Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris

Ike Mulder
Radboud University Nijmegen
The Netherlands
me@ikemulder.nl

Robbert Krebbers
Radboud University Nijmegen
The Netherlands
mail@robbertkrebbers.nl

Herman Geuvers
Radboud University Nijmegen
The Netherlands
herman@cs.ru.nl

Abstract

Fine-grained concurrent programs are difficult to get right, yet play an important role in modern-day computers. We want to prove strong specifications of such programs, with minimal user effort, in a trustworthy way. In this paper, we present **Diaframe**—an *automated* and *foundational* verification tool for fine-grained concurrent programs.

Diaframe is built on top of the Iris framework for higher-order concurrent separation logic in Coq, which already has a foundational soundness proof and the ability to give strong specifications, but lacks automation. Diaframe equips Iris with strong automation using a novel, extendable, goal-directed proof search strategy, using ideas from linear logic programming and bi-abduction. A benchmark of 24 examples from the literature shows that the proof burden of Diaframe is competitive with existing non-foundational tools, while its expressivity and soundness guarantees are stronger.

1 Introduction

Fine-grained concurrent programs, such as locks, reference counters, barriers, and queues, play a critical role in modern day programs and operating systems. Based on 15 years of research on concurrent separation logic [12, 13, 25, 29, 30, 32–35, 46, 65, 66, 72, 78, 79, 83–87], it has become possible to verify increasingly complicated versions of such programs. Yet, while several tools for verification of fine-grained concurrent programs based on these logics exist, none of them are both *automated* (the majority of the proof work is carried out by the tool) and *foundational* (a closed proof w.r.t. the operational semantics is produced in a proof assistant).

Tools with good automation like Caper [31], Starling [88] and Voila [89], generally use SMT [27] or separation-logic solvers [63, 71] as trusted oracles. They are capable of proving programs correct with relatively little help from the user, allowing quick experimentation when designing algorithms. However, they have a large *trusted computing base*—one needs to trust their implementation, the used solvers, the translation of the required side conditions to the used solvers, and sometimes also the soundness of the underpinned logic. In particular, the results of such tools do not come with closed proofs that can be checked independently.

Foundational tools like Iris [43, 44, 46, 50], FCSL [75] and VST [3, 17] are embedded in a proof assistant. Hence, one only needs to trust the implementation of the proof assistant and the operational semantics of the programming language, but not the solvers or underpinned logic. Foundational tools typically provide tactics [2, 6, 17, 49, 51, 58] to hide low-level proofs, but the bulk of the proof work needs to be spelled out. There are two reasons for this status quo. First, foundational tools cannot rely on trusted oracles, unless proofs are reconstructed so that the proof assistant can verify them independently. Second, foundational tools usually have a rich logic that can prove strong specifications, *e.g.,* using impredicative invariants [78], for which automation has received little attention, even in a non-foundational setting.

In this paper, we present **Diaframe**—a foundational tool for automatic verification of fine-grained concurrent programs. Diaframe extends Iris [43, 44, 46, 50]—a framework for interactive proofs in higher-order impredicative concurrent separation logic in Coq—with powerful tactics to perform the bulk of the proof work automatically. This means we get the best of both worlds: closed proofs to underpin our results, while needing relatively little help from the user.

An overview of the architecture of Diaframe is displayed in Figure 1. Diaframe takes two inputs from the user (marked in blue)—a program with a Hoare-style specification, and optionally a set of user-provided hints. The program and specification are turned into an Iris entailment that we prove using an extendable, goal-directed proof search strategy. Inspired by seminal work on linear logic programming [41] and recent work on separation logic programming [74], our strategy interprets logical connectives as proof search instructions. These instructions simplify and solve (a part of) this entailment, possibly generating remaining proofs obligation in the process. To make progress on the remaining obligations, our strategy looks for applicable hints.

Identifying good hints is one of the main challenges that we face. The proof rules of expressive logics like Iris (in particular, those for invariants and ghost state) are not syntax directed and therefore hard to apply automatically. We identify a suitable hint and entailment format that makes it possible to mechanically find and instantiate the appropriate hints. Iris's rules for symbolic execution, reasoning with invariants, and ghost state are translated into syntax-directed variants that match the hint format. An important feature of our entailment and hint format is that it supports a sufficiently large set of Iris's proof rules, while at the same time allowing for an efficient implementation with little backtracking. We

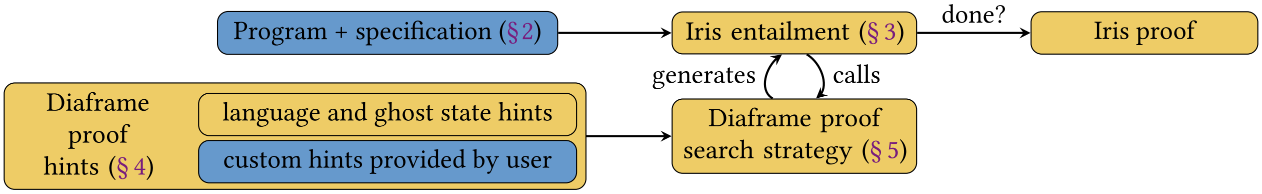

Figure 1. Overview of the architecture of Diaframe. User input is marked in blue.

achieve this by taking inspiration from bi-abduction [15], but adding novel ideas to support Iris's modalities and to postpone instantiation of existentials, which are both needed to support Iris's invariant and ghost state mechanism.

Due to Iris's expressive logic which includes higher-order quantification, impredicative invariants, and the entirety of Coq's logic, our proof strategy is inherently incomplete. Nonetheless, it is able to completely solve many verification goals that appear in Iris proofs in practice. We achieve this by letting our proof strategy (and entailment and hint format) focus on a subset of expressible Iris goals that often appear in formal verification. The proof strategy makes good partial progress on remaining goals where it allows the user to help out with an interactive proof or custom proof hints.

Contributions. We present **Diaframe**—a Coq library for Iris to automate the verification of fine-grained concurrent programs. Concretely, we make the following contributions:

- An entailment (§ 3) and hint format (§ 4) to capture goals and rules in Iris.
- A goal-directed proof search strategy for Iris that can be implemented with little backtracking in Coq (§ 5).
- A benchmark with proofs of correctness of 24 programs using fine-grained concurrency, and a comparison of proof-burden to Starling, Caper, and Voila (§ 6).

We start with two example verifications using Diaframe (§ 2). After covering our contributions (§ 3 to 6), we discuss related work (§ 7), and limitations and future work (§ 8).

2 Diaframe by Example

In this section we showcase Diaframe by verifying a spin lock (§ 2.1) and an Atomic Reference Counter (ARC) (§ 2.2). For both examples we will give Hoare-style specifications $\{P\}\ e\ \{\Phi\}$ in Iris, where $P : iProp$ is a separation logic assertion and $\Phi : Val \rightarrow iProp$ a separation logic predicate on values. The triple $\{P\}\ e\ \{\Phi\}$ means that for each thread that owns resources satisfying P, executing e is safe, and if the execution terminates with value w, the thread will end up owning resources satisfying $\Phi\, w$. The dependency on w allows us to give expected return values in specifications. Note that Iris uses partial, not total correctness. We use the notation SPEC $\{P\}\ e\ \{\vec{y}, \text{RET}\ v;\ Q\}$ for $\{P\}\ e\ \{w.\ \exists \vec{y}.\ \ulcorner v = w \urcorner * Q\}$ to more succinctly specify return values. Note that $\ulcorner \phi \urcorner$ embeds a pure Coq proposition ϕ into separation logic.

```
1  Definition newlock : val :=
2    λ: <>, ref #false.
3  Definition acquire : val :=
4    rec: "acquire" "l" :=
5      if: CAS "l" #false #true then #()
6      else "acquire" "l".
7  Definition release : val :=
8    λ: "l", "l" ← #false.
9  Definition lock_inv γ l R : iProp :=
10   ∃ b : bool, l ↦ #b ∗ (
11      ⌜b = true⌝
12    ∨ ⌜b = false⌝ ∗ locked γ ∗ R).
13 Definition is_lock γ (lk : val) R : iProp :=
14   ∃ l : loc, ⌜lk = #l⌝ ∗ inv N (lock_inv γ l R).
15 Global Program Instance newlock_spec R :
16   SPEC {{ R }}
17     newlock #()
18   {{ (lk : val) γ, RET lk; is_lock γ lk R }}.
19 Global Program Instance acquire_spec γ (lk : val) R:
20   SPEC {{ is_lock γ lk R }}
21     acquire lk
22   {{ RET #(); locked γ ∗ R }}.
23 Global Program Instance release_spec γ (lk : val) R:
24   SPEC {{ is_lock γ lk R ∗ locked γ ∗ R }}
25     release lk
26   {{ RET #(); True }}.
```

Figure 2. Verification of a spinlock in Diaframe.

2.1 Verification of a Spinlock

Lines 1–8 in Figure 2 give the implementation of a spin lock in the ML-like language HeapLang—Iris's default language [43]. The `newlock` method creates a new lock in unlocked state by allocating a new location with value `false`. The `acquire` method uses Compare And Set (`CAS`) to atomically *compare* the stored value of `l` to `false`, *and* only if these are equal, *set* it to `true`. It returns a Boolean to indicate if the equality test was successful. If the `CAS` succeeds, we have acquired the lock. If it fails, we spin by recursively calling the `acquire` method. To release the lock, the `release` method uses a regular store operation to put the lock back to the unlocked state (`false`). We do not need a `CAS`, since only one thread is in charge of releasing the lock.

Let us now consider the specification of the lock methods, given in lines 15–26 in Figure 2. These specifications use the *representation predicates* `is_lock` γ *lk* R and `locked` γ for locks [39, 78]. Here, `is_lock` γ *lk* R expresses that the lock at location *lk* protects assertions R, and `locked` γ expresses that the lock is in locked state. The *ghost identifier* γ is used to tie these two representation predicates together.

Given an arbitrary assertion R, the `newlock` method returns a value *lk*, for which `is_lock` γ *lk* R holds. The assertion `is_lock` γ *lk* R is *duplicable*, meaning it can be shared freely with multiple threads, and thus allows for multiple threads to call `acquire` in parallel. Calling `acquire` on a lock will result in evidence `locked` γ that the lock is locked, and access to assertion R. Contrary to `is_lock` γ *lk* R, the assertion `locked` γ is not duplicable, because at most one thread can hold the lock. To call `release`, we need to relinquish both `locked` γ and R, and get nothing in return.

Specifications of concurrent data structures based on representation predicates [30] allow for easy verification of clients by abstracting away from the implementation. The `is_lock` γ *lk* R representation predicate is particularly flexible, since it is impredicative [78]—meaning that the resources protected by the lock are described by an arbitrary separation logic predicate R that can contain other locks, Hoare triples, *etc.* To define impredicative representation predicates, we use Iris's invariant and ghost state mechanism.

Programs using fine-grained concurrency have multiple threads reading and mutating shared state. In the example, the location backing the spinlock needs to be shared so that multiple threads can attempt to `acquire` the lock in parallel. Since the *points-to assertion* $\ell \mapsto v$ of separation logic expresses exclusive ownership of the location ℓ with value v, we cannot just share it between multiple threads.

To reason about shared mutable state, we use Iris's *invariant assertion* $\boxed{L}^{\mathcal{N}}$, which says that there is a (shared) invariant with name $\mathcal{N}$ governing the resources satisfying Iris assertion L. Invariants $\boxed{L}^{\mathcal{N}}$ are duplicable, which means that the assertion L inside the invariant is accessible by all threads. To do this soundly, access to L is restricted. Only during atomic operations (like an assignment or `CAS`), invariants may be 'opened', which gives one temporary access to the assertion L in the verification of a thread. After the atomic operation, the invariant must be 'closed', meaning one must show the assertion L still holds.

Lines 9–14 contain the definition of `is_lock` γ *lk* R. It says that a value *lk* is a lock if it is equal to some location `l`, whose stored value is governed by an invariant `lock_inv`. Note that in Coq, we write `inv N L` for $\boxed{L}^{\mathcal{N}}$. The invariant `lock_inv` states that `l` should point to a Boolean. If this Boolean is `true`, the lock is locked, and we know nothing else since the resources satisfying R are currently owned by a thread which acquired the lock. If this Boolean is `false`, the lock is unlocked, and the resources satisfying R as well as the `locked` γ assertion are owned by the invariant.

The key ingredient for the verification of the spinlock is the *ghost assertion* `locked` γ, whose rules are:[1]

LOCKED-ALLOCATE	LOCKED-UNIQUE
$\vdash\ \dot{\Rrightarrow}\,\exists \gamma.\ \texttt{locked}\ \gamma$	$\texttt{locked}\ \gamma * \texttt{locked}\ \gamma \vdash \texttt{False}$

The first rule is used in the proof of `newlock`. It allows for the allocation of `locked` γ with a fresh ghost name γ. This assertion is needed to establish the invariant by proving the right disjunct of `lock_inv`. (The *update modality* $\dot{\Rrightarrow}$ signifies a logical update to the ghost state. It will be explained in §3.2, but for now, it is enough to know that after each program statement, we can perform a logical update in the proof.)

The second rule states that `locked` γ is a singleton—no two threads/resources can simultaneously satisfy this assertion. This means that the `locked` γ assertion gives us information about the global state. In the proof of `release`, just before executing the store, the right disjunct of `lock_inv` is contradictory because `locked` γ is in the precondition. Hence, the left disjunct must hold—the location `l` must point to the value true, *i.e.,* the lock is in locked state.

The general structure of verification in Diaframe is similar for other examples: we give the implementation and specification, and an invariant using appropriate ghost assertions, after which the verification will go mostly automatically. Other concurrent programs may use different ghost assertions, but all of these assertions have three types of rules: (a) allocation/creation rules, like LOCKED-ALLOCATE, (b) compatibility/interaction rules, like LOCKED-UNIQUE, and (c) mutation/update rules, of the form $P * Q \vdash\ \dot{\Rrightarrow} R * S$. We will see some update rules in the next example.

2.2 Verification of an ARC

We will now verify a version of an Atomic Reference Counter (ARC), similar to the one verified by Starling [88] and the one used in the Rust standard library [52]. An ARC can be used to safely give multiple threads read-access to a resource, while being able to recover write-access once all read-access references have been dropped. Lines 2–13 in Figure 3 give the implementation. Values of ARC are locations that store an integer containing the number of read-access references. The `mk_arc` method allocates a location with value 1, *i.e.,* an ARC with one read-access reference. The `count` method gives the number of read-access references. The `clone` method increments the reference count with 1, using the atomic Fetch And Add (`FAA`) instruction, while `drop` decrements the reference count with 1. The `unwrap` method is like `drop` in that it will decrement the reference count—but by using a `CAS` operation to set the reference count from 1 to 0, it ensures that it destroys the last reference, and spins as long as other references have not been dropped.

[1] For readers familiar with Iris, we simply define `locked` $\gamma \triangleq \boxed{\text{Excl}\ ()}^{\gamma}$.

```
1  Context (P : Qp → iProp) {HP : Fractional P}.
2  Definition mk_arc : val :=
3    λ: <>, ref #1.
4  Definition clone : val :=
5    λ: "a", FAA "a" #1 ;; #().
6  Definition count : val :=
7    λ: "a", ! "a".
8  Definition drop : val :=
9    λ: "a", (FAA "a" #-1) = #1.
10 Definition unwrap : val :=
11   rec: "unwrap" "a" :=
12     if: CAS "a" #1 #0 then #()
13     else "unwrap" "a".
14 Definition arc_inv γ l : iProp :=
15   ∃ (z : Z), l ↦ #z ∗ (
16       ⌜0 < z⌝%Z ∗ counter P γ (Z.to_pos z)
17     ∨ ⌜z = 0⌝ ∗ no_tokens P γ).
18 Definition is_arc γ (v : val) : iProp :=
19   ∃ (l : loc), ⌜v = #l⌝ ∗ inv N (arc_inv γ l).
20 Global Program Instance mk_arc_spec :
21   SPEC {{ P 1 }}
22     mk_arc #()
23   {{ (v : val) γ, RET v; is_arc γ v ∗ token P γ }}.
24 Global Program Instance clone_spec γ (v : val) :
25   SPEC {{ is_arc γ v ∗ token P γ }}
26     clone v
27   {{ RET #(); token P γ ∗ token P γ }}.
28 Global Program Instance count_spec γ (v : val) :
29   SPEC {{ is_arc γ v ∗ token P γ }}
30     count v
31   {{ (p : Z), RET #p; ⌜0 < p⌝%Z ∗ token P γ }}.
32 Global Program Instance drop_spec γ (v : val) :
33   SPEC {{ is_arc γ v ∗ token P γ }}
34     drop v
35   {{ (b : bool), RET #b; ⌜b = false⌝ ∨
36     ⌜b = true⌝ ∗ P 1 ∗ no_tokens P γ }}.
37 Next Obligation.
38   destruct (decide (x3 = 1)); iStepsS.
39 Qed.
40 Global Program Instance unwrap_spec γ (l : val) :
41   SPEC {{ is_arc γ l ∗ token P γ }}
42     unwrap l
43   {{ RET #(); P 1 ∗ no_tokens P γ }}.
```

Figure 3. Verification of an ARC in Diaframe.

To give a specification of the methods of ARC, we make use of shareable assertions, which are typically modeled with fractional permissions [11]. In Iris, shareable assertions are modeled as Iris predicates $P : \mathbb{Q}_p \to iProp$, where *iProp* is the type of Iris assertions, and $\mathbb{Q}_p \triangleq \{q \in \mathbb{Q} \mid q > 0\}$. Predicates P of this type must satisfy $P\,q_1 * P\,q_2 \dashv\vdash P\,(q_1 + q_2)$ to be called shareable (or `Fractional` in Coq). An example of a shareable assertion is the fractional mapsto connective $\ell \mapsto_q v$. If $q = 1$, it denotes full ownership of (or write-access to) heap-location ℓ. If $0 < q < 1$, it denotes fractional ownership of (or read-access to) heap-location ℓ.

TOKEN-ALLOCATE

$$P\,1 \vdash \dot{\Rrightarrow} \exists \gamma.\, \mathsf{counter}\ P\ \gamma\ 1 * \mathsf{token}\ P\ \gamma$$

TOKEN-INTERACT

$$\mathsf{no\_tokens}\ P\ \gamma * \mathsf{token}\ P\ \gamma \vdash \mathsf{False}$$

TOKEN-MUTATE-INCR

$$\mathsf{counter}\ P\ \gamma\ p \vdash \dot{\Rrightarrow} (\mathsf{counter}\ P\ \gamma\ (p+1) * \mathsf{token}\ P\ \gamma)$$

TOKEN-MUTATE-DECR

$$\frac{p > 1}{\mathsf{counter}\ P\ \gamma\ p * \mathsf{token}\ P\ \gamma \vdash \dot{\Rrightarrow} \mathsf{counter}\ P\ \gamma\ (p-1)}$$

TOKEN-MUTATE-DELETE-LAST

$$\mathsf{counter}\ P\ \gamma\ 1 * \mathsf{token}\ P\ \gamma \vdash \dot{\Rrightarrow} (\mathsf{no\_tokens}\ P\ \gamma * \mathsf{no\_tokens}\ P\ \gamma * P\,1)$$

TOKEN-ACCESS

$$\mathsf{token}\ P\ \gamma \vdash \exists q.\, P\,q * (P\,q \mathrel{-\!\!*} \mathsf{token}\ P\ \gamma)$$

Figure 4. Rules for the counter ghost assertions.

As shown on line 1 in Figure 3, the whole verification is abstracted over a shareable assertion P that describes the resources that are being protected by the ARC. The specification of the methods can be found in lines 20–43. Like for the spinlock, we use several representation predicates. The duplicable assertion `is_arc` $\gamma\ v$ says that a value v is an ARC. The non-duplicable assertion `token` $P\ \gamma$ indicates a read-access reference to P. The non-duplicable assertion `no_tokens` $P\ \gamma$ indicates that write-access has been recovered, *i.e.,* that no read-access tokens `token` $P\ \gamma$ exist.

With these predicates at hand, the specification of `mk_arc` requires $P\,1$ (write-access) and returns a value that `is_arc` guarding P, along with a single read-access `token`. The `count` method is essentially a no-op, but shows that if we have a single-read access token, the reference count must be positive. The method `clone` duplicates a read-access `token`—it requires one of them, and returns two. The method `drop` destroys a `token`, and either returns nothing, or, if this was the last token, write-access $P\,1$, along with the knowledge that `no_tokens` exist. The `unwrap` method, when it terminates, guarantees retrieving write-access $P\,1$ and `no_tokens`.

Let us look at the definition of `is_arc` in lines 14–19 in Figure 3. Similar to `locked` γ, we treat `token` and `no_tokens` abstractly (see our appendix [62] for the definition), and show only the allocation, interaction and update rules in Figure 4. As witnessed by TOKEN-ACCESS, these ghost-state assertions are used to convert fractional permissions into counting permissions [9], which are more natural for ARC.

Similar to the spinlock, we define a value to be `is_arc` if it is a location whose stored value is governed by an invariant. This invariant `arc_inv` tells us that the location points to some integer z, which satisfies: (1) $z = 0$, and we know that no tokens currently exist, or (2) $z > 0$, and we own resources satisfying `counter` $P\ \gamma\ z$. The `counter` $P\ \gamma\ p$ assertion states the knowledge that precisely $p > 0$ tokens currently exist—which matches what we want $\ell \mapsto p$ to mean.

To prove the specification of the `count` method, we use TOKEN-ALLOCATE, which allows us to establish the left disjunct of `arc_inv`. For proving the specification of `count`, we rely on TOKEN-INTERACT to prove that the right disjunct of `arc_inv` is contradictory. For the specification of `clone`, we again need TOKEN-INTERACT. When closing the invariant, we need to apply TOKEN-MUTATE-INCR at the right moment to change the obtained `counter` $P\ \gamma\ p$ to the required `counter` $P\ \gamma\ (p + 1)$. This also gives us the extra `token` that we need in the postcondition.

When proving `drop`, we need to use TOKEN-MUTATE-DECR if our token was not the last one. Otherwise, we need to use TOKEN-MUTATE-DELETE-LAST. Diaframe is not smart enough to figure out the required case distinction automatically. However, since Diaframe only performs limited backtracking, the automation simply stops when it encounters a goal it cannot make progress on. After manually performing the required case analysis, we resume the Diaframe automation with the `iStepsS` tactic, which then solves the goal.

The ghost assertions we use here (`token`, `no_token` and `counter`), are not connected to a memory location and are thus not specific for the verification of ARC. We also use them in the verification of *e.g.,* reader-writer locks. The only connection between these assertions and the ARC lies in the definition of the invariant `arc_inv`, which ties the physical state of the ARC to an appropriate ghost-state. The rules for the assertions in Figure 4 are available to the Diaframe proof search strategy, and applying them requires no extra annotations, except for the manual case distinction for `drop`.

3 Diaframe's Entailment Format

In this section we explain some of the challenges one faces when automating proofs of fine-grained concurrent programs in Iris. We start with some background on verifying weakest preconditions of sequential programs using symbolic execution (§3.1), as commonly done in interactive and automatic tools in proof assistants [20, 51, 74]. We then extend this approach with support for Iris's invariant mechanism to verify fine-grained concurrent programs (§3.2). We conclude with an overview of the Diaframe entailment format and proof strategy (§3.3), which serves as a starting point for the description of our hint format (§4).

WP-VALUE
$$\frac{}{\Phi\, v \vdash \mathsf{wp}\ v\ \{\Phi\}}$$

WP-BIND
$$\frac{}{\mathsf{wp}\ e\ \big\{w.\ \mathsf{wp}\ K[w]\ \{\Phi\}\big\} \vdash \mathsf{wp}\ K[e]\ \{\Phi\}}$$

WP-FRAME
$$\frac{}{R * \mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{v.\ R * \Phi\, v\}}$$

WP-MONO
$$\frac{\forall v.\ \Psi\, v \vdash \Phi\, v}{\mathsf{wp}\ e\ \{\Psi\} \vdash \mathsf{wp}\ e\ \{\Phi\}}$$

WP-FAA
$$\frac{}{\ell \mapsto z_1 \vdash \mathsf{wp}\ (\mathsf{FAA}\ \ell\ z_2)\ \{w.\ \ulcorner w = z_1 \urcorner * \ell \mapsto (z_1 + z_2)\}}$$

Figure 5. Some of Iris's rules for weakest preconditions.

3.1 Goal-Directed Reasoning with WP

Hoare triples are not a primitive of Iris, they are defined in terms of *weakest preconditions*:

$$\{P\}\ e\ \{\Phi\} \triangleq P \vdash \mathsf{wp}\ e\ \{\Phi\}.$$

To get some intuition for the semantics of $\mathsf{wp}\ e\ \{\Phi\}$, assume for a moment that P and Q are predicates on heaps (ignoring Iris's ghost state and step-indexing), and Φ is a predicate on values and heaps. Entailment $P \vdash Q$ means that for every heap h, if $P\, h$ holds, then $Q\, h$ holds. The assertion $\mathsf{wp}\ e\ \{\Phi\}$ describes the heaps for which execution of e is safe (cannot get stuck), and if e terminates with value v and heap h', then $\Phi\, v\, h'$ holds. Defining $\{P\}\ e\ \{\Phi\}$ as above then indeed gives the Hoare triple its intended and intuitive semantics.

Weakest preconditions make it possible to decouple the precondition from the Hoare triple, and view it as a regular separation logic entailment. In particular, they give us access to Iris's existing infrastructure [49, 51] for proving entailments. However, Iris's primitive rules for weakest preconditions in Figure 5 are not syntax directed and can thus not be directly applied in an interactive or automatic proof search strategy. Throughout this section, we focus on transforming the rule WP-FAA into a syntax-directed version. Recall that `FAA` is used in the `clone` and `drop` methods of ARC (§2.2).

Suppose we are proving the following entailment:

$$\Delta \vdash \mathsf{wp}\ (\mathsf{FAA}\ \ell\ v_2\ v_3)\ \{\Phi\}.$$

(From now on, we will often put an environment Δ before the turnstile. The environment Δ is a list of assertions $P_1, \ldots, P_n$, for which $\Delta \vdash Q$ iff $P_1 * \cdots * P_n \vdash Q$.)

We want to prove this entailment by applying WP-FAA, but we are not yet in shape to do so. That is because Δ will typically not be just $\ell \mapsto z_1$, and Φ will typically not be the precise postcondition of WP-FAA. Hence, to apply 'small footprint' specifications like WP-FAA we need to find a 'frame' R and a value z_1, such that $\Delta \vdash R * \ell \mapsto z_1$. We can then use a combination of WP-FAA, WP-FRAME and WP-MONO, to transform our entailment into $R * \ell \mapsto (z_1 + z_2) \vdash \Phi\, z_1$.

Instead of having to determine the frame R in advance, one can construct an alternative rule for goal-directed reasoning, which will be easier to apply automatically:

$$\textsc{wp-faa-ramify}\quad \frac{\Delta \vdash l \mapsto ?z_1 * \left(\forall v.\ (\ulcorner v = ?z_1 \urcorner * \ell \mapsto (?z_1 + z_2)) \mathrel{-\!\!*} \Phi\, v\right)}{\Delta \vdash \mathsf{wp}\ (\mathsf{FAA}\ \ell\ z_2)\ \{\Phi\}}$$

In this shape, the rule is an instance of the *ramified frame rule* [20, 40]. Note that we have put a question mark in front of z_1 to signify that z_1 will be an existential variable (evar) at rule application—we should be able to find a z_1 for which this is provable, but do not yet know which one it will be. When we find an hypothesis of shape $\ell \mapsto z$ in Δ, we can *unify* z_1 with z and continue.

We can generalize the rule wp-faa-ramify to any Hoare-style specification of an expression e:

$$\textsc{sym-ex}\quad \frac{\{P\}\ e\ \{\Psi\} \qquad \Delta \vdash P * (\forall v.\ \Psi\, v \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ \{\Phi\})}{\Delta \vdash \mathsf{wp}\ K[e]\ \{\Phi\}}$$

This rule additionally incorporates Iris's rule wp-bind, which allows the expression e to appear inside a call-by-value evaluation context K, instead of at the top-level.

Supposing we can prove separating conjunctions, sym-ex gives rise to a symbolic-execution based proof search strategy for straight-line sequential code. Suppose our goal is $\Delta \vdash \mathsf{wp}\ e\ \{\Phi\}$. If e is a value v, apply wp-value and prove $\Delta \vdash \Phi\, v$. Else, find an evaluation context K and subexpression e' with $e = K[e']$, and a specification $\{P\}\ e'\ \{\Psi\}$. Apply sym-ex, prove the separating conjunction, introduce variables, introduce the left-side of the magic wand, and repeat.

3.2 Goal-Directed Reasoning with Invariants

We will now extend the naive proof search strategy from §3.1 with support for Iris's invariant mechanism to handle programs with fine-grained concurrency. Concretely, we will present a rule that extends sym-ex, which can also be used in case the precondition P is inside an invariant (as is the case for all examples in §2). We will first recapitulate Iris's original proof rule for accessing invariants:

$$\textsc{inv-open-wp}\quad \frac{\Delta, \boxed{L}^{\mathcal{N}}, \triangleright L \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\ e\ \{v.\ \triangleright L * \Phi\, v\} \quad \mathsf{atomic}\ e \quad \mathcal{N} \subseteq \mathcal{E}}{\Delta, \boxed{L}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}}$$

This rule is quite a mouthful, so let us go over it step by step. First, to deal with invariants, weakest preconditions in Iris $\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}$ have a *mask* annotation $\mathcal{E}$, signifying the set of names of invariants that can be opened. This is necessary to ensure invariants are not opened more than once (*i.e.,* to avoid reentrancy, which is unsound). Omitted masks are $\top$, meaning all invariants can still be opened.

Suppose that we have an invariant $\boxed{L}^{\mathcal{N}}$, and are verifying an atomic expression e. Rule inv-open-wp states that we are allowed to look inside the invariant and obtain L in the proof context, but then must show that L still holds in the postcondition of the WP. After we have opened the invariant with name $\mathcal{N}$, the mask changes to $\mathcal{E} \setminus \mathcal{N}$ so that we cannot open the invariant twice. The *later modality* ($\triangleright$) [4, 64] is needed for technical reasons caused by the fact that invariants are *impredicative* [44, 78], *i.e.,* the resource L in an invariant can be *any* resource, including invariants and weakest preconditions. Dealing with later modalities requires some additional bookkeeping, which Diaframe deals with automatically, but we gloss over in this paper.

We now show why our sym-ex rule for symbolic execution from §3.1 needs to be extended for programs involving fine-grained concurrency. Consider the FAA operation in the clone method of the ARC (§2.2). The challenge of verifying this method is that the $\ell \mapsto \_$ we need as part of the precondition for FAA is not in the proof context, but in an invariant $\boxed{\exists (z : \mathbb{Z}).\ \ell \mapsto z * J\, z}^{\mathcal{N}}$. When we apply sym-ex eagerly, we lose the ability to open invariants using inv-open-wp.

One approach is to try to make progress with sym-ex—if this is possible, we are alright. If not, we backtrack, and open an invariant with inv-open-wp, and retry. This is similar to the approach employed by Caper [31]. We do not take a backtracking approach in Diaframe since it does not mix nicely with interactive proofs.

We therefore present an extended symbolic execution rule, sym-ex, which allows us to open invariants lazily:

$$\textsc{sym-ex-fupd-exist}\quad \frac{\forall \vec{x}.\ \{P\}\ e\ \{\Psi\} \qquad \mathsf{atomic}\ e \vee \mathcal{E}_1 =?\mathcal{E}_2 \qquad \Delta \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{?\mathcal{E}_2}\, \exists \vec{x}.\ P * \left(\forall w.\ \Psi\, w \mathrel{-\!\!*} {}^{?\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_1}\, \mathsf{wp}_{\mathcal{E}_1}\ K[w]\ \{\Phi\}\right)}{\Delta \vdash \mathsf{wp}_{\mathcal{E}_1}\ K[e]\ \{\Phi\}}$$

This rule contains Iris's *fancy update modality* ${}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}$, and a *quantified Hoare triple* $\forall \vec{x}.\ \{P\}\ e\ \{\Psi\}$.

The fancy update modality ${}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}$ is used in Iris's definition of weakest preconditions, and is the component that makes opening invariants possible. Semantically, ${}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P$ means: assuming all invariants with names in $\mathcal{E}_1$ hold, then P holds and additionally all invariants with names in $\mathcal{E}_2$ hold. To work with the fancy update, Iris has the following rules:

$$\textsc{inv-open-fupd}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{L}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright L * \left(\triangleright L \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\mid\!\Rrightarrow}^{\mathcal{E}}\, \mathsf{True}\right)\right)} \qquad \textsc{bupd-intro}\quad \frac{}{P \vdash \dot{\mid\!\Rrightarrow} P}$$

$$\textsc{bupd-fupd}\quad \frac{}{\dot{\mid\!\Rrightarrow} P \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} P} \qquad \textsc{fupd-elim}\quad \frac{P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} Q \qquad \Delta, Q \vdash {}^{\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_3} R}{\Delta, P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_3} R}$$

The inv-open-fupd rule makes the semantics of invariants precise: by removing $\mathcal{N}$ from the mask, we get access to L, and if we wish to restore the mask, we must hand back L via the *closing update* ($\triangleright L \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\mid\!\Rrightarrow}^{\mathcal{E}}\, \mathsf{True}$). The rule fupd-elim allows us to compose fancy updates, and by combining bupd-fupd and bupd-intro we can introduce the last fancy update when done. Note that BUPD-FUPD and FUPD-ELIM enable us to perform logical updates (like those in Figure 4) when the goal contains a fancy update after the turnstile.

The quantified Hoare triple $\forall \vec{x}.\, \{P\}\; e\; \{\Psi\}$ states that the Hoare triple $\{P\}\; e\; \{\Psi\}$ holds for all instantiations of the auxiliary variables in $\vec{x}$. Here, P should and Ψ may refer to the variables in $\vec{x}$. For FAA, we have:

$$\forall z_1.\, \{\ell \mapsto z_1\}\; \mathsf{FAA}\, \ell\, z_2\; \{w.\, \ulcorner w = z_1 \urcorner * \ell \mapsto (z_1 + z_2)\}.$$

The essential feature of SYM-EX-FUPD-EXIST is that once we apply the rule, we retain the ability to open (any number of) invariants through a combination of the rules FUPD-ELIM and INV-OPEN-FUPD. Our new rule is strictly stronger than the rule SYM-EX from §3.1—the update modalities can simply be introduced using BUPD-FUPD and BUPD-INTRO, and the existentials can be instantiated with evars.

We now show why the existential quantification in the new rule is necessary. Let us try to use SYM-EX-FUPD-EXIST *wrongly by instantiating existentials eagerly* in a goal that arises during the verification of an FAA in ARC (§2.2):

$$\dfrac{\dfrac{\dfrac{\dfrac{\ell \mapsto z, \triangleright J\, z, \ldots \vdash {}^{\top\setminus\mathcal{N}}{\mid\!\!\!\Rrightarrow}^{?\mathcal{E}}\, \ell \mapsto ?z_1 * \ldots}{\triangleright(\exists (z : \mathbb{Z}).\, \ell \mapsto z * J\, z), \ldots \vdash {}^{\top\setminus\mathcal{N}}{\mid\!\!\!\Rrightarrow}^{?\mathcal{E}}\, \ell \mapsto ?z_1 * \ldots}}{\boxed{\exists (z : \mathbb{Z}).\, \ell \mapsto z * J\, z}^{\mathcal{N}} \vdash {}^{\top}{\mid\!\!\!\Rrightarrow}^{?\mathcal{E}}\, \ell \mapsto ?z_1 * \ldots}}{\boxed{\exists (z : \mathbb{Z}).\, \ell \mapsto z * J\, z}^{\mathcal{N}} \vdash {}^{\top}{\mid\!\!\!\Rrightarrow}^{?\mathcal{E}}\, \exists z'.\, \ell \mapsto z' * \ldots}}{\boxed{\exists (z : \mathbb{Z}).\, \ell \mapsto z * J\, z}^{\mathcal{N}} \vdash \mathsf{wp}\; K[\mathsf{FAA}\; \ell\; 1]\; \{\Phi\}}$$

One should read this proof derivation from bottom to top. When encountering an FAA, we apply SYM-EX-FUPD-EXIST, but (wrongly) perform an eager instantiation of the existential z' with an evar $?z_1$. Then we use INV-OPEN-FUPD and FUPD-ELIM to open the invariant. The final step uses some properties of the later modality to eliminate the existential and the later around $\ell \mapsto z$. One might think we are now done: just unify $?z_1$ with z and $?\mathcal{E}$ with $\top \setminus \mathcal{N}$, and continue! However, this is not sound—the evar $?z_1$ cannot be unified with z, since z was introduced after z_1. Stated in other words, we could not have chosen z_1 to be equal to z, since at that point z was not in our context. To correctly deal with existentials, the Diaframe proof search strategy delays the instantiation of existentials.

3.3 Overview of the Diaframe Strategy

To automatically prove program specifications $\forall \vec{x}.\, \{P\}\; e\; \{\Phi\}$, Diaframe's proof strategy repeatedly performs the following actions (a formal presentation is given in §5):

1. If the goal is $\Delta \vdash \forall x.\, G$ or $\Delta \vdash U \twoheadrightarrow G$, introduce the $\forall$ or $\twoheadrightarrow$. Then "clean" the hypothesis U by (a) eliminating separating conjunctions, disjunctions, and existentials, (b) moving pure assertions $\ulcorner \phi \urcorner$ into the Coq context, (c) merging assertions, *e.g.*, $\ell \mapsto_q w$ and $\ell \mapsto_p v$ become $\ell \mapsto_{p+q} v$ and $v = w$, (d) deriving contradictions, *e.g.*, using LOCKED-UNIQUE.
2. If the goal is $\Delta \vdash \mathsf{wp}\; v\; \{\Phi\}$, with v a value, continue with $\Delta \vdash {}^{\top}{\mid\!\!\!\Rrightarrow}^{\top}\, \Phi\, v$.
3. If the goal is $\Delta \vdash \mathsf{wp}\; K[e]\; \{\Phi\}$, use our new symbolic execution rule SYM-EX-FUPD-EXIST. Our new goal has the shape $\Delta \vdash {}^{\mathcal{E}_1}{\mid\!\!\!\Rrightarrow}^{\mathcal{E}_2}\, \exists \vec{x}.\, L * G$.
4. If the goal is $\Delta \vdash {}^{\mathcal{E}_1}{\mid\!\!\!\Rrightarrow}^{\mathcal{E}_2}\, \exists \vec{x}.\, L * G$, use associativity of separating conjunction to rewrite it into ${}^{\mathcal{E}_1}{\mid\!\!\!\Rrightarrow}^{\mathcal{E}_2}\, \exists \vec{x}.\, A * G'$ where A is an atom. Pure conditions $\ulcorner \phi \urcorner$ that are appear in the process are solved with Coq tactics like `lia`. We make progress on A by finding a hint.

For this strategy to be effective, finding hints (in the last step) is crucial. These hints need to make sure that the resulting goal is again of one of the above entailment formats so the strategy can make repeated progress. When operating on entailments of format $\Delta \vdash {}^{\mathcal{E}_1}{\mid\!\!\!\Rrightarrow}^{\mathcal{E}_2}\, \exists \vec{x}.\, L * G$, it is essential that modalities and existentials are only introduced/instantiated when the right invariants have been opened and the necessary ghost updates have been performed—not earlier.

The Diaframe proof strategy is inspired by the idea of interpreting logical connectives as instructions to control the proof search, as done in the seminal work on linear logic programming [19, 41] and recent work on the separation logic programming language Lithium [74]. The fundamental difference is that we do not operate on top-level connectives, but on those that appear below a modality and a number of existentials, to support Iris's invariants and ghost state.

4 Diaframe's Hint Format

In this section, we describe the process of finding hints. We consider the following kinds of base hints: (a) hints for ghost state such as those corresponding to the rules in Figure 4, (b) hints for language-specific connectives such as the $\mapsto$ connective, and (c) user-defined hints to guide the proof of a specific program in case the automation falls short.

There are two ways how hints can be selected. *Goal-and-hypothesis directed hints* use the shape of the goal and the shape of a hypothesis as keys. Examples are hints for mutating ghost state. *Last-resort goal-directed hints* are used if no hints that key on a hypothesis can be found. Examples are invariant allocation and ghost state allocation.

Hints are specified using a *hint format* (§4.1) that is inspired by the technique of bi-abduction [15]. Aside from the base hints (§4.2), Diaframe provides *recursive hints* to close the base hints under connectives like invariants, magic wands, and separating conjunctions (§4.3).

4.1 Bi-Abduction Hints

The *hint format* of Diaframe is as follows:

$$H * [\vec{y}; L] \Vdash \left[{}^{\mathcal{E}_1}{\mid\!\!\!\Rrightarrow}^{\mathcal{E}_2}\right]; \vec{x}; A * [U] \triangleq \\ \forall \vec{y}.\, \big(H * L \vdash {}^{\mathcal{E}_1}{\mid\!\!\!\Rrightarrow}^{\mathcal{E}_2}\, (\exists \vec{x}.\, A * U)\big)$$

Hints use a hypothesis H and goal A as key/input. Outputs are denoted between [ ] syntax: L is a (possibly existentially quantified) side condition, while U is the residue we obtain after using the hint. The hypothesis H is ε_1 for a last-resort hint. The assertion ε_1 is an opaque marker whose semantics is True, but is treated differently by the proof search strategy.

It is instructive to check the scope of the existentials. The premise H is a given hypothesis, so $\vec{x}$ and $\vec{y}$ do not occur in H. The conclusion A is a given existential goal, so $\vec{x}$ occurs in A, but $\vec{y}$ does not. The side condition L is existentially quantified with $\vec{y}$. The residue U is allowed to contain both $\vec{x}$ and $\vec{y}$ so it can be related to the side condition L and the goal A.

We also call Diaframe's hints "bi-abduction hints" because in essence, they are bi-abduction [15] behind a modality and existentials. The bi-abduction problem in separation logic asks to find, given an hypothesis H and goal A, a 'frame' and 'antiframe' such that $H * ?antiframe \vdash A * ?frame$. Our hints' shape is also similar to the residuation judgment from Cervesato et al. [19], but has an additional frame.

We can apply a Diaframe bi-abduction hint as follows:

$$\textsc{biabd-hint-apply}\quad \frac{H * [\vec{y}; L] \Vdash \left[{}^{\mathcal{E}_3}\!\Rrightarrow^{\mathcal{E}_2}\right]; \vec{x}; A * [U] \qquad \Delta \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} \exists \vec{y}.\, L * (\forall \vec{x}.\, U \mathrel{-\!\!*} G)}{\Delta, H \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, A * G}$$

The Diaframe implementation will go over the hypotheses H in the context Δ from left to right (with ε_1 last) until it finds a hint $H * [\vec{y}; L] \Vdash \left[{}^{\mathcal{E}_3}\!\Rrightarrow^{\mathcal{E}_2}\right]; \vec{x}; A * [U]$ in the hint database. This involves some backtracking, but only *locally*—whenever a hint (and thus a side condition L and residue U) has been found for a hypothesis H, we use that hint and will never backtrack to consider a different choice. Note that after applying the rule, the resulting entailment has the same format, allowing for repeated applications of hints.

4.2 Base Hints

Example 1: Ghost state mutation. We transform the rule TOKEN-MUTATE-DECR (which is used to verify the `drop` method of ARC in §2.2) into the following hint:

$$\texttt{counter}\ P\ \gamma\ p * [\_; \texttt{token}\ P\ \gamma * \ulcorner p > 1 \urcorner] \Vdash \left[{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\right]; \_; \texttt{counter}\ P\ \gamma\ (p-1) * [\mathsf{True}]$$

If we use BIABD-HINT-APPLY with this hint, we get:

$$\frac{\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \texttt{token}\ P\ \gamma * \ulcorner p > 1 \urcorner * (\mathsf{True} \mathrel{-\!\!*} G)}{\Delta, \texttt{counter}\ P\ \gamma\ p \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \texttt{counter}\ P\ \gamma\ (p-1) * G}$$

Here we see that to decrement the counter, we need to solve the side condition $\texttt{token}\ P\ \gamma$, before we can continue with G.

Example 2: Invariant allocation. In Iris, invariants are allocated using the rule $\triangleright L \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \boxed{L}^{\mathcal{N}}$, which we transform into the following hint:

$$\varepsilon_1 * [\_; \triangleright L] \Vdash \left[{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\right]; \_; \boxed{L}^{\mathcal{N}} * \left[\boxed{L}^{\mathcal{N}}\right].$$

Due to the ε_1, this is a last-resort goal-directed hint. We do not make it hypothesis directed, because $\triangleright L$ will usually not be precisely in the context. Since invariants are duplicable we give back $\boxed{L}^{\mathcal{N}}$ in the residue, so that it can be used again.

Example 3: Ghost state allocation. We transform the rule LOCKED-ALLOCATE (which is used to verify the `newlock` method in §2.1) into the following hint:

$$\varepsilon_1 * [\_; \mathsf{True}] \Vdash \left[{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\right]; \gamma; \texttt{locked}\ \gamma * [\mathsf{True}]\,.$$

Due to the ε_1, this is again a last-resort goal-directed hint. That is simply because the rule has no premise.

Example 4: Point-to assertion. We have specific hints for HeapLang's fractional points-to assertion $\ell \mapsto_q v$:

$$\ell \mapsto_q v_1 * [\_; \ulcorner v_1 = v_2 \urcorner] \Vdash \left[{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\right]; \_; \ell \mapsto_q v_2 * [\mathsf{True}].$$

This hint says that if we have a points-to for ℓ, but need one with another value, we should prove that both values are equal. The following hint handles different fractions:

$$\frac{q_1 < q_2}{\ell \mapsto_{q_1} v_1 * \left[v_3; \ulcorner v_1 = v_2 \urcorner * \ell \mapsto_{(q_2-q_1)} v_3\right] \Vdash \left[{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\right]; \_; \ell \mapsto_{q_2} v_2 * [\ulcorner v_1 = v_3 \urcorner]}$$

This hint applies if the fraction q_2 in the goal is bigger than the fraction q_1 in the hypothesis, and hence has the side condition $\ell \mapsto_{q_2-q_1} v_3$. Note that v_3 is existentially quantified, meaning that the side condition can be established for any value. This is sound by the agreement property of $\mapsto$. This generality is used in the verification of *e.g.*, the CLH-lock. There is a dual hint for the case $q_1 > q_2$.

4.3 Recursive Hints

It is often the case that a base hint almost—but not precisely—matches. The premise might appear under a magic wand or in an invariant, or the goal might provide a specific witness while looking for an existential. Diaframe therefore includes a number of recursive hints to close the base hints under the connectives of higher-order separation logic. For example:

$$\frac{U * [\vec{z}; L_2] \Vdash \left[{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\right]; \vec{y}; A * [U]}{(L_1 \mathrel{-\!\!*} U) * [\vec{z}; L_2 * L_1] \Vdash \left[{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\right]; \vec{y}; A * [U]}$$

This rule states that if there is a hint from the conclusion U of the wand to the goal A, then there is a hint from the wand $L_1 \mathrel{-\!\!*} U$ itself, where the premise L_1 of the wand is added to the side condition L_2. A more complicated recursive hint is the rule for invariants:[2]

$$\frac{\triangleright L_1 * [\vec{z}; L_2] \Vdash \left[{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}}\right]; \vec{y}; A * [U]}{\boxed{L_1}^{\mathcal{N}} * [\vec{z}; L_2 * \ulcorner \mathcal{N} \subseteq \mathcal{E} \urcorner] \Vdash \left[{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}}\right]; \vec{y}; A * \left[U * (\triangleright L_1 \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \chi)\right]}$$

[2] In the implementation, this rule is a consequence of other recursive rules.

This rule states that there is a hint from an invariant $\boxed{L_1}^{\mathcal{N}}$ to a goal A, if there is a hint from the contained assertion L_1 to that atom. We get $\mathcal{N} \subseteq \mathcal{E}$ as an additional side condition, and receive the closing update $(\triangleright L_1 \twoheadrightarrow {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \chi)$ as the residue. Similar to ε_1, the assertion χ is an opaque marker whose semantics is True, but is treated differently by the proof search strategy to enforce closing invariants.

5 Formal Description of the Proof Strategy

In this section we will present an excerpt of the formal grammar of Diaframe (§5.1), and a number of cases of the formal proof search strategy (§5.2). We then present an extension of Diaframe to handle disjunctions (§5.3).

5.1 Grammar of Diaframe

We provide a representative subset of the grammar (a full description can be found in the appendix [62]):

$$
\begin{aligned}
\text{atoms} \quad & A ::= \mathrm{wp}\ e\ \{v.\ L\} \mid \chi \mid \boxed{L}^{\mathcal{N}} \mid \ldots \\
\text{left-goals} \quad & L ::= \ulcorner \phi \urcorner \mid A \mid L * L \mid \exists x.\ L \\
\text{unstructureds} \quad & U ::= \ulcorner \phi \urcorner \mid A \mid U * U \mid \exists x.\ L \\
& \qquad \mid \forall x.\ U \mid L \twoheadrightarrow U \mid {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} U \\
\text{extended} \quad & H ::= \varepsilon_1 \mid U \\
\text{clean hypotheses} \quad & H_C ::= A \mid \forall x.\ U \mid L \twoheadrightarrow U \mid {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} U \\
\text{environments (1)} \quad & \Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \phi \\
\text{environments (2)} \quad & \Lambda ::= \emptyset \mid \Lambda, H_C \qquad \Delta ::= \Lambda, \varepsilon_1 \\
\text{goals} \quad & G ::= \forall x.\ G \mid U \twoheadrightarrow G \mid \mathrm{wp}\ e\ \{v.\ L\} \\
& \qquad \mid {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} L \mid \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \Vert\ \exists \vec{x}.\ L * G
\end{aligned}
$$

The entailments we wish to solve are of the form $\Gamma; \Delta \vdash G$. The atoms A by default only consist of weakest preconditions $\mathrm{wp}\ e\ \{v.\ L\}$, the marker χ (§4.3) and invariants $\boxed{L}^{\mathcal{N}}$. The ellipsis $(\ldots)$ indicates that the set of atoms may be extended by libraries, adding language-specific constructs like $\ell \mapsto v$ or ghost assertions like $\mathtt{locked}\ \gamma$. The definition of Δ explicitly sets the last-resort marker ε_1 as the last hypothesis. Defining Δ in this way avoids having special cases in the description of the strategy, and is close to the Coq implementation.

We have two syntactical categories related to hypotheses H_C and U. Essentially, U is the class of hypotheses for which we are able to recursively find hints. At introduction into the context Δ, we can decompose these into H_C. The goal $\Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \Vert\ \exists \vec{x}.\ L * R$ in G is a 'synthetic' representation of ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \exists \vec{x}.\ L * R$ with the condition $\mathrm{FV}(L) = \vec{x}$. This condition ensures that during hint search we only consider the relevant variables for L. To uphold this condition, our strategy first transforms goals like ${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \exists v_1\, v_2.\ \ell_1 \mapsto v_1 * \ell_2 \mapsto v_1$ into $\Vert {}^{\mathcal{E}}{\Rrightarrow}^{?\mathcal{E}'} \Vert\ \exists v_1.\ \ell_1 \mapsto v_1 * {}^{?\mathcal{E}'}{\Rrightarrow}^{\mathcal{E}} \exists v_2.\ \ell_2 \mapsto v_1$.

5.2 The Proof Search Strategy

If our goal is $\Gamma; \Delta \vdash G$, we do a case analysis on G:

1. $G = \forall x.\ G'$: Continue with $\Gamma, x; \Delta \vdash G'$.
2. $G = U \twoheadrightarrow G'$: Case analysis on U:
   a. $U = \ulcorner \phi \urcorner$: Continue with $\Gamma, \phi; \Delta \vdash G'$.
   b. $U = (U_1 * U_2)$: Continue with $\Gamma; \Delta \vdash U_1 \twoheadrightarrow U_2 \twoheadrightarrow G'$.
   c. $U = (\exists x.\ L)$. Continue with $\Gamma; \Delta \vdash \forall x.\ (L \twoheadrightarrow G')$.
   d. $U = H_C$. Continue with $\Gamma; \Delta, H_C \vdash G'$.
3. $G = \mathrm{wp}\ e\ \{v.\ L\}$:
   a. If e is a value w, continue with $\Gamma; \Delta \vdash {}^{\top}{\Rrightarrow}^{\top} L[w/v]$.
   b. Else, find a K and e' with $e = K[e']$, and quantified specification $\forall \vec{x}.\ \{L\}\ e'\ \{w.\ L'\}$. Continue with $\Gamma; \Delta \vdash \Vert {}^{\top}{\Rrightarrow}^{?\mathcal{E}} \Vert\ \exists \vec{x}.\ L * \Big(\forall w.\ L' \twoheadrightarrow {}^{?\mathcal{E}}{\Rrightarrow}^{\top} \mathrm{wp}\ K[w]\ \{v.\ L\}\Big)$.
4. $G = {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} L$: We consider the following cases:
   a. If the modality ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ is not introducable, continue with $\Gamma; \Delta \vdash \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{?\mathcal{E}_3} \Vert\ \exists \_.\ \chi * {}^{?\mathcal{E}_3}{\Rrightarrow}^{\mathcal{E}_2} L$. The remaining cases assume that ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ is introducable.
   b. $L = \ulcorner \phi \urcorner$: Prove the pure goal ϕ to finish.
   c. $L = \mathrm{wp}\ e\ \{v.\ L'\}$: Continue with $\Gamma; \Delta, H_C \vdash \mathrm{wp}\ e\ \{v.\ L'\}$.
   d. Continue with $\Gamma; \Delta \vdash \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{?\mathcal{E}_3} \Vert\ \exists \_.\ L * {}^{?\mathcal{E}_3}{\Rrightarrow}^{\mathcal{E}_2} \mathrm{True}$ in all other cases.
5. $G = \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \Vert\ \exists \vec{x}.\ L * G'$: Case analysis on U:
   a. $L = \ulcorner \phi \urcorner$: Check that ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ is introducable, and try to solve $\phi[\vec{y}/\vec{x}]$. Continue with $\Gamma; \Delta \vdash G'[\vec{y}/\vec{x}]$.
   b. $L = L_1 * L_2$: Set $\vec{y}_1 = \mathrm{FV}(L_1)$ and $\vec{y}_2 = \vec{x} \setminus \vec{y}_1$, continue with $\Gamma; \Delta \vdash \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{?\mathcal{E}_3} \Vert\ \exists \vec{y}_1.\ L_1 * \Vert {}^{?\mathcal{E}_3}{\Rrightarrow}^{\mathcal{E}_2} \Vert\ \exists \vec{y}_2.\ L_2 * G$.
   c. $L = \exists y.\ L'$: Continue with $\Gamma; \Delta \vdash \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \Vert\ \exists y, \vec{x}.\ L' * G$.
   d. $L = A$: Find the first $H \in \Delta$ with L' and U for which $H * [\vec{y}; L'] \Vdash \big[{}^{\mathcal{E}_3}{\Rrightarrow}^{\mathcal{E}_2}\big]; \vec{x}; A * [U]$. Then continue with $\Gamma; \Delta \setminus H \vdash \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3} \Vert\ \exists \vec{y}.\ L' * (\forall \vec{x}.\ U \twoheadrightarrow G)$.

In the above, we say that ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ is *introducable*, if $\mathcal{E}_2$ can be unified with $\mathcal{E}_1$. Note that Item 3b is SYM-EX-FUPD-EXIST (§3) and Item 5d is BIABD-HINT-APPLY (§4). We have omitted steps in the introduction of magic wands to merge hypotheses and to detect incompatibilities. For example, if we introduce $\mathtt{locked}\ \gamma$ and already have a $\mathtt{locked}\ \gamma$ in our context, we obtain False by LOCKED-UNIQUE. We have also omitted the bookkeeping required to deal with Iris's later modality $(\triangleright)$.

5.3 Extending Diaframe with Disjunctions

The Diaframe grammar does not contain disjunctions. This is intended, as proving disjunctions in a linear logics is challenging. Consider $P * Q \vdash (P \vee Q) * P$. It is crucial to prove the disjunction using Q, since otherwise we are left with the unprovable goal $Q \vdash P$. But if we look at just the disjunction, there is no way to know this in advance.

To offer automation for some goals with disjunctions, we provide an extension of Diaframe. When introducing a disjunction $\Delta \vdash (U_1 \vee U_2) \twoheadrightarrow G$ into the context, continue with goals $\Delta \vdash U_1 \twoheadrightarrow G$ and $\Delta \vdash U_2 \twoheadrightarrow G$ by disjunction elimination. When proving $\Gamma; \Delta \vdash \Vert {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \Vert\ \exists \vec{x}.\ (\ulcorner \phi \urcorner * L_1 \vee L_2) * G$

(and symmetrically), check if we can prove $\neg\phi$, and if so, continue with the simpler goal $\Gamma; \Delta \vdash \| ^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \| \exists \vec{x}.\, L_2 * G$. This makes the pure goal ϕ act as a "guard" on the disjunct.

When a disjunction cannot be handled this way, the proof search strategy will simply stop. It is then up to the user to choose a disjunct, and continue the proof (see the proof of `drop` in §2.2 for an example). To automatically prove more involved examples, Diaframe allow users to *opt-in* on the use of backtracking to choose a disjunct.

6 Implementation and Evaluation

Diaframe is implemented as a library of ca. 14.000 lines of Coq code, built on top of Iris. We use Coq's type class mechanism [76] extensively to make the implementation parametric in (among others) the base proof hints. The recursive hint search strategy (§4.3) and the core proof search strategy (§5.2) are implemented as an Ltac [28] tactic called `iStepsS`. This tactic can be used to prove specifications entirely, and as part of interactive proofs in the Iris Proof Mode [49, 51]. Diaframe comes equipped with 4 ghost-state libraries with bi-abduction hints, to help verify concurrent programs.

To evaluate Diaframe and its implementation, we have verified 24 examples with different levels of complexity. These examples include all the examples used to evaluate Caper [31], Starling [88] and Voila [89], and 5 additional, closely related examples. Our examples do not always correspond line-for-line to the examples from other tools, since the programming languages are different, but the required concurrency reasoning is similar. The examples that we have verified and their statistics can be found in Table 1. This table also includes statistics for manual Iris proofs (if they are available in Iris's Coq distribution).

From this benchmark, we conclude that the use of Diaframe significantly reduces the proof work when using Iris to formally verify programs. Diaframe is competitive with automatic non-foundational tools such as Starling, Voila and Caper, while being *foundational*—generating closed proofs in the Coq proof assistant. The following caveats apply: (a) Starlings constraint-based approach reduces the proof work for some examples, *e.g.,* Peterson's algorithm. For most examples, Diaframe requires less proof work, and is more expressive. (b) Caper outperforms Diaframe with respect to proof work and number of annotations. However, verification with Diaframe is modular, meaning it is easier to verify clients. (c) Voila focuses on TaDA-style logically-atomic specifications [25], which are not supported by Diaframe. Because of this focus, Voila requires more proof work than Diaframe, also for the regular specifications used in this comparison.

We summarize some aggregated data from Table 1. Diaframe can verify 7 of the examples without any help from the user. Averaged over all examples, we require about half a line of manual proof per line of implementation (465 lines of proof for 915 lines of implementation). The highest proof work is in the verification of the Michael-Scott queue [61], requiring 51 lines of proof for an implementation of 37 lines.

We shortly discuss differences between the examples across the tools. Our `bounded_counter` is parametric in the bound, whereas Caper and Voila verify it for a fixed bound of 3. Starling verifies a static version of Peterson's algorithm and the bounded reader-writers lock, whereas we verify a heap-allocated version. The verification of the stack and queues in Diaframe requires (much) more proof search customization than the other examples. This is because they require a recursive definition specifying when a location is a concurrent stack/queue. Diaframe has no native support for recursive definitions yet, so some custom hints need to be provided.

Manual Iris proofs. When comparing with manual Iris proofs, we see that Diaframe takes care of most, if not all, of the proof work. Relatively easy examples like `spin_lock` and `cas_counter` are verified without manual proof work. For harder examples like `ticket_lock` and `bag_stack`, Diaframe saves more than 60 lines of proof work.

Starling. Starling [88] functions as a *proof outline checker*: the user has to supply the intermediate program states after each atomic step, and Starling will then verify whether this transition is valid. Starling is a standalone tool written in F#, and can use different backends as trusted oracles—the Z3 SMT solver [27], or GRASShopper for heap-based reasoning [71]. Its logic is based on the Views framework [29], which enables Starling to express various concurrent reasoning patterns into one core proof rule. This core proof rule produces a finite set of verification conditions for each atomic step, which can then be sent to the trusted oracle. This efficient mapping of atomic steps to verification conditions, together with the ease of defining custom concurrent reasoning patterns, gives Starlings proof automation its power. The downside of the relatively simple logic of Starling is reduced expressivity—it cannot prove functional correctness of *e.g.,* the `bag_stack`. There is also no support for verifying method calls, preventing verification of clients.

Comparing our statistics to those of Starling, one can see that we usually require fewer lines of proof work. This is not surprising, as Starling is a proof outline checker, and thus requires a pre- and postcondition for every atomic operation. A notable exception to the smaller proof obligation is Peterson's algorithm. Stating and proving the invariant for this algorithm in Iris turned out to be quite difficult, and it seems Starling's constraint-based approach is a better fit here. In Table 1, we counted postconditions of atomic operation that are not the last operation as proof work, as well as non-comment lines in program-specific external files.

Caper. Caper [31] is written in Haskell, and uses the Z3 SMT solver [27] as a trusted oracle. The target programs are written in a custom language, and the proof system is

name	impl	annot	custom	**total**	**Iris manual total**	**Starling total**	Caper impl	**Caper total**	Voila impl	**Voila total**
arc [52]	18	28/4	3	62/7		72/16	31	70/1		
bag_stack [18]	29	32	26	96/26	170/92		35	70/0	29	205/36
barrier	58	100/31	3	200/36			71	102/0		
barrier_client	58	106/40	6	184/46			127	189/0		
bounded_counter	20	41/7		74/7			25	50/2	24	79/9
cas_counter	14	31		56/0	95/39		20	40/0	25	68/9
cas_counter_client	16	9		36/0			41	94/0	45	267/36
clh_lock [56]	30	48	3	94/3		134/15				
fork_join	14	29		57/0			17	38/0	16	51/7
fork_join_client	13	9		30/0			32	70/0	28	124/20
inc_dec	23	44		79/0			29	54/0	28	99/12
lclist [16, 85]	28	38/9	13	90/22		197/134				
lclist_extra	119	53	2	182/2						
mcs_lock [59]	54	73/7	3	147/10						
msc_queue [61]	37	65/14	37	174/51						
peterson [69]	46	102/28		166/28		94/5				
queue	42	67/14	37	176/51			60	99/0		
rwlock_duolock [24]	45	54/14		113/14						
rwlock_lockless_faa	27	36/1		74/1			36	68/1		
rwlock_ticket_bounded	40	67/11	2	124/13		109/14				
rwlock_ticket_unbounded	38	62/5		116/5						
spin_lock	13	28		59/0	93/30	76/22	17	39/0	15	65/7
ticket_lock	23	48/5		89/5	168/78	66/11	33	59/0	23	90/12
ticket_lock_client	18	11		39/0			41	79/0	16	87/11
total	912	1338/224	239	2901/465	526/239	748/217	615	1121/4	249	1135/159

Table 1. Data on verified examples. Rows correspond to files in the supplementary material [62]. Columns show number of lines of *impl*ementation of the program, *annot*ation (specifications + invariants) and proof search *custom*ization. The format n/m stands for n lines in total, of which m lines consist of proof work. Proof search customization (*i.e.,* custom hints) is always counted as proof work. The column *total* also includes all remaining Coq boilerplate, like `Import` statements.

based on the CAP logic [30]. This logic contains shared regions (similar to Iris's invariants) and guard algebras (similar to Iris's ghost state/logical resources) to accomodate reasoning about fine-grained concurrency. The cornerstones of Capers proof automation are *backtracking* and *abduction*. These allow Caper to infer that regions should be opened when verifying the execution of a statement in a program. A failure to satisfy some precondition is used as an indication to reattempt the proof with opened regions.

When comparing Diaframe to Caper, we can see that Caper outperforms Diaframe in terms of proof work and annotation overhead. For one, their notations can give implementations and specifications of functions in one go. Caper's proof automation is also simply more powerful—notably, it will 'blindly' open regions in the hope they help proving the goal. Although this makes Caper's automation more powerful, it also makes it slow on failing examples as pointed out by Wolf et al. [89]. In these cases, Diaframe's automation will simply stop at the point where it cannot make progress, while Caper will backtrack through all possible options. In the verification of clients, we outperform Caper because Diaframe's verification is compositional—unlike Caper, we do not need to restate and re-verify a library to verify a client.

For Caper, the lines of proof work in Table 1 consist of no-ops such as `assert (cnt = 1 ? true : true)`, that are used to force case-splits in Caper's proof engine.

Voila. Voila [89] is a proof outline checker for the TaDA logic [25]. Voila takes a user-provided proof outline, turns it into a proof candidate, then verifies this with Viper [63]. Like Caper, Voila uses regions and guard algebras for fine-grained concurrent reasoning. Some program statements need additional annotations containing the relevant reasoning steps, like opening regions. Voila's automation is a combination of applying syntax-driven rules whenever possible, asking the user to provide key rules of the proof, and then using a set of heuristics to fill in gaps for nearly applicable rules.

In the examples in our benchmark, Diaframe usually requires fewer total lines, and fewer lines of user guidance than Voila. Again, this is not surprising, since like Starling, Voila is a proof outline checker. Voila also does not support all the guard algebras that Caper does. This prevents verification of *e.g.,* the queue. However, Voila is capable of (and focused at) verifying TaDA-style logically-atomic specifications. While Iris supports these, Diaframe does not. For Voila, the lines of proof work in Table 1 consist of explicit calls to open/close regions, and explicit uses of atomic specifications.

7 Related Work

There is a lot of work on non-automated verification [45, 48, 57] in foundational tools [3, 17, 37, 43, 65, 75]. We focus on related work in automated verification. Starling [88], Caper [31] and Voila [89] have been covered in §6.

Steel. Steel [36, 81] is a language for developing and verifying concurrent programs in a concurrent separation logic descendant of Iris [43], written in F* [80]. Similar to Diaframe, Steel designed a format to automate the application of certain rules. Their approach uses a notion of Hoare quintuples, and relies on a combination of SMT solving and AC-matching. Diaframe uses weakest preconditions, and avoids reasoning up to commutativity: the order in preconditions and invariants is relevant. Steel excels in automatically proving pure side conditions, leveraging F*'s native use of the Z3 SMT solver [27]. As listed in §8, our support for pure side conditions is rather weak. It is hard to compare Steel's automation for fine-grained concurrency to ours, since Fromherz et al. [36] only covered a spinlock and a parallel increment.

Verification in a weak-memory setting. Summers and Müller [77] presented a prototype tool which can automatically verify fine-grained concurrent programs in a weak memory model. It works by encoding parts of separation logics for weak memory [33, 34, 86] into Viper [63], similar to Voila's approach [89]. It would be interesting to extend Diaframe with support for weak memory using one of the Iris-based logics for weak memory [26, 47, 60].

Bedrock. Bedrock [21, 22] is a foundational tool aimed at verifying sequential programs in an assembly-like language. It is mostly automatic and also based on separation logic. Its automation employs techniques that are somewhat similar to those of Diaframe, in that it tries to syntactically match hypotheses and goals, 'crossing off' hypotheses that appear directly in the goal. Since it is focused at verifying sequential programs, Bedrock's language does not include primitives for fine-grained concurrency.

RefinedC. RefinedC [74] is a recent Iris-based tool for automatic and foundational verification of C-code. One of the main ingredients of RefinedC's automation is a 'separation logic programming language' called Lithium, which, like Diaframe, is based on ideas from linear logic programming. Lithium and Diaframe both prove separating conjunctions in a deterministic left-to-right fashion, and do not backtrack once a hint has been used. Lithium's grammar is more restricted than Diaframe's—it does not contain modalities, so it cannot handle complicated ghost state or Iris's invariants. It is also targeted specifically at proving RefinedC's typing judgments, while we target general Iris weakest preconditions. By encapsulating some concurrency reasoning in typing rules, RefinedC can support limited forms of fine-grained concurrency, like a spin-lock and a one-time barrier. RefinedC has stronger automation and simplification procedures for pure goals, focused at handling complicated sequential programs, but which might be valuable for Diaframe in the future too.

Other non-foundational verification tools. Other automated verification tools are Verifast [10, 42], SmallfootRG [8, 16], and VerCors [67]. The automation of Verifast is very fast and requires little help for sequential code, but requires lots of annotations for fine-grained concurrent code compared to other tools. SmallfootRG is targeted at verifying memory safety, thus cannot prove full functional correctness like Diaframe. VerCors uses process-algebras in addition to separation logic to reason about fine-grained concurrent programs. This approach does lead to reduced expressivity, but has been shown to scale to interesting examples [68].

Logic programming languages for linear and separation logic. There is much prior work on linear logic programming [5, 19, 38, 41], from which our work has drawn inspiration. Like Diaframe, these works use a goal-directed proof-search procedure, and interpret connectives as proof-search instructions. They are usually restricted to the (linear) hereditary Harrop fragment of the logic, but enjoy completeness results on this fragment. Diaframe poses less restrictions on goals, but is necessarily incomplete. Inspired by focusing [1, 55] Diaframe first performs invertible operations.

Separation logic solvers and biabduction. The literature abounds with solvers for (first-order) separation logic [23, 53, 54, 70, 73, 82]. These usually focus on a specific set of atoms (often a variant of the symbolic heap fragment [7]), or target intricate recursive structures, while still enjoying completeness results. Our approach is parametric in the set of atoms, but not able to handle recursive definitions without custom hints. An idea for future work is to investigate what we could learn from these tools.

Calcagno et al. [15] and Brotherston et al. [14] also use bi-abduction, but with a dual goal: shape-analysis, *i.e.,* inferring specifications for programs. They present recursive rules and a decision procedure to solve the bi-abduction problem, but in a more confined separation logic.

8 Limitations and Future Work

We have introduced Diaframe—the first *automated* and *foundational* tool for verification of fine-grained concurrent programs. As the benchmarks in Table 1 show, Diaframe is competitive with automatic non-foundational tools, but there are still plenty of directions for improvements.

Some manual proof work is caused by the lack of support for recursive definitions, for which we want to generate proof hints automatically. In this paper, we have focused on automating the separation logic part of the verification, but for larger examples we want to improve the automation and simplification procedures for pure conditions.

Since we use syntactic unification to drive automation, support for (general) indexing in an array is poor. Verification of data structures such as ring buffers seem like a challenge. It would be useful to develop appropriate hints for arrays.

The verification time of Diaframe is relatively slow. Although 15 out of 24 examples verify in under a minute, the `barrier` example is our slowest with a whopping 25 minutes. We think this can be improved, and wish to properly profile and investigate this slowdown.

Acknowledgments

This research was supported in part by the Dutch Research Council (NWO), project 016.Veni.192.259, and in part by generous awards from Android Security's ASPIRE program and from Google Research.

References

[1] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

[2] Andrew W. Appel. 2006. Tactics for Separation Logic. (2006). http://www.cs.princeton.edu/~appel/papers/septacs.pdf

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. https://doi.org/10.1017/CBO9781107256552

[4] 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

[5] Pablo A. Armelín and David J. Pym. 2001. Bunched Logic Programming. In *IJCAR (LNCS)*. 289–304. https://doi.org/10.1007/3-540-45744-5_21

[6] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. 2012. Charge!. In *ITP (LNCS)*. 315–331. https://doi.org/10.1007/978-3-642-32347-8_21

[7] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *FSTTCS (LNCS)*. 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

[8] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*. LNCS, Vol. 4111. 115–137. https://doi.org/10.1007/11804192_6

[9] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic *(POPL)*. 259–270. https://doi.org/10.1145/1040305.1040327

[10] Dragan Bošnački, Mark van den Brand, Joost Gabriels, Bart Jacobs, Ruurd Kuiper, Sybren Roede, Anton Wijs, and Dan Zhang. 2016. Towards Modular Verification of Threaded Concurrent Executable Code Generated from DSL Models. In *Formal Aspects of Component Software (LNCS)*. 141–160. https://doi.org/10.1007/978-3-319-28934-2_8

[11] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS*. 55–72. https://doi.org/10.1007/3-540-44898-5_4

[12] Stephen Brookes. 2007. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science* 375, 1 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

[13] Stephen Brookes and Peter W O'Hearn. 2016. Concurrent Separation Logic. *CACM* 3, 3 (2016), 19. https://dl.acm.org/citation.cfm?id=2984457

[14] James Brotherston, Nikos Gorogiannis, and Max Kanovich. 2017. Biabduction (and Related Problems) in Array Separation Logic. In *CADE (LNCS)*. 472–490. https://doi.org/10.1007/978-3-319-63046-5_29

[15] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction *(POPL)*. 289–300. https://doi.org/10.1145/1480881.1480917

[16] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. 2007. Modular Safety Checking for Fine-Grained Concurrency. In *SAS (LNCS)*. 233–248. https://doi.org/10.1007/978-3-540-74061-2_15

[17] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J Autom Reasoning* 61, 1 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[18] Thomas J. Watson IBM Research Center and R. K. Treiber. 1986. *Systems Programming: Coping with Parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center.

[19] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. 2000. Efficient Resource Management for Linear Logic Proof Search. *TCS* 232, 1 (2000), 133–163. https://doi.org/10.1016/S0304-3975(99)00173-5

[20] Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *PACMPL* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

[21] Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In *PLDI*. 234–245. https://doi.org/10.1145/1993498.1993526

[22] Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification *(POPL)*. 609–622. https://doi.org/10.1145/2676726.2677003

[23] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. 2015. Automatic Induction Proofs of Data-Structures in Imperative Programs *(PLDI)*. 457–466. https://doi.org/10.1145/2737924.2737984

[24] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. 1971. Concurrent Control with "Readers" and "Writers". *CACM* 14, 10 (1971), 667–668. https://doi.org/10.1145/362759.362813

[25] 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

[26] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *PACMPL* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[27] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[28] 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

[29] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs *(POPL)*. 287–300. https://doi.org/10.1145/2429069.2429104

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

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

[32] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP (LNCS)*. 363–377. https://doi.org/10.1007/978-3-642-00590-9_26

[33] Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI (LNCS)*. 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

[34] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP (LNCS)*. 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

[35] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP (LNCS)*. 173–188. https://doi.org/10.1007/978-3-540-71316-6_13

[36] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. *PACMPL* 5, ICFP (2021), 85:1–85:30. https://doi.org/10.1145/3473590

[37] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building Certified Concurrent OS Kernels. *CACM* 62, 10 (2019), 89–99. https://doi.org/10.1145/3356903

[38] James Harland, David Pym, and Michael Winikoff. 1996. Programming in Lygon: An Overview. In *Algebraic Methodology and Software Technology (LNCS)*. 391–405. https://doi.org/10.1007/BFb0014329

[39] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP (LNCS)*. 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

[40] Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures *(POPL)*. 523–536. https://doi.org/10.1145/2429069.2429131

[41] J.S. Hodas and D. Miller. 1991. Logic Programming in a Fragment of Intuitionistic Linear Logic. In *LICS*. 32–42. https://doi.org/10.1109/LICS.1991.151628

[42] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM (LNCS)*. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

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

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

[45] 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

[46] 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

[47] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP (LIPIcs, Vol. 74)*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[48] 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

[49] 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

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

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

[52] Rust Language. 2021. Arc in Std::Sync - Rust. https://doc.rust-lang.org/std/sync/struct.Arc.html

[53] Quang Loc Le, Jun Sun, and Shengchao Qin. 2018. Frame Inference for Inductive Entailment Proofs in Separation Logic. In *TACAS (LNCS)*. 41–60. https://doi.org/10.1007/978-3-319-89960-2_3

[54] Wonyeol Lee and Sungwoo Park. 2014. A Proof System for Separation Logic with Magic Wand *(POPL)*. 477–490. https://doi.org/10.1145/2535838.2535871

[55] Chuck Liang and Dale Miller. 2009. Focusing and Polarization in Linear, Intuitionistic, and Classical Logics. *TCS* 410, 46 (2009), 4747–4768. https://doi.org/10.1016/j.tcs.2009.07.041

[56] Peter S. Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Proc. of International Parallel Processing Symposium*. 165–171. https://doi.org/10.1109/IPPS.1994.288305

[57] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *PACMPL* 1, OOPSLA (2017), 87:1–87:28. https://doi.org/10.1145/3133911

[58] Andrew McCreight. 2009. Practical Tactics for Separation Logic. In *TPHOLs (LNCS)*. 343–358. https://doi.org/10.1007/978-3-642-03359-9_24

[59] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65. https://doi.org/10.1145/103727.103729

[60] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *PACMPL* 4, ICFP (2020), 96:1–96:29. https://doi.org/10.1145/3408978

[61] 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

[62] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2021. Appendix and Coq mechanization of Diaframe. https://gitlab.mpi-sws.org/iris/automation

[63] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*. LNCS, Vol. 9583. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

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

[65] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP (LNCS)*. 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[66] Peter W. O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *TCS* 375, 1 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

[67] Wytse Oortwijn, Stefan Blom, Dilian Gurov, Marieke Huisman, and Marina Zaharieva-Stojanovski. 2017. An Abstraction Technique for Describing Concurrent Program Behaviour. In *VSTTE*. LNCS, Vol. 10712. 191–209. https://doi.org/10.1007/978-3-319-72308-2_12

[68] Wytse Oortwijn and Marieke Huisman. 2019. Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System. In *Integrated Formal Methods (LNCS)*. 418–436. https://doi.org/10.1007/978-3-030-34968-4_23

[69] G. L. Peterson. 1981. Myths about the Mutual Exclusion Problem. *Inform. Process. Lett.* 12, 3 (1981), 115–116. https://doi.org/10.1016/0020-0190(81)90106-X

[70] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating Separation Logic with Trees and Data. In *CAV (LNCS)*. 711–728. https://doi.org/10.1007/978-3-319-08867-9_47

[71] 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

[72] Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *ESOP (LNCS)*. 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

[73] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. 2016. A Decision Procedure for Separation Logic in SMT. In *Automated Technology for Verification and Analysis (LNCS)*. 244–261. https://doi.org/10.1007/978-3-319-46520-3_16

[74] 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

[75] 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

[76] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics (LNCS)*. 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[77] Alexander J. Summers and Peter Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. In *TACAS (LNCS)*. 190–209. https://doi.org/10.1007/978-3-319-89960-2_11

[78] 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

[79] 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

[80] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. *ICFP* 46, 9 (2011), 266–278. https://doi.org/10.1145/2034574.2034811

[81] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *PACMPL* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

[82] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2018. Automated Lemma Synthesis in Symbolic-Heap Separation Logic. *PACMPL* 2, POPL (2018), 9:1–9:29. https://doi.org/10.1145/3158097

[83] 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

[84] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *OOPSLA*. 691–707. https://doi.org/10.1145/2660193.2660243

[85] Viktor Vafeiadis. 2008. *Modular Fine-Grained Concurrency Verification*. Ph.D. Dissertation. University of Cambridge. http://flint.cs.yale.edu/cs428/doc/viktor-phd-thesis.pdf

[86] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *OOPSLA*. 867–884. https://doi.org/10.1145/2509136.2509532

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

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

[89] 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