Completeness of Logical Atomicity for Linearizability in Concurrent Separation Logic

ZICHEN ZHANG, New York University, USA
SIMON ODDERSHEDE GREGERSEN, CISPA Helmholtz Center for Information Security, Germany
JOSEPH TASSAROTTI*, New York University, USA

Linearizability is a standard correctness condition for concurrent data structures. It guarantees that operations behave as if they took effect at some atomic instant between their call and return points. Despite the central role linearizability plays, in the context of concurrent separation logics, prior work has argued for instead using a style of specification that internalizes the atomicity of operations in terms of the logic's reasoning rules, known as *logical atomicity*. These logically atomic specifications are intended to be easier to compose inside of the logic than linearizability. Prior work has shown that in the Iris separation logic framework, a certain form of logically atomic specifications implies that a data structure is linearizable, which can be understood as a soundness theorem. However, the converse remained an open question: for every linearizable data structure, is it always possible to derive a corresponding logically atomic specification?

This paper resolves this question in the affirmative. We prove a completeness theorem for Iris that derives a logically atomic specification for any linearizable data structure. As a consequence, we are able to embed a variety of linearizability proof techniques into Iris and use them to derive logically atomic specifications. We apply this to three linearizability proof methods: aspect-oriented linearizability proofs, forward simulations with commit points, and meta-configuration tracking. Using these embeddings, we derive logically atomic specifications for the Herlihy-Wing queue and the Baskets Queue. We furthermore establish a connection between logical atomicity and an encoding of refinement in Iris that has been used in prior logical relations models. This result allows us to transport logically atomic specifications across refinements, which we apply to the Folly MPMC queue implementation. All of the results in this paper have been mechanized in the Rocq Prover.

Additional Key Words and Phrases: Iris, Separation Logic, Linearizability, Logical Atomicity, Prophecy Variables, Contextual Refinement

1 Introduction

Linearizability is a standard correctness condition for concurrent data structures [32, 33, 62]. It ensures that concurrent operations behave as if they took effect atomically in a way that is compatible with a sequential specification of the data structure. Because linearizability is such a widely used correctness condition for concurrent data structures, a vast array of formal methods techniques related to linearizability have been developed. These include various proof techniques [1, 6, 11, 20, 21, 30, 39, 56], program logics [28, 47, 69, 70], automated verifiers [50, 51, 72, 79], and tools for checking for linearizability violations [7, 10, 19, 27, 35, 44, 59]. Alongside this, researchers have developed various adaptations of linearizability, including relaxations for weak memory [8, 14, 16], more compositional formulations [74], restrictions that guarantee stronger properties [13, 25], and variants for durable or persistent memory [2, 3, 37].

Logical Atomicity. Yet despite the central role linearizability plays, a number of researchers have argued that linearizability, as formally defined, is not so easy to use to verify *clients* of a linearizable data structure. In particular, while concurrent separation logic (CSL) has been successfully used to verify a range of concurrent programs, and there are even ways to use CSL to *prove* that a

*Also affiliated with Amazon Web Services. This paper does not reflect the views of Amazon Web Services.

Authors' Contact Information: Zichen Zhang, zichenzhang@nyu.edu, New York University, New York, USA; Simon Oddershede Gregersen, gregersen@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany; Joseph Tassarotti, jt4767@nyu.edu, New York University, New York, USA.

data structure is linearizable, once such a specification has been shown, it is difficult to use in pre/post-condition style reasoning about clients. Thus, if one's goal is to use CSL to modularly verify a large application, linearizability may not be the most helpful specification for internal data structures and libraries used in the application. As a result, prior work has developed alternative specifications that capture the atomicity of a data structure's operations internally in a program logic in a way that is easier to use when verifying clients. Various forms of such *logically atomic* specifications have been developed [12, 38, 42, 55, 66, 67]. The common idea is that when an operation is logically atomic, a client can reason about that operation using the same reasoning patterns that are supported for the primitive, physically atomic commands of the language. For example, one of the key properties of physically atomic steps in most variants of CSLs is that they support *invariant* rules that allow a thread to assume that an invariant assertion P holds before the step, and require P to be reestablished after the step. Logically atomic operations are similarly allowed to access invariants, as long as the invariant can be shown to be preserved after the atomic operation takes effect. This enables modular and compositional reasoning, in which one logically atomic data structure can be used in the implementation of another.

There were long understood to be informal analogies between these logically atomic specifications and linearizability. For example, establishing logical atomicity typically requires identifying the atomic point where an operation's effect becomes visible, which is similar to proof techniques for linearizability that involve identifying the *linearization point*. Data structures whose linearizability is challenging to establish because of future-dependent linearization points similarly require special techniques like prophecy variables [1, 41] for establishing logical atomicity. Birkedal et al. [5] made this connection formal in one direction, by showing that if a data structure satisfies a logically atomic specification in Iris, a widely-used CSL framework [40, 42], then the data structure is linearizable.

But what about the opposite direction: if a data structure is linearizable, is it always possible to prove a logically atomic specification for it? We can view this as a kind of completeness property for logical atomicity, and view the previous result of Birkedal et al. as a soundness theorem. No prior work has established a completeness theorem for logical atomicity, but such a result would be useful for several reasons. First, it would tell us that the logic is not too "weak" or "missing" any rules or reasoning techniques: if a data structure is really linearizable, then at least in principle, a logically atomic specification can be shown using the logic. Second, it means that if we can prove, external to the logic, that a data structure is linearizable, then we can automatically reflect that into the logic as a logically atomic specification for the data structure. This is valuable because, as mentioned above, a range of proof techniques and automated verification methods have been developed for establishing linearizability, and many of these have no existing analogue inside Iris or other program logics. Instead of having to extend Iris and its formulation of logical atomicity to support these proof techniques, completeness for logical atomicity would allow us to just directly apply these proof techniques to establish linearizability and then obtain a logically atomic specification for free.

This paper proves the first completeness theorem for Iris's logically atomic triples. For concreteness, we establish our result for HeapLang, the default language included with Iris, but our technique is generally applicable to other languages used with the Iris framework. The key idea behind our proof is an inductive forward simulation-style argument. To derive the logically atomic Hoare triple about an operation, we maintain an invariant tracking the trace of call/return operations. From the assumption that the operation is linearizable, we know that there must exist some linearization point between the call/return operations, and we use these linearization points to know when to apply the "atomic update" assertion used in Iris's logically atomic triples. The main challenge is that formally, when given a partial prefix of execution, the definition of linearizability does not necessarily ensure that the linearization points we extract in this way are compatible with

future extensions to the trace. We solve this by instrumenting the methods of a concurrent data structure with *prophecy variables* that record the arguments of each method call and the eventual return value. Using these prophecy variables, the proof "predicts" the complete trace of calls and returns, so that we can extract a consistent set of linearization points from the linearizability assumption.

As alluded to above, by using this completeness theorem, we are able to prove linearizability of data structures using arbitrary proof methods and then derive logically atomic triples back in Iris. We apply this concretely with three linearizability proof techniques that have no existing Iris analogue: the aspect-oriented proof [30], the forward-simulation with commit points technique developed by Bouajjani et al. [6], and meta-configuration tracking developed by Jayanti et al. [39]. What is powerful about each of these methods is that they are capable of proving linearizability of data structures with future-dependent linearization points. We have applied these methods to prove logically atomic triples of two data structures with such future dependency, the Herlihy-Wing queue [33] and the Baskets Queue [34]. While our completeness results show that one could in principle just use prophecy variables directly to verify these data structures (and indeed, prior work has done so for the Herlihy-Wing queue [41]), we argue that it is useful to be able to reuse off-the-shelf proofs and ideas that have been developed in prior work, instead of being forced to redevelop a proof in terms of prophecy variables.

Refinement. In addition to linearizability and logical atomicity, there is a third commonly used approach to specifying concurrent data structures: refinement. Using contextual refinement, one can capture that a data structure behaves in an atomic way by proving that it refines a coarse-grained implementation that wraps a sequential implementation with a global lock or atomic block primitive. Because the contextual refinement shows that the data structure's observable behaviors are a subset of the coarse-grained version, clients can reason about the data structure *as if* it were atomic. Many formal verification techniques have been developed for relational reasoning about concurrent data structures and can be used for proving such refinements [29, 45, 48], including several based on concurrent separation logics and Iris in particular [23, 24, 46, 68].

How does this kind of contextual refinement specification relate to linearizability or logical atomicity? Filipović et al. [22] first proved an equivalence between linearizability and contextual refinement, and subsequent works have generalized this to a number of settings, including relaxed memory [18, 64] and durable variants of linearizability [75].

But for logical atomicity, there is no existing formal connection with contextual refinement. Some prior work has observed that from a logical atomicity specification, it is often easy to establish a contextual refinement to an atomic implementation of a data structure [23]. However, there has previously been no way to go from a proof of a refinement to a logically atomic triple.

The second main contribution of this paper is a result connecting logical atomicity to refinement. In particular, if e_I refines e_A, and e_A satisfies a logically atomic specification, then our result enables us to derive a logically atomic specification for e_I. Among other things, this is useful because it allows one to establish logical atomicity incrementally through a chain of refinements. Rather than having to prove e_I is logically atomic all at once, we can introduce a series of intermediary programs $e_1, \ldots, e_n$, and prove $e_I \preceq e_1$, $e_1 \preceq e_2$, $e_2 \preceq e_3$, $\ldots$, $e_n \preceq e_A$, deduce a logically atomic specification for e_A and then transport this transitively to a specification for e_I. This pattern of establishing chains of refinements is common in the literature based on refinement style reasoning, and our results make this accessible for the first time as a proof method for logical atomicity as well. For example, Vindum et al. [76] have proved that the MPMC queue from Meta's C++ library Folly [49] refines a coarse-grained queue with a global lock, but their result is not accessible from a

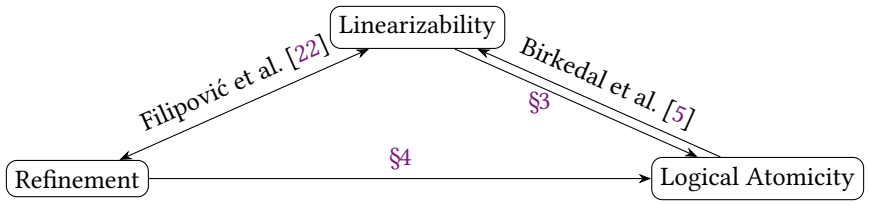

Fig. 1. Relation between Linearizability and Its Alternatives.

$$v, w \in \mathit{Val} ::= () \mid z \mid b \mid \ell \mid \texttt{rec}\ f(x) = e \mid (v, w) \mid \texttt{inl}(v) \mid \texttt{inr}(v) \mid \cdots \quad (z \in \mathbb{Z}, b \in \mathbb{B}, \ell \in \mathit{Loc})$$

$$\begin{aligned} e \in \mathit{Expr} ::= {} & v \mid x \mid \texttt{rec}\ f(x) = e \mid e_1(e_2) \mid \circledcirc_1 e \mid e_1 \circledcirc_2 e_2 \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid (e_1, e_2) \\ & \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid \texttt{inl}(e) \mid \texttt{inr}(e) \mid (\texttt{match}\ e\ \texttt{with}\ \texttt{inl} \Rightarrow e_1 \mid \texttt{inr} \Rightarrow e_2\ \texttt{end}) \\ & \mid \texttt{ref}\, e \mid !\, e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}(e_0, e_1, e_2) \mid \texttt{fork}\, \{e\} \mid \cdots \end{aligned}$$

$$K \in \mathit{Ctx} ::= \bullet \mid e(K) \mid K(v) \mid e \circledcirc_2 K \mid K \circledcirc_2 v \mid \cdots \quad \text{(Standard right-to-left evaluation context)}$$

$$\sigma \in \mathit{State} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \qquad \mathit{Cfg} \triangleq \mathit{Expr}^* \times \mathit{State}$$

BaseBeta
$$((\texttt{rec}\ f(x) = e)(v), \sigma) \to_{\text{base}} (e[(\texttt{rec}\ f(x) = e)/f][v/x], \sigma, \epsilon) \quad \cdots$$

BaseAlloc
$$\frac{\ell \notin \mathrm{dom}(\sigma)}{(\texttt{ref}\, v, \sigma) \to_{\text{base}} (\ell, \sigma[\ell \leftarrow v], \epsilon)}$$

BaseFork
$$(\texttt{fork}\, \{e\}, \sigma) \to_{\text{base}} ((), \sigma, [e]) \quad \cdots$$

Fig. 2. Syntax and Semantics for HeapLang (without Prophecy Variables).

unary separation logic. With the help of our technique, we are finally able to get a logically atomic triple for the Folly MPMC queue on top of their result.

Contributions. Figure 1 diagrammatically shows our theoretical results and where they fit with the existing literature. To summarize, our work makes the following contributions:

- The first completeness result for logical atomicity showing that any linearizable data structure satisfies a logically atomic triple specification.
- A proof connecting refinement to logical atomicity, which in particular allows atomic triples to be transported across a refinement.
- Machine-checked reformulations of various linearizability proof techniques in the context of Iris, and examples applying them to several challenging data structures.

Our work is mechanized in the Iris separation logic framework [40] and the Rocq Prover.

2 Preliminaries

This section gives preliminary background material needed for our results. We start with the semantics of the concurrent language we consider. Next, we give a formal definition of linearizability as used in our proof. Finally, we describe the basic features of the Iris separation logic that our completeness proof uses.

2.1 Language

Although our completeness proof method generalizes, for concreteness, our results are proven with respect to the default HeapLang included in the Iris Rocq development. This is a mini ML-like language with general references and concurrency primitives. Its syntax and semantics are shown in Figure 2. Constructions in this figure are standard. The language has syntactic categories for expressions e, values v, states σ, and evaluation contexts K. The operational semantics is first specified by a base step relation $(e_1, \sigma_1) \to_{\mathsf{base}} (e_2, \sigma_2, \vec{e}_f)$, which says that under state σ_1, expression e_1 reduces to e_2, updates the state to σ_2, and spawns threads $\vec{e}_f$. A pure step $e_1 \to_{\mathsf{pure}} e_2$ is a special base step that is deterministic and independent of the state. The base step relation is then closed under evaluation contexts K to get the primitive step relation $\to_{\mathsf{prim}}$, which is lifted to a concurrent thread pool step relation $\to_{\mathsf{tp}}$, as expressed in the following rules:

$$\textsc{BasePrim}\quad \frac{(e_1, \sigma_1) \to_{\mathsf{base}} (e_2, \sigma_2, \vec{e}_f)}{(K[e_1], \sigma_1) \to_{\mathsf{prim}} (K[e_2], \sigma_2, \vec{e}_f)} \qquad \textsc{PrimTp}\quad \frac{(e_1, \sigma_1) \to_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)}{(\vec{e}_a +\!\!+ e_1 :: \vec{e}_b, \sigma_1) \to_{\mathsf{tp}} (\vec{e}_a +\!\!+ e_2 :: \vec{e}_b +\!\!+ \vec{e}_f, \sigma_2)}$$

In particular, in PrimTp, the threads of the program are represented as a list of expressions, from which one expression is non-deterministically selected to step. In the resulting program configuration, that thread is updated along with the state, and the new list $\vec{e}_f$ of forked threads (if any) is appended to the end of the thread list. We denote the reflexive transitive closure of $\to_{\mathsf{tp}}$ as $\to_{\mathsf{tp}}^*$.

A standard correctness condition for a *closed* program is safety or partial correctness. It means executing a program e from initial state σ never gets stuck, and if e terminates with a value v and state σ', then $\phi(v, \sigma')$ holds.

Definition 2.1 (Safety). An expression e in state σ is safe *w.r.t.* ϕ, denoted as $\mathsf{safe}_\phi(e, \sigma)$, if

(1) $\forall v, \vec{e}', \sigma'.\ ([e], \sigma) \to_{\mathsf{tp}}^* (v :: \vec{e}', \sigma') \Rightarrow \phi(v, \sigma')$, and
(2) $\forall \vec{e}', \sigma'.\ ([e], \sigma) \to_{\mathsf{tp}}^* (\vec{e}', \sigma') \Rightarrow \forall e' \in \vec{e}'.\ e' \in \mathit{Val} \vee \mathsf{red}(e', \sigma')$,

where $\mathsf{red}(e, \sigma) \triangleq \exists e', \sigma', \vec{e}_f.\ (e, \sigma) \to_{\mathsf{prim}} (e', \sigma', \vec{e}_f)$. When ϕ is $(\lambda v, \sigma.\ \mathsf{True})$, we omit ϕ and write $\mathsf{safe}(e, \sigma)$.

2.2 Linearizability

For a *partial* program, *e.g.*, a library, in addition to safety, we also want to ensure the interaction between the library and other parts of the program is correct. Particularly for concurrent data structures, linearizability [32, 33, 62] is a standard correctness condition that relates a concurrent object's behaviors to a sequential specification. There are multiple equivalent definitions of linearizability. In this paper, we use a version that is stated in terms of *linearization points*, following the definition used by Birkedal et al. [5]. At a high level, this definition requires that the execution history H of a concurrent object can be annotated with marked *linearization points*, which are instantaneous moments at which an operation's effect appears to take place. By ordering operations according to the order of their linearization points, we obtain a total order on operations. Then, the observed call/return behaviors of the execution must match those of a sequential execution that runs operations in this total order.

To state this definition formally, we will assume that the interface for a data structure supports two functions, `init`, which creates a new object, and `op`, which takes an object *obj* and an argument x and invokes an operation on the object with request x. Restricting to a single `op` function does not result in a loss of generality, because multiple operations can be emulated by including a tag as part of the argument that selects the intended operation.

Sequential Behavior. The sequential specification of an object is given in terms of an abstract type S of object states. This state is initialized to $s_0 \in S$ upon the creation of an object. The effects of operations are specified in terms of relations on this state.

Definition 2.2 (Sequential Behavior). A sequential behavior μ of data structure $(\texttt{init}, \texttt{op})$ is a triple of $(s_0, vr, rr) : S \times (Val \to Prop) \times (S \times Val \times S \times Val \to Prop)$, such that

- The initial state of an object is s_0;
- For object obj of state s and request x, if $vr(x)$, then performing $\texttt{op}(obj, x)$ returns y and updates the object to state s' such that $rr(s, x, s', y)$.

The valid request predicate vr restricts which arguments to op are considered well-formed or valid. Given a valid argument, the behavior of op is captured by a request-response relation rr. This relation may be non-deterministic. In case there are multiple sequential behaviors in question, we write $\mu.vr$ or $\mu.rr$ to disambiguate.

Example 2.3 (Bag). A bag, *i.e.*, a multiset, is a container that supports push and pop operations. The push operation adds a new element to the container, while pop removes an arbitrary element from the container and returns this element, or reports that the container is empty. The sequential behavior of a bag, μ_{bag}, is defined as

$$\begin{aligned} s_0 &\triangleq \varnothing \\ vr(x) &\triangleq (\exists v.\, x = \texttt{some}(v)) \vee x = \texttt{none} \\ rr(s, x, s', y) &\triangleq (\exists v.\, x = \texttt{some}(v) \wedge s' = \{\!|v|\!\} \uplus s \wedge y = ()) \\ &\quad \vee (x = \texttt{none} \wedge s = \varnothing \wedge s' = \varnothing \wedge y = \texttt{none}) \\ &\quad \vee (x = \texttt{none} \wedge \exists v.\, v \in s \wedge s' = s \setminus \{\!|v|\!\} \wedge y = \texttt{some}(v)) \end{aligned}$$

where $\texttt{none}$ and $\texttt{some}(v)$ are syntactic sugar for $\texttt{inl}(())$ and $\texttt{inr}(v)$, respectively. This definition encodes $\texttt{push}(b, v)$ as $\texttt{op}(b, \texttt{some}(v))$ and $\texttt{pop}(b)$ as $\texttt{op}(b, \texttt{none})$. For the return value of a pop, $\texttt{none}$ means the bag is empty, and $\texttt{some}\, v$ means element v is removed from the bag.

Linearizability of a Trace. We first define what it means for a single interaction of a data structure with a client to be linearizable. An interaction is represented by a call-return trace, and a linearization of such a trace is obtained by adding suitable linearization point events to the trace.

Formally, we define Σ to be the type of events that can occur in physical traces, and let Σ_{lin} be an extension that includes linearization events.

$$\begin{aligned} \Sigma &\triangleq \{(\tau, \mathsf{Call}(v)) \mid \tau \in Tag, v \in Val\} \cup \{(\tau, \mathsf{Ret}(v')) \mid \tau \in Tag, v' \in Val\} \\ \Sigma_{\mathsf{lin}} &\triangleq \Sigma \cup \{(\tau, \mathsf{Lin}(v, v')) \mid \tau \in Tag, v, v' \in Val\} \end{aligned}$$

Here, τ is a tag that relates the call, linearization, and return event of an operation. The parameters of call and return events represent the argument to the operation and the returned value, respectively. Linearization events take two parameters that match the argument and return value for the call/return that they correspond to. A *physical trace* $\vec{t} \in \Sigma^*$ is then a finite sequence of call and return events, while an *annotated trace* $\vec{t}' \in \Sigma_{\mathsf{lin}}^*$ is a finite sequence that may also include linearization events. Both types of traces include the empty sequence ϵ. We define various projection operations on traces that return the subtrace consisting of certain selected events. Given a trace $\vec{u}$, $\pi_\tau(\vec{u})$ is the subtrace of events with tag τ, $\pi_\Sigma(\vec{u})$ is the subtrace of physical events, and $\pi_{\mathsf{lin}}(\vec{u})$ is the subtrace of linearization events. A trace is well-formed if the call, linearization, and return events of each tag occur in order, and each event only occurs once.

$$\mathsf{well\text{-}formed}(\vec{t}) \triangleq \forall \tau.\, \exists v, v'.\, \pi_\tau(\vec{t}) \sqsubseteq_{\mathsf{pre}} [(\tau, \mathsf{Call}(v)); (\tau, \mathsf{Ret}(v'))] \tag{1}$$

$$\mathsf{well\text{-}formed}_{\mathsf{lin}}(\vec{t'}) \triangleq \forall \tau.\, \exists v, v'.\, \pi_\tau(\vec{t'}) \sqsubseteq_{\mathsf{pre}} [(\tau, \mathsf{Call}(v)); (\tau, \mathsf{Lin}(v, v')); (\tau, \mathsf{Ret}(v'))] \tag{2}$$

where $\vec{x} \sqsubseteq_{\mathsf{pre}} \vec{y}$ means $\vec{x}$ is a prefix of $\vec{y}$.

Definition 2.4 (Trace Linearizability). A trace $\vec{t} \in \Sigma^*$ is linearizable *w.r.t.* a sequential behavior μ, written $\mathsf{trlin}_\mu(\vec{t})$, if the parameter of every call event in $\vec{t}$ satisfies vr and there exists an annotated trace $\vec{t'}$ and state s' such that $\vec{t} = \pi_\Sigma(\vec{t'})$, $\mathsf{well\text{-}formed}_{\mathsf{lin}}(\vec{t'})$, and $\mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t'}), s_0, s')$, where

$$\mathsf{sound}_\mu(\epsilon, s, s') \triangleq s = s'$$
$$\mathsf{sound}_\mu(\vec{t'} +\!\!+ [(\tau, \mathsf{Lin}(x, y))], s, s'') \triangleq \exists s'.\, \mathsf{sound}_\mu(\vec{t'}, s, s') \wedge vr(x) \wedge rr(s', x, s'', y)$$

Here, the annotated trace $\vec{t'}$ is a linearization of $\vec{t}$ because $\vec{t'}$ consists of the same physical events as $\vec{t}$, and the linearization event of each operation occurs between its call event and return event. Moreover, $\mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t'}), s_0, s')$ says starting from the initial state s_0, processing operations according to the linearization events in the order found in $\vec{t'}$ will transform the object to state s', *i.e.*, the linearization matches the sequential behavior μ.

In Definition 2.4, a "sound" non-empty trace is obtained by appending a new linearization event to the end of another sound trace. Alternatively, one can also choose to prepend an event to the beginning of another sound trace.

Remark 2.5. $\mathsf{sound}_\mu((\tau, \mathsf{Lin}(x, y)) :: \vec{t'}, s, s'') \Leftrightarrow \exists s'.\, vr(x) \wedge rr(s, x, s', y) \wedge \mathsf{sound}_\mu(\vec{t'}, s', s'')$

Linearizability of a Data Structure. With trace linearizability defined, we now define linearizability of a data structure by requiring that the traces generated by interactions with the data structure are linearizable. However, we must restrict our attention to *valid* traces, in which all arguments to `op` are valid. In addition, the client must only interact with objects via calls to `op`. For example, if an object has internal private state, the client is not allowed to break the data structure's abstractions and directly modify this private state. Aside from these restrictions, however, a client can send any possible valid sequence of requests to the object and spawn an unbounded number of threads to send requests to the object concurrently. To represent this set of traces, we define a new reduction relation that models these interaction rules. This relation is defined over *configurations* that track the operations that are executing.

Definition 2.6 (Configuration). Given an object *obj* of a data structure (`init`, `op`), a configuration ρ is a triple of $(\vec{\pi}, \vec{e}, \sigma)$: $(Tag \times Expr^?)^* \times Expr^* \times State$, where

- $\vec{\pi}$ is a list representing the executing operations on the object. Each element of the list is a pair of tag τ and expression e. For operations that have finished execution and returned, the expression component of the pair is a special value $\perp$.
- $\vec{e}$ is a list of other threads executing in the system that were created by the executing operations in $\vec{\pi}$.
- σ is the state of the program, *e.g.*, the heap. The state of the data structure is part of σ.

Then, a step of interaction between an object and a client is modeled by a *most adversarial step* relation between configurations.

Definition 2.7 (Most Adversarial Step). Given an object *obj*, method op, and predicate *vr*, a *most adversarial step* $\rho \xrightarrow[obj.\mathsf{op}/vr]{\vec{t}}_{\mathsf{adv}} \rho'$ is defined by the following set of inference rules:

Adv-Call
$$(\vec{\pi}, \vec{e}, \sigma) \xrightarrow[obj.\mathsf{op}/vr]{[(\tau, \mathsf{Call}(v))]}_{\mathsf{adv}} (\vec{\pi} +\!\!+ [(\tau, \mathsf{op}(obj, v))], \vec{e}, \sigma) \quad \text{if } \tau \notin \vec{\pi}.1 \text{ and } vr(v)$$

Adv-Interm
$$(\vec{\pi}_1 +\!\!+ (\tau, e_1) :: \vec{\pi}_2, \vec{e}, \sigma_1) \xrightarrow[obj.\mathsf{op}/vr]{\epsilon}_{\mathsf{adv}} (\vec{\pi}_1 +\!\!+ (\tau, e_2) :: \vec{\pi}_2, \vec{e} +\!\!+ \vec{e}_f, \sigma_2) \quad \text{if } (e_1, \sigma_1) \rightarrow_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)$$

Adv-Child
$$(\vec{\pi}, \vec{e}_a +\!\!+ e_1 :: \vec{e}_b, \sigma_1) \xrightarrow[obj.\mathsf{op}/vr]{\epsilon}_{\mathsf{adv}} (\vec{\pi}, \vec{e}_a +\!\!+ e_2 :: \vec{e}_b +\!\!+ \vec{e}_f, \sigma_2) \quad \text{if } (e_1, \sigma_1) \rightarrow_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)$$

Adv-Ret
$$(\vec{\pi}_1 +\!\!+ (\tau, v) :: \vec{\pi}_2, \vec{e}, \sigma) \xrightarrow[obj.\mathsf{op}/vr]{[(\tau, \mathsf{Ret}(v))]}_{\mathsf{adv}} (\vec{\pi}_1 +\!\!+ (\tau, \bot) :: \vec{\pi}_2, \vec{e}, \sigma) \quad \text{if } v \in Val$$

Adv-Call represents the invocation of a new operation with argument v, which must be valid according to the *vr* predicate. The transition is labeled with a call event with a fresh tag τ, and an expression executing $\mathsf{op}(obj, v)$ is added to the list $\vec{\pi}$ with the same tag τ. Adv-Interm performs a step of execution of an operation e in the list $\vec{\pi}$, adding any forked child threads to the $\vec{e}$ component of the configuration. Adv-Child executes a step of reduction of one of these child threads. Finally, Adv-Ret selects an operation that has terminated with a value v and labels the transition with a return event, replacing the value v with $\bot$ in the configuration afterwards.

The non-deterministic nature of these rules represents the ability of a client to invoke an arbitrary number of concurrent operations, so long as the arguments are valid, in the sense that they satisfy *vr*. These invocations are interleaved with steps of the operations themselves, as well as any children they produce. Critically, the non-deterministic steps by the environment do not alter the local state of the data structure.

The relation $\rightarrow^*_{\mathsf{adv}}$ is the reflexive transitive closure of $\rightarrow_{\mathsf{adv}}$, which concatenates the labels on transitions.

Advs-Nil
$$\rho \xrightarrow[obj.\mathsf{op}/vr]{\epsilon}{}^*_{\mathsf{adv}} \rho$$

Advs-Cons
$$\rho_0 \xrightarrow[obj.\mathsf{op}/vr]{\vec{t}_1}_{\mathsf{adv}} \rho_1 \wedge \rho_1 \xrightarrow[obj.\mathsf{op}/vr]{\vec{t}_2}{}^*_{\mathsf{adv}} \rho_2 \implies \rho_0 \xrightarrow[obj.\mathsf{op}/vr]{\vec{t}_1 +\!\!+ \vec{t}_2}{}^*_{\mathsf{adv}} \rho_2$$

We omit *obj*, op, and *vr*, when they are clear from the context, and simply write $\rho_1 \xrightarrow{\vec{t}}{}^*_{\mathsf{adv}} \rho_2$.

Using the most adversarial step relation, we can now define linearizability of a data structure.

Definition 2.8 (Linearizability). A data structure (`init`, op) is linearizable *w.r.t.* a sequential behavior μ, denoted as $\mathsf{lin}_\mu(\texttt{init}, \mathsf{op})$, if

(1) $\mathsf{safe}(\texttt{init}(), \emptyset)$, *i.e.*, the evaluation of `init()` never gets stuck; and
(2) if $([\texttt{init}()], \emptyset) \rightarrow^*_{\mathsf{tp}} ([obj], \sigma)$ and $(\epsilon, \epsilon, \sigma) \xrightarrow[obj.\mathsf{op}/vr]{\vec{t}}{}^*_{\mathsf{adv}} (\vec{\pi}, \vec{e}, \sigma')$, then all expressions in $\vec{\pi}$ and $\vec{e}$ are not stuck, and $\mathsf{trlin}_\mu(\vec{t})$.

The second part of this definition runs `init` under the normal reduction relation $\rightarrow^*_{\mathsf{tp}}$ until it terminates in some value *obj*, which is then used as part of the most adversarial step reduction. Every trace $\vec{t}$ generated in this manner must be linearizable. In addition, we require that execution of `init` and op never gets stuck, in order to rule out implementations that are unsafe and trigger some form of undefined behavior.

$$P, Q \in \mathit{iProp} ::= \ulcorner\varphi\urcorner \mid P \wedge Q \mid P \vee Q \mid \forall x.\, P \mid \exists x.\, P \mid \ell \mapsto v \qquad (\varphi \text{ is a meta-level proposition})$$
$$\mid P * Q \mid P \mathrel{-\!\!*} Q \mid \triangleright P \mid \{P\}\, e\, \{v.\, Q\} \mid \Rightarrow\!\!\Vdash P \mid \boxed{\bullet M}^{\gamma} \mid k \hookrightarrow^{\gamma} v \mid \cdots$$

HtConseq
$$\frac{\{P\}\, e\, \{\Phi\} \qquad P' \vdash P \qquad \forall x.\, (\Phi(x) \vdash \Phi'(x))}{\vdash \{P'\}\, e\, \{\Phi'\}}$$

HtFrame
$$\frac{\{P\}\, e\, \{v.\, Q\}}{\{P * R\}\, e\, \{v.\, Q * R\}}*$$

HtValue
$$\frac{\Phi(v)}{\{P\}\, v\, \{\Phi\}}*$$

HtSeq
$$\frac{\{P\}\, e\, \{v.\, Q\} \qquad \forall v.\, \{Q\}\, K[v]\, \{\Phi\}}{\{P\}\, K[e]\, \{\Phi\}}*$$

HtPure
$$\frac{e_1 \to_{\text{pure}} e_2}{\{P\}\, e_2\, \{\Phi\} \vdash \{\triangleright P\}\, e_1\, \{\Phi\}}$$

HtAlloc
$$\{\text{True}\}\ \texttt{ref}\, v\ \{\ell.\, \ell \mapsto v\}$$

HtLoad
$$\{\ell \mapsto v\}\ !\ell\ \{x.\, x = v * \ell \mapsto v\}$$

HtStore
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{x.\, x = () * \ell \mapsto w\}$$

HtFupd
$$\frac{\{P\}\, e\, \{\Phi\}}{\{\Rightarrow\!\!\Vdash P\}\, e\, \{\Phi\}}*$$

MapAlloc
$$\vdash \exists \gamma.\, \Rightarrow\!\!\Vdash \boxed{\bullet\varnothing}^{\gamma}$$

MapLookup
$$\boxed{\bullet M}^{\gamma} * k \hookrightarrow^{\gamma} v \vdash \ulcorner M(k) = v\urcorner$$

MapInsert
$$\frac{k \notin \mathrm{dom}(M)}{\boxed{\bullet M}^{\gamma} \vdash \Rightarrow\!\!\Vdash \boxed{\bullet(M[k \leftarrow v])}^{\gamma} * k \hookrightarrow^{\gamma} v}$$

MapUpdate
$$\boxed{\bullet M}^{\gamma} * k \hookrightarrow^{\gamma} v \vdash \Rightarrow\!\!\Vdash \boxed{\bullet(M[k \leftarrow w])}^{\gamma} * k \hookrightarrow^{\gamma} w$$

MapDelete
$$\boxed{\bullet M}^{\gamma} * k \hookrightarrow^{\gamma} v \vdash \Rightarrow\!\!\Vdash \boxed{\bullet(M \setminus \{k\})}^{\gamma}$$

HigherOrderMapLookup
$$\boxed{\bullet M}^{\gamma} * k \hookrightarrow^{\gamma} P \vdash \exists P'.\, \ulcorner M(k) = P\urcorner * \triangleright(P \equiv P')$$

Fig. 3. A Separation Logic for HeapLang.

2.3 An Iris-Based Separation Logic

Figure 3 shows a subset of rules from the default separation logic for HeapLang provided by Iris.[1] This logic can be used to derive judgments of the shape $P \vdash Q$, where P and Q are separation logic assertions. Separation logic enriches propositional logic with the separating conjunction $*$ and separating implication $\mathrel{-\!\!*}$, sometimes called the "magic wand". The separating conjunction $P * Q$ expresses ownership of P and Q separately, *i.e.*, P and Q hold for disjoint parts of the heap. In particular, $P \vdash P * P$ does not always hold. Iris is an *affine* separation logic that admits the weakening rule $P * Q \vdash P$. Assertion $\ulcorner\varphi\urcorner$ embeds a meta-level proposition (*i.e.*, a Rocq `Prop`) into the separation logic. The points-to assertion $\ell \mapsto v$ says that location ℓ stores the value v. We omit most of the rules for proving entailments; see Chapter 3 of Birkedal and Bizjak [4] for an expository introduction to separation logic and Iris.

To reason about programs, we use the Hoare triple $\{P\}\, e\, \{v.\, Q\}$. Here, v is a binder for the value returned by executing the expression e. We omit this binder if v does not appear free in Q. When writing inference rules, we use two different styles. When the line dividing the premises from the conclusion is a standard horizontal bar, then this rule should be read as an implication in the meta-logic. When the horizontal bar is replaced by a $\mathrel{-\!\!*}$, we use a rule with premises $P_1, \ldots, P_n$ and conclusion Q as notation for the separation logic entailment $(P_1 * \cdots * P_n) \vdash Q$.

[1] As usual for Iris, the actual logic we work on uses another connective wp $e\, \{\Phi\}$ and the Hoare triple is syntactic sugar for a certain form of wp.

Most of the Hoare triple rules in Figure 3 are standard (ignore the later modality "▷" for now). HtConseq is the rule of consequence from Hoare logic. HtFrame allows for local reasoning about the subset of state an expression accesses. HtSeq generalizes the standard sequential composition rule of Hoare logic to an arbitrary evaluation context K. The rules HtAlloc, HtLoad, HtStore reason about operations on mutable state by updating the points-to assertion appropriately. Finally, HtPure allows for reasoning about pure reduction steps of the program, which do not require any state assertions.

Ghost State and Updates. To reason about the physical state of a program, it is often useful to maintain some additional *auxiliary* or *ghost* states to track some abstract aspects of the program [58]. These ghost states can be updated by the prover using certain rules.

While Iris supports a large variety of ghost state [40], this paper mainly uses (higher-order) ghost maps. A ghost map comes with two types of resources: a full or authoritative resource $\boxed{\bullet M}^{\gamma}$ carrying the knowledge of the whole map and a fragmentary resource $k \hookrightarrow^{\gamma} v$ carrying the value v at key k. The fragmentary resource is also known as *ghost points-to*. Here, γ is the name of the ghost map. Rule MapAlloc allocates an empty ghost map with some fresh name. In this rule, "$\Rightarrow\!\!\!\Rightarrow$" is called the update modality, meaning that something is provable by updating the underlying ghost state. The update modality can be eliminated using HtFupd. There are two lookup rules for ghost maps: if the ghost map stores first-order values like $\mathbb{N}$ or *Expr*, then when owning both $\boxed{\bullet M}^{\gamma}$ and $k \hookrightarrow^{\gamma} v$, one immediately learns that $M(k)$ is equal to v by MapLookup; but if the ghost map stores higher-order data like *iProp*, one can only use HigherOrderMapLookup. This rule still concludes that there must exist some (higher-order) value P' at $M(k)$, but P' is only equivalent to P under a later modality "▷". Intuitively, $\triangleright P$ means P does not hold immediately, but will hold after the program executes one step. This also explains why there is a later modality in the precondition of HtPure—the later modality captures the fact that the program has taken one step.

Soundness and Completeness. The HeapLang program logic is sound and complete.

Theorem 2.9 (Soundness [40] and Completeness [36]).

$$\vdash \{\mathsf{True}\}\ e\ \{v.\ \ulcorner \varphi(v) \urcorner\} \iff \mathsf{safe}_{\lambda v, \_.\ \varphi(v)}(e, \varnothing)$$

Note that Theorem 2.9 only deals with closed programs. It does not necessarily say anything about our ability to prove specifications for individual operations or functions of a library, or whether we can prove specifications that are useful for clients. In contrast, specifications like linearizability are about a data structure that is intended to be used as part of a larger client application. As we will see in the next section, our results instead provide a completeness theorem that has to do with reusable specifications about a library's operations.

3 Logical Atomicity via Linearizability

Although linearizability is a standard correctness condition for concurrent data structures, in many concurrent separation logics it is common to instead use an alternate form of specification based on *logical atomicity*. Whereas linearizability captures atomicity by relating traces to sequential specifications, logical atomicity instead characterizes atomicity in terms of what logical reasoning rules an atomic operation should support. In particular, by asking what special reasoning rules the logic provides for primitive physically atomic operations, and then proving similar rules for the operations of a concurrent data structure, we allow clients of the data structure to reason *as if* these operations were physically atomic. And, what is special about physically atomic operations in many CSLs are the rules for interacting with *invariant assertions*. Thus, logically atomic specifications establish similar rules for working with invariants.

In particular, in Iris, the invariant assertions have the form $\boxed{R}$ which says that assertion R is an invariant, meaning that it holds before and after each step of execution. Unlike regular assertions in a separation logic, the assertion $\boxed{R}$ is *persistent*, meaning that it can be duplicated and shared by multiple threads, *i.e.*, $\boxed{R} \vdash \boxed{R} * \boxed{R}$. Atomicity is captured by the following invariant access rule.[2]

$$\frac{\textsc{InvAccPhysical} \quad \vdash \{P * \triangleright R\}\ e\ \{v.\ Q * \triangleright R\} \qquad \mathsf{atomic}(e)}{\boxed{R} \vdash \{P\}\ e\ \{v.\ Q\}}$$

To prove a specification under an invariant $\boxed{R}$, the rule says that we can open this invariant and add R to the precondition, as long as we reestablish R in the postcondition. The side condition $\mathsf{atomic}(e)$ means e is physically atomic, *i.e.*, in the operational semantics, e reduces to a value in a single execution step. Because e reduces to a value in a single step, this rule preserves the property that the invariant R must hold before and after each step. The later modality "$\triangleright$" guarding R is needed to make nesting of impredicative invariants sound. For so-called *timeless* assertions that do not involve impredicative nesting, this later modality can be removed. Although our proof uses an impredicative invariant, we only need to access its impredicative part after the program has taken one step, so this later modality is not a problem for our proof and we will ignore it afterwards.

For a logically atomic operation, the idea is to similarly allow for an invariant to be opened around the atomic point at which the operation takes effect. Following a style pioneered by the TaDA [12] logic, Iris supports such specifications through an alternate version of the Hoare triple called a *logically atomic triple*, $\langle P \rangle\ e\ \langle v.\ Q \rangle$. The most important rules are shown below.[3]

$$\frac{\textsc{InvAccLogical} \quad \vdash \langle P * \triangleright R \rangle\ e\ \langle v.\ Q * \triangleright R \rangle}{\boxed{R} \vdash \langle P \rangle\ e\ \langle v.\ Q \rangle} \qquad \frac{\textsc{LaIntro} \quad \vdash \forall \Phi.\ \left\{\mathsf{AU}_{P,Q}(\Phi)\right\}\ e\ \{\Phi\}}{\vdash \langle P \rangle\ e\ \langle v.\ Q \rangle} \qquad \frac{\textsc{AuFire} \quad \vdash \{P\}\ e_a\ \{Q\} \qquad \mathsf{atomic}(e_a)}{\vdash \left\{\mathsf{AU}_{P,Q}(\Phi)\right\}\ e_a\ \{\Phi\}}$$

Crucially, the logically atomic triple also provides an invariant access rule InvAccLogical, but notice here that e is *not* necessarily physically atomic. To initially prove a logically atomic triple about a data structure's operations, we use LaIntro to reduce it to a Hoare triple with an *atomic update* assertion AU as precondition and an arbitrary postcondition Φ. Because Φ is universally quantified, the only way for the data structure to obtain the postcondition Φ is to use AuFire to "fire" the $\mathsf{AU}_{P,Q}(\Phi)$. The point at which this rule is applied corresponds to the linearization point of the operation. At the linearization point, the data structure first obtains P from the AU, and it must then prove Q immediately after executing one physical step.

The goal of this paper is to relate such logically atomic specifications to linearizability. In particular, we will show that the following combination of (3) a Hoare triple about an object's initialization, and (4) a logically atomic triple about the object's operations, is equivalent to linearizability:

$$\mathsf{atom\text{-}spec}_{\mu}(\mathsf{IsP}, \mathsf{PState}, \texttt{init}, \texttt{op}) \triangleq$$

$$\vdash \{\mathsf{True}\}\ \texttt{init}()\ \{obj.\ \exists \gamma.\ \mathsf{IsP}(\gamma, obj) * \mathsf{PState}(\gamma, s_0)\} \tag{3}$$

$$\ulcorner vr(x) \urcorner * \mathsf{IsP}(\gamma, obj) \vdash \langle \mathsf{PState}(\gamma, s) \rangle\ \texttt{op}(obj, x)\ \langle y.\ \exists s'.\ \mathsf{PState}(\gamma, s') * \ulcorner rr(s, x, s', y) \urcorner \rangle \tag{4}$$

$$\forall \gamma, obj.\ \mathsf{persistent}(\mathsf{IsP}(\gamma, obj)) \tag{5}$$

This specification is parameterized by μ, *i.e.*, a sequential specification for the object, and two predicates, IsP and PState. The IsP assertion is a persistent assertion that states that an object is a valid instance of this class of objects, *e.g.*, satisfies some internal invariant of the data structure;

[2]The full version of this rule annotates invariant assertions with a *namespace* identifier, which is used to prevent opening the same invariant twice. These matters are orthogonal concerns and we omit these annotations here.

[3]Like the HeapLang logic, the actual rules about the logically atomic triple in Iris are also stated in terms of wp.

PState describes the current abstract state of the object. These assertions are connected by a *ghost name* γ, which ensures both assertions talk about the same object. The specification for `init` returns ownership of both of these assertions about the newly created object. Then, the specification for `op` requires that the request and the object must both be valid, and that `op` atomically updates the object from state s to state s' and returns y in accord with the rr relation.

Birkedal et al. [5] prove that such a specification implies linearizability.

Theorem 3.1 (Logical Atomicity ⇒ Linearizability [5]).

$$\text{atom-spec}_\mu(\text{IsP}, \text{PState}, \texttt{init}, \texttt{op}) \implies \text{lin}_\mu(\texttt{init}, \texttt{op})$$

We call this result a soundness theorem for logical atomicity because it connects the logical notion to the external linearizability property.[4] In this paper, we prove a result that is essentially the converse of this soundness theorem, *i.e.*, a completeness theorem for logical atomicity. The direct converse looks as follows.

Proposition 3.2. $\text{lin}_\mu(\texttt{init}, \texttt{op}) \implies \exists \text{IsP}, \text{PState}.\ \text{atom-spec}_\mu(\text{IsP}, \text{PState}, \texttt{init}, \texttt{op})$

As we will see, our proof strategy requires modifying this statement slightly, for reasons we explain next.

3.1 Proof Strategy

The difficult part of the proof is in deriving the logically atomic triple about `op`. To do so, we apply LaIntro, which reduces the goal to deriving a Hoare triple about the function. Because of the assumption that the data structure is linearizable, we know that the `op` function is safe. That is, it never gets stuck. Thus, we can use the proof rules of Iris to reason step by step as the operation executes. However, at some point we must fire the atomic update introduced by LaIntro. This point must correspond to what is essentially a linearization point of the operation. Since, by assumption, the data structure is linearizable, given a trace of call and return operations, this linearizability assumption will tell us where we can insert such linearization points, and therefore fire the atomic update. To use this assumption, we create an Iris invariant that stores an annotated trace of all of the call and return operations that have been executed so far. When we reach a point that is a linearization point in the annotated trace, we know that we can therefore fire the atomic update. However, there are two complications to this basic idea that pose challenges:

- C_1 **Future-dependent linearization points**. While proving a logically atomic triple, one only knows the historic trace $\vec{t}_h$ up to the current point of execution. From this partial historic trace, we can obtain a linearized trace $\vec{t}'_h$ that adds the linearization point annotations to $\vec{t}_h$. Suppose we fire atomic updates as we go according to this extension $\vec{t}'_h$. After a new call or return event u, the trace will be extended by u to become $\vec{t} = \vec{t}_h +\!\!+ u$. By linearizability, we can obtain another linearization $\vec{t}'$ of the new trace. Although $\vec{t}$ is obtained by appending a new event to $\vec{t}_h$, the definition of linearizability *does not* guarantee that $\vec{t}'$ can also be obtained by adding some new events to $\vec{t}'_h$. In fact, $\vec{t}'$ can choose completely different linearization points from $\vec{t}'_h$. This phenomenon is well-known as future-dependent linearization, and was observed in the very first work that proposed linearizability [33]. The problem is that if $\vec{t}'$ uses different linearization points than $\vec{t}'_h$, we would have to amend the AU's we have already fired, which is not possible.
- C_2 **External linearization points**. Proving a logically atomic triple requires firing the AU at a specific program step, *i.e.*, one must find a particular physically atomic operation to fire the AU, while the linearization points extracted from applying the definition of linearizability

[4] This theorem is a consequence of a more general result which we discuss further in §5.1.

$$\mathit{wrapi}(\texttt{init})() \triangleq (\texttt{init}(), \texttt{newproph})$$

$$\begin{aligned}&\mathit{wrapo}(\texttt{op})(\mathit{objp}, x) \triangleq\\ &\quad \texttt{let}\ (\mathit{obj}, p) := \mathit{objp}\ \texttt{in}\\ &\quad \texttt{let}\ \tau := \texttt{newghostID}\ \texttt{in}\ \texttt{resolve}\ p\ \texttt{to}\ \texttt{inl}(\tau, x);\\ &\quad \texttt{let}\ y := \texttt{op}(\mathit{obj}, x)\ \texttt{in}\ \texttt{resolve}\ p\ \texttt{to}\ \texttt{inr}(\tau, y); y\end{aligned}$$

Fig. 4. Wrapped Data Structure Methods.

are just some Lin events in an annotated trace. The definition of linearizability does not tell us specific physical steps tied to these Lin events. This means that to use the linearization points extracted from trlin, we must find a way to map these events to some physical steps.

Challenge 1 is solved by using *prophecy variables* [1, 41, 78]. Prophecy variables allow for "predicting" what the outcomes of future program steps will be during the course of a proof. Using prophecy variables, one can prophesy the complete trace of operations on an object at the time the object is created. Thus, we can extract all linearization points once and for all from this complete trace, and do not have to worry about the linearization points shifting when extending partial traces with new events.

Challenge 2 is solved by a technique called *helping* [9, 31, 65]. Rather than trying to fire the atomic update for an operation at a specific physical step, we instead store AU assertions for all operations that are in the middle of execution in a shared invariant. When an operation is called and emits a Call event, we add its AU to the invariant. Linearization itself happens at the Call and Ret events of operations: whenever an operation emits a Call or Ret event, we gather the AU assertions for all pending operations whose Lin events occur in the annotated trace just prior to this event, extract them from the invariant, and fire all of them. We then store the corresponding Φ assertions resulting from each back inside the invariant, so that they can later be collected by the corresponding operation. This means one physical step, *i.e.*, the step on which an operation emits its Call or Ret event, can linearize multiple operations, not limited to the AU of the operation taking the step. Thus, an operation logically "helps" others perform their AU.

Readers familiar with carrying out proofs of logical atomicity in Iris will not be surprised that we end up using prophecy variables and helping in our proof. Indeed, these techniques are familiar and often used in proving the logical atomicity of specific data structures. This pattern of transferring atomic updates for helping has been used from the very beginning in Iris logical atomicity proofs, and prophecy variables were specifically added to Iris in order to reason about future-dependent linearizability. What makes our results novel is the generality we work at, and the fact that we show that these two techniques suffice to do *all* such logical atomicity proofs.

3.2 Data Structure Wrapper

In order to use prophecy variables to "predict" the future stream of call and return operations, we need to annotate the operations with a wrapper that attaches the prophecy variables to the events we want to predict. In Iris, prophecy variables are implemented by two special "virtual" operations: `newproph` and `resolve`. These operations are virtual in the sense that the behavior of the program will not change after erasing them. The specifications for working with these primitives are as

follows.

ProphNe
$\mathsf{proph}(p_1, \vec{v}_1) * \mathsf{proph}(p_2, \vec{v}_2) \vdash \ulcorner p_1 \neq p_2 \urcorner$

HtNewProph
$\{\mathsf{True}\}\ \texttt{newproph}\ \{p.\ \exists \vec{v}\,.\ \mathsf{proph}(p, \vec{v})\}$

HtResolve
$\{\mathsf{proph}(p, \vec{v})\}\ \texttt{resolve}\ p\ \texttt{to}\ w\ \{().\ \exists \vec{v}\,'.\ \ulcorner \vec{v} = w :: \vec{v}\,' \urcorner * \mathsf{proph}(p, \vec{v}\,')\}$

Operation **newproph** allocates a fresh prophecy variable p. Resource $\mathsf{proph}(p, \vec{v})$ in its postcondition says that the resolutions on p in the future execution of the program will be the values in the list $\vec{v}$. It is an exclusive resource ensuring the thread currently owning $\mathsf{proph}(p, \vec{v})$ is the only thread allowed to resolve p. Under precondition $\mathsf{proph}(p, \vec{v})$, primitive **resolve** p **to** w *resolves* prophecy variable p to value w. The resolution guarantees that the first element in $\vec{v}$ must be w, and removes this element from the future resolutions list.

Using these operations, we define wrappers for data structures as shown in Figure 4. The wrapped `init` function associates a prophecy variable with the newly created object. The wrapped `op` function allocates a fresh tag for this invocation of the operation using a helper function **newghostID**, and resolves the associated prophecy variable before `op` is called and after `op` returns. These two resolutions capture the call and return events of the invocation, where the event $(\tau, \mathsf{Call}(x))$ is encoded as $\texttt{inl}(\tau, x)$, and event $(\tau, \mathsf{Ret}(y))$ is encoded as $\texttt{inr}(\tau, y)$.

To generate the fresh tag with **newghostID**, we can in fact reuse the prophecy variable mechanism, by treating the prophecy variable's identifier as the tag. Since $\mathsf{proph}(p, \vec{v})$ is exclusive, we can discard the $\vec{v}$ part and just use the prophecy variable p itself as a unique ghost token. Thus, we define **newghostID** as follows, and derive the following helper specifications:

$$\texttt{newghostID} \triangleq \texttt{newproph} \qquad \mathsf{token}\,(\tau) \triangleq \exists \vec{v}\,.\ \mathsf{proph}(\tau, \vec{v})$$

HtNewGhostID
$\{\mathsf{True}\}\ \texttt{newghostID}\ \{\tau.\ \mathsf{token}\,(\tau)\}$

TokenNe
$\mathsf{token}\,(\tau_1) * \mathsf{token}\,(\tau_2) \vdash \ulcorner \tau_1 \neq \tau_2 \urcorner$

Because we resolve the call and return events using the prophecy variable, this means that when we first create the prophecy variable during the wrapped initialization, the list of values $\vec{v}$ in the $\mathsf{proph}(p, \vec{v})$ permission will be the entire trace of events on the data structure. Thus, we can define a derived predicate, $\mathsf{prophTrace}_{obj}(\rho, p, \vec{t})$ that asserts ownership of the prophecy variable p and says that starting from configuration ρ, the decoded list of prophesied values is a call/return trace $\vec{t}$ generated by a valid interaction between the data structure and a client, *i.e.*, $\exists \rho'.\ \rho \xrightarrow{\vec{t}}{}^{*}_{\mathsf{adv}} \rho'$. Predicate prophTrace is encoded using a technique inspired by typed prophecy variables [41].[5] The full definition of prophTrace is included in Appendix A.1.

3.3 Invariant

As mentioned above, the proof needs an invariant that relates the abstract state of the object to the physical state, and tracks the correspondence between the trace prophecy and what atomic updates have been completed. Specifically, for an object *obj* associated with the ghost resource name γ, we use an invariant $\boxed{I_{\mathsf{atom}_\mu}(\gamma, obj, p)}$. Here p is the prophecy variable used to speculate the future trace of *obj*.

At a high level, the invariant relates four things: the current configuration $\rho = (\vec{\pi}, \vec{e}, \sigma)$ of the object and pending operations, the abstract sequential specification state s of *obj*, a store Φs of the atomic updates and receipts of the operations currently in flight, and a prophecy $\vec{t}\,'$ of the object's future trace. The future trace predicts where each pending operation will linearize so that the

[5] The encoding requires the use of the law of excluded middle because proposition $\exists \rho'.\ \rho \xrightarrow{\vec{t}}{}^{*}_{\mathsf{adv}} \rho'$ is generally undecidable [61].

$$\exists \gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s, \vec{\pi}, \vec{e}, \sigma, \Phi s, s, \vec{t}'.\ \text{let } \rho := (\vec{\pi}, \vec{e}, \sigma) \text{ in } \quad \boxed{\boxed{\square(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma}}$$

$$* \boxed{\bullet \mathit{l2m}(\mathit{activeOp}(\vec{\pi}))}^{\gamma_\pi} * \boxed{\bullet \vec{e}}^{\gamma_e} * \mathsf{S}_\circ(\sigma) * \boxed{\bullet [0 \leftarrow s]}^{\gamma_s} * \mathsf{prophTrace}_{obj}(\rho, p, \pi_\Sigma(\vec{t}')) \tag{P_1}$$

$$* \boxed{\bullet \Phi s}^{\gamma_\phi} * \left(\mathop{\Large *}_{\tau \leftarrow \Phi \in \Phi s} \mathit{interpret}(\vec{t}', \tau, \Phi) \right) \tag{P_2}$$

$$* \left(\mathop{\Large *}_{\tau \in \vec{\pi}.1} \mathsf{token}\,(\tau) \right) \tag{P_3}$$

$$* \ulcorner \mathit{pendingOp}(\vec{t}') \subseteq \mathsf{dom}(\Phi s) = \{\vec{\pi}.1\} \urcorner \tag{P_4}$$

$$* \ulcorner \mathsf{partial\text{-}well\text{-}formed}_{\mathsf{lin}}(\vec{t}') \wedge (\exists s'.\ \mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t}'), s, s')) \wedge \mathsf{cfg\text{-}safe}(\vec{\pi}, \vec{e}, \sigma) \urcorner \tag{P_5}$$

where $\mathit{interpret}(\vec{t}', \tau, \Phi) \triangleq$ match $\pi_\tau(\vec{t}')$ with

$$\begin{array}{ll} \mid (\_, \mathsf{Call}(\_)) :: \_ & \Rightarrow \mathsf{False} \\ \mid (\_, \mathsf{Lin}(x, \_)) :: \_ & \Rightarrow \mathsf{AU}_{\mathsf{PState}_\mu(\gamma,s),\, y.\ \exists s'.\ \mathsf{PState}_\mu(\gamma,s') * \ulcorner rr(s,x,s',y) \urcorner}(\Phi) \\ \mid (\_, \mathsf{Ret}(y)) :: \_ & \Rightarrow \Phi(y) \\ \mid \epsilon & \Rightarrow \mathsf{True} \end{array}$$

end

Fig. 5. The Definition of $I_{\mathsf{atom}_\mu}(\gamma, \mathit{obj}, p)$.

invariant can tell for every in-flight operation whether it has already linearized or not. We use this information to track the status of each operation's atomic update.

$\boxed{I_{\mathsf{atom}_\mu}(\gamma, \mathit{obj}, p)}$ is defined in Figure 5. It consists of five parts: P_1 tracks the system status, P_2 stores the atomic updates and receipts, P_3 and P_4 enforce its internal consistency, and P_5 ensures that the remaining future trace is linearizable. We now describe each of these pieces in more detail.

P_1: System Status. Along with the physical points-tos and the prophecy trace, the status of the object is tracked by several pieces of ghost state that are combined together on this line of the invariant to track components of the object.

- The current configuration $\rho = (\vec{\pi}, \vec{e}, \sigma)$. It is tracked by three pieces of resources:
  - Authoritative resource $\boxed{\bullet \mathit{l2m}(\mathit{activeOp}(\vec{\pi}))}^{\gamma_\pi}$ tracks pending operations, *i.e.*, the operations that have been called but not yet returned. Here, function *activeOp* returns operations that are not $\bot$, and function *l2m* converts an association list $(A \times B)^*$ to a finite map $A \xrightarrow{\mathsf{fin}} B$. The fragmentary copy of this state is kept in the proof environment.
  - Authoritative resource $\boxed{\bullet \vec{e}}^{\gamma_e}$ tracks other threads spawned by operations.
  - $\mathsf{S}_\circ(\sigma)$ asserts the ownership of the data structure's physical state, defined as $\mathop{\Large *}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v$, *i.e.*, the ownership of each location in σ.
- The abstract sequential state s of the object: $\boxed{\bullet [0 \leftarrow s]}^{\gamma_s}$.[6]
- The prophecy to the future trace $\vec{t}'$: $\mathsf{prophTrace}_{obj}(\rho, p, \pi_\Sigma(\vec{t}'))$.

P_2: Atomic Updates and Receipts. Recall that $\mathsf{AU}(\Phi)$ provides a receipt Φ after being fired, and this receipt is specified by a client of the data structure. An operation learns the type of the receipt when it is called. Because of helping, the current operation can be linearized by another operation. To make this work, we need a mapping between operations and their receipt types so that the helper

[6]The actual Iris mechanization does not use ghost map but another type of ghost state known as exclusive authoritative state.

and helpee agree on what the receipt is. This is represented by the authoritative ghost resource $\boxed{\bullet \Phi s}^{\gamma_\phi}$ for the receipt map Φs. Note that this is a higher-order ghost state because Φs contains Iris propositions. Alongside this map, this part of the invariant stores, for each pending operation with tag τ, either its atomic update or its receipt, selected by the function *interpret* according to the next event of τ in the future trace $\vec{t}'$. An operation τ in Φs thus has four possible stages:

(1) The next event of τ is $\mathsf{Call}(x)$, meaning that τ has not been called yet. This case is impossible because uncalled operations are not pending, and thus should not be in Φs.
(2) The next event of τ is $\mathsf{Lin}(x, y)$, meaning that τ has been called, but has yet to linearize. In this case, the invariant stores an atomic update for this operation. The pre- and postconditions of the AU come from specification (4), and the type of the receipt comes from Φs.
(3) The next event of τ is $\mathsf{Ret}(y)$, meaning that τ has linearized, but has not returned yet. In this case, the invariant stores the receipt of this operation.
(4) Tag τ is not in $\vec{t}'$, meaning that τ has returned. In this case, the mission of the invariant for this operation is done, and the invariant stores nothing about the operation.

P_3: Ghost Tokens. This part is an iterated separating conjunction of ghost tokens for all tags in $\vec{\pi}$.1 for two purposes. First, the exclusiveness of ghost tokens ensures that there is no duplication of tags in $\vec{\pi}$. Second, because the tag of a new operation is generated by `newghostID`, tokens are used to prove the freshness of a new tag.

P_4: Consistency of the Status. The pending operation list $\vec{\pi}$, the receipt map Φs, and the future trace $\vec{t}'$ all talk about pending operations, so this part ensures they describe the same set: the domain of Φs coincides with the tags appearing in $\vec{\pi}$, and the pending operations $\mathit{pendingOp}(\vec{t}')$ extracted from the future trace are among them. The subset inclusion $\mathit{pendingOp}(\vec{t}') \subseteq \mathsf{dom}(\Phi s)$ is not necessarily an equality because *pendingOp* is a conservative under-approximation: under partial correctness an operation may be called and never return, in which case its tag is indistinguishable from one that has already returned. The precise definition of *pendingOp* and the lemmas relating it to trace extension are given in Appendix A.2.2.

P_5: Partial Linearizability of Future Trace. This part ensures the future trace $\vec{t}'$ is a suffix of a linearizable trace. First, for each tag, the call, linearization, and return events of it must occur in order, but unlike well-formed$_{\mathsf{lin}}$, the first event does not have to be Call.

$$\mathsf{partial\text{-}well\text{-}formed}_{\mathsf{lin}}(\vec{t}') \triangleq \forall \tau.\, \exists v, v'.\, \pi_\tau(\vec{t}') \sqsubseteq_{\mathsf{sub}} [(\tau, \mathsf{Call}(v)); (\tau, \mathsf{Lin}(v, v')); (\tau, \mathsf{Ret}(v'))]$$

Here, $\vec{x} \sqsubseteq_{\mathsf{sub}} \vec{y}$ means $\vec{x}$ is a subtrace of $\vec{y}$, *i.e.*, $\vec{x}$ can be obtained by taking consecutive elements from $\vec{y}$. Second, the linearization events in the future coincide with the sequential behavior μ. Third, the current configuration is safe, *i.e.*, its pending operations and other threads will never get stuck under the most adversarial interaction.

$$\mathsf{cfg\text{-}safe}(\rho) \triangleq \forall \vec{t}, \vec{\pi}', \vec{e}', \sigma'.\, \rho \xrightarrow{\vec{t}}{}^{*}_{\mathsf{adv}} (\vec{\pi}', \vec{e}', \sigma') \Rightarrow \forall e \in \vec{\pi}'.2 +\!\!+ \vec{e}'.\, e \in \mathit{Val} \vee \mathsf{red}(e, \sigma')$$

Properties of partial-well-formed$_{\mathsf{lin}}$ and cfg-safe used in the proof are collected in Appendix A.2.3.

Additional Auxiliary Resources. The invariant also carries an agreement ghost resource $\boxed{\square(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma}$ to link four ghost names used by I_{atom} to the single ghost name γ exposed by IsP. The agreement ghost resource is a persistent assertion such that $\boxed{\square x}^{\gamma} * \boxed{\square y}^{\gamma} \vdash \ulcorner x = y \urcorner$.

***The Construction of* IsP *and* PState.** The validity of an object and the assertion for its abstract state are defined on top of I_{atom_μ}.

$$\mathsf{IsP}_\mu(\gamma, v) \triangleq \exists obj, p.\, \ulcorner v = (obj, p) \urcorner * \boxed{I_{\mathsf{atom}_\mu}(\gamma, obj, p)}$$

$$\mathsf{PState}_\mu(\gamma, s) \triangleq \exists \gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s. \boxed{\square(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma} * 0 \hookrightarrow^{\gamma_s} s$$

IsP_μ is a thin wrapper around I_{atom_μ} and PState_μ is the ownership of the fragmentary part of the abstract state. The agreement resource ensures γ_s is the same ghost name used in I_{atom_μ}. Although PState_μ is independent of the sequential behavior μ, we still add the μ subscript to emphasize that it is a specific construction used to prove a logically atomic triple about μ.

3.4 Proof Sketch

Since we need the prophecy variable to make this proof work, we can only prove the logical atomicity of the wrapped version of the data structure. Thus, we restate Proposition 3.2 as below:

Theorem 3.3 (Linearizability ⇒ Logical Atomicity).

$$\mathsf{lin}_\mu(\texttt{init}, \texttt{op}) \land \mathsf{seq}(\texttt{init}(), \varnothing) \implies \text{atom-spec}_\mu(\mathsf{IsP}_\mu, \mathsf{PState}_\mu, \mathit{wrapi}(\texttt{init}), \mathit{wrapo}(\texttt{op}))$$

where $\mathsf{seq}(e, \sigma) \triangleq \forall \vec{e}', \sigma'.\ ([e], \sigma) \to^*_{\mathsf{tp}} (\vec{e}', \sigma') \Rightarrow |\vec{e}'| = 1$

Due to a technical restriction, we can only prove the theorem when the `init` function is sequential, as specified by $\mathsf{seq}(\texttt{init}(), \varnothing)$. This is typically not a problem in practice because a concurrent data structure rarely spawns internal helping threads.

We sketch the proof here; a more detailed walk-through is given in Appendix A.3. We must show that the wrapped operations satisfy the logically atomic specification, reasoning under the invariant I_{atom_μ} of Figure 5. After initialization, the argument follows the life cycle of an operation invocation.

Initialization. Running $\mathit{wrapi}(\texttt{init})$ creates the object with initial state $\mathsf{S}_0(\sigma)$ and allocates its prophecy variable $\mathsf{prophTrace}((\epsilon, \epsilon, \sigma), p, \vec{t})$. Because the data structure is linearizable, the prophesied trace $\vec{t}$ is guaranteed to be the physical projection of some linearizable annotated trace $\vec{t}'$. This is exactly what is needed to establish I_{atom_μ} for the empty initial configuration. After creating the associated ghost state and invariant, we return the assertions IsP_μ and PState_μ in the postcondition.

Operation Invocation. The prophesied future trace records, for every pending operation, where its linearization point will be. When the wrapped operation resolves its call event, we open the invariant and consult this prophecy: every operation whose linearization point in the prophecy falls before this call is helped. For each helped operation, we fire its stored atomic update, advance the object's abstract state through the corresponding sequential transition, and replace the fired update with the receipt it produces. The current operation is then registered, with its own atomic update stored for later.

The internal steps of `op` merely preserve the invariant. For each internal step, since it is safe (by assumption of linearizability), we use the stored physical state in the invariant to justify the step, and then update the configuration and state in the invariant. The proof of this part follows the same pattern as Theorem 17 of Hostert et al. [36].

Finally, resolving the *return* event performs any remaining helping and extracts this operation's receipt, which is precisely the postcondition demanded by its logically atomic triple.

4 Transporting Logical Atomicity through Logical Refinement

Contextual refinement is another commonly used approach to specifying concurrent data structures. For example, with contextual refinement, one can capture that a fine-grained concurrent data structure behaves in an atomic way by proving that it refines a coarse-grained implementation where the operations are guarded by a global lock. Such a contextual refinement implies that any behaviors that a client of the fine-grained implementation can observe are a subset of the

behaviors that they could observe with the coarse-grained implementation. Because the latter behaves atomically by construction, this means the former also appears to do so from the client's perspective.

Since the coarse-grained implementation is atomic by construction, one can naturally also prove a logically atomic triple specification for it. One might expect that the contextual refinement would also imply that this logically atomic specification must hold for the fine-grained implementation as well. However, there has previously been no way to go from a refinement to a logically atomic triple. This is a notable gap, since in contrast, prior work *has* established equivalences between contextual refinement and linearizability [22].

Using our results from previous sections, we close this gap by showing how to transport a logically atomic triple across a refinement specification. Specifically, we target ReLoC [23], a relational logic defined on top of Iris [40] for proving contextual refinement of HeapLang programs. ReLoC introduces a *logical refinement judgment* $e \precsim e' : A$ where A is an Iris relation that holds on the return values of e and e'. For example, if $A \triangleq (=)$, *i.e.*, the Leibniz equality relation, then e and e' always return equal values. ReLoC's soundness theorem shows that logical refinement implies contextual refinement.

We show that ReLoC's notion of logical refinement transports logically atomic specifications.

Theorem 4.1 (Refinement Transports Logical Atomicity). *If*

(1) $\mathsf{seq}(\texttt{init}(), \varnothing)$ *and* $\mathsf{seq}(\texttt{init'}(), \varnothing)$,
(2) $\mathsf{atom\text{-}spec}_{\mu}(\mathsf{IsP}', \mathsf{PState}', \texttt{init}', \texttt{op}')$,
(3) $\texttt{init}() \precsim \texttt{init'}() : A$, *and*
(4) $A(obj, obj') * \ulcorner vr(x) \urcorner \vdash \texttt{op}(obj, x) \precsim \texttt{op'}(obj', x) : (=)$ *for all obj, obj', and x,*

then $\mathsf{atom\text{-}spec}_{\mu}(\mathsf{IsP}_{\mu}, \mathsf{PState}_{\mu}, \mathit{wrapi}(\texttt{init}), \mathit{wrapo}(\texttt{op}))$.

To use this theorem, a user picks a relation A that relates the result of `init` and `init'`. This is typically an invariant that relates the internal states of the data structures on the two sides of the refinement. Then given A-related data structures obj, obj' and a valid request x, the user has to show that $\texttt{op}(obj, x)$ refines $\texttt{op'}(obj', x)$ and they return equal results. As a result, if $(\texttt{init}', \texttt{op}')$ satisfies a logically atomic specification, then so does $(\texttt{init}, \texttt{op})$.

To establish our transport theorem, we first show that logical refinement can be used to transport linearizability. Theorem 4.1 then follows immediately from Theorem 3.1 and Theorem 3.3.

Lemma 4.2 (Refinement Transports Linearizability). *If*

(1) $\mathsf{seq}(\texttt{init}(), \varnothing)$ *and* $\mathsf{seq}(\texttt{init'}(), \varnothing)$,
(2) $\mathsf{lin}_{\mu}(\texttt{init}', \texttt{op}')$,
(3) $\texttt{init}() \precsim \texttt{init'}() : A$, *and*
(4) $A(obj, obj') * \ulcorner vr(x) \urcorner \vdash \texttt{op}(obj, x) \precsim \texttt{op'}(obj', x) : (=)$ *for all obj, obj', and x,*

then $\mathsf{lin}_{\mu}(\texttt{init}, \texttt{op})$.

The fact that this lemma should be true is not surprising, because, as alluded to above, prior work by Filipović et al. [22] has established correspondences between contextual refinements and linearizability, and we know that logical refinement implies contextual refinement. However, that prior work dealt with a considerably simpler first-order programming language that lacked dynamic allocation. Thus, we cannot directly appeal to prior results.

The key idea behind our proof of this lemma is to construct a program called the *most general client* (MGC) that covers all possible behaviors permitted by the most adversarial steps relation. We then use refinement to transport the MGC's interaction with `op` into an interaction with `op'`. Similar most general client constructions are used elsewhere in the linearizability and refinement literature. The

idea is that this client repeatedly invokes concurrent operations with non-deterministic arguments. This non-determinism ensures that for any sequence of invocations in the most adversarial step relation, the most general client could trigger a similar sequence of concurrent calls.

One subtlety is that, unlike the way this program is constructed in some prior work, we *cannot* construct a single most general client program in HeapLang: the language does not have primitives that can non-deterministically generate all possible valid arguments to invoke the operation with. We instead define a function $MGC : \Sigma^* \to \mathit{Val}$ at the meta-level, such that given a trace $\vec{t}$, the program $MGC(\vec{t})$ invokes concurrent operations in the pattern specified by $\vec{t}$. In particular, this $MGC(\vec{t})$ is a HeapLang function that takes an operation `op` and an object *obj* as arguments.

Concretely, to show that (`init`, `op`) is linearizable, we are given some execution of `init` returning an object *obj*, along with $\rho \xrightarrow[\mathit{obj.op/vr}]{\vec{t}}{}^{*}_{\mathsf{adv}} \rho'$, and we have to show that $\vec{t}$ is a linearizable trace. By construction, there is an execution of $MGC(\vec{t})(\mathtt{op}, \mathit{obj})$ that exhibits the same behavior and order of events as this adversarial interaction. From the assumption that `init` refines `init'`, we know that there is a corresponding execution of `init'` returning some object *obj'* such that *obj* refines *obj'*. We then show that $MGC(\vec{t})(\mathtt{op}, \mathit{obj})$ refines $MGC(\vec{t})(\mathtt{op'}, \mathit{obj'})$, which in turn gives us an execution trace for $MGC(\vec{t})(\mathtt{op'}, \mathit{obj'})$. We conclude by converting this MGC execution into a most adversarial interaction for (`init'`, `op'`) with trace $\vec{t}$ and applying the assumption that (`init'`, `op'`) is linearizable.

In order to make the argument in the previous paragraph work, we must ensure (1) that the MGC indeed fully generates equivalent traces, and (2) that a refinement between MGCs implies that the traces are equivalent. The latter involves another slight technical complication: contextual refinement is an extensional property and is only concerned with the externally observable behavior; however, in HeapLang, calls and returns are *not* externally observable, because functions are first-class values and calls are just beta-reductions of λ terms.

To deal with this, we have the MGC keep track of call/return events in a physical log stored at some reference cell ℓ_{tr}. The $MGC(\vec{t})(\mathtt{op}, \mathit{obj})$ construction forks a new thread for each $\mathsf{Call}(x)$ operation in the trace $\vec{t}$, with shared access to this reference cell. Each thread invokes `op` on *obj* with request x, but it also "emits" an event (containing the request) by adding the event to the physical log stored in the reference cell. Similarly, after the function returns, the result is also added to the log. By dereferencing ℓ_{tr} at the end of $MGC(\vec{t})$, we make the trace events physically observable, and as a consequence, the MGC refinement implies trace refinement. Additional details on the MGC construction and supporting lemmas appear in Appendix B.

5 Embedding External Linearizability Techniques

Besides formally characterizing the expressivity of the logic, our completeness result (Theorem 3.3) also provides a new approach for obtaining a logically atomic specification in Iris: we can verify linearizability of a data structure using any technique external to Iris and then formally embed the result as a logically atomic specification.

Among the vast array of linearizability proof methods in the literature, we reformulate three representative methods, aspect-oriented proofs for queues [30], forward simulation with commit points [6], and meta-configuration tracking [39], using the approach below. As a first step, we mechanize in Rocq the meta-theorems from these prior works showing that these methods indeed imply linearizability. Since these techniques have already been described in the literature, after mechanizing the statement of these meta-theorems, we were able to discharge most of the proofs using Large Language Models (LLMs).

Each of these methods requires showing some conditions on the trace of events that a data structure can generate. So, to apply these methods, one needs a way to prove that the traces generated by an implementation of a data structure satisfy these conditions. In our case, we will

use Iris itself to prove that these trace conditions hold. This means we can still take advantage of Iris's expressive features for modular reasoning about concurrent state manipulation, but we avoid having to explicitly reason about linearization points and atomic updates. Instead, we just prove the trace property holds, apply the meta-theorems to deduce linearizability, and thereby obtain logical atomicity.

To leverage Iris for this task, we adapt a *free theorem* result that Birkedal et al. [5] established for Iris. Their result allows one to deduce trace properties from the separation logic specifications of a library. Their approach models a trace as a separation logic resource, and puts this resource in an appropriate Iris invariant, so that the usual soundness theorem of Iris implies properties about the resulting trace that a program can generate.

In the remainder of this section, we start by adapting their theorem to account for our adversarial-steps trace reduction relation, and then we describe each of the external linearizability proof methods we have formulated in this manner. The following section (§6) describes how we then use these methods to get logically atomic triples for various example data structures.

5.1 Deriving Trace Invariants from Separation Logic Specifications

Birkedal et al. [5] propose a theorem to get a trace property about a data structure from its separation logic specification. Their key idea is adding a wrapper around data structure operations to emit a call-return trace via two special primitive operations **`emit`** and **`fresh`**. Because they design the specifications of these operations in a way that ensures the emitted trace must satisfy a predicate $I\colon \Sigma^* \to \mathit{Prop}$, every trace produced by an interaction between the data structure and a well-formed client that uses this specification must also belong to I.

In our case, we want to talk about the traces that would be generated under the most adversarial steps relation, in order to connect to linearizability. Thus, we reuse the most general client (MGC) construction from the proof of Lemma 4.2: a client that spawns one thread for each operation in a trace $\vec{t}$. Instead of using primitive **`emit`** and **`fresh`** commands, we simulate a trace by storing these events in a physical reference cell.

As we discussed in §4, executions of the MGC cover all behaviors captured by the $\rightarrow^*_{\mathsf{adv}}$ relation. Consequently, to derive a property of every adversarial trace, it suffices to prove a triple about every instance of the MGC using a data structure's operations.

That in turn only requires proving a specification we call trace-spec$_I$(MgcInv, *vr*, `init`, `op`) about `init` and `op` wrapped with the trace manipulating code. This specification asserts that, for some invariant MgcInv chosen by the prover, (1) initializing the data structure establishes MgcInv, and (2) for a valid request, the wrapped operation is safe given the trace invariant. Proving this requires showing, at each call to `emit` and `fresh`, that the appended event preserves I. Chaining everything together, we have this theorem.

Theorem 5.1. *If* trace-spec$_I$(MgcInv, *vr*, `init`, `op`), *then*

(1) safe(`init`(), $\emptyset$) *and*
(2) if ([`init`()], $\emptyset$) $\rightarrow^*_{\mathsf{tp}}$ ([*obj*], σ) *and* $(\epsilon, \epsilon, \sigma) \xrightarrow[\mathit{obj}.\mathsf{op}/\mathit{vr}]{\vec{t}}{}^*_{\mathsf{adv}} (\vec{\pi}, \vec{e}, \sigma')$, *then every expression in* $\vec{\pi}$ *and* $\vec{e}$ *is not stuck, and* $I(\vec{t})$.

Additional details about the lemmas and proof of this theorem are included in Appendix B. Each linearizability proof method we consider in the remainder of this section requires showing conditions on a data structure's traces that can be formulated in terms of an appropriate predicate I in the above theorem. Thus, we can use Iris to prove that these conditions hold.

5.2 Aspect-Oriented Proofs

The idea behind aspect-oriented proofs of linearizability is to find, for a given class of data structures such as queues or stacks, a collection of properties about traces that imply linearizability for any implementation of those data structures. For example, the original result by Henzinger et al. [30] establishes such a theorem for queues saying that if for every completion of an adversarial call-return trace, there exists an injective partial map M from dequeue (pop) operations to enqueue (push) operations, and M satisfies some consistency properties, then the data structure is linearizable as a queue. Some examples of these consistency properties are

- *Valid Match.* If a dequeue τ_2 is mapped to an enqueue τ_1, then the response of τ_2 must equal the request of τ_1, and τ_2 must not happen before τ_1 (*i.e.*, τ_2 must not finish before τ_1 starts).
- *Ordered Correct.* For two pairs of enqueue and dequeue operations $(M(\tau_1), \tau_1)$ and $(M(\tau_2), \tau_2)$, if $M(\tau_1)$ happens before $M(\tau_2)$, then τ_2 must not happen before τ_1.
- *Empty Sound.* If a dequeue reports that the queue is empty, then the queue must be logically empty at some point between the call and return of the dequeue.

Later, Dodds et al. [15] formulated an analogous theorem for stacks, and Ömhan and Nanevski [57] developed similar results for snapshottable arrays.

We have adapted Henzinger et al.'s queue theorem to our setting. To state our result, we first need to define a sequential specification μ_{queue} for queues, following the style of μ_{bag} in Example 2.3:

$$\begin{aligned} \mu_{queue}.s_0 &\triangleq \epsilon \qquad\qquad \mu_{queue}.vr \triangleq \mu_{bag}.vr \\ \mu_{queue}.rr(s, x, s', y) &\triangleq (\exists v.\, x = \mathbf{some}(v) \land s' = s +\!\!+ [v] \land y = ()) \\ &\quad \lor x = \mathbf{none} \land s = \epsilon \land s' = \epsilon \land y = \mathbf{none} \\ &\quad \lor x = \mathbf{none} \land \exists v.\, s = v :: s' \land y = \mathbf{some}(v) \end{aligned}$$

Then, our formulation of their theorem says

Theorem 5.2. *For a physical trace $\vec{t}$, if* well-formed-bag$(\vec{t})$ *and* $I_{queue}(\vec{t})$, *then* $\mathsf{trlin}_{\mu_{queue}}(\vec{t})$, *where*

$$\begin{aligned} \text{well-formed-bag}(\vec{t}) &\triangleq \forall \tau.\, (\exists v.\, \pi_\tau(\vec{t}) \sqsubseteq_{\mathsf{pre}} [(\tau, \mathsf{Call}(\mathbf{some}\, v)); (\tau, \mathsf{Ret}(()))]) \\ &\quad \lor (\exists v.\, \pi_\tau(\vec{t}) \sqsubseteq_{\mathsf{pre}} [(\tau, \mathsf{Call}(\mathbf{none})); (\tau, \mathsf{Ret}(\mathbf{some}\, v))]) \\ &\quad \lor \pi_\tau(\vec{t}) \sqsubseteq_{\mathsf{pre}} [(\tau, \mathsf{Call}(\mathbf{none})); (\tau, \mathsf{Ret}(\mathbf{none}))] \\ I_{queue}(\vec{t}) &\triangleq \exists M.\, \text{valid-match}(\vec{t}, M) \land \text{empty-sound}(\vec{t}, M) \\ &\quad \land \text{order-matched}(\vec{t}, M) \land \text{order-unmatched}(\vec{t}, M) \end{aligned}$$

I_{queue} mostly follows the consistency properties required by Henzinger et al., but is tweaked to account for partial correctness. Notably, unlike the original version that relies on a completion of the trace, our I_{queue} can be incrementally verified: the empty list satisfies I_{queue}, and knowing that $I_{queue}(\vec{t})$, verifying $I_{queue}(\vec{t} +\!\!+ u)$ only requires checking the new event u is compatible with $\vec{t}$. This requires slightly strengthening valid-match. In addition to the requirement by Henzinger et al., our valid-match further requires that for enqueues τ_1 and τ_2, if τ_1 happens before τ_2 and τ_2 is matched with some dequeue by M, τ_1 must also be matched by M. This additional restriction forces the user to construct M incrementally along with the execution of the queue, instead of delaying matching τ_1 with a dequeue forever.

5.3 Forward Simulation with Commit Points

In §4 we discussed methods of transporting linearizability (and logical atomicity) by establishing a refinement between an implementation M of some data structure and another implementation M', and then showing that M' is linearizable. However, one can also use refinement to transport

linearizability even when M' is not another code-level implementation but is instead an abstract state machine or labeled transition system (LTS). In particular, if one can prove a trace refinement between M and a transition system M', and the traces of the transition system are all linearizable, then M is linearizable. In many cases, we can take M' to simply be an LTS which performs the operations in atomic transitions according to the sequential specification. This M' is then linearizable by definition.

Then, to establish the refinement, one approach is to construct a forward simulation between M and M'. Such forward simulation proofs are natural to carry out inside of a concurrent separation logic such as Iris, because they can be constructed as we reason over the steps of M. However, in some cases, it is not possible to directly construct a forward simulation between M and an atomic transition system M'. In particular, if M has future-dependent linearization points, then such a forward simulation is not possible.

Bouajjani et al. [6] propose an alternate technique to work around this. Specifically, for proving linearizability of queues, they define an abstract state machine *AbsQ* that only maintains a partial order of enqueues rather than a full sequential order. This makes it possible to prove a forward simulation between a queue implementation and *AbsQ*, even for queues that have future-dependent enqueues. They then show that *AbsQ* is linearizable *w.r.t.* the standard queue model using a *backward* simulation proof. Thus, establishing a forward simulation to *AbsQ* implies linearizability. Bouajjani et al. also develop a similar abstract state machine for stacks.

Following their approach, we mechanize a definition of *AbsQ* as an LTS and formalize their key theorem:

Theorem 5.3. *For a physical trace $\vec{t}$, if there exists an annotated trace $\vec{t}'$ that forward simulates AbsQ and $\vec{t} = \pi_\Sigma(\vec{t}')$, then* $\mathsf{trlin}_{\mu_{queue}}(\vec{t})$.

The existence of such an annotated trace that forward simulates *AbsQ* can be established using an appropriate trace invariant in Iris.

5.4 Meta-Configuration Tracking

Jayanti et al. [39] note that when doing forward reasoning, if instead of maintaining one abstract state of the data structure, one considers a set of possible abstract states, one obtains a complete forward-reasoning-based linearizability proof technique. They call the set of states a meta-configuration and hence name their technique meta-configuration tracking. Unlike the two techniques above, this technique is universal in the sense that it is not specific to a particular class of data structures, such as queues or stacks.

In our mechanization, we define a single configuration as a pair of the form $(s, OpSt)$ with type $S \times (Tag \xrightarrow{\mathsf{fin}} (Val \times Val^?))$, where s represents the state of the object, and $OpSt$ is a map from operation invocation tags to a *status*. There are two possible statuses for an operation:

(1) called with v but not linearized, which is represented as $(v, \bot)$; or
(2) called with v and linearized with w, which is represented as (v, w).

A meta-configuration is defined as a set of single configurations. We define a transition relation on meta-configurations $M_1 \xrightarrow[\mu]{\vec{t}}{}^*_{\mathsf{mc}} M_2$ with three types of atomic transitions:

(1) an operation τ is called with value v, which emits event $(\tau, \mathsf{Call}(v))$ and adds a new operation $[\tau \leftarrow (v, \bot)]$ for *every* configuration in the set;
(2) an operation τ in a configuration $(s, OpSt[\tau \leftarrow (v, \bot)])$ possibly linearizes with w, which does not emit any event but inserts a new configuration $(s', OpSt[\tau \leftarrow (v, w)])$ satisfying $rr(s, v, s', w)$; and

(3) an operation τ returns with w, which emits event $(\tau, \mathsf{Ret}(w))$ and prunes every single configuration whose status of τ is not $(\_, w)$.

Then, we have the following theorem:

Theorem 5.4. *For a physical trace $\vec{t}$, if* $\mathsf{well\text{-}formed}(\vec{t})$ *(as defined in eq. (1)) and there exists a non-empty meta-configuration M such that* $\{(s_0, \varnothing)\} \xrightarrow{\vec{t}}{}^{*}_{\mu\,\mathsf{mc}} M$, *then* $\mathsf{trlin}_\mu(\vec{t})$.

Again, such a meta-configuration can be constructed in an Iris Hoare triple proof using ghost state, allowing us to apply this method within Iris.

6 End-to-End Logical Atomicity Results

As end-to-end case studies, we prove the linearizability of several queue algorithms using techniques presented in §4 and §5, and obtain logically atomic triples for them using Theorem 3.3. Since proofs of linearizability using these techniques have already been described in the literature, either with pencil and paper or mechanically using a different framework, writing these proofs using our reformulation requires little human effort, and substantial parts of the Rocq proofs for several of these examples were automated using LLMs.

Herlihy-Wing Queue. The Herlihy-Wing queue [33] is a famous lock-free queue algorithm known for its future dependency, and is a standard benchmark for linearizability proof techniques. Jung et al. [41] mechanized a proof of logical atomicity for the Herlihy-Wing queue using prophecy variables in Iris, and an implementation of this data structure in HeapLang is maintained as part of the Iris project. We prove the linearizability of this implementation in three different ways, using the aspect-oriented, forward simulation, and meta-configuration tracking methods described in the previous section, and obtain logically atomic triples that have the same form as the ones proven by Jung et al. [41]. For the proof of linearizability using each technique, we verify a trace invariant using Theorem 5.1, and then conclude this trace is linearizable using the technique-specific theorem. None of these linearizability proofs use prophecy variables. Notably, the aspect-oriented proof is 27% shorter than the direct Iris proof (2004 LoC vs 2763 LoC), though this does not include the proof of the aspect-oriented queue meta-theorem.

Baskets Queue. The Baskets Queue is another lock-free queue [34]. Like the Herlihy-Wing queue, it features a challenging form of future-dependent linearization in enqueue operations. Bouajjani et al. [6] mention it as one example for which their forward simulation proof technique should be applicable. We implement the Baskets Queue in HeapLang, prove its linearizability using their forward simulation, and obtain a logically atomic specification for it. Since HeapLang is designed as a garbage-collected language, our implementation of the algorithm removes several aspects of the original implementation designed for dealing with manual reclamation and avoiding the ABA-problem [53]. Nevertheless, it still retains the challenging future dependency in enqueues. To our knowledge, this is the first machine-checked proof of linearizability for the Baskets Queue.

Folly MPMC Queue. The multi-producer-multi-consumer (MPMC) queue from Meta's C++ library Folly [49] is a high-performance bounded concurrent queue. The queue is optimized, production-ready, and it scales to thousands of consumer and producer threads.

Vindum et al. [76] implement the MPMC queue in HeapLang and show using ReLoC that it contextually refines a coarse-grained queue. A key challenge of the refinement proof is the queue's use of external linearization points. Using Theorem 4.1, we directly reuse their refinement proof to derive a logically atomic specification for the MPMC queue. Proving the logically atomic specification for the coarse-grained queue is straightforward.

7 Related Work

Linearizability Proof Techniques. As discussed in §5, one of the benefits of our completeness result is that it allows us to use external linearizability proof techniques in order to establish logical atomicity inside of Iris. This is beneficial because, as we alluded to, there is a vast array of such techniques that have been developed in the research literature. Much of the focus of the research community has been specifically developing methods that work for data structures that exhibit challenging features, such as helping and future-dependent linearization points. We have already used and discussed several of these, including aspect-oriented linearizability [30], a forward simulation refinement technique [6], and meta-configuration tracking [39]. Other well-known methods that have found repeated use include simulation proofs with canonical automata [11] and hindsight reasoning [56]. Dongol and Derrick [17] provide a survey of many other techniques, along with what examples they have been applied to and known limitations. It would be interesting to similarly mechanize these other proof methods and use them in combination with our results.

Program Logics for Linearizability. A number of program logics are designed to directly establish linearizability, rather than trying to internalize atomicity in the logic. The RGSep logic of Vafeiadis and Parkinson [73] was used in this way to prove linearizability by instrumenting code with linearization-point actions and establishing a simulation [71, 72]. The RGSim [47, 48] framework handles helping and future-dependent linearization points. Its soundness theorem yields a contextual refinement that is equivalent to linearizability. Khyzha et al. [43] present a logic that allows for incrementally constructing a partial order over operations rather than a single linearization, allowing ordering decisions to be delayed for data structures with future-dependent linearizations. Several program logics have been constructed around the hindsight reasoning method mentioned above, which involves some form of temporal reasoning [51, 52, 56].

As mentioned earlier, one limitation of working with linearizability inside of a program logic is that it is often not possible to compose such linearizability specifications inside of the program logic, a limitation that motivated the development of logically atomic specifications. However, a line of work has sought to reconstruct and generalize linearizability in a way that is more directly compositional. Vale et al. [74] use concurrent game semantics to derive such a generalized form of linearizability and give a program logic that is sound with respect to this generalized notion.

Most recently, Hatti et al. [28] have developed a rely-guarantee logic that is both sound and complete for a version of this compositional form of linearizability. They achieve completeness by incorporating a form of *possibility reasoning*, which maintains a set of possible linearizations, in contrast to our use of prophecy variables. Their result also generalizes several variants of linearizability, including atomic, set, and interval linearizability. In comparing their approach to logical atomicity, they note that "[i]t is known that a logically atomic triple implies Herlihy-Wing linearizability… but there is no evidence that logical atomicity can be used to specify all Herlihy-Wing linearizable objects, so its completeness is unknown." Our results close this gap by showing that in fact logical atomicity is complete for linearizability.

Logical Atomicity. Jacobs and Piessens [38] introduced an approach to internalizing atomicity in a program logic by parameterizing specifications with a client-supplied action that updated ghost state. The operation would invoke this action at the point where the operation logically took effect. HOCAP [67] developed this higher-order style further. TaDA [12] introduced a distinct *atomic triple* judgment, though in its original form it could not reason about helping. Iris [42] showed how to encode the logically atomic triples of TaDA inside of a higher-order framework using atomic update predicates that are similar to the ghost updates in Jacobs and Piessens's framework.

None of these frameworks for logically atomic specifications could initially deal with future-dependent linearization points. This was resolved by introducing prophecy variables to Iris [41], which are essential for our completeness proof. Jung et al. [41] also noted that for many forms of future-dependent linearization, a stronger form of `resolve` called *atomic resolve* is needed to prove logical atomicity, but our work suggests that the atomic resolve is actually unnecessary for completeness. One of the motivations for later credits [65] was for reasoning about a form of helping in logical atomicity proofs that Spies et al. [65] called *unsolicited helping*. But again, for completeness, later credits turn out to be unnecessary.

Using completeness, our work has shown how to embed other linearizability proof methods in Iris in order to obtain logically atomic specifications. Prior to our work, Patel et al. [60] encoded a form of the hindsight reasoning linearizability proof method into Iris. Their encoding instruments a program with prophecy variables and uses these, along with helping, to derive logically atomic specifications from a user-supplied hindsight proof. Our proof uses a similar combination of prophecies with helping to derive logical atomicity from any linearizability proof.

Although we have focused on the correspondence between logical atomicity and linearizability, there are data structures that can be specified using logically atomic triples that fall outside the scope of linearizability. For example, da Rocha Pinto et al. [12] observe that atomic triples can capture behaviors that the sequential specifications used in linearizability cannot, because the atomic triple can constrain how a client may invoke operations through expressive preconditions.

An alternative to logical atomicity has been pursued by the FCSL logic [55], based on using Hoare triples over time-stamped histories tracked as ghost state. Sergey et al. [63] advocate such history-tracking specifications as a compositional correctness condition in their own right, showing how to specify certain classes of data structures and properties that lie outside the scope of linearizability specifications.

Automated Linearizability Provers. Many methods for establishing linearizability have been incorporated into automated tools. Vafeiadis [72] developed CAVE, a tool that automatically proves linearizability by searching for linearization points. CAVE cannot handle general future dependency. However, the original work on aspect-oriented linearizability showed that CAVE could be used to check some of the conditions needed to apply the queue aspect-oriented theorem. Zhu et al. [79]'s Poling tool goes beyond CAVE and is able to handle helping. More recently, Meyer et al. [51] and Meyer et al. [50] develop separation logic-based tools that use hindsight reasoning. Although our examples have used proof techniques that can be done with mechanized interactive proofs in Rocq, it would be interesting to try to connect to automated techniques for establishing linearizability as a method for incorporating logically atomic specifications in Iris.

In particular, it would be interesting to compare the automation achievable using tools designed for linearizability, as opposed to automated or semi-automated techniques targeting logical atomicity directly. Several works have taken this latter route. Voila [77] is a proof outline checker for the TaDA logic that checks a program annotated with key steps. Mulder and Krebbers [54] extend the Diaframe proof automation library for Iris to handle logical atomicity and contextual refinement. They achieve full automation on simpler examples, while more complex examples allow for a mix of automation and interactive tactics. Raven [26] is an SMT-based deductive verifier whose metatheory is based on a first-order fragment of Iris with support for logically atomic triples.

8 Conclusion and Future Work

This paper presented a completeness theorem for logically atomic specifications in Iris, showing that for any linearizable data structure, a corresponding logically atomic triple holds for the data structure's operations. Although the proof is mechanized for HeapLang, the idea behind it should

generalize to any Iris-based program logic that supports prophecy variables and satisfies Condition 18 of Hostert et al. [36]. Besides showing that Iris's existing reasoning mechanisms are sufficient to verify atomic triples for any such data structure, this result also enables alternative linearizability proof techniques to be encoded into Iris as a means of deriving logically atomic triples. We have carried out several such encodings in this paper, but it would be interesting to explore others, and to apply these results to verify additional data structures.

Although linearizability is an important correctness condition for concurrent data structures, there are many variations and generalizations beyond linearizability. In future work, it would be interesting to attempt to generalize our completeness results to these variants of linearizability. Another promising direction would be to use our completeness result as the starting point for a bridge to connect Iris to existing automated provers for linearizability.

Data Availability Statement

The Rocq mechanization accompanying this work is available on GitHub at https://github.com/u8cat/iris-logatom.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 2319168 and Grant No. 2524669, as well as the Carlsberg Foundation under Grant No. CF23-0791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these funding agencies.

References

[1] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theor. Comput. Sci.* 82, 2 (May 1991), 253–284. doi:10.1016/0304-3975(91)90224-P

[2] Marcos K. Aguilera and Svend Frølund. 2003. *Strict Linearizability and the Power of Aborting.* Technical Report HPL-2003-241. HP Laboratories. https://shiftleft.com/mirrors/www.hpl.hp.com/techreports/2003/HPL-2003-241.pdf

[3] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018.* ACM, 7–16. doi:10.1145/3212734.3212753

[4] Lars Birkedal and Aleš Bizjak. 2023. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. https://iris-project.org/tutorial-material.html. https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf Aarhus University.

[5] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (Aug. 2021), 29 pages. doi:10.1145/3473586

[6] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 542–563.

[7] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010.* ACM, 330–340. doi:10.1145/1806596.1806634

[8] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211).* Springer, 87–107. doi:10.1007/978-3-642-28869-2_5

[9] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. 2015. Help!. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015.* ACM, 241–250. doi:10.1145/2767386.2767415

[10] Berk Çirisci, Constantin Enea, Azadeh Farzan, and Suha Orhun Mutluergil. 2020. Root Causing Linearizability Violations. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224).* Springer, 350–375. doi:10.1007/978-3-030-53288-8_17

[11] Robert Colvin, Simon Doherty, and Lindsay Groves. 2005. Verifying Concurrent Data Structures by Simulation. In *REFINE (Electronic Notes in Theoretical Computer Science, Vol. 137)*. Elsevier, 93–110.

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

[13] Oksana Denysyuk and Philipp Woelfel. 2015. Wait-Freedom is Harder Than Lock-Freedom Under Strong Linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9363)*. Springer, 60–74. doi:10.1007/978-3-662-48653-5_5

[14] John Derrick and Graeme Smith. 2017. An Observational Approach to Defining Linearizability on Weak Memory Models. In *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10321)*. Springer, 108–123. doi:10.1007/978-3-319-60225-7_8

[15] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 233–246. doi:10.1145/2676726.2676963

[16] Simon Doherty and John Derrick. 2016. Linearizability and Causality. In *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9763)*. Springer, 45–60. doi:10.1007/978-3-319-41591-8_4

[17] Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48, 2, Article 19 (Sept. 2015), 43 pages. doi:10.1145/2796550

[18] Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On abstraction and compositionality for weak-memory linearisability. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*. Springer, 183–204. doi:10.1007/978-3-319-73721-8_9

[19] Michael Emmi and Constantin Enea. 2019. Violat: Generating Tests of Observational Refinement for Concurrent Objects. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*. Springer, 534–546. doi:10.1007/978-3-030-25543-5_30

[20] Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018 (LIPIcs, Vol. 121)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:21. doi:10.4230/LIPIcs.DISC.2018.23

[21] 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. doi:10.1145/3428196

[22] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51 (2010), 4379–4398. doi:10.1016/j.tcs.2010.09.021 European Symposium on Programming 2009.

[23] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). doi:10.46298/LMCS-17(3:9)2021

[24] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

[25] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*. ACM, 373–382. doi:10.1145/1993636.1993687

[26] Ekanshdeep Gupta, Nisarg Patel, and Thomas Wies. 2025. Raven: An SMT-Based Concurrency Verifier. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 15931)*, Ruzica Piskac and Zvonimir Rakamaric (Eds.). Springer, 81–106. doi:10.1007/978-3-031-98668-0_4

[27] Lee Zheng Han and Umang Mathur. 2025. Efficient Decrease-and-Conquer Linearizability Monitoring. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 2030–2057. doi:10.1145/3763123

[28] Eashan Hatti, Arthur Oliveira Vale, Zhongye Wang, Yueyang Feng, and Zhong Shao. 2026. A Complete Program Logic for Compositional Linearizability. In *40th European Conference on Object-Oriented Programming (ECOOP 2026) (LIPIcs, Vol. 372)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 11:1–11:28. doi:10.4230/LIPIcs.ECOOP.2026.11

[29] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*. Springer, 449–465. doi:10.1007/978-3-319-21668-3_26

[30] Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR 2013 – Concurrency Theory*, Pedro R. D'Argenio and Hernán Melgratti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 242–256.

[31] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. doi:10.1145/114005.102808

[32] M. P. Herlihy and J. M. Wing. 1987. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 13–26. doi:10.1145/41625.41627

[33] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. doi:10.1145/78969.78972

[34] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The Baskets Queue. In *OPODIS (Lecture Notes in Computer Science, Vol. 4878)*. Springer, 401–414.

[35] Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. In *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9039)*. Springer, 50–65. doi:10.1007/978-3-319-19195-9_4

[36] Johannes Hostert, Zichen Zhang, Puming Liu, Simon Oddershede Gregersen, Ralf Jung, and Joseph Tassarotti. 2026. Completeness of Iris-Based Program Logics. *Proc. ACM Program. Lang.* 10, ICFP, Article 284 (aug 2026), 36 pages. doi:10.1145/3828682

[37] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9888)*. Springer, 313–327. doi:10.1007/978-3-662-53426-7_23

[38] Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 271–282. doi:10.1145/1926385.1926417

[39] Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. 2024. A Universal, Sound, and Complete Forward Reasoning Technique for Machine-Verified Proofs of Linearizability. *Proc. ACM Program. Lang.* 8, POPL, Article 82 (jan 2024), 29 pages. doi:10.1145/3632924

[40] 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

[41] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (dec 2019), 32 pages. doi:10.1145/3371113

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

[43] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 639–667. doi:10.1007/978-3-662-54434-1_24

[44] Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. 2023. Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13964)*. Springer, 156–169. doi:10.1007/978-3-031-37706-8_8

[45] Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 79–102. doi:10.1007/978-3-319-96145-3_5

[46] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 218–231. doi:10.1145/3009837.3009877

[47] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 459–470. doi:10.1145/2491956.2462189

[48] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM, 455–468. doi:10.1145/2103656.2103711

[49] Meta. 2026. Folly: Facebook Open-Source Library. https://github.com/facebook/folly/tree/v2026.06.29.00

[50] Roland Meyer, Anton Opaterny, Thomas Wies, and Sebastian Wolff. 2023. nekton: A Linearizability Proof Checker. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13964)*, Constantin Enea and Akash Lal (Eds.). Springer, 170–183. doi:10.1007/978-3-031-37706-8_9

[51] Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A concurrent program logic with a future and history. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1378–1407. doi:10.1145/3563337

[52] Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. Embedding Hindsight Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1848–1871. doi:10.1145/3591296

[53] Maged M. Michael and Michael L. Scott. 1998. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *J. Parallel Distributed Comput.* 51, 1 (1998), 1–26.

[54] Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 462–491. doi:10.1145/3586043

[55] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–310. doi:10.1007/978-3-642-54833-8_16

[56] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *PODC*. ACM, 85–94.

[57] Joakim Öhman and Aleksandar Nanevski. 2022. Visibility reasoning for concurrent snapshot algorithms. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. doi:10.1145/3498694

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

[59] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. 2019. Checking linearizability using hitting families. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*. ACM, 366–377. doi:10.1145/3293883.3295726

[60] Nisarg Patel, Dennis Shasha, and Thomas Wies. 2024. Verifying Lock-Free Search Structure Templates. In *ECOOP (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:28.

[61] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366. doi:10.1090/S0002-9947-1953-0053041-6

[62] Gal Sela, Maurice Herlihy, and Erez Petrank. 2021. Brief Announcement: Linearizability: A Typo. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 561–564. doi:10.1145/3465084.3467944

[63] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 92–110. doi:10.1145/2983990.2983999

[64] Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency. *Proc. ACM Program. Lang.* 7, POPL (2023), 1542–1572. doi:10.1145/3571246

[65] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP, Article 100 (Aug. 2022), 29 pages. doi:10.1145/3547631

[66] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[67] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–188.

[68] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. ACM, 377–390. doi:10.1145/2500365.2500600

[69] Aaron Turon and Mitchell Wand. 2011. A separation logic for refining concurrent objects. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 247–258. doi:10.1145/1926385.1926415

[70] Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph. D. Dissertation. University of Cambridge, UK.

[71] Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5403)*. Springer, 335–348. doi:10.1007/978-3-540-93900-9_27

[72] Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 450–464. doi:10.1007/978-3-642-14295-6_40

[73] Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*. Springer, 256–271. doi:10.1007/978-3-540-74407-8_18

[74] Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. 2023. A Compositional Theory of Linearizability. *Proc. ACM Program. Lang.* 7, POPL (2023), 1089–1120. doi:10.1145/3571231

[75] Arthur Oliveira Vale, Zhongye Wang, Yixuan Chen, Peixin You, and Zhong Shao. 2024. Compositionality and Observational Refinement for Linearizability with Crashes. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 2296–2324. doi:10.1145/3689792

[76] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from Meta's Folly library. In *CPP*. ACM, 100–115.

[77] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 2021. Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 407–426. doi:10.1007/978-3-030-90870-6_22

[78] Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A Structural Approach to Prophecy Variables. In *Theory and Applications of Models of Computation - 9th Annual Conference, TAMC 2012, Beijing, China, May 16-21, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7287)*. Springer, 61–71. doi:10.1007/978-3-642-29952-0_12

[79] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 3–19. doi:10.1007/978-3-319-21668-3_1

Received 2026-07-09

A Additional Details about the Proof of Theorem 3.3

This section presents additional technical details about the proof of Theorem 3.3 outlined in §3.

A.1 The prophTrace Predicate

Assertion $\mathsf{prophTrace}_{obj}(\rho, p, \vec{t})$ is defined as

$$\begin{aligned}
\mathsf{prophTrace}_{obj}(\rho, p, \vec{t}) &\triangleq \exists \vec{v}\,.\, \mathsf{proph}(p, \vec{v}) * \mathsf{decValidTr}(obj, \rho, \vec{t}, \mathit{decTr}(\vec{v})) \\
\mathit{decTr}(\texttt{inl}(\tau, x) :: \vec{v}) &\triangleq (\tau, \mathsf{Call}(x)) :: \mathit{decTr}(\vec{v}) \\
\mathit{decTr}(\texttt{inr}(\tau, y) :: \vec{v}) &\triangleq (\tau, \mathsf{Ret}(y)) :: \mathit{decTr}(\vec{v}) \\
\mathit{decTr}(_) &\triangleq \epsilon \\
\mathsf{decValidTr}(obj, \rho, \vec{t}, \vec{t}_{raw}) &\triangleq \mathsf{ValidTr}(obj, \rho, \vec{t}) \\
&\quad \wedge (\vec{t} = \vec{t}_{raw} \vee \exists n < |\vec{t}_{raw}|.\ \vec{t} = \vec{t}_{raw}[0..n-1] \wedge \neg \mathsf{ValidTr}(obj, \rho, \vec{t}_{raw}[0..n])) \\
\mathsf{ValidTr}(obj, \rho, \vec{t}) &\triangleq \exists \rho'.\ \rho \xrightarrow[obj]{\vec{t}}{}^{*}_{\mathsf{adv}} \rho'
\end{aligned}$$

The idea is to decode the untyped trace into a typed trace and discard ill-typed traces. Specifically, the raw resolution list of values is first decoded into a physical trace by function *decTr*, and the trace is then further decoded to a "valid" trace by relation decValidTr. Function *decTr* tries decoding the head of the raw resolution list, and if it fails, it returns an arbitrary dummy value, which is ϵ in this case. Because the program will always resolve the prophecy variable to `inl` or `inr`, this failure will never happen, but we still need to account for the failure case as we have not proved anything about future resolutions yet. The idea for decValidTr is similar. However, because predicate ValidTr is undecidable in general, we cannot write a function to check whether a trace is generated by a valid interaction and discard its invalid part. This relation consists of two parts: $\mathsf{ValidTr}(obj, \rho, \vec{t})$ asserts the result of decoding is a valid trace. The second part says, if $\vec{t}_{raw}$ is fully valid before decoding, then, the whole raw trace will be kept; otherwise, $\vec{t}_{raw}$ must start to become invalid at some point, *i.e.*, there exists an index n, such that the first n elements in $\vec{t}_{raw}$ form a valid trace, but appending one more element makes the trace invalid.

$\mathsf{prophTrace}_{obj}(\rho, p, \vec{t})$ satisfies the following rules:

HtNewProphTrace

$$\{\mathsf{True}\}\ \texttt{newproph}\ \{p.\, \forall obj, \rho.\, \exists\, \vec{t}\,.\, \mathsf{prophTrace}_{obj}(\rho, p, \vec{t})\}$$

HtResolveTrace-Call

$$\frac{\tau \notin \vec{\pi}.1 \qquad \mathit{vr}(x)}{\begin{array}{c}\{\mathsf{prophTrace}_{obj}((\vec{\pi}, \vec{e}, \sigma), p, \vec{t}_1)\} \\ \texttt{resolve}\ p\ \texttt{to}\ \texttt{inl}(\tau, x) \\ \{().\, \exists \vec{t}_2.\, \ulcorner \vec{t}_1 = (\tau, \mathsf{Call}(x)) :: \vec{t}_2 \urcorner * \mathsf{prophTrace}_{obj}((\vec{\pi} +\!\!+ [(\tau, \mathsf{op}(obj, x))], \vec{e}, \sigma), p, \vec{t}_2)\}\end{array}}$$

HtResolveTrace-Ret

$$\frac{y \in \mathit{Val} \qquad \mathsf{NoDup}((\vec{\pi}_1 +\!\!+ (\tau, y) :: \vec{\pi}_2).1)}{\begin{array}{c}\{\mathsf{prophTrace}_{obj}((\vec{\pi}_1 +\!\!+ (\tau, y) :: \vec{\pi}_2, \vec{e}, \sigma), p, \vec{t}_1)\} \\ \texttt{resolve}\ p\ \texttt{to}\ \texttt{inr}(\tau, y) \\ \{().\, \exists \vec{t}_2.\, \ulcorner \vec{t}_1 = (\tau, \mathsf{Ret}(y)) :: \vec{t}_2 \urcorner * \mathsf{prophTrace}_{obj}((\vec{\pi}_1 +\!\!+ (\tau, \bot) :: \vec{\pi}_2, \vec{e}, \sigma), p, \vec{t}_2)\}\end{array}}$$

ResolveTrace-Interm

$$\frac{(e_1, \sigma_1) \to_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)}{\mathsf{prophTrace}_{obj}\big((\vec{\pi}_1 +\!\!+ (\tau, e_1) :: \vec{\pi}_2, \vec{e}, \sigma_1), p, \vec{t}_1\big) \;\Rrightarrow\; \exists \vec{t}_2.\, \ulcorner \vec{t}_2 \sqsubseteq_{\mathsf{pre}} \vec{t}_1 \urcorner * \mathsf{prophTrace}_{obj}\big((\vec{\pi}_1 +\!\!+ (\tau, e_2) :: \vec{\pi}_2, \vec{e} +\!\!+ \vec{e}_f, \sigma_2), p, \vec{t}_2\big)}{}^{*}$$

ResolveTrace-Child

$$\frac{(e_1, \sigma_1) \to_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)}{\mathsf{prophTrace}_{obj}\big((\vec{\pi}, \vec{e}_a +\!\!+ e_1 :: \vec{e}_b, \sigma_1), p, \vec{t}_1\big) \;\Rrightarrow\; \exists \vec{t}_2.\, \ulcorner \vec{t}_2 \sqsubseteq_{\mathsf{pre}} \vec{t}_1 \urcorner * \mathsf{prophTrace}_{obj}\big((\vec{\pi}, \vec{e}_a +\!\!+ e_2 :: \vec{e}_b +\!\!+ \vec{e}_f, \sigma_2), p, \vec{t}_2\big)}{}^{*}$$

Rule HtNewProphTrace creates a fresh typed prophecy variable p. This rule allows us to associate p with any object at any time after p was created. A critical step to prove this rule is to show that for any trace $\vec{t}_{raw}$, there exists a trace $\vec{t}$ such that decValidTr($obj, \rho, \vec{t}, \vec{t}_{raw}$). To do so, we consider each prefix of $\vec{t}_{raw}$: $\epsilon, \vec{t}_{raw}[0..0], \vec{t}_{raw}[0..1], \ldots, \vec{t}_{raw}$, and find the last prefix still satisfying ValidTr. Notably, this requires using the law of excluded middle because given a program from a Turing-complete language, deciding whether it can generate a trace is equivalent to the halting problem [61].

Rules HtResolveTrace-Call and HtResolveTrace-Ret handle the emission of physical events. To emit a Call/Ret event, one must show it is valid to perform this event on the current configuration. These rules then resolve the prophecy variable and meanwhile update the configuration. In the premise of HtResolveTrace-Ret, NoDup means no duplication, *i.e.*, a list does not contain the same item twice.

Rules ResolveTrace-Interm and ResolveTrace-Child handle the evaluation of the configuration. Since there is no event for intermediate steps, these rules only update the configuration but do not resolve the prophecy variable. Note that performing an intermediate step could truncate the trace: Relation decValidTr implies that $\vec{t}_1$ is the *longest* possible trace emitted by evaluating the current configuration, which requires the current configuration to take a specific first step, but if the prim step used by the rules is not the expected first step, then the new configuration would not be able to emit a trace as long as $\vec{t}_1$. For example, program "`if NondetBool() then op(obj, 1) else ()`" can emit a trace of $[(\tau, \mathsf{Call}(1))]$ under all possible executions, but can emit ϵ if the first prim step is $(K[\texttt{NondetBool}], \sigma) \to_{\mathsf{prim}} (K[\texttt{false}], \sigma, \epsilon)$. This explains why $\vec{t}_2$ can only be a prefix of $\vec{t}_1$.

A.2 The $I_{\mathbf{atom}}$ Invariant

A.2.1 The Exclusive Authoritative Ghost State. The actual invariant does not use a ghost map to track the abstract sequential state of the object, but rather uses a more restrictive resource $\boxed{\bullet\mathsf{ex}(s)}^{\gamma_s}$. For the definition of PState_μ, we actually use $\boxed{\circ\mathsf{ex}(s)}^{\gamma_s}$. The exclusive authoritative ghost resource satisfies the rules below.

ExAuthAlloc

$$\vdash \Rrightarrow \exists \gamma.\, \boxed{\bullet\mathsf{ex}(x)}^{\gamma} * \boxed{\circ\mathsf{ex}(x)}^{\gamma}$$

ExAuthAgree

$$\boxed{\bullet\mathsf{ex}(x)}^{\gamma} * \boxed{\circ\mathsf{ex}(y)}^{\gamma} \vdash \ulcorner x = y \urcorner$$

ExAuthUpdate

$$\boxed{\bullet\mathsf{ex}(x)}^{\gamma} * \boxed{\circ\mathsf{ex}(y)}^{\gamma} \vdash \Rrightarrow \boxed{\bullet\mathsf{ex}(z)}^{\gamma} * \boxed{\circ\mathsf{ex}(z)}^{\gamma}$$

A.2.2 Definition and Properties of pendingOp. To compute $\mathit{pendingOp}(\vec{t}')$, let's consider a tag τ of some event that has happened or will happen. If $(\tau, \mathsf{Call}(\_)) \in \vec{t}'$, then operation τ has not been called yet, so it is definitely not pending. If $(\tau, \mathsf{Call}(\_)) \notin \vec{t}'$ but $(\tau, \mathsf{Lin}(\_, \_)) \in \vec{t}'$ or $(\tau, \mathsf{Ret}(\_)) \in \vec{t}'$, then operation τ must have been called before the current time, and has not yet returned, so it is definitely pending. Finally, if no event in $\vec{t}'$ has tag τ, then τ may have returned before the current time, or τ may have been called, but will never return. The latter is possible because we work on

partial correctness, so an operation is permitted to loop infinitely. As a result, in the last case, we cannot determine whether τ is pending or not, and $\mathit{pendingOp}(\vec{t}')$ will conservatively exclude τ in its result. This also explains why the invariant uses $\mathit{pendingOp}(\vec{t}') \subseteq \mathrm{dom}(\Phi s)$ rather than strict equality: because function $\mathit{pendingOp}$ is an under-approximation, some pending operations can be missing in $\mathit{pendingOp}(\vec{t}')$.

Remark A.1. $\mathsf{well\text{-}formed}_{\mathsf{lin}}(\vec{t}') \implies \mathit{pendingOp}(\vec{t}') = \varnothing$

Remark A.2. $\mathit{pendingOp}(\vec{t}') \subseteq \{\tau\} \cup \mathit{pendingOp}((\tau, \mathsf{Call}(x)) :: \vec{t}')$

Remark A.3. $\mathit{pendingOp}(\vec{t}') \subseteq \mathit{pendingOp}((\tau, \mathsf{Lin}(x, y)) :: \vec{t}')$

Remark A.4. $\mathit{pendingOp}(\vec{t}') \subseteq \mathit{pendingOp}((\tau, \mathsf{Ret}(y)) :: \vec{t}') \setminus \{\tau\}$

A.2.3 Properties of Partial Linearizability. The following properties of $\mathsf{partial\text{-}well\text{-}formed}_{\mathsf{lin}}$ and cfg-safe (defined in §3.3) are used in the proof of Theorem 3.3.

Remark A.5. $\mathsf{partial\text{-}well\text{-}formed}_{\mathsf{lin}}(\vec{t}') \land \vec{t}'' \sqsubseteq_{\mathsf{suf}} \vec{t}' \implies \mathsf{partial\text{-}well\text{-}formed}_{\mathsf{lin}}(\vec{t}'')$

Remark A.6. $\mathsf{lin}_\mu(\mathtt{init}, \mathsf{op}) \land ([\mathtt{init}()], \varnothing) \to^*_{\mathsf{tp}} ([obj], \sigma) \implies \mathsf{cfg\text{-}safe}(\epsilon, \epsilon, \sigma)$.

Remark A.7. $\mathsf{cfg\text{-}safe}(\rho) \land \rho \xrightarrow{\vec{t}}{}^*_{\mathsf{adv}} \rho' \implies \mathsf{cfg\text{-}safe}(\rho')$

A.3 Detailed Proof of Theorem 3.3

Part of the proof is stated in terms of the wp assertion. The connection between wp and Hoare triple is

$$\{P\}\ e\ \{v.\ Q\} \triangleq \Box\left(\forall \Phi.\ P \mathrel{-\!\!*} \triangleright(\forall v.\ Q \mathrel{-\!\!*} \Phi(v)) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\}\right)$$

The proof consists of four parts.

A.3.1 Specification of Wrapped `init`. We first prove specification (3):

$$\{\mathsf{True}\}\ \mathit{wrapi}(\mathtt{init})()\ \left\{obj.\ \exists \gamma.\ \mathsf{IsP}_\mu(\gamma, obj) * \mathsf{PState}_\mu(\gamma, s_0)\right\}$$

Observe that Clause (1) of Definition 2.8 can be strengthened to $\mathsf{safe}_{vo}(\mathtt{init}(), \varnothing)$ where $vo(v, \sigma) \triangleq ([\mathtt{init}()], \varnothing) \to^*_{\mathsf{tp}} ([v], \sigma)$. Using Theorem 34 by Hostert et al. [36], we obtain a specification for `init`:

$$\{\mathsf{True}\}\ \mathtt{init}()\ \left\{obj.\ \exists \sigma.\ \mathsf{S}_\circ(\sigma) * \ulcorner vo(obj, \sigma)\urcorner\right\}$$

Using this specification and HtNewProphTrace, and picking the initial configuration as $(\epsilon, \epsilon, \sigma)$, we obtain

$$\mathsf{S}_\circ(\sigma) * \ulcorner([\mathtt{init}()], \varnothing) \to^*_{\mathsf{tp}} ([obj], \sigma)\urcorner * \mathsf{prophTrace}_{obj}\big((\epsilon, \epsilon, \sigma), p, \vec{t}\big)$$

and we are obligated to prove

$$\exists \gamma.\ \mathsf{IsP}_\mu(\gamma, (obj, p)) * \mathsf{PState}_\mu(\gamma, s_0)$$

which is equivalent to

$$\exists \gamma, \gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s.\ \boxed{\Box(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma} * \boxed{\circ \mathsf{ex}(s_0)}^{\gamma_s} * I_{\mathsf{atom}_\mu}(\gamma, obj, p)$$

The ownership of the typed prophecy variable $\mathsf{prophTrace}_{obj}\big((\epsilon, \epsilon, \sigma), p, \vec{t}\big)$ guarantees $(\epsilon, \epsilon, \sigma) \xrightarrow{\vec{t}}{}^*_{\mathsf{adv}} \rho'$ for some ρ'. Further because $([\mathtt{init}()], \varnothing) \to^*_{\mathsf{tp}} ([obj], \sigma)$, by $\mathsf{lin}_\mu(\mathtt{init}, \mathsf{op})$, we have $\mathsf{trlin}_\mu(\vec{t})$. Namely, there exists an annotated trace $\vec{t}'$, such that $\vec{t} = \pi_\Sigma(\vec{t}')$, $\mathsf{well\text{-}formed}_{\mathsf{lin}}(\vec{t}')$, and $\exists s'.\ \mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t}'), s_0, s')$. By Remark A.6, $\mathsf{cfg\text{-}safe}(\epsilon, \epsilon, \sigma)$ holds.

Allocate the initial ghost resources as such:

$$\boxed{\bullet\epsilon}^{\gamma_\pi} * \boxed{\bullet\epsilon}^{\gamma_e} * \boxed{\bullet\varnothing}^{\gamma_\phi} * \boxed{\bullet \mathsf{ex}(s_0)}^{\gamma_s} * \boxed{\circ \mathsf{ex}(s_0)}^{\gamma_s} * \boxed{\Box(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma}$$

It is then straightforward to establish I_{atom} for this initial state: the operation lists and receipt map are empty, so (P_2)–(P_4) hold vacuously; (P_5) follows from well-formed$_{\text{lin}}(\vec{t}')$ and the cfg-safe fact just established; and the allocated ghost resources together with prophTrace prove (P_1).

This concludes the proof of the specification for *wrapi*(`init`).

A.3.2 The Call of an Operation. We now turn to the proof of specification (4):

$$\ulcorner vr(x) \urcorner * \mathsf{IsP}_\mu(\gamma, obj) \vdash \langle \mathsf{PState}_\mu(\gamma, s) \rangle\ wrapo(\mathsf{op})(objp, x)\ \langle y.\ \exists s'.\ \mathsf{PState}_\mu(\gamma, s') * \ulcorner rr(s, x, s', y) \urcorner \rangle$$

By the definition of logically atomic triples, this is equivalent to proving $\mathsf{AU}\ldots(\Phi) \twoheadrightarrow \mathsf{wp}\ wrapo(\mathsf{op})(objp, x)\ \{\Phi\}$ for an arbitrary Φ under the hypotheses that $vr(x)$ and $\boxed{I_{\text{atom}_\mu}(\gamma, obj, p)}$.

We first use HtNewGhostID to allocate a fresh token for this operation and obtain resource token (τ). We now need to verify `resolve` p `to inl`(τ, x). This requires opening the invariant to register the new operation.

By repeatedly using TokenNe for each component in (P_3), we have $\tau \notin \vec{\pi}.1$.

Using HtResolveTrace-Call, we update $\mathsf{prophTrace}(\rho, p, \pi_\Sigma(\vec{t}'))$ to $\mathsf{prophTrace}(\rho', p, \vec{u})$, where $\rho' = (\vec{\pi} +\!\!+ [(\tau, \mathsf{op}(obj, x))], \vec{e}, \sigma)$ and $\pi_\Sigma(\vec{t}') = (\tau, \mathsf{Call}(x)) :: \vec{u}$. Split $\vec{t}'$ into three parts: $\vec{t}' = \vec{t}_{\,\mathsf{lin}} +\!\!+ (\tau, \mathsf{Call}(x)) :: \vec{t}''$, such that $\vec{u} = \pi_\Sigma(\vec{t}'')$. Because $(\tau, \mathsf{Call}(x))$ is the first physical event in $\vec{t}'$, $\vec{t}_{\,\mathsf{lin}}$ only consists of Lin events.

Update ghost state in the invariant:

$$\boxed{\bullet l2m(activeOp(\vec{\pi}))}^{\gamma_\pi} \Rrightarrow \boxed{\bullet l2m(activeOp(\vec{\pi} +\!\!+ [(\tau, \mathsf{op}(obj, x))]))}^{\gamma_\pi} * \tau \hookrightarrow^{\gamma_\pi} \mathsf{op}(obj, x)$$
$$\boxed{\bullet \Phi s}^{\gamma_\phi} \Rrightarrow \boxed{\bullet (\Phi s[\tau \leftarrow \Phi])}^{\gamma_\phi} * \tau \hookrightarrow^{\gamma_\phi} \Phi$$

Note that $\boxed{\bullet \vec{e}}^{\gamma_e}$ and $\boxed{\bullet \mathsf{ex}(s)}^{\gamma_s}$ stay unchanged. The former is unchanged because we are only registering a new operation, which should not affect existing child threads. For the latter, we will update the abstract state of the object later when linearizing operations in $\vec{t}_{\mathsf{lin}}$.

Let's try to close the invariant and see what is left: Most of (P_1) can be proved by the new states, except for $\boxed{\bullet \mathsf{ex}(s)}^{\gamma_s}$. Part ($P_3$) holds by inserting the new token token (τ). Part (P_4) holds by Remark A.2. Part (P_5) is proved by Remarks A.5 and A.7. The only part left is (P_2), which requires linearizing operations in $\vec{t}_{\mathsf{lin}}$ using Lemma A.8 below and inserting the $\mathsf{AU}\ldots(\Phi)$ in the context into the big separation conjunction.

Lemma A.8 (Helping). *Let $\vec{t}_{\mathsf{lin}}$ be a trace of linearization events, $\vec{t}''$ be an annotated trace, and Φs be a finite map from tags to iProp's, such that* partial-well-formed$_{\mathsf{lin}}(\vec{t}_{\mathsf{lin}} +\!\!+ \vec{t}'')$, $\mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t}_{\mathsf{lin}} +\!\!+ \vec{t}''), s, s_{end})$, *and* $\vec{t}_{\mathsf{lin}}.1 \subseteq \mathsf{dom}(\Phi s)$, *then*

$$\frac{\boxed{\square(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma} \qquad \boxed{\bullet \mathsf{ex}(s)}^{\gamma_s} \qquad \mathop{\Large *}_{\tau \leftarrow \Phi \in \Phi s} interpret(\vec{t}_{\mathsf{lin}} +\!\!+ \vec{t}'', \tau, \Phi)}{\Rrightarrow \exists s'.\ \ulcorner \mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t}''), s', s_{end}) \urcorner * \boxed{\bullet \mathsf{ex}(s')}^{\gamma_s} * \mathop{\Large *}_{\tau \leftarrow \Phi \in \Phi s} interpret(\vec{t}'', \tau, \Phi)}\ *$$

Proof. Induction on list $\vec{t}_{\mathsf{lin}}$. The base case is trivial: if $\vec{t}_{\mathsf{lin}} = \epsilon$, then we don't need to update anything.

For the inductive case: suppose $\vec{t}_{\mathsf{lin}} = (\tau, \mathsf{Lin}(x, y)) :: \vec{t}'_{\mathsf{lin}}$. To linearize the first event $(\tau, \mathsf{Lin}(x, y))$, we need to prove the goal below by firing the AU

$$\frac{\boxed{\bullet \mathsf{ex}(s)}^{\gamma_s} \qquad \mathsf{AU}_{\mathsf{PState}_\mu(\gamma, s),\, y.\ \exists s'.\ \mathsf{PState}_\mu(\gamma, s') * \ulcorner rr(s, x, s', y) \urcorner}(\Phi)}{\Rrightarrow \exists s'.\ \ulcorner \mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t}'_{\mathsf{lin}} +\!\!+ \vec{t}''), s', s_{end}) \urcorner * \boxed{\bullet \mathsf{ex}(s')}^{\gamma_s} * \Phi(y)}\ *$$

To fire the AU, we first obtain $\boxed{\circ\mathsf{ex}(s)}^{\gamma_s}$ from the precondition of the AU. Because $\mathsf{sound}_\mu((\tau, \mathsf{Lin}(x,y)) :: \pi_{\mathsf{lin}}(\vec{t}'_{\mathsf{lin}} +\!\!+ \vec{t}''), s, s_{end})$, by Remark 2.5, there exists a new state s' and a response y such that $vr(x)$, $rr(s,x,s',y)$, and $\mathsf{sound}_\mu(\pi_{\mathsf{lin}}(\vec{t}'_{\mathsf{lin}} +\!\!+ \vec{t}''), s', s_{end})$.

Update the ghost state: $\boxed{\bullet\mathsf{ex}(s)}^{\gamma_s} * \boxed{\circ\mathsf{ex}(s)}^{\gamma_s} \Rrightarrow \boxed{\bullet\mathsf{ex}(s')}^{\gamma_s} * \boxed{\circ\mathsf{ex}(s')}^{\gamma_s}$.

Next, we supply $\boxed{\circ\mathsf{ex}(s')}^{\gamma_s}$ and $\ulcorner rr(s,x,s',y)\urcorner$ in exchange for the receipt $\Phi(y)$. This completes the goal for the first event.

Using the induction hypothesis, we linearize remaining events in $\vec{t}'_{\mathsf{lin}}$, and conclude the proof. □

After applying Lemma A.8 and inserting the AU for the new operation, we can conclude (P_2) and use the new state $\boxed{\bullet\mathsf{ex}(s')}^{\gamma_s}$ to complete (P_1).

After closing the invariant, we still own $\tau \hookrightarrow^{\gamma_\pi} \mathsf{op}(obj, x)$ and $\tau \hookrightarrow^{\gamma_\phi} \Phi$.

A.3.3 Intermediate Steps of the Operation. The next step is to verify $\mathsf{op}(obj, x)$. Using an argument in the flavor of Theorem 17 by Hostert et al. [36], we reduce the goal to

$$\forall e.\ \tau \hookrightarrow^{\gamma_\pi} e \twoheadrightarrow \tau \hookrightarrow^{\gamma_\phi} \Phi \twoheadrightarrow \mathsf{wp}\ e\ \{v.\ \tau \hookrightarrow^{\gamma_\pi} v * \tau \hookrightarrow^{\gamma_\phi} \Phi\}$$

and use Löb induction to prove this goal. To make progress, we do case analysis on whether $e \in \mathit{Val}$. The postcondition is trivially true if e is a value, and for the non-value case, we apply a stronger invariant access rule [36, Lemma 15]:

InvAccMaybe

$$\frac{{}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2}\Big(\big(\exists K, e'.\ \ulcorner e = K[e']\urcorner \wedge \mathsf{atomic}(e')\big) * \mathsf{wp}_{\mathcal{E}_2}\, e'\ \big\{v.\ {}_{\mathcal{E}_2}\Rrightarrow_{\mathcal{E}_1} \mathsf{wp}_{\mathcal{E}_1}\, K[v]\ \{\Phi\}\big\}\Big) \vee {}_{\mathcal{E}_2}\Rrightarrow_{\mathcal{E}_1} \mathsf{wp}_{\mathcal{E}_1}\, e\ \{\Phi\}\Big)}{\mathsf{wp}_{\mathcal{E}_1}\, e\ \{\Phi\}} *$$

While the standard InvAccPhysical rule only allows opening an invariant when the expression is atomic, this rule allows us to open an invariant for any expression and decide whether the expression is atomic *afterwards*. We have two choices after opening the invariant: either prove the first redex is atomic and close the invariant after that atomic step, or close the invariant immediately. With the help of InvAccMaybe, we can open the I_{atom} invariant and get the latest configuration $\rho = (\vec{\pi}_1 +\!\!+ (\tau, e_1) :: \vec{\pi}_2, \vec{e}, \sigma_1)$. Next, we want to use the language-independent completeness condition [36, Condition 18] to reduce e for one step.

LangCompleteness

$$\ulcorner \mathsf{red}(e_1, \sigma_1)\urcorner * \mathrm{S}_\circ(\sigma_1) \vdash \Rrightarrow_{\mathcal{E}} \Big(\big(\exists K, e'_1.\ \ulcorner e_1 = K[e'_1]\urcorner \wedge \mathsf{atomic}(e'_1)\big) * \forall \Phi.\ \triangleright C_1 \twoheadrightarrow \mathsf{wp}_{\mathcal{E}}\, e'_1\ \{\Phi\}\Big) \vee \mathrm{S}_\circ(\sigma_1) * \forall \Phi.\ \triangleright C_2 \twoheadrightarrow \mathsf{wp}\ e_1\ \{\Phi\}\Big)$$

where

$$C_1 \triangleq \forall \vec{\kappa}, v_2, \sigma_2, \vec{e}_f.\ \ulcorner (e'_1, \sigma_1) \xrightarrow{\vec{\kappa}}_{\mathsf{base}} (v_2, \sigma_2, \vec{e}_f)\urcorner \Rrightarrow (\mathrm{S}_\circ(\sigma_2) \twoheadrightarrow \Phi(v_2)) * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}\ e_f\ \{\_.\ \mathsf{True}\}$$

$$C_2 \triangleq \forall e_2, \vec{e}_f.\ \Big(\forall \sigma'_1.\ \mathrm{S}_\circ(\sigma'_1) \Rrightarrow_{\mathcal{E}} \exists \vec{\kappa}, \sigma_2.\ \ulcorner (e_1, \sigma'_1) \xrightarrow{\vec{\kappa}}_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)\urcorner * \mathrm{S}_\circ(\sigma_2)\Big) \Rrightarrow_{\top} \mathsf{wp}\ e_2\ \{\Phi\} * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}\ e_f\ \{\_.\ \mathsf{True}\}$$

To use this condition, we need to prove $\mathsf{red}(e_1, \sigma_1)$ and provide the ownership of the current program state $\mathrm{S}_\circ(\sigma_1)$. The former holds because $\text{cfg-safe}(\rho)$, and the latter comes from the invariant. Applying this condition leaves two cases.

For case C_1, we know the first redex of e_1 is atomic, so we can keep the invariant open for one step. In fact, $\forall\Phi.\ \triangleright C_1 \twoheadrightarrow \mathsf{wp}_{\mathcal{E}}\ e_1'\ \{\Phi\}$ from the condition is the exact rule needed to justify that step. After that step, we are obligated to prove[7]

$$\left(\mathrm{S}_\circ(\sigma_2) \twoheadrightarrow {}_{\top\setminus\mathcal{N}}\Rightarrow_{\top} \mathsf{wp}\ K[v_2]\ \{\Phi\}\right) * \mathop{\Large *}_{e_f\in\vec{e}_f} \mathsf{wp}\ e_f\ \{\_.\ \mathsf{True}\}$$

under the assumption that $(e_1', \sigma_1) \xrightarrow{\vec{\kappa}}_{\mathsf{base}} (v_2, \sigma_2, \vec{e}_f)$.

Using ResolveTrace-Interm, we update $\mathsf{prophTrace}\big((\vec{\pi}_1 +\!\!+ (\tau, e_1) :: \vec{\pi}_2, \vec{e}, \sigma_1), p, \pi_\Sigma(\vec{t}')\big)$ to $\mathsf{prophTrace}\big((\vec{\pi}_1 +\!\!+ (\tau, K[v_2]) :: \vec{\pi}_2, \vec{e} +\!\!+ \vec{e}_f, \sigma_2), p, \pi_\Sigma(\vec{t}'')\big)$, where $\vec{t}'' \sqsubseteq_{\mathsf{pre}} \vec{t}'$. We then update the ghost state γ_π and γ_e accordingly; this includes allocating new ghost points-tos for each thread in $\vec{e}_f$:

$$\mathop{\Large *}_{i=0}^{|\vec{e}_f|-1} (|\vec{e}| + i) \hookrightarrow^{\gamma_e} \vec{e}_f(i)$$

We now split the goal into two parts: $\mathrm{S}_\circ(\sigma_2) \twoheadrightarrow {}_{\top\setminus\mathcal{N}}\Rightarrow_{\top} \mathsf{wp}\cdots$ and the iterated separating conjunction of wp's over $\vec{e}_f$.

For part 1, we obtain the ownership of the new state after that atomic step, and we need to close the invariant. Most parts are mechanical, and the only interesting part is (P_2), for which we need to prove $\mathit{interpret}(\vec{t}', \tau, \Phi) \vdash \mathit{interpret}(\vec{t}'', \tau, \Phi)$. If $\pi_\tau(\vec{t}'')$ is not nil, because $\vec{t}'' \sqsubseteq_{\mathsf{pre}} \vec{t}'$, $\pi_\tau(\vec{t}')$ must also be non-nil and have the same head as $\pi_\tau(\vec{t}'')$, which implies $\mathit{interpret}(\vec{t}'', \tau, \Phi) = \mathit{interpret}(\vec{t}', \tau, \Phi)$. Otherwise, if $\pi_\tau(\vec{t}'')$ is nil, then $\mathit{interpret}(\vec{t}'', \tau, \Phi)$ is trivially true.

After closing the invariant, we can use the Löb induction hypothesis to finish part 1.

Part 2 is proved by Lemma A.9 below.

For case C_2, we do not know whether the first redex of e_1 is atomic, so we have to immediately close the invariant. As a result, we lose all the knowledge learned from the invariant, including the current configuration ρ and the ownership of the current state $\mathrm{S}_\circ(\sigma_1)$. Fortunately, with the help of $\forall\Phi.\ \triangleright C_2 \twoheadrightarrow \mathsf{wp}\ e_1\ \{\Phi\}$, we can still take one step with the invariant closed. To prove C_2, we need to prove

$$\mathsf{wp}\ e_2\ \{\Phi\} * \mathop{\Large *}_{e_f\in\vec{e}_f} \mathsf{wp}\ e_f\ \{\_.\ \mathsf{True}\}$$

under the assumption that

$$\forall\sigma_1'.\ \mathrm{S}_\circ(\sigma_1') \Rrightarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}} \exists\,\vec{\kappa}, \sigma_2.\ \ulcorner(e_1, \sigma_1') \xrightarrow{\vec{\kappa}}_{\mathsf{prim}} (e_2, \sigma_2, \vec{e}_f)\urcorner * \mathrm{S}_\circ(\sigma_2)$$

This is a rather strong assumption because it allows us to take one step from e_1 under *any* state.

To use this assumption, we open the invariant again and get a (possibly different) configuration $\rho' = (\vec{\pi}_1 +\!\!+ (\tau, e_1) :: \vec{\pi}_2, \vec{e}, \sigma_1')$ and the ownership of $\mathrm{S}_\circ(\sigma_1')$. The remaining proof for this case is identical to case C_1, which also uses Lemma A.9 to discharge the goal for threads in $\vec{e}_f$.

Lemma A.9. $\boxed{I_{\mathsf{atom}_\mu}(\gamma, \mathit{obj}, p)} * \boxed{\square(\gamma_\pi, \gamma_e, \gamma_\phi, \gamma_s)}^{\gamma} \vdash i \hookrightarrow^{\gamma_e} e \twoheadrightarrow \mathsf{wp}\ e\ \{\_.\ \mathsf{True}\}$

Proof Sketch. Similar to Appendix A.3.3, the proof is based on Löb induction, but this time, we use ResolveTrace-Child to update the prophTrace resource. □

[7] In the goal, assume $\top \setminus \mathcal{N}$ is the mask after opening the I_{atom} invariant.

A.3.4 The Return of the Operation. As the final step, we need to verify $\texttt{resolve}\ p\ \texttt{to}\ \texttt{inr}(\tau, y)$ under the precondition that $\tau \hookrightarrow\!\!\!\!\rightarrow^{\gamma_\pi} y$ and $\tau \hookrightarrow\!\!\!\!\rightarrow^{\gamma_\phi} \Phi$. The proof is symmetric to Appendix A.3.2: we open the invariant, use HtResolveTrace-Ret to update the prophecy variable, split the new trace into $\vec{t}_{\mathsf{lin}}$ and $\vec{t}''$, update ghost states, use Lemma A.8 to linearize operations in $\vec{t}_{\mathsf{lin}}$, and close the invariant. But this time, for (P_2), we extract the receipt $\Phi'(y)$ from the iterated separating conjunction, where $\Phi' = \Phi s(\tau)$. Since we still own the fragment $\tau \hookrightarrow\!\!\!\!\rightarrow^{\gamma_\phi} \Phi$ recording this operation's receipt type, its agreement with the authoritative receipt map $\boxed{\bullet \Phi s}^{\gamma_\phi}$ yields $\triangleright(\Phi \equiv \Phi')$.

We are now left with proving $\triangleright(\Phi \equiv \Phi') * \Phi'(y) \vdash \mathsf{wp}\ ();y\ \{\Phi\}$. Take a pure step and apply HtValue, and we get $(\Phi \equiv \Phi') * \Phi'(y) \vdash \Phi(y)$, which is trivially true.

This concludes the proof of the specification for *wrapo*(op).

B Details about the Most General Client Construction

This appendix gives additional details about the proofs involving the most general client (MGC) described in §4 and §5.1.

The Trace Library. The MGC uses a trace library to record a trace at location ℓ_{tr}. The library comes with two functions satisfying the specifications below.

AhtEmit

$$\frac{\mathsf{traceInv}(\ell_{tr}, I)}{\langle \forall\, \vec{t}\,.\, \mathsf{traces}(\ell_{tr}, \vec{t}) * \ulcorner I(\vec{t} +\!\!+ [(\tau, u)]) \urcorner \rangle\ \ \texttt{emit}(\ell_{tr}, \tau, u)\ \ \langle ().\, \mathsf{traces}(\ell_{tr}, \vec{t} +\!\!+ [(\tau, u)]) \rangle}$$

AhtFresh

$$\frac{\mathsf{traceInv}(\ell_{tr}, I)}{\langle \forall\, \vec{t}\,.\, \mathsf{traces}(\ell_{tr}, \vec{t}) * \ulcorner \forall \tau \notin \vec{t}\,.\, 1 \Rightarrow I(\vec{t} +\!\!+ [(\tau, u)]) \urcorner \rangle\ \ \texttt{fresh}(\ell_{tr}, u)\ \ \langle \tau.\, \mathsf{traces}(\ell_{tr}, \vec{t} +\!\!+ [(\tau, u)]) * \ulcorner \tau \notin \vec{t}\,.1 \urcorner \rangle}$$

Assertion $\mathsf{traces}(\ell_{tr}, \vec{t})$ says the trace currently stored at ℓ_{tr} is $\vec{t}$, and persistent assertion $\mathsf{traceInv}(\ell_{tr}, I)$ says the trace always satisfies a predicate $I\colon \Sigma^* \to \mathit{Prop}$. The user of the library is free to pick the event type Σ, provided each event is a pair of a tag τ and a value. Function `emit` appends a new event with tag τ to the trace, and `fresh` (demonically) non-deterministically picks a fresh tag τ *w.r.t.* the existing trace, appends (τ, u) to the trace and returns the tag. The two operations are logically atomic.

The Most General Client. To construct the MGC, we first define an auxiliary function *wrapcr* to record the call-return trace

$$\mathit{wrapcr}(\mathsf{op}, \mathit{obj}, x) \triangleq \texttt{let}\ \tau := \texttt{fresh}(\ell_{tr}, \texttt{inl}(x))\ \texttt{in}\ \texttt{let}\ y := \mathsf{op}(\mathit{obj}, x)\ \texttt{in}\ \texttt{emit}(\ell_{tr}, \tau, \texttt{inr}(y)); y$$

It is similar to *wrapo* in Figure 4, but instead of resolving a prophecy variable, it records the trace using the trace library. Here location ℓ_{tr} is an external parameter that is allocated at meta-level. Unlike *wrapo*, *wrapcr* takes the operation op as an object-level parameter, so that the same wrapper can be applied to both sides of a refinement, as in the proof of Lemma 4.2.

To define *MGC*, we first define a meta-level fixed point $\mathit{concCalls}(\overrightarrow{req}, \mathsf{op}, \mathit{obj}) : \mathit{Val}^* \times \mathit{Expr} \times \mathit{Expr} \to \mathit{Expr}$, and the *MGC* itself is a thin wrapper over *concCalls*

$$\begin{aligned} \mathit{concCalls}(\epsilon, \mathsf{op}, \mathit{obj}) &\triangleq !\ell_{tr} \\ \mathit{concCalls}(r :: \overrightarrow{req}, \mathsf{op}, \mathit{obj}) &\triangleq \texttt{fork}\,\{\mathit{wrapcr}(\mathsf{op}, \mathit{obj}, r); ()\}; \mathit{concCalls}(\overrightarrow{req}, \mathsf{op}, \mathit{obj}) \\ \mathit{MGC}(\vec{t}) &\triangleq \lambda \mathsf{op}, \mathit{obj}.\ \mathit{concCalls}(\mathit{extractRequests}(\vec{t}), \mathsf{op}, \mathit{obj}) \end{aligned}$$

The MGC spawns one thread for each operation in $\vec{t}$, where each thread calls *wrapcr* with the parameter extracted from $\vec{t}$, and the main thread of the MGC returns the trace stored at ℓ_{tr}, making the emitted events externally observable.

Supporting Lemmas. The trace-agnostic specification trace-spec used in Theorem 5.1 is defined as follows.

$\text{trace-spec}_I(\text{MgcInv}, vr, \texttt{init}, \texttt{op}) \triangleq$

$$\{\text{True}\}\ \texttt{init}()\ \{obj.\ \exists \gamma.\ \forall \ell_{tr}.\ \text{traceIs}(\ell_{tr}, \epsilon) \twoheadrightarrow \Rrightarrow \text{MgcInv}(\ell_{tr}, \gamma, obj)\} \tag{6}$$

$$\ulcorner vr(x) \urcorner \vdash \{\text{traceInv}(\ell_{tr}, I) * \text{MgcInv}(\ell_{tr}, \gamma, obj)\}\ wrapcr(\texttt{op}, obj, x)\ \{\_.\ \text{True}\} \tag{7}$$

$$\text{persistent}(\text{MgcInv}) \tag{8}$$

Although the MGC is parameterized by a trace, its specification follows from this trace-agnostic condition.

Lemma B.1. *If* $\text{trace-spec}_I(\text{MgcInv}, vr, \texttt{init}, \texttt{op})$, *then* $\text{mgc-spec}_I(vr, \vec{t}, \texttt{init}, \texttt{op})$, *where* $\text{mgc-spec}_I(vr, \vec{t}, \texttt{init}, \texttt{op}) \triangleq$

$$\ulcorner \forall (\tau, \text{Call}(x)) \in \vec{t}\ .\ vr(x) \urcorner \vdash \{\text{traceInv}(\ell_{tr}, I) * \text{traceIs}(\ell_{tr}, \epsilon)\}\ MGC(\vec{t})(\texttt{op}, \texttt{init}())\ \left\{\vec{u}\ .\ \ulcorner I(\vec{u}) \urcorner\right\}$$

The following lemma ensures that the MGC construction covers all possible adversarial steps.

Lemma B.2 (MGC Completeness). *Given location* ℓ_{tr}, *if* $(\epsilon, \epsilon, \sigma) \xrightarrow[obj.\texttt{op}/vr]{\vec{t}}{}^{*}_{\text{adv}}\ (\vec{\pi}', \vec{e}', \sigma')$ *and* $\ell_{tr} \notin \text{dom}(\sigma')$, *then* $([MGC(\vec{t})(\texttt{op}, obj)], compose(\sigma, \epsilon)) \rightarrow^{*}_{\text{tp}}\ (\vec{t} :: \ll \vec{\pi}' \gg +\!\!+ \vec{e}', compose(\sigma', \vec{t}))$, *where* $compose(\sigma, \vec{t}) \triangleq \sigma\left[\ell_{tr} \leftarrow \vec{t}\right]$ *and* $\ll \vec{\pi} \gg$ *is a function to translate threads in the* $\rightarrow^{*}_{\text{adv}}$ *relation to the threads spawned by the MGC.*[8]

Symmetrically, every value returned by the MGC must be a trace generated by some adversarial steps.

Lemma B.3 (MGC Soundness). *Given location* ℓ_{tr}, *if* $([MGC(\vec{t}_0)(\texttt{op}, \texttt{init}())], compose(\sigma_0, \epsilon)) \rightarrow^{*}_{\text{tp}} (v :: \_, \_)$ $(v \in Val)$, $\ell_{tr} \notin \text{dom}(\sigma_0)$, *and the technical conditions below hold, then* $v \in \Sigma^*$ *and there exists* $obj \in Val$ *and* ρ' *such that* $([\texttt{init}()], \sigma_0) \rightarrow^{*}_{\text{tp}} ([obj], \sigma)$ *and* $(\epsilon, \epsilon, \sigma) \xrightarrow[obj.\texttt{op}/vr]{v}{}^{*}_{\text{adv}}\ \rho'$. *The technical conditions are*

- $\forall \sigma.\ \text{safe}(\texttt{init}(), \sigma) \wedge \text{seq}(\texttt{init}(), \sigma)$,
- $\forall \sigma_0, obj, \sigma.\ ([\texttt{init}()], \sigma_0) \rightarrow^{*}_{\text{tp}} ([obj], \sigma) \Rightarrow \text{cfg-safe}(\epsilon, \epsilon, \sigma)$, *and*
- *the parameter of every call event in* $\vec{t}_0$ *satisfies vr.*

Finally, we turn to prove Theorem 5.1.

Proof of Theorem 5.1. Conclusion (1) is proved by the soundness direction of Theorem 2.9. For conclusion (2), assume $([\texttt{init}()], \varnothing) \rightarrow^{*}_{\text{tp}} ([obj], \sigma)$ and $(\epsilon, \epsilon, \sigma) \xrightarrow[obj.\texttt{op}/vr]{\vec{t}}{}^{*}_{\text{adv}}\ (\vec{\pi}, \vec{e}, \sigma')$. By Lemma B.2, there exists an execution such that $MGC(\vec{t})(\texttt{op}, obj)$ returns $\vec{t}$. Further, by Lemma B.1 and the soundness direction of Theorem 2.9, every result of $MGC(\vec{t})(\texttt{op}, obj)$ satisfies I. Therefore, $I(\vec{t})$.

For the non-stuckness, also because $MGC(\vec{t})(\texttt{op}, obj)$ is safe, every expression in $\ll \vec{\pi} \gg +\!\!+ \vec{e}$ is not stuck. This concludes the non-stuckness of $\vec{e}$, and for $\ll \vec{\pi} \gg$, it is not hard to show that the reverse translation preserves non-stuckness. □

[8] This lemma statement is simplified to omit technical details about the trace library, *e.g.*, the actual trace library stores a mutex at $\ell_{tr} + 1$ and spawns a child thread to generate non-deterministic fresh tags.