Recent advances in concurrent separation logic enabled the formal verification of increasingly sophisticated fine-grained (i.e., lock-free) concurrent programs. For such programs, the golden standard of correctness is linearizability, which expresses that concurrent executions always behave as some valid sequence of sequential executions. Compositional approaches to linearizability (such as contextual refinement and logical atomicity) make it possible to prove linearizability of whole programs or compound data structures (e.g., a ticket lock) using proofs of linearizability of their individual components (e.g., a counter). While powerful, these approaches are also laborious—state-of-the-art tools such as Iris, FCSL, and Voila all require a form of interactive proof.

This paper develops proof automation for contextual refinement and logical atomicity in Iris. The key ingredient of our proof automation is a collection of proof rules whose application is directed by both the program and the logical state. This gives rise to effective proof search strategies that can prove linearizability of simple examples fully automatically. For more complex examples, we ensure the proof automation cooperates well with interactive proof tactics by minimizing the use of backtracking.

We implement our proof automation in Coq by extending and generalizing Diaframe, a proof automation extension for Iris. While the old version (Diaframe 1.0) was limited to ordinary Hoare triples, the new version (Diaframe 2.0) is extensible in its support for program verification styles: our proof search strategies for contextual refinement and logical atomicity are implemented as modules for Diaframe 2.0. We evaluate our proof automation on a set of existing benchmarks and novel proofs, showing that it provides significant reduction of proof work for both approaches to linearizability.
logically performs the operation on the data structure. This point is called the linearization point, and the effects of concurrent operations must then match the effects of the corresponding sequential operations, when ordered by linearization point.

Linearizability has originally been formulated as a property on program traces by Herlihy and Wing [1990]. This formulation is a good fit for automated proofs, as witnessed by fully automated methods based on shape analysis [Henzinger et al. 2013; Vafeiadis 2010; Zhu et al. 2015] and model checking [Burckhardt et al. 2010]—see Dongol and Derrick [2015] for a detailed survey. However, Dongol and Derrick [2015] classify these methods as not compositional: they are unable to abstractly capture the behavior of the environment. Accordingly, there has been an avalanche of research on formulations and proof methods for linearizability that enable compositional verification: proving linearizability of compound data structures (e.g., a ticket lock) using proofs of linearizability of their individual components (e.g., a counter). Unfortunately, proof automation for these compositional approaches to linearizability is still lacking.

**Compositional approaches to linearizability.** Notable examples of compositional approaches to linearizability are contextual refinement [Filipović et al. 2010; Liang and Feng 2013; Turon et al. 2013], logical atomicity [Birkedal et al. 2021; da Rocha Pinto et al. 2014; Jung et al. 2015], and resource morphisms [Nanevski et al. 2019]. We focus on the first two: they are both available in the Iris framework for separation logic in Coq [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a,b], and recent work by Mulder et al. [2022] provides a starting point for proof automation in Iris.

Linearizability follows from contextual refinement $e \preceq_{\text{ctx}} e'$, where $e$ is the fine-grained concurrent program, and $e'$ is a coarse-grained (i.e., lock-based) version of $e$. A program $e$ contextually refines $e'$, if for all well-typed contexts $C$, if $C[e]$ terminates with value $v$, there then exists an execution so that $C[e']$ also terminates with value $v$. The quantification over all contexts $C$ makes refinements compositional, but also difficult to prove. Turon et al. [2013] pioneered an approach based on separation logic that made it feasible to prove refinements of sophisticated concurrent algorithms on paper. Krebbers et al. [2017b] incorporated this work into Iris to enable interactive proofs using Coq. The state of the art for refinement proofs in Iris is the ReLoC framework [Frumin et al. 2018, 2021b], which has been applied to sophisticated examples such as the Michael-Scott queue [Vindum and Birkedal 2021] and a queue from Meta’s Folly library [Vindum et al. 2022].

Linearizability also follows from a logically atomic triple $⟨P⟩ e ⟨Q⟩$. Intuitively, such a triple gives a specification for the linearization point of the program $e$. Even though $e$ itself may not be physically atomic, $e$ will atomically update the resources in $P$ to the resources in $Q$, somewhere during its execution. Logically atomic triples can be composed inside the logic, i.e., the triple for one data structure (say, a counter) can be used to verify to another (say, a ticket lock). Logical atomicity has been pioneered in the TaDA logic by da Rocha Pinto et al. [2014], and was embedded in Iris and extended with support for higher-order programs and programs with “helping” (delegation of the linearization point to another thread) by Jung et al. [2015]. Logical atomicity in Iris has been used to verify challenging examples such as the Herlihy-Wing queue and RDCSS [Jung et al. 2020], and by engineers at Meta to verify a high-performance queue [Carbonneaux et al. 2022]. GoJournal [Chajed et al. 2021] uses logical atomicity in Iris to verify a concurrent, crash-safe journaling system of significant size (≈1.300 lines of Go code, ≈25.000 lines of Coq proofs). Compositionality is crucial in GoJournal’s verification: the implementation consists of four layers, and the verification of each layer uses the logically atomic specification of the previous layer.

**State of the art on proving linearizability compositionally.** The state of the art for compositional approaches to linearizability is to construct proofs interactively. Refinement and logical atomicity proofs in Iris are constructed interactively using the Iris Proof Mode in Coq [Krebbers et al. 2018, 2017b]. Similarly, linearizability proofs using the resource morphism approach [Nanevski et al. 2019]
et al. 2019] are constructed interactively using the FCSL framework in Coq [Sergey et al. 2015]. Both Iris and FCSL use a tactic-based style. That is, one writes down the program and specification (and all auxiliary definitions) and then carries out the proof using a sequence of tactics, where each tactic decomposes the proof obligation into simpler proof obligations.

An alternative proof style is used in the Voila tool [Wolf et al. 2021]—a proof outline checker for logical atomicity in TaDA [da Rocha Pinto et al. 2014] (a logic that preceded and influenced Iris). Contrary to the tactic-based style, Voila provides a proof style where the program is annotated with assertions and pragmas to guide the proof search. Being a proof outline checker, Voila’s goal is not full automation—it requires the user to provide (with pragmas) key steps of the proof. This significantly reduces the proof burden compared to interactive proofs in tactic-based tools such as Iris and FCSL, but still requires annotations for all lines of code that touch shared state.

This discussion indicates that proving linearizability is currently a laborious endeavor. This is also emphasized by Carbonneaux et al. [2022] (who verified a queue for Meta using Iris):

We were also surprised that the most important lemmas took only a couple lines to prove while using the invariants and writing the code proofs required hundreds of rather straightforward lines. While Iris’ proof mode made using CSL [Concurrent Separation Logic] easy, this observation seems to indicate that there remains untapped potential to increase the reasoning density.

This paper presents a step forward to obtain this untapped potential. We present Diaframe 2.0—a proof automation extension for Iris, which we have successfully used to automate (parts of) contextual refinement and logical atomicity proofs. Before describing the key ideas and architecture of Diaframe 2.0, let us first outline our design goals.

**Design goal #1: Fully automated proofs for ‘simple’ programs.** Our goal is to prove linearizability of ‘simple’ programs fully automatically. That is, once the program and specification are written down, the tool should find a proof without user assistance. This brings the tooling for compositional approaches closer to the tooling for non-compositional (trace-based) approaches.

**Design goal #2: Assistance using interactive proofs for ‘complex’ programs.** Although we aim for full proof automation of ‘simple’ programs, this should not come at the cost of expressivity. We also want to verify arbitrarily ‘complex’ programs and give them strong specifications. Providing full automation that works in every situation is impossible—due to Iris’s expressive logic, any proof automation is inherently incomplete (in fact, propositional separation logic is already undecidable [Brotherston and Kanovich 2014]). For more complex examples, the proof automation should be predictable and behave in an acceptable manner when it encounters a goal it cannot solve. This means the proof automation should be able to make partial progress (instead of only being able to fully solve a goal or fail), so that the user can assist if needed.

**Design goal #3: Declarative and modular definitions of proof automation.** Logics for refinement and logical atomicity are very different—they use different judgments with bespoke proof rules. To avoid having to reinvent the wheel for both logics, we would like to write our proof automation in a way that is declarative (i.e., that abstracts over low-level aspects) and modular (i.e., that can be composed out of different ‘modules’). Despite the differences between both logics, both are based on separation logic. This means that the proof automation for both logics needs to deal with the fact that resources are substructural (can be used at most once), and should share features provided by Iris such as modalities, impredicative invariants and custom ghost state. It is thus desirable to have a shared ‘core’ module. We want to have an integration between (the automation for) both logics so that logically atomic triples (which provide internal compositionality) can be used to prove refinements (which provide external compositionality). This should be achievable by
combining the two modules. During the development, we wish to be able to quickly experiment with different rules and priorities. This should be possible by changing the relevant module locally instead of the proof automation globally. In the future, we want to support new features of Iris (such as prophecy variables [Jung et al. 2020] and later credits [Spies et al. 2022]) or new specification styles in Iris (such as termination-preserving refinement [Gäher et al. 2022] and the security condition non-interference [Frumin et al. 2021a; Gregersen et al. 2021]). Ideally, this should also be possible by adding additional modules instead of having to change the proof automation globally.

**Design goal #4: Foundational proofs in a proof assistant.** To ensure that our proof automation is as trustworthy as possible, we want it to be foundational [Appel 2001]. This means that proofs are conducted in a proof assistant against the operational semantics of the programming language. To achieve this, the proof rules of the logic need to be proved sound (which has already been done for Iris) and our automation needs to be proved sound against the Iris proof rules (which is one of the contributions of this paper).

**Key ideas for achieving the design goals.** Our desired proof automation should not only be able to fully automatically construct simple proofs of linearizability (Design goal #1), it should allow user assistance with interactive proofs (Design goal #2), and be defined declaratively (Design goal #3). We list the key design choices that we hold responsible for achieving this combination of constraints. Our final design goal is to produce foundational proofs (Design goal #4), but we believe our key ideas could be useful even in a non-foundational setting (i.e., outside of a proof assistant).

- **Minimize backtracking.** To ensure the proof automation cooperates well with interactive proofs, we avoid the use of backtracking in our proof automation whenever possible. In many cases, it is not apparent that backtracking can be avoided—but it can be avoided more frequently than one might expect. By avoiding backtracking, it becomes much easier to alternate between proof automation and interactive proof tactics: the proof automation can simply be ‘run’ until it gets stuck, at which point the user can use a tactic (or other means) to direct the proof.

- **Use program and logical state to select proof rules.** While we want to minimize backtracking, multiple proof rules are often applicable during the verification of a program. To select the correct proof rule, the proof automation also inspects the logical state of the proof. This gives Diaframe 2.0 an edge on other proof automation tools, where such information is not available or fully leveraged. For example, this allows Diaframe 2.0 to automatically perform some key steps for dealing with shared state in logical atomicity proofs, while they must be provided explicitly in proof outlines for Voila.

- **Represent proof rules as instances of a general format, and leverage near-applicability.** To implement our proof automation in a declarative and modular way, we identify general formats to capture proof rules. These formats describe the ‘current’ and ‘new’ verification goal, and optionally, a piece of required logical state. To extend the proof search strategy with additional proof rules, one simply shows that they can be written as instances of the general formats. Modules for our proof automation are then just collections of rules, executed by the proof automation strategy. We also add flexibility for when the logical state or current goal nearly matches a rule—for example, when the required logical state can be found beneath a connective of the logic. In such cases, the rule is still applied automatically, but the automation will first deal with the connective. This keeps the modules of our proof automation declarative and concise, while becoming applicable in more situations.

**Implementation of Diaframe 2.0.** The implementation of Diaframe 2.0 is guided by the design goals and choices described above. An overview of Diaframe 2.0’s architecture is shown in Fig. 1.
Proof Automation for Linearizability in Separation Logic

Fig. 1. Overview of the architecture of Diaframe 2.0.

The key ingredients are the proof strategies underpinning the refinement and logical atomicity modules. To realize these strategies, we start with the original proof rules of ReLoC and logically atomic triples in Iris, and design derived rules whose application is directed by the program and logical state. These derived rules are proved sound in Coq (Design goal #4), and make up our proof search strategy. To ensure good integration with interactive proofs (Design goal #2) and as per our design choices, our strategies make minimal use of backtracking. Backtracking is sometimes needed to find the linearization point, but our strategies are otherwise deterministic. Backtracking can be disabled altogether, allowing the user to intervene at key steps in the proof.

Proof automation for linearizability in Iris critically relies on dealing with the cornerstones of Iris’s concurrent separation logic: invariants and ghost resources. For these, we build upon our earlier work Diaframe 1.0 [Mulder et al. 2022]. Diaframe 1.0 provides proof automation for the verification of fine-grained concurrent programs, but is restricted to Hoare triples for functional correctness—and thus does not support linearizability. However, we reuse Diaframe 1.0’s key innovation: its ability to automatically reason with invariants and ghost resources. In accordance with Design goal #3, this is a separate proof automation module used by both the refinement and logical atomicity proof search strategies.

To express the proof search strategies for contextual refinement and logical atomicity in a declarative manner (Design goal #3), we identify two general formats for rules in these strategies. Abduction hints are used to replace a program specification goal with a successive goal. One can specify whether this must be done unconditionally, only when a certain hypothesis is spotted, or just as a last resort. A simple collection of abduction hints can describe the original Diaframe 1.0 strategy for Hoare triples (so Diaframe 2.0 is backwards compatible w.r.t. Diaframe 1.0). Transformer hints apply to goals where we need to reason about the entire context. Simple instances of transformer hints are the introduction rules for Iris’s various modalities, such as the later (\(\triangleright\)) and persistence (\(\Box\)) modality. The combination of abduction and transformer hints can express a crucial proof rule in the verification of logically atomic triples. Additionally, they allow us to apply (Löb) induction automatically (which was impossible in Diaframe 1.0).

Following ideas from Gonthier et al. [2011]; Krebbers et al. [2017b]; Spitters and Weegen [2011], we represent these hints using type classes in Coq [Sozeau and Oury 2008]. The modules for our strategies for contextual refinement and logical atomicity are given as collections of type class instances. Diaframe 2.0’s proof automation is implemented as an Ltac tactic [Delahaye 2000], that uses type class search to select an applicable hint (i.e., a rule in the strategy) for a given goal. Type class search is also used to close off our rules under the connectives of separation logic, thus achieving our third key idea of near-applicability. Coq requires us to prove soundness of each rule represented as a type class instance, thus achieving foundational proofs (Design goal #4). Aside
from enabling declarative definitions of proof search strategies (Design goal #3), the use of type classes is more robust compared to implementing the strategies directly as an Ltac tactic. Type class instances are strongly typed, so many errors show up during the implementation of the strategy as hints, instead of during the execution of the proof strategy.

**Contributions and outline.** Our contributions are as follows:

- In §2 we describe our proof automation strategy for refinements in ReLoC.
- In §3 we describe our proof automation strategy for logically atomic triples in Iris.
- In §4 we describe the extensible proof automation strategy that underpins Diaframe 2.0. This strategy is parametric in the program specification style through the use of three kinds of hints—for abduction (new), transformer (new), and bi-abduction (from Diaframe 1.0). The proof automation strategies for our first two contributions are encoded in Diaframe 2.0.
- In §5 we evaluate our proof automation on existing and new benchmarks. We compare to existing proofs in Voila [Wolf et al. 2021], showing an average proof size reduction by a factor 4, while adding foundational guarantees (§5.1). We compare to existing interactive proofs of RDCSS and the elimination stack in Iris, showing an average proof size reduction by a factor 4 (§5.2). Our new result is a proof of logical atomicity for the Michael-Scott queue [Michael and Scott 1996] (§5.3). For refinement, we compare to existing interactive proofs in ReLoC, showing an average proof size reduction by a factor 7 (§5.4).
- All of our results have been implemented and verified using the Coq proof assistant. The Coq sources can be found in Mulder and Krebbers [2023].

We conclude the paper with related work (§6) and future work (§7).

## 2 PROOF AUTOMATION FOR CONTEXTUAL REFINEMENT

This section introduces the main ideas for automating contextual refinement proofs in the Iris-based logic ReLoC [Frumin et al. 2018, 2021b]. We start with an example verification (§2.1), providing intuition for ReLoC. After providing some formal background for ReLoC’s proof rules (§2.2), we describe our proof automation strategy (§2.3).

### 2.1 Contextual Refinement of an Incrementer

Contextual refinement specifies the behavior of one program in terms of another, usually simpler, program. For linearizability, we take a coarse-grained version as the simpler program, i.e., a version that uses a lock to guard access to shared resources. Filipović et al. [2010] shows that such refinements imply the classical definition of linearizability based on traces. Consider the example in Fig. 2, a slightly modified version of the example presented in the first ReLoC paper [Frumin et al. 2018]. We consider two implementations of an “incrementer”: `fg_incrementer` and `cg_incrementer`. Whenever either such an incrementer is called with the unit value, it returns a closure. In-}

ituitively, both versions "have the same behavior"—although they use different methods, both programs guarantee a consistent tally of calls to the closure. Whenever this returned closure is called with the unit value, it returns an integer indicating the number of times the closure has been called in total, across all threads.

Where the fine-grained version `fg_incrementer` uses a CAS-loop (Compare And Swap) to deal with concurrent calls to the closure, the coarse-grained version `cg_incrementer` uses a lock. Intuitively, both versions “have the same behavior”—although they use different methods, both programs guarantee a consistent tally of calls to the closure. We wish to prove a contextual refinement `fg_incrementer \leq_{\text{ctx}} cg_incrementer : () \rightarrow () \rightarrow \mathbb{Z}` that expresses that any behavior of `fg_incrementer` is a behavior of `cg_incrementer`. More precisely, a contextual refinement `e_1 \leq_{\text{ctx}} e_2 : A` expresses that for all contexts `C` that respect the type `A` of `e_1` and `e_2`, if `C[e_1]` terminates with value `z`, then there exists an execution sequence such that `C[e_2]` also terminates with value `z`. 

Proof Automation for Linearizability in Separation Logic

\[ \text{Definition } \text{fg}_\text{incrementer} : \text{val} := \lambda : \langle \rangle \rightarrow \langle \rangle \rightarrow \text{int} \]

\[ \text{let: } "l" := \text{ref #1 in} \]

\[ \text{rec: } "f" := \langle \rangle \rightarrow \text{let: } "n" := ! "l" \text{ in} \]

\[ \text{if: CAS } "l" "n" ("n" + #1) \text{ then} \]

\[ "n" \]

\[ \text{else} \]

\[ "f" () \]

\[ \text{end.} \]

\[ \text{Lemma } \text{fg}_\text{cg}_\text{incrementer} \text{ refinement :} \]

\[ \vdash \text{REL } \text{fg}_\text{incrementer} \preceq \text{cg}_\text{incrementer} : () \rightarrow () \rightarrow \text{lrel int}. \]

\[ \text{Proof.} \]

iStepsS.

iAssert (|={\top}=> \text{inv} (\text{nroot."incr"}) \]

\[ (\exists (n : \text{nat}), x \mapsto #n \times \text{x} \mapsto #x \rightarrow \text{is_locked_r x} ! \text{false})%I \]

with "[-]" as ">#Hinv"; first iStepsS.

iSmash.

Qed.

Fig. 2. Verification of a refinement for a fine-grained concurrent incrementer in Diaframe 2.0.

It is well known that it is difficult to prove such contextual refinements, since they quantify over all contexts \( C \). A common way to make these proofs tractable, is by introducing a notion of \textit{logical refinement}, which implies contextual refinement, but is easier to prove [Pitts 2005]. There exist many approaches to define a notion of logical refinement, but in this paper we focus on approaches based on separation logic as pioneered in the work by Dreyer et al. [2010] and Turon et al. [2013]. Approaches based on separation logic enable the use of resource and ownership reasoning and are thereby well-suited for programs that use mutable state and concurrency. A state-of-the-art separation logic for refinements based on this idea is ReLoC [Frumin et al. 2018, 2021b]. ReLoC is embedded in Iris and comes with a judgment \(|\vdash e_1 \preceq e_2 : A|\) for logical refinements.

ReLoC’s soundness theorem states that to prove the contextual refinement \( e_1 \preceq_{\text{ctx}} e_2 : A \), it suffices to prove a (closed) Iris entailment \( \vdash e_1 \preceq e_2 : A \). Here, \( \vdash e_1 \preceq e_2 : A \) is a proposition in separation logic, which allows us to write refinements that are conditional on mutable state. For example, we can prove that \( \ell_1 \mapsto z + \ell_r \mapsto z \vdash |!\ell_1 \preceq !\ell_r : \text{Z} \), i.e., a load of \( \ell_1 \) contextually refines a load of \( \ell_r \), if both locations are valid pointers and point to the same value \( z \). The maps-to connectives \( \ell_1 \mapsto z \) and \( \ell_r \mapsto z \) represent the right to read and write to a location \( \ell \). Since we are reasoning about two programs (and thus, two heaps), ReLoC uses the subscripted \( \mapsto_{\text{s}} \) (with ‘s’ for specification) to indicate the heap of the right-hand side execution.

Proofs of ReLoC’s refinement judgment \( \vdash e_1 \preceq e_2 : A \) use symbolic execution to reduce expressions \( e_1 \) and \( e_2 \). The execution of \( e_1 \) can be thought of as demonic: all possible behaviors of \( e_1 \) need to be considered. The execution of \( e_2 \) is angelic—we just need to find one behavior that matches with \( e_1 \). In a concurrent setting, this means \( e_1 \) needs to account for (possibly uncooperative) other threads, while \( e_2 \) can assume cooperative threads and scheduling.

\textbf{Verification of the example.} Let us now return to the verification of the example in Fig. 2. Our top-level goal (line 21) is the following logical refinement of closures:

\[ \vdash \text{fg}_\text{incrementer} \preceq \text{cg}_\text{incrementer} : () \rightarrow () \rightarrow \text{Z}. \]

The proof consists of 4 phases:
(1) Symbolically execute both outer closures. This will create shared mutable state used by the inner closures.

(2) Determine and establish a proper invariant for the shared mutable state.

(3) Perform induction to account for the recursive call in \( f_g \_\text{incrementer} \).

(4) Symbolically execute the inner closures, using the established invariant. This should allow us to conclude the refinement proof.

These phases are representative for proofs of logical refinements. For this example, Diaframe 2.0 can automatically deal with Proof Phase 1, Proof Phase 3 and Proof Phase 4. Automatically determining proper invariants is very difficult, so we leave Proof Phase 2 up to the user (line 24–26).

As shown in Fig. 2, the Diaframe 2.0 proof takes 5 lines. The user’s main proof burden is writing down the invariant \( \exists n. \ell_1 \mapsto n \land \ell_r \mapsto n \land \text{isLock}(v, \text{false}) \), i.e., Proof Phase 2. (In Coq, we write \( \text{inv } N \text{ R for } R^N \).) Iris’s invariant assertion \( R^N \) states that there is a (shared) invariant with name \( N \), governing resources satisfying Iris assertion \( R \). Since \( R \) can be shared, access to the resources in \( R \) must be obtained at a price. They can only be accessed temporarily, during the execution of a single atomic expression (e.g., a load, store, or CAS) on the left-hand side of the refinement. After this expression, the invariant must be closed, i.e., one must show that assertion \( R \) still holds. Since execution of the right-hand side is angelic, we can execute the right-hand side multiple steps while an invariant is opened.

Our proof proceeds as follows. We open the invariant to symbolically execute the load on the left-hand side. This does not change the stored value, so we can immediately close the invariant. We now reach the CAS on the left. We open the invariant again, and distinguish two cases. If the CAS succeeds, we symbolically execute the entire right-hand side, which signifies the linearization point. The invariant guarantees that the right-hand side expression returns \( n \) as desired. If the CAS fails, we close the invariant and use the induction hypothesis to finish the proof.

### 2.2 Background: Formal Rules for Contextual Refinement

To put the proof on a formal footing, we introduce some of Iris’s and ReLoC’s (existing) proof rules. An overview can be found in Fig. 3. We go through the phases of the proof, introducing relevant concepts (such as the □ modality, invariant reasoning, and Löb induction) when necessary.

**Proof Phase 1: Symbolic execution of outer closures and the □ modality.** Recalling our initial proof obligation \( \vdash \ f_g \_\text{incrementer} \leq cg \_\text{incrementer} : () \rightarrow () \rightarrow \mathbb{Z} \), we can start our proof by using \textsc{Refines-Closure}. This rule is applicable for any proof context \( \Delta \), where \( \Delta \) stands for a list of assertions \( P_1, \ldots , P_n \). We denote \( \Delta \vdash Q \) for \( P_1 \ast \ldots \ast P_n \vdash Q \).

Let us consider the premise of \textsc{Refines-Closure}: we need to prove \( \vdash \Box (\vdash v_1 () \leq v_2 () : A) \). This means Iris’s persistence modality \( \Box \)—the new proof obligation can be read as “it is persistently true that \( v_1 () \) logically refines \( v_2 () \) at type \( A \)”. A proof of \( \Box G \) implies that \( G \) is duplicable, and can thus be used more than once—this is not a given in substructural logics. To see why this modality is necessary, note that clients may use the closure any number of times (and concurrently). Since the two closures have not introduced any state (and the proof context \( \Delta \) is thus empty), we can apply \textsc{Iris-□-Intro}, introducing the □ modality, and continue symbolic execution.

We can then use \textsc{Alloc-L}, \textsc{Alloc-R} and \textsc{NewLock-R} to symbolically execute instructions on both sides. Our proof obligation now looks as follows:

\[
\ell_1 \rightarrow 1 \quad \ell_r \rightarrow s_1 \quad \text{isLock}(v, \text{false}) \vdash \vdash (\text{rec} \ldots) \leq (\lambda \ldots) : () \rightarrow \mathbb{Z}
\]  

(2)

We obtain two maps-to connectives \( \ell_1 \rightarrow 1 \) and \( \ell_r \rightarrow s_1 \) in our proof context. Remember that these are exclusive resources that can only be owned by one thread, and which signify the right to read and write to a location \( \ell \). Similarly, \text{isLock}(v, \text{false}) \) is an exclusive resource that says the lock \( v \) is...
unlocked. In our proof obligation Equation (2), the two references and lock are captured and used by the closures. Moreover, the left-hand closure will perform a CAS on \( \ell_1 \), meaning that concurrent calls to this closure will all try to write to the same location. However, only one thread can hold the \( \ell_1 \mapsto n \) resource, so we need a way to give shared access to this resource in the logic.

**Proof Phase 2: Establish an invariant.** In Iris, we can verify concurrent accesses using an invariant \( [R] \). At any point during the verification, a resource \( R \) can be turned into \([R] \) using INV-ALLOC. This is called invariant allocation. The assertion \([R] \) is persistent, so unlike exclusive using, we do actual work before we apply the induction hypothesis. After doing some of this work, we reach the recursion point and need to prove \( \Gamma \) holds, but only later, as witnessed by CAS operations of the left-hand expression. Additionally, we need to show that a successful CAS from \( n \) to \( n + 1 \) (the linearization point) corresponds to an execution path for the right-hand expression that terminates in \( n \). The invariant we have established guarantees that.

We return to our proof obligation Equation (2). To continue, we will first allocate an invariant using INV-ALLOC. We take \( R \triangleq \exists n. \ell_1 \mapsto n * \ell_r \mapsto \top * \text{isLock}(v, \text{false}) \), which expresses that the values stored at \( \ell_1 \) and \( \ell_r \) are in sync. After \text{REFINES-CLOSURE} and \text{IRIS-□-INTRO}, we are left with:

\[
\exists n. \ell_1 \mapsto n * \ell_r \mapsto \top * \text{isLock}(v, \text{false})^N \vdash (\text{rec} \ldots) () \preceq (\lambda \ldots) () : Z. \tag{3}
\]

The left-hand side is now a recursive function applied to the unit value (), which will repeat until the CAS on line 6 succeeds. To finish the proof, we need to account for the recursive call.

**Proof Phase 3: Löb induction.** To verify recursive functions, step-indexed separation logics such as Iris and ReLoC use a principle called Löb induction. In essence, whenever we are proving a goal \( G \), we are allowed to assume the induction hypothesis \( \triangleright G \) — the same goal, but guarded by the later modality \( (\triangleright) \) [Appel et al. 2007; Nakano 2000]. We are allowed to strip later modalities of hypotheses only after we perform a step of symbolic execution on the left-hand side. This ensures we do actual work before we apply the induction hypothesis. After doing some of this work, we reach the recursion point and need to prove \( G \) again. Since the work stripped off the later modality of our induction hypothesis, we are in shape to apply the induction hypothesis and finish the proof.

A selection of Iris’s rules for the \( \triangleright \) and \( \Box \) modality and Löb induction are shown in Fig. 3b. Rule LÖB states that, if we are proving that \( \Delta \vdash G \), we can assume that the induction hypothesis \( \Box (\Delta \vdash G) \) holds, but only later. We can get rid of this later (\( \triangleright \)) whenever our goal gets prefixed by a later, as witnessed by \( \triangleright - \text{INTRO} \). Iris’s \( \Box \) modality ensures that the induction hypothesis \( \Delta \vdash G \) can be used more than once. This is reflected in the logic by the rules \( \Box - \text{ELIM} \) and \( \Box - \text{DUP} \).

We can now continue proving our goal Equation (3). After LÖB and UNFOLD-REC-L, our goal is:

\[
\left( \exists n. \ell_1 \mapsto n * \ell_r \mapsto \top * \text{isLock}(v, \text{false})^N, \Box (\vdash (\text{rec} \ldots) () \preceq (\lambda \ldots) () : Z) \right) \vdash (\text{let } n := !f \ldots) \preceq \ldots : Z \tag{4}
\]

**Proof Phase 4: Symbolic execution of inner closures.** To finish the proof, we need to justify the safety of the load and CAS operations of the left-hand expression. Additionally, we need to show that a successful CAS from \( n \) to \( n + 1 \) (the linearization point) corresponds to an execution path for the right-hand expression that terminates in \( n \). The invariant we have established guarantees that. Some additional rules for symbolic execution with invariants in ReLoC can be found in Fig. 3c.

As mentioned before, we can only access the resources in \( R \) for the duration of atomic expressions. Let us consider the LOAD-L rule, to see how this is enforced. The premise of the rule mentions the fancy update modality \( E_1 \triangleright E_2 \). The semantics of \( E_1 \triangleright E_2 P \) is: assuming all invariants with names in \( E_1 \) hold, then \( P \) holds, and additionally all invariants with names in \( E_2 \) hold. The masks \( E \) thus allow Iris to keep track of the opened invariants, and avoids opening invariants twice (i.e., invariant reentrancy, which is unsound). Note that LOAD-L also shows that refinement judgments \( \models E_1 \models E_2 \models A \) have a mask parameter. We let \( E = \top \) when the mask is omitted.
Fig. 3. A selection of the existing rules of Iris and ReLoC.
After some pure reduction our goal becomes the Löb induction hypothesis, concluding our proof:

At this point, we cannot restore the invariant:

\[ \ell \]

(which are called timeless in Iris), the later modalities can be discarded.

Returning to \texttt{LOAD-L:} with \( \mathcal{E} = \top \setminus \mathcal{N} \), we can combine \texttt{INV-ACCESS}, \texttt{FUPD-ELIM} and \texttt{FUPD-INTRO} to prove \( \exists \alpha. \ell \mapsto \nu \) with the resources from our invariant. We then receive \( \ell \mapsto \nu \) back, since the load operation does not change the state. Our new proof obligation is:

\[
\left( \begin{array}{c}
\mathcal{R} \setminus \mathcal{N}, \ell \mapsto n, \ell_r \mapsto \nu, \text{isLock}(\nu, \text{false}), \\
(\triangleright R \to \top \setminus \mathcal{N} \Rightarrow \top \text{True}), \\
\square(\models (\text{rec} \ldots) () \preceq (\lambda \ldots) () : Z)
\end{array} \right) \vdash \models_{\top \setminus \mathcal{N}} (\text{let } n := n \ldots) \preceq \ldots : Z
\]

Since we opened an invariant, the refinement judgment after the turnstile has \( \mathcal{N} \) removed from its mask. All symbolic execution rules for the left-hand side require the mask to be \( \top \), while the symbolic execution rules for the right-hand side work for every mask \( \mathcal{E} \). This reflects the demonic and angelic nature of left-hand-side and right-hand-side execution: we can keep invariants open for multiple steps on the right, but only during a single atomic step on the left.

We refrain from symbolically executing the right-hand side until the \texttt{CAS} succeeds. After the load, we restore the invariant using \texttt{FUPD-ELIM}, and our hypothesis \( (\triangleright R \leftrightarrow \top \setminus \mathcal{N} \Rightarrow \top \text{True}) \). We then use \texttt{CAS-L}. Like at the load, our invariant will provide us with some \( n' \) for which \( \ell \mapsto n' \), and the \texttt{CAS} will succeed precisely when \( n = n' \). Note that it is crucial to also consider the case \( n \neq n' \): this happens when another thread incremented \( \ell \) between the load and the \texttt{CAS} of the current thread.

The conjunction \( (\land) \) in \texttt{CAS-L} means that the proof splits into two separate proof obligations. In the successful case, we receive the updated \( \ell \mapsto (n + 1) \), together with the resource \( \triangleright n = n' \). This embeds the true fact \( n = n' \) into Iris’s separation logic. Likewise, in the failing case we receive the unchanged \( \ell \mapsto n' \), together with the pure information \( \triangleright n \neq n' \).

\textbf{For case } \( n = n' \)\textbf{:} the \texttt{CAS} succeeds, and the left-hand-side expression will be returning \( n \). After some pure reduction, our proof obligation becomes:

\[
\left( \begin{array}{c}
\mathcal{R} \setminus \mathcal{N}, \ell \mapsto (n + 1), \ell_r \mapsto \nu, \text{isLock}(\nu, \text{false}), \\
(\triangleright R \to \top \setminus \mathcal{N} \Rightarrow \top \text{True}) \\
\square(\models (\text{rec} \ldots) () \preceq (\lambda \ldots) () : Z)
\end{array} \right) \vdash \models_{\top \setminus \mathcal{N}} n \preceq (\text{acquire}(\nu); \text{let } n = !\ell_r \ldots) : Z
\]

At this point, we cannot restore the invariant: \( \ell \) and \( \ell_r \) point to different values. Only after symbolically executing the right-hand side will we be able to restore the invariant, which indicates that the linearization point must be now. With rules like \texttt{LOAD-R} and \texttt{STORE-R}, we can acquire the lock, execute the load and store operations, and finally release the lock. We conclude the proof of this case by closing the invariant, and using \texttt{REFINES-Z}.

\textbf{For case } \( n \neq n' \)\textbf{, the }\texttt{CAS} fails, and we receive back \( \ell \mapsto n' \) unchanged. We restore the invariant. After some pure reduction our goal becomes the Löb induction hypothesis, concluding our proof:

\[
\mathcal{R} \setminus \mathcal{N}, \square(\models (\text{rec} \ldots) () \preceq (\lambda \ldots) () : Z) \vdash \models (\text{rec} \ldots) () \preceq (\lambda \ldots) () : Z
\]

### 2.3 Proof Automation Strategy

The above proof phases introduce different challenges for proof automation, in rising complexity:

- **Proof Phase 1:** Symbolic execution without preconditions, introducing the \( \square \) modality.
∀® \cdot L \cdot (\forall v. U \rightarrow \models_E \cdot K[v] \leq e_2 : A)$$

\[ \Delta \vdash K[e_1] \leq e_2 : A \]

\text{EXEC-R}

\[ \forall \cdot \cdot (L \cdot e_2 \cdot \cdot U) \cdot (\forall v. U \rightarrow \models_E \cdot e_1 \leq K[e_2] : A) \]

\[ \Delta \vdash e_1 \leq K[e_2] : A \]

\text{VAL-Z}

\[ \Delta \vdash \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
(5) If \( e_1 \) is not a value and \( E \neq \top \) and \( e_2 \) is not a value, there are two valid ways to proceed: either restore the invariant with \texttt{refines-fupd}, or perform symbolic execution on the right with \texttt{exec-r}. Depending on the user’s preference, the proof automation will backtrack on these choices, or stop and let the user choose how to proceed.

Additionally, for other goals \( \Delta \vdash G \):

(6) \( G = \square G' \): Apply \texttt{iris-\square-intro}, but only if all hypotheses in \( \Delta \) are persistent. Stop otherwise.

(7) \( G = \Rightarrow G' \): Apply rule \texttt{\Rightarrow-intro} to introduce the later and striplaters from the context.

(8) \( G = \exists \xi. \mathcal{L} G' \): Use proof automation from Diaframe 1.0 to make progress.

\textbf{Verification of the example in Fig. 2.} The strategy above is available using the \texttt{iStepsS} tactic in Coq. In the verification of the example in Fig. 2, the \texttt{iStepsS} tactic stops at line 24 after applying \texttt{val-fun} for the second time. Item 6 (\( \square \) introduction) would be applicable, except that the proof context \( \Delta \) is not persistent. Iris allows one to weaken the context before introducing the \( \square \) modality, but our automation refrains from doing so—it often leads to improvable goals down the line. Our automation thus stops and allows the user to allocate an invariant before proceeding. To allocate the invariant, we use the \texttt{iAssert} tactic from the Iris Proof Mode.

\textbf{Why these rules?} Let us motivate our proof strategy and indicate how it reflects the design goals described in §1. After the invariant is established, the refinement of the two closures is established completely automatically, as is Design goal #1. Automatically inferring invariants is outside Diaframe 2.0’s scope. The strategy as a whole makes explicit the pattern followed in most interactive proofs, although the details differ. To be precise, the pattern is: symbolically execute the left-hand side, until you reach an expression that may be subject to interference from the environment (i.e., for which an invariant must be opened). The right-hand side expression may need to be symbolically executed some number of times at these points.

Design goal #2 is to enable assistance with interactive tactics for difficult refinements. To do so, it is crucial that the proof automation does not perform backtracking, unless requested. None of the steps of our strategy perform backtracking, except for Item 5. This step needs to choose between restoring the invariant, and symbolically executing the right-hand expression. For linearizability, this corresponds to deferring or identifying the linearization point, which is known to be hard. The \texttt{iSmash} tactic will backtrack on this choice, and is used in Fig. 2 to finish the proof. The sequence \texttt{iStepsS; iApply \texttt{refines-fupd}; iStepsS} also constitutes a valid proof: \texttt{iStepsS} will not backtrack, and instead stop the proof automation. In that case, Iris’s \texttt{iApply \texttt{refines-fupd}} can be used to instruct the proof automation to restore the invariant (defer linearization), after which the proof can be finished with a second call to \texttt{iStepsS}.

Finally, Design goal #3 is declarative and modular proof automation. In the implementation, Items 7 and 8 are part of the core proof automation module. Item 6 comes in a separate module for handling \( \square G' \) goals, that may be of independent use for other goals. Items 1 to 5 are all part of the refinement module. We achieve foundational proofs (Design goal #4) by establishing that all rules used in our proof strategy can be derived from the primitive rules of ReLoC and Iris (i.e., they are not axiomatic). These derivations have been mechanized in Coq. Combined with the existing soundness proof of ReLoC and Iris, this makes sure that our automation constructs closed Coq proofs w.r.t. the operational semantics of the programming language involved.

3 PROOF AUTOMATION FOR LOGICAL ATOMICITY

This section considers logically atomic triples to establish linearizability. We start by giving intuition about the need and meaning of such triples (§3.1). After discussing the formal proof rules in Iris (§3.2), we show our strategy for proof automation of these triples (§3.3).
Definition inc : val :=
rec: "f" "1" :=
let: "n" := ! "l" in
if: CAS "l" "n" ("n" + #1) then
"n"
else
"f" "l".

Global Instance inc_spec (l : loc) :
SPEC (z : Z), << l →→ #z >> inc #l << RET #z; l →→ (#(z + 1)) >>.

Proof. iSmash. Qed.

Fig. 5. Verification of a logically atomic triple for a fine-grained concurrent incrementer in Diaframe 2.0.

3.1 Logical Atomicity in Iris
Consider the inc function defined in lines 1-7 of Fig. 5. The pattern of recursively trying to CAS occurs in various concurrent programs: we have seen it in fg_incrementer in §2, and it also occurs in the implementation of e.g., a ticket lock. To enable modular verification, we would like to give inc a useful specification that can be used in the verification of other concurrent algorithms.

Let us try to specify inc using a regular Hoare triple \{P\} e \{Φ\}, where P is an Iris assertion and Φ is an Iris predicate on values. The Hoare triple expresses that for each thread that owns resources satisfying the precondition P, executing e is safe, and if the execution terminates with value w, the thread will end up owning resources satisfying the postcondition Φ w. A naive specification is

\{ℓ →→ z\} inc ℓ \{v.⌜v = z⌝∗ℓ →→ (z + 1)\}.

This states that to execute inc ℓ, we need exclusive write-access to location ℓ, as indicated by the precondition ℓ →→ n. Once inc ℓ terminates, it returns value z, and the ℓ →→ (z + 1) in the postcondition tells us that the value stored by ℓ has been incremented. Although provable, this specification is not useful in a concurrent setting. It requires a thread to give up ℓ →→ z during inc ℓ, while it usually does not have exclusive access to ℓ →→ z.

We have seen that for refinements, calls to CAS can be verified in a concurrent setting. This is because CAS is a physically atomic instruction, which gives us access to invariant reasoning. To see how this works, we state Iris’s invariant rule for Hoare triples, and the specification for load:

\[
\text{hoare-load} \quad \text{hoare-inv-access}
\]

\[
\text{HOARE-LOAD} \quad \{\ell \mapsto v\} !\ell \{w.\exists w = v \mapsto \ell \mapsto v\}_E
\]
\[
\text{atomic e} \quad N \subseteq E
\]
\[
\{R * P\} e \{v. Q\}_E
\]

HOARE-LOAD gives a straightforward specification for loading a value: the expression returns the stored value v, and one keeps access to \ell \mapsto v. Like refinement judgments, every Hoare triple is annotated with a mask E. When opening invariants with HOARE-INV-ACCESS, the invariant names are removed from the masks, which prevents invariant reentrancy.

We can open invariants around the load instruction with HOARE-INV-ACCESS only because it is a physically atomic instruction, i.e., we have ‘atomic (!\ell)’. Since we do not have ‘atomic (inc \ell)’, this rule is not applicable. But although inc is not physically atomic, the effects of inc appear to take place atomically for clients. That is, at a certain point during the execution of inc, namely, when the CAS succeeds, \ell \mapsto z is atomically consumed to produce \ell \mapsto (z + 1). This gives us a characterization of linearizability: an operation is linearizable if it appears to take place atomically/instantly somewhere during its execution, and the precise moment when this happens place is called the linearization point. Inspired by the TaDA logic [da Rocha Pinto et al. 2014], Iris features a special kind of Hoare triple to specify this, called a logically atomic triple [Jung 2019; Jung et al. 2020, 2015]. We specify
the behavior of \texttt{inc} using the following logically atomic triple:
\[
\textsc{inc-logatom} \quad \langle z, \ell \mapsto z \rangle \texttt{inc} \langle \nu. \ell = z^1 \mapsto \ell \mapsto (z + 1) \rangle_0
\]

We replaced \{ with \langle, what did we gain? In words, the meaning of a logically atomic triple \langle P \rangle_e \langle \Phi \rangle is: at the linearization point in the execution of \(e\), the resources in \(P\) are atomically consumed to produce the resources in \(\Phi v\), where \(v\) is the final return value of expression \(e\). Birkedal et al. [2021] established formally that such triples indeed imply linearizability. Logically atomic triples have the additional benefit that they can be used \textit{inside the logic}, with the following reasoning rules:

\[
\begin{align*}
\text{LA-INVA} & \quad \frac{R \vdash N}{\langle \bar{x} \cdot \alpha \mapsto R \rangle E (\langle v \cdot \beta \mapsto R \rangle E)_{E \setminus N}} \\
\text{LA-HOARE} & \quad \frac{\langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle E)}{\forall \bar{x}. \langle \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^T}
\end{align*}
\]

\text{LA-INVA} shows that it is indeed possible to open invariants around logically atomic triples. The \text{LA-HOARE} rule shows that logically atomic triples are stronger than ordinary Hoare triples.

The curious use of binder \(z\) in \textsc{inc-logatom} deserves a comment. Logically atomic triples allow a certain amount of interference from other threads, such as concurrent calls to \texttt{inc}. In such cases, it is enough that at each moment there is some \(z\) for which \(\ell \mapsto z\). This \(z\) needs not be known when the function is called, and may well be different at different moments. To reflect this in the logic, the pre- and postconditions of logically atomic triples can be bound by (a number of) quantifiers.

### 3.2 Background: Proof Rules for Logically Atomic Triples

To see how we use logically atomic triples, we will first discuss Hoare triples in Iris in more detail. Hoare triples in Iris are not a primitive notion, but defined in terms of \textit{weakest preconditions}:

\[
\{P\} e \{\Phi\} \triangleq \square \{P \dashv wp e \{\Phi\}\}
\]

The weakest precondition \(wp e \{\Phi\}\) asserts that execution of \(e\) is safe (cannot get stuck), and if \(e\) terminates with value \(v\), we get \(\Phi v\). The Hoare triple \{\(P\)\} \(e\) \{\(\Phi\)\} thus states that we can persistently (so, multiple times) relinquish \(P\) to execute \(e\), after which we obtain \(\Phi v\) for the return value \(v\).

Like Hoare triples, logically atomic triples are defined in terms of weakest preconditions:\(^1\)

\[
\text{LA-DEF} \quad \langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E \triangleq \forall \bar{x}. (\langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E)_{\uparrow}^E
\]

This expresses that for any postcondition \(\Phi\), to prove \(wp e \{\Phi\}\) it is enough to show an \textit{atomic update} of the form \(\langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E\). Atomic updates represent the possibility to witness variables \(\bar{x}\) for which \(\alpha\) holds, at any instant. If one uses this possibility, one either needs to hand back ownership of this exact \(\alpha\) to recover the atomic update, or hand back \(\beta\) to obtain \(\Phi v\) (commit the linearization point). By quantifying over \(\Phi\), Iris makes sure that the only way to prove a logically atomic triple is by using the atomic update \(\langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E\).

\textit{Proving logically atomic triples}. Proving a logically atomic triple \(\langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E\) is a matter of ‘just’ proving a weakest precondition, \textit{i.e.}, a goal \(\Delta \vdash wp e \{\Phi\}\). However, we need the atomic update to get temporary access to \(\alpha\) and eventually get \(\Phi\). Atomic updates can be accessed as:

\[
\text{AU-ACCESS-IRIS} \quad \langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E \vdash E \Rightarrow^\emptyset \exists \bar{x}. \alpha \ast \left( (\alpha \Rightarrow^0 \Rightarrow^E (\langle \bar{x} \cdot \alpha \rangle e (\langle v \cdot \beta \rangle)_{\uparrow}^E)_{\uparrow}^E (Q)_{\uparrow}^E ) \land (\forall \nu. \beta \Rightarrow^0 \Rightarrow^E Q) \right)
\]

\(^1\)The definition of logically atomic triples does not feature the \(\square\) modality to allow for \textit{private} preconditions, \textit{i.e.}, preconditions that one must relinquish completely at the start of the execution of \(e\). To make a logically atomic triple persistent, one has to add the persistence modality explicitly. This is for example visible in rule \text{LA-HOARE}.
This rule states that (similar to Iris’s rule for invariants INV-ACCESS) an atomic update provides access to $\alpha$ by changing the masks of a fancy update ($E \Rightarrow^\theta$). After we obtain $\alpha$, there are two ways to restore the mask, corresponding to the two sides of the (regular) conjunction. In the left conjunct, we need to return precisely $\alpha$. This corresponds to ‘peeking’ at the state, without changing it (in our example, this happens when the CAS fails). After peeking, we receive back the atomic update, deferring the linearization point. For the right conjunct, we need to provide $\beta$, which corresponds to committing to the linearization point (in our example, this happens when the CAS succeeds).

We then get access to $Q$, the postcondition in LA-DEF. One might be surprised to see a regular conjunction ($\land$) in separation logic, where the separating conjunction ($\ast$) is more common. Regular conjunction corresponds to a form of internal choice: if one owns a regular conjunction $P \land Q$, one can either use it as $P$ (here, defer linearization) or as $Q$ (here, commit linearization), but not as both.

A proof of the logically atomic triple for $\text{inc}$ in Fig. 5 needs to account for the recursive call when the CAS fails. We will use Löb induction once more—Fig. 6 contains the relevant rules. By combining Löb, unfold-rec and $\bowtie$-intro, we perform induction and start symbolic execution of the function. REC-APPLY shows how to apply the induction hypothesis at recursive calls.

**Using logically atomic triples.** With a proof of a logically atomic triple at hand, clients can use a combination of LA-HOARE, LA-INV and related rules to open invariants around the expression. In actual proofs, this is done differently, since working beneath binder $\exists x$ is cumbersome in Coq. Client verifications in Iris usually rely on the following rule:

\[
\begin{align*}
\text{SYM-EX-LOGATOM} & & \Delta \vdash \langle \exists x. \alpha \rangle \langle v. \beta \rangle_E & & \text{SYM-EX-LOGATOM} \\
\Delta \vdash \langle \exists x. \alpha \mid v. \beta \rangle \Rightarrow K[v] \{\Phi\} & & \Delta \vdash \langle \exists x. \alpha \mid v. \beta \rangle \Rightarrow Q \rangle_E
\end{align*}
\]

Instead of proving a logically atomic triple directly, one is now asked to prove an atomic update. Atomic updates can be introduced as follows:

\[
\begin{align*}
\text{AU-INTRO} & & \Delta \vdash \langle \exists x. \alpha \rangle \Rightarrow^\star E' \exists x. \alpha \ast ((\alpha \ast \vec{\alpha} \Rightarrow^G \Delta) \land (\forall v. \beta \ast \vec{\beta} \Rightarrow^G Q)) & & \text{AU-INTRO} \\
\Delta \vdash \langle \exists x. \alpha \mid v. \beta \rangle \Rightarrow Q \rangle E
\end{align*}
\]

This rule asks us to show that opening some invariants in $E$ gives us $\alpha$. Additionally, we need to prove that obtaining $\alpha$ is non-destructive: the original context $\Delta$ can be restored. This ensures that when the implementation peeks at $\alpha$, it does not affect the client. The other side of the conjunction shows that the atomic postcondition $\beta$ can be used to restore the invariants, and prove $Q$.

### 3.3 Proof Automation Strategy

Our proof automation for logical atomicity should be able to make progress on the following goals:

- Weakest preconditions: $\Delta \vdash \text{wp } e \{\Phi\}$, by definition of logically atomic triples LA-DEF.
Atomic updates: \( \Delta \vdash (\vec{x}. \alpha \mid o. \beta \Rightarrow Q)_E \), when applying a known triple \( \text{SYM-EX-LOGATOM} \).

Goals of the form \( \Delta \vdash E_1 \Rightarrow E_2 \ X. L \ast G \), after introducing atomic updates \( \text{AU-INTRO} \) or to establish the precondition of heap operations such as load and CAS. The context \( \Delta \) might contain atomic updates that should be eliminated via \( \text{AU-ACCESS-IRIS} \).

Goals prefixed by a later modality: \( \Delta \vdash G \), when using \( \text{UNFOLD-REC} \) after Löb induction.

Our proof search strategy for these goals extends the existing proof search strategy from Diaframe 1.0 by internalizing Löb induction, and by adding support for logically atomic triples.

Suppose our goal is \( \Delta \vdash wp e \{ \Phi \} \). We proceed by case analysis on \( e \), trying the following rules in order (omitting some cases, e.g., those related to pure reductions and higher-order functions):

1. If \( e \) is a value \( v \), then directly continue with proving \( \Delta \vdash \top \Rightarrow \top \Phi v \).
2. If \( e = K[e'] \), then either:
   - We have a regular specification \( \forall \vec{x}. \{L\} e' \{U\} \) for \( e' \). Use Diaframe 1.0’s existing approach to make progress, which applies a rule similar to \( \text{EXEC-L} \).
   - We have a specification \( (\vec{x}, L) e' \langle v, U \rangle E \). Apply \( \text{SYM-EX-LOGATOM} \), continue with new goal \( \Delta \vdash (\vec{x}, L \mid v. U \Rightarrow wp K[v] \{\Phi\}) \top \Rightarrow E \).
3. Otherwise, try to find an induction hypothesis to use with \( \text{REC-APPLY} \).
   - If \( e = (r \text{rec } f x := b)v \), i.e., a possibly recursive function applied to a value \( v \). Two cases:
     - There is no actual recursion, i.e., \( f \) does not occur in \( b \). Apply \( \text{UNFOLD-REC} \) and continue with new goal \( \Delta \vdash \top \Rightarrow wp b[v/\{\phi\} \{\phi\}] \).
     - For recursive functions, apply Löb, then apply \( \text{UNFOLD-REC} \). Continue with new goal \( \Delta \vdash \top \Rightarrow wp b[e/f][v'/x][\{\phi\}] \).
   - Note that \( \Delta \) will contain an atomic update, which we will have to relinquish on recursive calls.

For \( \Delta \vdash G \) with \( G \) not a weakest precondition, we distinguish the following cases:

4. \( G \Rightarrow G' \). Apply rule \( \Rightarrow\text{-INTRO} \) to introduce the later and strip laters from the context.
5. \( G = (\vec{x}, L \mid v. U \Rightarrow G') E \). Two cases:
   - If \( G \in \Delta \), directly use it to finish the proof. This situation occurs after applying the induction hypothesis with \( \text{REC-APPLY} \).
   - Otherwise, we introduce the atomic update with \( \text{AU-INTRO} \). Our new goal becomes \( \Delta \vdash E_1 \Rightarrow E_2 \vec{x}. L \ast ((L \Rightarrow *E' \Rightarrow E) \land (v. U \Rightarrow *E' \Rightarrow E G)) \).
6. \( G = E_1 \Rightarrow E_2 \vec{x}. L \ast G \). Use proof automation from Diaframe 1.0 to make progress. If enabled and when relevant, Diaframe 1.0 will backtrack to determine the linearization point.

This strategy can prove the logically atomic triple in Fig. 5 without user assistance. It uses the \( \text{isMash} \) instead of the \( \text{iStepsS} \) tactic, which enables backtracking for automatically determining the linearization point in Item 6. Proving atomic updates is covered by Item 5; we now provide some details on how we use atomic updates in Item 6.

Using atomic updates. The verification of a logically atomic triple crucially depends on eliminating atomic updates \( (\vec{x}. \alpha \mid v. \beta \Rightarrow Q)_E \) with \( \text{AU-ACCESS-IRIS} \). The elimination of atomic updates needs to happen in Item 6 when the Diaframe 1.0 automation needs to obtain ownership of \( \alpha \).

This can be done by allowing Diaframe 1.0 to ’look inside’ atomic updates, allowing it to determine ways of obtaining ownership of resources inside \( \alpha \). Note that \( \text{AU-ACCESS-IRIS} \) is similar to the invariant accessing rule \( \text{INV-ACCESS} \), which Diaframe 1.0 can also apply automatically. The main difference is that we have two independent ways to restore the mask (indicated by the \( \land \)): we either defer or commit the linearization point. We need to ensure this choice is not made too early,

\[ \text{In the Coq implementation we additionally generalize the Löb induction hypothesis over the arguments supplied to } e. \]
and achieve this by replacing the conjunction with a disjunction on the left-hand side of a wand:\(^3\)

\[
\text{AU-ACCESS-DIAFRAME} \quad \langle \alpha \mid v. \beta \Rightarrow Q \rangle_{\mathcal{E}} \twoheadrightarrow \exists \alpha \ast \forall mv. \\
\quad ((\alpha \ast \forall mv = \text{None}^\gamma) \lor (\exists \alpha. \beta \ast \forall mv = \text{Some} v^\gamma)) \Rightarrow \exists \alpha \ast \forall mv.
\]

This disjunction needs to be proven to restore the mask, and the side of the disjunction will indicate whether the linearization point should be deferred or committed. The rule AU-ACCESS-DIAFRAME is derived from the rules for atomic updates of Iris. This result is mechanized in Coq.

Let us describe how this is used in the example from Fig. 5. To symbolically execute the load and CAS, ownership of \(\ell \mapsto z\) is needed. Since the atomic update \(\langle z. \ell \mapsto z \mid v. \ell \mapsto (z + 1) \Rightarrow Q \rangle_{\mathcal{E}}\) is in our context, Diaframe 1.0 will use AU-ACCESS-DIAFRAME to obtain temporary ownership of \(\ell \mapsto z\). After symbolic execution, we receive back a possibly changed \(\ell \mapsto z',\) and the remaining ‘closing resource’ of shape \((\forall mv. _{\mathcal{E}} \lor \_ = \text{Some} v^\gamma)\). Diaframe notices it can use this closing resource to restore the mask, so the goal becomes (note that \(mv\) is bound in \(G\)):

\[
\Delta \vdash \exists \forall mv. ((\alpha \ast \forall mv = \text{None}^\gamma) \lor (\exists \alpha. \beta \ast \forall mv = \text{Some} v^\gamma)) \ast G.
\]

The IStepsS tactic uses backtracking to pick the correct side of this disjunction—i.e., to decide if the linearization point should be deferred or committed. We can also use the non-backtracking tactic iLeft/iRight and pick the correct disjunct interactively with the Iris tactics iLeft/iRight.

**Functions.** There are two cases for functions. Item 3b handles the situation in which the function case where there is no actual recursion. Omitting this specialized version would work, but would cause Item 3b to generate useless induction hypotheses that in turn increase the search space in Item 2c, and thus slow down the automation. Omitting Item 3a would also make the goal less readable if the user wants to help out with an interactive proof.

**Why these rules?** The above rules constitute a strategy that can prove logical atomicity of ‘simple’ examples (Design goal #1). We have demonstrated this on the example in Fig. 5, and show a number of other simple examples in §5. To ensure good integration with interactive proofs (Design goal #2), we once again minimize the use of backtracking. Backtracking is only needed in Item 6 to identify the linearization point, just like for refinements. The proof automation is modular (Design goal #3): Items 4 and 6 are part of the core automation module, Items 1 to 3 are part of the weakest precondition module, while Item 5 comes in a separate module for proving atomic updates. Similar to our automation for refinements, we achieve foundational proofs (Design goal #4) by mechanizing that all rules used in our proof strategy can be derived from Iris’s primitive rules.

### 4 IMPLEMENTATION AS EXTENSIBLE PROOF STRATEGY

In § 2 and 3 we have seen descriptions of our proof search strategies for contextual refinement and logical atomicity, respectively. This section discusses their implementation; specifically, how they fit in the extensible proof automation strategy that underpins Diaframe 2.0.

Proof search strategies operate on Iris entailments \(\Delta \vdash G\), where (in our cases) \(G\) is a refinement judgment, later or persistence modality (§2), a weakest precondition, or an atomic update (§3). As we will see in §4.5, rules of these strategies cannot be represented by the automation of Diaframe 1.0. However, our insight is that each rule in such a strategy falls into one of the following categories:

1. Rules of the form \(\Delta \vdash G' \implies (\Delta \vdash G)\), and \(G' \vdash G\) is provable.
2. Rules of the form \(\Delta \setminus H \vdash G' \implies (\Delta \vdash G)\) for some \(H \in \Delta\), and \(H \ast G' \vdash G\) is provable.

\(^3\)This transformation is sound since both sides of the \(\land\) feature a fancy update \(\text{fs} \text{fs}^E\) with the same mask.
(3) Rules of the form \((\Delta' \vdash G') \implies (\Delta \vdash G)\), where \(\Delta'\) and \(G'\) can be calculated from \(\Delta\) and \(G\) by just inspecting their head symbols (i.e., modalities).

(4) Rules of the form \((\Delta \vdash G') \implies (\Delta \vdash G)\), where \(G'\) mentions the entire context \(\Delta\).

We repeat a select number of cases of the proof search strategy in § 2 and 3 to make this apparent:

1. If \(G = \text{wp } e \{ \Phi \}\) and \(e\) is a value \(v\), continue with \(\Delta \vdash T \Rightarrow T \Phi \cdot v\).
2. If \(G = \langle \{ x. L \} \mid v. U \Rightarrow \Phi \rangle_E\), check if \(G \in \Delta\). If so, we can continue with \(\Delta \setminus G \vdash \text{True}\).
3. If \(G = \Delta \setminus G'\), and all hypotheses in \(\Delta\) are persistent, continue with \(\Delta \vdash G'\). Note that the entailment \(G' \vdash \Delta \setminus G'\) does not hold. This rule is only valid because of the condition on \(\Delta\).
4. If \(G = \langle \{ x. L \} \mid v. U \Rightarrow \Phi \rangle_E\), and the above Rule 2 is not applicable, apply \text{AU-INTRO}. The new goal has shape \(\Delta \vdash E \Rightarrow E' \exists \{ x. L \} \cdot (L \rightarrow E' \Rightarrow E) \land (\forall v. U \rightarrow E' \Rightarrow E)'\). The \(\Delta\) occurs on the right-hand-side of the turnstile, so this rule falls outside the first two categories.

We describe a generic proof strategy based on this insight, that can be extended to support new goals (§4.5). We have implemented this proof strategy in Ltc [Delahaye 2000]. Support for new goals and proof rules can be added by providing appropriate hints (registered as type class instances in Coq [Sozeau and Oury 2008]), corresponding to Category 1 to 4. Rules of Category 1 and 2 fit into our abduction hints (§ 4.1 and 4.2), while rules of Category 3 and 4 fit into our transformer hints (§4.3). A combination of abduction hints and transformer hints (§4.4) can be used to implement composite procedures such as Löb induction.

### 4.1 Abduction Hints

This section defines abduction hints to capture rules in Category 1 and 2:

\[
H * [G'] \models A \quad \triangleq \quad H * G' \vdash A
\]

Here, we give some hypothesis \(H \in \Delta\) and current goal \(A\) as input to type class search, and receive the new goal \(G'\) as an output, indicated by the square brackets. Given some \(H\) and \(A\), we want to find a ‘good’ new goal \(G'\)—which might not exist. If a good \(G'\) cannot be found, we start the search again for a different \(H \in \Delta\). We leave ‘good’ undefined, but consider False and \(H \leftrightarrow G\) bad choices since they will make the proof automation get stuck, or loop.

The format of abduction hints directly represents hints of Category 2, but what about Category 1? Category 1 is encoded by performing a technical trick by Mulder et al. [2022], relying on the fact that \(G' \vdash A\) implies \(\text{True} * G' \vdash A\). Since \(\Delta \vdash \text{True}\) vacuously holds, we can pretend to have \(\text{True} \in \Delta\) for the purpose of fitting Category 1 into Category 2. To account for the case where a priority of rules is desired (some Category 1 rules should be tried either before or after Category 2 rules), we define two syntactical markers \(\varepsilon_0 \triangleq \text{True}\) and \(\varepsilon_1 \triangleq \text{True}\). Our proof search strategy will always find \(\varepsilon_0 \in \Delta\) before any actual hypothesis in \(\Delta\), while \(\varepsilon_1 \in \Delta\) will always be found last. This technique is similar to techniques by Gonthier et al. [2011], where multiple equivalent definitions are used to obtain proof automation rules with different priorities.

The proof search strategy proceeds as follows. If our goal is \(\Delta \vdash A\), use type classes to find \(H \in \Delta\) and \(G'\) such that \(H * [G'] \not\models A\). Continue with goal \(\Delta' \vdash G'\), where \(\Delta'\) is obtained from context \(\Delta\) by removing \(H\), unless \(H\) is persistent or equal to \(\varepsilon_0\) or \(\varepsilon_1\).

As an example, these abduction hints implement two cases of the strategy for logical atomicity:

\[
\begin{align*}
\text{ABDUCT-WP-VAL} & \quad \varepsilon_0 \star \left[ T \Rightarrow T \Phi \cdot v \right] \not\equiv \text{wp } v \{ \Phi \} \\
\text{ABDUCT-SYM-EX-LOGATOM} & \quad \vdash (\langle x. L \rangle \cdot e \langle v. \Psi \rangle_E) \\
& \quad \varepsilon_0 \star \left[ \langle x. L \rangle \mid v. U \Rightarrow \text{wp } K[v] \{ \Phi \} \right]_{T \setminus \varepsilon_0} \not\equiv \text{wp } K[e] \{ \Phi \}
\end{align*}
\]

Both rules will be directly applied (indicated by \(\varepsilon_0\)) if the goal matches the conclusion and the side-conditions can be solved. After applying a rule, the goal will be replaced by the part between
square brackets [ and ]. To make Diaframe 2.0 use these hints, one provides type class instances of above form—which requires a proof of the claimed entailment. Hints thus serve two purposes: they both implement the proof search strategy and prove it sound.

4.2 Near-applicability
Diaframe 2.0 can apply abduction hints when the logical state or current goal nearly matches a rule. Let us demonstrate this on the rule REC-APPLY from §3.3 to apply Löb induction hypotheses. This rule is monolithic since it takes care of two tasks: apply a weakest precondition below an evaluation context $K$ in the goal, and find a weakest precondition below a magic wand in the context $\Delta$. In the implementation in Diaframe 2.0, this rule is decomposed in separate hints for each task:

\[
\begin{align*}
\text{ABDUCT-WP-BIND} & \quad \text{wp } e \{\Phi\} \ast [\forall v. \Phi v \rightarrow \text{wp } K[v] \{\Psi\}] \# \text{wp } K[e] \{\Psi\} \\
\text{ABDUCT-WAND} & \quad H \ast [G'] \not\equiv G \\
& \quad (L \leftarrow H) \ast [L \ast G'] \not\equiv G
\end{align*}
\]

The key hypothesis wp $e \{\Phi\}$ of ABDUCT-WP-BIND does not precisely match the induction hypothesis $\Box(\Delta \leftarrow \text{wp } e \{\Phi\})$ that was generated by LÖB. To address this, abduction hints are closed under the connectives of separation logic by recursive rules such as ABDUCT-WAND (similar recursive rules exist for other connectives of separation logic, e.g., universal quantification). The recursive rules ensure that every abduction hint $A \ast [G'] \not\equiv G$ is not just relevant when $A \in \Delta$, but also when for example $(H \leftarrow A) \in \Delta$ or $(H_1 \leftarrow H_2 \leftarrow (A \ast B)) \in \Delta$.

These two rules also come in handy for situations besides REC-APPLY. For example, ABDUCT-WAND and similar recursive rules are used for Löb induction in refinement proofs. The abduction hint ABDUCT-WP-BIND is useful when verifying examples with higher-order functions. There, one might have a specification for a closure in the proof context, and ABDUCT-WP-BIND makes it possible to use this specification in any evaluation context.

4.3 Transformer Hints for Modalities
This section defines transformer hints, which capture rules in Category 3. We show how these hints support the introduction of the $\Box$ and $\triangleright$ modalities. Transformer hints come in two flavors—hypothesis and context transformer hints:

\[
\begin{align*}
\text{HYP} & \quad H, T \sim_{\text{hyp}} [T'] \Leftrightarrow T' \vdash (H \leftarrow T) \\
\text{CTX} & \quad \Delta, T \sim_{\text{ctx}} [G] \Leftrightarrow (\Delta \vdash G) \implies (\Delta \vdash T)
\end{align*}
\]

Like before, terms between brackets are outputs of type class search, the other terms are inputs. We use the class $T$ to indicate goals on which transformer hints should be used—this class is disjoint from ordinary goals $G$ on which abduction hints should be used. Examples of transformer hints are the introduction rules for the later ($\triangleright$) and persistence ($\Box$) modalities:

\[
\begin{align*}
\triangleright H, \triangleright G \sim_{\text{hyp}} [\triangleright (H \leftarrow G)] & \quad \text{no } H \in \Delta \text{ prefixed by } \triangleright \\
\Delta, \triangleright G \sim_{\text{ctx}} [G] & \quad \text{all } H \in \Delta \text{ are persistent}
\end{align*}
\]

If we are proving a goal of shape $\Delta \vdash T$, the proof search strategy takes the following steps:

\[\text{One might note that a hypothesis transformer hint } H, T \sim_{\text{hyp}} [T'] \text{ is logically equivalent to an abduction hint } H \ast [T'] \not\equiv T. \text{ While logically equivalent, these hints are different operationally. Hypothesis transformer hints only act on the head-symbol/modality of hypothesis } H, \text{ while abduction hints will look beneath connectives of } H \text{ using rules like ABDUCT-WAND, as explained in §4.2.}\]
(1) Find \( H \in \Delta \) and \( T' \) such that \( H,T \vdash_{\text{hyp}} [T'] \). Continue with goal \( \Delta' \vdash T' \), where \( \Delta' \) is the context \( \Delta \) in which \( H \) is removed. Unlike abduction hints, \( H \) is also removed if it is persistent, and \( \varepsilon_0 \) and \( \varepsilon_1 \) are not detected by these hints.

(2) Otherwise, find \( G \) such that \( \Delta,T \vdash_{\text{ctx}} [G] \). Continue with goal \( \Delta \vdash G \).

One can check that the transformer hints for the later modality first ‘revert’ and strip the later off of all hypotheses with a later, and only then introduce the later modality.

### 4.4 Transformer Hints for Other Rules

In §4.3, we saw that transformer hints are flexible enough to support the introduction of modalities. In this section, we show how transformer hints can be combined with abduction hints to support rules in Category 4, like \textsc{au-intro} and \textsc{löb}. Recall our instance of the proof strategy for the introduction rule for atomic updates from §3.3:

- If \( G = \langle \bar{x}. L \mid v. U \Rightarrow \Phi \rangle_E \), and \( G \) does not occur in our environment \( \Delta \). Apply \textsc{au-intro}, the new goal has shape \( \Delta \vdash E_{\text{pre}} \exists \bar{x}. L \ast \left( (L \Rightarrow ?E' \Rightarrow E) \land (\forall v. U \Rightarrow ?E' \Rightarrow E \mid G) \right) \).

Note that \( \Delta \) occurs on the right-hand-side of the turnstile, so this rule falls outside the first two categories. Checking that \( \langle \bar{x}. L \mid v. U \Rightarrow \Phi \rangle_E \not\in \Delta \) is crucial—proof search will otherwise loop on the goal \( \langle \bar{x}. L \mid v. U \Rightarrow \Phi \rangle_E \vdash \langle \bar{x}. L \mid v. U \Rightarrow \Phi \rangle_E \). On such a goal, we want to use the abduction hint \( G \ast [\text{True}] \not\models G \), instead of applying \textsc{au-intro}. We therefore add an intermediate form \( \textsc{au-pre}(\bar{x}, \alpha, v, \beta, \Phi, E) \not\models \langle \bar{x}. \alpha \mid v. \beta \Rightarrow \Phi \rangle_E \) and a combination of transformer and abduction hints:

\[
\textsc{au-intro-pre} \\
\varepsilon_1 \ast \left[ \textsc{au-pre}(\bar{x}, \alpha, v, \beta, \Phi, E) \right] \not\models \langle \bar{x}. \alpha \mid v. \beta \Rightarrow \Phi \rangle_E
\]

\[
\textsc{au-intro-go} \\
\Delta, \textsc{au-pre}(\bar{x}, \alpha, v, \beta, \Phi, E) \vdash_{\text{ctx}} E_{\text{pre}} \exists \bar{x}. \alpha \ast \left( \alpha \Rightarrow ?E' \Rightarrow E \ast \Delta \right) \land (\forall v. \beta \Rightarrow ?E' \Rightarrow E \mid \Phi \mid v)
\]

Since \textsc{au-intro-pre} is a last-resort hint (indicated by \( \varepsilon_1 \)), we ensure that the assumption hint \( G \ast [\text{True}] \not\models G \) is preferred. After applying \textsc{au-intro-pre}, the proof search strategy tries to establish \textsc{au-pre}. This will directly find \textsc{au-intro-go}, and enact \textsc{au-intro}.

The collection of these hints gives precisely the required behavior. By introducing a new construct \textsc{au-pre} and giving above hints, we are quite literally ‘programming the proof search’ to act according to our wishes. A similar approach works for performing \textsc{löb} induction, where we use two intermediate goals \( \textsc{lob}_{\text{pre}} (G) \not\models G \) and \( \textsc{lob}_{\text{post}} (G) \not\models G \), and the following hints:

\[
\varepsilon_1 \ast [\textsc{lob}_{\text{pre}} (\wp ((\text{rec } f x := e) v) \{\Phi\})] \not\models \wp ((\text{rec } f x := e) v) \{\Phi\}
\]

\[
\Delta, \textsc{lob}_{\text{pre}} (G) \vdash_{\text{ctx}} (\Diamond (\Delta \Rightarrow G)) \Rightarrow \textsc{lob}_{\text{post}} (G)
\]

\[
\varepsilon_0 \ast [\wp e (\text{rec } f x := e) / f[v/x] \{\Phi\}] \not\models \textsc{lob}_{\text{post}} (\wp ((\text{rec } f x := e) v) \{\Phi\})
\]

By delegating \textsc{löb} induction to the \textsc{lob}_{\text{pre}} and \textsc{lob}_{\text{post}} constructs, we can easily reuse the procedure for refinement judgments. We simply need to add variants of the first and third hint for the refinement judgment. The second hint we can reuse because it is generic in the goal \( G \). This modularity is useful for the full-blown version of automatic \textsc{löb} induction in the supplementary material. The full-blown version generalizes over variables and thus has a more sophisticated version of the second hint.
4.5 Overview of the Proof Search Strategy

We now give a more formal description of the proof search strategy that underpins Diaframe 2.0. It acts on goals of the form \( \Delta \vdash G \), where \( G \) is defined roughly according to the following grammar:

\[
\begin{align*}
\text{atoms} & \quad A ::= \ldots \\
\text{transformers} & \quad T ::= \ldots \\
\text{left-goals} & \quad L ::= \left[ \phi \right] \mid A \mid L \ast L \mid \exists x. L \\
\text{unstructured} & \quad U ::= \left[ \phi \right] \mid A \mid U \ast U \mid \exists x. L \mid \forall x. U \mid L \leftarrow U \mid E_1 \supseteq E_2 U \\
\text{goals} & \quad G ::= \forall x. G \mid U \leftarrow G \mid A \mid E_1 \supseteq E_2 \exists x. L \ast G \mid T
\end{align*}
\]

To prove \( \Delta \vdash G \), the strategy proceeds by case analysis on \( G \):

(1) \( G = \forall x. G' \). Introduce variable \( x \) and continue.
(2) \( G = U \leftarrow G' \). Introduce \( U \) into the context and similar to Diaframe 1.0, ‘clean’ it. That is, eliminate existentials, disjunctions and separating conjunctions.
(3) \( G = A \). Look for an abduction hint from some \( H \in \Delta \) to \( A \). That is, find a side-condition \( G' \) such that \( H \ast [G'] \not\models A \). Continue with \( \Delta \setminus H \vdash G' \).
(4) \( G = E_1 \supseteq E_2 \exists x. L \ast G' \). Use the existing procedure of Diaframe 1.0 [Mulder et al. 2022] to solve these goals. Roughly, that is, first, use associativity of \( \ast \) to obtain either:
   (a) \( L = \left[ \phi \right] \). Prove \( \exists x. \phi \), then continue with proving \( G' \).
   (b) \( L = A \). Now, find a bi-abduction hint from some \( H \in \Delta \) to \( A \). That is, find a side-condition \( L' \) and residue \( U \) such that \( \forall y. H \ast L \leftarrow E_1 \supseteq E_2 \exists x. A \ast U \). Our new goal will be of shape \( \Delta \setminus H \vdash E_1 \supseteq E_2 \exists y. L' \ast (\forall x. U \leftarrow G') \), which also fits our grammar.
(5) \( G = T \). Try the following, in order:
   (a) Find \( H \in \Delta \) and \( T' \) such that \( H, T \not\models^\text{hyp} [T'] \). Continue with goal \( \Delta \setminus H \vdash T' \).
   (b) Otherwise, find \( G' \) such that \( \Delta, T \not\models^\text{ctx} [G'] \). Continue with goal \( \Delta \vdash G' \).

\textbf{Diaframe 1.0 vs Diaframe 2.0.} There are two main reasons why Diaframe 1.0’s bi-abduction hints cannot express the proof search strategies from \( \S \) 2.3 and 3.3. Firstly, context transformer hints (Item 5b) have a shape that is simply incompatible with Item 4b. Secondly, the side-conditions of abduction hints are in \( G \), while those of bi-abduction hints are in \( L \). Goals \( G \) are strictly more flexible than left-goals \( L \), giving abduction hints the additional power to express proof strategies for program specification styles. One could attempt to extend the grammar of \( L \), but then we risk ending up in a goal of shape \( (\forall x. G_1) \ast (\forall y. G_2) \) after Item 4b, causing the proof search to get stuck.

5 EVALUATION

We evaluate our proof automation on four sets of benchmarks. To evaluate Design goal #1, we compare to Voila [Wolf et al. 2021]—a proof outline checker for logical atomicity (\( \S \) 5.1). We discuss the differences in the underpinned logics, and the performance and proof burden of the proof automation of both tools. To evaluate Design goal #2, we redo some of the trickier examples in the Iris literature: an elimination stack, and Harris et al. [2002]’s RDCSS (restricted double-compare single-swap) (\( \S \) 5.2). Besides reverifying existing examples, we use our results to verify logical atomicity of the Michael-Scott queue [Michael and Scott 1996] (\( \S \) 5.3). This queue is known to be linearizable, but we are not aware of a mechanized proof of logical atomicity. For refinements in concurrent separation logic there exist—to the best of our knowledge—no existing semi-automated tools. We thus compare to existing interactive proofs done in ReLoC (\( \S \) 5.4).
5.1 Comparison to Logical Atomicity Proofs in Voila

We verify the 9 examples from Voila’s evaluation suite in Diaframe 2.0. Details can be found in Fig. 7. There are some differences between Voila and Diaframe 2.0 that are important to point out. Voila is based on the TaDa logic [da Rocha Pinto 2016; da Rocha Pinto et al. 2014], whose notion of logical atomicity inspired that of Iris, but is slightly different. To give a specification for a logically atomic triple in TaDa, one needs to define an abstraction around the resources, in the form of a region (akin to an invariant in Iris). This is not always required in Iris, which makes our specifications of e.g., cas_counter and inc_dec_counter a lot shorter.

Another difference is that Diaframe 2.0 is foundational (built in a proof assistant), while Voila is non-foundational. The main difference between foundational and non-foundational verification lies in the size of the Trusted Computing Base (TCB). Non-foundational tools typically have a large TCB, which may include external solvers, the bespoke program logic that underpins the tool, and the implementation of the proof automation. Foundational tools typically have a small TCB: just the definition of the operational semantics and the kernel of the proof assistant, the program logic and the proof automation need not be trusted.

Finally, Voila is a proof outline checker, requiring the user to specify key steps in the proof of a logically atomic triple. In particular, one needs to specify when regions or atomic specifications need to be used, and when the linearization point happens. This offers an improvement over fully interactive proofs, but does not achieve the degree of automation Diaframe 2.0 provides—for all but 2 examples, we can find the linearization point automatically. Wolf et al. [2021] explicitly do not attempt to build an automated verifier for logical atomicity, about which they remark:

Automated verifiers, on the other hand, significantly reduce the proof effort, but compromise on expressiveness and require substantial development effort, especially, to devise custom proof search algorithms. It is in principle possible to increase the automation of proof checkers by developing proof tactics, or to increase the expressiveness of automated verifiers by developing stronger custom proof search algorithms. However, such developments are too costly for the vast majority of program logics, which serve mostly a scientific or educational purpose.

We summarize aggregated data from Fig. 7. On average, Diaframe 2.0 has ca. 0.4 lines of proof burden per line of implementation (63 lines of proof burden on 156 lines of implementation), while
Voila has, in our count, 1.7 lines of proof burden per line of implementation. The total proof burden over these 9 examples is reduced by a factor of about 4, from 246 lines in Voila to 63 lines in Diaframe 2.0. For 6 out of the 9 examples, the logically atomic triples can be verified by Diaframe 2.0 without any help from the user. This shows we achieve Design goal #1—full automation for ‘simple’ proofs of logical atomicity. The other three examples require some help for arithmetic modulo $n$ (bounded_counter), case distinctions which need to be performed at a specific place in the proof (ticket_lock and bag_stack), or custom hints with non-automatable proofs (bag_stack).

### 5.2 Comparison to Complex Interactive Logical Atomicity Proofs in Iris

To ensure Diaframe 2.0 is usable in interactive proofs of ‘complex’ programs (Design goal #2), we partially automate two existing interactive proofs in Iris. The results are shown in Fig. 8. Since these examples are challenging—both feature “helping”, where the linearization point is delegated to another thread—full proof automation is not achieved. The proof burden was reduced by a factor of 4. We found that some intermediate lemmas were no longer necessary, as their effects were applied automatically. Most of the ‘easier’ parts of the verifications of these programs (such as recursive calls on a failing CAS) could be completely discharged by Diaframe 2.0. This allowed us to focus on the interesting part of the verification. In these examples, we have seen 4 patterns where the proof automation may need assistance: (a) linearization points for operations that do not logically alter the state, (b) case distinctions whose necessity requires ‘foresight’/human intuition, (c) pure side-conditions that are too hard for Diaframe, (d) mutation rules of recursive data structures. Items (c) and (d) can sometimes be overcome through appropriate hints in Diaframe 1.0. We leave good proof automation for Items (a) and (b) for future work. Vafeiadis [2010] also points out that Item (a) is very difficult in the context of CAVE.

### 5.3 Experiences Verifying Logical Atomicity of the Michael-Scott Queue

To evaluate the applicability of our proof automation on new proofs, we verify logical atomicity of the Michael-Scott queue. To our knowledge, this is a novel result. Contextual refinement is established by Vindum and Birkedal [2021], but logical atomicity is stronger and implies contextual refinement (we have worked this out in more detail in our artifact [Mulder and Krebbers 2023]). Our proof reuses some of their techniques (the persistent maps-to predicate), but represents the queue data structure invariant differently—in a way that is both natural, and allows suitable hints for mutating the queue. After establishing hints and pure automation for this data structure, most of the separation-logic reasoning can be dealt with automatically. The remaining proof burden consists of dealing with prophecy variables [Jung et al. 2020], for which our automation has partial support, and establishing pure facts outside of the reach of our automation—for this example, reasoning about lists without duplicates.

---

Wolf et al. [2021] report 0.8 line of proof annotation per line of code in Voila, which Diaframe 2.0 still improves on by a factor 2. We consider lines with explicit calls to open/close regions, and explicit uses of atomic specifications as proof work in Voila. It is unclear what counting metric is used by Wolf et al. [2021].

---

### Table 9

<table>
<thead>
<tr>
<th>name</th>
<th>impl</th>
<th>total</th>
<th>time</th>
<th>proof</th>
<th>interactive total</th>
<th>interactive proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>10</td>
<td>33</td>
<td>0:04</td>
<td>3</td>
<td>44</td>
<td>14</td>
</tr>
<tr>
<td>cell</td>
<td>27</td>
<td>64</td>
<td>0:31</td>
<td>4</td>
<td>128</td>
<td>68</td>
</tr>
<tr>
<td>coinflip</td>
<td>48</td>
<td>118</td>
<td>1:56</td>
<td>25</td>
<td>319</td>
<td>230</td>
</tr>
<tr>
<td>counter</td>
<td>19</td>
<td>65</td>
<td>0:25</td>
<td>5</td>
<td>225</td>
<td>63</td>
</tr>
<tr>
<td>lateearlychoice</td>
<td>26</td>
<td>88</td>
<td>0:22</td>
<td>16</td>
<td>129</td>
<td>62</td>
</tr>
<tr>
<td>namegen</td>
<td>9</td>
<td>70</td>
<td>0:11</td>
<td>26</td>
<td>112</td>
<td>68</td>
</tr>
<tr>
<td>Treiber stack (\preceq) stack with lock</td>
<td>46</td>
<td>136</td>
<td>1:02</td>
<td>36</td>
<td>185</td>
<td>124</td>
</tr>
<tr>
<td>symbol</td>
<td>28</td>
<td>112</td>
<td>1:38</td>
<td>27</td>
<td>376</td>
<td>234</td>
</tr>
<tr>
<td>ticket lock (\preceq) spin lock</td>
<td>17</td>
<td>85</td>
<td>0:59</td>
<td>7</td>
<td>266</td>
<td>120</td>
</tr>
<tr>
<td>various</td>
<td>54</td>
<td>158</td>
<td>3:34</td>
<td>30</td>
<td>582</td>
<td>372</td>
</tr>
<tr>
<td>total</td>
<td>284</td>
<td>929</td>
<td>10:42</td>
<td>179</td>
<td>2366</td>
<td>1355</td>
</tr>
</tbody>
</table>

Fig. 9. Statistics on proof automation for ReLoC. Each row contains the name of the verified example, lines of implementation, total amount of lines, verification time in minutes:seconds, and lines of proof burden—also for the original, interactively constructed version of the example.

Challenging verifications like this will usually not be successful the first time, and some amount of time must be spent figuring out the reason for failure. Three typical problems occur during failed verifications: (a) faulty specifications or invariants (b) missing or faulty hints for ghost resources or recursive data structures (c) the default proof search strategy is not sufficient. The general approach for debugging these problems is to let Diaframe 2.0 perform a fixed number of automation steps, instead of letting it run until it gets stuck. This allows the user to determine when the strategy takes a wrong turn, and act accordingly: change invariants, add hints, or manually perform a part of the proof. Diaframe 2.0 provides some tools for debugging a failing type class search for hints.

### 5.4 Comparison to Interactive Refinement Proofs in ReLoC

We evaluate our automation on 10 out of the 13 concrete examples from the ReLoC repository. The 3 remaining examples feature "helping", which is currently unsupported by our refinement proof automation. Statistics on the examples can be found in Fig. 9. The proof of ticket lock \(\preceq\) spin lock differs slightly from the original proof: instead of relying on ReLoC’s logically atomic relational specifications, we use Iris’s regular logically atomic specifications (§3) for the same effect.\(^\text{6}\)

We summarize some aggregated data from Fig. 9. On average, the proof size is reduced by a factor of 7 (179 vs 1355 lines of proof burden), coming down to 0.6 line of proof burden per line of implementation. For the largest refinement example, which proves that the Treiber stack contextually refines a course grained stack, we still reduce the proof size by over a factor of 3. Assistance from the user is required in the same cases as those discussed in §5.2. Additionally, it may be necessary to manually establish an invariant like in §2, or to manually perform right-hand side execution. A tactic \texttt{iStepR} is available for this last case.

### 6 RELATED WORK

**Viper.** Viper [Müller et al. 2016] is a non-foundational tool for automated verification using separation logic. Viper provides a common verification language, which is used as a backend of verification tools for a number of different program specification styles. Aside from functional

---

\(^{6}\)We believe it is folklore that logically atomic triples can be used in refinement proofs, but have not seen it worked out. In the implementation, this requires adding a slightly altered version of atomic updates, and accompanying hints.
correctness, Viper is used for logical atomicity in the TaDA logic [Wolf et al. 2021] (called Voila) and the security condition non-interference [Eilers et al. 2021]. An extensive comparison between Voila and our automation for logical atomicity can be found in §5.1. In summary, we show an average proof size reduction by a factor 4, and we support more complicated examples (RDCCS, elimination stack, and the Michael-Scott queue).

With regard to extensibility, Viper has the same goal as Diaframe 2.0—to provide a common verification backend that can handle multiple specification styles. There are some notable differences that make the two approaches difficult to compare in detail. First, Viper targets non-foundational verification instead of foundational verification in a proof assistant (see §5.1 for a discussion on the differences). Second, the embedding into Viper’s verification language is a syntactic program transformation that is performed before verification, while Diaframe 2.0 operates directly on program specifications during the verification. Third, Viper uses separation logic based on implicit dynamic frames [Parkinson and Summers 2011], which is different from Iris’s separation logic.

**Automated linearizability checkers.** CAVE [Henzinger et al. 2013; Vafeiadis 2010], Poling [Zhu et al. 2015] and Line-up [Burckhardt et al. 2010] are automated non-foundational tools for establishing linearizability. CAVE uses shape analysis to find linearization points, and Line-up uses model checking to refute linearizability. Poling extends CAVE with support for external linearization points. These tools use the trace-based formulation of linearizability [Herlihy and Wing 1990], which is less compositional than contextual refinement and logical atomicity. Poling does not support future-dependent linearization points, which are present in algorithms such as RDCCS and the Michael-Scott queue, and Line-up does not support non-deterministic concurrent data structures. The advantage of restricting supported target programs is that these tools do not need much user assistance.

**Verified concurrent search data structures.** Krishna et al. [2020, 2021] develop methods to prove logical atomicity of a particular class of concurrent algorithms: concurrent search structures. Their key idea is to subdivide the verification of a data structure into two parts: the verification of a *template algorithm* and verifying that a data structure is an instance of the template. The verification of the template algorithm is done interactively in Iris using the Iris Proof Mode. The template-instance verification is done automatically using the tool GRASShopper [Piskac et al. 2014]. This work is thus only partly foundational. To obtain a full foundational proof, it would be interesting to investigate if our work could be used to automate the verification of the instances currently done using GRASShopper.

**Automated verifiers for concurrent refinements.** Civl [Hawblitzel et al. 2015; Kragl and Qadeer 2021] is an automated tool for establishing refinement of concurrent programs. Their approach is based on establishing multiple layers of refinement, where each layer simplifies and refines the previous layer. By employing the Boogie verifier [Barnett et al. 2005], Civl can automatically prove these layered refinements—although inductive invariants and non-interference conditions need to be specified by the user. This approach has also been shown to scale to larger examples: in particular, Civl has been used to verify a concurrent garbage collector of significant size. Civl focuses on refinements in general, and not on linearizability in particular. Linearizability has been established for e.g., the Treiber stack [Treiber 1986], but not for more complex examples such as the Michael-Scott queue.

**Other logics for linearizability.** Our work builds upon Iris, which consolidates prior work on logical atomicity and refinements in separation logic [da Rocha Pinto et al. 2014; Dreyer et al. 2010; Jacobs and Piessens 2011; Svendsen et al. 2013; Turon et al. 2013]. Aside from Iris, there are a number of other expressive logics for linearizability that employ different approaches to compositional. 
While none of this work addresses the challenge of automating linearizability proofs, we briefly discuss some of this work. FCSL [Nanevski et al. 2019; Sergey et al. 2015] is a Coq-based separation logic, where linearizability can be established by keeping track of timestamped histories. Liang and Feng [2013] have designed a program logic based on rely-guarantee for proving linearizability. They can handle challenging examples (such as RDCSS), but their proofs are not mechanized in a proof assistant. Kim et al. [2017] verify linearizability and liveness of a C implementation of an MCS lock using the certified concurrent abstraction layer framework in Coq [Gu et al. 2015].

7 FUTURE WORK

We would like to improve the usability of Diaframe 2.0. As can be seen in Fig. 2, variable names are automatically generated by Coq. This can make it difficult to relate generated Coq goals to the program subject to verification. A further improvement would be to avoid interaction with Coq altogether by using annotations in source code, akin to auto-active verification tools [Leino and Moskal 2010]. RefinedC [Sammler et al. 2021] demonstrates that a proof strategy in Iris can be used as a backend for a foundational auto-active tool for functional correctness. For refinement and logical atomicity it is currently unclear what suitable annotations would look like.

We focused on automating the separation logic part of refinement and logical atomicity proofs. To automate the pure conditions that arise in the verification, we use standard solvers from Coq such as lia and set_solver. It would be interesting to investigate if recent approaches to improve pure automation Coq could be incorporated [Besson 2021; Czajka 2020; Ekici et al. 2017]. We focused on proof strategies for refinement and logical atomicity, but we conjecture that the generic Diaframe 2.0 strategy is more widely applicable. We would like to instantiate it with other logics and languages. We have some initial experiments for Similuris [Gäher et al. 2022] and λ-rust [Jung et al. 2018a]. Languages like Georges et al. [2022]’s capability machines, and logics like VST (which Mansky [2022] has recently ported to the Iris Proof Mode, and also supports logical atomicity) are also interesting targets. Finally, it would be interesting to investigate automation for recent work by Dang et al. [2022] on logical atomicity under weak memory.

As mentioned in the evaluation (§5), our proof automation cannot always automatically determine the required case distinctions for a proof. Additionally, we rely on backtracking to determine linearization points. A recent extension of Diaframe [Mulder et al. 2023] provides better support for disjunctions and avoids backtracking, which could address these problems.

ACKNOWLEDGMENTS

We thank Jules Jacobs for useful discussions, and the anonymous reviewers for their helpful feedback. This research was supported by the Dutch Research Council (NWO), project 016.Veni.192.259, and by generous awards from Google Android Security’s ASPIRE program.

REFERENCES


Received 2022-10-28; accepted 2023-02-25