Proof Automation for Linearizability in Separation Logic

IKE MULDER, Radboud University Nijmegen, The Netherlands
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

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.

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

Additional Key Words and Phrases: Separation logic, linearizability, fine-grained concurrency, refinement, logical atomicity, proof automation, Iris, Coq

ACM Reference Format:
Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 91 (April 2023), 30 pages. https://doi.org/10.1145/3586043

1 INTRODUCTION

Concurrent algorithms and data structures play an increasingly important role in modern computers. For efficiency, such algorithms and data structures often rely on *fine-grained concurrency*—they use low-level operations such as Compare And Swap (CAS) instead of high-level synchronization primitives such as locks. The "golden standard" of correctness for such data structures is *linearizability* [Herlihy and Wing 1990]. An operation on a concurrent data structure is linearizable if its effect appears to take place instantaneously, and if the effects of concurrently running operations always constitute a valid sequential history. This can be formalized by requiring that somewhere during every operation on the concurrent data structure, there exists a single atomic step which

Authors' addresses: Ike Mulder, Radboud University Nijmegen, The Netherlands, me@ikemulder.nl; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl.

2475-1421/2023/4-ART91
https://doi.org/10.1145/3586043

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, then there 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 $\langle P \rangle\, e\, \langle Q \rangle$. 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] 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.

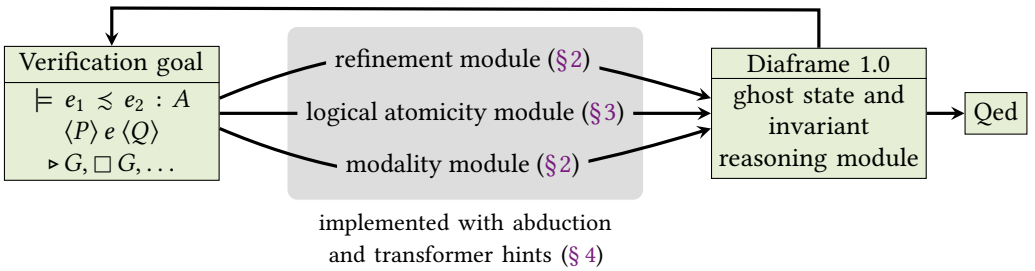

Fig. 1. Overview of the architecture of Diaframe 2.0.

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. 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 $\texttt{fg\_incrementer} \preceq_{\text{ctx}} \texttt{cg\_incrementer} : () \to () \to \mathbf{Z}$ that expresses that any behavior of `fg_incrementer` is a behavior of `cg_incrementer`. More precisely, a contextual refinement $e_1 \preceq_{\text{ctx}} e_2 : A$ 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.

```
1  Definition fg_incrementer : val :=
2    λ: <>,
3      let: "l" := ref #1 in
4      (rec: "f" <> :=
5        let: "n" := ! "l" in
6        if: CAS "l" "n" ("n" + #1) then
7          "n"
8        else
9          "f" #()).
10 Definition cg_incrementer : val :=
11   λ: <>,
12     let: "l" := ref #1 in
13     let: "lk" := newlock #() in
14     (λ: <>,
15       acquire "lk";;
16       let: "n" := ! "l" in
17       "l" ← "n" + #1 ;;
18       release "lk";;
19       "n").
20 Lemma fg_cg_incrementer_refinement :
21   ⊢ REL fg_incrementer << cg_incrementer : () → () → lrel_int.
22 Proof.
23   iStepsS.
24   iAssert (|={⊤}=> inv (nroot.@"incr")
25       (∃ (n : nat), x ↦ #n ∗ x0 ↦ₛ #n ∗ is_locked_r x1 false))%I
26       with "[-]" as ">#Hinv"; first iStepsS.
27   iSmash.
28 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 *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 ($\models e_1 \precsim e_2 : A$) 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 ($\vdash \models e_1 \precsim e_2 : A$). Here, $\models e_1 \precsim e_2 : A$ is a proposition in separation logic, which allows us to write refinements that are conditional on mutable state. For example, we can prove that $\ell_l \mapsto z * \ell_r \mapsto_s z \vdash \models !\,\ell_l \precsim !\,\ell_r : \mathbf{Z}$, *i.e.,* a load of ℓ_l contextually refines a load of ℓ_r, if both locations are valid pointers and point to the same value z. The *maps-to connectives* $\ell_l \mapsto z$ and $\ell_r \mapsto_s z$ represent the right to read and write to a location ℓ. Since we are reasoning about two programs (and thus, two heaps), ReLoC uses the subscripted $\mapsto_s$ (with 's' for specification) to indicate the heap of the right-hand side execution.

Proofs of ReLoC's refinement judgment $\models e_1 \precsim e_2 : A$ 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.

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:

$$\vdash \models \texttt{fg\_incrementer} \precsim \texttt{cg\_incrementer} : () \to () \to \mathbf{Z}. \tag{1}$$

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 `fg_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 $\boxed{\exists n.\, \ell_l \mapsto n * \ell_r \mapsto n * \mathsf{isLock}(v, \mathsf{false})}^{\mathcal{N}}$, *i.e.,* Proof Phase 2. (In Coq, we write `inv N R` for $\boxed{R}^{\mathcal{N}}$.) Iris's *invariant assertion* $\boxed{R}^{\mathcal{N}}$ states that there is a (shared) invariant with name $\mathcal{N}$, governing resources satisfying Iris assertion R. Since $\boxed{R}$ can be shared, accessing the resources in R must come 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 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 $\Box$ *modality*, *invariant reasoning*, and *Löb induction*) when necessary.

***Proof Phase 1*: Symbolic execution of outer closures and the $\Box$ modality.** Recalling our initial proof obligation $\vdash \models \mathtt{fg\_incrementer} \lesssim \mathtt{cg\_incrementer} : () \to () \to \mathbf{Z}$, we can start our proof by using REFINES-CLOSURE. This rule is applicable for any proof context Δ, where Δ stands for a list of assertions $P_1, \ldots, P_n$. We denote $\Delta \vdash Q$ for $P_1 * \ldots * P_n \vdash Q$.

Let us consider the premise of REFINES-CLOSURE: we need to prove $\vdash \Box (\models v_1\ () \lesssim v_2\ () : A)$. This mentions 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 Δ is thus empty), we can apply IRIS-$\Box$-INTRO, introducing the $\Box$ modality, and continue symbolic execution.

We can then use ALLOC-L, ALLOC-R and NEWLOCK-R to symbolically execute instructions on both sides. Our proof obligation now looks as follows:

$$\ell_l \mapsto 1, \ell_r \mapsto_s 1, \mathsf{isLock}(v, \mathsf{false}) \vdash \models (\mathrm{rec}\ldots) \lesssim (\lambda\ldots) : () \to \mathbf{Z} \tag{2}$$

We obtain two maps-to connectives $\ell_l \mapsto 1$ and $\ell_r \mapsto_s 1$ 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 ℓ. Similarly, $\mathsf{isLock}(v, \mathsf{false})$ 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 ℓ_l, meaning that concurrent calls to this closure will all try to write to the same location. However, only one thread can hold the $\ell_l \mapsto n$ 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 $\boxed{R}$. At any point during the verification, a resource R can be turned into $\boxed{R}$ using INV-ALLOC. This is called *invariant allocation*. The assertion $\boxed{R}$ is persistent, so unlike exclusive resources such as $\ell_l \mapsto 1$ and $\ell_r \mapsto_s 1$, the invariant assertion can be kept in the proof context when applying IRIS-□-INTRO. In Proof Phase 4, we will see how to access the invariant resource R.

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

$$\boxed{\exists n.\, \ell_l \mapsto n * \ell_r \mapsto_s n * \mathsf{isLock}(v, \mathsf{false})}^{\mathcal{N}} \vdash \models (\mathrm{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbf{Z}. \tag{3}$$

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 □ 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 $\square(\Delta \mathrel{-\!\!*} G)$ holds, but only *later*. We can get rid of this later ($\triangleright$) whenever our goal gets prefixed by a later, as witnessed by $\triangleright$-INTRO. Iris's □ modality ensures that the induction hypothesis $\Delta \mathrel{-\!\!*} G$ can be used more than once. This is reflected in the logic by the rules □-ELIM and □-DUP.

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

$$\left(\begin{array}{l}\boxed{\exists n.\, \ell_l \mapsto n * \ell_r \mapsto_s n * \mathsf{isLock}(v, \mathsf{false})}^{\mathcal{N}},\\ \square\,(\models (\mathrm{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbf{Z})\end{array}\right) \vdash \models (\mathrm{let}\, n := !\ell\ \ldots) \lesssim \ldots : \mathbf{Z} \tag{4}$$

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* ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$. The semantics of ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ is: assuming all invariants with names in $\mathcal{E}_1$ hold, then P holds, and additionally all invariants with names in $\mathcal{E}_2$ hold. The masks $\mathcal{E}$ 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 $\models_{\mathcal{E}} e_1 \lesssim e_2 : A$ have a mask parameter. We let $\mathcal{E} = \top$ when the mask is omitted.

REFINES-CLOSURE
$$\frac{\Delta \vdash \Box(\models v_1\ () \lesssim v_2\ () : A)}{\Delta \vdash \models v_1 \lesssim v_2 : () \to A}$$

IRIS-$\Box$-INTRO
$$\frac{\text{All hypotheses in } \Delta \text{ are persistent} \qquad \Delta \vdash G}{\Delta \vdash \Box G}$$

ALLOC-L
$$\frac{\forall \ell.\quad \Delta, \ell \mapsto v \vdash \models K[\ell] \lesssim e : A}{\Delta \vdash \models K[\mathtt{ref}\, v] \lesssim e : A}$$

ALLOC-R
$$\frac{\forall \ell.\quad \Delta, \ell \mapsto_{\mathrm{s}} v \vdash \models_{\mathcal{E}} e \lesssim K[\ell] : A}{\Delta \vdash \models_{\mathcal{E}} e \lesssim K[\mathtt{ref}\, v] : A}$$

NEWLOCK-R
$$\frac{\forall v.\quad \Delta, \mathsf{isLock}(v, \mathsf{false}) \vdash \models_{\mathcal{E}} e \lesssim K[v] : A}{\Delta \vdash \models_{\mathcal{E}} e \lesssim K[\mathtt{newlock}\ ()] : A}$$

INV-ALLOC
$$\frac{\Delta \vdash \triangleright R * (\boxed{R}^{\mathcal{N}} \mathrel{-\!\!*} \models e_1 \lesssim e_2 : A)}{\Delta \vdash \models e_1 \lesssim e_2 : A}$$

(a) Proof rules relevant for Proof Phase 1 and Proof Phase 2.

LÖB
$$\frac{\Delta, \triangleright \Box(\Delta \mathrel{-\!\!*} G) \vdash G}{\Delta \vdash G}$$

$\Box$-ELIM
$$\Box P \vdash P$$

$\Box$-DUP
$$\Box P \vdash \Box P * \Box P$$

UNFOLD-REC-L
$$\frac{\Delta \vdash \triangleright(\models e[(\mathrm{rec}\, f x := e)/f][v/x] \lesssim e' : A)}{\Delta \vdash \models (\mathrm{rec}\, f x := e)\, v \lesssim e' : A}$$

$\triangleright$-INTRO
$$\frac{\Delta' \text{ obtained from } \Delta \text{ by stripping at most one } \triangleright \text{ of every hypothesis} \qquad \Delta' \vdash P}{\Delta \vdash \triangleright P}$$

(b) Proof rules relevant for Proof Phase 3.

INV-ACCESS
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{R}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright R * \left(\triangleright R \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right)}$$

FUPD-ELIM
$$\frac{P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \qquad \Delta, Q \vdash {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} R}{\Delta, P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} R}$$

FUPD-INTRO
$$\frac{}{P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P}$$

LOAD-L
$$\frac{\Delta \vdash {}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists v.\, \ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} \models_{\mathcal{E}} K[v] \lesssim e : A)}{\Delta \vdash \models K[!\ell] \lesssim e : A}$$

REFINES-FUPD
$$\frac{\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\top} \models e_1 \lesssim e_2 : A}{\Delta \vdash \models_{\mathcal{E}} e_1 \lesssim e_2 : A}$$

CAS-L
$$\frac{\Delta \vdash {}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists v.\, \ell \mapsto v * \triangleright \begin{pmatrix} \ulcorner v = v_1 \urcorner * \ell \mapsto v_2 \mathrel{-\!\!*} \models_{\mathcal{E}} K[\mathtt{true}] \lesssim e : A \quad \wedge \\ \ulcorner v \neq v_1 \urcorner * \ell \mapsto v \mathrel{-\!\!*} \models_{\mathcal{E}} K[\mathtt{false}] \lesssim e : A \end{pmatrix}}{\Delta \vdash \models K[\mathrm{CAS}\, \ell\, v_1\, v_2] \lesssim e : A}$$

LOAD-R
$$\frac{\Delta, \ell \mapsto_{\mathrm{s}} v \vdash \models_{\mathcal{E}} e \lesssim K[v] : A}{\Delta, \ell \mapsto_{\mathrm{s}} v \vdash \models_{\mathcal{E}} e \lesssim K[!\ell] : A}$$

STORE-R
$$\frac{\Delta, \ell \mapsto_{\mathrm{s}} v \vdash \models_{\mathcal{E}} e \lesssim K[()] : A}{\Delta, \ell \mapsto_{\mathrm{s}} w \vdash \models_{\mathcal{E}} e \lesssim K[\ell \leftarrow v] : A}$$

REFINES-Z
$$\frac{}{\Delta \vdash \models z \lesssim z : \mathbf{Z}}$$

(c) Proof rules relevant for Proof Phase 4.

Fig. 3. A selection of the existing rules of Iris and ReLoC.

The INV-ACCESS rule shows the interplay between invariants and fancy updates. For an invariant $\boxed{R}^{\mathcal{N}}$ with name $\mathcal{N}$, if $\mathcal{N}$ is contained in $\mathcal{E}$, then removing $\mathcal{N}$ from $\mathcal{E}$ gives us access to the resource R. The original mask $\mathcal{E}$ can only be restored by handing back R. (Note that one only obtains R under a later modality ($\triangleright$). This is necessary since invariants in Iris are *impredicative* [Jung et al. 2018b; Svendsen and Birkedal 2014], *i.e.,* they may contain any resource, including invariants themselves. The later modality allows Iris to soundly deal with such cases, but for simple resources like $\ell \mapsto n$ (which are called *timeless* in Iris), the later modalities can be discarded.)

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

$$\begin{pmatrix} \boxed{R}^{\mathcal{N}}, \ell_l \mapsto n, \ell_r \mapsto_{\mathsf{s}} n, \mathsf{isLock}(v, \mathsf{false}), \\ (\triangleright R \twoheadrightarrow {}^{\top\setminus\mathcal{N}}\!\Rightarrow^{\top} \mathsf{True}), \\ \Box(\models (\mathsf{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbf{Z}) \end{pmatrix} \vdash \models_{\top\setminus\mathcal{N}} (\mathsf{let}\, n := n\ \ldots) \lesssim \ldots : \mathbf{Z}$$

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 CAS succeeds. After the load, we restore the invariant using FUPD-ELIM, and our hypothesis ($\triangleright R \twoheadrightarrow {}^{\top\setminus\mathcal{N}}\!\Rightarrow^{\top} \mathsf{True}$). We then use CAS-L. Like at the load, our invariant will provide us with some n' for which $\ell_l \mapsto n'$, and the 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 ℓ_l between the load and the CAS of the current thread.

The conjunction ($\wedge$) in CAS-L means that the proof splits into two separate proof obligations. In the successful case, we receive the updated $\ell_l \mapsto (n+1)$, together with the resource $\ulcorner n = n' \urcorner$. This embeds the pure fact $n = n'$ into Iris's separation logic. Likewise, in the failing case we receive the unchanged $\ell_l \mapsto n'$, together with the pure information $\ulcorner n \neq n' \urcorner$.

For case $n = n'$, the CAS succeeds, and the left-hand side expression will be returning n. After some pure reduction, our proof obligation becomes:

$$\begin{pmatrix} \boxed{R}^{\mathcal{N}}, \ell_l \mapsto (n+1), \ell_r \mapsto_{\mathsf{s}} n, \mathsf{isLock}(v, \mathsf{false}), \\ (\triangleright R \twoheadrightarrow {}^{\top\setminus\mathcal{N}}\!\Rightarrow^{\top} \mathsf{True}) \\ \Box(\models (\mathsf{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbf{Z}) \end{pmatrix} \vdash \models_{\top\setminus\mathcal{N}} n \lesssim (\mathtt{acquire}(v); \mathsf{let}\, n = !\ell_r\ldots) : \mathbf{Z}$$

At this point, we cannot restore the invariant: ℓ_l and ℓ_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 LOAD-R and 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 REFINES-Z.

For case $n \neq n'$, the CAS fails, and we receive back $\ell_l \mapsto n'$ unchanged. We restore the invariant. After some pure reduction our goal becomes the Löb induction hypothesis, concluding our proof:

$$\boxed{R}^{\mathcal{N}}, \Box(\models (\mathsf{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbf{Z}) \vdash \models (\mathsf{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbf{Z}$$

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 $\Box$ modality.

$$\textsc{exec-l}\quad \frac{\forall \vec{x}.\ \{L\}\ e_1\ \{U\} \qquad \text{atomic}\, e_1 \qquad e_1 \notin \mathit{Val} \qquad \Delta \vdash {}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists \vec{x}.\ L * \triangleright(\forall v.\ U \mathrel{-\!\!*} \models_{\mathcal{E}} K[v] \precsim e_2 : A)}{\Delta \vdash \models K[e_1] \precsim e_2 : A}$$

$$\textsc{exec-r}\quad \frac{\forall \vec{x}.\ \{L\}\ e_2\ \{U\}_{\mathrm{s}} \qquad \Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \exists x.\ L * (\forall v.\ U \mathrel{-\!\!*} \models_{\mathcal{E}} e_1 \precsim K[v] : A)}{\Delta \vdash \models_{\mathcal{E}} e_1 \precsim K[e_2] : A}$$

$$\textsc{val-z}\quad \frac{\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\top} \ulcorner z_1 = z_2 \urcorner}{\Delta \vdash \models_{\mathcal{E}} z_1 \precsim z_2 : \mathbf{Z}} \qquad \textsc{val-fun}\quad \frac{\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\top} \square(\models v_1\,() \precsim v_2\,() : A)}{\Delta \vdash \models_{\mathcal{E}} v_1 \precsim v_2 : () \to A}$$

$$\textsc{reloc-apply}\quad \frac{\Delta, \square(\Delta' \mathrel{-\!\!*} \models e_1 \precsim e_2 : A) \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \Delta' * {}^{\mathcal{E}}\!\Rrightarrow^{\top} (\forall v_1 v_2.\ A\, v_1\, v_2 \mathrel{-\!\!*} \models K_1[v_1] \precsim K_2[v_2] : B)}{\Delta, \square(\Delta' \mathrel{-\!\!*} \models e_1 \precsim e_2 : A) \vdash \models_{\mathcal{E}} K_1[e_1] \precsim K_2[e_2] : B}$$

Fig. 4. Derived proof rules for ReLoC suitable for proof automation.

- Proof Phase 2: *Not* introducing the $\square$ modality to allow the user to allocate the invariant.
- Proof Phase 3: Automatically performing Löb induction when it is necessary.
- Proof Phase 4: Symbolic execution where the preconditions are inside an invariant, followed by automatic application of induction hypothesis.

In this section, we give a description of our proof strategy that can handle these challenges. The strategy operates on goals $\Delta \vdash G$, where the grammar of G is given by:

$$G ::= \models_{\mathcal{E}} e_1 \precsim e_2 : A \mid \triangleright G \mid \square G \mid {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\ L * G.$$

(L are 'easy' goals like $\ell \mapsto v$, described in §4.5). If G is of one of the first three shapes, the strategy either provides a rule to apply, or stops. If G has the last shape, we reuse the existing automation of Diaframe 1.0 [Mulder et al. 2022] to handle invariants, which operates on precisely these goals.

Our proof strategy is the result of restating the original rules of ReLoC (Fig. 3) so that they can be applied systematically. Our new rules can be found in Fig. 4. We have verified in Coq that these rules can be derived from the existing rules of ReLoC and Iris. Rule EXEC-L generalizes symbolic execution rules like LOAD-L over the expression e_1, where $\forall \vec{x}.\ \{L\}\ e_1\ \{U\}$ is a Hoare triple for e_1. In Coq, $\forall \vec{x}.\ \{L\}\ e_1\ \{U\}$ is represented by a type class, so that given an expression e_1, the precondition L and postcondition U can be found automatically. Rule EXEC-R is similar, but uses Hoare triples $\forall \vec{x}.\ \{L\}\ e_2\ \{U\}_{\mathrm{s}}$ for the right-hand side. Finally, VAL-Z and VAL-FUN keep the fancy update around and have been generalized to all masks $\mathcal{E}$ so that the strategy can postpone closing invariants.

We can now give our proof search strategy for refinement judgments. Suppose the goal is $\Delta \vdash \models_{\mathcal{E}} e_1 \precsim e_2 : A$. We proceed by case distinction on both e_1 and e_2, and try the following rules in order (omitting some cases, *e.g.*, those related to pure reductions and higher-order functions):

(1) If e_1 and e_2 are values, apply rules similar to VAL-Z and VAL-FUN, depending on the type A.
(2) If e_1 is a value and e_2 is not, apply EXEC-R.
(3) If e_1 is not a value and $\mathcal{E} = \top$, try the following:
   (a) Find e with $e_1 = K[e]$ for which EXEC-L is applicable, otherwise
   (b) Try to find an induction hypothesis to apply with RELOC-APPLY, otherwise
   (c) If $e_1 := (\mathrm{rec}\, f x := b)\, v$, apply Löb induction with LÖB, followed by UNFOLD-REC-L.
(4) If e_1 is not a value and $\mathcal{E} \neq \top$, but e_2 *is* a value, apply REFINES-FUPD.

(5) If e_1 is not a value and $\mathcal{E} \neq \top$ and e_2 is not a value, there are two valid ways to proceed: either restore the invariant with REFINES-FUPD, or perform symbolic execution on the right with 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) $G = \Box G'$: Apply IRIS-□-INTRO, but *only* if all hypotheses in Δ are persistent. Stop otherwise.
(7) $G = \triangleright G'$: Apply rule ▷-INTRO to introduce the later and strip laters from the context.
(8) $G = {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G'$: Use proof automation from Diaframe 1.0 to make progress.

Verification of the example in Fig. 2. The strategy above is available using the `iStepsS` tactic in Coq. In the verification of the example in Fig. 2, the `iStepsS` tactic stops at line 24 after applying VAL-FUN for the second time. Item 6 (□ introduction) would be applicable, except that the proof context Δ is not persistent. Iris allows one to weaken the context before introducing the □ 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 `iAssert` tactic from the Iris Proof Mode.

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 `iSmash` tactic will backtrack on this choice, and is used in Fig. 2 to finish the proof. The sequence `iStepsS; iApply` REFINES-FUPD`; iStepsS` also constitutes a valid proof: `iStepsS` will not backtrack, and instead stop the proof automation. In that case, Iris's `iApply` 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 `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 $\Box G'$ 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).

```
1 Definition inc : val :=
2   rec: "f" "l" :=
3     let: "n" := ! "l" in
4     if: CAS "l" "n" ("n" + #1) then
5       "n"
6     else
7       "f" "l".
8 Global Instance inc_spec (l : loc) :
9   SPEC (z : Z), << l ↦ #z >> inc #l << RET #z; l ↦ #(z + 1) >>.
10 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 $\{P\}\ e\ \{\Phi\}$, 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 $\Phi\, w$. A naive specification is $\{\ell \mapsto z\}\ \texttt{inc}\ \ell\ \{v.\ulcorner v = z \urcorner * \ell \mapsto (z+1)\}$. This states that to execute `inc` ℓ, we need exclusive write-access to location ℓ, as indicated by the precondition $\ell \mapsto n$. Once `inc` ℓ terminates, it returns value z, and the $\ell \mapsto (z+1)$ 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 $\ell \mapsto z$ during `inc` ℓ, while it usually does *not* have exclusive access to $\ell \mapsto z$.

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:

$$\textsc{hoare-load}\quad \frac{}{\{\ell \mapsto v\}\ !\ell\ \{w.\ulcorner w = v \urcorner * \ell \mapsto v\}_{\mathcal{E}}} \qquad \textsc{hoare-inv-access}\quad \frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \text{atomic } e \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{R}^{\mathcal{N}} * P\right\}\ e\ \{v.\, Q\}_{\mathcal{E}}}$$

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* $\mathcal{E}$. 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` ℓ)', 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 `inc` using the following logically atomic triple:

$$\textsc{inc-logatom}\quad \langle z.\, \ell \mapsto z\rangle\ \texttt{inc}\ \ell\ \langle v.\, \ulcorner v = z \urcorner * \ell \mapsto (z+1)\rangle_{\emptyset}$$

We replaced { with ⟨, 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 *inside the logic*, with the following reasoning rules:

$$\textsc{la-inv}\quad \frac{\boxed{R}^{\mathcal{N}} \qquad \langle \vec{x}.\, \alpha * \triangleright R\rangle\ e\ \langle v.\, \beta * \triangleright R\rangle_{\mathcal{E}\setminus\mathcal{N}}}{\langle \vec{x}.\, \alpha\rangle\ e\ \langle v.\, \beta\rangle_{\mathcal{E}}} \qquad \textsc{la-hoare}\quad \frac{\Box\, \langle \vec{x}.\, \alpha\rangle\ e\ \langle v.\, \beta\rangle_{\mathcal{E}}}{\forall \vec{x}.\, \{\alpha\}\ e\ \{v.\, \beta\}_{\top}}$$

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

The curious use of binder z in INC-LOGATOM deserves a comment. Logically atomic triples allow a certain amount of interference from other threads, such as concurrent calls to `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 $\vec{x}$.

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 *weakest preconditions*:

$$\{P\}\ e\ \{\Phi\} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\}\right)$$

The weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ 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]

$$\textsc{la-def}\quad \langle \vec{x}.\, \alpha\rangle\ e\ \langle v.\, \beta\rangle_{\mathcal{E}} \triangleq \forall \Phi.\ \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \Phi\, v\rangle_{\top\setminus\mathcal{E}} \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\}$$

This expresses that for any postcondition Φ, to prove $\mathsf{wp}\ e\ \{\Phi\}$ it is enough to show an *atomic update* of the form $\langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \Phi\, v\rangle_{\top\setminus\mathcal{E}}$. Atomic updates represent the possibility to witness variables $\vec{x}$ for which α holds, at any instant. If one uses this possibility, one either needs to hand back ownership of this exact α to recover the atomic update, or hand back β to obtain $\Phi\, v$ (commit the linearization point). By quantifying over Φ, Iris makes sure that the only way to prove a logically atomic triple is by using the atomic update $\langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \Phi\, v\rangle_{\top\setminus\mathcal{E}}$.

Proving logically atomic triples. Proving a logically atomic triple $\langle \vec{x}.\, \alpha\rangle\ e\ \langle v.\, \beta\rangle_{\mathcal{E}}$ is a matter of 'just' proving a weakest precondition, *i.e.,* a goal $\Delta \vdash \mathsf{wp}\ e\ \{\Phi\}$. However, we need the atomic update to get temporary access to α and eventually get Φ. Atomic updates can be accessed as:

$$\textsc{au-access-iris}\quad \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q\rangle_{\mathcal{E}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \exists \vec{x}.\, \alpha * \left(\left(\alpha \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q\rangle_{\mathcal{E}}\right) \wedge \left(\forall v.\, \beta \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} Q\right)\right)$$

[1] The definition of logically atomic triples does not feature the □ modality to allow for *private* preconditions, *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 LA-HOARE.

$$\textsc{Löb}\ \frac{\Delta, \triangleright \Box(\Delta \mathrel{-\!\!*} G) \vdash G}{\Delta \vdash G} \qquad \textsc{unfold-rec}\ \frac{\Delta \vdash \triangleright \,\mathsf{wp}\, e[(\mathrm{rec}\, f x := e)/f][v/x]\ \{\Phi\}}{\Delta \vdash \mathsf{wp}\ (\mathrm{rec}\, f x := e)\, v\ \{\Phi\}}$$

$$\textsc{rec-apply}\ \frac{\Delta, \Box(\Delta' \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Psi\}) \vdash {}^{\top}\!\Rrightarrow^{\top} \Delta' * (\forall w.\, \Psi w \mathrel{-\!\!*} \mathsf{wp}\ K[w]\ \{\Phi\})}{\Delta, \Box(\Delta' \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Psi\}) \vdash \mathsf{wp}\ K[e]\ \{\Phi\}}$$

Fig. 6. Selection of Iris's proof rules for Löb induction on weakest preconditions.

This rule states that (similar to Iris's rule for invariants INV-ACCESS) an atomic update provides access to α by changing the masks of a fancy update (${}^{\mathcal{E}}\!\Rrightarrow^{\emptyset}$). After we obtain α, 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 α. 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 β, 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 ($\wedge$) in separation logic, where the separating conjunction ($*$) is more common. Regular conjunction corresponds to a form of internal choice: if one owns a regular conjunction $P \wedge Q$, 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 `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 ▷-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 $\vec{x}$ is cumbersome in Coq. Client verifications in Iris usually rely on the following rule:

$$\textsc{sym-ex-logatom}\ \frac{\vdash \langle \vec{x}.\, \alpha \rangle\, e\, \langle v.\, \beta \rangle_{\mathcal{E}} \qquad \Delta \vdash \left\langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \mathsf{wp}\ K[v]\ \{\Phi\} \right\rangle_{\top \setminus \mathcal{E}}}{\Delta \vdash \mathsf{wp}\ K[e]\ \{\Phi\}}$$

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

$$\textsc{au-intro}\ \frac{\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{?\mathcal{E}'} \exists \vec{x}.\, \alpha * \left((\alpha \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Delta) \wedge (\forall v.\, \beta \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} Q)\right)}{\Delta \vdash \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}}}$$

This rule asks us to show that opening some invariants in $\mathcal{E}$ gives us α. Additionally, we need to prove that obtaining α is non-destructive: the original context Δ can be restored. This ensures that when the implementation peeks at α, it does not affect the client. The other side of the conjunction shows that the atomic postcondition β 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: $\Delta \vdash \mathsf{wp}\ e\ \{\Phi\}$, by definition of logically atomic triples LA-DEF.

- Atomic updates: $\Delta \vdash \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}}$, when applying a known triple SYM-EX-LOGATOM.
- Goals of the form $\Delta \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G$, after introducing atomic updates AU-INTRO or to establish the precondition of heap operations such as load and CAS. The context Δ might contain atomic updates that should be eliminated via AU-ACCESS-IRIS.
- Goals prefixed by a later modality: $\Delta \vdash \triangleright G$, when using UNFOLD-REC after Löb induction.

Our proof search strategy for these goal 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 $\Delta \vdash \mathsf{wp}\ e\ \{\Phi\}$. 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 $\Delta \vdash {}^{\top}\!\Rrightarrow^{\top} \Phi\, v$.
(2) If $e = K[e']$, then either:
   (a) 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 EXEC-L.
   (b) We have a specification $\langle \vec{x}.\, L \rangle\ e'\ \langle v.\, U \rangle_{\mathcal{E}}$. Apply SYM-EX-LOGATOM, continue with new goal $\Delta \vdash \left\langle \vec{x}.\, L \mid v.\, U \Rrightarrow \mathsf{wp}\ K[v]\ \{\Phi\} \right\rangle_{\top \setminus \mathcal{E}}$.
   (c) Otherwise, try to find an induction hypothesis to use with REC-APPLY.
(3) If $e = (\mathtt{rec}\, f x := b)\, v$, *i.e.,* a possibly recursive function applied to a value v. Two cases:
   (a) There is no actual recursion, *i.e.,* f does not occur in b. Apply UNFOLD-REC and continue with new goal $\Delta \vdash \triangleright \mathsf{wp}\ b[v/x]\ \{\Phi\}$.
   (b) For recursive functions. Apply LÖB, then apply UNFOLD-REC. Continue with new goal $\Delta, \triangleright \square(\Delta \twoheadrightarrow \mathsf{wp}\ e\ \{\Phi\}) \vdash \triangleright \mathsf{wp}\ b[e/f][v'/x]\ \{\Phi\}$.[2] Note that Δ 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) $G = \triangleright G'$. Apply rule $\triangleright$-INTRO to introduce the later and strip laters from the context.
(5) $G = \langle \vec{x}.\, L \mid v.\, U \Rrightarrow G' \rangle_{\mathcal{E}}$. Two cases:
   (a) If $G \in \Delta$, directly use it to finish the proof. This situation occurs after applying the induction hypothesis with REC-APPLY.
   (b) Otherwise, we introduce the atomic update with AU-INTRO. Our new goal becomes $\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{?\mathcal{E}'} \exists \vec{x}.\, L * \left(\left(L \twoheadrightarrow {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Delta\right) \wedge \left(\forall v.\, U \twoheadrightarrow {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} G\right) \right)$.
(6) $G = {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G$. 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 `iSmash` instead of the `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 $\langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}}$ with 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 α.

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 α. Note that AU-ACCESS-IRIS is similar to the invariant accessing rule 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 $\wedge$): we either defer or commit the linearization point. We need to ensure this choice is not made too early,

[2] 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]

AU-ACCESS-DIAFRAME

$$\langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \exists \vec{x}.\, \alpha * \forall mv.\ ((\alpha * \ulcorner mv = \mathsf{None} \urcorner) \lor (\exists v.\, \beta * \ulcorner mv = \mathsf{Some}\, v \urcorner)) \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \mathsf{match}\, mv\, \mathsf{with}\, \mathsf{None} \Rightarrow \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}} \mid \mathsf{Some}\, v \Rightarrow Q\ \mathsf{end}$$

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 $\langle z.\, \ell \mapsto z \mid v.\, \ell \mapsto (z+1) \Rrightarrow Q \rangle_{\mathcal{E}}$ 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 $(\forall mv.\ \_ \lor \_ \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \_)$. Diaframe notices it can use this closing resource to restore the mask, so the goal becomes (note that mv is bound in G):

$$\Delta \vdash {}^{\emptyset}\!\Rrightarrow^{\emptyset} \exists\, mv.\ \big((\alpha * \ulcorner mv = \mathsf{None} \urcorner) \lor (\exists v.\, \beta * \ulcorner mv = \mathsf{Some}\, v \urcorner)\big) * G.$$

The `iSmash` 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 `iStepsS` 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 is recursive and generates a Löb induction hypothesis. Item 3a is a specialized version that handles the 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 $H * G' \vdash G$ is provable.

[3] This transformation is sound since both sides of the $\land$ feature a fancy update ${}^{\emptyset}\!\Rrightarrow^{\mathcal{E}}$ with the same mask.

(3) Rules of the form $(\Delta' \vdash G') \implies (\Delta \vdash G)$, where Δ' and G' can be calculated from Δ 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 Δ.

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

(1) If $G = \mathsf{wp}\ e\ \{\Phi\}$ and e is a value v, continue with $\Delta \vdash {}^{\top}\!\Rrightarrow^{\top} \Phi\, v$.
(2) If $G = \langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}}$, check if $G \in \Delta$. If so, we can continue with $\Delta \setminus G \vdash \mathsf{True}$
(3) If $G = \Box G'$, and all hypotheses in Δ are persistent, continue with $\Delta \vdash G'$. Note that the entailment $G' \vdash \Box G'$ does not hold. This rule is only valid because of the condition on Δ.
(4) If $G = \langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}}$, and the above Rule 2 is not applicable, apply AU-INTRO. The new goal has shape $\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{?\mathcal{E}'} \exists \vec{x}.\, L * \big((L \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Delta) \wedge (\forall v.\, U \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} G)\big)$. The Δ 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 Ltac [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:

$$H * [G'] \Vdash A \quad \triangleq \quad H * G' \vdash A$$

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 $H \mathrel{-\!\!*} G$ 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 $\mathsf{True} * G' \vdash A$. Since $\Delta \vdash \mathsf{True}$ vacuously holds, we can pretend to have $\mathsf{True} \in \Delta$ 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 $\varepsilon_0 \triangleq \mathsf{True}$ and $\varepsilon_1 \triangleq \mathsf{True}$. Our proof search strategy will always find $\varepsilon_0 \in \Delta$ before any actual hypothesis in Δ, 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 $H * [G'] \Vdash A$. Continue with goal $\Delta' \vdash G'$, where Δ' is obtained from context Δ by removing H, unless H is persistent or equal to ε_0 or ε_1.

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

ABDUCT-WP-VAL
$$\varepsilon_0 * \left[{}^{\top}\!\Rrightarrow^{\top} \Phi\, v\right] \Vdash \mathsf{wp}\ v\ \{\Phi\}$$

ABDUCT-SYM-EX-LOGATOM
$$\frac{\vdash \langle \vec{x}.\, L \rangle\ e\ \langle v.\, \Psi \rangle_{\mathcal{E}}}{\varepsilon_0 * \left[\left\langle \vec{x}.\, L \mid v.\, U \Rrightarrow \mathsf{wp}\ K[v]\ \{\Phi\} \right\rangle_{\top \setminus \mathcal{E}}\right] \Vdash \mathsf{wp}\ K[e]\ \{\Phi\}}$$

Both rules will be directly applied (indicated by ε_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 Δ. In the implementation in Diaframe 2.0, this rule is decomposed in separate hints for each task:

ABDUCT-WP-BIND
$$\mathsf{wp}\ e\ \{\Phi\} * \left[\forall v.\, \Phi\, v \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ \{\Psi\}\right] \Vdash \mathsf{wp}\ K[e]\ \{\Psi\}$$

ABDUCT-WAND
$$\frac{H * [G'] \Vdash G}{(L \mathrel{-\!\!*} H) * [L * G'] \Vdash G}$$

The key hypothesis $\mathsf{wp}\ e\ \{\Phi\}$ of ABDUCT-WP-BIND does not precisely match the induction hypothesis $\square(\Delta \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$ 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 $A * [G'] \Vdash G$ is not just relevant when $A \in \Delta$, but also when for example $(H \mathrel{-\!\!*} A) \in \Delta$ or $(H_1 \mathrel{-\!\!*} H_2 \mathrel{-\!\!*} (A * B)) \in \Delta$.

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 $\square$ and $\triangleright$ modalities. Transformer hints come in two flavors—hypothesis and context transformer hints:[4]

$$H, \mathcal{T} \overrightarrow{\sim}_{\mathsf{hyp}}\ [\mathcal{T}'] \triangleq \mathcal{T}' \vdash (H \mathrel{-\!\!*} \mathcal{T})$$
$$\Delta, \mathcal{T} \overrightarrow{\sim}_{\mathsf{ctx}}\ [G] \triangleq (\Delta \vdash G) \implies (\Delta \vdash \mathcal{T})$$

Like before, terms between brackets are outputs of type class search, the other terms are inputs. We use the class $\mathcal{T}$ 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 ($\square$) modalities:

$$\triangleright H, \triangleright G \overrightarrow{\sim}_{\mathsf{hyp}}\ [\triangleright(H \mathrel{-\!\!*} G)] \qquad \frac{\text{no } H \in \Delta \text{ prefixed by } \triangleright}{\Delta, \triangleright G \overrightarrow{\sim}_{\mathsf{ctx}}\ [G]} \qquad \frac{\text{all } H \in \Delta \text{ are persistent}}{\Delta, \square G \overrightarrow{\sim}_{\mathsf{ctx}}\ [G]}$$

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

[4] One might note that a hypothesis transformer hint $H, \mathcal{T} \overrightarrow{\sim}_{\mathsf{hyp}}\ [\mathcal{T}']$ is logically equivalent to an abduction hint $H * [\mathcal{T}'] \Vdash \mathcal{T}$. While logically equivalent, these hints are different operationally. Hypothesis transformer hints only act on the head-symbol/modality of hypothesis H, while abduction hints will look beneath connectives of H using rules like ABDUCT-WAND, as explained in §4.2.

(1) Find $H \in \Delta$ and $\mathcal{T}'$ such that $H, \mathcal{T} \overset{\rightarrow}{\sim}_{\text{hyp}} [\mathcal{T}']$. Continue with goal $\Delta' \vdash \mathcal{T}'$, where Δ' is the context Δ in which H is removed. Unlike abduction hints, H is also removed if it is persistent, and ε_0 and ε_1 are not detected by these hints.
(2) Otherwise, find G such that $\Delta, \mathcal{T} \overset{\rightarrow}{\sim}_{\text{ctx}} [G]$. 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 that transformer hints can be combined with abduction hints to support rules in Category 4, like AU-INTRO and LÖB. Recall our instance of the proof strategy for the introduction rule for atomic updates from §3.3:

- If $G = \langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}}$, and G does not occur in our environment Δ. Apply AU-INTRO, the new goal has shape $\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{?\mathcal{E}'} \exists \vec{x}.\, L * \Big(\big(L \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Delta\big) \wedge \big(\forall v.\, U \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} G\big)\Big)$.

Note that Δ occurs on the right-hand-side of the turnstile, so this rule falls outside the first two categories. Checking that $\langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}} \notin \Delta$ is crucial—proof search will otherwise loop on the goal $\langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}} \vdash \langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}}$. On such a goal, we want to use the abduction hint $G * [\text{True}] \Vdash G$, instead of applying AU-INTRO. We therefore add an intermediate form $\mathbf{AU}_{\text{pre}}(\vec{x}, \alpha, v, \beta, \Phi, \mathcal{E}) \triangleq \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \Phi \rangle_{\mathcal{E}}$ and a combination of transformer and abduction hints:

AU-INTRO-PRE
$$\varepsilon_1 * \big[\mathbf{AU}_{\text{pre}}(\vec{x}, \alpha, v, \beta, \Phi, \mathcal{E})\big] \Vdash \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \Phi \rangle_{\mathcal{E}}$$

AU-INTRO-GO
$$\Delta, \mathbf{AU}_{\text{pre}}(\vec{x}, \alpha, v, \beta, \Phi, \mathcal{E}) \overset{\rightarrow}{\sim}_{\text{ctx}} \Big[{}^{\mathcal{E}}\!\Rrightarrow^{?\mathcal{E}'} \exists \vec{x}.\, \alpha * \big((\alpha \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} L * \Delta) \wedge (\forall v.\, \beta \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Phi\, v)\big)\Big]$$

Since AU-INTRO-PRE is a last-resort hint (indicated by ε_1), we ensure that the assumption hint $G * [\text{True}] \Vdash G$ is preferred. After applying AU-INTRO-PRE, the proof search strategy tries to establish $\mathbf{AU}_{\text{pre}}$. This will directly find AU-INTRO-GO, and enact AU-INTRO.

The collection of these hints gives precisely the required behavior. By introducing a new construct $\mathbf{AU}_{\text{pre}}$ and giving above hints, we are quite literally 'programming the proof search' to act according to our wishes. A similar approach works for performing Löb induction, where we use two intermediate goals $\mathbf{löb}_{\text{pre}}\,(G) \triangleq G$ and $\mathbf{löb}_{\text{post}}\,(G) \triangleq G$, and the following hints:

$$\frac{(\text{rec}\, f x := e) \text{ performs recursion, } \textit{i.e.,}\ f \in \text{FV}(e)}{\varepsilon_1 * \big[\mathbf{löb}_{\text{pre}}\,\big(\text{wp}\,((\text{rec}\, f x := e)\, v)\,\{\Phi\}\big)\big] \Vdash \text{wp}\,((\text{rec}\, f x := e)\, v)\,\{\Phi\}}$$

$$\Delta, \mathbf{löb}_{\text{pre}}\,(G) \overset{\rightarrow}{\sim}_{\text{ctx}} \big[(\triangleright \Box(\Delta \mathrel{-\!\!*} G)) \mathrel{-\!\!*} \mathbf{löb}_{\text{post}}\,(G)\big]$$

$$\varepsilon_0 * \big[\triangleright\, \text{wp}\, e[(\text{rec}\, f x := e)/f][v/x]\,\{\Phi\}\big] \Vdash \mathbf{löb}_{\text{post}}\,\big(\text{wp}\,((\text{rec}\, f x := e)\, v)\,\{\Phi\}\big)$$

By delegating Löb induction to the $\mathbf{löb}_{\text{pre}}$ and $\mathbf{löb}_{\text{post}}$ 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 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:

$$\begin{aligned}
\text{atoms} \quad & A ::= \dots \\
\text{transformers} \quad & \mathcal{T} ::= \dots \\
\text{left-goals} \quad & L ::= \ulcorner \phi \urcorner \mid A \mid L * L \mid \exists x.\, L \\
\text{unstructureds} \quad & U ::= \ulcorner \phi \urcorner \mid A \mid U * U \mid \exists x.\, L \mid \forall x.\, U \mid L \mathrel{-\!\!*} U \mid {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} U \\
\text{goals} \quad & G ::= \forall x.\, G \mid U \mathrel{-\!\!*} G \mid A \mid {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G \mid \mathcal{T}
\end{aligned}$$

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

(1) $G = \forall x.\, G'$. Introduce variable x and continue.
(2) $G = U \mathrel{-\!\!*} G'$. 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 $H * [G'] \Vdash A$. Continue with $\Delta \setminus H \vdash G'$.
(4) $G = {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G'$. Use the existing procedure of Diaframe 1.0 [Mulder et al. 2022] to solve these goals. Roughly, that is, first, use associativity of $*$ to obtain either:
   (a) $L = \ulcorner \phi \urcorner$. Prove $\exists \vec{x}.\, \phi$, 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 $\forall \vec{y}.\, H * L \vdash {}^{\mathcal{E}_3}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, A * U$. Our new goal will be of shape $\Delta \setminus H \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} \exists \vec{y}.\, L' * (\forall \vec{x}.\, U \mathrel{-\!\!*} G')$, which also fits our grammar.
(5) $G = \mathcal{T}$. Try the following, in order:
   (a) Find $H \in \Delta$ and $\mathcal{T}'$ such that $H, \mathcal{T} \overrightarrow{\sim}_{\text{hyp}} [\mathcal{T}']$. Continue with goal $\Delta \setminus H \vdash \mathcal{T}'$.
   (b) Otherwise, find G' such that $\Delta, \mathcal{T} \overrightarrow{\sim}_{\text{ctx}} [G']$. Continue with goal $\Delta \vdash G'$.

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 § 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 $(\forall x.\, G_1) * (\forall y.\, G_2)$ 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 (§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) (§ 5.2). Besides reverifying existing examples, we use our results to verify logical atomicity of the Michael-Scott queue [Michael and Scott 1996] (§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 (§5.4).

name	impl	total	time	proof	Voila total	Voila proof
bag_stack	30	142	1:13	53	220	74
bounded_counter	20	61	0:32	6	86	19
cas_counter	20	46	0:24	0	98	24
fork_join	14	43	0:21	0	64	17
fork_join_client	13	46	0:20	0	134	35
inc_dec_counter	22	52	0:31	0	111	26
spin_lock	13	56	0:16	0	71	17
ticket_lock	17	74	1:12	4	112	27
ticket_lock_client	7	29	0:39	0	91	17
total	156	549	5:28	63	987	246

Fig. 7. Data on examples with logical atomicity, in comparison with Voila. Rows correspond to files in the supplementary artifact [Mulder and Krebbers 2023]. Columns contain information on lines of *impl*ementation, *total* amount of lines, average verification *time* in minutes:seconds, and lines of *proof* burden, also for Voila.

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

name	impl	total	time	proof	IPM total	IPM proof
rdcss	50	422	6:42	63	689	294
elimination_stack	50	239	4:56	58	375	180
msc_queue	51	427	8:30	168		

Fig. 8. Data on examples with logical atomicity, in comparison with Iris Proof Mode (IPM) proofs.

Voila has, in our count, 1.7 lines of proof burden per line of implementation.[5] 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.

[5] 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].

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

Fig. 9. Statistics on proof automation for ReLoC. Each row contains the name of the verified example, lines of *impl*ementation, *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 $\precsim$ 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.[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 `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 RDCSS 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 compositionality.

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

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. 247–256. https://doi.org/10.1109/LICS.2001.932501

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System *(POPL)*. 109–122. https://doi.org/10.1145/1190216.1190235

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS)*. 364–387. https://doi.org/10.1007/11804192_17

Frédéric Besson. 2021. Itauto: An Extensible Intuitionistic SAT Solver. In *ITP (LIPIcs, Vol. 193)*. 9:1–9:18. https://doi.org/10.4230/LIPIcs.ITP.2021.9

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *PACMPL* 5, ICFP (2021), 81:1–81:29. https://doi.org/10.1145/3473586

James Brotherston and Max Kanovich. 2014. Undecidability of Propositional Separation Logic and Its Neighbours. *J. ACM* 61, 2 (2014), 14:1–14:43. https://doi.org/10.1145/2542667

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: A Complete and Automatic Linearizability Checker *(PLDI)*. 330–340. https://doi.org/10.1145/1806596.1806634

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying Formal Verification to Microkernel IPC at Meta. In *CPP*. 116–129. https://doi.org/10.1145/3497775.3503681

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: A Verified, Concurrent, Crash-Safe Journaling System. In *OSDI*. 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

Łukasz Czajka. 2020. Practical Proof Search for Coq by Type Inhabitation. In *IJCAR (LNCS)*. 28–57. https://doi.org/10.1007/978-3-030-51054-1_3

Pedro da Rocha Pinto. 2016. *Reasoning with Time and Data Abstractions*. Ph. D. Dissertation. Imperial College London. https://doi.org/10.25560/47923

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

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic *(PLDI)*. 792–808. https://doi.org/10.1145/3519939.3523451

David Delahaye. 2000. A Tactic Language for the System Coq. In *LPAR (LNCS)*. 85–95. https://doi.org/10.1007/3-540-44404-1_7

Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48, 2 (2015), 19:1–19:43. https://doi.org/10.1145/2796550

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A Relational Modal Logic for Higher-Order Stateful ADTs *(POPL)*. 185–198. https://doi.org/10.1145/1706299.1706323

Marco Eilers, Severin Meier, and Peter Müller. 2021. Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security. In *CAV (LNCS)*. 718–741. https://doi.org/10.1007/978-3-030-81685-8_34

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *CAV (LNCS)*. 126–133. https://doi.org/10.1007/978-3-319-63390-9_7

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for Concurrent Objects. *TCS* 411, 51 (2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency *(LICS)*. 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021a. Compositional Non-Interference for Fine-Grained Concurrent Programs. In *IEEE Symposium on Security and Privacy (SP)*. 1416–1433. https://doi.org/10.1109/SP40001.2021.00003

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021b. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *LMCS* Volume 17, Issue 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *PACMPL* 6, POPL (2022), 28:1–28:31. https://doi.org/10.1145/3498689

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities. 6, OOPSLA (2022), 74:1–74:30. https://doi.org/10.1145/3527318

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to Make Ad Hoc Proof Automation Less Ad Hoc *(ICFP)*. 163–175. https://doi.org/10.1145/2034773.2034798

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized Logical Relations for Termination-Insensitive Noninterference. *PACMPL* 5, POPL (2021), 10:1–10:29. https://doi.org/10.1145/3434291

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers *(POPL)*. 595–608. https://doi.org/10.1145/2676726.2676975

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *DISC (LNCS)*. 265–279. https://doi.org/10.1007/3-540-36108-1_18

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV (LNCS)*. 449–465. https://doi.org/10.1007/978-3-319-21668-3_26

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR (LNCS)*. 242–256. https://doi.org/10.1007/978-3-642-40184-8_18

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

Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification *(POPL)*. 271–282. https://doi.org/10.1145/1926385.1926417

Ralf Jung. 2019. Logical Atomicity in Iris: The Good, the Bad, and the Ugly. https://people.mpi-sws.org/~jung/iris/logatom-talk-2019.pdf Slides of talk given at Iris Workshop 2019.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State *(ICFP)*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *JFP* 28 (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future Is Ours: Prophecy Variables in Separation Logic. *PACMPL* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning *(POPL)*. 637–650. https://doi.org/10.1145/2676726.2676980

Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *APLAS (LNCS)*. 273–297. https://doi.org/10.1007/978-3-319-71237-6_14

Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *FMCAD*. 143–152. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic *(POPL)*. 205–217. https://doi.org/10.1145/3009837.3009855

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates *(PLDI)*. 181–196. https://doi.org/10.1145/3385412.3386029

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2021. *Automated Verification of Concurrent Search Structures*. Springer. https://doi.org/10.1007/978-3-031-01806-0

K Rustan M Leino and Michał Moskal. 2010. Usable Auto-Active Verification. (2010). https://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf

Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points *(PLDI)*. 459–470. https://doi.org/10.1145/2491956.2462189

William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. https://doi.org/10.48550/arXiv.2207.06574 arXiv:arXiv:2207.06574

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms *(PODC)*. 267–275. https://doi.org/10.1145/248052.248106

Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023. Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. https://ikemulder.nl/media/papers/diaframe-vee-draft.pdf Manuscript.

Ike Mulder and Robbert Krebbers. 2023. Artifact of 'Proof Automation for Linearizability in Separation Logic'. https://doi.org/10.5281/zenodo.7712620 Project webpage: https://gitlab.mpi-sws.org/iris/diaframe.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris *(PLDI)*. 809–824. https://doi.org/10.1145/3519939.3523432

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–255. https://doi.org/10.1109/LICS.2000.855774

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations. In *OOPSLA*, Vol. 3. 161:1–161:30. https://doi.org/10.1145/3360587

Matthew J. Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *ESOP (LNCS)*. 439–458. https://doi.org/10.1007/978-3-642-19718-5_23

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper. In *TACAS (LNCS)*. 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

Andrew M. Pitts. 2005. Typed Operational Reasoning. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 7, 245–289.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types *(PLDI)*. 158–174. https://doi.org/10.1145/3453483.3454036

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-Grained Concurrent Programs *(PLDI)*. 77–87. https://doi.org/10.1145/2737924.2737964

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs (LNCS)*. 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later Credits: Resourceful Reasoning for the Later Modality. ICFP (2022). https://doi.org/10.1145/3547631

Bas Spitters and Eelis Van Der Weegen. 2011. Type Classes for Mathematics in Type Theory. *MSCS* 21, 4 (2011), 795–825. https://doi.org/10.1017/S0960129511000119

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

Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP (LNCS)*. 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

Richard Kent Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency *(ICFP)*. 377–390. https://doi.org/10.1145/2500365.2500600

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

Simon Friis Vindum and Lars Birkedal. 2021. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *CPP*. 76–90. https://doi.org/10.1145/3437992.3439930

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In *CPP*. 100–115. https://doi.org/10.1145/3497775.3503689

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

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

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