A Concurrent Program Logic with a Future and History

ROLAND MEYER, TU Braunschweig, Germany
THOMAS WIES, New York University, USA
SEBASTIAN WOLFF, New York University, USA

Verifying fine-grained optimistic concurrent programs remains an open problem. Modern program logics provide abstraction mechanisms and compositional reasoning principles to deal with the inherent complexity. However, their use is mostly confined to pencil-and-paper or mechanized proofs. We devise a new separation logic geared towards the lacking automation. While local reasoning is known to be crucial for automation, we are the first to show how to retain this locality for (i) reasoning about inductive properties without the need for ghost code, and (ii) reasoning about computation histories in hindsight. We implemented our new logic in a tool and used it to automatically verify challenging concurrent search structures that require inductive properties and hindsight reasoning, such as the Harris set.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Hoare logic**; **Automated reasoning**; *Program verification*; Programming logic.

Additional Key Words and Phrases: Linearizability, Non-blocking Data Structures, Harris Set

ACM Reference Format:
Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 174 (October 2022), 30 pages. https://doi.org/10.1145/3563337

1 INTRODUCTION

Concurrency comes at a cost, at least, in terms of increased effort when verifying program correctness. There has been a proliferation of concurrent program logics that provide an arsenal of reasoning techniques to address this challenge [Bell et al. 2010; Delbianco et al. 2017; Elmas et al. 2010; Fu et al. 2010; Gotsman et al. 2013; Gu et al. 2018; Hemed et al. 2015; Jung et al. 2018; Liang and Feng 2013; Manna and Pnueli 1995; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and Parkinson 2007]. In addition, a number of general approaches have been developed to help structure the high-level proof argument [Feldman et al. 2018, 2020; Kragl et al. 2020; O'Hearn et al. 2010; Shasha and Goodman 1988]. However, the use of these techniques has been mostly confined to manual proofs done on paper, or mechanized proofs constructed in interactive proof assistants. We distill from these works a concurrent separation logic suitable for automating the construction of local correctness proofs for highly concurrent data structures. We focus on concurrent search structures (sets and maps indexed by keys), but the developed techniques apply more broadly. Our guiding principle is to perform all inductive reasoning, both in time and space, in lock-step with the program execution. The reasoning about inductive properties of graph structures and computation histories is relegated to the meta-theory of the logic by choosing appropriate semantic models.

Running Example. We motivate our work using the Harris non-blocking set data structure [Harris 2001], which we will also use as a running example throughout the paper.

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Thomas Wies, New York University, USA, wies@cs.nyu.edu; Sebastian Wolff, New York University, USA, sebastian.wolff@cs.nyu.edu.

2475-1421/2022/10-ART174
https://doi.org/10.1145/3563337

```
 1  struct N = { val key: K; var next: N }

 2  val tail = new N { key = ∞; next = tail }
 3  val head = new N { key = −∞; next = tail }

 4  procedure traverse(k: K, l: N, ln: N, t: N): (N*N*N) {
 5    val tn = t.next
 6    val tmark = is_marked(tn)
 7    if (tmark) return traverse(k, l, ln, tn)
 8    else if (t.key < k) return traverse(k, t, tn, tn)
 9    else return (l, ln, t)
10  }

11  procedure find(k: K): N * N {
12    val hn = head.next
13    val l, ln, r := traverse(k, head, hn, hn)
14    if ((ln == r || CAS(l.next, ln, r))
15        && !is_marked(r.next)) return (l, r)
16    else return find(k)
17  }

18  procedure search(k: K) : Bool {
19    val _, r = find(k)
20    return r.key == k
21  }
```

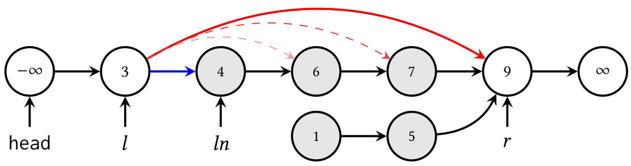

Fig. 1. The Harris [2001] set algorithm. The lower half shows a state of the Harris set containing keys $\{3, 9\}$. Nodes are labeled with the value of their `key` field. Edges indicate `next` pointers. Marked nodes are shaded gray. The blue edge between the nodes l and ln represents the state of l before the `CAS` and the red edge between l and r the state after the `CAS` from Line 14. Dashed edges represent the hypothetical update chunks that inductively capture the effect of the `CAS`.

We assume a garbage-collected programming language, supporting (first-order) recursive functions, product types, and mutable heap-allocated structs. The language further provides a *compare-and-set* operation, `CAS(`x`.f,`o`,`n`)`, that atomically sets field `f` of x to n and returns `true` if `f`'s current value is o, or otherwise returns `false` leaving x`.f` unchanged.

Harris' algorithm implements a set data structure that takes elements from a totally ordered type `K` of keys and provides operations for concurrently searching, inserting, and deleting a given *operation key* `k`. We focus on the `search` operation shown in Figure 1. The data structure is represented as a linked list consisting of nodes implemented by the struct type `N`. Each node stores a `key` and a `next` pointer to the successor node in the list. A potential state of the data structure is illustrated in Figure 1. The algorithm maintains several important invariants. First, the list is strictly sorted by the keys in increasing order and has a sentinel head node, pointed to by the immutable shared pointer `head`. The key of the head node is $-\infty$. Likewise, there is a sentinel tail node with key ∞. We assume $-\infty < k < \infty$ for all operation keys k. To support concurrent insertions and deletions without lock-based synchronization, a node that is to be removed from the list is first *marked* to indicate that it has been logically deleted before it is physically unlinked. Node marking is implemented by *bit-stealing* on the `next` pointers. We abstract from the involved low-level bit-masking using the function `is_marked`. A call `is_marked(`p`)` returns `true` iff the mark bit of pointer p is set. We say that a node x is marked if `is_marked(`x`.next)` returns `true`. The sentinel nodes are never marked.

The workhorse of the algorithm is the function `find`. It takes an operation key k and returns a pair of nodes (l, r) such that l.key $< k \leq r$.key, and the following held true at a single point in time during `find`'s execution: r was unmarked and l's direct successor, and l was reachable from `head`. All client-facing functions such as `search` then use `find`.

Contributions. Recall that in order to prove linearizability of a concurrent data structure, one has to show that each of the data structure's operations takes effect instantaneously at some time point between its invocation and return, the *linearization point*, and behaves according to its sequential specification [Herlihy and Wing 1990]. The Harris set exhibits two key challenges in automating the linearizability proof that are common to non-blocking data structures and that we aim to address in our work.

The first challenge is that linearization points may not be statically fixed, but instead depend on the interference of concurrent operations performed by other threads. We discuss this issue using `search`, whose sequential specification says that the return value is `true` iff k is present in the data structure. Consider a thread T executing a call `search(8)`. Figure 1 shows a possible intermediate state of the list observed right after the successful execution of the `CAS` on Line 14 during T's execution of `search`. This is also the linearization point of `search` for T, because the `CAS` guarantees that if value 8 is present in the data structure, it must be the key of node r. Hence, if the check on r's mark bit on Line 15 still succeeds, then the test on Line 20 will produce the correct return value for the `search` relative to the abstract state of the data structure at the linearization point. However, in an alternative execution, a concurrently executing `delete(9)` may mark r before T executes Line 15, but after T executes the `CAS`. In this situation, T will abort and restart causing its linearization point to be delayed. Thus, the correct linearization point is only known in hindsight [O'Hearn et al. 2010] at a later point in the execution when all interferences have been observed. As a consequence, the proof must track information about earlier states in the execution history to enable reasoning about linearization points that already happened in the past.

Our first contribution is a lightweight embedding of computation histories into separation logic [O'Hearn et al. 2001] that supports local proofs using hindsight arguments, but without having to perform explicit induction over computation histories.

The second challenge is that the proof needs to reason about maintenance operations that affect an unbounded heap region. The procedure `traverse` guarantees that all nodes between l and r are marked. The `CAS` on Line 14 then unlinks the segment of marked nodes between l and r from the structure making r the direct successor of l. This step is depicted in Figure 1. The blue edge between l and ln refers to the pre state of the `CAS` and the red edge between l and r to the post state. A traditional automated analysis needs to infer the precise inductive shape invariant of the traversed region (e.g. a recursive predicate stating that it is a list segment of marked nodes). Then, at the point where the segment is unlinked, it has to infer that the global data structure invariant is maintained. This involves an inductive proof argument, and the analysis needs to rediscover how this induction relates to the invariant of the traversal.

Our second contribution is a mechanism for reasoning about such updates with non-local effects. The idea is to compose these updates out of *ghost update chunks*. This is illustrated in Figure 1, where the effect of the `CAS` is composed out of simpler updates that move the edge from l towards r one node at a time (indicated by the dashed red edges). One only needs to reason about four nodes to prove that the edge can be moved forward by one node. We refer to a correctness statement of such a ghost update chunk as a *future*. The crux is to construct these futures during the traversal of the marked segment, i.e., in lock-step with the program execution. This avoids the need for explicit inductive reasoning at the point where the `CAS` takes effect. When proving the future for unlinking a single traversed node t, we directly apply interference-free facts learned during the traversal, e.g., that t must be marked and can therefore be unlinked. We call this mechanism accounting. The final future can then be invoked on Line 14 to prove the correctness of the `CAS`.

The focus of the paper is on the development of the new program logic rather than algorithmic details on efficient automatic proof search. However, we have implemented a prototype tool called `plankton` [Meyer et al. 2022a] that uses the logic to verify linearizability of highly concurrent search structures automatically, provided an appropriate structural invariant is given by the user. The tool's implementation follows a standard abstract interpretation approach [Cousot and Cousot 1977].

We have successfully applied the tool to verify several fine-grained non-blocking and lock-based concurrent set implementations, including: Harris set [Harris 2001], Michael set [Michael 2002], Vechev and Yahav CAS set [Vechev and Yahav 2008, Figure 2], ORVYY set [O'Hearn et al. 2010], and the Lazy set [Heller et al. 2005]. All these benchmarks require hindsight reasoning. To our knowledge, `plankton` is the first tool that automates hindsight reasoning for such a variety of benchmarks. The Harris set additionally requires futures that are also automated in plankton. With this, `plankton` is the first tool that can automatically verify the Harris set algorithm.

A companion technical report containing additional details is available as [Meyer et al. 2022b].

2 OVERVIEW

We aim for a proof strategy that is compatible with local reasoning principles and agnostic to the detailed invariants of the specific data structure under consideration. In particular, we want to avoid proof arguments that devolve into explicit reasoning about heap reachability or other inductive heap properties, which tend to be difficult to automate.

Our strategy builds on the *keyset framework* [Krishna et al. 2020a; Shasha and Goodman 1988] for designing and verifying concurrent search structures. In this framework, the data structure's heap graph is abstracted by a mathematical graph (N, E) where each node $x \in N$ is labeled by its local contents, a set of keys $\mathsf{C}(x)$. The abstract state of the data structure $\mathsf{C}(N)$ is then given by the union of all node-local contents. Moreover, each node x has an associated set of keys $\mathsf{KS}(x)$ called its *keyset*. The keysets are defined inductively over the graph such that the following *keyset invariants* are maintained: (1) the keysets of all nodes partition the set of all keys, and (2) for all nodes x, $\mathsf{C}(x) \subseteq \mathsf{KS}(x)$. For the Harris set, we define $\mathsf{C}(x) \triangleq \mathit{mark}(x)\ ?\ \emptyset : \{\mathit{key}(x)\}$ and let $\mathsf{KS}(x)$ be the empty set if x is not reachable from `head` and otherwise the interval $(\mathit{key}(y), \mathit{key}(x)]$ where y is the predecessor of x in the list (cf. Figure 2). Here, we denote by $\mathit{key}(x)$ the value of field `key` in a given state, and similarly for $\mathit{mark}(x)$. Throughout the rest of the paper, we will follow this convention of naming variables in a way that reflects the field dereferencing mechanism.

The keyset invariants ensure that for any node $x \in N$ and key k we have

$$k \in \mathsf{KS}(x) \ \Rightarrow\ \big(k \in \mathsf{C}(N) \Leftrightarrow k \in \mathsf{C}(x)\big)\ .$$

This property allows us to reduce the correctness of an insertion, deletion, and search for k from the global abstract state $\mathsf{C}(N)$ to x's local contents $\mathsf{C}(x)$, provided we can show $k \in \mathsf{KS}(x)$. For example, suppose that a concurrent invocation of `search`(k) returns `false`. To prove that this invocation is linearizable, it suffices to show that there exists a node x such that both $k \in \mathsf{KS}(x)$ and $k \notin \mathsf{C}(x)$ were true at the same point in time during `search`(k)'s execution. We refer to x as the decisive node of the operation. The point in time where the two facts about x hold is the linearization point.

The ingredients for the linearizability proofs are thus (i) defining the keysets for the data structure at hand, (ii) proving that the keyset invariants are maintained by the data structure's operations, and (iii) identifying the linearization point and decisive node x for an operation on key k by establishing the relevant facts about k's membership in the keyset and contents of x.

Our contributions focus on the automation of (ii) and (iii). While we do not automate (i), the keyset definitions follow general principles and can be reused across many data structures [Shasha and Goodman 1988] (e.g., we use the same definition for all the list-based set implementations considered in our evaluation, cf. §8). In the remainder of this section, we provide a high-level overview of the reasoning principles that underlie our new program logic and enable proof automation.

Automating History Reasoning. We start with the linearizability argument. Consider a concurrent execution of `search`(k) that returns value b. The decisive node of `search` is always the node r returned by the call to `find`. The proof thus needs to establish that at some point during the execution, $k \in \mathsf{KS}(r)$ and $b \Leftrightarrow (k \in \mathsf{C}(r))$ were true. The issue is that when we reach the

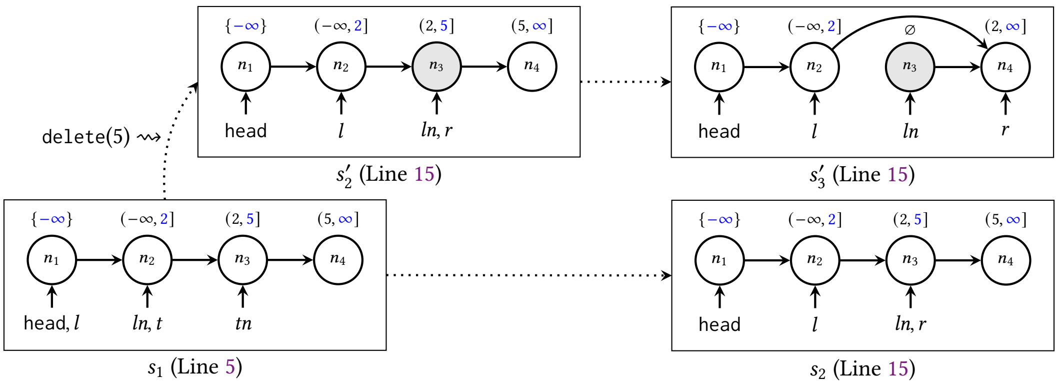

Fig. 2. Some states observed during two executions of search(5) on a Harris set that initially contains the keys 2 and 5 and no marked nodes. Each node in a state is labeled above by its keyset. If the node is reachable from head, the right bound of the keyset interval indicates the key stored in the node (highlighted in blue).

corresponding point in the execution during proof construction, we may not be able to linearize the operation right away because the decisive node and linearization point depend on the modifications that may still be done by other threads before search returns. We can thus only linearize the execution in hindsight, once all the relevant interferences have been observed.

To illustrate this point, consider the scenario depicted in Figure 2. It shows intermediate states of two executions of search(5) on a Harris set that initially contains the keys $\{2, 5\}$. The two executions agree up to the point when state s_1 is reached after execution of Line 5 in the first call to traverse (i.e., after n_2's next pointer and mark bit have been read). The execution depicted on the bottom proceeds without interference to state s_2 at the beginning of Line 15 and will return $b = \mathit{true}$. Here, the decisive node is n_3 and the linearization point is s_1. Note that both $5 \in \mathsf{KS}(n_3)$ and $5 \in \mathsf{C}(n_3)$ hold in s_1. On the other hand, the execution depicted on the top of Figure 2 is interleaved with a concurrently executing delete(5). The delete thread marks n_3 before the search thread reaches the beginning of Line 15, yielding state s'_2 at this point. The test whether r is marked will now fail, causing the search thread to restart. After traversing the list again, the search thread will unlink n_3 from the list with the CAS on Line 14. This yields state s'_3 when the search reaches Line 15 again. The search thread will then proceed to compute the return value $b = \mathit{false}$. For this execution, the decisive node is n_4 and the linearization point is s'_3.

Our program logic provides two ingredients for dealing with the resulting complexity in the linearizability proof. We discuss these formally in §7. The first ingredient is the *past predicate* $\diamondminus p$, which asserts that the current thread owned the resource p at some point in the past. The second ingredient is a set of proof rules for introducing and manipulating past predicates. In particular, we will use the following three rules in our proof:

$$\text{H-INTRO}\quad \vdash \{\, p \,\}\ \mathtt{skip}\ \{\, p * \diamondminus p \,\} \qquad \text{H-HINDSIGHT}\ \frac{p \text{ pure}}{p * \diamondminus q \vdash \diamondminus(p * q)} \qquad \text{H-INFER}\ \frac{p \vdash q}{\diamondminus p \vdash \diamondminus q}\ .$$

Rule H-INTRO states the validity of the Hoare triple $\{\, p \,\}$ skip $\{\, p * \diamondminus p \,\}$ which introduces a past predicate $\diamondminus p$ using a *stuttering step*. Here, $*$ is separating conjunction. The rule expresses that if the thread owns p now, it trivially owned p at some past point up until now. Rule H-HINDSIGHT captures the essence of hindsight reasoning: ownership of p can be transferred from the now into the past, if p is a purely logical fact that is independent of the program state. Rule H-INFER states the monotonicity of the past operator with respect to logical weakening.

We demonstrate the use of past predicates and their associated rules by sketching the linearizability proof for executions of `search(k)` that follow the same code path as the one in the bottom half of Figure 2. The proof relies on a predicate $\mathsf{Node}(x)$, which expresses the existence of the physical representation of x in the heap and binds the *logical variables* $key(x)$ and $mark(x)$ to the values stored in the relevant fields of x—it is a *points-to* predicate $x \mapsto key(x), mark(x), \ldots$ for all fields of x. The predicate also expresses important properties needed for maintaining the keyset invariants. §5 discusses the definition of predicate $\mathsf{Node}(x)$ in detail.

The proof proceeds by symbolic execution of the considered path. The goal is to infer

$$\diamondminus\big(\grave{}\mathsf{Node}(r) \;*\; k \in \grave{}\mathsf{KS}(r) \wedge (b \Leftrightarrow k \in \grave{}\mathsf{C}(r))\big)$$

as postcondition where b is the return value of `search(k)`. This implies the existence of a linearization point as discussed earlier. Here, we write $\grave{}e$ for the expression obtained from expression e by replacing all logical variables like $mark(r)$ by fresh variables $\grave{}mark(r)$. That is, $\grave{}e$ can be thought of as the expression e evaluated with respect to the old state of r captured by $\grave{}\mathsf{Node}(r)$ inside the past predicate, rather than the current state.[1]

The symbolic execution starts from a global invariant that maintains $\mathsf{Node}(x)$ for all nodes x in the heap. When it reaches Line 5 in the proof, we can establish $\mathsf{Node}(t)$ and $\mathsf{Node}(tn)$ using the derived invariant of the traversal. The two predicates imply that if t is unmarked, then its keyset must be non-empty. In turn, this implies $\mathsf{KS}(tn) = (key(t), key(tn)]$, because tn is the successor of t. Together, we deduce (a) $\neg mark(t) \wedge key(t) < k \wedge k \leq key(tn) \Rightarrow k \in \mathsf{KS}(tn)$. Moreover, the definition of $\mathsf{C}(tn)$ gives us (b) $k \in \mathsf{C}(tn) \Leftrightarrow \neg mark(tn) \wedge key(tn) = k$. We let $H(tn, t)$ be the conjunction of (a), (b), and $\mathsf{Node}(tn)$.

Next, we use rule H-INTRO to transfer $H(tn, t)$ into a past predicate, yielding $H(tn, t) * \diamondminus(H(tn, t))$. For our proof to be valid, we need to make sure that all intermediate assertions are stable under interferences by other threads. Unfortunately, this is not the case for the assertion $H(tn, t) * \diamondminus(H(tn, t))$. Notably, this assertion implies that the current value of tn's mark bit is the same as the value of its mark bit in the past state referred to by the past predicate.

To make the assertion stable under interference, we first introduce fresh logical variables $\grave{}key(tn)$ and $\grave{}mark(t)$ which we substitute for $key(tn)$ and $mark(tn)$ under the past operator. This yields the equivalent intermediate assertion:

$$H(tn, t) \;*\; \diamondminus(\grave{}H(tn, t)) \;*\; (\grave{}mark(tn) = mark(tn) \wedge \grave{}key(tn) = key(tn)) \;\;.$$

Next, we observe that other threads executing `search`, `insert`, and `delete` operations can only interfere by marking node tn in case it is not yet marked. Such interference invalidates the equality $mark(tn) = \grave{}mark(tn)$. So we weaken it to $\grave{}mark(tn) \Rightarrow mark(tn)$. In §4 we introduce a general Owicki-Gries-style separation logic framework that formalizes this form of interference reasoning. In addition, we only keep $\mathsf{Node}(tn)$ from $H(tn, t)$, leaving us with the interference-free assertion

$$P(tn, t) \triangleq \mathsf{Node}(tn) \;*\; \diamondminus(\grave{}H(tn, t)) \;*\; (\grave{}mark(tn) \Rightarrow mark(tn)) \wedge \grave{}key(tn) = key(tn) \;\;.$$

We then propagate this assertion forward along the considered execution path of `search(k)`, obtaining $P(ln, t)$ when Line 15 is reached in the proof. During the propagation, we accumulate the facts $\neg mark(t)$, $key(t) < k$, $k \leq key(r)$, and $ln = r$ according to the branches of the conditional expressions taken along the path. As these facts are all pure, we use rule H-HINDSIGHT to transfer them, together with the equality $\grave{}key(ln) = key(ln)$, into the past predicate $\diamondminus(\grave{}H(ln, t))$.

Using rule H-INFER we can then simplify the resulting past predicate as follows:

$$\diamondminus\big(\grave{}\mathsf{Node}(r) \;*\; k \in \grave{}\mathsf{KS}(r) \;\wedge\; (k \in \grave{}\mathsf{C}(r) \Leftrightarrow \neg\grave{}mark(r) \wedge \grave{}key(r) = k)\big) \;\;.$$

[1]Observe that $\mathsf{Node}(r) * \diamondminus\mathsf{Node}(r)$ would implicitly state that the fields of r are the same in the past and the present state. Renaming the past state, $\mathsf{Node}(r) * \diamondminus\grave{}\mathsf{Node}(r)$, allows the fields of r to have changed between the past and present.

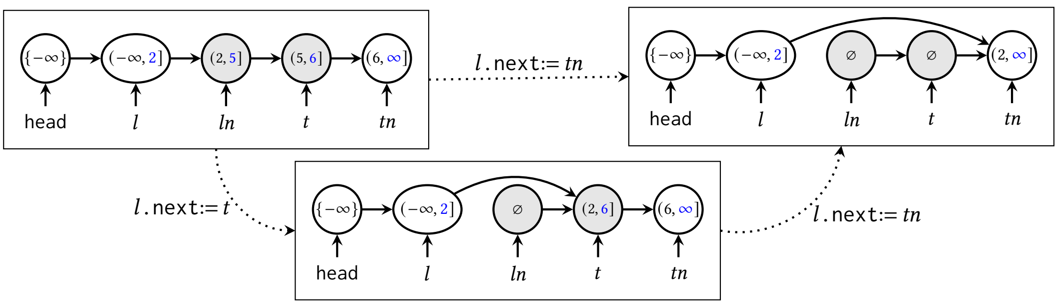

Fig. 3. A compound update (top) and its decomposition into update chunks (bottom). Compound updates may affect an unbounded number of nodes, e.g., changing their keysets. Update chunks localize this effect of an update to a bounded (and small) number of nodes.

As we propagate the overall assertion further to the return point of search(k), we accumulate the additional pure facts $\neg mark(r)$ and $b \Leftrightarrow key(r) = k$. We again invoke rule H-HINDSIGHT to transfer these into the past predicate, together with $`mark(r) \Rightarrow mark(r)$ and $`key(r) = key(r)$. The resulting past predicate can then be simplified with H-INFER to finally obtain the desired:

$$\diamondminus\big(`\mathsf{Node}(r) \ * \ k \in `\mathsf{KS}(r) \ \wedge \ (b \Leftrightarrow k \in `\mathsf{C}(r)\big) \ .$$

While this proof is non-trivial, it is easy to automate. The analysis performs symbolic execution of the code. After each atomic step, it applies the rules H-INTRO and H-HINDSIGHT eagerly. The resulting past predicates are then simplified and weakened with respect to interferences by other threads. This way, the analysis maintains the strongest interference-free information about the history of the computation. These steps are integrated into a classical fixpoint computation to infer loop invariants, applying standard widening techniques to enforce convergence [Cousot and Cousot 1979].

It is worth pointing out that the above reasoning could also be done with prophecies [Jung et al. 2020; Liang and Feng 2013]. However, prophecies are not amenable to automation in the same way as past predicates. The main reason for this is the hindsight rule: it works relative to facts that have already been discovered during symbolic execution. Prophecies would require to *guess* the same facts prior to being discovered. This guessing step is notoriously hard to automate [Bouajjani et al. 2017].

Automating Future Reasoning. Our second major contribution is the idea of futures and their governing reasoning principles. We motivate futures with the problem of automatically proving that the Harris set maintains the keyset invariants. As noted earlier, this is challenging because the CAS performed by find may unlink unboundedly many marked nodes. Unlinking a node changes its keyset to the empty set. Hence, the CAS may affect an unbounded heap region. Showing that the CAS maintains the keyset invariants therefore inevitably involves an inductive proof argument.

Our observation is that one can reason about the effect of the CAS that unlinks the marked segment by composing it out of a sequence of *update chunks* that unlink the nodes in the segment one by one as indicated in Figure 1. This yields an inductive proof argument where we reason about simpler updates that only affect a bounded number of nodes at a time. The correctness of an update chunk com is represented by a future $\langle\, P \,\rangle$ com $\langle\, Q \,\rangle$. A future can be thought of as a Hoare triple with precondition P and postcondition Q. However, futures inhabit the assertion level of the logic, rather than the meta level. The crux of our program logic is that it allows one to derive a future *on the side* in a subproof, while proving the correctness of the thread's traversal of the marked segment. The advantage of this approach is that one can reuse the loop invariant for the proof of traverse towards proving the correctness of $\langle\, P \,\rangle$ com $\langle\, Q \,\rangle$. This aids proof automation: the analysis no longer needs to synthesize an induction hypothesis for reasoning about the affected heap region out of thin air at the point where the `CAS` is executed. In a way, `traverse` moonlights as ghost code that aids the correctness proof of the `CAS`.

At the point where the `CAS` is executed, the proof then invokes the constructed future. A future can thus be thought of as a subproof that is saved up to be applied at some future point.

This idea is illustrated in Figure 3. The top left of the figure shows a state that is reached during the traversal of the marked segment from l to tn. The transition at the top depicts the effect of the update $l.\texttt{next} := tn$ applied to this state. The update unlinks the marked nodes between l and tn in one step. The correctness of this update is expressed by the future

$$F \triangleq \langle\, P * \mathit{next}(l) = \mathit{ln} * \mathit{mark}(\mathit{ln}) \,\rangle \;\; l.\texttt{next} := \mathit{tn} \;\; \langle\, P * \mathit{next}(l) = \mathit{tn} \,\rangle$$

where P is an appropriate invariant holding the relevant physical resources of the involved nodes. We derive this future by composing two futures for simpler update chunks as depicted at the bottom half of Figure 3. The left resp. right update chunk is described by the future F_1 resp. F_2 as follows:

$$\begin{aligned} F_1 &\triangleq \langle\, P * \mathit{next}(l) = \mathit{ln} * \mathit{mark}(\mathit{ln}) \,\rangle \;\; l.\texttt{next} := t \;\; \langle\, P * \mathit{next}(l) = t \,\rangle \\ F_2 &\triangleq \langle\, P * \mathit{next}(l) = t * \mathit{mark}(t) * \mathit{next}(t) = \mathit{tn} \,\rangle \;\; l.\texttt{next} := \mathit{tn} \;\; \langle\, P * \mathit{next}(l) = \mathit{tn} \,\rangle \quad . \end{aligned}$$

The future F_1 is derived inductively during the traversal of the marked segment from l to t using the same process that we are about to describe for deriving F. The future F_2 can be easily proved in isolation. In particular, the precondition $\mathit{mark}(t)$ implies $\mathsf{C}(t) = \emptyset$. Hence, the keyset invariant $\mathsf{C}(t) \subseteq \mathsf{KS}(t)$ is maintained. The condition $\mathit{next}(t) = \mathit{tn}$ guarantees that the update does not affect keysets of other nodes beyond t and tn.

We would now like to compose F_1 and F_2 using the standard sequential composition rule of Hoare logic to derive the future

$$F' \triangleq \langle\, P * \mathit{next}(l) = \mathit{ln} * \mathit{mark}(\mathit{ln}) \,\rangle \;\; l.\texttt{next} := t; l.\texttt{next} := \mathit{tn} \;\; \langle\, P * \mathit{next}(l) = \mathit{tn} \,\rangle \quad .$$

Once we have F', we get F by replacing the command $l.\texttt{next} := t; l.\texttt{next} := \mathit{tn}$ in F' with $l.\texttt{next} := \mathit{tn}$ using a simple subsumption argument.

However, sequential composition requires that the postcondition of F_1 implies the precondition of F_2. Unfortunately, the precondition of F_2 makes the additional assumptions $\mathit{mark}(t)$ and $\mathit{next}(t) = \mathit{tn}$ that are not guaranteed by F_1. Now, observe that both facts are readily available in the outer proof context of the traversal: $\mathit{next}(t) = \mathit{tn}$ follows from Line 5 of `traverse` and $\mathit{mark}(t)$ is obtained from the condition on Line 7. We transfer the facts from the outer proof context into F_2. This eliminates them from the precondition and enables the sequential composition to obtain F'. We refer to this transfer of facts as *accounting*. In a concurrent setting, accounting is sound provided the accounted facts are interference-free. This is the case here since the `next` fields of marked nodes are never changed and marked nodes are never unmarked. We explain this reasoning in more detail in §6.

The idea of futures applies more broadly. They are useful whenever complex updates are prepared in advance by a traversal (as e.g. in Bw trees [Levandoski et al. 2013] and skip lists [Fraser 2004]).

3 PROGRAMMING MODEL

We develop our reasoning principles in the context of concurrency libraries, which offer code to client applications that may be executed by an arbitrary number of threads. With this definition, not only the previously discussed search structures but also storage structures like stacks or lists and mutual exclusion mechanisms form concurrency libraries, and our techniques will apply to them as well. In this section, we formalize concurrency libraries. Our development is parametric in the set of states and the set of commands, following the approach of abstract separation logic [Calcagno et al. 2007; Dinsdale-Young et al. 2013; Jung et al. 2018].

States. We assume that states form a *separation algebra*, a partial commutative monoid $(\Sigma, *, \mathsf{emp})$. For the set of units $\mathsf{emp} \subseteq \Sigma$ we require that (i) for all $\mathsf{s} \in \Sigma$, there exists some $1_\mathsf{s} \in \mathsf{emp}$ with $\mathsf{s} * 1_\mathsf{s} = \mathsf{s}$ and (ii) for all distinct $1, 1' \in \mathsf{emp}$, $1 * 1'$ is undefined. We use $\mathsf{s}_1 \# \mathsf{s}_2$ to indicate definedness.

Predicates are sets of states $p \in \mathbb{P}(\Sigma)$. To simplify the exposition, we stay on the semantic level and do not introduce an assertion language. Predicates form a Boolean algebra $(\mathbb{P}(\Sigma), \cup, \cap, \subseteq, \overline{}, \emptyset, \Sigma)$. *Separating conjunction* $p * q$ lifts the composition from states to predicates. This yields a commutative monoid with unit emp. *Separating implication* $p \mathrel{-\!\!*} q$ gives residuals:

$$p * q \ \triangleq\ \{\, \mathsf{s}_1 * \mathsf{s}_2 \ \mid\ \mathsf{s}_1 \in p \ \wedge\ \mathsf{s}_2 \in q \ \wedge\ \mathsf{s}_1 \# \mathsf{s}_2 \,\} \quad \text{and} \quad p \mathrel{-\!\!*} q \ \triangleq\ \{\, \mathsf{s} \ \mid\ \{\mathsf{s}\} * p \ \subseteq\ q \,\}\ .$$

In our development, $(\Sigma, *, \mathsf{emp})$ is a product of two separation algebras $(\Sigma_\mathsf{G}, *_\mathsf{G}, \mathsf{emp}_\mathsf{G})$ and $(\Sigma_\mathsf{L}, *_\mathsf{L}, \mathsf{emp}_\mathsf{L})$. We require $\mathsf{emp} \triangleq \mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L} \subseteq \Sigma \subseteq \Sigma_\mathsf{G} \times \Sigma_\mathsf{L}$. In addition, Σ must be closed under decomposition: if $(\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2) \in \Sigma$ then $(\mathsf{g}_1, \mathsf{l}_1) \in \Sigma$, for all $\mathsf{g}_1, \mathsf{g}_2 \in \Sigma_\mathsf{G}$ and $\mathsf{l}_1, \mathsf{l}_2 \in \Sigma_\mathsf{L}$.

For $(\mathsf{g}, \mathsf{l}) \in \Sigma$ we call g the *global state* and l the *local state*. States are composed component-wise, $(\mathsf{g}_1, \mathsf{l}_1) * (\mathsf{g}_2, \mathsf{l}_2) \triangleq (\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2)$, and this composition is defined only if the product is again in Σ.

Lemma 3.1. $(\Sigma, *, \mathsf{emp})$ *is a separation algebra.*

Commands. The second parameter to our development is the set of commands $(\mathtt{COM}, [\![-]\!])$ that may be used to modify states. The set may be infinite, which allows us to treat atomic blocks as single commands. The effect of commands on states is defined by an interpretation. It assigns to each command a non-deterministic state transformer $[\![\mathtt{com}]\!]$ that takes a state and returns the set of possible successor states, $[\![\mathtt{com}]\!] : \Sigma \to \mathbb{P}(\Sigma)$. We lift the state transformer to predicates [Dijkstra 1976] in the expected way, $[\![\mathtt{com}]\!](p) \triangleq \bigcup_{\mathsf{s} \in p} [\![\mathtt{com}]\!](\mathsf{s})$. We assume to have a command skip that is interpreted as the identity. For the frame rule to be sound, we expect the following monotonicity to hold [Calcagno et al. 2007; Dinsdale-Young et al. 2013], for all p, q, o:

$$[\![\mathtt{com}]\!](p) \subseteq q \quad \text{implies} \quad [\![\mathtt{com}]\!](p * o) \subseteq q * o\ . \tag{LocCom}$$

We handle commands that may fail as in [Calcagno et al. 2007] by letting them return abort which is added as a new top element to the powerset lattice $\mathbb{P}(\Sigma)$, so that (LocCom) is trivially satisfied.

Concurrency Libraries. Having fixed the set of states and the set of commands, a concurrency library is defined by a single program that is executed in every thread. The assumption of a single program can be made without loss of generality. The program code is drawn from the standard while-language ST defined by:

$$\mathtt{st} \ ::=\ \mathtt{com} \ \mid\ \mathtt{st} + \mathtt{st} \ \mid\ \mathtt{st}; \mathtt{st} \ \mid\ \mathtt{st}^*\ .$$

The semantics of the library is defined in terms of unlabeled transitions among configurations. A *configuration* is a pair $\mathsf{cf} = (\mathsf{g}, \mathsf{pc})$ consisting of a global state $\mathsf{g} \in \Sigma_\mathsf{G}$ and a program counter $\mathsf{pc} : \mathbb{N} \to \Sigma_\mathsf{L} \times \mathsf{ST}$. The program counter assigns to every thread, modeled as a natural number, the current local state and the statement to be executed next. We use CF to denote the set of all configurations. A configuration $(\mathsf{g}, \mathsf{pc})$ is *initial* for predicate p and library code st, if the program counter of every thread yields a local state $(\mathsf{l}, \mathtt{st})$ where the code is the given one and the state satisfies $(\mathsf{g}, \mathsf{l}) \in p$. The configuration is *accepting* for predicate q, if every terminated thread $(\mathsf{l}, \mathtt{skip})$ satisfies the predicate, $(\mathsf{g}, \mathsf{l}) \in q$. We write these configuration predicates as the following sets

$$\mathsf{Init}_{p,\mathtt{st}} \triangleq \{(\mathsf{g}, \mathsf{pc}) \mid \forall i, \mathsf{l}, \widehat{\mathtt{st}}.\, \mathsf{pc}(i) = (\mathsf{l}, \widehat{\mathtt{st}}) \Rightarrow (\mathsf{g}, \mathsf{l}) \in p \wedge \widehat{\mathtt{st}} = \mathtt{st}\}$$
$$\mathsf{Acc}_q \triangleq \{(\mathsf{g}, \mathsf{pc}) \mid \forall i, \mathsf{l}.\ \mathsf{pc}(i) = (\mathsf{l}, \mathtt{skip}) \Rightarrow (\mathsf{g}, \mathsf{l}) \in q\}\ .$$

The unlabeled transition relation among configurations is defined in Figure 4. It relies on a labeled transition relation capturing the flow of control. A command may change the global state and the local state of the executing thread. It will not change the local state of other threads. A computation

$$\mathtt{com} \xrightarrow{\mathtt{com}} \mathtt{skip} \qquad\qquad \mathtt{skip};\mathtt{st} \xrightarrow{\mathtt{skip}} \mathtt{st} \qquad\qquad \mathtt{st}^* \xrightarrow{\mathtt{skip}} \mathtt{skip} + \mathtt{st};\mathtt{st}^*$$

$$\frac{i \in \{1,2\}}{\mathtt{st}_1 + \mathtt{st}_2 \xrightarrow{\mathtt{skip}} \mathtt{st}_i} \qquad \frac{\mathtt{st}_1 \xrightarrow{\mathtt{com}} \mathtt{st}_1'}{\mathtt{st}_1;\mathtt{st}_2 \xrightarrow{\mathtt{com}} \mathtt{st}_1';\mathtt{st}_2} \qquad \frac{\mathtt{st}_1 \xrightarrow{\mathtt{com}} \mathtt{st}_2 \qquad (\mathsf{g}_2,\mathsf{l}_2) \in [\![\mathtt{com}]\!](\mathsf{g}_1,\mathsf{l}_1)}{(\mathsf{g}_1, \mathsf{pc}[i \mapsto (\mathsf{l}_1, \mathtt{st}_1)]) \to (\mathsf{g}_2, \mathsf{pc}[i \mapsto (\mathsf{l}_2, \mathtt{st}_2)])}$$

Fig. 4. Transition relation $\to \subseteq \mathsf{CF} \times \mathsf{CF}$ based on the control-flow relation $\xrightarrow{\ \ } \subseteq \mathtt{ST} \times \mathtt{COM} \times \mathtt{ST}$.

of the library is a finite sequence of consecutive transitions. A configuration is reachable if there is a computation that leads to it. We write $\mathsf{Reach}(\mathsf{cf})$ for the set of all configurations reachable from cf and lift the notation to sets where needed.

4 OWICKI-GRIES FOR CONCURRENCY LIBRARIES

We formulate the correctness of concurrency libraries as the validity of Hoare triples $\{\,p\,\}\ \mathtt{st}\ \{\,q\,\}$. A Hoare triple is valid if for every configuration cf that is initial wrt. p and st, every reachable configuration cf′ is accepting wrt. q. The definition refers to all threads executing the library code.

Definition 4.1. $\models \{\,p\,\}\ \mathtt{st}\ \{\,q\,\} \triangleq \mathsf{Reach}(\mathsf{Init}_{p,\mathtt{st}}) \subseteq \mathsf{Acc}_q$.

To establish this validity, we develop a thread-modular reasoning principle [Owicki and Gries 1976] that proceeds in two steps. First, we verify the library code as if it was run by an isolated thread using judgments of the form $\mathbb{P}, \mathbb{I} \Vdash \{\,p\,\}\ \mathtt{st}\ \{\,q\,\}$.

The Hoare triple of interest is augmented by two pieces of information. The set $\mathbb{P}$ contains the intermediary predicates encountered during the proof of the isolated thread. The set $\mathbb{I}$ contains the interferences, the changes the isolated thread may perform on the shared state. The notion of interference will be made precise in a moment. Recording both sets during the proof is an idea we have taken from [Dinsdale-Young et al. 2013, Section 7.3].

The second phase of the thread-modular reasoning is to check that the local proof still holds in the presence of other threads. This is the famous interference-freedom check. It takes as input the computed sets $\mathbb{P}$ and $\mathbb{I}$ and verifies that no interference can invalidate a predicate, denoted by $\boxtimes_{\mathbb{I}}\,\mathbb{P}$.

Interference. An *interference* is a pair $(o, \mathtt{com})$ consisting of a predicate and a command. It represents the fact that from states in o environment threads may execute command com. A state (g, l) held by the isolated thread of interest will change under the interference to a state in

$$[\![(o,\mathtt{com})]\!](\mathsf{g},\mathsf{l}) \;\triangleq\; \{\, (\mathsf{g}', \mathsf{l}) \;\mid\; \exists \mathsf{l}_1, \mathsf{l}_2.\;\; (\mathsf{g}, \mathsf{l}_1) \in o \;\wedge\; (\mathsf{g}', \mathsf{l}_2) \in [\![\mathtt{com}]\!](\mathsf{g}, \mathsf{l}_1)\,\} \;\;.$$

We consider every state $(\mathsf{g}, \mathsf{l}_1) \in o$ that agrees with (g, l) on the global component, compute the post, and combine the resulting global component with the local component l. The agreement of different threads on the global state is precisely what is used in program logics like RGSep [Vafeiadis 2008; Vafeiadis and Parkinson 2007]. We lift $[\![(o,\mathtt{com})]\!]$ to predicates in the expected way.

We only record interferences that have an effect on the global state. An interference $(o, \mathtt{com})$ is *effectful*, denoted by $\mathsf{eff}(o,\mathtt{com})$, if it changes the shared state of an element in o:

$$\mathsf{eff}(o,\mathtt{com}) \;\triangleq\; \exists (\mathsf{g}_1,\mathsf{l}_1) \in o.\exists (\mathsf{g}_2,\mathsf{l}_2) \in [\![\mathtt{com}]\!](\mathsf{g}_1,\mathsf{l}_1).\mathsf{g}_2 \neq \mathsf{g}_1 \;\;.$$

The thread-local proof computes a set of interferences. For a predicate o and a command com, the interference set is $\mathsf{inter}(o,\mathtt{com}) \triangleq \{(o,\mathtt{com})\}$ if $\mathsf{eff}(o,\mathtt{com})$ and $\mathsf{inter}(o,\mathtt{com}) \triangleq \varnothing$ otherwise. We consider interference sets up to the operation of joining predicates for the same command, $\{(p,\mathtt{com})\} \cup \{(q,\mathtt{com})\} = \{(p \cup q,\mathtt{com})\}$. Then $\mathsf{inter}(o,\mathtt{com}) \subseteq \mathbb{I}$ means there is no interference to capture or there is an interference $(r,\mathtt{com}) \in \mathbb{I}$ with $o \subseteq r$. We write $\mathbb{I} * r$ for the set of interferences $(o * r, \mathtt{com})$ with $(o,\mathtt{com}) \in \mathbb{I}$. Similarly, we write $\mathbb{P} * r$ for the set of predicates $p * r$ with $p \in \mathbb{P}$.

$$\textsc{com-sem}\ \frac{[\![\mathsf{com}]\!](p) \subseteq q}{\{q\}, \mathsf{inter}(p, \mathsf{com}) \Vdash \{\, p \,\}\ \mathsf{com}\ \{\, q \,\}} \qquad \textsc{infer-sem}\ \frac{p \subseteq p' \quad \mathbb{P}_1, \mathbb{I}_1 \Vdash \{\, p' \,\}\ \mathsf{st}\ \{\, q' \,\} \quad q' \subseteq q}{\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2 \Vdash \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}}$$

$$\textsc{frame}\ \frac{\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}}{\mathbb{P} * o, \mathbb{I} * o \Vdash \{\, p * o \,\}\ \mathsf{st}\ \{\, q * o \,\}} \qquad \textsc{seq}\ \frac{\mathbb{P}_1, \mathbb{I}_1 \Vdash \{\, p \,\}\ \mathsf{st}_1\ \{\, q \,\} \qquad \mathbb{P}_2, \mathbb{I}_2 \Vdash \{\, q \,\}\ \mathsf{st}_2\ \{\, o \,\}}{\{q\} \cup \mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2 \Vdash \{\, p \,\}\ \mathsf{st}_1; \mathsf{st}_2\ \{\, o \,\}}$$

$$\textsc{loop}\ \frac{\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathsf{st}\ \{\, p \,\}}{\{p\} \cup \mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathsf{st}^*\ \{\, p \,\}} \qquad \textsc{choice}\ \frac{\mathbb{P}_1, \mathbb{I}_1 \Vdash \{\, p \,\}\ \mathsf{st}_1\ \{\, q \,\} \qquad \mathbb{P}_2, \mathbb{I}_2 \Vdash \{\, p \,\}\ \mathsf{st}_2\ \{\, q \,\}}{\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2 \Vdash \{\, p \,\}\ \mathsf{st}_1 + \mathsf{st}_2\ \{\, q \,\}}$$

Fig. 5. Program logic.

The *interference-freedom check* takes as input a set of interferences $\mathbb{I}$ and a set of predicates $\mathbb{P}$. It checks that no interference can invalidate a predicate, $[\![(o, \mathsf{com})]\!](p) \subseteq p$ for all $(o, \mathsf{com}) \in \mathbb{I}$ and all $p \in \mathbb{P}$. If this is the case, we write $\boxtimes_{\mathbb{I}} \mathbb{P}$ and say that the set of predicates $\mathbb{P}$ is interference-free wrt. $\mathbb{I}$.

The interference-freedom check is non-compositional, and in manual/mechanized program verification this has been the reason to prefer rely-guarantee methods [Feng 2009; Vafeiadis 2008; Vafeiadis and Parkinson 2007]. From the point of view of automated verification, the difference does not matter. After all, there is no compositional way of computing the relies and guarantees.

Program Logic. To verify library code as if it was run by an isolated thread, we derive (augmented) Hoare triples using the proof rules in Figure 5. The rules are standard except that they work on the semantic level. This is best seen in rule com-sem, which explicitly checks the postcondition for over-approximating the postimage. The rule only adds the postcondition to the set of predicates to be checked for interference freedom. Similarly, the consequence rule infer-sem neither adds the strengthened precondition nor the weakened postcondition. We can freely manipulate predicates as long as there is an interference-free predicate between every pair of consecutive statements, rule seq. To ensure this for loops which may be left without execution, rule loop adds p to the set of predicates. The initial predicate of the overall Hoare triple is added to the set of predicates by the assumption of Theorem 4.2.

Theorem 4.2 (Soundness). *$\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}$ and $\boxtimes_{\mathbb{I}} \mathbb{P}$ and $p \in \mathbb{P}$ imply $\models \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}$.*

Linearizability. We extend the above program logic in order to utilize it for linearizability proofs. Our extension draws on ideas from atomic triples [da Rocha Pinto et al. 2014]. That is, we enforce that every operation linearizes exactly once and satisfies a given sequential specification when doing so. In the context of concurrent search structures (CSS), sequential specifications Ψ take the form:

$$\Psi \;=\; \{\, C.\ \mathsf{CSS}(C) \,\}\ op(k)\ \{\, v.\ \exists C'.\ \mathsf{CSS}(C') * \mathsf{UP}(C, C', k, v) \,\}\ .$$

The sets C and C' are the logical contents of the structure before and after operation $op(k)$. The predicate $\mathsf{CSS}(C)$ connects the structure's physical state with the logical contents C. The relation $\mathsf{UP}(C, C', k, v)$ encodes the admissible updates and the expected return value v of $op(k)$.

To enforce exactly one linearization point, we use update tokens [Vafeiadis 2008] of the form $\mathsf{Obl}(\Psi)$ and $\mathsf{Ful}(\Psi, v)$. The former states that an operation has not yet encountered its linearization point, it is still obliged to linearize. The latter states that the linearization point has been encountered and that value v must be returned to comply with Ψ. Technically, the update tokes are thread-local ghost resources. We assume the program semantics to simply ignore this ghost component. Note that having thread-local update tokens does not allow for proving impure future-dependent linearization points as they require intricate helping protocols where threads exchange their

$$\textsc{lin-none}\ \frac{\mathbb{P},\mathbb{I} \Vdash \{\, a \,\}\ \mathsf{com}\ \{\, b \,\} \quad a \subseteq \mathsf{CSS}(C) \qquad b \subseteq \mathsf{CSS}(C)}{\mathbb{P},\mathbb{I} \Vdash_{lin} \{\, a \,\}\ \mathsf{com}\ \{\, b \,\}} \qquad \textsc{lin-past}\ \frac{a \subseteq \Diamond\big(\mathsf{CSS}(C) \cap \mathsf{UP}(C, C, k, v)\big)}{\mathbb{P},\mathbb{I} \Vdash_{lin} \{\, \mathsf{Obl}(\Psi) * a \,\}\ \mathsf{skip}\ \{\, \mathsf{Ful}(\Psi, v) * a \,\}}$$

$$\textsc{lin-now}\ \frac{a \subseteq \mathsf{CSS}(C) \qquad \mathbb{P},\mathbb{I} \Vdash \{\, a \,\}\ \mathsf{com}\ \{\, b \,\} \qquad b \subseteq \mathsf{CSS}(C') \cap \mathsf{UP}(C, C', k, v)}{\mathbb{P},\mathbb{I} \Vdash_{lin} \{\, \mathsf{Obl}(\Psi) * a \,\}\ \mathsf{com}\ \{\, \mathsf{Ful}(\Psi, v) * b \,\}}$$

Fig. 6. Proof rules handling linearizability tokens for commands. Rule LIN-PAST is detailed in §7.

update tokes through the global state so that other threads can resolve them. A generalization is straight-forward, but we prefer the simpler setting to not distract from our contributions.

To prove linearizability, we introduce a new proof system $\Vdash_{lin}$ that coincides with $\Vdash$ from Figure 5 except that it replaces rule COM-SEM with the rules from Figure 6 (ignore rule LIN-PAST for now, we explain it in §7). The new rules handle the update tokens, leaving the task of establishing the validity of the actual Hoare triple to the base proof system $\Vdash$. To do this, Rule LIN-NONE ensures that the command does not change the logical contents of the structure, meaning that the update tokens are unaffected. Rule LIN-NOW converts the update token $\mathsf{Obl}(\Psi)$ into $\mathsf{Ful}(\Psi, v)$ if the command is a linearization point satisfying the sequential specification Ψ. This conversion is applicable only once (because $\mathsf{Obl}(\Psi)$ is not duplicable), which makes sure there can be at most one linearization point. Note that this rule handles both pure and impure linearization points. Similar to atomic triples [da Rocha Pinto et al. 2014], the rule requires threads to observe the very moment the linearization point occurs. To ensure there is at least one linearization point, we require the operation's postcondition to contain the appropriate fulfillment token $\mathsf{Ful}(\Psi, v)$. Formally, we seek to establish Hoare triples of the following form:

$$\Vdash_{lin} \{\, C.\ \mathsf{CSS}(C) * \mathsf{Obl}(\Psi) \,\}\ op(k)\ \{\, v.\ \exists C'.\ \mathsf{CSS}(C') * \mathsf{Ful}(\Psi, v) \,\}\ .$$

We take the derivability of such a Hoare triple as the ground truth for linearizability, trusting that a semantic result connecting the derivability to a statement about computation histories would be routine to derive [Liang and Feng 2013].

5 REASONING ABOUT KEYSETS USING FLOWS

Recall from §2 that we localize the reasoning about the abstract state $\mathsf{C}(N)$ of the data structure to the contents $\mathsf{C}(x)$ of a single node x using its keyset $\mathsf{KS}(x)$. In this section, we define the keysets as a derived quantity that we can reason about locally in a separation logic. Then, we use this formalism to define the node-local invariant $\mathsf{Node}(x)$ of our running example. This node-local invariant is used by our tool to fully automatically generate a proof of the Harris set (cf. §8).

We derive the keyset of a node x from another quantity $\mathsf{IS}(x)$, the node's *inset*. Intuitively, $k \in \mathsf{IS}(x)$ if a thread searching for k will traverse node x. For the Harris set, we define $\mathsf{IS}(\mathsf{head}) = [-\infty, \infty]$ and for every other node we obtain $\mathsf{IS}(x)$ as the solution of the following fixpoint equation:

$$\mathsf{IS}(x) \ = \ \textstyle\bigcup_{(y,x) \in E} \ \mathsf{IS}(y) \cap (y.\mathsf{key}, \infty]\ \ .$$

Here, the set of edges E is induced by the next pointers in the heap. If we remove those keys k from $\mathsf{IS}(x)$ for which a search leaves x (i.e., if $k > x.\mathsf{key}$ in the Harris set), we obtain $\mathsf{KS}(x)$. These definitions ensure for free that the keysets are disjoint, the first of our keyset invariants. They also generalize to any search structure [Shasha and Goodman 1988].

To express keysets in separation logic, we use the flow framework [Krishna et al. 2018, 2020b]. In this framework, the heap is augmented by associating every node x with a quantity *flow*(x) that is defined as a solution of a fixpoint equation over the heap like the one defining the inset above. Assertions describe disjoint fragments of the augmented global heap, similar to classical separation logic. The augmented heap fragments are called *flow graphs*. In addition to tracking the flow of each node, a flow graph also has an associated *interface* consisting of an *inflow* and an *outflow*. The inflow $in(y, x)$ captures the contribution to the flow of x inside the heap fragment via an edge from a heap node y outside the fragment, and conversely for the outflow $out(x, y)$. Flow graphs fg and fg' compose if they are disjoint and their interfaces are compatible (i.e., the composed flow graph $fg * fg'$ has the same flow as the components). The framework then enables local reasoning about the effects of heap updates on flow graphs. In essence, if a local update inside a region fg of a larger flow graph $fg * fg'$ maintains fg's interface, then the flow in fg' does not change. Hence, any property about fg', such as that each of its nodes x satisfies the keyset invariant $\mathsf{C}(x) \subseteq \mathsf{KS}(x)$, can be framed across the update. Note that the flow and interfaces associated with the physical heap constitute ghost state. Intuitively, they are recomputed after each update.

For the remainder of the paper, it suffices to know that we instantiate the framework such that a node's inset can be obtained from its flow. To apply the framework to concurrent search structures and to reason locally about their sequential specifications, we define the CSS predicate from §4 by:

$$\mathsf{CSS}(C) \triangleq \exists N.\ \mathsf{Inv}(C, N, N)\,,$$

where N is the set of nodes the search structure is composed of and $\mathsf{Inv}(C, N', N)$ is a search structure specific invariant. The invariant is parameterized in C and N as well as a subset $N' \subseteq N$ for which $\mathsf{Inv}(C, N', N)$ holds actual resources. It must ensure that C is the contents of the subregion N', i.e. $C = \mathsf{C}(N')$. It must also ensure the keyset invariant $\mathsf{C}(x) \subseteq \mathsf{KS}(x)$ for all $x \in N'$. Krishna et al. [2018, 2020b] show how the two constraints allow us to split and merge the invariant for disjoint subregions of the structure for the purpose of framing. It is this splitting/merging that localizes the reasoning. The following definition makes the desired property formally precise.

Definition 5.1. An invariant Inv is *decomposable* if it satisfies:

$$\mathsf{Inv}(C, N'_1 \uplus N'_2, N) \iff \exists\, C_1\, C_2.\ C = C_1 \uplus C_2 \ \wedge\ \mathsf{Inv}(C_1, N'_1, N) * \mathsf{Inv}(C_2, N'_2, N)\,.$$

For linearizability, it thus suffices to identify a small subregion that contains the decisive node of the operation. A search for key k, for instance, only requires the region $\mathsf{Inv}(C_x, \{x\}, N)$ with $k \in \mathsf{KS}(x)$ in order to linearize its return value v because we obtain

$$\mathsf{Inv}(C', N \setminus \{x\}, N) * \mathsf{Inv}(C_x, \{x\}, N) \wedge k \in \mathsf{KS}(x) \wedge v = (k \in C_x) \vdash \exists C.\ \mathsf{CSS}(C) \wedge v = (k \in C)\ .$$

As shown in [Krishna et al. 2020a], there is a generic construction for a decomposable Inv that works for all search structures. To avoid additional technical machinery, we next present a simplified version of this construction that is specific to the Harris set and similar list-based search structures.

The Harris Set Invariant. We represent predicates $p \subseteq \Sigma$ syntactically using separation logic assertions that are for the most part standard. In particular, we use boxed assertions $\boxed{A}$ that are inspired by RGSep [Vafeiadis 2008; Vafeiadis and Parkinson 2007] to mean that A is interpreted in the global state. Unboxed assertions are interpreted in the local state. A *points-to* predicate takes the form $x \mapsto \langle \overline{\mathsf{sel}_i : t_i}, \texttt{flow} : t_{flow}, \texttt{in} : t_{in} \rangle$ and describes a flow graph consisting of a single node x. Here, each sel_i is a field selector and t_i is a term denoting the field's associated value. The *ghost field* `flow` stores x's flow and `in` stores its inflow. The semantics of assertions is $[\![A]\!] \subseteq \Sigma$.

We next define the resources associated with a node x, its inset, and keyset. In proofs, we will assume that assertions are existentially closed, and will omit the corresponding outer quantifiers. Formulas like $\mathsf{Node}(x)$ defined in the following introduce logical variables like $mark(x)$ that are visible beyond $\mathsf{Node}(x)$, for example, to define the keyset term $\mathsf{KS}(x)$. We define:

$$\mathsf{Node}(x) \triangleq \boxed{x \mapsto \langle \mathtt{mark}\colon \mathit{mark}(x), \mathtt{next}\colon \mathit{next}(x), \mathtt{key}\colon \mathit{key}(x), \mathtt{flow}\colon \mathit{flow}(x), \mathtt{in}\colon \mathit{in}(x)\rangle}$$

$$\mathsf{IS}(x) \triangleq x = \mathsf{head}\ ?\ [-\infty, \infty]\ :\ \mathit{flow}(x) \qquad \mathsf{KS}(x) \triangleq \mathsf{IS}(x) \setminus (\mathit{key}(x), \infty]$$

$$\mathsf{C}(x) \triangleq \mathit{mark}(x)\ ?\ \varnothing\ :\ \{\,\mathit{key}(x)\,\} \qquad \mathsf{C}(N) \triangleq \bigcup_{x\in N} \mathsf{C}(x)\ .$$

With these definitions in place, we define the invariant $\mathsf{Inv}(C, N', N)$ that is maintained by each subregion $N' \subseteq N$ of a Harris set structure consisting of nodes N:

$$\begin{aligned}
\mathsf{Inv}(C, N', N) &\triangleq C = \mathsf{C}(N') * \mathsf{HD}(N) * \mathop{\Huge *} x \in N'.\ \mathsf{Node}(x) * \varphi^1(x) * \varphi^2(x) * \varphi^3(x, N) * \varphi^4(x)\\
\mathsf{HD}(N) &\triangleq \mathsf{head} \in N\ *\ \mathit{key}(\mathsf{head}) = -\infty\ *\ \neg\mathit{mark}(\mathsf{head})\\
\varphi^1(x) &\triangleq \neg\mathit{mark}(x)\ \Rightarrow\ \mathsf{IS}(x) \neq \varnothing\\
\varphi^2(x) &\triangleq \mathsf{IS}(x) \neq \varnothing\ \Rightarrow\ [\mathit{key}(x), \infty) \subseteq \mathsf{IS}(x)\\
\varphi^3(x, N) &\triangleq \{x, \mathit{next}(x)\} \in N \wedge (\mathit{key}(x) = \infty\ \Rightarrow\ \neg\mathit{mark}(x))\\
\varphi^4(x) &\triangleq \forall y, z.\ \mathit{in}(x)(y, x) \neq \varnothing * \mathit{in}(x)(z, x) \neq \varnothing\ \Rightarrow\ y = z\ .
\end{aligned}$$

Formula $\varphi^1(x)$ captures that all unmarked nodes are reachable from head. Formula $\varphi^2(x)$ implies the second keyset invariant $C(x) \subseteq \mathsf{KS}(x)$ and will also allow us to establish $k \in \mathsf{KS}(x)$ at the appropriate points in the proof. Formula $\varphi^3(x)$ ensures $N' \subseteq N$ and that N is closed under traversal of next pointers. It also implies that the tail node is unmarked. Finally, $\varphi^4(x)$ implies that there exists at most one path from head to each x that a traversal would actually follow. This is needed to prove that unlinking marked nodes from the structure preserves the invariant. It is worth noting that the invariant does not put many constraints on the data structure shape. The sole purpose is to provide enough information to reason about the keysets. If N is clear, we abbreviate $\mathsf{Inv}(C, N', N)$ to $\mathsf{Inv}(C, N')$. The invariant of the Harris set is decomposable as per Definition 5.1.

Lemma 5.2. *The Harris set invariant* $\mathsf{Inv}(C, N', N)$ *is decomposable.*

6 FUTURES

We now make our program logic future-proof. We refer to the set of nodes affected by an update as the update's *footprint*. An update affects a node x if it changes a field value of x or the flow at x. The footprint of an update on a flow graph is in general larger than the footprint of the same update on the underlying heap graph alone. For instance, $l.\mathtt{next} := r$ does not abort as long as the location at l is in the heap graph. However, if the same command is executed on a flow graph, it will typically require other nodes such as r and $l.\mathtt{next}$ to be present in order for the command not to abort. This is because the command may change the flow of these other nodes. For instance, the footprint of the $\mathtt{CAS}(l.\mathtt{next}, ln, r)$ on Line 14 of Figure 1 comprises the entire marked segment between l and r, because it changes the flow and hence the keysets of all nodes in the segment. We introduce futures to reason about updates with such unbounded footprints.

As futures admit general reasoning principles, we study them in the abstract semantic setting of §4 and then apply the developed principles to our concrete running example.

Reasoning about Futures. Futures are expressed in terms of weakest preconditions. We define the weakest precondition $\mathit{wp}(\mathsf{com}, q)$ of a command com and predicate q in the expected way: $\mathit{wp}(\mathsf{com}, q) \triangleq \{\, \mathrm{s} \in \Sigma \mid [\![\mathsf{com}]\!](\mathrm{s}) \subseteq q \,\}$. The weakest precondition of the sequential composition $\mathsf{com}_1; \mathsf{com}_2$ is also defined as usual: $\mathit{wp}(\mathsf{com}_1; \mathsf{com}_2, q) \triangleq \mathit{wp}(\mathsf{com}_1, \mathit{wp}(\mathsf{com}_2, q))$.

Definition 6.1. Futures are $\langle\, p \,\rangle\ \mathsf{com}\ \langle\, q \,\rangle \triangleq p \mathrel{-\!\!*} \mathit{wp}(\mathsf{com}, q)$.

$$\textsc{f-intro}\ \frac{p \subseteq wp(\mathsf{com}, q)}{\mathsf{emp} \ \subseteq\ \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle} \qquad \textsc{f-seq}\ \frac{wp(\mathsf{com}_1; \mathsf{com}_2, o) \subseteq wp(\mathsf{com}, o)}{\langle\, p \,\rangle\, \mathsf{com}_1\, \langle\, q \,\rangle * \langle\, q \,\rangle\, \mathsf{com}_2\, \langle\, o \,\rangle \ \subseteq\ \langle\, p \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle}$$

$$\textsc{f-infer}\ \frac{p_2 \subseteq p_1 \qquad q_1 \subseteq q_2}{\langle\, p_1 \,\rangle\, \mathsf{com}\, \langle\, q_1 \,\rangle \ \subseteq\ \langle\, p_2 \,\rangle\, \mathsf{com}\, \langle\, q_2 \,\rangle} \qquad \textsc{f-frame}\ \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle \ \subseteq\ \langle\, p * o \,\rangle\, \mathsf{com}\, \langle\, q * o \,\rangle$$

$$\textsc{f-account}\ p * \langle\, p * q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle \ \subseteq\ \langle\, q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle \qquad \textsc{f-invoke}\ p * \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle \ \subseteq\ wp(\mathsf{com}, q)$$

Fig. 7. Implications among futures.

Readers familiar with Iris [Jung et al. 2018] will note that our definition of futures resembles Iris' notion of Hoare triples. Technically, Hoare triples in Iris are duplicable resources (they are guarded by a persistence modality) while our futures are not (they may carry resources and are therefore subject to interference). However, one can directly encode our definition of futures in Iris via *wp*. The key novelty of our approach is the way futures are used, in particular the reasoning technique of accounting and composition as showcased in §2.

Figure 7 gives the implications we use for reasoning about futures. Rule F-INTRO turns an ordinary Hoare triple that proves the correctness of a command com into a future. Rules F-INFER and F-FRAME correspond to rules INFER-SEM and FRAME for Hoare triples. Rule F-INVOKE allows us to invoke a future $\langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle$ at the point in the proof where the update chunk com is actually executed. That is, we can use this rule to discharge the premise of rule COM-SEM.

The composition of ghost update chunks is implemented by rule F-SEQ. It is similar to the rule for sequential composition in Hoare logic with two important differences. First, it requires a separating conjunction of the futures for the update chunks com_1 and com_2. The reason is that futures may carry resources. Second, it replaces the composition of com_1 and com_2 by a new update chunk com that is equivalent. Unlike rule SEQ, rule F-SEQ does not take into account interferences on the intermediate assertion p. This is correct, since update chunks represent ghost computation that takes effect instantaneously, meaning com_1 and com_2 are executed uninterruptedly at the moment when the new update chunk com is invoked.

Finally, rule F-ACCOUNT enables the partial invocation of a future $\langle\, p * q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle$ by eliminating the premise p if it is present in the current proof context. We refer to this rule as *accounting*. We have already seen in §2 that accounting is useful to enable the composition of two futures using F-SEQ. Note that if p is subject to inference, so is the future $\langle\, q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle$ obtained from rule F-ACCOUNT.

LEMMA 6.2. *Rules F-INTRO, F-SEQ, F-INFER, F-FRAME, F-ACCOUNT, F-INVOKE are sound (valid implications).*

Proving the Harris Set Invariant. We demonstrate the versatility of futures by using them to prove that the CAS on Line 14 of the Harris set preserves the invariant of the data structure. Figure 8 shows the proof outline. The code is equivalent to the one in Figure 1, except that the mark bits have been made explicit. We discuss the key aspects of the proof in more detail.

The precondition of traverse contains the predicate $\mathsf{TInv}(C, N, M, l, ln, lmark, t)$. It is the invariant of traverse and states that the data structure's invariant $\mathsf{Inv}(C, N)$ is maintained. Additionally, the precondition contains the future $\mathsf{Fut}(M, l, ln, t)$. It captures the fact that the segment from l to t consisting of the nodes $M \setminus \{\, l, t \,\}$ can be safely unlinked via the update $l.\mathtt{next} := t$, provided $l.\mathtt{next} = ln$ and $\neg mark(l)$. After the update, the future guarantees that we have $(key(l), \infty] \subseteq flow(t)$, a crucial fact we will later use in the linearizability proof (cf. §7). It also guarantees that the contents of the modified segment is not changed by the update.

```
22 $\mathsf{TInv}(C, N, M, l, ln, lmark, t) \triangleq \mathsf{Inv}(C, N) * \mathsf{C}(M) \subseteq \mathsf{C}(\{l, t\}) * \neg lmark * key(l) < k < \infty * \{l, ln, t\} \subseteq M \subseteq N$
23 $\mathsf{Fut}(M, l, ln, t) \triangleq \langle \mathsf{P}(M, l, ln, t, ln) \rangle\ l.\mathsf{next} := t\ \langle \mathsf{Q}(M, l, ln, t, t) \rangle$
24 $\mathsf{P}(M, l, ln, t, u) \triangleq \mathsf{Inv}(\mathsf{C}(\{l, t\}), M) * \{l, t\} \subseteq M * next(l) = u * \neg mark(l)$
25 $\mathsf{Q}(M, l, ln, t, u) \triangleq \mathsf{Inv}(\mathsf{C}(\{l, t\}), M) * \{l, t\} \subseteq M * next(l) = u * \neg mark(l) * (key(l), \infty] \subseteq flow(t)$
26 $\{ \exists N\, M\, C.\ \mathsf{TInv}(C, N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) \}$
27 procedure traverse(k: K, l: N, ln: N, lmark: Bool, t: N) {
28   val tn, tmark = atomic {t.next, t.mark}
29   $\{ \mathsf{TInv}(C, N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) * tn \in N * (tmark \Rightarrow mark(t) = tmark * next(t) = tn) \}$
30   if (tmark) {
31     $\{ \mathsf{TInv}(C, N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) * tn \in N * mark(t) * mark(t) = tmark * next(t) = tn \}$
32     $\{ \mathsf{TInv}(C, N, M, l, ln, lmark, tn) * \mathsf{Fut}(M, l, ln, tn) \}$
33     return traverse(k, l, ln, tn)
34   } else if (t.key < k) {
35     $\{ \mathsf{TInv}(C, N, M, l, ln, lmark, tn) * \mathsf{Fut}(M, l, ln, tn) * key(t) < k \}$
36     return traverse(k, t, tn, tmark, tn)
37   } else {
38     $\{ \mathsf{TInv}(C, N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) * t \neq \mathsf{head} * t \neq l * key(l) < k \leq key(t) \}$
39     return (l, ln, lmark, t)
40 } }
41 $\{ (l, ln, lmark, r).\ \exists N\, M\, C.\ \mathsf{TInv}(C, N, M, l, ln, lmark, r) * \mathsf{Fut}(M, l, ln, r) * r \neq \mathsf{head} * r \neq l * key(l) < k \leq key(r) \}$
```

Fig. 8. Proof outline showing that Harris set `traverse` prepares the `CAS` from `search`. The preparation guarantees that the `CAS` *will* maintain the invariant. We capture this with a future.

To satisfy the precondition when invoking `traverse` from `find`, observe that the invocation is of the form `traverse`$(k, \mathsf{head}, hn, hn)$ where hn stems from $hn = \mathsf{head.next}$. The invariant of `traverse`, $\mathsf{TInv}(C, N, \{\mathsf{head}, hn\}, \mathsf{head}, hn, hmark, hn)$, follows from the data structure invariant $\mathsf{Inv}(C, N)$. The future $\mathsf{Fut}(M, \mathsf{head}, hn, hn)$ can be obtained trivially via F-INTRO because the update $\mathsf{head.next} := hn$ has no effect if $next(\mathsf{head}) = hn$.

The postcondition of `traverse` contains the invariant $\mathsf{TInv}(C, N, M, l, ln, lmark, r)$ and the future $\mathsf{Fut}(M, l, ln, r)$. By applying rule F-INVOKE, we can use the future to prove the correctness of the case where the `CAS`$(l.\mathtt{next}, ln, r)$ at Line 14 (Figure 1) succeeds. The remaining facts of the postcondition state that `traverse` has found the part of the data structure that contains the search key k if present.

The most interesting part of the proof is the transition between Lines 31 and 32, particularly the transition from $\mathsf{Fut}(M, l, ln, t)$ to $\mathsf{Fut}(M, l, ln, tn)$. Here, we need to extend the marked segment M by adding tn. This step involves an application of F-SEQ to compose the update chunk for $l.\mathsf{next} := t$ with the one for $l.\mathsf{next} := tn$. We elaborate this step in detail, extending the discussion in §2.

We start from $\mathsf{Fut}(M, l, ln, t)$ and use rule F-FRAME to extend both sides of this future with $\mathsf{Inv}(\mathsf{C}(tn), tn)$. The resulting future can be rewritten into the form

$$\langle \hat{\mathsf{P}}(l, t, tn, ln) * \mathsf{Inv}(\emptyset, M \setminus \{l, t, tn\}) \rangle\ l.\mathsf{next} := t\ \langle \hat{\mathsf{Q}}(l, t, tn, t) * \mathsf{Inv}(\emptyset, M \setminus \{l, t, tn\}) \rangle$$

where

$$\hat{\mathsf{P}}(l, t, tn, u) \triangleq \mathsf{Inv}(\mathsf{C}(\{l, t, tn\}), \{l, t, tn\}) * next(l) = u * \neg mark(l)$$

$$\hat{\mathsf{Q}}(l, t, tn, u) \triangleq \mathsf{Inv}(\mathsf{C}(\{l, u, tn\}), \{l, t, tn\}) * next(l) = u * \neg mark(l) * (key(l), \infty] \subseteq flow(t)\ .$$

This future plays the role of $\langle p \rangle\ \mathsf{com}_1\ \langle q \rangle$ in our application of rule F-SEQ. To obtain the future playing the role of $\langle q \rangle\ \mathsf{com}_2\ \langle o \rangle$, we proceed in multiple steps. First, we use F-INTRO to derive

$$\langle \hat{\mathsf{P}}(l, t, tn, t) * next(t) = tn * mark(t) \rangle\ l.\mathsf{next} := tn\ \langle \hat{\mathsf{Q}}(l, t, tn, tn) \rangle\ .$$

To satisfy the premise of F-INTRO, we need to show that (i) the update $l.\mathsf{next} := tn$ is frame-preserving, i.e., the interface of the flow graph consisting of the nodes $\{l, t, tn\}$ does not change, and (ii) the invariants $\varphi^i(x)$ are preserved for all $x \in \{l, t, tn\}$. First observe that $\neg mark(l)$ and $\varphi^1(l)$ imply $\mathsf{IS}(l) \neq \emptyset$. Using $key(l) < \infty$, $\varphi^4(t)$, and the fixpoint equation defining insets, we obtain $\mathsf{IS}(t) = \mathsf{IS}(l) \cap (key(l), \infty] \neq \emptyset$. From $\varphi^3(t)$ and $mark(t)$ we obtain $key(t) \neq \infty$. Thus, using similar reasoning as above, we conclude $\mathsf{IS}(tn) = \mathsf{IS}(t) \cap (key(t), \infty] \neq \emptyset$. The inset and inflow of l are unaffected by the update, so its invariant is trivially preserved. For t, let $\mathsf{IS}'(t) = \emptyset$ denote the new inset. Since t is marked, this means that all its invariants are preserved and its content is empty. The new inset of tn is $\mathsf{IS}'(tn) = \mathsf{IS}(l) \cap (key(l), \infty]$. Observe that we have $\mathsf{IS}(tn) \subseteq \mathsf{IS}'(tn) \neq \emptyset$, so the invariants for tn are also maintained. Finally, to show that the interface of the modified region remains the same, it suffices to prove $\mathsf{IS}(tn) \cap (key(tn), \infty] = \mathsf{IS}'(tn) \cap (key(tn), \infty]$. This holds true if $key(t) < key(tn)$, which follows from $\varphi^2(tn)$ and $\emptyset \neq \mathsf{IS}(tn) \subseteq (key(t), \infty]$.

Next, we apply F-ACCOUNT for $next(t) = tn * mark(t)$ from the proof context and use F-FRAME to add the remaining part $\mathsf{Inv}(\emptyset, M \setminus \{l, t, tn\})$ of the segment M as a frame. This yields

$$\left\langle\, \hat{\mathsf{P}}(l, t, tn, t) * \mathsf{Inv}(\emptyset, M \setminus \{\, l, t, tn \,\}) \,\right\rangle\ l.\mathsf{next} := tn\ \left\langle\, \hat{\mathsf{Q}}(l, t, tn, tn) * \mathsf{Inv}(\emptyset, M \setminus \{\, l, t, tn \,\}) \,\right\rangle .$$

We can now use F-SEQ with this future and the one derived above. Note that the premise of the rule follows easily because com_1 and com_2 update the same memory location and $\mathsf{com}_2 = \mathsf{com}$. We obtain

$$\left\langle\, \hat{\mathsf{P}}(l, t, tn, ln) * \mathsf{Inv}(\emptyset, M \setminus \{\, l, t, tn \,\}) \,\right\rangle\ l.\mathsf{next} := tn\ \left\langle\, \hat{\mathsf{Q}}(l, t, tn, tn) * \mathsf{Inv}(\emptyset, M \setminus \{\, l, t, tn \,\}) \,\right\rangle .$$

Applying F-INFER and introducing a fresh existential M', we rewrite this into the form

$$M' = M \cup \{\, l, t, tn \,\} * \left\langle\, \mathsf{P}(M', l, ln, tn, ln) * t \in M' \,\right\rangle\ l.\mathsf{next} := tn\ \left\langle\, \mathsf{Q}(M', l, ln, tn, tn) \,\right\rangle .$$

Now, we apply F-ACCOUNT one more time for $t \in M'$ and use $tn \in N$, $ln \in M$, and $M \subseteq N$ from the proof context, to obtain

$$M' \subseteq N \ * \ \{\, l, ln, tn \,\} \subseteq M' \ * \ \mathsf{Fut}(M', l, ln, tn) \ .$$

This allows us to reestablish $\mathsf{TInv}(C, N, M', l, ln, lmark, tn) * \mathsf{Fut}(M', l, ln, tn)$. As M' and M are existentially quantified, we can finally rename M' to M, which yields the assertion on Line 32.

Checking Interference Freedom. We briefly discuss why the proof is interference-free relative to other threads performing set operations. First, all commands maintain $\mathsf{Inv}(C, N)$ and N can only grow larger. Next, assertions depending on the field `key` are interference-free since a node's `key` is never changed after initialization. Similarly, `mark` is only changed monotonically from *false* to *true*. Moreover, `next` is only changed for unmarked nodes (e.g., the proof guarantees $\neg mark(l)$ in the successful case of the `CAS` on Line 14 and the insert operation provides a similar guarantee). This is why assertions such as $mark(t)$ on Line 31 are interference-free. Because of that, the contents $\mathsf{C}(M \setminus \{l, t\})$ cannot change. Finally, futures constructed using rule F-INTRO are always interference-free. All remaining futures are constructed by accounting interference-free facts or by composing interference-free futures via F-SEQ.

7 HISTORIES

We next present an extension of our developed theory that allows us to reason about *separated computation histories*. We integrate a form of hindsight reasoning for propagating knowledge between current and past states—hindsight is a key technique to handle non-fixed linearization points [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010]. We develop the new theory again in the general setting of §3 and §4 and then apply it to our running example.

Read-and-validate Pattern. Optimistic implementations commonly have future-dependent linearization points: whether or not a thread's next action is its linearization point depends on future

```
42 val S = /* last index */
43 val arr = new Int[S+1] { 0, ..., 0 }
44 procedure copy() {
45   val res = new Int[S] { 0, ..., 0 }
46   { $arr \mapsto i_0, \ldots, i_S * res \mapsto j_0, \ldots, j_S * \bigwedge_{n=0}^{S} j_n \le i_n$ }
47   for (val k in [0, S]) res[k] = arr[k]
48   { $arr \mapsto i_0, \ldots, i_S * res \mapsto j_0, \ldots, j_S * \bigwedge_{n=0}^{S} j_n \le i_n$ } skip
49   { $arr \mapsto i_0, \ldots, i_S * res \mapsto j_0, \ldots, j_S * \diamond(arr \mapsto {}^{\backprime}i_0, \ldots, {}^{\backprime}i_S) * \bigwedge_{n=0}^{S} j_n \le {}^{\backprime}i_n \le i_n$ }
50   for (val k in [0, S]) {
51     { $arr \mapsto i_0, \ldots, i_S * res \mapsto j_0, \ldots, j_S * \diamond(arr \mapsto {}^{\backprime}i_0, \ldots, {}^{\backprime}i_S) * \bigwedge_{n=0}^{S} j_n \le {}^{\backprime}i_n \le i_n * \bigwedge_{n=0}^{k-1} j_n = {}^{\backprime}i_n$ }
52     if (res[k] != arr[k]) restart
53     { $arr \mapsto i_0, \ldots, i_S * res \mapsto j_0, \ldots, j_S * \diamond(arr \mapsto {}^{\backprime}i_0, \ldots, {}^{\backprime}i_S) * \bigwedge_{n=0}^{S} j_n \le {}^{\backprime}i_n \le i_n * \bigwedge_{n=0}^{k} j_n = {}^{\backprime}i_n$ }
54   }
55   { $arr \mapsto i_0, \ldots, i_S * res \mapsto {}^{\backprime}i_0, \ldots, {}^{\backprime}i_S * \diamond(arr \mapsto {}^{\backprime}i_0, \ldots, {}^{\backprime}i_S)$ }
56   return res
57 }

58 procedure inc(k: Int) {
59   if (k > S) return false
60   FAA(arr[k], 1)
61   return true
62 }
```

Fig. 9. A simple array of counters. The counters can be incremented individually. The entire array can be snapshot in an optimistic fashion, resulting in a non-fixed linearization point.

interferences from other threads. This issue often arises in uses of the *read-and-validate pattern* where threads (i) read out some shared heap region, (ii) later on validate the read value, and (iii) succeed with their operation if the validation succeeds or roll-back otherwise. The read and validation step are neither executed atomically nor within a critical/lock-protected section. Hence, the read heap region is subject to interference and may change.

The Harris set employs the read-and-validate pattern. Method `find`, for instance, validates (some of) the values read by `traverse` by re-reading them with the `CAS` on Line 14. If the `CAS` fails, so does the validation and `find` rolls back by restarting. Method `traverse` implements the pattern with a more intricate validation and roll-back mechanism: the `next` field of node t read on Line 5 is validated by inspecting t's mark bit, Line 7, and if the mark bit is set then the validation of `traverse`'s search for the right node fails and continues with the subsequent nodes.

The pattern is not restricted to search structures. For an example, consider the counter array from Figure 9. The implementation maintains an array *arr* of $S+1$ integer counters. Counters are individually increased by 1 using `inc`. A snapshot of the counter array is created by `copy`. As a first stage, a simple copy *res* of the array is created by reading out the individual counter values non-atomically, Line 47. In a second stage, the copy is validated against the current counter values, Lines 50 to 52. The procedure is restarted if there are any discrepancies. Otherwise, the copy is guaranteed to be a consistent snapshot of the counter array as of the moment immediately after the last counter was read on Line 47. It is worth noting that, while the copy is being validated, some counters that have been validated already may be changed. Nevertheless, the validation succeeds (rightfully so).

The read-and-validate pattern in `copy` results in a future-dependent linearization point. As alluded to above, the linearization point is the moment the last counter is read on Line 47. However, this moment depends on whether the subsequent validation *will succeed*. This, in turn, is unpredictable and not under the control of the executing thread.

To handle this in a proof, we suggest the following strategy which mimics closely the behavior of the implementation. During the first stage, we track the interference-free fact that the entries of the copy array *res* are less than or equal to the current value of the corresponding counter, $j_n \le i_n$ for all n. This fact follows easily from the counters increasing monotonically. Then, we snapshot the current counter array into a past predicate, Line 49. (Technically, this requires the `skip` on Line 48, cf. Lemma 7.11 below.) Note that we rename the counter values i_n under the past predicate to $\grave{i}_n$. Because $i_n = \grave{i}_n$ at the moment of the snapshot, we obtain $j_n \leq \grave{i}_n$ for all n. However, the equality $i_n = \grave{i}_n$ is not interference-free as the counters may change, but the point in time and thus the values the past predicate refers to are fixed. This justifies our renaming to $\grave{i}_n$. As before, we record that the counter values under the past predicate are less than or equal to the current counter values, $\grave{i}_n \leq i_n$ for all n. During the second stage, a successful validation implies that the k-th copy is equal to the current counter, $j_k = i_k$. Together with the estimate $j_k \leq \grave{i}_k \leq i_k$, we obtain $j_k = \grave{i}_k$. Overall, this means that the copy *res* corresponds to the counter array snapshot in the past predicate, Line 55. Hence, *res* is a consistent snapshot of the counters in the sense that there was a point in time where the counter array was equal to *res*—the operation is linearizable as desired. In the following, we formalize past predicates and show their usefulness for linearizability proofs.

History Separation Algebras. Recall that our states are taken from a separation algebra $(\Sigma, *, \mathsf{emp})$. We refer to a non-empty sequence of states $\sigma \in \Sigma^+$ as a *computation history*. Computation histories also form a separation algebra by lifting the composition on states as follows. First, for sequences $\sigma = \mathsf{s}_1 \cdots \mathsf{s}_n$ and $\tau = \mathsf{t}_1 \cdots \mathsf{t}_m$ with $\cdot$ denoting sequence concatenation, the composition $\sigma * \tau$ is defined, written $\sigma \# \tau$, iff $n = m$ and for all i we have $\mathsf{s}_i \# \mathsf{t}_i$. In this case, we let $\sigma * \tau \triangleq (\mathsf{s}_1 * \mathsf{t}_1) \cdots (\mathsf{s}_n * \mathsf{t}_n)$. The set of units is given by emp^+.

Lemma 7.1. $(\Sigma^+, *, \mathsf{emp}^+)$ *is a separation algebra.*

Predicates $a, b, c \subseteq \Sigma^+$ now refer to sets of computations. We lift the semantics of commands to computation predicates in the expected way.

Definition 7.2. $[\![\![\mathsf{com}]\!]\!](\sigma.\mathsf{s}_1) \triangleq \{\, \sigma \cdot \mathsf{s}_1 \cdot \mathsf{s}_2 \mid \mathsf{s}_2 \in [\![\mathsf{com}]\!](\mathsf{s}_1) \,\}$.

However, the locality assumption (LocCom) on the semantics of commands that is needed for the soundness of framing does not necessarily carry over from state predicates to computation predicates. If we want to frame computation predicates, we have to make an additional assumption.

Definition 7.3. A predicate $a \subseteq \Sigma^+$ is *frameable*, if it satisfies $\forall \sigma. \forall \mathsf{s}.\ \sigma \cdot \mathsf{s} \in a \implies \sigma \cdot \mathsf{s} \cdot \mathsf{s} \in a$.

Lemma 7.4. *If c is frameable,* $[\![\![\mathsf{com}]\!]\!](a * c) \subseteq [\![\![\mathsf{com}]\!]\!](a) * c$.

We lift the semantics of concurrency libraries to history separation algebras $(\Sigma_\mathsf{G} \times \Sigma_\mathsf{L})^+$. The notions of initial and accepting configurations as well as soundness remain unchanged except that they now range over computation predicates instead of state predicates. The technical details of this lifting are straightforward. The soundness guarantee in Theorem 4.2 continues to hold for history separation algebras modulo a subtlety. We can only apply rule FRAME if the predicate to be added is frameable in the sense of Definition 7.3.

Theorem 7.5 (Soundness). $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathsf{st}\ \{\, b \,\}$ *and* $\boxplus_\mathbb{I} \mathbb{P}$ *and* $a \in \mathbb{P}$ *imply* $\models \{\, a \,\}\ \mathsf{st}\ \{\, b \,\}$.

Frameable Computation Predicates. We next discuss general principles for constructing frameable computation predicates from state predicates.

Definition 7.6. A state predicate $p \subseteq \Sigma$ yields the following predicates over computations histories:
(i) The *now predicate* $\_p \ \triangleq\ \Sigma^* \cdot p$.
(ii) The *past predicate* $\diamond p \ \triangleq\ \Sigma^* \cdot p \cdot \Sigma^*$.

The now predicate refers to the current state. The past predicate allows us to track auxiliary information about the computation. These predicates work well in our setting in that they are frameable.

Lemma 7.7. *(i) $\_p$ and $\diamond p$ are frameable. (ii) If a and b are frameable, so are $a * b$, $a \cap b$, and $a \cup b$.*

Frameability is not preserved under complementation and separating implication. However, the now operator is compatible with the SL operators in a strong sense.

Lemma 7.8. *$\_(p \oplus q) = \_p \oplus \_q$ for all $\oplus \in \{\cap, \cup, *, -\!\!*\}$, $\_(\overline{p}) = \overline{\_p}$, $false = \_false$, $true = \_true$, and $\_p \subseteq \_q$ iff $p \subseteq q$.*

For the past operator, we rely on the properties stated by the following lemma. In particular, the last equivalence justifies rule H-INFER used in §2.

Lemma 7.9. *$\_p \subseteq \Diamond p$, $true = \Diamond true$, $true * \Diamond p = \Diamond(p * true)$, $false = \Diamond false$, $\Diamond(p * q) \subseteq \Diamond p * \Diamond q$, $\Diamond p -\!\!* \Diamond q \subseteq \Diamond(p -\!\!* q)$, $\Diamond(p \cap q) \subseteq \Diamond p \cap \Diamond q$, $\Diamond(p \cup q) = \Diamond p \cup \Diamond q$, and $\Diamond p \subseteq \Diamond q$ iff $p \subseteq q$.*

The interplay between computation predicates and commands is stated in the following lemma. Recall that we defined $wp(\texttt{com}, a) \triangleq \{ \sigma \mid [\![\texttt{com}]\!](\sigma) \subseteq a \}$.

Lemma 7.10. *We have (i) $wp(\texttt{com}, \_p) = \_wp(\texttt{com}, p)$, and (ii) $wp(\texttt{com}, \Diamond p) = \Diamond p \cup wp(\texttt{com}, \_p)$.*

The first identity of Lemma 7.10 implies that interference checking for a now predicate reduces to inference checking for the underlying state predicate. The second identity implies that past predicates are interference-free for all commands.

Next we justify rule H-INTRO used in §2. Recall that this rule provides a way to record information about the current state in a past predicate so that we can use this information later in the proof. This involves a stuttering step.

Lemma 7.11. *$\_p \subseteq wp(\texttt{skip}, \_p * \Diamond p)$.*

Hindsight Reasoning. We now use history separation algebras to justify the hindsight reasoning principle introduced in §2. The key idea is *state-independent quantification*, and best explained with reference to an assertion language. An assertion language over computations will support quantified logical variables. As those quantifiers live on the level of computations, the resulting valuation of the logical variables will be independent of (the same for all) the states inside the computation. This means facts that we learn about the variables in one state will also be true in all other states. In particular, if we learn facts about a quantified variable now, we can draw conclusions in hindsight. We illustrate this on an example. In the assertion $\exists v.(\Diamond(x \mapsto v) * \_(v = 0))$, the logical variable v is quantified on the level of computations, meaning its value is independent of the actual state in the computation. We learn that now v is zero, and since the valuation is state independent, v has also been zero when x pointed to it. Hence, from the now state we can conclude, in hindsight, that also $\Diamond(x \mapsto 0)$ holds. This is indeed a consequence of the previous assertion (entailment holds). Rather than moving to an assertion language, we formalize this reasoning on the semantic level.

For hindsight reasoning, we construct a product separation algebra $\Sigma \times I$. The first component is the above state separation algebra $(\Sigma, *, \texttt{emp})$. We refer to the second component as the *valuation* separation algebra $(I, *, I)$, because its elements can be understood as variable valuations. There are no requirements on I, it is just an arbitrary set, but we note that it is also the set of units. The multiplication $*$ between valuations $i, j \in I$ is defined if and only if $i = j$, in which case $i * i = i$.

The semantics of commands $[\![\texttt{com}]\!]$ is lifted to predicates of the product separation algebra by leaving the valuation component untouched. We then lift $\Sigma \times I$ to a separation algebra of computation histories $(\Sigma \times I)^+$ as before. We adapt the definition of $\_p$ and $\Diamond p$ for $p \subseteq (\Sigma \times I)^+$ so that the valuation component is kept constant over the whole computation, in accordance with the lifted semantics of commands:

$$\_p \triangleq \{ (\Sigma \times \{ i \})^* \cdot (\mathrm{s}, i) \mid (\mathrm{s}, i) \in p \} \qquad \Diamond p \triangleq \{ (\Sigma \times \{ i \})^* \cdot (\mathrm{s}, i) \cdot (\Sigma \times \{ i \})^* \mid (\mathrm{s}, i) \in p \} .$$

Separation logic assertions are called pure, if they are independent of the heap and only refer to the valuation of logical variables. In the semantic setting, we define a predicate p to be *pure*, if it leaves the heap unconstrained, $p = \Sigma \times \mathcal{J}$ for some $\mathcal{J} \subseteq I$. The following lemma then justifies rule H-HINDSIGHT used in §2.

LEMMA 7.12. *If p is pure, then* $\_p * \diamond q \;=\; \diamond(p * q)$.

Linearizability in Hindsight. Past predicates together with the hindsight principle have an appealing application in linearizability proofs: they allow for retrospective linearization. That is, the proof does not need to observe the very moment when a thread executes the linearization point. It suffices to show that the linearization point must have occurred in the past, after the remainder of the thread's execution has been observed. This is inspired by concurrent data structure practice, like the read-and-validate pattern from before. To support this kind of linearizability argument, we extend the proof system $\Vdash_{lin}$ from §4 with rule LIN-PAST from Figure 6, repeated here for convenience:

$$\text{LIN-PAST}\ \frac{a \subseteq \diamond\big(\mathsf{CSS}(C) \cap \mathsf{UP}(C, C, k, v)\big)}{\mathbb{P}, \mathbb{I} \Vdash_{lin} \{\, \mathsf{Obl}(\Psi) * a \,\}\ \mathtt{skip}\ \{\, \mathsf{Ful}(\Psi, v) * a \,\}}\ .$$

The rule formalizes our intuition. It trades the update token $\mathsf{Obl}(\Psi)$ for $\mathsf{Ful}(\Psi, v)$ if a past predicate can certify that the sequential specification was satisfied at some point during the computation, i.e., a linearization point definitely occurred. It is worth stressing that the way past predicates are introduced (cf. Lemmas 7.9 and 7.10) guarantees that they refer to a moment during the execution of the corresponding operation, as required for linearizability. Also observe that the precondition of the rule requires the linearization point to be pure. With the restriction to pure linearization points, we avoid the complexity of ensuring that there is a one-to-one correspondence between updates of the logical contents of the structure and threads *claiming* an update as their linearization point. Put differently, our rule exploits the fact that arbitrarily many threads may linearize in a single state that satisfies a pure case of the sequential specification.

Proving Linearizability of the Harris Set. By using the now predicate, we obtain a conservative extension of separation logic. In the following, the application of the now operator is kept implicit: a state predicate that occurs in a context expecting a computation predicate is interpreted as a now predicate. This is justified by Lemma 7.8.

We demonstrate the reasoning power of the resulting logic by proving linearizability of the Harris set `search` operation. The proof outline is in Figure 10, reusing our earlier proof of `traverse`. The code of `find` makes the semantics of the `CAS` explicit. We have also eliminated the case $ln == r$ before the `CAS`. This focuses the discussion on a single linearization point.

As in §2, we decorate logical variables occurring below a past operator with a prime to consistently rename existentially quantified variables in order to avoid clashes with variables describing the current state. For example, $`next(x)$ will refer to the old value of $x$'s `next` field in some past state.

We proceed with the proof. First, we focus on the overall linearizability argument in `search`. To that end, observe the history assertion resulting from a call to `find`, Line 89:

$$\diamond\big(\, \mathsf{Inv}(`C, `N) \;*\; r \in `N \;*\; k \in `\mathsf{KS}(r)\big) \;*\; \neg`mark(r) \;*\; key(r) = `key(r)\ .$$

Following our discussion from §2, the keysets guide the proof in the sense that $k \in `\mathsf{KS}(r)$ means that $r$ is the decisive node for `search(`$k$`)`. We localize the reasoning to this decisive node by rewriting the invariant $\mathsf{Inv}(`C, `N)$ under the past operator along Lemma 5.2:[2]

$$\mathsf{Inv}(\, `C \setminus \{\, k \,\},\ `N \setminus \{\, r \,\}\,) \;*\; \mathsf{Inv}(\, `\mathsf{C}(r),\ \{\, r \,\}\,)\ .$$

The lemma is applicable as its precondition $\mathsf{C}(`N \setminus \{\, r \,\}) \cap \{\, k \,\} = \varnothing$ follows from contraposition: if there was a non-r node with k in its contents, then k was also in its keyset by the invariant, which

[2]Compared to §2, we record here the full invariant $\mathsf{Inv}(`C, `N)$ under the past operator, not just the predicate $\mathsf{Node}(r)$. This is needed to be compatible with the sequential specification and rule LIN-PAST.

```
63 $\mathsf{Past}(r) \triangleq key(r) = {}^{\backprime}key(r) * ({}^{\backprime}mark(r) \Rightarrow mark(r)) * \diamondminus(\,\mathsf{Inv}({}^{\backprime}C, {}^{\backprime}N) * r \in {}^{\backprime}N * {}^{\backprime}(k \in \mathsf{KS}(r))\,)$
64 $\{\, \exists C\, N.\ \mathsf{Inv}(C, N) * -\infty < k < \infty \,\}$
65 procedure find(k: K) : N * N {
66   val hn, hmark = atomic {head.next, head.mark}
67   $\{\, \mathsf{TInv}(C, N, \{\mathsf{head}, hn\}, \mathsf{head}, hn, hmark, hn) * \mathsf{Fut}(\{\mathsf{head}, hn\}, \mathsf{head}, hn, hn) \,\}$
68   val l, ln, lmark, r = traverse(k, head, hn, hmark, hn)
69   $\{\, \mathsf{TInv}(C, N, M, l, ln, lmark, r) * \mathsf{Fut}(M, l, ln, r) * r \neq \mathsf{head} * r \neq l * key(l) < k \leq key(r) \,\}$
70   val succ = atomic { // CAS
71     l.next == ln && l.mark == lmark ? {
72       $\{\, \mathsf{Inv}(C, N) * \mathsf{Fut}(M, l, ln, r) * \mathsf{C}(M) \subseteq \mathsf{C}(l, r) * next(l) = ln$
         $* \neg mark(l) * \{\, l, ln, r \,\} \subseteq M \subseteq N * key(l) < k \leq key(r) \,\}$
73       l.next := r
74       $\{\, \mathsf{Inv}(C, N) * \{\, l, r \,\} \subseteq N * (key(l), \infty] \subseteq flow(r)$
         $* key(l) < k \leq key(r) \,\}$
75       skip
76       $\{\, \mathsf{Inv}(C, N) * \{\, l, r \,\} \subseteq N * (key(l), \infty] \subseteq flow(r)$
         $* key(l) < k \leq key(r) * \mathsf{Past}(r) \,\}$
77     } true : false
78   }
79   $\{\, \mathsf{Inv}(C, N) * \{\, l, r \,\} \subseteq N * (succ \Rightarrow \mathsf{Past}(r)) \,\}$
80   if (succ && !r.mark) {
81     $\{\, \mathsf{Inv}(C, N) * \{\, l, r \,\} \subseteq N * \neg{}^{\backprime}mark(r) * \mathsf{Past}(r) \,\}$
82     return (l, r)
83   } else find(k)
84 }
85 $\{\, (l, r).\ \exists C\, N.\ \mathsf{Inv}(C, N) * \{\, l, r \,\} \subseteq N * \neg{}^{\backprime}mark(r) * \mathsf{Past}(r) \,\}$

86 $\{\, \exists C\, N.\ \mathsf{Inv}(C, N) * -\infty < k < \infty$
       $* \mathsf{Obl}(\mathtt{search}(k)) \,\}$
87 procedure search(k: K) : Bool {
88   val _, r = find(k)
89   $\{\, \mathsf{Inv}(C, N) * r \in N * \neg{}^{\backprime}mark(r)$
     $* \mathsf{Past}(r) * \mathsf{Obl}(\mathtt{search}(k)) \,\}$
90   val res = r.key == k
91   $\{\, \mathsf{Inv}(C, N) * r \in N * \mathsf{Obl}(\mathtt{search}(k))$
     $* \diamondminus(\mathsf{Inv}({}^{\backprime}C, {}^{\backprime}N) * res \Leftrightarrow k \in {}^{\backprime}C) \,\}$
92   $\{\, \mathsf{Inv}(C, N) * \mathsf{Ful}(\mathtt{search}(k), res) \,\}$
93   return res
94 }
95 $\{\, res.\ \exists C\, N.\ \mathsf{Inv}(C, N) * \mathsf{Ful}(\mathtt{search}(k), res) \,\}$
```

Fig. 10. Proof outline showing that Harris set search preserves the data structure invariant and is linearizable.

contradicts the disjointness of keysets because $k \in {}^{\backprime}\mathsf{KS}(r)$. So we get $k \in {}^{\backprime}\mathsf{C}(r)$ iff $k = {}^{\backprime}key(r)$. By the above localization, this means $k \in {}^{\backprime}C$ iff $k = {}^{\backprime}key(r)$. Because the key of r has not changed, $key(r) = {}^{\backprime}key(r)$, we arrive at $k \in {}^{\backprime}C$ iff $k = key(r)$. That is, the current value of r.key reveals whether or not k has been in the contents of the data structure at some point in the past. With this, it is immediate that the past predicate certifies the existence of a linearization point for the usual sequential specification of search:

$$\{\, C.\ \mathsf{CSS}(C) \,\}\ \mathtt{search}(k)\ \{\, res.\ \mathsf{CSS}(C) * res \Leftrightarrow k \in C \,\}\ .$$

We know that res is equal to the truth of the equality $k = key(r)$. Hence, we can linearize in hindsight using rule LIN-PAST from above.

To prove the postcondition of find, we need to establish that at some point during the execution of find, $k \in \mathsf{KS}(r)$ and $\neg mark(r)$ were satisfied. This is on Line 74, but we determine r's mark bit only later as execution continues past the condition on Line 80. The high-level proof idea is, thus, to record r's keyset and mark bit on Line 74 in the past assertion shown on Line 76 (we explain this step in more detail below). We then propagate the assertion on Line 76 into the *then* branch of the conditional (using Lemma 7.10). The logical variables link the past and current state, which allows us to apply hindsight reasoning (rule H-HINDSIGHT). Specifically, on Line 81 we know that $mark(r)$ is false in the current state. Then, using ${}^{\backprime}mark(r) \Rightarrow mark(r)$, we can conclude that ${}^{\backprime}mark(r)$ is also false, thus learning retrospectively the crucial fact about the past state at Line 74. We then transfer this pure fact into the past predicate to derive $\mathsf{C}(r) = \{\, key(r)\,\}$ and thus $res \Leftrightarrow k \in C'$ on Line 91.

We briefly discuss the mechanical aspects of deriving the past predicate on Line 76. We start with the intermediate assertion established on Line 74 in the earlier proof. First, using $key(l) < k$ and $(key(l), \infty] \subseteq flow(r)$ we derive $k \in \mathsf{IS}(r)$. Then, using $k \leq key(r)$ we obtain $k \in \mathsf{KS}(r)$. We then perform a stuttering step to record a copy of this assertion below a past operator using rule H-INTRO. The resulting assertion is not interference-free since the current state of r referred to inside the past assertion can be changed by concurrent threads. So we perform a series of weakening steps by introducing fresh logical variables to arrive at the assertion on Line 76.

It is worth pointing out that the proof of `find` relies on the future constructed in §6 and the fact that its update is pure, Line 73. To see this, we rewrite the assertion from Line 72 along Lemma 5.2:

$$\mathsf{Inv}(C \setminus \mathsf{C}(\{\, l, r\,\}), N \setminus M) \ * \ \mathsf{Inv}(\mathsf{C}(\{\, l, r\,\}), M) \ * \ \mathsf{Fut}(M, l, ln, r)\ .$$

Then, we frame out $\mathsf{Inv}(C \setminus \mathsf{C}(\{\, l, r\,\}), N \setminus M)$ and use the remaining $\mathsf{Inv}(\mathsf{C}(\{\, l, r\,\}), M)$ to invoke the future $\mathsf{Fut}(M, l, ln, r)$. This gives $\mathsf{Inv}(\mathsf{C}(\{\, l, r\,\}), M)$ which, combined with the frame, results in $\mathsf{Inv}(C, N)$. Consequently, the update does not alter the logical contents of the structure so that rule LIN-NONE is applicable. This also justifies framing out the update token before invoking `find` from within `search` on Line 88—the update token is not needed for the proof of `find`.

We note that throughout the entire proof, all explicit inductive reasoning was carried out at the level of the program logic in lock-step with the program execution, using only local facts about the nodes in the data structure captured by the resource invariant, futures, and history predicates. In particular, we did not need explicit inductive reasoning over heap graph predicates or computation histories. All such reasoning is carried out *for free* by our developed meta-theory.

8 PROTOTYPE IMPLEMENTATION

We substantiate our claim that the presented techniques aid automation and are useful in practice. To that end, we implemented a `C++` prototype called `plankton` [Meyer et al. 2022a]. `plankton` takes as input the program under scrutiny and a candidate node invariant. It then fully automatically generates a proof within our novel program logic, establishing that the given program is linearizable and does adhere to the given invariant. We give a brief overview of `plankton`'s proof generation and report on our findings. We stress that the present paper focuses on the theoretical foundations and as such does not give a detailed discussion of `plankton`'s implementation.

Implementation. The proof generation in `plankton` is implemented as a fixpoint computation that saturates an increasing sequence $\mathbb{I}_0 \subseteq \mathbb{I}_1 \subseteq \cdots$ of interference sets [Henzinger et al. 2003]. Initially, the interference set is empty, $\mathbb{I}_0 \triangleq \varnothing$. Once $\mathbb{I}_k$ has been obtained, a proof of the input program with respect to $\mathbb{I}_k$ is constructed and the interferences $\mathbb{I}_{new}$ discovered during this proof are recorded, yielding $\mathbb{I}_{k+1} \triangleq \mathbb{I}_k \cup \mathbb{I}_{new}$. A fixpoint $\mathbb{I}_{lfp} \triangleq \mathbb{I}_k$ is reached if no new interference is found, $\mathbb{I}_k = \mathbb{I}_{k+1}$. The proof generated for $\mathbb{I}_k$ is then the overall proof for the input program.

For efficiency reasons, it is crucial to reduce the size of the computed interference sets [Vafeiadis 2010b]. We reduce an interference set $\mathbb{I}$ by dropping any interference $(o, \mathtt{com})$ that is already covered, that is, there is $(r, \mathtt{com}) \in \mathbb{I}$ with $o \subseteq r$.

Given an interference set $\mathbb{I}$, `plankton` constructs a proof $\mathbb{I} \Vdash \{\, p_0\,\}\ \mathtt{fun}\ \{\, q\,\}$ for each function `fun` of the input program. The proof construction starts from the precondition p_0, which captures just the invariant, as done, e.g., in Figure 10. From there, the rules of the program logic (Figure 5) are applied to inductively construct the postcondition. As $\mathbb{I}$ is fixed, plankton does not track the predicates $\mathbb{P}$. We elaborate on the interesting ingredients of the proof construction.

Rule COM-SEM for atomic commands `com` requires `plankton` to compute $[\![\mathtt{com}]\!](p)$ for some precondition p. The behavior of $[\![\mathtt{com}]\!](p)$ is prescribe by the standard axioms of separation logic [O'Hearn et al. 2001]. If com updates the heap, however, we have to additionally infer a flow footprint such that, roughly, (i) all nodes updated by com are contained in the footprint, (ii) the interface of the footprint remains unchanged by the update, and (iii) after the update all nodes inside the footprint still satisfy their node-local invariant. As discussed in §5, this localizes the update to the footprint: the nodes outside the footprint continue to satisfy the invariant. plankton chooses the footprint by collecting all nodes whose (non-flow) fields are updated by com and adds those nodes that are reachable in a small, constant number of steps. If this choice does not satisfy (i), verification fails. The restriction to finite footprints is essential for automating (ii) and (iii). Yet, the restriction does not limit our approach: unbounded footprints are handled with futures, as seen in §6. Conditions (ii) and (iii) are then encoded into SMT and discharged using Z3 [de Moura and Bjørner 2008]. Lastly, we apply the interferences $\mathbb{I}$ to $[\![\texttt{com}]\!](p)$. The result is $q \triangleq [\![\mathbb{I}]\!]([\![\texttt{com}]\!](p))$ whose computation is inspired by [Vafeiadis 2010b]. Overall, we obtain $\mathbb{I} \Vdash \{\, p \,\}\ \texttt{com}\ \{\, q \,\}$.

Rule LOOP requires a loop invariant I for program $\texttt{st}^*$ and precondition p such that $\mathbb{I} \Vdash \{\, I \,\}\ \texttt{st}\ \{\, I \,\}$ and $p \subseteq I$. To find one, plankton generates a sequence $I_0, I_1, \ldots$ of candidates. The first candidate is $I_0 \triangleq p$. Candidate I_{n+1} is obtained from a sub-proof $\mathbb{I} \Vdash \{\, I_n \,\}\ \texttt{st}\ \{\, I'_n \,\}$ whose pre- and postcondition are joined, i.e., $I_{n+1} \triangleq I_n \sqcup I'_n$. Intuitively, this join corresponds to the disjunction $I_n \cup I'_n$. For performance reasons, however, plankton uses a disjunction-free domain [Rival and Mauborgne 2007; Yang et al. 2008], which means the join is actually weaker than union. A loop invariant $I \triangleq I_n$ is found, if the implication $I_{n+1} \subseteq I_n$ holds.

A core aspect of our novel program logic are history and future predicates. plankton tries to construct a strongest proof for the input program. Hence, new history and future predicates are added eagerly to an assertion p whenever it participates in a join or interference is applied to it. The rational behind this strategy is to *save* information from p in a history/future before it is lost. More specifically, all boxed points-to predicates from p that are subject to interference are added to a new history predicate. New futures are introduced either from scratch with rule F-INTRO followed by rule F-ACCOUNT or from existing futures with rules F-SEQ and F-FRAME. It is worth pointing out that plankton uses rule F-ACCOUNT only to account duplicable facts, as in the proof from Figure 8. The introduction of futures is guided by a set of candidates. These candidates are computed upfront by collecting all CAS commands in the input program. A CAS may be dropped from the candidates if its footprint is statically known to be finite, e.g., because it only updates the mark bit of a pointer or inserts a new (and thus owned) node. The approach discovers the necessary futures needed for our experiments. Avoiding unnecessary futures produced by this method is considered future work.

Evaluation. We used plankton to automatically verify linearizability of fine-grained state-of-the-art set implementations from the literature: a lock-coupling set [Herlihy and Shavit 2008, Chapter 9.5], the Lazy set [Heller et al. 2005], FEMRS tree [Feldman et al. 2018] which is a variation of the contention-friendly binary tree [Crain et al. 2013, 2016], Vechev&Yahav 2CAS set [Vechev and Yahav 2008, Figures 8 and 9], Vechev&Yahav CAS set [Vechev and Yahav 2008, Figure 2], ORVYY set [O'Hearn et al. 2010], Michael set [Michael 2002], Harris set [Harris 2001], and a variation with wait-free search of the Michael and Harris set algorithms. For the FEMRS tree, plankton cannot handle the maintenance operations because they have updates with an unbounded footprint that is not traversed. This is a limitation of our current future reasoning. However, we are not aware of any other tool that can automatically verify even this simplified version of FEMRS trees. Also, plankton is the first tool to automate hindsight reasoning for the Harris set.

The results are summarized in Table 1. The first three columns of the table list (i) the number of iterations until the fixpoint $\mathbb{I}_{lfp}$ is reached, (ii) the size of $\mathbb{I}_{lfp}$, and (iii) the number of future candidates. The next five columns list the percentage of runtime spent on (iv) rule COM-SEM, (v) future reasoning, (vi) history reasoning, (vii) joins, and (viii) applying interferences. The last column gives (ix) the

Table 1. Experimental results for verifying set implementations with `plankton`, conducted an Apple M1 Pro.

Benchmark	#Iter	$\#\mathbb{I}_{lfp}$	#Cand	Com.	Fut.	Hist.	Join	Inter.	Lineariz.
Fine-Grained set	2	5	2	11%	15%	43%	15%	8%	46*s* ✓
Lazy set	2	6	2	10%	13%	54%	11%	5%	77*s* ✓
FEMRS tree (no maintenance)	2	5	2	19%	0%	49%	1%	9%	130*s* ✓
Vechev&Yahav 2CAS set	2	3	1	14%	0%	33%	31%	9%	125*s* ✓
Vechev&Yahav CAS set	2	4	1	15%	7%	39%	23%	6%	54*s* ✓
ORVYY set	2	3	0	17%	0%	40%	26%	6%	47*s* ✓
Michael set	2	4	2	11%	29%	30%	15%	6%	306*s* ✓
Michael set (wait-free search)	2	4	2	11%	28%	30%	15%	6%	246*s* ✓
Harris set	2	4	2	7%	8%	19%	32%	4%	1378*s* ✓
Harris set (wait-free search)	2	4	2	8%	10%	17%	34%	3%	1066*s* ✓

overall runtime, averaged across 10 runs, and the linearizability verdict (success is marked with ✓). Across all benchmarks we observe that two iterations are sufficient to reach the fixpoint $\mathbb{I}_{lfp}$: the first iteration discovers all interferences, the second iteration confirms that none are missing. This is remarkable because the first iteration uses $\mathbb{I}_0 \triangleq \varnothing$, i.e., considers the sequential setting. Further, we observe that most benchmarks spend significantly more time reasoning about the past than the future. The reason is twofold. (1) Introducing new futures either succeeds, meaning that a future candidate is resolved and can be ignored going forward, or it *fails fast*, which we attribute to Z3 finding counterexamples much faster than proving the validity of our SMT encoding of heap updates. (2) For histories, we do not have a heuristic identifying candidates. Instead, we eagerly introduce histories upon interference. We also apply hindsight reasoning eagerly. Lastly, we observe that the overall runtime tends to increase with the nesting depth and complexity of loops, as `plankton` requires several loop iterations (often between 3 and 5) to find an invariant. A proper investigation of how finding loop invariants affects the overall runtime is future work.

We also stress-tested `plankton` with faulty variants of the benchmarks. All buggy benchmarks failed verification. Note that `plankton` does not implement error explanation techniques, which are beyond the scope of the present paper.

9 RELATED WORK

Program Logics with History and Prophecy. Program logics have been extended by mechanisms for temporal reasoning in various ways [Abadi and Lamport 1991; Bell et al. 2010; Delbianco et al. 2017; Fu et al. 2010; Gotsman et al. 2013; Hemed et al. 2015; Liang and Feng 2013; Manna and Pnueli 1995; Parkinson et al. 2007; Schneider 1997; Sergey et al. 2015].

The work closest to ours is HLRG [Fu et al. 2010], a separation logic based on local rely-guarantee [Feng 2009] that tracks and reasons about history information, and its variation [Gotsman et al. 2013]. The separation algebra behind HLRG is constructed like ours. The focus of [Fu et al. 2010; Gotsman et al. 2013], however, are temporal operators in the assertion language and means of reasoning about them in the program logic. We only have now and past, but add the ability to propagate information between them. The simplicity of our approach enables automation ([Fu et al. 2010; Gotsman et al. 2013] has not been implemented in any automated or interactive tool, as far as we know). A minor difference is that we work over general separation algebras to integrate flows [Krishna et al. 2018, 2020b] easily and make the requirement of frameability explicit.

A program logic with temporal information based on different principles appears in [Delbianco et al. 2017; Sergey et al. 2015]. There, histories are sub-computations represented by timestamped sets of events. The product of histories is disjoint union. While highly expressive, we are not aware of implementations of the approach. Since our goal is automated proof construction, we strive for assertions that are simple to prove, instead.

A separation logic for proving producer-consumer applications is proposed in [Bell et al. 2010]. The logic uses history but is domain-specific and provides no mechanism to reason about the temporal development. A non-blocking stack with memory reclamation is verified in [Parkinson et al. 2007]. The proof relies on history information stored in auxiliary variables and manipulated by ghost code. The ghost code is justified by informal arguments (outside the program logic). We do not consider memory reclamation as it can be verified separately [Meyer and Wolff 2019, 2020]. Beyond linearizability, history variables have recently been used to give specs to non-linearizable objects [Hemed et al. 2015].

Several separation logics have been extended with prophecy variables [Jung et al. 2020; Liang and Feng 2013], which complement history-based reasoning with a mechanism to speculate about future events. However, prophecies are not well-suited for automatic proofs because they rely on backward reasoning [Bouajjani et al. 2017].

History reasoning has been used early on in program verification [Abadi and Lamport 1991]. In program logics [Manna and Pnueli 1995; Schneider 1997], the focus has been on causality formulas which, in our notation, take the form $\_p \Rightarrow \diamond q$. Our history reasoning is more flexible, in particular incorporates hindsight reasoning (see below), and inherits the benefits of modern separation logics.

Overall, the existing program logics with history are heavier than ours while missing the important trick of communicating information from the current state to the past by means of logical variables shared between the two.

Hindsight Reasoning. The idea of propagating information from the current state into the past is inspired by the recent hindsight theory [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010]. Hindsight lemmas ensure that information about a data structure obtained by sequential reasoning (typically the reachability of keys) remains valid for concurrent executions. The argument behind such results is that the existence of a sequentially-reachable state implies the existence of a related concurrently-reachable state in the past. The implication requires that updates to the structure do not interfere with the reachability condition one tries to establish (*forepassing condition* in [Feldman et al. 2020]).

So far, hindsight reasoning has been limited to pencil-and-paper proofs, with the exception of the `poling` tool [Zhu et al. 2015]. `poling` automates the specific hindsight lemma of O'Hearn et al. [2010]. Unlike histories, it does not immediately generalize to other forms of retrospective reasoning, like [Feldman et al. 2018, 2020; Lev-Ari et al. 2015].

Our program logic makes past states explicit and can be understood as a formal framework in which to execute hindsight reasoning. Indeed, the sequential-to-concurrent lifting of hindsight matches the thread-modular nature of our logic. Executing hindsight arguments in our framework brings several benefits. For our engine, hindsight arguments provide a *strategy* for finding history information. For hindsight theory, it not only gives the classical benefits of program logics like precision and mechanization resp. automation. One also inherits the other features of our logic. We found futures indispensable to prove the Harris set. As an interesting remark, our work solves a limitation that has been criticized in [Feldman et al. 2020], namely that the local-to-global lifting of the keyset theorem would not apply to optimistic algorithms with future-dependent linearization points. We simply invoke the theorem below past predicates.

A limitation of the existing hindsight theory as well as our work is that it does not apply to algorithms with impure future-dependent linearization points like the Herlihy-Wing queue [Herlihy and Wing 1990]. We leave the extension of the theory to such algorithms as future work. However, pure future-dependent linearization points are more common in concurrent data structures.

Atomic triples in TADA [da Rocha Pinto et al. 2014], CAP [Dinsdale-Young et al. 2010], and Iris [Jung et al. 2015] are specifications that justify a logical notion of atomicity for operations whose execution may take more than one physical step. This makes them suitable for compositional reasoning about nested modules with logically atomic specifications. The lifting of our program logic to prove linearizability is inspired by the reasoning principles underlying atomic triples.

When it comes to pure future-dependent linearization points, our retrospective linearization with rule LIN-PAST compares favorably to atomic triples. Proofs relying on atomic triples must typically implement intricate helping protocols that transfer ownership of the update tokens $\mathsf{Obl}(\Psi)$ and $\mathsf{Ful}(\Psi, v)$ between the thread that is linearized and the thread where the linearization point occurs. This is necessary because the update token trade must happen at the very moment when the linearization point occurs [Jung et al. 2020; Patel et al. 2021]. Our technique avoids such helping protocols altogether, which aids proof automation.

Futures. Our futures are nothing but Hoare triples in separation logic (with separating implication), and their use as assertions is well-known from program logics like Iris [Jung et al. 2018]. What we add is the observation that futures capture complex heap updates by iteratively combining futures of small updates found during the traversal preparing the complex update. This iterative combination is the key novelty of our development. It allows us to reason about updates of unbounded heap regions by means of updates of bounded regions.

Futures can be thought of as the opposite of atomic triples in that they prove the specification of a single physically atomic command like a `CAS` using a sequence of logical ghost steps.

A method for automatically handling updates affecting unbounded heap regions is proposed in [Ter-Gabrielyan et al. 2019], however, their method is tailored towards reachability. Being Hoare triples, our futures are not restricted to a specific class of properties.

Automation. There is a considerable body of work on the automated verification of concurrent data structures. For static linearization points, there are tools [Abdulla et al. 2013] and well-chosen abstract domains [Abdulla et al. 2018]. For dynamic linearization points, there are reductions to safety verification [Bouajjani et al. 2013, 2015, 2017]. Common to these works is that, in the end, they rely on a state-space search whereas our approach reasons in a program logic. Notably, the `poling` tool [Zhu et al. 2015] extends `cave` [Vafeiadis 2009, 2010a,b] to support dynamic linearization points, e.g., to verify intricate stacks and queues (which our tool `plankton` does not support because they are not search structures). Related is also [Itzhaky et al. 2014] in the sense that flows in particular can express heap paths. But we are not interested in verification condition generation and complete reductions to SMT, but rather proof generation, including invariant synthesis.

Other promising tools automating program logics include `Starling` [Windsor et al. 2017], `Caper` [Dinsdale-Young et al. 2017], `Voila` [Wolf et al. 2021], and `Diaframe` [Mulder et al. 2022]. However, these are closer to proof-outline checkers when compared to our tool. In particular, they do not perform loop invariant and interference inference or try to identify linearization points. Instead, they target more complex logics that are not designed for ease of automation.

ACKNOWLEDGMENTS

This work is funded in parts by the National Science Foundation under grant 1815633. The third author is supported by a Junior Fellowship from the Simons Foundation (855328, SW).

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *TACAS (LNCS, Vol. 7795)*. Springer, 324–338. https://doi.org/10.1007/978-3-642-36742-7_23

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2018. Fragment Abstraction for Concurrent Shape Analysis. In *ESOP (LNCS, Vol. 10801)*. Springer, 442–471. https://doi.org/10.1007/978-3-319-89884-1_16

Christian J. Bell, Andrew W. Appel, and David Walker. 2010. Concurrent Separation Logic for Pipelined Parallelization. In *SAS (LNCS, Vol. 6337)*. Springer, 151–166. https://doi.org/10.1007/978-3-642-15769-1_10

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *ESOP (LNCS, Vol. 7792)*. Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. On Reducing Linearizability to State Reachability. In *ICALP (2) (LNCS, Vol. 9135)*. Springer, 95–107. https://doi.org/10.1007/978-3-662-47666-6_8

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *CAV (2) (LNCS, Vol. 10427)*. Springer, 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM Press, 269–282. https://doi.org/10.1145/567752.567778

Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A Contention-Friendly Binary Search Tree. In *Euro-Par (LNCS, Vol. 8097)*. Springer, 229–240. https://doi.org/10.1007/978-3-642-40047-6_25

Tyler Crain, Vincent Gramoli, and Michel Raynal. 2016. A Fast Contention-Friendly Binary Search Tree. *Parallel Process. Lett.* 26, 3 (2016), 1650015:1–1650015:17. https://doi.org/10.1142/S0129626416500158

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (LNCS, Vol. 8586)*. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall. https://www.worldcat.org/oclc/01958445

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *POPL*. ACM, 287–300. https://doi.org/10.1145/2429069.2429104

Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP (LNCS, Vol. 10201)*. Springer, 420–447. https://doi.org/10.1007/978-3-662-54434-1_16

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (LNCS, Vol. 6183)*. Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (LNCS, Vol. 6015)*. Springer, 296–311. https://doi.org/10.1007/978-3-642-12002-2_25

Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC (LIPIcs, Vol. 121)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:21. https://doi.org/10.4230/LIPIcs.DISC.2018.23

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 128:1–128:29. https://doi.org/10.1145/3428196

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. ACM, 315–327. https://doi.org/10.1145/1480881.1480922

Keir Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation. University of Cambridge, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (LNCS, Vol. 6269)*. Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *ESOP (LNCS, Vol. 7792)*. Springer, 249–269. https://doi.org/10.1007/978-3-642-37036-6_15

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. ACM, 646–661. https://doi.org/10.1145/3192366.3192381

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC (LNCS, Vol. 2180)*. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. 2005. A Lazy Concurrent List-Based Set Algorithm. In *OPODIS (LNCS, Vol. 3974)*. Springer, 3–16. https://doi.org/10.1007/11795490_3

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *DISC (LNCS, Vol. 9363)*. Springer, 371–387. https://doi.org/10.1007/978-3-662-48653-5_25

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. 2003. Thread-Modular Abstraction Refinement. In *CAV (LNCS, Vol. 2725)*. Springer, 262–274. https://doi.org/10.1007/978-3-540-45069-6_27

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

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

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014. Modular reasoning about heap paths via effectively propositional formulas. In *POPL*. ACM, 385–396. https://doi.org/10.1145/2535838.2535854

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

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 *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2020. Refinement for Structured Concurrent Programs. In *CAV (1) (LNCS, Vol. 12224)*. Springer, 275–298. https://doi.org/10.1007/978-3-030-53288-8_14

Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. 2020a. Verifying concurrent search structure templates. In *PLDI*. ACM, 181–196. https://doi.org/10.1145/3385412.3386029

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020b. Local Reasoning for Global Graph Properties. In *ESOP (LNCS, Vol. 12075)*. Springer, 308–335. https://doi.org/10.1007/978-3-030-44914-8_12

Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. 2015. A Constructive Approach for Proving Data Structures' Linearizability. In *DISC (LNCS, Vol. 9363)*. Springer, 356–370. https://doi.org/10.1007/978-3-662-48653-5_24

Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE Computer Society, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*. ACM, 459–470. https://doi.org/10.1145/2491956.2462189

Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems - safety*. Springer.

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022a. *Artifact for "A Concurrent Program Logic with a Future and History"*. https://doi.org/10.5281/zenodo.7080459 or https://wolff09.github.io/plankton/.

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022b. A Concurrent Program Logic with a Future and History. *CoRR* abs/2207.02355 (2022). https://doi.org/10.48550/arXiv.2207.02355

Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.* 3, POPL (2019), 58:1–58:31. https://doi.org/10.1145/3290371

Roland Meyer and Sebastian Wolff. 2020. Pointer life cycle types for lock-free data structures with memory reclamation. *Proc. ACM Program. Lang.* 4, POPL (2020), 68:1–68:36. https://doi.org/10.1145/3371136

Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. ACM, 73–82. https://doi.org/10.1145/564870.564881

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI*. ACM, 809–824. https://doi.org/10.1145/3519939.3523432

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *PODC*. ACM, 85–94. https://doi.org/10.1145/1835698.1835722

Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340. https://doi.org/10.1007/BF00268134

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *POPL*. ACM, 297–302. https://doi.org/10.1145/1190216.1190261

Nisarg Patel, Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2021. Verifying concurrent multicopy search structures. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. https://doi.org/10.1145/3485490

Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26. https://doi.org/10.1145/1275497.1275501

Fred B. Schneider. 1997. *On Concurrent Programming*. Springer. https://doi.org/10.1007/978-1-4612-1830-2

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP (LNCS, Vol. 9032)*. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Dennis E. Shasha and Nathan Goodman. 1988. Concurrent Search Structure Algorithms. *ACM Trans. Database Syst.* 13, 1 (1988), 53–90. https://doi.org/10.1145/42201.42204

Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019. Modular verification of heap reachability properties in separation logic. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 121:1–121:28. https://doi.org/10.1145/3360547

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Ph. D. Dissertation. University of Cambridge, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *VMCAI (LNCS, Vol. 5403)*. Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27

Viktor Vafeiadis. 2010a. Automatically Proving Linearizability. In *CAV (LNCS, Vol. 6174)*. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Viktor Vafeiadis. 2010b. RGSep Action Inference. In *VMCAI (LNCS, Vol. 5944)*. Springer, 345–361. https://doi.org/10.1007/978-3-642-11319-2_25

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (LNCS, Vol. 4703)*. Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

Martin T. Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. In *PLDI*. ACM, 125–135. https://doi.org/10.1145/1375581.1375598

Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. 2017. Starling: Lightweight Concurrency Verification with Views. In *CAV (1) (LNCS, Vol. 10426)*. Springer, 544–569. https://doi.org/10.1007/978-3-319-63387-9_27

Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 2021. Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA. In *FM (LNCS, Vol. 13047)*. Springer, 407–426. https://doi.org/10.1007/978-3-030-90870-6_22

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV (LNCS, Vol. 5123)*. Springer, 385–398. https://doi.org/10.1007/978-3-540-70545-1_36

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (2) (LNCS, Vol. 9207)*. Springer, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1