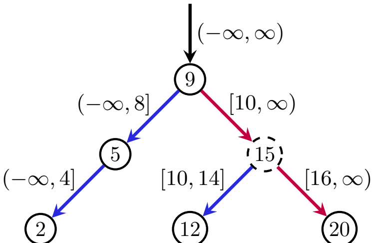

Figure 1: A state of the FEMRS tree.

the following generator *gen*:

$$gen(\texttt{fld}) \;\triangleq\; \lambda f.\, x{\rightarrow}\texttt{del} \;?\; f \;:\; f \setminus \big(\texttt{fld} = \texttt{left} \,?\, [x{\rightarrow}\texttt{key}, \infty) : (-\infty, x{\rightarrow}\texttt{key}]\big)\;.$$

The predicates defining the keyset and membership are

$$\begin{aligned} rsp(x,k) \;&\triangleq\; k \in x{\rightarrow}\texttt{flow.is} * k = x{\rightarrow}\texttt{key} \\ &\quad\vee\, k \in x{\rightarrow}\texttt{flow.is} * k < x{\rightarrow}\texttt{key} * x{\rightarrow}\texttt{left} = \texttt{nil} \\ &\quad\vee\, k \in x{\rightarrow}\texttt{flow.is} * k > x{\rightarrow}\texttt{key} * x{\rightarrow}\texttt{right} = \texttt{nil} \\ cnts(x,k) \;&\triangleq\; k \in x{\rightarrow}\texttt{flow.is} * k = x{\rightarrow}\texttt{key} * \neg x{\rightarrow}\texttt{del}\;. \end{aligned}$$

In the example, $rsp(5,7)$, $rsp(15,15)$, $rsp(20,17)$, $cnts(12,12)$ and more hold.

The set of interferences expresses this: (I1) As long as the lock of the node is not held by the thread under consideration and as long as the node has not been marked unlinked, the child pointers and the (logical and physical) removal flags may change arbitrarily. The proof does not rely, e.g., on the fact that the `rem` flag is raised only once and only when the `del` flag is true. (I2) A lock that is not held by the thread may change arbitrarily. (I3) A node that is being physically unlinked ceases to receive flow. The following `nekton` actions formalize this:

$$NInv(x).\{x{\rightarrow}\texttt{lock} {\neq} \texttt{owned} * \neg x{\rightarrow}\texttt{rem}\} \rightsquigarrow [\texttt{left},\texttt{right},\texttt{del},\texttt{rem}]\{\, \mathit{true} \,\} \tag{I1}$$

$$NInv(x).\{x{\rightarrow}\texttt{lock} {\neq} \texttt{owned}\} \rightsquigarrow [\texttt{lock}]\{\, \mathit{true} \,\} \tag{I2}$$

$$NInv(x).\{x{\rightarrow}\texttt{lock} {\neq} \texttt{owned} * x{\rightarrow}\texttt{flow.is} {\neq} \varnothing * x{\rightarrow}\texttt{rem}\} \rightsquigarrow [\texttt{is}]\{x{\rightarrow}\texttt{flow.is} {=} \varnothing\}. \tag{I3}$$

We prove the linearizability of the functions $\texttt{contains}(k)$, $\texttt{insert}(k)$, and $\texttt{remove}(k)$. All of them call the auxiliary function $\texttt{locate}(k)$, which returns the last edge it traversed during a search for key k. Figure 2 gives the proof outline of `locate`. The proof for the full implementation can be found in [17].

We use a product flow domain $\mathbb{P}(\mathbb{N}) \times \mathbb{N}$. The first component is the inset flow with the generator function discussed above. The second component is the pathcount, whose $gen()$ simply yields the identity for all edges. The benefit of the product flow is that we can prove memory safety on the side, while conducting the linearizability proof.

In the node-local invariant, we introduce logical variables like $x{\rightarrow}\texttt{left}$ to make the proof more readable. We refer to these variables in the generator function. The invariant for the node pointed to by the shared `Root` differs from that of the remaining nodes:

$$\begin{aligned} NInv(x) \;&\triangleq\; x \mapsto \langle\, \texttt{flow} = (x{\rightarrow}\texttt{flow.is},\; x{\rightarrow}\texttt{flow.pcount}), \\ &\qquad\quad \texttt{left} = x{\rightarrow}\texttt{left},\; \texttt{right} = x{\rightarrow}\texttt{right},\; \texttt{key} = x{\rightarrow}\texttt{key}, \\ &\qquad\quad \texttt{lock} = x{\rightarrow}\texttt{lock},\; \texttt{del} = x{\rightarrow}\texttt{del},\; \texttt{rem} = x{\rightarrow}\texttt{rem} \,\rangle \\ &\quad * NInv_{\texttt{all}}(x) * (x = \texttt{Root} \Rightarrow NInv_{\texttt{Root}}(x)) \\ NInv_{\texttt{Root}}(x) \;&\triangleq\; x{\rightarrow}\texttt{key} = -\infty * \neg x{\rightarrow}\texttt{del} * \neg x{\rightarrow}\texttt{rem} \\ &\quad * x{\rightarrow}\texttt{flow.is} = (-\infty, \infty) * x{\rightarrow}\texttt{flow.pcount} = 1 \\ NInv_{\texttt{all}}(x) \;&\triangleq\; (\neg x{\rightarrow}\texttt{rem} \Rightarrow x{\rightarrow}\texttt{key} \in x{\rightarrow}\texttt{flow.is}) * x{\rightarrow}\texttt{flow.pcount} < 3 \\ &\quad * (x{\rightarrow}\texttt{rem} \Rightarrow x{\rightarrow}\texttt{del}) * (x{\rightarrow}\texttt{left} = x{\rightarrow}\texttt{right} \Rightarrow x{\rightarrow}\texttt{left} = \texttt{nil})\;. \end{aligned}$$

```
1  { −∞ < k < ∞ ∗ NInv(Root) }
2  inline <Node*, Node*> locate(data_t k) {
3    Node* p, c; p = Root; c = Root;
4    { p = c = Root ∗ NInv(Root) ∗ ◇[NInv(c) ∗ k ∈ c→flow.is] ∗ −∞ < k < ∞ }
5    do { NInv(p) ∗ (NInv(c) ∗ ◇[NInv(c) ∗ k ∈ c→flow.is] ∨ ◇[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
6    { { NInv(p) ∗ NInv(c) ∗ ◇[NInv(c) ∗ k ∈ c→flow.is] ∗ c→key ≠ k }
7      p = c;
8      if (p→key < k) {
9        assert(p→right = nil || p→right ≠ nil);
10       c = p→right;
11       { NInv(p) ∗ (NInv(c) ∨ ◇[NInv(p) ∗ p→right = nil] ∗ c = nil)
             ∗ ◇[NInv(p) ∗ k ∈ p→flow.is] ∗ p→right = c ∗ p→key < k }
12       { NInv(p) ∗ (NInv(c) ∗ ◇[NInv(c) ∗ k ∈ c→flow.is] ∨ ◇[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
13     } else { /* symmetric to 'then' branch */ }
14     { NInv(p) ∗ (NInv(c) ∗ ◇[NInv(c) ∗ k ∈ c→flow.is] ∨ ◇[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
15   } while (c ≠ nil && c→key ≠ k);
16   { NInv(p) ∗ (NInv(c) ∗ ◇[NInv(c) ∗ k ∈ c→flow.is] ∗ c→key = k
                                   ∨ ◇[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
17   return <p, c>;
18 }
```

Figure 2: Proof outline for `locate` as verified by `nekton`.

The node-local invariant makes the expected claims. The root has key $-\infty$, is neither logically deleted nor unlinked, has as incoming keys $(-\infty, \infty)$ and the pathcount is 1. These flow values are established by the data structure's initialization function using an auxiliary edge with an appropriate generator. For all nodes, we have that their key is in the inflow, provided the node has not yet been unlinked, the path count is at most 3, a node has to be first logically deleted before it can be unlinked, and the only case in which the left and the right child can coincide is when they are both the null pointer. We treat `nil` as a node outside the set of nodes $\mathbb{N}$. This in particular means the node-local invariant does not apply to it. It will follow from the definition of the generator function that the keysets are disjoint. We do not need to state this in the invariant as it is only important when interpreting the verification results.

The assertion on line 9 helps our implication engine, which is designed for conjunctive assertions, deal with the disjunctions.

We explain the implication between Lines 11 and 12. It starts with the assertion $\{\ NInv(\mathtt{p}) * NInv(\mathtt{c}) * \diamondsuit[NInv(\mathtt{p}) * k \in \mathtt{p{\to}flow.is}] * \mathtt{p{\to}right} = \mathtt{c} * \mathtt{p{\to}key} < k\ \}$. To apply the hindsight principle, we derive the following guarantees from the set of interferences. A node's key is never changed. The only way a node's inset can shrink is by unlinking, after which its `left` and `right` pointers are no longer changed. The right child of `p` is not `nil` in the current state. From this information, the hindsight principle concludes $\{\ \diamondsuit[NInv(\mathtt{p}) * NInv(\mathtt{c}) * k \in \mathtt{p{\to}flow.is} * \mathtt{p{\to}key} < k * \mathtt{p{\to}right} = \mathtt{c}]\ \}$. Together with the definition of the transfer functions labeling the edges, this asser-

tion yields $\{\, \diamondsuit[NInv(\mathtt{c}) * k \in c \rightarrow \mathtt{flow.is}] \,\}$. Another hindsight application starts with $\{\, NInv(\mathtt{p}) * \mathtt{c} = \mathtt{nil} * \diamondsuit[\, NInv(\mathtt{p}) * k \in \mathtt{p} \rightarrow \mathtt{flow.is} \,] * \mathtt{p} \rightarrow \mathtt{right} = \mathtt{c} * \mathtt{p} \rightarrow \mathtt{key} < k \,\}$ and moves the facts known in the current state into the past predicate. The definition of $rsp(x, k)$ then yields $\{\diamondsuit[NInv(\mathtt{p}) * rsp(\mathtt{p}, k)]\}$.

The full proof consists of 99 lines of code, 48 lines of assertions to prove them linearizable, and 56 lines of definitions for the flow domain, interferences, and invariants. nekton takes $45s$ to verify the proof's correctness on an Apple M1 Pro.

4 Correctness and Implementation

nekton checks that the verification conditions generated from the given proof outlines hold and that the assertions are interference-free. The program logic from [18,19] then gives the following semantic guarantee: no matter how many client threads execute the data structure functions, partial correctness holds. That is, if a function is executed from a state satisfying the precondition and terminates, it must have reached a state in which the postcondition held true. Termination itself is not guaranteed. The postcondition will relate the function's return value to a statement about membership of the given key in the data structure, and the keyset framework will allow us to conclude linearizability from this relation. The verification conditions will in particular make sure the node invariant is maintained. We discuss the actual checks.

The first step is to derive and check verification conditions for all commands com. If the command is surrounded by assertions, $\{\, p \,\}$; com; $\{\, q \,\}$, the verification condition is $sp(p, \mathtt{com}) \models q$, the strongest postcondition sp of p under com entails q. If the assertion $\{\, q \,\}$ is not given, nekton completes the given proof by using $q = sp(p, \mathtt{com})$. The verification conditions for loops are similar. For two consecutive assertions $\{\, p \,\}$; $\{\, q \,\}$, as they occur for example at the end of a branch, the verification condition is $p \models q$.

The second step is to check that the assertions $\{\, p \,\}$ and $\{\, q \,\}$ in the proof are interference-free, i.e., cannot be invalidated by the actions of other threads.

Finally, nekton checks that the interferences given by the user cover the actual interferences of the program. We review the above steps in more detail.

Strongest Postconditions. The computation of the strongest postcondition follows the standard axioms for separation logic [23]. However, they do not deal with the flow which may not only be directly modified by com but also indirectly by an update elsewhere. To deal with such indirect updates, nekton computes a *footprint fp*: a subset of the heap locations that the standard axioms require plus those locations whose flow changes due to com. The footprint yields a decomposition $p = fp * f$ of predicate p, where f is a frame that is not affected by the update. From this decomposition, we compute the strongest postcondition as $sp(p, \mathtt{com}) = sp(fp, \mathtt{com}) * f$, using the frame rule. Actually, nekton also shows that the update maintains the node invariant, which only requires a check for $sp(fp, \mathtt{com})$.

For fp to be a footprint wrt. com, all nodes outside fp should receive the same flow from $sp(fp, \mathtt{com})$ as from fp. This holds if fp and $sp(fp, \mathtt{com})$ induce the same flow transformer function [20]. To determine a footprint, nekton takes a strategy that is justified by lock-free programming [18]. Starting from the updated nodes, it gathers a small (fixed) set of locations that forms an acyclic subgraph. Acyclicity guarantees that

fp and $sp(fp,\ \texttt{com})$ have the same transformer iff they agree on the transformation along all paths: if n belongs to fp and $n{\rightarrow}\texttt{fld}$ does not, then $n{\rightarrow}\texttt{fld}$ must point to the same location and transform inflows to outflows in the same way in fp and in $sp(fp,\ \texttt{com})$.

The strongest postcondition above is for state-based reasoning. For predicates over computations, which have state and past predicates, we use the following observation: past predicates are never invalidated by commands. This allows us to just copy them to the postcondition: $sp(p * \diamond q,\ \texttt{com}) = sp(p,\ \texttt{com}) * \diamond p * \diamond q$. Note that we add the precondition as a new past predicate. Moreover, we may add *new* past predicates derived by hindsight arguments. As these derived past predicates are implied by the postcondition, they formally do not strengthen the assertion, but of course help the tool.

Hindsight Reasoning. Recall from Section 2 that hindsight reasoning draws conclusions of the form $\diamond p * q \Rightarrow \diamond r$: every computation from a p-state must inevitably transition through r in order to reach q. In `nekton`, p and q are restricted to node-local predicates in the sense defined above, and r is fixed to $p \wedge q$.

To prove the implication, assume it did not hold. Then there is a computation where p is invalidated before q is established. This is covered by the interference: there is an action act_p invalidating p and an action act_q establishing q. Let act_p and act_q be $NInv(n).\,\{\ o_p\ \} \rightsquigarrow [\dots]\{\ \dots\}$ resp. $NInv(n).\,\{\ o_q\ \} \rightsquigarrow [\dots]\{\ \dots\}$. There is (always) a decomposition $o_p = o_p^i * o_p^m$ such that o_p^i is immutable. Immutability holds if o_p^i is shared and interference-free. Consequently, o_p^i must still hold when q is established. Now, we check if o_p^i and o_q are contradictory, $o_p^i \wedge o_q \models \mathit{false}$. If so, act_q is not enabled after act_p. This, in turn, means q cannot be established after p is invalidated—the computation cannot exist. `nekton` draws the hindsight conclusion if it can prove the contradiction for all pairs act_p, act_q of interferences that invalidate p and establish q.

Entailment. Our assertions $p * \bigast_{i\in I} \diamond p_i$ consist of a predicate p for the current state and a set of past predicates $\diamond p_i$ tracking information about the computation. We have $p * \bigast_{i\in I} \diamond p_i \models q * \bigast_{j\in J} \diamond q_j$, if $p \models q$ and $\forall j\, \exists i.\ \diamond p_i \models \diamond q_j$. To show $\diamond p_i \models \diamond q_j$, we rely on the algorithm for state predicates and prove $p_i \models q_j$.

Entailment checks $p \models q$ between state predicates decompose into reasoning about resources and reasoning about logically pure facts. The latter degenerates to an implication in classical logic: `nekton` uses a straightforward encoding into SMT and discharges it with Z3 [21]. For reasoning about resources, `nekton` implements a custom matching procedure to correlate the resources in p and q. The procedure is guided by the program variables x: if the value of x is a in p and b in q, then a and b are matched, meaning b is renamed to a. The procedure then continues to match the fields of already matched addresses. Finally, `nekton` checks syntactically if all the resources in q occur in p.

If `nekton` fails to prove an implication, it consults the implication list. It takes the implications as they are, and does not try to embed them into a context as would be justified by congruence. `nekton` does not track the precise implications it has used.

Interference Freedom. A state predicate p is interference-free wrt. act of the form $NInv(n).\,\{\ r\ \} \rightsquigarrow [\texttt{fld}_1, \dots, \texttt{fld}_n]\{\ o\ \}$, if the strongest postcondition of p under act entails p itself, $sp(p,\ act) \models p$. Towards $sp(p,\ act)$, let $p = NInv(x) * q$, meaning x is an accessible location. Applying act to x in p acts like an assignment to the fields

such that their new values satisfy o. The strongest postcondition for this is standard [3]:

$$sp_x(p,\ act) \;\triangleq\; o[n\backslash x] * \exists y_1 \cdots y_n.\ (p * r[n\backslash x])[x{\rightarrow}\mathtt{fld}_1\backslash y_1, \ldots, x{\rightarrow}\mathtt{fld}_n\backslash y_n]\ .$$

We strengthen p with the precondition r of act to make sure the action is enabled. We use $r[n\backslash x]$ for r with n replaced by x, meaning we instantiate r to location x. We replace the old values of the updated fields with fresh quantified variables and add the fields' new valuation $o[n\backslash x]$. Then, the strongest postcondition $sp(p,\ act)$ applies $sp_x(p,\ act)$ to all locations x in p.

Interference Coverage. Consider $act_1 = \mathit{NInv}(x).\{\, p \,\} \rightsquigarrow [\mathtt{fld}_1, \ldots, \mathtt{fld}_n]\{\, q \,\}$ and $act_2 = \mathit{NInv}(x).\{\, r \,\} \rightsquigarrow [\mathtt{fld}'_1, \ldots, \mathtt{fld}'_m]\{\, o \,\}$. We say that act_1 covers act_2 if act_1 can produce all updates induced by act_2. This is the case if $r \models p$, $o \models q$, and $\{\, \mathtt{fld}'_1, \ldots, \mathtt{fld}'_m \,\} \subseteq \{\, \mathtt{fld}_1, \ldots, \mathtt{fld}_n \,\}$. It remains to extract the actual interferences of the program and check if they are covered by the user-specified ones. The extraction is done while computing the strongest postcondition sp: the computed footprints *fp* and $sp(\mathit{fp},\ \mathtt{com})$ from above reveal the updated fields as well as the pre- and post-states.

Flow Encoding. The flow monoid is not yet parsed from the user input but defined programmatically in nekton. The transfer function generator is parsed. nekton has five flow domains predefined, including path counting and keysets, which are easy to extend. nekton does not check whether the flow monoid is indeed a monoid and satisfies the requirements of an ω-cpo, nor whether $\leq$ coincides with the natural partial order.

The main task in dealing with a parametric rather than fixed flow domain is to encode predicates involving the flow into SMT formulas. This encoding is then used to implement the aforementioned components for strongest postconditions, hindsight, entailment, and interferences. Devising the encoding is challenging because it requires a representation of flow values that is sufficiently expressive to define relevant flow domains, yet sufficiently restricted to have efficient SMT solver support (we use Z3 [21]). With the input format described in Sect. 2, we encode flows using the theory of integers and uninterpreted functions.

Limitations. For the future, we see several directions for extensions of our current implementation: (i) a parser for flow monoids rather than a programmatic interface, (ii) support for *partial* annotations that are automatically completed by nekton, (iii) the ability to prove atomic triples instead of just linearizability for sets, and (iv) more helpful error messages or counterexamples to guide the proof-writing user.

Data Availability Statement

The nekton tool and case studies generated and/or analysed in the present paper are available in the Zenodo repository [17], https://doi.org/10.5281/zenodo.7931936.

Acknowledgments

This work was funded in part by an Amazon Research Award. The work was also supported by the DFG project *EDS@SYN: Effective Denotational Semantics for Synthesis*. The fourth author is supported by a Junior Fellowship from the Simons Foundation (855328, SW).

References

1. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Fragment abstraction for concurrent shape analysis. In: ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 442–471. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_16
2. Blom, S., Huisman, M.: The vercors tool for verification of concurrent programs. In: FM. Lecture Notes in Computer Science, vol. 8442, pp. 127–131. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_9
3. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science, Springer (1990). https://doi.org/10.1007/978-1-4612-3228-5
4. Feldman, Y.M.Y., Enea, C., Morrison, A., Rinetzky, N., Shoham, S.: Order out of chaos: Proving linearizability using local views. In: DISC. LIPIcs, vol. 121, pp. 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.DISC.2018.23
5. Feldman, Y.M.Y., Khyzha, A., Enea, C., Morrison, A., Nanevski, A., Rinetzky, N., Shoham, S.: Proving highly-concurrent traversals correct. Proc. ACM Program. Lang. **4**(OOPSLA), 128:1–128:29 (2020). https://doi.org/10.1145/3428196
6. Flanagan, C., Freund, S.N.: The anchor verifier for blocking and non-blocking concurrent software. Proc. ACM Program. Lang. **4**(OOPSLA), 156:1–156:29 (2020). https://doi.org/10.1145/3428224
7. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: CAV (2). Lecture Notes in Computer Science, vol. 9207, pp. 449–465. Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_26
8. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and java. In: NASA Formal Methods. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
9. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: APLAS. Lecture Notes in Computer Science, vol. 6461, pp. 304–311. Springer (2010). https://doi.org/10.1007/978-3-642-17164-2_21
10. Kragl, B., Qadeer, S.: Layered concurrent programs. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 79–102. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_5
11. Krishna, S., Patel, N., Shasha, D.E., Wies, T.: Verifying concurrent search structure templates. In: PLDI. pp. 181–196. ACM (2020). https://doi.org/10.1145/3385412.3386029
12. Krishna, S., Shasha, D.E., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL), 37:1–37:31 (2018). https://doi.org/10.1145/3158125
13. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 308–335. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_12
14. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with chalice. In: FOSAD. Lecture Notes in Computer Science, vol. 5705, pp. 195–222. Springer (2009). https://doi.org/10.1007/978-3-642-03829-7_7
15. Lev-Ari, K., Chockler, G.V., Keidar, I.: A constructive approach for proving data structures' linearizability. In: DISC. Lecture Notes in Computer Science, vol. 9363, pp. 356–370. Springer (2015). https://doi.org/10.1007/978-3-662-48653-5_24

16. Lorch, J.R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J.R., Zhao, X.: Armada: low-effort verification of high-performance concurrent programs. In: PLDI. pp. 197–210. ACM (2020). https://doi.org/10.1145/3385412.3385971
17. Meyer, R., Opaterny, A., Wies, T., Wolff, S.: Artifact for "nekton: a linearizability proof checker" (Apr 2023). https://doi.org/10.5281/zenodo.7931936
18. Meyer, R., Wies, T., Wolff, S.: A concurrent program logic with a future and history. Proc. ACM Program. Lang. **6**(OOPSLA2), 1378–1407 (2022). https://doi.org/10.1145/3563337
19. Meyer, R., Wies, T., Wolff, S.: Embedding hindsight reasoning in separation logic. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591296
20. Meyer, R., Wies, T., Wolff, S.: Make flows small again: Revisiting the flow framework. In: TACAS (1). Lecture Notes in Computer Science, vol. 13993, pp. 628–646. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_32
21. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
23. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001). https://doi.org/10.1007/3-540-44802-0_1
24. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC. pp. 85–94. ACM (2010). https://doi.org/10.1145/1835698.1835722
25. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
26. Patel, N., Krishna, S., Shasha, D.E., Wies, T.: Verifying concurrent multicopy search structures. Proc. ACM Program. Lang. **5**(OOPSLA), 1–32 (2021). https://doi.org/10.1145/3485490
27. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
28. Shasha, D.E., Goodman, N.: Concurrent search structure algorithms. ACM Trans. Database Syst. **13**(1), 53–90 (1988). https://doi.org/10.1145/42201.42204
29. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI. Lecture Notes in Computer Science, vol. 5403, pp. 335–348. Springer (2009). https://doi.org/10.1007/978-3-540-93900-9_27
30. Vafeiadis, V.: Automatically proving linearizability. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 450–464. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_40
31. Vafeiadis, V.: Rgsep action inference. In: VMCAI. Lecture Notes in Computer Science, vol. 5944, pp. 345–361. Springer (2010). https://doi.org/10.1007/978-3-642-11319-2_25
32. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. Lecture Notes in Computer Science, vol. 4703, pp. 256–271. Springer (2007). https://doi.org/10.1007/978-3-540-74407-8_18
33. Wolf, F.A., Schwerhoff, M., Müller, P.: Concise outlines for a complex logic: A proof outline checker for tada. In: FM. Lecture Notes in Computer Science, vol. 13047, pp. 407–426. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_22

34. Zhu, H., Petri, G., Jagannathan, S.: Poling: SMT aided linearizability proofs. In: CAV (2). Lecture Notes in Computer Science, vol. 9207, pp. 3–19. Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_1