Embedding Hindsight Reasoning in Separation Logic

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

Automatically proving linearizability of concurrent data structures remains a key challenge for verification. We present temporal interpolation as a new proof principle to guide automated proof search using hindsight arguments within concurrent separation logic. Temporal interpolation offers an easy-to-automate alternative to prophecy variables and has the advantage of structuring proofs into easy-to-discharge hypotheses. Additionally, we advance hindsight theory by integrating it into a program logic, bringing formal rigor and complementary proof machinery. We substantiate the usefulness of temporal interpolation by implementing it in a tool and using it to automatically verify the Logical Ordering tree. The proof is challenging due to future-dependent linearization points and complex structure overlays. It is the first formal proof of this data structure. Interestingly, our formalization revealed an unknown bug and an existing informal proof as erroneous.

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

Additional Key Words and Phrases: Hindsight, Linearizability, Logical Ordering Tree

ACM Reference Format:
Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. Embedding Hindsight Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 7, PLDI, Article 182 (June 2023), 24 pages. https://doi.org/10.1145/3591296

1 INTRODUCTION

We are concerned with automatically proving linearizability, the standard correctness criterion for concurrent data structures [Herlihy and Wing 1990]. A concurrent data structure is linearizable subject to a sequential specification of its methods, if each method takes effect in a single atomic step of its concurrent execution, the method's *linearization point*, and satisfies the sequential specification in this step.

Concurrent separation logics [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; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and Parkinson 2007] provide a powerful toolbox of deductive reasoning techniques to verify complex concurrent data structures. However, the proof construction heavily relies on the proof author's creativity and expertise in wielding the available tools effectively. For instance, in order to construct the inductive invariant of the data structure, the proof author may have to devise proof-specific resource algebras to express ghost state that captures the key aspects of the computation history. This hinders proof automation due to the vast complexity of the proof space that needs to be explored. Similarly, the proofs may make use of prophecy variables [Abadi and Lamport 1991] to predict future-dependent linearization points [Jung et al. 2020; Liang and Feng 2013; Vafeiadis 2008]. Constructing such proofs involves backward reasoning, which is difficult to automate [Bouajjani et al. 2017]. It stands to reason that there is a need for guiding principles that help to structure the proof and that provide effective strategies for automated tools to prune the search space.

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.

Hindsight theory [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010] provides such a guiding principle, which we refer to as *temporal interpolation.* One proves lemmas of the form: if there existed a past state that satisfied property p and the current state satisfies q, then there must have existed an intermediate state that satisfied o. Such lemmas can then be applied, e.g., to prove the existence of a future-dependent linearization point in hindsight. Hindsight is 20/20, the arguments only involve forward reasoning, which is easier to automate than, say, prophecy-based arguments.

One limitation of the existing hindsight theory is that it has only explored the general idea of temporal interpolation very narrowly. Concretely, it has been used only to prove hindsight lemmas about concurrent traversals of data structures. These are variations of statements of the form *"if the current node x of the traversal was reachable from the root at some point in the past (p), and y is the successor of x in the present state (q), then y was reachable from the root at some point in the past (o)".* We show that temporal interpolation applies more broadly in other contexts as well.

Another limitation is that the proof and application of these hindsight lemmas has so far been confined to meta-level linearizability arguments. As a consequence, existing hindsight proofs can lack the rigor enforced by a program logic. We show that this has resulted in at least one incorrect hindsight-based proof in the past [Feldman et al. 2020].

Contributions. Building on [Meyer et al. 2022a], we present a concurrent separation logic that integrates temporal interpolation as a general proof rule. The logic offers the best of both worlds: it enables the intuitive reasoning of hindsight theory within the rigorous framework of a formal proof system. As in [Meyer et al. 2022a], the logic's semantic foundation is based on computations rather than states, which it exposes at the syntactic level in the form of a lightweight temporal operator. This operator provides a uniform mechanism for tracking history information. This reduces the need for introducing proof-specific auxiliary ghost state and helps to prune the space of possible proofs to consider for automatic proof construction. At the same time, the logic offers all advantages of separation logic, including the ability to reason locally about state mutation and concurrency via the frame rule, and to introduce ghost state if and when needed.

The key innovation over [Meyer et al. 2022a] is a new proof rule that enables general hindsight reasoning via temporal interpolation. The proof rule postulates and then applies hypotheses $h(p, q, o)$ that state the correctness of the temporal interpolation. These hypotheses are collected by the main proof and then discharged in subproofs. This approach provides a proof-structuring mechanism: the subproofs can use a coarse-grained abstraction of the program behavior, which often simplifies the overall proof argument and aids automation. The nature of temporal interpolation as a proof-structuring mechanism is made formally precise in our soundness proof by showing that the proof rule can be eliminated from the logic.

To demonstrate the usefulness of our development, we have integrated temporal interpolation into `plankton` [Meyer et al. 2022a], an automated verifier for concurrent search structures based on separation logic. As a case study, we have used the extended tool [Meyer et al. 2023] to automatically verify the logical-ordering (LO-)tree [Drachsler et al. 2014]. The proof exercises the full power of our logic by combining a linearizability argument based on temporal interpolation with local reasoning in separation logic. To our knowledge, there has been no formal proof of the LO-tree prior to this work (either automated or mechanized). In fact, our efforts identified one previously unreported bug in the original implementation of the data structure. Another bug was identified by Feldman et al. [2020], who presented an informal hindsight-based proof. While the fix proposed by Feldman et al. [2020] addresses the original bug, we show that it introduces a new linearizability violation. This underscores the benefit of supporting hindsight proofs in a formal logic.

Limitations. Our focus is on automating linearizability proofs for concurrency library implementations. In particular, our program logic was not designed for modular verification of library clients

```
1  struct C { var l: Int; var r: Int }
2  predicate counter(c: C, n: Int) {
3    ∃ $n_l$ $n_r$ ::
4      c.l ↦ $n_l$ ∗ c.r ↦ $n_r$ ∧
5      n == $n_l$ + $n_r$
6  }
```

```
7  { counter(c, n) }
8  method read(c: C) {
9    val x = c.l
10   val y = c.r
11   return x + y
12 } { v. counter(c, n) ∗ v = n }
```

```
13 { counter(c, n) }
14 method inc(c: C) {
15   if (nondet())
16     FAA(c.l, 1)
17   else FAA(c.r, 1)
18 } { counter(c, n + 1) }
```

Fig. 1. Distributed counter object.

against the proved linearizability specifications. Moreover, `plankton` is not yet fully automated: the user provides an invariant describing the properties of each node comprising the data structure in the shared heap. Finally, the implementation of temporal interpolation in `plankton` is currently geared towards reasoning about *pure* future-dependent linearization points (i.e., those that do not modify the abstract state of the data structure). We leave the handling of impure cases in the implementation as future work. Though, we note that these cases are not prevalent in the context of concurrent search structures.

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

2 OVERVIEW

We illustrate our approach using the idealized distributed counter shown in Figure 1. A counter object c has an abstract state that tracks an integer value n and supports two methods: `inc(c)` atomically increments n by 1 and `read(c)` returns n. The counter is distributed in the sense that n is the sum of two integer values stored in separate memory locations `c.l` and `c.r`. The implementation of `inc` non-deterministically chooses one of the two locations and then atomically increments it using a *fetch-and-add* (`FAA`) instruction. The implementation of `read` non-atomically reads the values of the two memory locations and then returns their sum.

Our goal is to prove that the distributed counter is linearizable with respect to its sequential specification, which is given in Figure 1 as Hoare annotations expressed in separation logic. The specification uses the predicate `counter`(c, n) to define the abstract state of the counter c in terms of the underlying memory representation. Here, a *points-to* predicate $a \mapsto v$ expresses ownership of the memory location at address a and, moreover, that this location stores value v. The operator $p * q$ is *separating conjunction*, which expresses that p and q hold over disjoint memory regions. In the following, we assume an *intuitionistic semantics* of these predicates, i.e. $p * \mathit{true} = p$.

To prove linearizability, we need to show that each method transforms its precondition to its postcondition in a single atomic step. Due to interferences by concurrent `inc` methods, the counter value may change throughout the execution of a method. Hence, the value n in the precondition of the specification does not refer to the counter's initial abstract state when the method is invoked, but rather to its abstract state at the linearization point. This semantics of the Hoare annotations corresponds to that of logically atomic triples [da Rocha Pinto et al. 2014]. Note that the variable v in the postcondition of `read` is bound to the method's return value.

The linearization point of `inc` is when `FAA` is executed and the desired Hoare specification follows immediately from the specification of `FAA`. So we focus on the more interesting case of `read`. The `read` method does not change the value of the counter. Hence, it suffices to show that the returned value $x + y$ is equal to the counter value n at the linearization point. The challenge is that the linearization point depends on the future interferences of concurrent `inc` operations. In fact, it may lie in a concurrently executing `inc`. For example, consider the scenario where at the point when `read` executes Line 9, we have $c.\mathtt{l} = c.\mathtt{r} = 0$ and before it proceeds to Line 10, two concurrent `inc`s increment first $c.\mathtt{l}$ and then $c.\mathtt{r}$ to 1. That is, when `read` executes Line 9 we have $n = 0$ and when it executes Line 10 we have $n = 2$, yet the return value is 1. Nevertheless, this execution of read is linearizable because there is a time point in between when $n = 1$, namely right after the linearization point of the first concurrently executing inc. Note that if the second inc incremented c.l instead of c.r, then the return value of read would be 0 and its linearization point would already be when it reads c.l. This is why the linearization point of read is future-dependent.

Intuitively, the linearizability of read follows from the fact that the two memory locations increase monotonically by increments of 1. So if the counter has value n at some point t and value $n' > n$ at some later point t', then for each value n'' with $n \leq n'' \leq n'$ there is an intermediate state between t and t' where the value of the counter is n''. We demonstrate how to formalize this intuitive argument in our program logic. The logic enables temporal reasoning about computations using *past predicates*, $\Diamond p$, which express that the *state predicate* p held true at some prior state of the computation. Our goal is to derive that $\Diamond(\texttt{counter}(c, x + y))$ is true after Line 10. This implies the existence of a linearization point for read.

The proof proceeds in two parts. The first part proves the goal above but assumes the validity of an auxiliary hypothesis that is derived during the proof. This hypothesis captures the intuitive reasoning used above to conclude the existence of an unobserved intermediate state due to interferences by other threads. The second part of the proof discharges this hypothesis.

An outline of the first part of the proof is shown in Figure 2. Throughout the proof, variables that do not occur in the program code such as n_l are implicitly existentially quantified. The program logic follows a thread-modular approach that mostly uses sequential Hoare-style reasoning. The soundness of this reasoning is guaranteed by ensuring that each two consecutive atomic commands are separated by an *interference-free* intermediate assertion. That is, concurrently executing threads will not affect the truth value of this assertion. In the following, we elude the details of the mechanism used to check interference freedom as it is orthogonal to our core contributions. The details of this mechanism are presented in §3.

```
19  { $\texttt{counter}(c, n)$ }
20  method read(c: C) {
21      { $c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r$ }
22      val x = c.l
23      { $c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r \wedge x = n_l$
          $\wedge \Diamond(c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r)$ }
24      { $c.\texttt{l} \mapsto n'_l * c.\texttt{r} \mapsto n'_r \wedge x = n_l$
          $\wedge \Diamond(c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r \wedge n_r \leq n'_r)$ }
25      val y = c.r
26      { $c.\texttt{l} \mapsto n'_l * c.\texttt{r} \mapsto n'_r \wedge x = n_l \wedge y = n'_r$
          $\wedge \Diamond(c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r \wedge n_r \leq n'_r)$ }
27      // Hypothesis: $\forall n_l\, n_r\, n'_r.\, \{ a \}\ \mathbb{I}^*\ \{ b \}$
28      // $a \triangleq c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r \wedge n_r \leq n'_r$
29      // $b \triangleq c.\texttt{r} \mapsto n'_r \rightarrow \underline{\Diamond}(\texttt{counter}(c, n_l + n'_r))$
30      { $\texttt{counter}(c, n') \wedge x = n_l \wedge y = n'_r$
          $\wedge \Diamond(\texttt{counter}(c, n_l + n'_r))$ }
31      { $\texttt{counter}(c, n') \wedge \Diamond(\texttt{counter}(c, x + y))$ }
32      return x + y
33  } { $v.\, \texttt{counter}(c, n) * v = n$ }
```

Fig. 2. Proof outline for the read method.

The proof starts by unfolding the definition of $\texttt{counter}(c, n)$ in the precondition, yielding the assertion on Line 21. After reading c.l we know that x is bound to the old value n_l of c.l. We also record the state of the counter before the read command in a past predicate $\Diamond(c.\texttt{l} \mapsto n_l * c.\texttt{r} \mapsto n_r)$, yielding the assertion on Line 23. This assertion is not interference-free because concurrent inc threads may change the values of c.r and c.l. We therefore weaken the assertion by introducing fresh variables n'_l and n'_r for these values. We leave n'_l unconstrained but preserve $n_r \leq n'_r$, capturing that concurrent threads can only increase c.r. Since $n_r \leq n'_r$ only concerns logical variables, we can push this fact into the past predicate. The resulting interference-free assertion is shown on Line 24.

We proceed similarly for the read of c.r resulting in the assertion on Line 26. Again, this assertion is not interference-free because concurrent threads may change the value of c.r. We want to weaken this assertion to the interference-free assertion on Line 31, which implies our desired goal. Observe that Line 31 follows from Line 30 using equality reasoning. So it remains to connect lines 26 and 30. First, observe that the predicate $\texttt{counter}(c, n')$ is obtained from $c.\texttt{l} \mapsto n'_l * c.\texttt{r} \mapsto n'_r$ by choosing $n' = n'_l + n'_r$. To derive, $\Diamond(\texttt{counter}(c, n_l + n'_r))$, the proof conjectures the validity of the hypothesis on Line 27. This hypothesis is a Hoare triple of the shape

$\{\,p\,\}\ \mathbb{I}^*\ \{\,q \longrightarrow \underline{\diamondsuit}o\,\}$. Here, $\longrightarrow$ is logical implication and $\underline{\diamondsuit}o$ is syntactic sugar for $o \vee \diamondsuit o$. The variable $\mathbb{I}$ stands for a set of *interferences* that the overall proof infers as an auxiliary output of its derivation. The set $\mathbb{I}$ consists of pairs $(g, \mathtt{com})$ where com is any atomic command in the program that affects the thread-local or shared program state, and g is the intermediate assertion preceding com in the proof. In our example, the derived interferences all come from the inc method. They comprise the set

$$\mathbb{I} = \{(c.\mathsf{l} \mapsto v, \mathsf{FAA}(c.\mathsf{l}, 1)), (c.\mathsf{r} \mapsto v, \mathsf{FAA}(c.\mathsf{r}, 1))\}\ .$$

Each interference can be viewed as a guarded command that first assumes g and then executes com. From these guarded commands, we build the new program $\mathbb{I}^*$ which nondeterministically executes the interferences in $\mathbb{I}$ an arbitrary number of times. That is, $\mathbb{I}^*$ can be viewed as abstracting the overall program. Thus, the hypothesis $\{\,p\,\}\ \mathbb{I}^*\ \{\,q \longrightarrow \underline{\diamondsuit}o\,\}$ states that if execution starts from a state that satisfies p and after any number of program steps it reaches a state that satisfies q, then o must have been true in some intermediate state. The temporal interpolation rule allows us to derive from such a hypothesis that if the program is in a state s that satisfies $\diamondsuit p \wedge q$, then also $\underline{\diamondsuit}o$ holds in s. We use temporal interpolation to derive Line 30 from Line 26 using the hypothesis on Line 27.

The second part of the proof is then to establish the validity of the hypothesis. This part can also be carried out in the logic, using the same thread-modular and local reasoning principles. Effectively, the proof boils down to finding an invariant *inv* that is implied by p, implies $q \longrightarrow \underline{\diamondsuit}o$, and is preserved by each of the interferences. In our example, the following invariant does the trick:

$$inv \ \triangleq\ \exists n_l''\, n_r''.\ c.\mathsf{l} \mapsto n_l'' * c.\mathsf{r} \mapsto n_r'' \wedge n_l \le n_l'' \wedge n_l'' + n_r'' < n_l + n_r' \vee \underline{\diamondsuit}(\mathsf{counter}(c, n_l + n_r'))$$

Intuitively, the first disjunct of the invariant holds up to the linearization point and afterwards, the second disjunct holds. Note that *inv* contains a past operator and is therefore a *computation predicate*, not a state predicate.

We contrast the above proof with one based on prophecy reasoning in the style of [Jung et al. 2020]. Without temporal interpolation, the proof has to witness the linearization point of a read thread t at the exact moment where the relevant inc thread sets n to $n_l + n_r'$ for the value n_r' that will be later read by t. However, n_r' depends on how many other inc threads will still increment r between these two points. One can introduce a prophecy variable for t that predicts the number of such increments between the points when t reads l and r. To establish the linearizability argument, the prophecy variables and linearization obligations for the unboundedely many read threads need to be shared with all inc threads that may execute concurrently. This involves a complex *helping protocol* construction that governs the transfer of resources between threads. This construction is reflected in the proof in the form of a more complex invariant capturing the shared state of the data structure.

3 PRELIMINARIES

We study concurrency libraries, i.e., a single program executed by a potentially unbounded number of threads. We give a formal account of concurrency libraries and introduce a Hoare-style proof system for verifying them. Our formalism is based on [Meyer et al. 2022a].

3.1 Programming Model

Along the lines of abstract separation logic [Calcagno et al. 2007; Dinsdale-Young et al. 2013; Jung et al. 2018], the actual sets of states and commands are a parameter to our development.

States and Computations. We draw *states* from a separation algebra, a partial commutative monoid $(\Sigma, *, \mathsf{emp})$ with a set of units emp so that (i) each state $\mathsf{s} \in \Sigma$ has a unit $1 \in \mathsf{emp}$ with $\mathsf{s} * 1 = \mathsf{s}$, and (ii) $1 * 1'$ is undefined for any two distinct units $1, 1' \in \mathsf{emp}$. Definedness of $\mathsf{s} * \mathsf{s}'$ is denoted $\mathsf{s} \# \mathsf{s}'$.

We work over a separation algebra with a certain structure. We expect states from $(\Sigma, *, \mathsf{emp})$ to be composed from a *global* and a *local* state. The global resp. local states are again drawn from separation algebras $(\Sigma_\mathsf{G}, *_\mathsf{G}, \mathsf{emp}_\mathsf{G})$ resp. $(\Sigma_\mathsf{L}, *_\mathsf{L}, \mathsf{emp}_\mathsf{L})$. We require that (i) states in Σ are multiplied elementwise, $(\mathsf{g}_1, \mathsf{l}_1) * (\mathsf{g}_2, \mathsf{l}_2) \triangleq (\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2)$ provided the resulting state is in Σ and undefined otherwise, (ii) states Σ can be decomposed, $(\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2) \in \Sigma$ implies $(\mathsf{g}_1, \mathsf{l}_1) \in \Sigma$, and (iii) units emp are also composed, $\mathsf{emp} \triangleq \mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L}$. It is readily checked that this is a separation algebra.

The temporal interpolation principle we propose reasons over knowledge obtained at different points in time during a computation. To formulate it, we lift the given separation algebra $(\Sigma, *, \mathsf{emp})$ to a separation algebra over computations $(\Sigma^+, *, \Sigma^*.\mathsf{emp})$. A computation is a non-empty sequence of states. We write $\sigma.\tau$ for the concatenation of two computations σ and τ. The multiplication of two computations $\sigma.\mathsf{s}, \tau.\mathsf{t} \in \Sigma^+$ is defined, $\sigma.\mathsf{s} \# \tau.\mathsf{t}$, if $\sigma = \tau$ and $\mathsf{s} \# \mathsf{t}$. In this case, the multiplication yields $\sigma.\mathsf{s} * \tau.\mathsf{t} \triangleq \sigma.(\mathsf{s} * \mathsf{t})$. The two computations share the same history, which is preserved by the multiplication. In the current state, we use the composition given by the separation algebra. This construction works in general, not just for our product separation algebra.

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

Predicates. For clarity of the exposition, we refrain from introducing an assertion language that needs to be interpreted but work on the semantic level. Given a separation algebra $(\Gamma, *, \mathsf{emp})$, a *predicate* is a set of elements from Γ. The predicates form a Boolean algebra $(\mathbb{P}(\Gamma), \cup, \cap, \subseteq, \overline{}, \emptyset, \Gamma)$ with disjunction, conjunction, implication, negation, false, and true. We moreover have the standard connectives *separating conjunction* $*$ and *separating implication* $-\!\!*$:

$$p * q \;\triangleq\; \{\, \gamma_1 * \gamma_2 \mid \gamma_1 \in p \wedge \gamma_2 \in q \wedge \gamma_1 \# \gamma_2 \,\} \quad \text{and} \quad p -\!\!* q \;\triangleq\; \{\, \gamma \mid \{\gamma\} * p \subseteq q \,\}\,.$$

A predicate p is *intuitionistic*, if $p * \Gamma \subseteq p$.

In our setting, we have the separation algebra of states $(\Sigma, *, \mathsf{emp})$ and *state predicates* $p, q, o \subseteq \Sigma$. We moreover have the separation algebra of computations $(\Sigma^+, *, \mathsf{emp}^+)$ and *computation predicates* $a, b, c \subseteq \Sigma^+$. For our temporal interpolation principle developed in §4, it suffices to consider simple computation predicates that reason about single states of the computation. These computation predicates are derived from state predicates.

Definition 3.2. From state predicates $p \subseteq \Sigma$ we construct (i) the *now predicate* $\_p \;\triangleq\; \Sigma^*.p$ and (ii) the *past predicate* $\diamonddot p \;\triangleq\; \Sigma^*.p.\Sigma^+$ and (iii) the *weak past predicate* $\underline{\diamonddot} p \;\triangleq\; \_p \cup \diamonddot p$.

Now predicates lift state predicates to hold in the last (the current) state of a computation. Past predicates lift state predicates to hold at some time in the past of the computation. The precise moment when the state predicate was true is not known, which means framing is not relevant for past predicates, and lead us to define the multiplication of computations as an intersection in the past. Intuitionism carries over from state to computation predicates.

Lemma 3.3. *If p is intuitionistic, so is $\_p$. Predicate $\diamonddot p$ is intuitionistic.*

The predicates are compatible with the separation logic operators as follows.

Lemma 3.4. $\_\,(p \oplus q) \;=\; \_p \oplus \_q$ *for all* $\oplus \;\in\; \{\cap, \cup, *, -\!\!*\}$, $\_\overline{p} = \overline{\_p}$, *false* $= \_$*false*, *true* $= \_$*true*, $\_p \subseteq \_q$ *iff* $p \subseteq q$, $\diamonddot(p \cap q) \subseteq \diamonddot p \cap \diamonddot q$, $\diamonddot(p \cup q) = \diamonddot p \cup \diamonddot q$, *and* $\diamonddot p \subseteq \diamonddot q$ *iff* $p \subseteq q$.

Commands. We assume a potentially infinite set of commands $(\mathtt{COM}, [\![-]\!])$. The actual set is a parameter and not relevant for our development. Commands $\mathtt{com} \in \mathtt{COM}$ transform a pre state into a post state which, due to non-determinism, need not be unique. This state transformer is given by the interpretation $[\![\mathtt{com}]\!] : \Sigma \to \mathbb{P}(\Sigma)$ of $\mathtt{com}$. We lift the transformer to computations by appending the post state: $[\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}) \triangleq \{\, \sigma.\mathsf{s}.\mathsf{s}' \mid \mathsf{s}' \in [\![\mathtt{com}]\!](\mathsf{s}) \,\}$. The transformer extends to predicates in the usual way [Dijkstra 1976]: $[\![\mathtt{com}]\!](a) \triangleq \bigcup_{\sigma\in a} [\![\mathtt{com}]\!](\sigma)$. We expect to have a neutral command $\mathtt{skip} \in \mathtt{COM}$ that is interpreted as the identity. To model failing commands, we follow [Calcagno et al. 2007] and assume their post state to be abort, a dedicated top value in the lattice of predicates.

For the frame rule to be sound, we require the following locality:

$$\forall a, b, c \subseteq \Sigma^+.\quad [\![\mathtt{com}]\!](a) \subseteq b \quad\text{implies}\quad [\![\mathtt{com}]\!](a * c) \subseteq b * c\,. \tag{LocCom}$$

Note that (LocCom) requires the computation predicate c to perform a stuttering step when being framed on the right-hand side of the latter inclusion. We call a computation predicate $c \subseteq \Sigma^+$ *frameable*, if $\sigma.s \in c$ implies $\sigma.s.s \in c$ for all $\sigma.s \in \Sigma^+$. Fortunately, all computation predicates that are constructed by union, intersection, and separating conjunction from now and past predicates are frameable. Unless otherwise stated, we will assume that all predicates we encounter are frameable.

Concurrency libraries. Concurrency libraries consist of an unbounded number of threads that all execute the same program st. Different functions would be modeled by an initial non-deterministic choice among the function bodies, which is supported in our while language together with sequential composition and repetition:

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

A configuration $\mathsf{cf} = (\gamma, \mathsf{pc})$ of the library comprises a global computation $\gamma \in \Sigma_G^+$ and a program counter pc. The program counter maps thread identifiers $i \in \mathbb{N}$ to pairs $\mathsf{pc}(i) = (\lambda, \mathtt{st})$ containing thread-i-local information: a computation $\lambda \in \Sigma_L^+$ and a program fragment st the execution of which remains. The transition rules among configurations are as expected: a step of thread i changes the shared and the thread-i-local information according to the transformer of the executed command, and leaves all other threads unchanged.

Towards a Hoare-style proof system, we call a configuration (γ, pc) initial wrt. computation predicate a and program st, if all threads i with $\mathsf{pc}(i) = (\lambda, \mathtt{st}')$ satisfy $(\gamma, \lambda) \in a$ and $\mathtt{st}' = \mathtt{st}$. Similarly, (γ, pc) is accepting wrt. b, if all terminated threads with $\mathsf{pc}(i) = (\lambda, \mathtt{skip})$ satisfy $(\gamma, \lambda) \in b$. Reachability is defined as usual. We refer to the initial, accepting, and reachable configurations by $\mathsf{Init}_{a,\mathtt{st}}$, Acc_b, and $\mathsf{Reach}(\mathsf{cf})$, respectively.

The correctness condition we would like to prove for concurrency libraries is whether all configurations reachable from a-st-initial configurations are b-accepting, $\mathsf{Reach}(\mathsf{Init}_{a,\mathtt{st}}) \subseteq \mathsf{Acc}_b$. In this case, we say that a Hoare triple of the form $\{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$ is *valid*, denoted by $\models \{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$.

3.2 Program Logic

We use a proof system to establish the validity of Hoare triples, Figure 3 below (ignore the marked parts for now). The proof system is thread-modular [Berdine et al. 2008; Jones 1983] in nature, thus verifies a single thread in isolation. To account for the actions of other threads which may affect the isolated thread, we ensure *interference freedom* [Owicki and Gries 1976] of the overall proof.

Technically, the proof system establishes judgements $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$ with the following components: (i) a Hoare triple $\{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$ for the isolated thread, (ii) a set $\mathbb{P}$ of intermediary assertions used during the proof of the Hoare triple, and (iii) a set $\mathbb{I}$ of interferences that the isolated thread is subject to. Recording the intermediary assertions allows us to separate the interference freedom check from the derivation of the Hoare triple [Dinsdale-Young et al. 2013, Section 7.3]. We denote the interference freedom of $\mathbb{P}$ under $\mathbb{I}$ by $\boxtimes_{\mathbb{I}}\, \mathbb{P}$. The resulting proof system is sound.

Theorem 3.5 (Meyer et al. [2022a]). *$\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\, \mathtt{st}\, \{\, b \,\}$ and $\boxtimes_{\mathbb{I}}\, \mathbb{P}$ and $a \in \mathbb{P}$ imply $\models \{\, a \,\}\, \mathtt{st}\, \{\, b \,\}$.*

In our development, we will use the set of computations $[\![\mathtt{st}]\!]_{\mathbb{I}}(a)$ defined by extending each computation in a by every sequence of states encountered when executing program st to completion while admitting interferences from $\mathbb{I}$. The formal definition is the straightforward lift of $[\![\![-]\!]\!]$ to sequences of commands and interferences. A consequence of the soundness result is the following.

Lemma 3.6. *If there is a set $\mathbb{P}$ with $a \in \mathbb{P}$, $\boxast_{\mathbb{I}}\, \mathbb{P}$, and $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$, then $[\![\![\mathtt{st}]\!]\!]_{\mathbb{I}}(a) \subseteq b$.*

Interference Freedom. The isolated thread is influenced by the actions of other, interfering threads. We capture those actions as *interferences* $(\, c, \mathtt{com})$, meaning that com may be executed by an interfering thread from a configuration satisfying c. Observe that the global portion of c imposes restrictions on when the interference may happen while the local portion of c supplies the local computation the interfering thread needs for its execution. From the point of view of the isolated thread with computation (γ, λ), only the global portion γ changes, formally:

$$[\![\![\, (c, \mathtt{com}) \,]\!]\!](\gamma, \lambda) \;\triangleq\; \{\, (\gamma', \lambda) \;\mid\; \exists \lambda_1, \lambda_2.\; (\gamma, \lambda_1) \in c \;\wedge\; (\gamma', \lambda_2) \in [\![\![\mathtt{com}]\!]\!](\gamma, \lambda_1) \,\}\,.$$

The *interference freedom check* wrt. a set $\mathbb{I}$ of interferences then proceeds as follows. It takes a computation predicate a and tests whether $[\![\![\, (c, \mathtt{com}) \,]\!]\!](a) \subseteq a$ for all $(c, \mathtt{com}) \in \mathbb{I}$. If this is the case, the interference does not invalidate a and the predicate is interference-free. The interference freedom check extends naturally to the set of predicates $\mathbb{P}$. We write $\mathbb{I} * b$ for the set of interferences $(a * b, \mathtt{com})$ with $(a, \mathtt{com}) \in \mathbb{I}$. We also use the notation for sets of predicates $\mathbb{P}$ and write $\mathbb{P} * b$ for the set of predicates $a * b$ with $a \in \mathbb{P}$. We also remark that past information is always interference-free, because interferences append states and this does not change the past of the computation.

4 TEMPORAL INTERPOLATION

Temporal interpolation is a reasoning principle to derive information about intermediary states that have not been observed in the program proof. Coming back to the example of a distributed counter, if the counter value has been n_1 in the past and is now $n_2 > n_1$, then we wish to derive that there has been a moment in which the counter has been n with $n_2 \geq n \geq n_1$. Temporal interpolation will allow us to do so, although an assertion with counter value n is not interference-free and hence will not be observable in the program proof. We can actually guarantee that the moment in which the counter was n is in between the past and the current state, but defer the timing aspect for now. Another example of temporal interpolation is reachability in concurrent data structures, as studied by the hindsight principle which inspired this work [Feldman et al. 2018, 2020; O'Hearn et al. 2010]. If a node n_1 has been reachable in the past, and the node now points to n_2, then there has been a moment in which the node was reachable and pointed to n_2. Also this moment will not be interference-free and hence cannot be recorded in the program proof (the set of predicates $\mathbb{P}$).

To derive the intermediary information, temporal interpolation proves inclusions of the form

$$\underline{\odot}\, p \,\cap\, \_q \quad\subseteq\quad \underline{\odot}\, o\,. \tag{1}$$

The inclusion indeed formulates an interpolation property for the set of computations: if state predicate p has been true in the past of the computation and we now have q, then there has been a moment in which o was true, and typically o will be $p \cap q$. Unfortunately, the inclusion will rarely hold in this generality. The first problem is that the set of computations leading from p to q is too liberal. Rather than considering all sequences of states, we should only consider the ones generated by the program at hand. The second problem is that even if we restrict the computations, we need to prove the inclusion. Our technical contribution is to embed the above inclusion into a proof system in which it can justifiably be used.

To restrict the set of computations leading from p to q, we introduce a new predicate that reflects the influence of the program on the course of the computation. The observation behind the definition of the predicate is that the set of interferences $\mathbb{I}$ which we collect during the proof gives us precise information about the program behavior. An interference $(a, \mathtt{com}) \in \mathbb{I}$ not only says that a command com is executable, it also records in predicate a the conditions under which the command will be executed. Notably, these conditions refer to the shared as well as the local state, meaning the interference captures the thread-local behavior as well. The new predicate thus employs the set of interferences as an abstraction of the overall program behavior.

To make the idea formal, we transform interferences into programs as follows:

$$\texttt{2com}(a, \texttt{com}) \triangleq \texttt{atomic}\{\, \texttt{assume}(a * \mathit{true});\ \texttt{com}\,\} \quad \text{and} \quad \texttt{2stmt}(\mathbb{I}) \triangleq \big(\textstyle\sum_{(a,\texttt{com})\in\mathbb{I}} \texttt{2com}(a, \texttt{com})\big)^{*} .$$

We turn an interference $(a, \texttt{com})$ into an atomic block the execution of which is guarded by an assumption. Recall that atomic blocks are not part of our programming constructs, but the above expression will be treated as a single command with the expected semantics. The reason we need a single command is that $\texttt{2com}(a, \texttt{com})$ should abstract command com in the program, and that command leads to a single state change. Also note that a is a predicate from the assertion language that we deliberately use within an assumption. To be closer to programming practice, one can weaken a to information about the current state that can be checked over the program variables. We use $a * \mathit{true}$ rather than a to make sure the command satisfies (LocCom). We also call $\texttt{2com}(a, \texttt{com})$ a *self-interference*. Function $\texttt{2stmt}(-)$ lifts the construction to a set of interferences. The resulting program repeatedly executes all self-interferences in random order.

The new predicate $\mathrm{Gov}(\mathbb{I})$ describes the set of $\mathbb{I}$-*governed* computations, the computations in which every state change is due to an interference or a self-interference:

$$\mathrm{Gov}(\mathbb{I}) \quad \triangleq \quad [\![\![\texttt{2stmt}(\mathbb{I})]\!]\!]_{\mathbb{I}}(\Sigma) .$$

We view here Σ as a set of computations that consist of a single state. With this definition, we intend to replace Inclusion (1) by

$$\underline{\diamonddot} p \cap \_q \cap \mathrm{Gov}(\mathbb{I}) \quad \subseteq \quad \underline{\diamonddot} o . \tag{2}$$

This inclusion may or may not hold depending on the set of interferences. To prove the inclusion for the set of interferences at hand, we define Hoare triples that take a set of interferences as a parameter. We justify the need for this parameterization in moment. A so-called *hypothesis* h has the form

$$\mathbb{X} \Vdash \{\, a \,\}\ \texttt{2stmt}(\mathbb{X})\ \{\, b \,\} .$$

Variable $\mathbb{X}$ will be evaluated by a set of interferences. The hypothesis is said to *hold for* $\mathbb{I}$, denoted by $\mathbb{I} \checkmark h$, if we can prove the Hoare triple with $\mathbb{X}$ replaced by $\mathbb{I}$: there is a set of predicates $\mathbb{P}$ with $a \subseteq a' \in \mathbb{P}$ so that $\mathbb{P}, \mathbb{I} \Vdash \{\, a' \,\}\ \texttt{2stmt}(\mathbb{I})\ \{\, b \,\}$ is derivable and $\boxplus_{\mathbb{I}} \mathbb{P}$. We elaborate on the weakening of a to a' further below. For a set of hypothesis $\mathbb{H}$, we write $\mathbb{I} \checkmark \mathbb{H}$ to mean $\mathbb{I} \checkmark h$ for all $h \in \mathbb{H}$.

The hypotheses we are interested in have the shape

$$\mathbb{X} \Vdash \{\, \_p \,\}\ \texttt{2stmt}(\mathbb{X})\ \{\, \_q \rightarrow \underline{\diamonddot} o \,\} .$$

Since the shape is fixed, we write the hypothesis as $h(p, q, o)$. It states that from a computation ending in p, every execution of the interferences and the self-interferences that leads to a state from $\_q$ satisfies $\underline{\diamonddot} o$. This is precisely the information that has been missing to justify Inclusion (2).

Lemma 4.1. *If* $\mathbb{I} \checkmark h(p, q, o)$*, then* $\underline{\diamonddot} p \cap \_q \cap \mathrm{Gov}(\mathbb{I}) \subseteq \underline{\diamonddot} o$.

We incorporate temporal interpolation into the separation logic presented in §3 by means of the new proof rule temporal-interpolation given in Figure 3. It draws a conclusion as in Equation (1) at the expense of recording a hypothesis $h(p, q, o)$. There are a few things worth noting. The rule does not expect the predicate $\mathrm{Gov}(\mathbb{I})$ to be present in the premise. The soundness result will show that any program proof can be strengthend to maintain the set of governed computations, and we can therefore leave this set implicit. We draw the conclusion after a skip command, which turns the weak past predicate $\underline{\diamonddot} o$ from the hypothesis into a proper past predicate $\diamonddot o$. This is

$$\textsc{com-ti}\quad \frac{[\![\![\mathtt{com}]\!]\!](a) \subseteq b}{\{b\}, \{\, (a, \mathtt{com})\,\}, \varnothing \Vdash_{ti} \{\, a\,\}\ \mathtt{com}\ \{\, b\,\}}
\qquad
\textsc{temporal-interpolation}\quad \frac{p, q \text{ intuitionistic} \qquad \mathbb{I} = \{\, (a, \mathtt{skip})\,\} \qquad \mathbb{H} = \{\, h(p, q, o)\,\}}{\{\, a \cap \Diamonddot o\,\}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a \cap \underline{\Diamonddot} p \cap \_q\,\}\ \mathtt{skip}\ \{\, a \cap \Diamonddot o\,\}}$$

$$\textsc{temporal-interpolation-unordered}\quad \frac{p, q \text{ intuitionistic} \qquad \mathbb{I} = \{\, (a, \mathtt{skip})\,\} \qquad \mathbb{H} = \{\, h(p, q, o), h(q, p, o)\,\}}{\{\, a \cap \Diamonddot o\,\}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a \cap \underline{\Diamonddot} p \cap \underline{\Diamonddot} q\,\}\ \mathtt{skip}\ \{\, a \cap \Diamonddot o\,\}}
\qquad
\textsc{consequence-ti}\quad \frac{\mathbb{P}', \mathbb{I}', \mathbb{H}' \Vdash_{ti} \{\, a'\,\}\ \mathtt{st}\ \{\, b'\,\} \quad a \subseteq a' \quad \mathbb{P}' \subseteq \mathbb{P} \quad \mathbb{I}' \subseteq \mathbb{I} \quad \mathbb{H}' \subseteq \mathbb{H} \quad b' \subseteq b}{\mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a\,\}\ \mathtt{st}\ \{\, b\,\}}$$

$$\textsc{frame-ti}\quad \frac{\mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a\,\}\ \mathtt{st}\ \{\, b\,\}}{\mathbb{P} * c, \mathbb{I} * c, \mathbb{H} \Vdash_{ti} \{\, a * c\,\}\ \mathtt{st}\ \{\, b * c\,\}}
\qquad
\textsc{seq-ti}\quad \frac{\mathbb{P}_1, \mathbb{I}_1, \mathbb{H}_1 \Vdash_{ti} \{\, a\,\}\ \mathtt{st}_1\ \{\, b\,\} \qquad \mathbb{P}_2, \mathbb{I}_2, \mathbb{H}_2 \Vdash_{ti} \{\, b\,\}\ \mathtt{st}_2\ \{\, c\,\}}{\{b\} \cup \mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2, \mathbb{H}_1 \cup \mathbb{H}_2 \Vdash_{ti} \{\, a\,\}\ \mathtt{st}_1; \mathtt{st}_2\ \{\, c\,\}}$$

$$\textsc{loop-ti}\quad \frac{\mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a\,\}\ \mathtt{st}\ \{\, a\,\}}{\{a\} \cup \mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a\,\}\ \mathtt{st}^*\ \{\, a\,\}}
\qquad
\textsc{choice-ti}\quad \frac{\mathbb{P}_1, \mathbb{I}_1, \mathbb{H}_1 \Vdash_{ti} \{\, a\,\}\ \mathtt{st}_1\ \{\, b\,\} \qquad \mathbb{P}_2, \mathbb{I}_2, \mathbb{H}_2 \Vdash_{ti} \{\, a\,\}\ \mathtt{st}_2\ \{\, b\,\}}{\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2, \mathbb{H}_1 \cup \mathbb{H}_2 \Vdash_{ti} \{\, a\,\}\ \mathtt{st}_1 + \mathtt{st}_2\ \{\, b\,\}}$$

Fig. 3. Program logic from Meyer et al. [2022a] with our extension of hypotheses and temporal interpolation. We denote the former by $\Vdash$ and the latter by $\Vdash_{ti}$ (the subscript is short for "temporal interpolation").

needed to harmonize the implicit treatment of $\mathrm{Gov}(\mathbb{I})$ with framing. However, one can easily avoid the `skip` by applying the rule to the preceding command. The state predicates p and q should be intuitionistic. This is also related to framing. Rule temporal-interpolation-unordered is a variant in which we do not know whether p or q has been observed first and we rely on both hyptheses.

The hypotheses spawned by temporal-interpolation have to be discharged against the full set of interferences collected for the overall program. This is the reason why we work with hypotheses as parameterized Hoare triples rather than ordinary Hoare triples: in the moment we interpolate, we do not yet know the full set of interferences. Instead, we may only have a fraction of the program (and hence the interferences) at hand. It is also the reason why the separation logic judgements given in Figure 3 maintain a set $\mathbb{H}$ of hypotheses, and the rules are modified to join these sets. We are not allowed to forget a hypothesis while building up the correctness judgement for the overall program.

We elaborate on why we weaken a to a' in the definition of $\mathbb{I} \checkmark h$. The purpose of temporal-interpolation is to derive $\underline{\Diamonddot} o$ from $\underline{\Diamonddot} p \cap \_q$. Typically, p occurs within a weak past predicate, because it is not interference-free. This means no interference-free set of predicates $\mathbb{P}$ can prove the hypothesis $\{\, \_p\,\}\ \mathtt{2stmt}(\mathbb{X})\ \{\, \_q \rightarrow \underline{\Diamonddot} o\,\}$. A way out would be to prove the hypothesis for a weaker predicate $p \subseteq p'$ and replace the predicate $\underline{\Diamonddot} p$ in the main proof by $\underline{\Diamonddot} p'$. Unfortunately, the predicates p that require temporal interpolation not only fail the interference freedom test, it also seems to be impossible to weaken them to interference-free state predicates. All we can do is weaken them by introducing past information. Consider the example of a distributed counter given in §2. There, p is the predicate $c.\mathtt{l} \mapsto n_l * c.\mathtt{r} \mapsto n_r \wedge n_r \leq n'_r$. We weaken it to the invariant inv defined as $\exists n''_l\, n''_r.\ c.\mathtt{l} \mapsto n''_l * c.\mathtt{r} \mapsto n''_r \wedge n_l \leq n''_l \wedge n''_l + n''_r < n_l + n'_r \vee \underline{\Diamonddot}(\mathtt{counter}(c, n_l + n'_r))$. Although we have $\_p \subseteq inv$, the invariant does not have the shape $\_p'$. This means the invariant does not lead to a hypothesis $h(p', q, o)$ as required for temporal interpolation. By weakening the condition of when $h(p, q, o)$ holds, we bridge the gap between $\_p$ and inv.

Hypotheses require an ordinary program proof, using a method of choice. Yet, their shape suggests an invariance-based proof strategy: since program $\mathtt{2stmt}(\mathbb{I})$ repeats self-interferences $\mathtt{2com}(a, \mathtt{com})$, it suffices to find a predicate that is stable under these commands, contains the precondition, and entails the postcondition. Call $inv \subseteq \Sigma^+$ an *inductive invariant for* $\mathbb{I}$ if $[\![\![\mathtt{2com}(a, \mathtt{com})]\!]\!](inv) \subseteq inv$ for all $(a, \mathtt{com}) \in \mathbb{I}$ and $\boxast_{\mathbb{I}}\ inv$. We say that *inv proves* $h(p, q, o)$, if $\_p \subseteq inv$ and $inv \cap \_q \subseteq \underline{\Diamonddot} o$.

Lemma 4.2 (Strategy). *Let inv be an inductive invariant for* $\mathbb{I}$ *proving* $h(p, q, o)$. *Then* $\mathbb{I} \checkmark h(p, q, o)$.

4.1 Soundness

We show that every proof in the new program logic of Figure 3 gives rise to a proof in the program logic of §3, provided the hypotheses hold for the overall set of interferences. Also successful interference freedom checks will carry over. This means we can take full advantage of temporal interpolation, trusting that a traditional program proof will exist which discharges all hypotheses along the way. Temporal interpolation can therefore be understood as a way of structuring and shortening traditional program proofs that involve temporal reasoning. Technically, soundness shows that any derivation in the new program logic can be strengthened by an intersection with $\mathsf{Gov}(\mathbb{I})$. This allows us to replace TEMPORAL-INTERPOLATION by CONSEQUENCE-TI relying on Lemma 4.1.

THEOREM 4.3 (SOUNDNESS). *Consider a derivation* $\mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a \,\}\, \mathtt{st} \,\{\, b \,\}$ *with* $a \in \mathbb{P}$, $\boxplus_{\mathbb{I}}\, \mathbb{P}$, *and* $\mathbb{I} \checkmark \mathbb{H}$. *Then* $\mathbb{P} \cap \mathsf{Gov}(\mathbb{I}), \mathbb{I} \Vdash \{\, a \cap \mathsf{Gov}(\mathbb{I}) \,\}\, \mathtt{st} \,\{\, b \cap \mathsf{Gov}(\mathbb{I}) \,\}$ *with* $a \cap \mathsf{Gov}(\mathbb{I}) \in \mathbb{P} \cap \mathsf{Gov}(\mathbb{I})$ *and* $\boxplus_{\mathbb{I}}\, (\mathbb{P} \cap \mathsf{Gov}(\mathbb{I}))$.

The difficulty in proving the theorem is the interplay between the intersection we intend to add and the frame rule. Therefore, our first step is to eliminate the frame rule and show that whenever a correctness statement can be derived, then it can be derived without FRAME-TI. Let $\Vdash_{ti,nf}$ denote the restriction of $\Vdash_{ti}$ that avoids FRAME-TI.

LEMMA 4.4 (FRAME-TI ELIMINATION). $\mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti} \{\, a \,\}\, \mathtt{st} \,\{\, b \,\}$ *iff* $\mathbb{P}, \mathbb{I}, \mathbb{H} \Vdash_{ti,nf} \{\, a \,\}\, \mathtt{st} \,\{\, b \,\}$.

At the heart of the lemma is the fact that the frame rule commutes with the remaining rules of the program logic. This allows us to organize proofs in such a way that the frame rule is applied right after COM-TI. A combination of COM-TI and FRAME-TI, in turn, can be captured by COM-TI alone. The difficult case is TEMPORAL-INTERPOLATION, for the proof of which we rely on the following identity.

LEMMA 4.5. $b * c \cap \Diamonddot o = (b \cap \Diamonddot o) * c$.

With the previous result, the derivation that makes use of temporal interpolation can be assumed to be FRAME-TI-free. We now show that also TEMPORAL-INTERPOLATION can be eliminated, provided we strengthen the correctness statement by the governed computations.

LEMMA 4.6. *If* $\mathbb{P}, \mathbb{I}', \mathbb{H} \Vdash_{ti,nf} \{\, a \,\}\, \mathtt{st} \,\{\, b \,\}$ *is derivable, then for all* $\mathbb{I}$ *with* $\mathbb{I}' \subseteq \mathbb{I}$ *and* $\mathbb{I} \checkmark \mathbb{H}$ *we have* $\mathbb{P} \cap \mathsf{Gov}(\mathbb{I}), \mathbb{I} \Vdash \{\, a \cap \mathsf{Gov}(\mathbb{I}) \,\}\, \mathtt{st} \,\{\, b \cap \mathsf{Gov}(\mathbb{I}) \,\}$.

The previous lemmas allow us to prove Theorem 4.3. For interference freedom, note that the governed computations are interference-free, $\boxplus_{\mathbb{I}}\, \mathsf{Gov}(\mathbb{I})$, and we have $\boxplus_{\mathbb{I}}\, \mathbb{P}$ by the assumption. The intersection of two interference-free predicates is interference-free.

5 TEMPORAL INTERPOLATION FOR LINEARIZABILITY

We present an extension of our program logic from §4 to verify linearizability. The approach is akin to atomic triples [da Rocha Pinto et al. 2014], except that we do not aim to support compositional reasoning about clients against atomic specifications of libraries. Instead, we only focus on verifying library implementations. We use update tokens that encode a method's obligation to execute a linearization point. Once the method executes a command that resembles the linearization point, the update token is traded into a receipt token certifying successful linearization. This also prevents the method from having further linearization points since tokens are not duplicable and thus no more tokens can be traded. Here, we focus on concurrent search structures (CSS), however, the approach applies more generally. Sequential specifications Ψ of concurrent search structure methods op and key k take the following form:

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

$$\textsc{com-lin-void}\quad \frac{\mathbb{P},\mathbb{I},\mathbb{H} \Vdash_{ti} \{\, a \,\}\ \mathtt{com}\ \{\, b \,\} \qquad a \subseteq \mathsf{CSS}(C) \qquad b \subseteq \mathsf{CSS}(C)}{\mathbb{P},\mathbb{I},\mathbb{H} \Vdash^{lin}_{ti} \{\, a \,\}\ \mathtt{com}\ \{\, b \,\}}$$

$$\textsc{com-lin-pure}\quad \frac{a \subseteq \Diamond\big(\mathsf{CSS}(C) * \mathsf{UP}(C,C,y,v)\big) \qquad \mathbb{P} = \mathsf{RCT}_v * a \qquad \mathbb{I} = \{\, (a, \mathtt{skip}) \,\} \times (\mathsf{OBL} \rightsquigarrow \mathsf{RCT}_v)}{\mathbb{P},\mathbb{I},\mathbb{H} \Vdash^{lin}_{ti} \{\, \mathsf{OBL} * a \,\}\ \mathtt{skip}\ \{\, \mathsf{RCT}_v * a \,\}}$$

$$\textsc{com-lin-impure}\quad \frac{a \subseteq \mathsf{CSS}(C) \qquad \mathbb{P},\mathbb{I},\mathbb{H} \Vdash_{ti} \{\, a \,\}\ \mathtt{com}\ \{\, b \,\} \qquad b \subseteq \mathsf{CSS}(C') \cap \mathsf{UP}(C,C',y,v)}{\mathbb{P} * \mathsf{RCT}_v,\ \mathbb{I} \times (\mathsf{OBL} \rightsquigarrow \mathsf{RCT}_v),\ \mathbb{H} \Vdash^{lin}_{ti} \{\, \mathsf{OBL} * a \,\}\ \mathtt{com}\ \{\, \mathsf{RCT}_v * b \,\}}$$

Fig. 4. Proof rules for commands that ensure proper handling of the linearizability tokens OBL and RCT.

Here, C and C' are the logical contents of the structure before and after the operation takes effect. The predicate $\mathsf{CSS}(C)$ ties the physical state of the structure to C. How the method call $op(k)$ changes the contents is prescribed by the relation $\mathsf{UP}(C, C', k, v)$.

The linearizability obligation is denoted by OBL_Ψ and the receipt token by $\mathsf{RCT}_{\Psi,v}$, and we drop Ψ if it is clear from the context. Receipts are parameterized in the result value of the operation to reconcile the actual return value with the one prescribed by Ψ. For concurrent search structures, the sequential specifications of the methods `contains`(k), `insert`(k), and `delete`(k) are as expected and we denote their obligations by CTN_k, INS_k, and DEL_k (their receipts are just RCT_v).

To deal with the tokens in a proof, we lift the proof system $\Vdash_{ti}$ from §4 to a new proof system $\Vdash^{lin}_{ti}$ which inherits all the rules of $\Vdash_{ti}$ except for Rule com-ti. Rule com-ti is replaced by the three new rules from Figure 4. The rules extract the tokens, invoke $\Vdash_{ti}$, and then add the tokens back. However, in the process, they potentially transform the tokens if a linearization point is registered. That is, the updates of tokens are handled by $\Vdash^{lin}_{ti}$ rather than $\Vdash_{ti}$. To do this, we lift the program semantics $[\![\mathtt{com}]\!]$ in a trivial way: the ghost component of the state is simply ignored. However, for temporal interpolation to remain sound, we need to capture the effect of ghost state updates in the interferences. So, we decorate commands $\mathtt{com} \times (\mathsf{OBL} \rightsquigarrow \mathsf{RCT}_v)$. Then, decorating an interference $(a, \mathtt{com})$ decorates the command and adds the required token to the premise, $(a, \mathtt{com}) \times (\mathsf{OBL} \rightsquigarrow \mathsf{RCT}_v) = (a * \mathsf{OBL}, \mathtt{com} \times (\mathsf{OBL} \rightsquigarrow \mathsf{RCT}_v))$. With this, we are ready for the proof rules of $\Vdash^{lin}_{ti}$.

Rule com-lin-void deals with commands that do not alter the logical contents of the structure. Consequently, they maintain the current obligation/receipt token. Rule com-lin-impure trades an obligation for a receipt if the executed command is the linearization point, that is, if it updates the logical contents of the structure according to the sequential specification. If a command changes the logical contents but does not satisfy the specification or has no obligation token, the proof fails. Rule com-lin-pure also trades an obligation for a receipt. However, the rule does so in hindsight. That is, there is no need to perform the trade at the very moment the sequential specification is satisfied, it can be done later if a past predicate can certify the existence of the linearization point. It is this rule that sets our approach apart from atomic triples [da Rocha Pinto et al. 2014]. We allow for this retrospective linearization only if the linearization point is pure, i.e., does not alter the logical contents of the structure. The reason is this: such pure linearization points can be used by arbitrarily many threads to linearize whereas impure linearization points require a one-to-one correspondence to threads. The approach can be extended to support impure linearization points.

Theorem 5.1. *If there are* $\mathbb{P},\mathbb{I},\mathbb{H}$ *with* $\mathbb{P},\mathbb{I},\mathbb{H} \Vdash^{lin}_{ti} \{\, \mathsf{CSS}(\bullet) * \mathsf{OBL}_\Psi \,\}\ \mathtt{st}\ \{\, res.\ \mathsf{CSS}(\bullet) * \mathsf{RCT}_{\Psi,res} \,\}$ *and* $\mathsf{CSS}(\bullet) * \mathsf{OBL}_\Psi \in \mathbb{P}$ *and* $\boxplus_{\mathbb{I}}\, \mathbb{P}$ *and* $\mathbb{I} \checkmark \mathbb{H}$*, then* `st` *is linearizable wrt.* Ψ.

6 CASE STUDY: THE LO-TREE

We substantiate the usefulness of the developed program logic by verifying the linearizability of a challenging concurrent data structure: the the logical-ordering (LO-)tree [Drachsler et al. 2014].

We identify and fix bugs in the original implementation from Drachsler et al. [2014] as well as in the correction attempt by Feldman et al. [2020].

6.1 The LO-Tree in a Nutshell

Overview. The LO-tree [Drachsler et al. 2014] is a self-balancing binary search tree implementing a set data type. Self-balancing refers to the tree periodically restructuring itself to maintain a low height in order to speed up accesses. The restructuring mechanism in the LO-tree are standard tree rotations. For an example rotation consider Figure 5. There, node 13 experiences a *right rotation*: its left child 7 takes the position of node 13 and node 13 becomes the right child of 7. The formerly right subtree of 7 becomes the left subtree of 13. The resulting tree is a binary search tree again.

In a concurrent setting, rotations pose a major challenge. To avoid performance bottlenecks, one wishes to traverse the tree without synchronization, e.g., without acquiring locks that prevent rotations from happening. Without synchronization, however, one cannot prevent traversals to *go astray* in the presence of rotations. In Figure 5, if a tree traversal searching for node 5 arrives at node 13 and node 13 experiences the right rotation before the tree traversal continues, then the tree traversal will never reach node 5 but end up at node 9. For the implementation to be linearizable, it must detect this and be able to find node 5 despite the rotation.

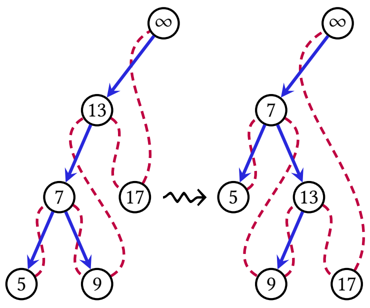

Fig. 5. A right rotation of the node storing 13. While the tree layout changes, the logical ordering remains unaffected.

The LO-tree solves the problem by organizing the nodes in a doubly-linked list, the eponymous logical ordering. In fact, it is this list which dictates the contents of the LO-tree. The tree structure is merely an overlay to that list which helps to speed up accesses. In Figure 5, the logical ordering - - - contains all nodes in ascending order while the tree overlay ⟶ does not yet contain node 17. Hence, the previous tree traversal, which arrives at node 9 on its way to node 5, can follow the logical ordering backward to find 5. Similarly, a tree traversal searching for 17 arrives at node 13 and then follows the logical ordering forward to find it.

Implementation. We link the above ideas to the implementation of the LO-tree in Figure 6 (ignore the proof outline annotations for now). The nodes of the tree are represented by the struct type `Node`. Each node stores an integer `key` and a Boolean `mark` as well as several pointers and locks. The `mark` field is used to indicate that the node is being or has been removed from the tree. For the doubly-linked logical ordering list each node stores a forward `succ` and a backward `pred` pointer. To synchronize mutations of the list, there is a lock `listLock`. For the tree overlay, each node stores pointers `left` and `right` to its children and a pointer `parent` to its parent. Tree mutations are synchronized with a lock `treeLock`. There are two sentinel nodes `min` resp. `max` storing values $-\infty$ resp. ∞. The initial logical ordering consists of these two nodes. The root of the tree is `max`.

The user-facing API of the LO-tree consists of the three methods of a concurrent search structure: `contains`, `insert`, and `delete`. The methods return a Boolean indicating success of the operation. Methods `insert` and `remove` use fine-grained locking to synchronize mutators. Both methods rely on the helper method `locate`(k) which finds (and locks) the position in the logical ordering to which value k belongs. This position can be thought of as the interval between two successive nodes x and z, x`.succ` $= z$, so that k is logically ordered between the two or in z, $k \in (x$`.key`$, z$`.key`$]$. To arrive at this location, a straightforward binary tree traversal is used, as implemented by `traverse`(k). Since the traversal may yield x or z depending on the tree structure, the remaining node is determined using `pred`/`succ` of the logical ordering. To account for the tree traversal going astray due to rotations, `locate` validates the found position. More precisely, it checks for $k \in (x\texttt{.key}, z\texttt{.key}]$ and ensures that x is unmarked, i.e., still part of the logical ordering. The validation happens after locking `listLock` of x so that the position cannot be invalidated by concurrent mutators.

Insertions of value k proceed as follows. They first `locate` the position x, z in the logical ordering where k should be inserted. The returned position also reveals whether k is already present in the logical ordering. If so, the insertion fails and returns *false*. Otherwise, a new node y is inserted in between x and z. The new node's `pred` and `succ` are pointed to x and z, respectively. Then, y is inserted into the logical ordering. It is first inserted into the forward ordering by pointing x`.succ` to y. Only after this, it is inserted into the backward ordering by pointing z`.pred` to y. This order deviates from the original version [Drachsler et al. 2014] for reasons we explain in §6.2. Finally, y is inserted into the tree by a call to `performTreeInsertion(`y`,`p`)`. This call expects the node p that is the parent of x. The parent p is determined before x is inserted into the logical ordering by `prepareTreeInsertion(`x`,`z`)`, which does not alter the logical ordering nor the tree but may acquire locks. We do not got into the details of the tree modifications as they are orthogonal to our linearizability proof. Finally, *true* is returned by `insert`.

Deletions of value k are similar to insertions. They `locate` the position x, y where k resides. If $y\texttt{.key} \neq k$, then k is not present and the deletion fails, returning *false*. Otherwise, it acquires y's `listLock` and reads y's successor z. To remove y, it is marked by setting $y\texttt{.mark} = \mathit{true}$, unlinked from the backward logical ordering by setting $z\texttt{.pred} = x$, and then unlinked from the forward logical ordering by setting $x\texttt{.succ} = z$. Afterwards, y is removed from the tree using `performTreeDeletion(`y`)` which expects `prepareTreeDeletion(`y`)` has been called before y was marked. Similar to insertions, `prepareTreeDeletion` does not alter the logical ordering nor the tree but may acquire locks. Again, we elide `performTreeDeletion` and `prepareTreeDeletion` as they are unimportant for our discussion.

Unlike the above mutations, the `contains(`k`)` method is wait-free, in particular it does not acquire locks. It traverses the tree, follows `pred` pointers, and finally follows `succ` pointers to check whether there is an unmarked node containing k. In addition to the original version [Drachsler et al. 2014], we need to follow `pred` pointers at least until the first unmarked node to guarantee that k is found indeed, see §6.2.

6.2 Bugs and their Fixes

The original version of the LO-tree [Drachsler et al. 2014] has two bugs which we fixed in Figure 6. Due to space constraints, we refer to the technical report [Meyer et al. 2022b] for more details.

Bug 1: Duplicate Values. A subtle quirk of the LO-tree is the fact that an insertion of value k may be unaware of a concurrent deletion of k because the tree traversal of the insertion experienced a rotation but still ended up in the right position for the insertion (the validation in `locate` succeeds). Successful validation requires that the deletion already removed k from the logical ordering. So, the insertion can proceed and insert k into the logical ordering and into the tree. If the deletion has not yet removed the old marked version of k, then the tree contains two nodes with value k that disagree on the mark bit. Hence, rotations influence the result of `contains(`k`)`—it is not linearizable.

Our implementation from Figure 6 fixes the above problem by adding Line 55: the logical ordering is followed backward (`pred` fields) at least until an unmarked node is encountered. This ensures that the final result is not *confused* by concurrent deletions. Other than that `contains` proceeds as originally devised by Drachsler et al. [2014]. Interestingly, adding Line 55 renders the mark bit check on Line 59 superfluous.

Bug 2: Insertion Order. Feldman et al. [2020] identified another bug in the `insert` method. In the original version [Drachsler et al. 2014], new nodes are inserted first into the backward logical ordering and then into the forward one (compared to Figure 6, Lines 89 and 91 are reversed). To see

```
struct Node { int key; bool mark; Lock treeLock, listLock; Node* left, right, parent, pred, succ; }
val min = new Node { key = -∞; mark = false; }; val max = new Node { key = ∞; mark = false; }
min.pred, min.succ := max, max; max.pred, max.succ := min, min
```

$\mathsf{LocInv}(C,N,x,z) \triangleq \mathsf{Inv}(C, N \cup x \cup z) * key(x) < k \le key(z)$
$\quad * \mathsf{Locked}(x) * k \in \mathsf{KS}(z) * z = succ(x) * \neg mark(x)$

$\mathsf{LinkInv}(C,N,x,y,z) \triangleq \mathsf{Inv}(C, N \cup x \cup y \cup z) * \mathsf{IS}(x) \neq \emptyset$
$\quad * \mathsf{Locked}(x) * \mathsf{Locked}(y) * y = succ(x) * z = succ(y)$
$\quad * \neg mark(x) * key(x) < k = key(y) < key(z)$

$\mathsf{SuccInv}(C, C', N, M, x, v) \triangleq \mathsf{Inv}(C, N \cup v) * x = v$
$\quad \cap \Diamond\,({}^{\backprime}\mathsf{Inv}(C', M \cup v) * k \in {}^{\backprime}\mathsf{IS}(v))$

```
{ ∃C, N. Inv(C, N) }
method traverse(k: Int): Node {
  val y = max
  while (true) { { Inv(C, N ∪ y) }
    val c = k < y.key ? y.left : y.right
    if (y.key == k || c == NULL) return y
    { Inv(C, N ∪ c) } y := c
} }
{ y. ∃C, N. Inv(C, N ∪ y) }

{ ∃C, N. CTN_k * Inv(N) * −∞ < k < ∞ }
method contains(k: Int): Bool {
  val y = traverse(k)
  { CTN_k * Inv(C, N ∪ v) * y = v }
  while (k < y.key) { val x = y.pred; y := x }
  { CTN_k * Inv(C, N ∪ v) * y = v * key(v) ≤ k }
  while (y.mark) { val x = y.pred; y := x }
  { CTN_k * SuccInv(C, C', N, M, y, v) }
  while (y.key < k) { val z = y.succ; y := z }
  { CTN_k * SuccInv(C, C', N, M, y, v) * k ≤ key(v) }
  val res = y.key == k ~~&& !y.mark~~
  { CTN_k * Inv(C, N) * res = t
    * ⟐ (`Inv(C', M) * t ⇔ k ∈ C') }
  { RCT_res * Inv(C, N) } // hindsight
  return res
}
{ res. ∃C, M. RCT_res * Inv(C, N) }

{ ∃C, N. Inv(C, N) * −∞ < k < ∞ }
method locate(k: Int): Node * Node {
  val y = traverse(k)
  val x = y.key < k ? y : y.pred
  lock(x.listLock)
  val z = x.succ
  { Inv(C, N ∪ x ∪ z) * Locked(x) * z = succ(x) }
  if (x.key < k <= z.key && !x.mark) return x, z
  unlock(x.listLock); restart
}
{ x, z. ∃C, N. LocInv(C, N, x, z) }

{ ∃C, N. INS_k * Inv(C, N) * −∞ < k < ∞ }
method insert(k: Int): Bool {
  val x, z = locate(k)
  if (z.key == k) {
    { INS_k * LocInv(C, N, x, z) * k ∈ C }
    { RCT_false * LocInv(C, N, x, z) }
    unlock(x.listLock); return false
  }
  { INS_k * LocInv(C, N, x, z) * key(z) ≠ k ∉ C }
  val p = prepareTreeInsertion(x, z)
  val y = new Node { key := k; mark := false;
          pred := x; succ := z; parent := p }
  { INS_k * LocInv(C, N, x, z) * TreeIns(x, z)
    * key(y) = k ∉ C }
  x.succ := y  // logical insertion
  { RCT_true * LocInv(C, N, x, y) * TreeIns(x, z) *
    key(y) = k < key(z) }
  z.pred := y
  unlock(x.listLock)
  { RCT_true * Inv(C, N) * TreeIns(x, z) }
  performTreeInsertion(y, p); return true
}
{ res. ∃C, M. RCT_res * Inv(C, N) }

{ ∃C, N. DEL_k * Inv(C, N) * −∞ < k < ∞ }
method delete(k: Int): Bool {
  val x, y = locate(k)
  if (y.key != k) {
    { DEL_k * LocInv(C, N, x, y) * k ∉ C }
    { RCT_false * LocInv(C, N, x, y) }
    unlock(x.listLock); return false
  }
  lock(y.listLock)
  prepareTreeDeletion(y)
  val z = y.succ
  { DEL_k * LinkInv(C, N, x, y, z) * k ∈ C * TreeDel(y) }
  y.mark := true
  z.pred := x
  { DEL_k * LinkInv(C, N, x, y, z) * k ∈ C * TreeDel(y) }
  x.succ := z // logical deletion
  { RCT_true * LinkInv(C, N, x, y, z) * k ∉ C
    * TreeDel(y) }
  unlock(y.listLock); unlock(x.listLock)
  { RCT_true * Inv(C, N ∪ y) * TreeDel(y) }
  performTreeDeletion(y); return true
}
{ res. ∃C, M. RCT_res * Inv(C, N ∪ y) }
```

Fig. 6. Implementation, bug fixes, and linearizability proof outline of the LO-tree [Drachsler et al. 2014]. The proof of contains requires hindsight reasoning to handle the future-dependent linearization point, see §6.5.4.

why this is problematic, assume an insertion of a new node y with value k between nodes x and z already linked `z.pred` to y but `x.succ` is still pointing to z. Then, `contains(k)` will find y only if the tree traversal takes it to nodes that appear after z in the logical order. For earlier nodes, `contains` will only follow `succ` fields which cannot yet reach y. It is easy to see that this violates linearizability.

We fixed this bug by changing the order in which y is linked into the logical ordering, cf. Lines 89 and 91. Feldman et al. [2020] apply the same fix. However, they also change `insert` to link new nodes first into the tree overlay and then into the logical ordering (without modifying `contains`). This violates linearizability: if a new node y with value k is inserted into the tree but not yet into the logical ordering, `contains` will find k if and only if it is not affected by concurrent rotations.[1]

6.3 Local Reasoning Principle

While our program logic from §5 tells us how to establish linearizability, it leaves us with a hard task: show that a command does or does not alter the contents of the structure. The contents is defined inductively over the data structure graph. To localize the reasoning about this inductive quantity, we build on the keyset framework [Krishna et al. 2020a, 2021; Shasha and Goodman 1988].

Suppose the global data structure graph consists of a set of nodes N. We will define a predicate $\mathsf{Inv}(C, \mathcal{K}, N, M)$ that describes the resources and properties of a subregion $M \subseteq N$ in the graph. Here, C will be the *logical contents* of the subregion, which is the union of the logical contents $\mathsf{C}(x)$ of all nodes $x \in M$. The set $\mathcal{K}$ is the *keyset* of the region M, which consists of all those keys that *could be* in M. We require the invariant to guarantee $C \subseteq \mathcal{K}$. The keyset will be defined inductively over the graph structure as we explain below. We then define the invariant $\mathsf{CSS}(C)$ of the entire structure as follows: $\mathsf{CSS}(C) \triangleq \exists N.\ \mathsf{Inv}(C, (-\infty, \infty), N, N)$.

To enable local reasoning, we aim for a definition of Inv that yields the following compositionality:

$$\begin{aligned}\mathsf{Inv}(C, \mathcal{K}, N, M \uplus M') \iff \exists C_1, C_2, \mathcal{K}_1, \mathcal{K}_2.\ & \mathsf{Inv}(C_1, \mathcal{K}_2, N, M) * \mathsf{Inv}(C_2, \mathcal{K}_2, N, M') * \\ & C = C_1 \uplus C_2 * \mathcal{K} = \mathcal{K}_1 \uplus \mathcal{K}_2 .\end{aligned}$$

That is, the predicate allows us to decompose the graph arbitrarily into disjoint subregions M and M' and compose them back together. In particular, separating conjunction will guarantee that the keysets (and hence the logical contents) of disjoint subregions will also be disjoint.

For proofs, this means that we can focus our reasoning on appropriate fragments $\mathsf{Inv}(C, \mathcal{K}, N, M)$ with a small set M. When reasoning about updates we can focus on the fragment M that contains only those nodes whose fields or keysets change. As we will see, three nodes will suffice to handle the LO-tree. Also, Inv enables a local-to-global lifting of the specification $\mathsf{UP}(C, C', k, v)$ of our search structure methods. For example, if we have identified a fragment of the form $\mathsf{Inv}(C, \mathcal{K}, N, \{ x \})$ with $k \in \mathcal{K}$, then $k \in C = \mathsf{C}(x)$ iff k is in the logical contents of the entire structure.

Flows. To obtain a definition of Inv with the desired properties, we build on the flow framework [Krishna et al. 2018, 2020b] which enables local reasoning about inductively-defined quantities of graphs. We sketch the main ideas for our specific application of the flow framework to keysets.

Each node is augmented with a ghost quantity called *inset*. Intuitively, the inset of a node x is the set $\mathsf{IS}(x)$ of all keys k, such that a thread searching for k will traverse x. That x is traversed means that the search eventually considers x; the search may or may not continue from there. The keyset $\mathsf{KS}(x)$ of x is the subset of $\mathsf{IS}(x)$ for which the traversal will terminate at x. For the LO-tree, the inset is $\mathsf{IS}(\mathtt{min}) = [-\infty, \infty]$ for the root node of the logical ordering and for the remaining nodes it is obtained as a solution to the following recursive equation:

$$\mathsf{IS}(x) \;=\; \bigcup\nolimits_y \; y.\mathtt{succ} = x \ ?\ \mathsf{IS}(y) \cap (y.\mathsf{key}, +\infty] \ :\ \varnothing .$$

[1]This is a mistake in the proof of the LO-tree by Feldman et al. [2020]. We do not make any claims regarding the soundness of their meta theory.

We then define $\mathsf{KS}(x) = \mathsf{IS}(x) \cap [-\infty, x.\mathtt{key}]$. The inset propagates via succ links only, because it is the list of succ links that makes up the logical contents of the LO-tree, as alluded to in §6.1. With this, we formally express the logical contents of node x by $\mathsf{C}(x) \triangleq \{\, x.\mathtt{key} \,\} \cap \mathsf{KS}(x)$.

To express insets in a separation algebra, the flow framework adds an additional ghost resource component. The technical details are not relevant for our discussion. In our proofs, we use the separation algebras proposed by Meyer et al. [2022a] and defer the interested reader there. What is important here, is that the above definitions guarantee that the keysets of subregions are always disjoint.

6.4 The Structural Invariant

We use standard separation logic assertions to represent the semantic predicates used so far. In particular, we use (i) boxed assertions $\boxed{A}$ to denote that A refers to shared state [Vafeiadis 2008; Vafeiadis and Parkinson 2007], (ii) fractional permissions [Boyland 2003] for points-to predicates $\xmapsto{1/n}$ to allow reads but prevent interfering updates to lock-protected resources, and (iii) persistent points-to [Vindum and Birkedal 2021] predicates $\xmapsto{\square}$ to easily share knowledge about immutable fields.

We define a predicate $\mathsf{N}(x)$ for the shared resources of a node x. For simplicity, we assume that proofs are implicitly existentially closed. This enables the naming convention where a use of $f(x)$ in the outer proof context refers to the value of field f as defined within $\mathsf{N}(x)$. We define:

$$\mathsf{N}(x) \triangleq \boxed{\begin{array}{l} \;\; x.\mathtt{key} \xmapsto{\square} \mathit{key}(x) * \mathit{pred}(x).\mathtt{key} \xmapsto{\square} \mathit{key}(\mathit{pred}(x)) * \mathit{succ}(x).\mathtt{key} \xmapsto{\square} \mathit{key}(\mathit{succ}(x)) \\ * \, x.\mathtt{pred} \xmapsto{1} \mathit{pred}(x) * x.\mathtt{succ} \xmapsto{1/2} \mathit{succ}(x) * x.\mathtt{mark} \xmapsto{1/2} \mathit{mark}(x) * x.\mathtt{in} \xmapsto{1} \mathit{in}(x) \\ * \, x.\mathtt{listLock} \xmapsto{1/2} \mathit{llock}(x) * (\mathit{llock}(x) = 0 \twoheadrightarrow \mathsf{Guarded}(x)) * x.\mathtt{treeLock} \xmapsto{1} \mathit{tlock}(x) \\ * \, x.\mathtt{left} \xmapsto{1} \mathit{left}(x) * x.\mathtt{right} \xmapsto{1} \mathit{right}(x) * x.\mathtt{parent} \xmapsto{1} \mathit{parent}(x) \end{array}}$$

$$\mathsf{Guarded}(x) \triangleq x.\mathtt{listLock} \xmapsto{1/2} \mathit{llock}(x) * x.\mathtt{succ} \xmapsto{1/2} \mathit{succ}(x) * x.\mathtt{mark} \xmapsto{1/2} \mathit{mark}(x)$$

Field in is the ghost field storing the node's inflow (cf. §6.3). We use fractional permissions for the fields listLock, succ, and mark. The listLock protects succ which is why $\mathsf{N}(x)$ has a full permission for succ only if listLock is unlocked. Otherwise, there is half a permission, the other half is transferred to the local state of the locking thread. The setup for mark is similar.

As noted above, the lock protects the resources $\mathsf{Guarded}(x)$ whose ownership is transferred from the shared state to the local state of the thread acquiring the lock. To make this precise, we define $\mathsf{Locked}(x) \triangleq x.\mathtt{listLock} \xmapsto{1/2} 1 * \mathsf{Guarded}(x)$ and obtain the following behavior of locks:

$$\begin{array}{rl} & \{\, \mathsf{N}(x) \,\} \;\; \mathtt{lock}(x.\mathtt{listLock}) \;\; \{\, \mathsf{N}(x) * \mathsf{Locked}(x) \,\} \\ \text{and} & \{\, \mathsf{N}(x) * \mathsf{Locked}(x) \,\} \;\; \mathtt{unlock}(x.\mathtt{listLock}) \;\; \{\, \mathsf{N}(x) \,\} \end{array}$$

For the first Hoare triple, note that its pre condition does not require $x.\mathtt{listLock}$ to be unlocked, $\mathit{llock}(x) = 0$. This is established by lock as it blocks until $x.\mathtt{listLock}$ can be acquired. The post condition realizes the ownership transfer: $\mathsf{Locked}(x)$ contains the protected resources $\mathsf{Guarded}(x)$ in the local state while maintaining the node's shared resources $\mathsf{N}(x)$.

With the resources of individual nodes set up, we are ready to state the invariant of the LO-tree:

$$\begin{array}{rcll} \mathsf{Inv}(C, \mathcal{K}, N, M) & \triangleq & \mathsf{SInv}(C, \mathcal{K}, N, M) * \mathop{\Huge *}_{x \in M} \mathsf{N}(x) * \mathsf{NInv}(N, M, x) & \\ \mathsf{SInv}(C, \mathcal{K}, N, M) & \triangleq & \mathtt{min}, \mathtt{max} \in N * \mathtt{nil} \notin N * M \subseteq N * C = \mathsf{C}(M) * \mathcal{K} = \mathsf{KS}(M) & \\ \mathsf{NInv}(N, M, x) & \triangleq & \mathsf{C}(x) \subseteq \mathsf{KS}(x) * \mathit{pred}(x), \mathit{succ}(x) \in N * \mathit{left}(x), \mathit{right}(x) \in N \cup \{\mathtt{nil}\} & \text{(I1)} \\ & & * \big(x = \mathtt{min} \Rightarrow \neg \mathit{mark}(x) * \mathit{key}(x) = -\infty\big) * \big(x = \mathtt{max} \Rightarrow \neg \mathit{mark}(x) * \mathit{key}(x) = \infty\big) & \text{(I2)} \\ & & * \big(\neg \mathit{mark}(x) \Rightarrow \mathsf{IS}(x) \neq \varnothing\big) * \big(\mathsf{IS}(x) \neq \varnothing \Rightarrow [\mathit{key}(x), \infty] \subseteq \mathsf{IS}(x)\big) * \mathit{indegree\text{-}one}(x) & \text{(I3)} \\ & & * \, \mathit{key}(\mathit{pred}(x)) < \mathit{key}(x) < \mathit{key}(\mathit{succ}(x)) & \text{(I4)} \end{array}$$

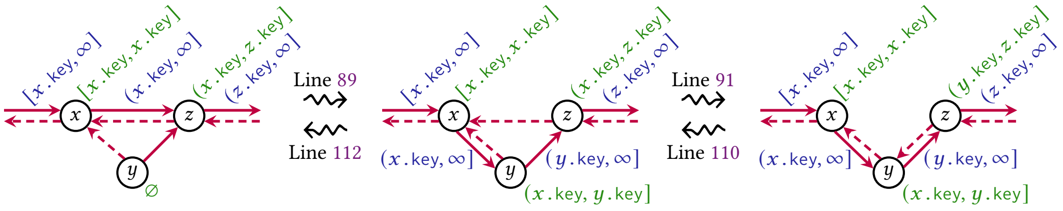

Fig. 7. Physical linking (Lines 89 and 91) and unlinking (Lines 110 and 112) of node y. The arrows $\dashrightarrow$ resp. $\longrightarrow$ indicate pred resp. succ pointers. The intervals on succ links denote the insets, the intervals on nodes denote their keysets. Mark bits (y is marked prior to the unlinking) and acquired locks are not depicted.

The invariant follows the form and satisfies the properties laid out in §5 and §6.3. Its main part is the node-x-local invariant NInv, which restricts the resources held by the overall invariant Inv. The properties are as follows. (I1) The contents of a node are governed by its keyset. Moreover, the invariant is closed under following pointer fields of x. Observe that we require the overall invariant containing full N to be closed, not the fragment comprising M. (I2) Nodes `min` resp. `max` are unmarked and store values $-\infty$ resp. ∞. (I3) Unmarked nodes have a non-empty inset which contains all values greater or equal to the node's own value. Moreover, nodes receive inset from at most one node, meaning that the `succ` list between `min` and `max` is a path. The abstract predicate *indegree-one*(x) can be expressed using flows. (I4) Nodes are sorted in the sense that a node's predecessor (successor) stores a lesser (greater) key. It is worth pointing out that (I4) is a node-x-local property indeed, because $\mathsf{N}(x)$ holds the required resources.

We may simply write $\mathsf{Inv}(C, M)$ instead of $\mathsf{Inv}(C, \mathcal{K}, N, M)$ if N is clear from the context.

6.5 Proof Outline

The proof outline can be found in Figure 6. While the proof for `insert` and `delete` requires mostly standard reasoning, it reveals the interference that other threads are subjected to. The hindsight reasoning for method `contains` is performed relative to this interference.

Using our proof system $\Vdash^{lin}_{ti}$, we give a proof *template* of the LO-tree: we do not make any assumptions about the operations manipulating the tree overlay other than them being memory-safe.

6.5.1 Locating Nodes. Recall from §6.1 that `insert` and `delete` use the helper `locate` to find the position x, z to which a given key k belongs. Node x is the result of a tree traversal, Line 67. Since we elide the mechanics of the tree overlay, we only know that the resulting pointer is non-`nil`—this little information suffices. Next, x is locked, Line 69. This provides us with the protected resources, $\mathsf{Guarded}(x)$. They guarantee that x.`succ` and x.`mark` cannot change due to interference. Reading x.`succ`, Line 70, binds z to $succ(x)$. Hence, the validation of position x, z on Line 72 results in the interference-free knowledge that x is unmarked, z is the successor of x, and that k indeed belongs in-between x and z, $k \in (key(x), key(z)]$. This together with the obtained resources forms the predicate $\mathsf{LocInv}(N, x, z)$, formally defined in Figure 6, and is the post condition of `locate` on Line 75. Later, we will use the fact that $\mathsf{LocInv}(N, x, z)$ implies $k \in \mathsf{KS}(z)$. To see this, invoke invariant (I3) for the unmarked x. We get $[key(x), \infty] \subseteq \mathsf{IS}(x)$. The keys $(key(x), \infty]$ distributes via x.`succ` as inset to z according to §6.3. Hence, $k \in \mathsf{KS}(z) = (key(x), key(z)]$.

6.5.2 Insertions. An Insertion of key k first calls `locate` to find the position x, z to which k belongs. The position reveals if k is already contained because $k \in \mathsf{KS}(z)$ as inferred above. If $key(z) = k$, then $k \in \mathsf{C}(z)$ and thus $k \in C$. That is, if the conditional in Line 79 succeeds, the specification of an unsuccessful insertion is met. We trade the obligation INS_k for the receipt RCT_{false}.

Otherwise, k is inserted into the structure. To do that, a new node y containing k is allocated in Line 86. The `pred` and `succ` fields are set to x and z, respectively. It remains to link y into the logical ordering, as depicted in Figure 7. First, Line 89 redirects x`.succ` to y. This is the linearization point: y receives the inset $(key(x), \infty]$ from x so that we get $\mathsf{C}(y) = \{ k \}$. Hence, the update turns C into $C \cup \{ k \}$ so that INS_k can be traded for RCT_{true}. Next, Line 91 redirects z`.pred` to y. The command has no effect on the logical contents C which is why we need no INS_k to proceed. It is readily checked that the update maintains the node-local invariants of the nodes x, y, z.

Our proof outline does not consider the methods for inserting the new node y into the tree overlay. We simply assume that `prepareTreeInsertion` in Line 85 produces an interference-free predicate $\mathsf{TreeIns}(x, z)$ that is maintained by the updates of the logical ordering in Lines 89 and 91 and consumed by the later `performTreeInsertion` in Line 94.

6.5.3 Deletions. Deletions are similar to insertions (see Figure 7). We omit the details.

6.5.4 Contains. The proof in Figure 6 uses implicitly existentially quantified symbolic variables v, u, t to share knowledge between now and past predicates. We cannot use program variables for this purpose because their values change during computation, meaning they may be valuated differently in now and past predicates. To further avoid confusion between now and past states, we write $`e$ to replace in expression $e$ all symbolic variables like $v$`.mark` with $`v$`.mark`. We think of $`e$ as the *old* version and use it under past operators. For example, in $\_\mathsf{Inv}(C, \{ v \}) * \diamonddot `\mathsf{Inv}(C', \{ v \})$ we would use $\mathsf{IS}(v)$ resp. $`\mathsf{IS}(v)$ to clearly refer to the inset of $v$ in the current resp. past state. The proof of `contains`$(k)$ has these five stages:

(1) The tree traversal, Line 51, finds a starting node y for traversing the logical ordering. The only guarantee for y is that it is non-`nil`, Line 52.

(2) The logical ordering is traversed by following `pred` fields as long as k is less than the key in the traversed node, Line 53. The resulting node y is non-`nil` by (I1). Moreover, we obtain the interference-free fact $k \geq y$`.key`, Line 54.

(3) The traversal continues to follow `pred` pointers until an unmarked node is reached, Line 55. By invariant (I1), the resulting node y is non-`nil`. That y is unmarked means that its inset is at least $[key(y), \infty]$ by invariant (I3). Moreover, $k \geq y$`.key` from the previous stage is preserved due to (I4). Together, this implies that k is in y's inset. This fact is not interference-free because y is not locked. To preserve it, we turn it into a past predicate, Line 56.

```
{ $\mathsf{CTN}_k * \mathsf{SuccInv}(C, C', N, M, y, v)$ }
while ($y$.key < $k$) {
  val $z$ = $y$.succ
  { $\mathsf{CTN}_k * \mathsf{SuccInv}(C, C', N, M, y, v)$
    $* key(v) < k * z = u = succ(v)$ }
  // $h(p, q, p \cap q)$ with
  //   $p \triangleq \mathsf{Inv}(C', M \cup v) * \mathsf{IS}(v) \neq \emptyset$
  //   $q \triangleq \mathsf{Inv}(C, N \cup v) * succ(v) = u * key(v) < k$
  skip
  { $\mathsf{CTN}_k * \mathsf{SuccInv}(C, C', N, M, z, v)$ }
  $y$ := $z$
}
{ $\mathsf{CTN}_k * \mathsf{SuccInv}(C, C', N, M, y, v) * k \leq key(v)$ }
```

Fig. 8. Detailed proof outline and temporal interpolation for Lines 56 to 58.

(4) The traversal follows `succ` pointers as long as k is greater than the key in the traversed node, Line 57. Using temporal interpolation (details below), we conclude that also the reached node y had k in its inset at some point. Note that this together with $k \leq y$`.key` from Line 58 means $k \in \mathsf{KS}(y)$ in some past state. So k was in the structure at this past state iff $k = key(y)$.

(5) Using temporal interpolation (details below), we derive from the past contents and the current `key` field of y whether or not k has been logically contained, Line 60. This past state is, in fact, the linearization point. We retrospectively linearize, Line 61, before returning.

We turn to the details of the temporal interpolation that goes into stages (4) and (5).

Temporal Interpolation in Stage (4). The proof outline for the loop from Line 57 is given in Figure 8. The temporal interpolation needed here is this: that y had flow in the past and its `succ` field currently points to z and its `key` field currently is less than k means that all three facts were true simultaneously at some point. Intuitively, this is the case because y has a non-empty inset whenever y.`succ` is changed and because $key(y)$ is never changed. Technically, we show the hypothesis $h(p, q, p \cap q)$ on Line 123 with

$$p \triangleq \mathsf{Inv}(C', M \cup v) * \mathsf{IS}(v) \neq \varnothing \qquad \text{and} \qquad q \triangleq \mathsf{Inv}(C, N \cup v) * succ(v) = u * key(v) < k\,.$$

The symbolic variables v resp. u are bound to y resp. x by the outer proof context; we use v/u instead of y/x as they are logically pure and thus do not change their valuation. To prove the hypothesis, we establish $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathtt{2stmt}(\mathbb{I})\ \{\, \_q \longrightarrow \underline{\diamonddot}\,(p \cap q)\,\}$ and $\boxast_{\mathbb{I}}\, \mathbb{P}$ for some set $\mathbb{P}$ of predicates with $\_p \subseteq a \in \mathbb{P}$ (cf. §4). We cannot simply use $a = \_p$ because p is not interference-free. Instead, we use $a \triangleq \_q \longrightarrow \underline{\diamonddot}\,(p \cap q)$. It is easy to see that a is weaker than $\_p$, $\_p \subseteq a$. Note that a is the invariant that the hypothesis proof strategy from Lemma 4.2 asks for.

Next, we show that a is interference-free, i.e., $[\![\![\,(c, \mathtt{com})\,]\!]\!]\,(a) \subseteq a$ for all interferences $(c, \mathtt{com}) \in \mathbb{I}$ of the LO-tree. For an interference $(c, \mathtt{com})$ to invalidate a it must change the truth of q in the current state. If the truth of q is changed to *false*, then a is vacuously true. Otherwise, the interference changes $succ(v)$ to u ($key(v)$ is not changed by any interference). This means `com` stems from Line 89 in `insert` or Line 112 in `delete`. In both cases we know from the proof (Figures 6 and 7) that v has a non-empty inset after the interfering update. Concretely, this means $[\![\![\,(c, \mathtt{com})\,]\!]\!]\,(a) \subseteq \_p$. Because we already established $\_p \subseteq a$, we obtain the interference-freedom of a, as required.

It remains to show that a is invariant under the self-interferences $\mathtt{2stmt}(\mathbb{I})$. To see this, observe that a concerns only the global state, not the local state. Hence, the self-interferences invalidate a iff the interferences of other threads do so. Since the latter is not the case, nothing needs to be shown.

With the hypothesis proved, we obtain $\diamonddot(p \cap q)$ from Rule TEMPORAL-INTERPOLATION. The rule is applied to a command which we make explicit in the form of `skip` on Line 126. One can avoid this `skip` by applying the rule together with the previous command. Finally, we invoke invariant (I3) under the past predicate to obtain $[`key(v), \infty] \subseteq `\mathsf{IS}(v)$. By definition, this means that $`succ(v) = u$ receives $`\mathsf{IS}(v) \setminus (`key(v), \infty]$. Because $`key(v) < k$, this means $k \in `\mathsf{IS}(u)$. Altogether, we arrive at the desired assertion on Line 127, namely $\diamonddot(`\mathsf{Inv}(C', M \cup u) * k \in `\mathsf{IS}(u))$.

Temporal Interpolation in Stage (5) We proceed in two steps. First, we prove that $h(p, q, p \cap q)$ holds for arbitrary p and $q \triangleq key(v) = t$. As before, we use Lemma 4.2 with invariant $a \triangleq \_q \longrightarrow \underline{\diamonddot}\,(p \cap q)$. Since $key(v)$ is immutable, a is immediately stable under (self-)interferences. This justifies to move facts about the `key` freely between now and past states.

Towards the assertion on Line 60, assume $key(v) = k$. We move this fact into the past predicate from Line 58 using the above argument. The result is: $\diamonddot(`\mathsf{Inv}(C', M) * k \in `\mathsf{IS}(v) * k = `key(v))$. This means that $k$ was contained in the structure in the past: $\diamonddot(`\mathsf{Inv}(C', M) * k \in `\mathsf{C}(v) \subseteq C')$. This conclusion uses the fact that $k \in `\mathsf{IS}(v) * k = `key(v)$ implies $k \in `\mathsf{KS}(v)$. The case for $key(v) \neq k$ is similar. Overall, rewriting both cases into one yields the desired assertion, Line 60. Finally, this allows us to retrospectively linearize as the past predicate witnesses a past state where k was resp. was not in the structure as reflected by the return value. This concludes the linearizability proof.

6.6 Proof Automation

We substantiate our claims that temporal interpolation and the resulting proof system for linearizability aid automated proof construction. To this end, we adapted the `plankton` tool [Meyer et al. 2022a]. `plankton` is a verifyer for non-blocking data structures that constructs proofs in the program logic from §3 extended by rules for linearizability akin to those from §5. To be more precise, `plankton` takes as input the implementation under scrutiny together with a candidate node invariant, like $\mathsf{NInv}(N, M, x)$ from §6.4. It then performs an exhaustive proof search.

We extended `plankton` to use our new proof rules from Figures 3 and 4, in particular Rule TEMPORAL-INTERPOLATION. Our implementation [Meyer et al. 2023] applies temporal interpolation only for

Table 1. Runtime comparison of our novel temporal interpolation proof rule with the tool from Meyer et al. [2022a]. The experiments were conducted on an Apple M1 Pro.

Benchmark	Meyer et al. [2022a]	This Paper	Factor
Fine-Grained set	$44s$ ✓	$45s$ ✓	×1.02
Lazy set	$1m\,21s$ ✓	$2m\,13s$ ✓	×1.65
FEMRS tree (no maintenance)	$2m\,22s$ ✓	$3m\,50s$ ✓	×1.62
Vechev&Yahav 2CAS set	$1m\,09s$ ✓	$1m\,15s$ ✓	×1.08
Vechev&Yahav CAS set	$0m\,52s$ ✓	$2m\,20s$ ✓	×2.70
ORVYY set	$0m\,54s$ ✓	$1m\,36s$ ✓	×1.79
Michael set	$3m\,06s$ ✓	$6m\,53s$ ✓	×2.22
Michael set (wait-free search)	$3m\,42s$ ✓	$6m\,53s$ ✓	×1.86
Harris set	$18m\,14s$ ✓	$57m\,20s$ ✓	×3.15
Harris set (wait-free search)	$19m\,54s$ ✓	$43m\,00s$ ✓	×2.16
LO-tree (maintenance stubs)	— ✗	$16m\,43s$ ✓	—

hypotheses of the form $h(p, q, p \cap q)$ and only if it is able to discharge the hypothesis using Lemma 4.2 with invariant $\_q \longrightarrow \_\diamond(p \cap q)$. This eager approach ensures that we do not pollute the proof search with temporal interpolations that are doomed to fail because their hypotheses do not hold. Note that this is possible despite a potentially incomplete interference set as `plankton` restarts proof construction whenever a new interference is discovered. Altogether, our implementation establishes linearizability results along Theorem 4.3.

We used our tool to verify automatically the LO-tree from Figure 6. Similarly to the presented proof, we did not use the actual implementation of the helper functions modifying the tree overlay. Instead, we used *most general stubs*, functions that change the tree overlay arbitrarily (leaving the logical ordering list unchanged). The node invariant we specified is the one from §6.4. With this, `plankton` is able to fully automatically construct a linearizability proof for the LO-tree within twenty minutes (see Table 1). We stress that this includes fully automatic applications of temporal interpolation, which are strictly necessary to prove the LO-tree linearizable.

We also compared our new version of `plankton` against the original version form Meyer et al. [2022a]. See Table 1 for the results: temporal interpolation incurs a slow down of factor 3.15 in the worst case and factor 2 on average. We believe that this slowdown is justified by the reasoning power brought by temporal interpolation. We consider a more extensive evaluation of our implementation future work. As of now, `plankton`'s proof construction is limited by orthogonal concerns (e.g. imprecise joins, the handling of updates with non-local effects) that still limit its applicability.

7 RELATED WORK

The hindsight principle [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010] and our temporal interpolation have relatives in classical program verification [Manna and Pnueli 1995; Schneider 1997]. So-called causality formulas, in our notation written as $\_p \longrightarrow \diamond q$, express that q is a prerequisite for seeing p. Temporal interpolation is more general in that it may take past information into account in order to infer the existence of an intermediary state. Yet, the past invariance proof principle by Manna and Pnueli [1995, §4.1] inspired an application of TEMPORAL-INTERPOLATION in the RDCSS proof [Meyer et al. 2022b, §F] to derive a contradiction in a case distinction. The careful identification of verification conditions by Manna and Pnueli [1995] has also lead us to the definition of hypotheses that can be proven in isolation. What sets our work apart is that we incorporate temporal interpolation into a modern program logic with powerful reasoning techniques [Jung et al. 2018] such as framing [O'Hearn et al. 2001], atomic triples [da Rocha Pinto et al. 2014], and general separation algebras [Calcagno et al. 2007], in particular flows [Krishna et al. 2018, 2020b].

There are first tools that automate linearizability proofs based on hindsight reasoning. The `poling` tool [Zhu et al. 2015] implements the hindsight lemma in the formulation of O'Hearn et al. [2010]. The `plankton` tool [Meyer et al. 2022a] automates a restricted form of hindsight reasoning that can be expressed via state-independent variables shared between a past and the current state. However, it did not support general temporal interpolation prior to our extension. Without this extension, the tool would have been unable to verify the LO-tree and other structures that require more complex hindsight reasoning.

We are not the first to study program logics defined over computations instead of states. History-based local rely-guarantee [Fu et al. 2010; Gotsman et al. 2013] has an elaborate assertion language whose temporal operators are carefully harmonized with the rules of the program logic. Our approach builds on the logic proposed by Meyer et al. [2022a] from which it inherits the notion of past predicates over computations. We introduce temporal interpolation by means of a new proof rule. The soundness result shows that the proof rule can be eliminated, and hence is really a mechanism for structuring complex proofs. This means that, in principle, all of our proofs can also be expressed in the logic of Meyer et al. [2022a]. Doing so, however, requires one to repeat the soundness arguments within each program proof anew. In particular, this (i) requires reasoning about the governed computations explicitly and (ii) thwarts the use of the frame rule. Realistically, this would make the proofs intractable, even manual ones. The conclusions [Meyer et al. 2022a] can draw directly about the past of the computation are all based on immutability arguments, and compared to what we propose here this is a very weak form of hindsight reasoning. Notably, the version of `plankton` presented in [Meyer et al. 2022a] cannot handle the example from §2 nor the LO-tree from §6. Comparing to other computation-based separation logics, we note that the formalization of computations matters: definitions based on interleaving products [Bell et al. 2010] or the union of sets of events [Delbianco et al. 2017; Sergey et al. 2015] seem to be less suited for temporal interpolation.

Prophecies were introduced to separation logic by Vafeiadis [2008] and formalized by Zhang et al. [2012] to structural prophecies that foresee the actions of one thread, a restriction overcome by Jung et al. [2020]. Temporal interpolation conducts full subproofs in the presence of interferences. However, it is in the nature of Owicki-Gries, and has been observed early on [Owicki and Gries 1976], that interferences may require auxiliary variables to increase precision. What seems to make prophecies more difficult to use is the need to reason about the computation backward, against the control flow [Bouajjani et al. 2017]. This is shared with simulation and refinement-based proofs [Liang and Feng 2013; Turon et al. 2013], where backward reasoning is known to be complete [Schellhorn et al. 2012].

Our proofs use standard techniques like boxed assertions [Vafeiadis 2008; Vafeiadis and Parkinson 2007], fractional permissions [Boyland 2003], and persistent points-to predicates [Vindum and Birkedal 2021]. Combining these techniques is no contribution of ours. In fact, they were already combined in the original `plankton` tool from Meyer et al. [2022a], although the use of fractional permissions and persistent points-to predicates has not been discussed there (probably due to their focus on lock-free implementations).

ACKNOWLEDGMENTS

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

DATA-AVAILABILITY STATEMENT

Our extended version of `plankton` and the dataset (Table 1) analysed in the present paper are available in the Zenode repository [Meyer et al. 2023], https://zenodo.org/record/7829982/.

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

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

Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. 2008. Thread Quantification for Concurrent Shape Analysis. In *CAV (Lecture Notes in Computer Science, Vol. 5123)*. Springer, 399–413. https://doi.org/10.1007/978-3-540-70545-1_37

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

John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

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

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

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

Dana Drachsler, Martin T. Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In *PPoPP*. ACM, 343–356. https://doi.org/10.1145/2555243.2555269

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (Lecture Notes in Computer Science, 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

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (Lecture Notes in Computer Science, 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 (Lecture Notes in Computer Science, 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

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

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

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

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

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, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. 2021. *Automated Verification of Concurrent Search Structures*. Morgan & Claypool Publishers. https://doi.org/10.2200/S01089ED1V01Y202104CSL013

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 (Lecture Notes in Computer Science, 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 (Lecture Notes in Computer Science, Vol. 9363)*. Springer, 356–370. https://doi.org/10.1007/978-3-662-48653-5_24

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. A concurrent program logic with a future and history. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1378–1407. https://doi.org/10.1145/3563337

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022b. Embedding Hindsight Reasoning in Separation Logic. *CoRR* abs/2209.13692 (2022). https://doi.org/10.48550/arXiv.2209.13692

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. *Artifact for "Embedding Hindsight Reasoning in Separation Logic"*. https://doi.org/10.5281/zenodo.7829982

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (Lecture Notes in Computer Science, 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

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to Prove Algorithms Linearisable. In *CAV (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 243–259. https://doi.org/10.1007/978-3-642-31424-7_21

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 (Lecture Notes in Computer Science, 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

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. ACM, 343–356. https://doi.org/10.1145/2429069.2429111

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

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP*. ACM, 76–90. https://doi.org/10.1145/3437992.3439930

Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A Structural Approach to Prophecy Variables. In *TAMC (Lecture Notes in Computer Science, Vol. 7287)*. Springer, 61–71. https://doi.org/10.1007/978-3-642-29952-0_12

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

Received 2022-11-10; accepted 2023-03-31