Proof Automation

for

Fine-Grained Concurrent Separation Logic

by Ike Mulder

Radboud Universiteit 

The research reported in this thesis has been carried out under the auspices of iCIS (institute for Computing and Information Science) of the Radboud University and the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation series, number 2025-02
Printed by: Gildeprint
Cover design by: Anna Mulder, Eveline Beukers, Ike Mulder
Typeset with LaTeX 2_ε
ISBN: 978-94-6496-310-6

Proof Automation for Fine-Grained Concurrent Separation Logic

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

maandag 3 februari 2025
om 16.30 uur precies

door

Ike Niki Mulder

geboren op 1 september 1995
te Leiden

Promotoren:

dr. R.J. Krebbers
prof. dr. J.H. Geuvers

Manuscriptcommissie:

prof. dr. S.B. Scholz	(voorzitter)
prof. dr. A.W. Appel	(Princeton University, Verenigde Staten)
prof. dr. P. Gardner	(Imperial College London, Verenigd Koninkrijk)
dr. F. Pottier	(Inria, Frankrijk)
prof. dr. T. Wies	(New York University, Verenigde Staten)

Acknowledgements

This book marks the end of my journey towards a PhD degree. Although the journey was mine, it would not have been possible at all without the support of many amazing people, helping me both with getting the academic work done, and with staying sane and happy along the way.

First and foremost, I owe thanks to my supervisors Robbert and Herman. Robbert, thank you for the prodding questions about my work, feedback about my writing, and all the painstaking revisions just before a paper deadline. I really felt that you were invested in making my PhD succeed. Herman, thank you for bringing the bigger picture into the view every now and then, for insightful discussions and career advice.

Doing research sometimes means questioning the use and purpose of your work. Marc, Alexandre, Anton, Léon and Ke, thank you for taking the time to try and use some of my research. Seeing it in action and be helpful really motivated me to keep going, and to try and improve things.

Covid-measures prevented me from going to the office the first year and a half. This made having Jules, Marc and Michael as office mates afterwards all the more valuable. I will miss our discussions on whale-mathematics and other far-fetched stories. Jules, thank you for always looking out for us, and helping me out that time I lost my laptop charger the day before a presentation. Marc, thank you for introducing me to Hanabi. I hope you find a suitable place to do the research you like.

In the summer of 2023, I (physically!) attended OPLSS in Eugene, Oregon. I want to thank Chinmayi, Ellen, Rini, Bruno, Raphael, Yiyang, Marc, Michael, Thomas and all the others for making this a lot of fun. Besides complaining about the food and category theory, we sure made the most of the Eugene dance scene.

Betty&Mora, thank you for serving the best lunch in the neighborhood. I hope you replace the university canteen one day. The 'Wednesdays at the Italian' were something I looked forward to every week, together with (a subset of) Marc, Linus, Léon, Deivid, Michael, Niels, Kasper and Alyzia.

Hanna, thank you for the surprise coffee-breaks and vrijmibo's every once in a while. Nijmegen became more fun when you and Jasper decided to join us in the East.

Aukje, Anna, Lars, thank you for being close-by and having my back through these years. Hans, Sophie, Willem, Liesbeth, thank you for relaxing weekends in Friesland and good conversations. Thank you, Mulder-family, for the mutual support and openness, especially after 'Opa Bas' passed away.

Moving from Delft to Nijmegen often meant a two-hour trainride when we wanted to go and see friends. Fortunately, we did not often need to take the train back on the

same day ;) having a wide range of sleep-over addresses for our ‘weekendjes westen’. Thank you, everyone who hosted us over the years. Furthermore, thank you to all the Cotinga’s, vogels, vosjes, Vola’s, Kaleido’s and others for being wonderful friends.

Finally, I owe the deepest thanks to Eveline. Eef, thank you for keeping me fed when I was swamped by paper deadlines, cheering me up when I was feeling down, and deciding to join me on the Nijmegen-adventure. I really couldn’t have done it without you.

Contents

Chapter 1

Introduction

Software is everywhere in our modern world. It runs in our phones, websites, and probably in some of your household appliances. Ideally, this software should behave precisely like the programmer intended.[1] In reality, software behavior often differs from programmer intention, causing problems ranging from minor inconveniences (*e.g.,* apps closing unexpectedly), to security errors (*e.g.,* Heartbleed (Mitre, 2014), a subtle mistake that caused malicious inputs to a service to leak sensitive data), to major catastrophes (*e.g.,* a power grid failure affecting 50 million people (CBC, 2010)).

Software behavior becomes especially unpredictable in the face of *concurrency*: when multiple cores or threads execute programs simultaneously, while interacting with shared resources. Since almost all modern devices come with multi-core processors, and concurrent algorithms have the potential to outperform their sequential/single-threaded counterparts, concurrency is everywhere. Concurrent algorithms are notoriously hard to get right: one needs to ensure that all possible program interleavings (and their interactions with shared resources) result in intended behavior. Indeed, the power grid failure mentioned before was caused by a concurrency error known as a *race condition*.

Goal of this thesis. We aim to develop methods and tools which:

1. can establish the *correctness* of concurrent programs (*i.e.,* that all possible behavior of the program is 'intended');
2. can do so in a *modular* and *compositional* fashion (*i.e.,* after verifying individual program parts, we should be able to verify any correct composition of these parts);
3. give *trustworthy* results about such programs (*i.e.,* one should not need to trust the tool: if a program is reported to be 'correct', this must be independently verifiable);
4. require *minimal user effort* (*i.e.,* besides the program and a description of its intended behavior, the tool should require minimal additional input).

The verification of concurrent programs is an active area of research, and methods have been developed that satisfy three of these four criteria—we shall discuss these in more detail in §1.2. This thesis extends Iris (Jung et al., 2018b; Krebbers et al., 2017a), existing research that satisfies criteria #1 to #3, with strong automation, to satisfy criteria #4.

[1] *Ideally* software should behave precisely like the user expects. The possible (and probable) mismatch between user expectation and programmer intention is out of scope for this thesis.

Concurrent programs are often built using libraries that provide high-level building blocks for concurrency, such as locks or efficient concurrent data structures. The correctness of these concurrent libraries is our prime verification target—client programs rely on libraries precisely because concurrent programs are hard to get right. Compositional verification is especially relevant in this setting: verified libraries pave the way for verifying client programs, and allow client program verifications to remain agnostic of the implementation intricacies of the library.

Fine-grained vs coarse-grained concurrency. Concurrent programs can roughly be divided into coarse- and fine-grained concurrent programs. *Coarse-grained* concurrent programs rely on so-called 'locks' to guard access to shared resources. This approach allows programs to temporarily get exclusive access to these resources—which prevents other threads from interfering. Exclusivity makes verifying coarse-grained concurrent programs significantly easier, since parts of the program become effectively sequential. However, this usually comes at the cost of performance. Threads will sometimes need to wait for other threads to finish, while it may be safe for them to run concurrently.

On the other hand, *fine-grained* concurrent programs rely on low-level hardware primitives for synchronization. For example, they might use a primitive Fetch And Add (FAA) instruction, which reads and increments a given integer reference in a single program step. Fine-grained concurrent programs are generally more performant, but harder to verify than their coarse-grained (*i.e.,* lock-based) counterparts.

We focus on the verification of fine-grained concurrent programs in this thesis. Implementations of locks usually rely on fine-grained concurrency. Another example that uses fine-grained concurrency is given in Figure 1.1: an Atomic Reference Counter (ARC) adopted from Rust (Rust Language, 2021). ARCs are used to safely share ownership of (and read-access to) a resource among multiple threads. Concurrent graph structures typically use ARCs as a building block. Figure 1.1 shows a simpler client program `client`.

The methods developed in this thesis allow us to automatically prove that the ARC library from Figure 1.1 behaves as intended, *i.e.,* behaves according to a provided description of its indended behavior. Additionally, we can use this to verify clients of the ARC. For example, we can prove that the program `client ()` never crashes: the `asserts` will never fail, and both Use-After-Free memory errors (reading from a freed reference), and Double-Free memory errors (freeing a freed reference) can never happen.

Correctness using separation logic. In this thesis, we use a program logic to describe program behavior. This allows us to express intended program behavior in the language of logic, *i.e.,* with logical specifications. Furthermore, program logics provide proof rules that enable us to formally prove that programs satisfy these specifications. Program logics make it possible to reason about programs in a more convenient and abstract manner—without mentioning the low-level details of the underlying semantics of the language, such as the memory or possible interleavings of other threads.

In particular, we will use *concurrent separation logic* (O'Hearn, 2004, 2007; Brookes, 2007). Concurrent separation logic is an extension of separation logic (O'Hearn et al., 2001; Reynolds, 2002), which is itself an extension of Hoare logic (Hoare, 1969; Floyd, 1967). Hoare logic is concerned with establishing *functional correctness* of programs, *i.e.,* that for each valid input state, running the program results in a valid output state. Separation logic streamlines this process for sequential programs that manipulate pointers, and enables

```
let mk_arc v =
  ref {cnt: 1, val: v}
let clone a =
  FAA (a.cnt) 1; ()
let read a =
  ! (a.val)
let drop a =
  let c = FAA (a.cnt) (-1) in
  if c == 1 then free a else ()
```

```
let thread a =
  let v = read a in
  drop a;
  assert (v == 42)
let client _ =
  let a = mk_arc 42 in
  clone a;
  (thread a || thread a)
```

ARCs are used to safely share ownership of (and read-access to) a resource among multiple threads. This ARC achieves that by keeping a (thread-safe) tally of the number of active readers. The intended behavior of this ARC is as follows: a) `mk_arc` creates an ARC, representing shared ownership of the provided value, b) `clone` increases the reference count, allowing the ARC to be passed to one forked-off thread, c) `read` reads the provided value from an ARC, d) `drop` decreases the reference count, possibly freeing the ARC. Note that `FAA` fetches and increases the integer reference by the given amount, and returns the previous value of the reference.
With these properties, we can show that running `client ()` will never crash. In particular, the `asserts` will never fail, and two important classes of memory errors can never appear: Use-After-Free errors (reading from a freed reference), and Double-Free errors (freeing a freed reference). Note that the call to `clone` is crucial here: if it is omitted, `client ()` will cause a memory error.

Figure 1.1: Implementation of an ARC library and client in an ML like language

compositional verification in this setting. It does so by strictly enforcing local reasoning: specifications describe precisely the part of the heap they operate on, guaranteeing that other parts of the heap stay untouched. Separation logic also turned out to be well-suited for reasoning about concurrently running programs, as long as each program operates on disjoint parts of the heap—building on design principles already proposed by Dijkstra (1968). *Concurrent* separation logic addresses the case we are interested in for this thesis: concurrently running programs operating on shared state. We will get back to it in §1.1.2.

There are many different kinds of 'correctness' to consider for concurrent programs. We shall concern ourselves with two of these: functional correctness, and linearizability. Functional correctness characterizes the input-output behavior of programs: assuming that certain conditions hold on the input state, we try to prove that certain conditions must be true for the output state. Linearizability (Herlihy and Wing, 1990) is a more intricate notion of correctness for concurrent data structures, equivalent to behaving precisely as a sequential data structure guarded by a lock (Filipović et al., 2010). Other correctness properties (*e.g.*, termination, memory leak freedom, dead-lock freedom, confidentiality, fairness of lock acquisitions) are left for future work (see also Chapter 6).

State of the art. Concurrent separation logic is the basis of various recent tools that can be used to verify functional correctness or linearizability of fine-grained concurrent programs. By building on separation logic, these tools satisfy our criteria #1 and #2: being able to prove correctness of concurrent programs in a compositional fashion. For criteria #3 and #4, we can distinguish two approaches in existing tools. On the one hand, *foundational* (Appel, 2001) tools (often built in proof assistants) focus on expressivity and trustworthiness. Foundational tools usually require the user to spell out many of the verification details. On the other hand, *automated* tools focus on minimizing user effort

for verification. These are usually standalone tools with a custom strategy for automating (parts of) the verification, and often rely on calls to external SMT solvers.

Foundational tools take it upon themselves to ensure successful verifications rely only on the axioms of the underlying logic and the operational semantics of the programming language. This is often achieved by embedding the tool in a proof assistant—a computer program that can verify elementary steps of a proof, which we will cover in §1.3. By design, foundational tools do not accept imprecisions or jumps in reasoning: all reasoning steps must be spelled out and explained. Providing these steps automatically is challenging, since foundational tools typically have a rich higher-order logic—incompatible with the first-order logics of automated tools, and incompatible with their automation strategies.

Our contributions. In this thesis, we design, verify and implement logical systems based on concurrent separation logic that can (semi-)automatically verify fine-grained concurrent programs. The resulting tool *Diaframe* is foundational, allows for compositional verification, and provides a high degree of automation. We obtain these first two properties by extending Iris, a framework for concurrent separation logic in the proof assistant Coq. The main challenge in this thesis is therefore *proof automation*.

Iris's logic is an expressive higher-order concurrent separation logic. This poses a challenge for automation—in fact, propositional separation logic is already undecidable (Brotherston and Kanovich, 2014), so any proof automation for Iris will inherently be incomplete. To make matters worse, Iris's more advanced features like higher-order quantification, step-indexing (Appel et al., 2007) and impredicative invariants (Svendsen and Birkedal, 2014) form an integral part of program verifications. These features are fundamentally incompatible with the logics (and therefore the automation) of existing automated tools. We need new proof search strategies for automation.

Separation logic is a form of linear logic, for which significant research effort has been invested in automatically finding proofs. In linear logic programming (Hodas and Miller, 1991; Cervesato et al., 2000), the shape of a proof obligation is interpreted as an instruction for proof search—meaning that a certain logical rule is always applied when the goal has a particular shape. Such syntax-directed proof search was shown to give effective automation in foundational sequential separation logic verification projects like Bedrock (Chlipala, 2011) and RefinedC (Sammler et al., 2021). Unfortunately, this approach is not directly applicable to concurrency in Iris. In particular, resources that are shared are not readily available in a proof—these can only be obtained with Iris's special proof rules for concurrency. Diaframe formulates these rules in a more syntax-directed fashion, giving rise to an efficient, predictable and goal-directed proof search strategy.

Such a strategy needs to overcome the following challenge: how can we know which hypotheses are relevant for the current proof obligation? The naive approach is to exhaustively try each hypothesis and continue proof search, backtracking for failed proof attempts. However, this approach is costly and makes failing verifications very hard to debug. Diaframe overcomes this challenge by taking inspiration from connection calculi (Wallen, 1990; Otten and Kreitz, 1995; Waaler, 2001), and subformula linking (Chaudhuri, 2021; Donato et al., 2022). Diaframe deeply inspects formulas of hypotheses to determine their relevance—hypotheses found relevant are used and never backtracked on.

We evaluated Diaframe by verifying various examples from existing automated tools for fine-grained concurrency. We found that Diaframe's proof burden is competitive with these tools, while adding foundational guarantees.

1.1 Foundations of Concurrent Program Verification

Concurrent program verification has been actively studied for the the past 50 years. We will recap some of that history in §1.1.1, to give the reader an idea of how the methods developed in this thesis relate to earlier methods for concurrent program verification. We continue with a brief primer on (concurrent) separation logic in §1.1.2, as it lies at the basis of many modern methods for concurrent program verification, including ours.

1.1.1 History of Concurrent Program Verification

The seminal work from Owicki (1975) and Owicki and Gries (1976) coined several key techniques for concurrent program verification. They introduced one of the first formal systems for verifying concurrent programs: an extension of Hoare logic that used a check on *interference-freedom* for verifying concurrent programs with shared variables. Another important contribution is that of *proof outlines*: verifications were carried out by annotating every atomic operation in a thread with pre- and postconditions. These proof outlines captured program verifications in a readable and relatively concise format.

Owicki and Gries (1976)'s interference-freedom check operates not just on the programs running in parallel, but also on their proof derivations. Successfully verified individual threads could only be checked for interference after verification, making this approach not compositional. Jones (1981, 1983) addressed this by introducing *rely-guarantee* style reasoning. With rely- and guarantee-conditions, one can explicitly and formally constrain the amount of interference the program expects and causes—before starting the verification. Verifications of threads with matching rely-guarantee conditions can be composed, *i.e.,* such verified threads can safely be run in parallel.

These prior efforts were mostly focused on programs that manipulate simple data such as booleans or integers—not on data structures. This changed when Herlihy and Wing (1990) coined the concept of *linearizability*, a correctness condition for concurrent data structures. A concurrent data structure is linearizable if the effects of operations on the data structure appear to take place instantaneously for clients. Linearizability was not expressed in terms of a Hoare-like logic, and this would remain the case until the work of da Rocha Pinto et al. (2014) and Jacobs and Piessens (2011). Today, linearizability is still widely regarded as the gold standard of correctness for concurrent data structures.

The history of concurrent program verification is richer than we can capture in a subsection, so we have focused on work that is pertinent to this thesis. Other influential work is *e.g.,* Lamport (2002) and Pnueli (1977)'s work on temporal logic for concurrent program verification, and the diverse work on process calculi like CSP (Hoare, 1978) and CCS (Milner, 1980). For more details on the history of concurrent program verification, also see *e.g.,* Brookes and O'Hearn (2016, §1.1) and de Roever et al. (2001, §1.7).

1.1.2 Concurrent Separation Logic

Until the early 2000s, concurrent program verification was mostly focused on programs with shared (global) variables. Realistic programs that manipulated pointers posed a major problem for both sequential and concurrent program verification. Informal verifications of such programs often implicitly assume that the pointers point to appropriately disjoint

regions in memory. It was a major challenge to formalize this reasoning in a way that avoids explicitly stating all the required disjointness conditions.

This changed when O'Hearn et al. (2001) and Reynolds (2002) introduced separation logic, which made modular program verification possible without pointer aliasing problems. Separation logic is an extension of Hoare logic (Hoare, 1969; Floyd, 1967), *i.e.,* a program logic for proving specifications for programs. The main change w.r.t. Hoare logic is in the assertion language of the specifications. Separation logic assertions describe parts of the heap, and can be combined with a special operator that implicitly forces these parts to be disjoint. The logic of these assertions is usually also called separation logic. We will briefly introduce the assertion language before continuing with the program logic.

Assertion language. There are two key ingredients to separation logic's assertion language. Firstly, there is the points-to predicate $\ell \mapsto v$, which describes heaps for which location ℓ points to value v. Secondly, there is the eponymous *separating conjunction* $*$, where $P * Q$ means that the heap can be subdivided into two *disjoint* parts, with P holding in the first part, and Q in the second. The disjointness requirement is the crucial element that makes separation logic so useful. For example, we get that $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \vdash \ell_1 \neq \ell_2$: if both ℓ_1 and ℓ_2 point to some value in disjoint parts of a heap, ℓ_1 and ℓ_2 must be distinct pointers. The separating conjunction thus prevents the need for explicit conditions to prevent pointer aliasing. One such condition might be acceptable for two distinct pointers, but note that for n distinct pointers, one requires non-aliasing conditions for every pair—totalling $\frac{n(n-1)}{2}$ conditions.

A curious property of separation logic is its substructurality: in general, assertions can only be used once. A striking example of this is $\ell \mapsto v \not\vdash \ell \mapsto v * \ell \mapsto v$: unlike with regular conjunction $\wedge$, we cannot duplicate assertions with $*$. This makes it common to interpret separation logic propositions as *resources*. Validity of separation logic proposition P can thus be seen as having ownership of the corresponding resources.

Program logic. To verify programs in separation logic, one uses the primitive rules of the program logic to prove Hoare triples (Hoare, 1969) of the form $\{P\}\ e\ \{Q\}$. This states that whenever the heap satisfies predicate P, executing program e is safe (*i.e.,* does not get stuck), and if e terminates, the resulting heap satisfies predicate Q.[2] One such triple is $\{\ell \mapsto n\}\ \texttt{FAA}\,\ell\,m\ \{\ell \mapsto (n+m) * \texttt{ret} = n\}$, which states that the fetch-and-add instruction FAA fetches the value n, adds m to it, stores the result in ℓ, and returns n.

Note that this triple on FAA does not state anything about what happens to remaining parts of the heap—for example, if you call $\texttt{FAA}\,\ell_1\ 1$ on a heap satisfying $\ell_1 \mapsto 7 * \ell_2 \mapsto 2$, one would hope that ℓ_2 remains untouched. This is indeed the case, and follows automatically from separation logic's *frame rule* FRAME, shown in Figure 1.2.[3] The frame rule states that Hoare triples only operate on the part of the heap described by the precondition: any remaining part of the heap that satisfies R will not be touched, and so R keeps holding on that part of the heap. Modular and compositional verification in separation logic is possible largely because of this frame rule. The frame rule ensures that we can

[2] This is partial correctness: $\{P\}\ e\ \{Q\}$ does not claim that e terminates. Total correctness has also been studied extensively, but guaranteeing termination in a concurrent setting is hard and outside the scope of this thesis.

[3] Note that the frame rule still applies when we choose $R = \ell \mapsto n$ for our Hoare triple for FAA: since no heap satisfies the precondition $\ell \mapsto n * \ell \mapsto n$, the triple holds vacuously.

$$\textsc{frame}\ \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}} \qquad \textsc{par}\ \frac{\{P_1\}\ e\ \{Q_1\} \qquad \{P_2\}\ e\ \{Q_2\}}{\{P_1 * P_2\}\ (e_1 \mid\mid e_2)\ \{Q_1 * Q_2\}}$$

Figure 1.2: Selected proof rules of (concurrent) separation logic

verify sub-programs in isolation, *i.e.,* with pre- and postconditions that only mention the relevant parts of the state, and reuse these results to verify bigger programs.

Concurrent Separation Logic. This idea of disjointness also turns out to be very useful in a concurrent setting, as shown by O'Hearn (2004, 2007) and Brookes (2007) with the introduction of *concurrent* separation logic (CSL). They realized that as long as two programs operate on disjoint resources, it is safe to run the two programs in parallel—and that this can be captured in separation logic with the PAR rule, shown in Figure 1.2.

Although this insight already simplifies reasoning for some class of concurrent programs, it is not sufficient to verify parallel programs that operate on shared state. To verify programs such as $(\mathsf{FAA}\,\ell\,1 \mid\mid \mathsf{FAA}\,\ell\,1)$, or the `client ()` example from Figure 1.1, one needs additional machinery. The original work by O'Hearn (2007) therefore includes a rule for verifying programs that use a *critical region* to guard access to shared state. Later work relaxed this to enable verification of less structured access to shared state, *e.g.,* for programs that use storable locks (Gotsman et al., 2007; Hobor et al., 2008), or programs that use operations marked as atomic (Parkinson et al., 2007).

Another important realization was that one could come up with more resources that satisfied the laws of separation logic, and that these could be useful for program verification. For example, Boyland (2003) suggested a notion of *fractional permissions*, which Bornat et al. (2005) then brought to separation logic. The idea here is to annotate points-to connectives with a fraction $0 < q \leq 1$, with the property that

$$\ell \overset{q_1}{\longmapsto} v * \ell \overset{q_2}{\longmapsto} v \dashv\vdash \ell \overset{q_1+q_2}{\longmapsto} v. \tag{1.1}$$

Here, $\ell \overset{q}{\mapsto} v$ should be interpreted as having a q amount of ownership of location ℓ, *i.e.,* full ownership/write-access if $q = 1$, and partial ownership/read-access for $0 < q < 1$. These fractional permissions allow one to give various threads read-access to a shared location, and recover write-access after the threads have finished.

Note that although the left-hand side of Equation (1.1) has a separating conjunction, the two conjuncts actually refer to resources that are not entirely separate: they describe the same location ℓ. This idea of 'fictional separation' has been very influential. Dinsdale-Young et al. (2010) first brought this idea to CSL in their logic for Concurrent Abstract Predicates (or CAP), a descendant of RGSep (Vafeiadis and Parkinson, 2007) that combines rely-guarantee style reasoning with concurrent separation logic. These predicates can fictionally separate partial knowledge or ownership of fine-grained concurrent data structures, and thereby prove intuitive specifications for them. For example, we can define an `is_arc` predicate to describe the ARC from Figure 1.1, and prove the specifications shown in Figure 1.3. CAP has subsequently been extended to support more flexible usage patterns (Svendsen et al., 2013), and impredicativity (Svendsen and Birkedal, 2014).

```
let mk_arc v =                          {True} mk_arc v {is_arc(ret, v)}
  ref {cnt: 1, val: v}
let clone a =                    {is_arc(a, v)} clone a {is_arc(a, v) ∗ is_arc(a, v)}
  FAA (a.cnt) 1; ()
let read a =                     {is_arc(a, v)} read a {is_arc(a, v) ∗ ret = v}
  ! (a.val)
let drop a =                     {is_arc(a, v)} drop a {True}
  let c = FAA (a.cnt) (-1) in
  if c == 1 then free a else ()
```

Figure 1.3: Specification for the ARC library

Iris. The mid 2010s saw the introduction of a wide variety of concurrent separation logics: extensions of and improvements on earlier versions, often tailored to specific examples. Combining features from different logics was no easy feat, often requiring tweaks to the model of the logic, and thus revisiting soundness arguments. At the same time, more complicated models also meant that mistakes were harder to spot. Researchers were looking for a unifying framework for experimenting with CSL, mechanized in a proof assistant.

An influential attempt at overcoming this challenge is the Iris framework for higher-order concurrent separation logic (Jung et al., 2018b). Iris comes with various state-of-the-art features for CSL, such as impredicative invariants and user-defined ghost state. Indeed, one of Iris's key insights is that the combination of invariants and user-defined ghost state is 'all you need' for concurrent reasoning patterns (Jung et al., 2015). At the same time, Iris is very extensible and allows for effective formal reasoning inside the logic with the dedicated Iris Proof Mode (Krebbers et al., 2017a, 2018)—and all of this is foundational, being embedded in the proof assistant Coq. Since its inception, Iris has been used to formally verify *e.g.,* increasingly complicated fine-grained concurrent programs (Jung et al., 2020), concurrent programs under relaxed memory consistency (Kaiser et al., 2017), a representative part of Rust's type system (Jung et al., 2018a), and a performant concurrent queue by engineers at Meta (Carbonneaux et al., 2022).

Iris thus brings many boons, both for researchers aiming to use or extend CSL, as well as for practitioners trying to verify concurrent programs. In particular, it satisfies our criteria #1 to #3: #1) it can prove the correctness of concurrent programs; #2) it can do so in a compositional fashion, by using CSL; #3) it can do so in a trustworthy way, since it is embedded in a proof assistant. However, despite the relative ease of reasoning that the Iris Proof Mode (Krebbers et al., 2017a, 2018) brings, concurrent program verification in Iris remains a labor-intensive endeavor. In particular, Carbonneaux et al. (2022) note:

> 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 thesis attempts to do precisely that: lessen Iris's verification overhead by coming up with methods to automate proofs in Iris, thereby satisfying criteria #4.

1.2 State of the Art

Concurrent separation logic (or some extension of it) lies at the heart of many modern verification tools for concurrent programs. To see how our approach relates to the state of the art, we discuss modern automated and other noteworthy tools in § 1.2.1 and 1.2.2, respectively.

1.2.1 Automated Concurrent Program Verification Tools

We shall discuss four state-of-the-art tools for automated concurrent program verification: Caper, Voila, Starling and Plankton. All these tools use a form of concurrent separation logic, and aim to prove functional correctness or linearizability of concurrent programs.

Caper (Dinsdale-Young et al., 2017). Caper is an automated verification tool written in Haskell. Caper's proof system is based on the CAP (Concurrent Abstract Predicates) logic (Dinsdale-Young et al., 2010) for fictional separation. This logic has a notion of 'shared regions' that can encode protocols on shared state, allowing Caper to prove functional correctness of fine-grained concurrent programs. Caper achieves an impressive degree of automation: after providing the program, its specification, and the shared regions, verification is fully automatic for most examples.

Caper relies on the SMT solver Z3 (de Moura and Bjørner, 2008) to handle some (sequential) separation logic reasoning. Caper's automation for fine-grained concurrency builds on three ideas: symbolic execution (as pioneered for separation logic by Smallfoot (Berdine et al., 2006)), backtracking, and abduction. In short, Caper checks that a program satisfies a specification by symbolically executing the program. If, somewhere in the program, a precondition of a statement is not met, Caper uses backtracking to use or create shared regions to meet this precondition. This approach of inferring missing proof steps is known as logical abduction.

Starling (Windsor et al., 2017). Starling takes a different approach to automation—building on ideas by Owicki (1975), Starling is a *proof outline checker*. This means that (unlike Caper) Starling requires the user to give a proof outline, *i.e.,* a pre- and postcondition for every atomic statement in the program. Together with a set of 'constraints' for encoding concurrency protocols, Starling can then prove functional correctness.

Starling's logic is based on Views (Dinsdale-Young et al., 2013), a metatheoretical framework that can encode various concurrent reasoning patterns. This allows Starling to recast Owicki and Gries (1976)'s proof rule for checking non-interference and generate verification conditions from the provided proof outline. These verification conditions are then sent to appropriate solvers: to Z3 when verifying concurrent programs with shared variables, and to the separation logic solver GRASShopper (Piskac et al., 2014b) when verifying heap-based concurrent programs.

Voila (Wolf et al., 2021). Voila is another proof outline checker, but aimed at proving the (stronger) property of linearizability. Programs are linearizable if their effects appear to take place instantaneously for clients. Voila verifies proof outlines in the TaDA logic (da Rocha Pinto et al., 2014; da Rocha Pinto, 2016), which proposed the notion of *logical atomicity* as a way to prove linearizability.

Voila generates proofs from proof outlines in two steps. First, proof outlines are expanded to proof candidates using various (syntax-driven) heuristics. These proof candidates are then sent to Viper (Müller et al., 2016), a (sequential) separation logic solver and verification back-end for various automated tools.

Plankton (Meyer et al., 2022, 2023b). Plankton is a recent tool for verifying linearizability of a specific type of concurrent programs: concurrent search structures. Plankton is based on a custom separation logic that was developed with proof automation in mind. By sacrificing some compositionality, their method achieves an astounding degree of automation: it just needs a local 'node' invariant to prove that a program is linearizable.

Plankton builds on the flow framework (Krishna et al., 2018), which enables local reasoning on graph structures—even though local changes may affect the global state of the search structure. Plankton's program logic also uses a version of Owicki and Gries (1976)'s interference check. Their challenge for proof automation is to determine the exact interference caused by other threads, and verify that this does not invalidate the specifications. Plankton determines the interference by iteratively approximating it. Once the interference has been found, verification conditions are discharged using the Z3 SMT solver.

1.2.2 Other Notable Automated Verification Tools

We have just covered tools that aim to achieve goals #1 and #4 (and optionally #2) using separation logic: automated (and compositional) verification of the correctness of concurrent programs. We will now cover some automated tools that focus on a different subset of our goals.

Foundational, automated verification of sequential programs. Various tools target foundational and automated verification with separation logic, focusing mainly on *sequential* programs in realistic languages. We discuss Bedrock, RefinedC and VST.

Bedrock (Chlipala, 2011) targets an assembly-like language, in which Chlipala (2015) verifies a multithreaded, database-backed web application.[4] Bedrock was the first work that achieved foundational and automated separation logic verification. Its approach to automation is a simple yet effective syntax-driven strategy, about which they remark:

> Users of SMT-based verification tools often describe separation logic as too hard to automate, but we think of that statement as only true in the context of normal SMT solvers. A simple syntactic algorithm can be very effective at discharging separation logic implications.

RefinedC (Sammler et al., 2021) targets the C language and builds on the Iris framework for concurrent separation logic. RefinedC uses a combination of refinement and ownership types to describe the state of a program. This allows Lithium, RefinedC's 'separation logic programming language', to drive the proof search entirely by the syntax of the proof obligations. Lithium's proof search is partly inspired by the linear logic programming language Lolli (Hodas and Miller, 1991), but Lithium is simpler and never backtracks.

[4] Although the application is multithreaded, it is not a concurrent program. Threads are run cooperatively (and not in parallel), yielding control to the processor by calling particular library functions. Bedrock's underlying semantics are fundamentally single-threaded/sequential.

RefinedC can automatically verify some simple concurrent libraries, such as a spinlock, if the underlying concurrent reasoning patterns are manually verified beforehand. Lithium has been shown to be applicable to other real-world languages: besides C, it has been used to automatically verify RISC-V and Armv8-A machine-code programs in the Islaris project (Sammler et al., 2022).

VST-Floyd (and VST) (Cao et al., 2018; Appel et al., 2014) also target the C language—or to be precise, CompCert C light. Since Leroy and his team have implemented and verified the CompCert compiler (Leroy, 2009) for this language, a verification in VST guarantees that both the C-program and its compiled version are correct. This means VST can actually guarantee the correctness of the executable built from C code. VST-Floyd provides various semi-automated tactics for symbolic execution and entailment solving, but is focused on sequential programs. Mansky et al. (2017) also verified a concurrent messaging system in VST, but the concurrency reasoning is mostly manual.

Standalone (SMT-based) solvers for separation logic. Automatic verification of sequential programs has also been tackled with SMT-based solvers. For example, GRASShopper (Piskac et al., 2014b) is able to verify various operations on data structures automatically. Due to GRASShopper's decidable specification language, it can give detailed counterexamples for incorrect programs or specifications. GRASShopper achieves decidability by targeting a decidable fragment of separation logic, thereby limiting its expressivity. GRASShopper is used as verification back-end by Starling (see §1.2.1).

Viper (Müller et al., 2016) sacrifices decidability for more expressivity. It supports various permission-based reasoning patterns, such as fractional permissions (Boyland, 2003), and is used as a verification back-end for various program verification tools. Although concurrency is not directly supported by Viper, concurrent reasoning patterns can be encoded into Viper. This is the approach taken by Voila (see §1.2.1).

Other automated tools for concurrent programs. SmallfootRG (Calcagno et al., 2007) was one of the first automated verifiers for concurrent programs. It uses the RGSep logic (Vafeiadis and Parkinson, 2007), and focuses on proving memory safety properties—not full functional correctness. Chalice (Leino and Müller, 2009) is an influential automated verifier that focuses on proving the absence of data races and deadlocks. Another automated verifier is Verifast (Jacobs et al., 2011; Bošnački et al., 2016). Verifast can verify sequential Java and C programs with little help, but requires many annotations for concurrent programs.

VerCors (Blom and Huisman, 2014; Oortwijn et al., 2017) mixes separation logic with process algebraic reasoning to verify (concurrent) Java programs. It also uses Viper as verification back-end. By design, VerCors is less expressive than state-of-the-art concurrent separation logics, but has been successfully applied to large-scale case studies (Oortwijn and Huisman, 2019).

Steel (Swamy et al., 2020; Fromherz et al., 2021) is an imperative language aimed at co-developing concurrent programs with correctness proofs in a concurrent separation logic inspired by Iris. It is embedded in F* (Swamy et al., 2011), a proof assistant that natively uses the Z3 SMT solver. This means Steel has great support for proving entailments that do not mention separation logic. Various concurrent reasoning patterns must be spelled out with ghost code, however.

Non-separation logic approaches to linearizability. Besides separation logic, several other methods have been considered for automatically establishing linearizability. For example, CAVE (Vafeiadis, 2010) (and its extension Poling (Zhu et al., 2015)) uses shape analysis to automatically verify linearizability of various concurrent data structures. The Line-up tool (Burckhardt et al., 2010) instead uses model checking to refute linearizability. For a more thorough survey of methods for establishing linearizability, see Dongol and Derrick (2015).

1.3 Proof Assistants and Proof Automation

We intend to verify correctness of concurrent programs by constructing proofs inside a separation logic. But how can we trust that a tool generates correct proofs? One approach is to mechanize these proofs in a *proof assistant* such as Coq (Coq Development Team, 2024), Lean (de Moura et al., 2015), or Isabelle/HOL (Wenzel et al., 2008). A proof assistant is a computer program capable of verifying mathematical statements and proofs. Proof assistants require the user to spell out and explain all required reasoning steps. Completed proofs are sent to the *kernel* of the proof assistant, a relatively small program that checks the validity of the proof (see also the overview on proof assistants by Geuvers (2009)). To trust results shown in the proof assistant, one only needs to trust its kernel. This keeps the *trusted computing base* as small as possible—in particular, one does not need to trust the intricacies of the bespoke program logic used for verification.

Foundational proofs for separation logic. The native logic of most proof assistants is not separation logic, but rather some form of higher-order logic. We need four additional ingredients to reason about concurrent programs with separation logic in a proof assistant:

1. a formal description of the programming language and its semantics;
2. an embedding of the separation logic inside the proof assistant;
3. a soundness proof of the separation logic: soundness allows us to conclude facts about the behavior of a program from proofs in the separation logic,
4. an effective and verified way to construct proofs inside the separation logic.

Formal proofs are naturally more verbose, and so it is a major challenge to combine these ingredients in a way that proofs of program correctness still have an acceptable size. The Iris framework for concurrent separation logic comes with all these ingredients, and introduced the Iris Proof Mode (Krebbers et al., 2017a, 2018) precisely to ease reasoning in the embedded separation logic. Nevertheless, as also remarked by Carbonneaux et al. (2022), concurrent program verification in Iris remains a labor-intensive endeavor.

We will improve this situation by reusing Iris's first three ingredients—an embedding of a concurrent language and its semantics, Iris's concurrent separation logic, and its soundness proof. Our goal is to replace the fourth ingredient, *i.e.,* the manual program reasoning steps in the Iris Proof Mode, with an automatic procedure that generates verifiable proofs. To see how, let us discuss the typical way of interacting with proof assistants.

Interacting with a proof assistant. The proof assistant we use in this thesis is Coq. Proofs in Coq are usually constructed in an interactive manner. One first enters the statement to be proven, along with any required hypotheses. The proof assistant then

presents this goal to the user, who is asked to fill in the proof. They can fill in the proof using procedures called *tactics*, which replace the goal with (hopefully easier) sub-goals, or which derive information from a given hypothesis. Tactics can be used to *e.g.,* apply inference rules, rewrite with equalities, or to do (partial) computations. To complete a proof, the user has to repeatedly choose appropriate tactics until all sub-goals are solved.

We have several tools available for trying to automate this process. Firstly, we can try to prove inference rules and corollaries which make the required reasoning more concise and ergonomical. Secondly, we can define custom tactics using Coq's meta-programming facilities, such as the tactic languages Ltac (Delahaye, 2000) and Ltac2 (Pédrot, 2019). Custom tactics can express procedures like 'try to prove the goal with one of the following tactics', or 'first apply this lemma, then use that tactic to discharge the first subgoal'.

A priori, there is no clear-cut procedure that automatically and effectively constructs proofs of correctness for concurrent programs. Our approach therefore makes use of both tools: we construct (and prove sound) new inference rules that are helpful for automation, and provide tactics that automatically try to apply appropriate steps in the proof. We now describe some techniques that helped us come up with rules and tactics.

Goal-directed proof search. An important technique for our proof automation is goal-directed proof search. It is a key element in seminal work on linear logic programming (Hodas and Miller, 1991; Cervesato et al., 2000), but also in recent work on separation logic proof automation like Bedrock (Chlipala, 2011) and RefinedC (Sammler et al., 2021).

Consider trying to find a proof for some proof goal G, for which we have a set of hypotheses Δ available to use—usually denoted as $\Delta \vdash G$ in sequent calculi. The logical system in which we work will provide several inference rules to continue the proof. The idea of goal-directed proof search is to choose an inference rule by *just* looking at G. For example, if G is equal to $H \to G'$ (*i.e., H* implies G'), we could choose to always apply the rule for introducing implications. Just looking at the top-level connective for picking inference rules drastically reduces the search space for constructing proofs.

Finding inference rules for atomic goals. Goal-directed search helps us proceed until the goal is a lone atomic formula. At that point, we need to find and use a relevant hypothesis or lemma. Here, we take inspiration from the way connection calculi (Wallen, 1990; Otten and Kreitz, 1995; Waaler, 2001) and subformula linking (Chaudhuri, 2021; Donato et al., 2022) find relevant hypotheses. Connection calculi offer an automated proof search procedure in various non-classical logics, such as intuitionistic or modal logic. They do this by looking for *connections*, *i.e.,* occurrences of a shared atom in two places. The idea behind subformula linking is similar: it looks for shared subformulas in two given formulas, but also computes a sufficient condition for one formula to imply another. Determining the relevance of hypotheses upfront allows us to further reduce the search space for proofs.

1.4 Key Considerations and Ideas

Concretely, the goal of this thesis is to design, verify and implement strategies for proving the correctness of fine-grained concurrent programs in Iris (semi-)automatically. To build strategies that have acceptable performance and are effective on a wide range of examples, we pose the following considerations:

1. *Predictability.* Proof search strategies should take similar actions on similar proof obligations. When the strategy fails to prove something, predictability is key to figuring out why, and thereby contributes to minimizing the proof effort.
2. *Partial progress.* Proof search strategies should try to do as much of the work as possible, even if they fail to construct a complete proof. This means proofs can be constructed with a mixture of automatic and interactive techniques.
3. *Extensibility.* It should be possible to extend the proof search strategy to support new goals, resources or proof rules. This ensures the strategy can remain useful when faced with user-defined resources and in new situations.
4. *Flexibility.* The system should notice when proof rules are almost applicable, and try to generalize appropriately. In particular, the strategy should be able to apply straightforward consequences of proof rules known to the strategy.

These considerations led to the following key ideas for designing proof search strategies:

1. *Goal-directed proof search.* Predictability can be achieved with *goal-directed* proof search. As pioneered by work on linear logic programming (Hodas and Miller, 1991; Cervesato et al., 2000), goal-directed proof search strategies determine an appropriate tactic or inference rule by just looking at the (shape of the) goal, thereby also greatly reducing the search space for proofs.
2. *Avoid backtracking.* Backtracking proof search is a ubiquitous technique for proof automation. However, it has various unwanted properties: for example, when it fails, zero progress is made. Perhaps surprisingly, backtracking can often be avoided. As also advocated by RefinedC (Sammler et al., 2021), we think it should be—thereby improving efficiency, predictability and facilitating partial progress.
3. *Capture proof rules in a general format.* Verifications in Iris use a variety of proof rules. Although the specifics of these rules differ, many of them can be made to fit a general format. By designing our strategy to operate on rules of this general format, we ensure the strategy is broadly applicable and extensible.
4. *Use subformula linking.* Many proof rules require the presence of a specific hypothesis. Sometimes this hypothesis is not explicitly present, but appears in a hypothesis beneath logical connectives. By using ideas from *connection calculi* (Wallen, 1990; Otten and Kreitz, 1995; Waaler, 2001) and *subformula linking* (Chaudhuri, 2021; Donato et al., 2022) to deeply inspect hypotheses, we can detect and still make use of these proof rules, even beneath quantifiers.

1.5 Contributions and Outline

The ideas from §1.4 have been central to the development of Diaframe over the course of this thesis. Diaframe is a proof automation library for Iris, capable of automatically proving the correctness of various fine-grained concurrent programs. In Chapters 2 to 5 of this thesis, we describe the research contributions backing Diaframe. Each of these chapters is based on existing peer-reviewed publications with light edits, and can be read in isolation. In Chapter 6 we reflect, and present conclusions and directions for future work. We give an overview of the research content of the main chapters below.

Chapter 2: Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In this chapter, we present the first version of Diaframe—an automated and foundational tool for verifying functional correctness of fine-grained concurrent programs. We show that Diaframe provides automation competitive to state-of-the-art tools, while adding foundational guarantees. This chapter is based on the publication

- Ike Mulder, Robbert Krebbers, and Herman Geuvers. "*Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris*". Presented at the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2022.

The main contributions of this paper are:

- An entailment and rule format that can capture the proof goals and rules used for proving Hoare triples in Iris's concurrent separation logic.
- A goal-directed proof search strategy operating on these formats, that requires backtracking only when faced with disjunctions.
- An implementation of this strategy in Coq ('Diaframe 1.0').
- A benchmark that compares the proof-burden of Diaframe for the correctness of 24 fine-grained concurrent programs to that of Starling, Caper and Voila.

Chapter 3: Proof Automation for Linearizability in Separation Logic. In this chapter, we present the second version of Diaframe—extending and generalizing the techniques to establish *linearizability* of fine-grained concurrent programs. Linearizability can be established using Iris's support for logically atomic triples, or by proving contextual refinement using ReLoC (Frumin et al., 2018, 2021b). We extend Diaframe to allow encoding proof search strategies for general program verification goals, and provide strategies for both approaches of establishing linearizability. This chapter is based on the publication

- Ike Mulder and Robbert Krebbers. "*Proof Automation for Linearizability in Separation Logic*". Presented at the 38th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), 2023.

We received a *Distinguished Artifact Award* for the artifact accompanying this paper. The main contributions of this paper are:

- Two proof search strategies for establishing linearizability: one for proving refinements in ReLoC, one for proving Iris's logically atomic triples.
- A goal-directed proof search strategy that can be extended for general program verification goals, which we used to encode above two strategies.
- An implementation of this strategy in Coq ('Diaframe 2.0').
- An evaluation of the proof search strategies on existing and new benchmarks. We compare the proof burden for establishing linearizability in Diaframe 2.0 to that of Voila, and to existing interactive proofs in ReLoC and Iris.

Chapter 4: Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. In this chapter, we develop a general approach for developing proof search strategies that can deal with disjunctions *without backtracking*. The approach is

inspired by ideas from connection calculus (Wallen, 1990; Otten and Kreitz, 1995; Waaler, 2001). We describe how our approach can be applied to various logics, and demonstrate this by extending Diaframe. This chapter is based on the publication

- Ike Mulder, Łukasz Czajka, and Robbert Krebbers. "*Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic*". Presented at the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2023.

The main contributions of this paper are:

- A calculus for connection-based proof search in intuitionistic logic.
- A calculus for connection-based proof search in propositional separation logic.
- A calculus for connection-based proof search in Iris's higher-order concurrent separation logic, and a description of how we integrated this into Diaframe.
- An evaluation of Diaframe's improved support for disjunctions, by benchmarking it on the same 24 examples from Chapter 2. The original procedure required backtracking to verify 12/24 examples, we reduce this to just 1/24 examples.

Chapter 5: Unification for Subformula Linking under Quantifiers. In this chapter, we investigate an underlying technique of Diaframe's automation: *subformula linking* (Chaudhuri, 2021; Donato et al., 2022). Subformula linking is a technique for simplifying proof goals by looking for atomic formulas that occur (deeply) in both the goal and an hypothesis. Computing quality simplifications is difficult when quantifiers are involved: the two existing approaches to subformula linking fail to compute good simplifications in some cases. We propose a third approach: Quantifying on the Uninstantiated (QU). This chapter is based on the publication

- Ike Mulder and Robbert Krebbers. "*Unification for Subformula Linking under Quantifiers*". Presented at the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), 2024.

The main contributions of this paper are:

- We present the QU rules for subformula linking under quantifiers, and formally prove that this system lies between the approaches of Chaudhuri (2021) and Donato et al. (2022).
- We present a simple subformula linking procedure in Coq based on QU.
- We show the practical applicability of this system by extending and improving a tactic from the Iris Proof Mode (Krebbers et al., 2017a) with QU linking.
- We describe how Diaframe makes essential use of QU linking in the verification of a classical readers-writer lock by Courtois et al. (1971).

Statement of contributions. The main chapters of this thesis are existing publications with co-authors, with very minor edits. Let me clarify my personal contributions.

For the publications in Chapters 2, 3 and 5, I am the main author. I was responsible for the main research ideas and their implementation, and came up with first drafts of the paper. My co-authors supervised the research process, provided feedback and helped write the final versions of the papers.

For the publication in Chapter 4, my co-author Łukasz Czajka and I closely collaborated to develop the calculi and implementations that improve proof automation support for disjunctions. My contribution was decisive in integrating these ideas into a practical proof automation tool for Iris. I wrote most of the initial draft of the paper, then worked together with my co-authors to edit and improve it to its current version.

Work not included in this thesis. The following work by the author is not included in this thesis. It describes a recipe for proving linearizability of programs under relaxed memory semantics, and explores the use of Diaframe for this recipe.

- Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. "*A Proof Recipe for Linearizability in Relaxed Memory Separation Logic*". Presented at the 45th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2024.

Chapter 2

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

2.1 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 (O'Hearn, 2007; Brookes, 2007; Brookes and O'Hearn, 2016; Feng et al., 2007; Vafeiadis and Parkinson, 2007; Vafeiadis, 2008; Dodds et al., 2009; Dinsdale-Young et al., 2010; Svendsen et al., 2013; Vafeiadis and Narayan, 2013; Turon et al., 2013; Dinsdale-Young et al., 2013; Turon et al., 2014; da Rocha Pinto et al., 2014; Svendsen and Birkedal, 2014; Nanevski et al., 2014; Raad et al., 2015; Jung et al., 2015; Doko and Vafeiadis, 2016, 2017), 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* (producing a closed proof w.r.t. the operational semantics of the language).

Tools with good automation like Caper (Dinsdale-Young et al., 2017), Starling (Windsor et al., 2017) and Voila (Wolf et al., 2021), generally use SMT (de Moura and Bjørner, 2008) or separation-logic solvers (Piskac et al., 2014b; Müller et al., 2016) 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 (Jung et al., 2015, 2016; Krebbers et al., 2017b; Jung et al., 2018b), FCSL (Sergey et al., 2015) and VST (Appel et al., 2014; Cao et al., 2018) are embedded in a proof assistant. Hence, one only needs to trust the implementation of

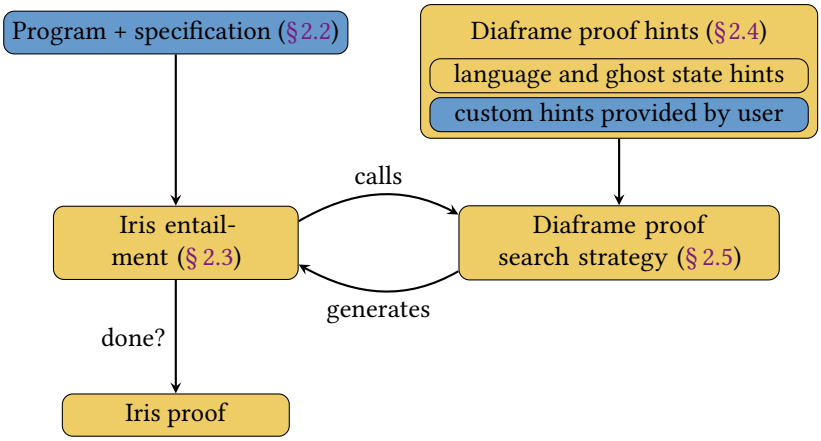

Figure 2.1: Overview of the architecture of Diaframe. User input is marked in blue.

the proof assistant and the operational semantics of the programming language, but not the solvers or underpinned logic. Foundational tools typically provide tactics (Bengtson et al., 2012; Appel, 2006; McCreight, 2009; Krebbers et al., 2017a; Cao et al., 2018; Krebbers et al., 2018) 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 (Svendsen and Birkedal, 2014), for which automation has received little attention, even in a non-foundational setting.

In this chapter, we present **Diaframe**—a foundational tool for automatic verification of fine-grained concurrent programs. Diaframe extends Iris (Jung et al., 2015, 2016; Krebbers et al., 2017b; Jung et al., 2018b)—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 2.1. 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 (Hodas and Miller, 1991) and recent work on separation logic programming (Sammler et al., 2021), our strategy interprets logical connectives as proof search instructions. These instructions simplify and solve (a part of) this entailment, possibly generating remaining proof obligations 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, rules 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 achieve this by taking inspiration from bi-abduction (Calcagno et al., 2009b), 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 (§2.3) and hint format (§2.4) to capture goals and rules in Iris.
- A goal-directed proof search strategy for Iris that can be implemented with little backtracking in Coq (§2.5).
- A benchmark with proofs of correctness of 24 programs using fine-grained concurrency, and a comparison of proof-burden to Starling, Caper, and Voila (§2.6).

We start with two example verifications using Diaframe (§2.2). After covering our contributions (§ 2.3 to 2.6), we discuss related work (§2.7), and limitations and future work (§2.8).

2.2 Diaframe by Example

In this section we showcase Diaframe by verifying a spin lock (§2.2.1) and an Atomic Reference Counter (ARC) (§2.2.2). For both examples we will give Hoare-style specifications $\{P\}\ e\ \{\Phi\}$ in Iris, where $P : \mathit{iProp}$ is a separation logic assertion and $\Phi : \mathit{Val} \to \mathit{iProp}$ 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 $\mathrm{SPEC}\ \{P\}\ e\ \{\vec{y},\ \mathrm{RET}\ v;\ Q\}$ for $\{P\}\ e\ \{w.\ \exists \vec{y}.\ \ulcorner v = w \urcorner * Q\}$ to more succinctly specify return values. We are explicit about the embedding $\ulcorner \phi \urcorner$ of pure Coq proposition ϕ into Iris.

2.2.1 Verification of a Spinlock

Lines 1–8 in Figure 2.2 give the implementation of a spin lock in Iris's default ML-like language HeapLang (Jung et al., 2016). The `newlock` method creates a new lock in the unlocked state by allocating a new location with value `false`. The `acquire` method

```
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 2.2: Verification of a spinlock in Diaframe.

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 puts the lock back to the unlocked state (`false`). The `#` is used to inject Booleans (and other literals) into HeapLang values.

Let us now consider the specification of the lock methods, given in lines 15–26 in Figure 2.2. These specifications use the *representation predicates* `is_lock` γ *lk* C and `locked` γ for locks (Hobor et al., 2008; Svendsen and Birkedal, 2014). Here, `is_lock` γ *lk* C expresses that the lock at location *lk* protects assertions C, 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 C, the `newlock` method returns a value *lk*, for which `is_lock` γ *lk* C holds. The assertion `is_lock` γ *lk* C 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 C. Contrary to `is_lock` γ *lk* C, 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 C, and get nothing in return.

By specifying concurrent data structures with representation predicates (Dinsdale-Young et al., 2010), clients can be easily verified since the implementation details are abstracted away. The `is_lock` γ lk C representation predicate is particularly flexible, since it is impredicative (Svendsen and Birkedal, 2014; Hobor et al., 2008)—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 C. 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. Note that this predicate is not defined in Figure 2.2.[1] For the purpose of this section, we treat it as an abstract Iris primitive with the following rules:

LOCKED-ALLOCATE
$\vdash \dot{\Rrightarrow} \exists \gamma.\ \texttt{locked}\ \gamma$

LOCKED-UNIQUE
$\texttt{locked}\ \gamma * \texttt{locked}\ \gamma \vdash \mathsf{False}$

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 § 2.3.2, 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.

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

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 $P * Q \vdash \Rrightarrow R * S$. We will see some update rules in the next example.

2.2.2 Verification of an ARC

We will now verify a version of an Atomic Reference Counter (ARC), similar to the one verified by Starling (Windsor et al., 2017) and the one used in the Rust standard library (Rust Language, 2021). 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 2.3 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.

To give a specification of the methods of ARC, we make use of shareable assertions, which are typically modeled with fractional permissions (Boyland, 2003). In Iris, shareable assertions are modeled as Iris predicates $P : \mathbb{Q}_p \to \mathit{iProp}$, 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.

As shown on line 1 in Figure 2.3, 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

```
Context (P : Qp → iProp) {HP : Fractional P}.
Definition mk_arc : val :=
  λ: <>, ref #1.
Definition count : val :=
  λ: "a", ! "a".
Definition clone : val :=
  λ: "a", FAA "a" #1 ;; #().
Definition drop : val :=
  λ: "a", (FAA "a" #-1) = #1.
Definition unwrap : val :=
  rec: "unwrap" "a" :=
    if: CAS "a" #1 #0 then #()
    else "unwrap" "a".
Definition arc_inv γ l : iProp :=
  ∃ (z : Z), l ↦ #z ∗ (
     ⌜0 < z⌝%Z ∗ counter P γ (Z.to_pos z)
   ∨ ⌜z = 0⌝ ∗ no_tokens P γ).
Definition is_arc γ (v : val) : iProp :=
  ∃ (l : loc), ⌜v = #l⌝ ∗ inv N (arc_inv γ l).
Global Program Instance mk_arc_spec :
  SPEC {{ P 1 }}
    mk_arc #()
  {{ (v : val) γ, RET v; is_arc γ v ∗ token P γ }}.
Global Program Instance count_spec γ (v : val) :
  SPEC {{ is_arc γ v ∗ token P γ }}
    count v
  {{ (p : Z), RET #p; ⌜0 < p⌝%Z ∗ token P γ }}.
Global Program Instance clone_spec γ (v : val) :
  SPEC {{ is_arc γ v ∗ token P γ }}
    clone v
  {{ RET #(); token P γ ∗ token P γ }}.
Global Program Instance drop_spec γ (v : val) :
  SPEC {{ is_arc γ v ∗ token P γ }}
    drop v
  {{ (b : bool), RET #b; ⌜b = false⌝ ∨
    ⌜b = true⌝ ∗ P 1 ∗ no_tokens P γ }}.
Next Obligation.
  destruct (decide (x2 = 1)); iStepsS.
Qed.
Global Program Instance unwrap_spec γ (v : val) :
  SPEC {{ is_arc γ v ∗ token P γ }}
    unwrap v
  {{ RET #(); P 1 ∗ no_tokens P γ }}.
```

Figure 2.3: Verification of an ARC in Diaframe.

TOKEN-ALLOCATE
$P\,1 \vdash \Rrightarrow \exists \gamma.\ \texttt{counter}\ P\ \gamma\ 1 * \texttt{token}\ P\ \gamma$

TOKEN-INTERACT
$\texttt{no\_tokens}\ P\ \gamma * \texttt{token}\ P\ \gamma \vdash \mathsf{False}$

TOKEN-MUTATE-INCR
$\texttt{counter}\ P\ \gamma\ p \vdash \Rrightarrow (\texttt{counter}\ P\ \gamma\ (p+1) * \texttt{token}\ P\ \gamma)$

TOKEN-MUTATE-DECR
$$\frac{p > 1}{\texttt{counter}\ P\ \gamma\ p * \texttt{token}\ P\ \gamma \vdash \Rrightarrow \texttt{counter}\ P\ \gamma\ (p-1)}$$

TOKEN-MUTATE-DELETE-LAST
$\texttt{counter}\ P\ \gamma\ 1 * \texttt{token}\ P\ \gamma \vdash \Rrightarrow (\texttt{no\_tokens}\ P\ \gamma * \texttt{no\_tokens}\ P\ \gamma * P\,1)$

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

Figure 2.4: Rules for the counter ghost assertions.

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 2.3. Similar to `locked` γ, we treat `token`, `no_tokens` and `counter` abstractly (these are defined via Iris's extensible ghost state mechanism, see our appendix (Mulder et al., 2022a) for the definition), and show only the allocation, interaction and update rules in Figure 2.4. As witnessed by TOKEN-ACCESS, these ghost-state assertions are used to convert fractional permissions into counting permissions (Bornat et al., 2005), 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 $\texttt{counter}\ P\ \gamma\ z$. The $\texttt{counter}\ P\ \gamma\ p$ 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 $\texttt{counter}\ P\ \gamma\ p$ to the required $\texttt{counter}\ P\ \gamma\ (p+1)$. This also gives us the extra `token` that we need in the postcondition.

Integration with interactive proofs. In the verification of `drop`, Diaframe encounters a goal it cannot solve automatically, and gets stuck. The user is presented with the following (slightly simplified) proof state in the Iris Proof Mode (Krebbers et al., 2017a, 2018), where they can use Coq or Iris tactics to help:

```
H : 0 < x2
--------------------------------------------------------------------------
"H1" :  inv N (arc_inv γ l)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - □
"H2" :  token P γ
"H5" :  counter P γ (Z.to_pos x2)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
‖ (⊤\↑N ⊨ ⊤\↑N) ‖
    ⌜0 < x2 + -1⌝ ∗ counter P γ (Z.to_pos (x2 + -1))
  ∨ ⌜x2 + -1 = 0⌝ ∗ no_tokens P γ
(∗) ⊤ ⊨ ⊤ WP #x2 = #1 {{ v, ... }}
```

The statement below - - -∗ indicates our current goal, and contains a disjunction. Both sides of the disjunction contain a pure statement $\ulcorner\phi\urcorner$, but neither of these follow from the relevant hypothesis `H`. On inspection, we need to distinguish two cases: `x2` $= 1$ and `x2` > 1. In the first case, our token was the last one, and we need to use TOKEN-MUTATE-DELETE-LAST to finish the proof. In the second case, other tokens remain, and we need to use TOKEN-MUTATE-DECR.

In Figure 2.3, the manual step consists precisely of the case distinction between `x2` $= 1$ and `x2` > 1, after which Diaframe's `iStepsS` can finish the proof. Even though Diaframe could not figure out the required case distinction automatically, it makes good partial progress here. This is because the automation only performs limited backtracking, and simply stops when it encounters a goal it cannot make progress on.

Note that TOKEN-MUTATE-DELETE-LAST creates *two* `no_tokens` resources: one for the invariant, and one for the postcondition of `drop` or `unwrap`. The `no_tokens` resource in the postcondition ensures that `drop` cannot return `true` (*i.e.,* delete the last remaining `token`) while there are still other `tokens` available. To be precise, if with two `tokens` we call `drop` and `clone` in parallel, we can prove that `drop` returns `false` using TOKEN-INTERACT.

Generality. The ghost assertions `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 2.4 are available to the Diaframe proof search strategy, and applying them requires no extra annotations, except for the manual case distinction for `drop`.

2.3 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 (§ 2.3.1), as commonly done in interactive and automatic tools in proof assistants (Krebbers et al., 2017a; Charguéraud, 2020; Sammler et al., 2021). We then extend this approach with support for Iris's invariant mechanism to verify fine-grained concurrent programs (§2.3.2). We conclude with an overview of the Diaframe entailment format and proof strategy (§2.3.3), which serves as a starting point for the description of our hint format (§2.4).

$$\textsc{wp-value}\ \frac{}{\Phi\, v \vdash \mathsf{wp}\, v\, \{\Phi\}} \qquad \textsc{wp-bind}\ \frac{}{\mathsf{wp}\, e\, \{w.\, \mathsf{wp}\, K[w]\, \{\Phi\}\} \vdash \mathsf{wp}\, K[e]\, \{\Phi\}}$$

$$\textsc{wp-frame}\ \frac{}{R * \mathsf{wp}\, e\, \{\Phi\} \vdash \mathsf{wp}\, e\, \{v.\, R * \Phi\, v\}} \qquad \textsc{wp-mono}\ \frac{\forall v.\, \Psi\, v \vdash \Phi\, v}{\mathsf{wp}\, e\, \{\Psi\} \vdash \mathsf{wp}\, e\, \{\Phi\}}$$

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

Figure 2.5: Some of Iris's rules for weakest preconditions.

2.3.1 Goal-Directed Reasoning with WP

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

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

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 (Krebbers et al., 2017a, 2018) for proving entailments. However, Iris's primitive rules for weakest preconditions in Figure 2.5 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, *i.e.,* one that is applicable for arbitrary premises and postconditions. Recall that `FAA` is used in the `clone` and `drop` methods of ARC (§ 2.2.2).

Suppose we are proving the following entailment:

$$\Delta \vdash \mathsf{wp}\ (\mathsf{FAA}\, \ell\, z)\ \{\Phi\}.$$

(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* (Charguéraud, 2020; Hobor and Villard, 2013). Note that we have put a question mark in front of z_1 to (informally) 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.[2] When we find an hypothesis of shape $\ell \mapsto z$ in Δ, we can *unify* z_1 with z and continue.

We have refrained from substituting $?z_1$ for v in WP-FAA-RAMIFY so that it is more apparent how the rule generalizes for 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. In the terminology of Chalice (Leino and Müller, 2009) and Viper (Müller et al., 2016): exhale P, then inhale $\Psi\, v$ and go on to prove $\mathsf{wp}\ K[v]\ \{\Phi\}$.

2.3.2 Goal-Directed Reasoning with Invariants

We will now extend the naive proof search strategy from §2.3.1 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 §2.2). We will first recapitulate Iris's original proof rule for accessing invariants:

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

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

[2] We could instead weaken the premise of WP-FAA-RAMIFY to $\Delta \vdash \exists z_1.\ l \mapsto z_1 * \ldots$. This existential quantification will play an important role in §2.3.2. We choose to present WP-FAA-RAMIFY without explicit existential quantification here to show the relation to the ramified frame rule.

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* ($\triangleright$) (Nakano, 2000; Appel et al., 2007) is needed for technical reasons caused by the fact that invariants are *impredicative* (Svendsen and Birkedal, 2014; Jung et al., 2018b), *i.e.,* the resource L in an invariant can be *any* resource, including invariants and weakest preconditions. Handling later modalities involves some additional bookkeeping, which Diaframe performs automatically, but we gloss over in this thesis.

We now show why our SYM-EX rule for symbolic execution from §2.3.1 needs to be extended for programs involving fine-grained concurrency. Consider the `FAA` operation in the `clone` method of the ARC (§2.2.2). 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 (Dinsdale-Young et al., 2017). 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:

SYM-EX-FUPD-EXIST

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

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

The fancy update modality ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}$ is used in Iris's definition of weakest preconditions, and is the component that makes opening invariants possible. Semantically, ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ 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 modality, Iris has the following rules:

INV-OPEN-FUPD

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

BUPD-INTRO

$$\frac{}{P \vdash \dot{\Rrightarrow} P}$$

BUPD-FUPD

$$\frac{}{\dot{\Rrightarrow} P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P}$$

FUPD-ELIM

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

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* ($\triangleright L \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$). 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 2.4) 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 § 2.3.1—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 it is necessary to existentially quantify under the fancy update in the new rule. 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 (§2.2.2):

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

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.

2.3.3 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 §2.5):

1. If the goal is $\Delta \vdash \forall x.\, G$ or $\Delta \vdash U \mathrel{-\!\!*} G$, introduce the $\forall$ or $\mathrel{-\!\!*}$. 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 $\Delta \vdash {}^{\top}\!\Rrightarrow^{\top} \Phi\, v$.

3. If the goal is $\Delta \vdash \mathsf{wp}\ K[e]\ \{\Phi\}$, use our new rule SYM-EX-FUPD-EXIST to symbolically execute e. Our new goal has the shape $\Delta \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{?\mathcal{E}_2} \exists \vec{x}.\, L * G$.

4. If the goal is $\Delta \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G$, use associativity of separating conjunction to rewrite it into ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, A * G'$ where A is an atom. Pure conditions $\ulcorner \phi \urcorner$ that 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 $\Delta \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G$, 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 (Hodas and Miller, 1991; Cervesato et al., 2000) and recent work on the separation logic programming language Lithium (Sammler et al., 2021). Other recent work by Chlipala (2011, 2015) has also shown that using the syntax of the goal to guide proof search works well for automatic foundational verification. The inspiration by Lithium can be seen most clearly in the reversible actions described in Items (1-a) and (1-b)—these are the same as those performed by Lithium. The key difference is that we do not operate on top-level connectives, but on connectives that appear below a modality and a number of existentials, to support Iris's invariants and ghost state.

2.4 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 2.4, (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 in which hints can be selected. First, *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. Second, *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* (§ 2.4.1) that is inspired by the technique of bi-abduction (Calcagno et al., 2009b). Aside from the base hints (§ 2.4.2), Diaframe provides *recursive hints* to close the base hints under connectives like invariants, magic wands, and separating conjunctions (§ 2.4.3).

2.4.1 Bi-Abduction Hints

The *hint format* of Diaframe is as follows:

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

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. Last-resort hints have ε_1 for the hypothesis H. 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 (Calcagno et al., 2009b) 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 $H * ?\mathit{antiframe} \vdash A * ?\mathit{frame}$. Our hints' shape is also similar to the residuation judgment from Cervesato et al. (2000), 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.

There are no theoretical guarantees that choosing the first hypothesis for which a hint can be found is always desirable. Indeed, it is easy to construct artificial examples where this approach makes the goal unprovable. However, in the practical examples we have seen, there is usually either precisely one such hypothesis, or there are more such hypotheses, but the choice is immaterial. This may be a result of the substructural nature of separation logic: since resources cannot be duplicated, any way to obtain a required resource is usually the correct way.

2.4.2 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 §2.2.2) 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 §2.2.1) 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: Points-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 $q_1 > q_2$.

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:

$$\frac{U_1 * [\vec{z}; L_2] \Vdash \left[{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\right] \vec{y}; A * [U_2]}{(L_1 \mathrel{-\!\!*} U_1) * [\vec{z}; L_2 * L_1] \Vdash \left[{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\right] \vec{y}; A * [U_2]}$$

This rule states that if there is a hint from the conclusion U_1 of the wand to the goal A, then there is a hint from the wand $L_1 \mathrel{-\!\!*} U_1$ 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:[3]

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

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 $(\triangleright L_1 \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \chi)$ 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.

2.5 Formal Description of the Proof Strategy

In this section we will present an excerpt of the formal grammar of Diaframe (§2.5.1), and a number of cases of the formal proof search strategy (§2.5.2). We then present an extension of Diaframe to handle disjunctions (§2.5.3).

2.5.1 Grammar of Diaframe

We provide a representative subset of the grammar (a full description can be found in the appendix (Mulder et al., 2022a)):

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

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

The entailments we wish to solve are of the form $\Gamma; \Delta \vdash G$. The atoms A by default only consist of weakest preconditions $\mathsf{wp}\ e\ \{v.\, L\}$, the marker χ (§2.4.3) and invariants $\boxed{L}^{\mathcal{N}}$. The ellipsis (...) indicates that the set of atoms may be extended by libraries, adding language-specific constructs like $\ell \mapsto v$ or ghost assertions like $\texttt{locked}\ \gamma$. 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 $\|{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \|\, \exists \vec{x}.\, L * R$ 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 and instantiate variables that are bound in L, and leave the rest existentially quantified. To uphold this condition, our strategy first transforms goals like ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists v_1\, v_2.\, \ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$ into $\|{}^{\mathcal{E}_1}\!\Rrightarrow^{?\mathcal{E}'} \|\, \exists v_1.\, \ell_1 \mapsto v_1 * {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}_2} \exists v_2.\, \ell_2 \mapsto v_2$. The strategy will continue on this goal by first obtaining $\ell_1 \mapsto v'$ and instantiating just v_1 with v'. Only after that, it will consider the remaining goal ${}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}_2} \exists v_2.\, \ell_2 \mapsto v_2$.

2.5.2 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. $G = U \mathrel{-\!\!*} G'$: Case analysis on U:
   (a) $U = \ulcorner \phi \urcorner$: Continue with $\Gamma, \phi; \Delta \vdash G'$.
   (b) $U = (U_1 * U_2)$: Continue with $\Gamma; \Delta \vdash U_1 \mathrel{-\!\!*} U_2 \mathrel{-\!\!*} G'$.
   (c) $U = (\exists x.\, L)$. Continue with $\Gamma; \Delta \vdash \forall x.\ (L \mathrel{-\!\!*} G')$.
   (d) $U = H_C$. Continue with $\Gamma; H_C, \Delta \vdash G'$.
3. $G = \mathsf{wp}\ e\ \{v.\, L\}$:
   (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 $\forall \vec{x}.\, \{L_1\}\ e'\ \{w.\, L_2\}$. Continue with $\Gamma; \Delta \vdash \|{}^{\top}\!\Rrightarrow^{?\mathcal{E}} \|\, \exists \vec{x}.\, L_1 * \left(\forall w.\, L_2 \mathrel{-\!\!*} {}^{?\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{wp}\ K[w]\ \{v.\, L\}\right)$.
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, *i.e.*, $\mathcal{E}_1 \neq \mathcal{E}_2$, continue with goal $\Gamma; \Delta \vdash \|{}^{\mathcal{E}_1}\!\Rrightarrow^{?\mathcal{E}_3} \|\, \exists \_.\, \chi * {}^{?\mathcal{E}_3}\!\Rrightarrow^{\mathcal{E}_2} L$. As mentioned in §2.4.3, this case is responsible for closing invariants. 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) $L = \mathsf{wp}\ e\ \{v.\, L'\}$: Remove the fancy update, and continue with $\Gamma; \Delta \vdash \mathsf{wp}\ e\ \{v.\, L'\}$.

(d) In all other cases, continue with $\Gamma; \Delta \vdash \|^{\mathcal{E}_1}\Rrightarrow^{?\mathcal{E}_3} \| \exists \_.\, L \ast {}^{?\mathcal{E}_3}\Rrightarrow^{\mathcal{E}_2} \mathsf{True}$. This will be one of the final steps of a successful proof: once L has been established with Item 5, we will reach Item (4-b) and terminate.

5. $G = \|^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \| \exists \vec{x}.\, L \ast G'$: Case analysis on L:

(a) $L = \ulcorner \phi \urcorner$: Check that ${}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2}$ is introducable, and try to solve $\phi[\vec{y}/\vec{x}]$ for fresh Coq evars $\vec{y}$. Continue with $\Gamma; \Delta \vdash G'[\vec{y}/\vec{x}]$.

(b) $L = L_1 \ast L_2$: Set $\vec{y}_1 = \mathrm{FV}(L_1)$ and $\vec{y}_2 = \vec{x} \setminus \vec{y}_1$, then continue with goal: $\Gamma; \Delta \vdash \|^{\mathcal{E}_1}\Rrightarrow^{?\mathcal{E}_3} \| \exists \vec{y}_1.\, L_1 \ast \|^{?\mathcal{E}_3}\Rrightarrow^{\mathcal{E}_2} \| \exists \vec{y}_2.\, L_2 \ast G$.

(c) $L = \exists y.\, L'$: Continue with $\Gamma; \Delta \vdash \|^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \| \exists y, \vec{x}.\, L' \ast G$.

(d) $L = A$: Find the first $H \in \Delta$ with side condition L' and residue U for which $H \ast [\vec{y}; L'] \Vdash \left[{}^{\mathcal{E}_3}\Rrightarrow^{\mathcal{E}_2}\right] \vec{x}; A \ast [U]$. Then continue with new goal: $\Gamma; \Delta \setminus H \vdash \|^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_3} \| \exists \vec{y}.\, L' \ast (\forall \vec{x}.\, U \mathrel{-\!\!\ast} G)$.

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 (3-b) is SYM-EX-FUPD-EXIST (§ 2.3) and Item (5-d) is BIABD-HINT-APPLY (§ 2.4). We have omitted steps in the introduction of magic wands to merge hypotheses and to detect incompatibilities. For example, if we introduce `locked` γ and already have a `locked` γ in our context, we obtain False by LOCKED-UNIQUE. We have also omitted the bookkeeping required to deal with Iris's later modality ($\triangleright$).

2.5.3 Extending Diaframe with Disjunctions

The Diaframe grammar does not contain disjunctions. This is intended, as proving disjunctions in linear logics is challenging. Consider $P \ast Q \vdash (P \lor Q) \ast P$. 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 $\Delta \vdash (U_1 \lor U_2) \mathrel{-\!\!\ast} G$ into the context, continue with goals $\Delta \vdash U_1 \mathrel{-\!\!\ast} G$ and $\Delta \vdash U_2 \mathrel{-\!\!\ast} G$ by disjunction elimination. When proving $\Gamma; \Delta \vdash \|^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \| \exists \vec{x}.\, (\ulcorner \phi \urcorner \ast L_1 \lor L_2) \ast G$ (and symmetrically), check if we can prove $\neg\phi$, and if so, continue with the simpler goal $\Gamma; \Delta \vdash \|^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \| \exists \vec{x}.\, L_2 \ast G$. 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 §2.2.2 for an example). To automatically prove more involved examples, Diaframe allow users to *opt-in* on the use of backtracking to choose a disjunct.

2.6 Implementation and Evaluation

Diaframe is implemented as a library of ca. 15.000 lines of Coq code, built on top of Iris. We use Coq's type class mechanism (Sozeau and Oury, 2008) extensively to make the implementation parametric in (among others) the base proof hints. The recursive hint search strategy (§2.4.3) and the core proof search strategy (§2.5.2) are implemented as an Ltac (Delahaye, 2000) tactic called `iStepsS`. This tactic can be used to prove specifications

entirely, and as part of interactive proofs in the Iris Proof Mode (Krebbers et al., 2017a, 2018). Diaframe comes equipped with 5 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 (Dinsdale-Young et al., 2017), Starling (Windsor et al., 2017) and Voila (Wolf et al., 2021), 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. These examples and their statistics are shown in Figure 2.6. 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 (da Rocha Pinto et al., 2014), 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 Figure 2.6. Diaframe can verify 7 of the examples without any help from the user. Averaged over all examples, we require about 0.4 line of manual proof per line of implementation (321 lines of proof for 823 lines of implementation). The highest proof work is in the verification of the Michael-Scott queue (Michael and Scott, 1996), requiring 46 lines of proof for an implementation of 37 lines. All but two examples can be verified in under two minutes on our 3960X Threadripper (averaged over 10 runs). The two exceptions are slow mainly because their invariants contain an n-fold disjunction, with n relatively high (≥ 10).

Hints and proof search customization. Diaframe has access to 30 bi-abduction hints, available in our 5 ghost-state libraries. In Figure 2.6, user-provided hints and their required lemmas count as proof search customization. 8 user-provided bi-abduction hints were necessary to verify examples with recursive definitions, as Diaframe does not have native support for such definitions as of yet. Other ways to customize the proof search are: strengthening the pure solver, and instructing Diaframe to merge some hypotheses. Merging hypotheses may be necessary to find relevant equalities or contradictions. The process of designing a user-provided hint is generally as follows. First, run Diaframe until it gets stuck. Next, inspect the available hypotheses and goal, looking for a hypothesis that indicates a way to prove the left-most atom in the goal. Finally, create and prove this new hint, and repeat.

Ghost-state libraries. Diaframe provides 5 libraries with bi-abduction hints for ghost-state resources. These libraries are concerned with (a) allocating Iris invariants, (b) the `token` resources from §2.2.2, (c) ticket-like resources, (d) Iris's `own` connective for general

name	impl	annot	custom	hints used	time	total	iris manual total	starling total	caper total	voila total
arc (Rust Language, 2021)	18	28/4	3	5	0:10	62/7		72/16	70/1	
bag_stack (Treiber, 1986)	29	45/2	34	7(3)	0:17	117/36	170/92		70/0	205/36
barrier	58	100/31	5	14	13:22	200/38			102/0	
barrier_client	58	98/38	6	6	0:50	175/44			189/0	
bounded_counter	20	41/7		4	0:11	73/7			50/2	79/9
cas_counter	14	31		2	0:08	56/0	95/39		40/0	68/9
cas_counter_client	16	9		4	0:06	36/0			94/0	267/36
clh_lock (Magnusson et al., 1994)	30	48	3	7	0:22	94/3		134/15		
fork_join	14	29		2	0:08	57/0			38/0	51/7
fork_join_client	13	9			0:04	30/0			70/0	124/20
inc_dec	23	44		6	0:31	78/0			54/0	99/12
lclist (Vafeiadis, 2008; Calcagno et al., 2007)	28	34/5	13	2(2)	0:27	86/18		197/134		
lclist_extra	119	53	2	3(2)	1:31	182/2				
mcs_lock (Mellor-Crummey and Scott, 1991)	54	73/7	4	9	1:11	147/11				
msc_queue (Michael and Scott, 1996)	37	56/5	41	13(3)	1:42	168/46				
peterson (Peterson, 1981)	46	102/28		7	7:51	166/28		94/5		
queue	42	58/5	41	12(3)	1:17	170/46			99/0	
rwlock_duolock (Courtois et al., 1971)	45	50/10		7	0:21	109/10				
rwlock_lockless_faa	27	36/1		8	0:20	74/1			68/1	
rwlock_ticket_bounded	40	68/10	2	13	0:54	124/12		109/14		
rwlock_ticket_unbounded	38	62/5		8	0:21	116/5				
spin_lock	13	28		3	0:06	59/0	93/30	76/22	39/0	65/7
ticket_lock	23	49/6		5	0:23	90/6	168/78	66/11	59/0	90/12
ticket_lock_client	18	11		1	0:06	39/0			79/0	87/11
total	823	1162/164	154	38(8)	32:30	2518/321	526/239	748/217	1121/4	1135/159

Figure 2.6: Data on verified examples. Rows correspond to files in the supplementary material (Mulder et al., 2022a). 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.,* user-provided hints) is always counted as proof work. In the hints column, notation $h(c)$ stands for h distinct hints used for the proof, c of which were custom/user-provided. The time column displays the average verification time in minutes:seconds. The column *total* also includes all remaining Coq boilerplate, like `Import` statements.

ghost-state resources, (e) writing integers as the difference between two naturals that can only increase. All of these libraries are used by at least two of the examples in Figure 2.6.

Performance for failing verifications. One rarely gets the verification of these examples right in one go. It is therefore important to consider the performance of Diaframe when verification fails. In our artifact (Mulder et al., 2022a)one can find several examples that intentionally fail, obtained by changing the code, postcondition or omitting induction hypotheses. In all these cases, failing times were lower than the final verification time in Figure 2.6.

Differences between the examples across tools. We verify `bounded_counter` for a parametric bound, whereas Caper and Voila fix the bound to 3. Starling verifies a static version of Peterson's algorithm and a bounded reader-writers lock, whereas we verify a heap-allocated version.

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 50 lines of proof work.

Starling. Starling (Windsor et al., 2017) 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 (de Moura and Bjørner, 2008), or GRASShopper for heap-based reasoning (Piskac et al., 2014b). Its logic is based on the Views framework (Dinsdale-Young et al., 2013), 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 Figure 2.6, 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 (Dinsdale-Young et al., 2017) is written in Haskell, and uses the Z3 SMT solver (de Moura and Bjørner, 2008) as a trusted oracle. The target programs are written in a custom language, and the proof system is based on the CAP logic (Dinsdale-Young et al., 2010). This logic contains shared regions (similar to Iris's invariants) and guard algebras (similar to Iris's ghost state/logical resources) to accommodate reasoning about

fine-grained concurrency. The cornerstones of Caper's 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. (2021). 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 Figure 2.6 consist of no-ops inserted in programs such as `assert (cnt = 1 ? true : true)`. These no-ops are used to force case-splits in Caper's proof engine.

Voila. Voila (Wolf et al., 2021) is a proof outline checker for the TaDA logic (da Rocha Pinto et al., 2014). Voila takes a user-provided proof outline, turns it into a proof candidate, then verifies this with Viper (Müller et al., 2016). 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 Figure 2.6 consist of explicit calls to open/close regions, and explicit uses of atomic specifications.

2.7 Related Work

There is a lot of work on non-automated verification (Mansky et al., 2017; Jung et al., 2020; Kim et al., 2017) in foundational tools (Nanevski et al., 2014; Sergey et al., 2015; Appel et al., 2014; Cao et al., 2018; Jung et al., 2016; Gu et al., 2019). We focus on related work in automated verification. Starling (Windsor et al., 2017), Caper (Dinsdale-Young et al., 2017) and Voila (Wolf et al., 2021) have been covered in §2.6.

Steel. Steel (Swamy et al., 2020; Fromherz et al., 2021) is a language for developing and verifying concurrent programs in a concurrent separation logic descendant of Iris (Jung et al., 2016), written in F* (Swamy et al., 2011). 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 (de Moura and Bjørner, 2008). As listed in §2.8, our support for pure side conditions is rather weak, and would benefit from stronger pure automation. It is hard to compare Steel's automation for fine-grained concurrency to ours, since Fromherz et al. (2021) only covered a spinlock and a parallel increment.

Verification in a weak-memory setting. Summers and Müller (2018) 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 (Vafeiadis and Narayan, 2013; Doko and Vafeiadis, 2016, 2017) into Viper (Müller et al., 2016), similar to Voila's approach (Wolf et al., 2021). It would be interesting to extend Diaframe with support for weak memory using one of the Iris-based logics for weak memory (Kaiser et al., 2017; Dang et al., 2020; Mével et al., 2020).

Bedrock. Bedrock (Chlipala, 2011, 2015) is a mostly-automatic foundational tool for verifying sequential programs in an assembly-like language. Its separation-logic based automation employs techniques that are somewhat similar to those of Diaframe. It tries to syntactically match hypotheses and goals, 'crossing off' hypotheses that appear directly in the goal. More involved reasoning steps, like updating ghost-state, require explicit annotations, and we expect that this would not give the amount of automation that Diaframe provides.

RefinedC. RefinedC (Sammler et al., 2021) is a recent Iris-based tool for automatic and foundational verification of C programs. One of the main ingredients of RefinedC's automation is the 'separation logic programming language' Lithium, which, like Diaframe, is based on ideas from linear logic programming. Lithium and Diaframe employ the same rules for introducing variables and hypotheses, 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, which might be valuable for Diaframe in the future too.

Other non-foundational verification tools. Other automated verification tools are Verifast (Jacobs et al., 2011; Bošnački et al., 2016), SmallfootRG (Berdine et al., 2006; Calcagno et al., 2007), and VerCors (Oortwijn et al., 2017). The automation of Verifast is very fast and requires little help for sequential code, but many annotations for fine-grained concurrent code compared to other tools. SmallfootRG is targeted at memory safety, thus cannot prove full functional correctness like Diaframe. Like Diaframe, Verifast and Smallfoot use automation by symbolic execution. An important difference is the use of a symbolic heap, which facilitates permission and value queries. We do not have this option in Iris, so instead of operating on the entire heap at once, we operate on a single hypothesis at a time. 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 (Oortwijn and Huisman, 2019).

Logic programming languages for linear and separation logic. There is much prior work on linear logic programming (Armelín and Pym, 2001; Harland et al., 1996; Hodas and Miller, 1991; Cervesato et al., 2000), 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 (Andreoli, 1992; Liang and Miller, 2009) Diaframe first performs invertible operations.

Separation logic solvers and bi-abduction. The literature abounds with solvers for (first-order) separation logic (Lee and Park, 2014; Reynolds et al., 2016; Piskac et al., 2014a; Le et al., 2018; Ta et al., 2018; Chu et al., 2015). These usually focus on a specific set of atoms (*e.g.,* the symbolic heap fragment (Berdine et al., 2004)), or intricate recursive structures while enjoying completeness results. Diaframe is parametric in the set of atoms, but not able to handle recursive definitions without user-defined hints. Calcagno et al. (2009b) and Brotherston et al. (2017) 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.

2.8 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 Figure 2.6 show, Diaframe is competitive with automatic non-foundational tools, but there are still plenty of directions for improvements.

A limitation of Diaframe is that it cannot handle goals that do not fit the grammar. In particular, there is no support for magic wands in invariants. Although these can be avoided in most cases, some examples remain out of reach—for example, the barrier verified by Jung et al. (2016). The specification of this barrier is tricky, allowing clients to weaken and split the resource on which the threads are waiting. The verification of this barrier features a magic wand inside an invariant, which does not appear to be rewritable to simpler terms.

Some manual proof work is caused by the lack of support for recursive definitions for resources in Iris. Such resources are ubiquitous when verifying recursive data structures like stacks and queues. Proof hints for such resources should be automatically generated.

In this chapter, 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.

When our automation gets stuck on a goal, it can sometimes be unclear why this goal remains, and what happened before. This occurs most often in programs with multiple

branches and/or invariants with disjunctions. We leave improving the user interaction in these cases for future work.

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 18 out of 24 examples verify in under a minute, the `barrier` example is our slowest, taking 14 minutes. We think this can still be improved, and wish to investigate this.

Diaframe's proof search strategy could, in principle, be used whenever goals can be rewritten into Diaframe's entailment format. This can be done for logically-atomic specifications, and can also be done for ReLoC's refinement judgment (Frumin et al., 2018, 2021b). However, both these types of goals present additional challenges for automatic verification—one of which is that there are multiple seemingly valid (and syntactically similar) ways to proceed with a proof. We shall get back to this in Chapter 3.

Chapter 3

Proof Automation for Linearizability in Separation Logic

3.1 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 "gold 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 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 was originally 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 (Vafeiadis, 2010; Henzinger et al., 2013; 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 (da Rocha Pinto et al., 2014; Jung et al., 2015; Birkedal et al., 2021), 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., 2015, 2016; Krebbers et al., 2017a,b; Jung et al., 2018b), and our work in Chapter 2 provides a starting point for proof automation in Iris.

Linearizability follows from contextual refinement $e \preceq_{\mathsf{ctx}} e'$, 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. (2017a) 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 was 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., 2017a, 2018). 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 chapter 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 (Gregersen et al., 2021; Frumin et al., 2021a)). 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 machine-checked 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 chapter).

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

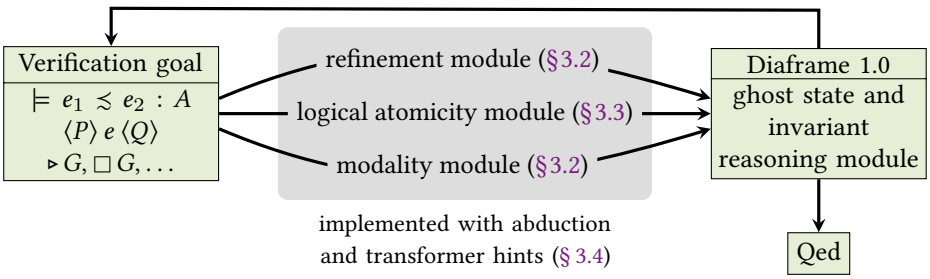

Figure 3.1: Overview of the architecture of Diaframe 2.0.

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 Figure 3.1. 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, as presented in Chapter 2. Diaframe 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'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 strategy for Hoare triples (so Diaframe 2.0 is backwards compatible w.r.t. Diaframe). *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).

Following ideas from Gonthier et al. (2011), Spitters and Weegen (2011) and Krebbers et al. (2017a), 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 §3.2 we describe our proof automation strategy for verifying contextual refinement in ReLoC.
- In §3.3 we describe our proof automation strategy for verifying logically atomic triples in Iris.
- In §3.4 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). The proof automation strategies for our first two contributions are encoded in Diaframe 2.0.
- In §3.5 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 (§3.5.1). We compare to existing interactive proofs of RDCSS and the elimination stack in Iris, showing an average proof size reduction by a factor 4 (§3.5.2). Our new result is a proof of logical atomicity for the Michael-Scott queue (Michael and Scott, 1996) (§3.5.3). For refinement, we compare to existing interactive proofs in ReLoC, showing an average proof size reduction by a factor 7 (§3.5.4).
- All of our results have been implemented and verified using the Coq proof assistant. The Coq sources can be found in Mulder and Krebbers (2023a).

We conclude this chapter with related work (§3.6) and future work (§3.7).

```
Definition fg_incrementer : val :=
  λ: <>,
    let: "l" := ref #1 in
    (rec: "f" <> :=
      let: "n" := ! "l" in
      if: CAS "l" "n" ("n" + #1) then
        "n"
      else
        "f" #()).

Definition cg_incrementer : val :=
  λ: <>,
    let: "l" := ref #1 in
    let: "lk" := newlock #() in
    (λ: <>,
      acquire "lk";;
      let: "n" := ! "l" in
      "l" ← "n" + #1 ;;
      release "lk";;
      "n").

Lemma fg_cg_incrementer_refinement :
  ⊢ REL fg_incrementer << cg_incrementer : () → () → lrel_int.
Proof.
  iStepsS.
  iAssert (|={⊤}=> inv (nroot.@"incr")
      (∃ (n : nat), x ↦ #n ∗ x0 ↦ₛ #n ∗ is_locked_r x1 false))%I
      with "[-]" as ">#Hinv"; first iStepsS.
  iSmash.
Qed.
```

Figure 3.2: Verification of a refinement for a fine-grained concurrent incrementer in Diaframe 2.0. Some variable names in the proof are (unfortunately) autogenerated: `x` stands for the location of the fine-grained incrementer, `x0` for the location of the coarse-grained incrementer, and `x1` for the lock of the coarse-grained incrementer.

3.2 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 (§ 3.2.1), providing intuition for ReLoC. After providing some formal background for ReLoC's proof rules (§ 3.2.2), we describe our proof automation strategy (§ 3.2.3).

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 Figure 3.2, 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 `fg_incrementer` $\preceq_{\text{ctx}}$ `cg_incrementer` : $() \to () \to \mathbb{Z}$ 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.

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 chapter 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 $\ell_l \mapsto z * \ell_r \mapsto_{\mathsf{s}} z \vdash \models !\,\ell_l \precsim !\,\ell_r : \mathbb{Z}$, *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 $\mapsto_{\mathsf{s}}$ (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 Figure 3.2. Our top-level goal (line 21) is the following logical refinement of closures:

$$\vdash \models \texttt{fg\_incrementer} \precsim \texttt{cg\_incrementer} : () \to () \to \mathbb{Z}. \tag{3.1}$$

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 Figure 3.2, 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 $\boxed{R}^{\mathcal{N}}$, and `is_locked_r` for isLock.) 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.

3.2.2 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 Figure 3.3. 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 $\vdash \models \texttt{fg\_incrementer} \precsim \texttt{cg\_incrementer} : () \to () \to \mathbb{Z}$, 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 $\vdash \Box (\models v_1\, () \precsim v_2\, () : A)$. 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:

$$\ell_l \mapsto 1, \ell_r \mapsto_{\mathsf{s}} 1, \mathsf{isLock}(v, \mathsf{false}) \vdash \models (\mathrm{rec}\ldots) \precsim (\lambda\ldots) : () \to \mathbb{Z} \tag{3.2}$$

We obtain two maps-to connectives $\ell_l \mapsto 1$ and $\ell_r \mapsto_{\mathsf{s}} 1$ 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. (This differs from the usual way locks in concurrent separation logic are specified (Dinsdale-Young et al., 2010; Gotsman et al., 2007; Hobor et al., 2008) due to the angelic nature of right-hand side execution in contextual refinements.) In our proof obligation Equation (3.2), 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 $\ell_r \mapsto_{\mathsf{s}} 1$, 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 (3.2). To continue, we will first allocate an invariant using INV-ALLOC. We take $R \triangleq \exists n.\, \ell_l \mapsto n * \ell_r \mapsto_{\mathsf{s}} n * \mathsf{isLock}(v, \mathsf{false})$, 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:

$$\boxed{\exists n.\, \ell_l \mapsto n * \ell_r \mapsto_{\mathsf{s}} n * \mathsf{isLock}(v, \mathsf{false})}^{\mathcal{N}} \vdash \models (\mathrm{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbb{Z}. \tag{3.3}$$

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 off 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 can apply the induction hypothesis and finish the proof.

A selection of Iris's rules for the $\triangleright$ and $\square$ modality and Löb induction are shown in Figure 3.3b. 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 $\square$ 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 (3.3). After LÖB and UNFOLD-REC-L, our goal is:

$$\left(\begin{array}{l}\boxed{\exists n.\, \ell_l \mapsto n * \ell_r \mapsto_{\mathsf{s}} n * \mathsf{isLock}(v, \mathsf{false})}^{\mathcal{N}},\\ \square\,(\models (\mathrm{rec}\ldots)\ () \lesssim (\lambda\ldots)\ () : \mathbb{Z})\end{array}\right) \vdash \models (\mathrm{let}\, n := !\ell\ \ldots) \lesssim \ldots : \mathbb{Z} \tag{3.4}$$

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

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
$$\frac{\Delta \vdash \triangleright R * (\boxed{R}^{\mathcal{N}} \twoheadrightarrow \models e_1 \lesssim e_2 : A)}{\Delta \vdash \models e_1 \lesssim e_2 : A}$$

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

LÖB
$$\frac{\Delta, \triangleright \Box(\Delta \twoheadrightarrow G) \vdash G}{\Delta \vdash G}$$

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

(b) Proof rules relevant for Proof Phase 3.

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

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

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

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

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

ACQUIRE-R
$$\frac{\Delta, \mathsf{isLock}(v, \mathsf{true}) \vdash \models_{\mathcal{E}} e \lesssim K[()] : A}{\Delta, \mathsf{isLock}(v, \mathsf{false}) \vdash \models_{\mathcal{E}} e \lesssim K[\mathtt{acquire}\, v] : A}$$

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

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

RELEASE-R
$$\frac{\Delta, \mathsf{isLock}(v, \mathsf{false}) \vdash \models_{\mathcal{E}} e \lesssim K[()] : A}{\Delta, \mathsf{isLock}(v, \mathsf{true}) \vdash \models_{\mathcal{E}} e \lesssim K[\mathtt{release}\, v] : A}$$

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

(c) Proof rules relevant for Proof Phase 4.

Figure 3.3: A selection of the existing rules of Iris and ReLoC.

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 Figure 3.3c.

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 $\models_{\mathcal{E}} e_1 \precsim e_2 : A$ have a mask parameter. As shown by REFINES-FUPD, this represents the initial mask of a fancy update. We let $\mathcal{E} = \top$ when the mask is omitted.

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:

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

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 ($\triangleright R \mathrel{-\!\!*} {}^{\top\setminus\mathcal{N}}\!\Rightarrow^{\top} \mathsf{True}$). 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 $\ell_l \mapsto n'$ back, 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:

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

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. We therefore use ACQUIRE-R, LOAD-R, STORE-R and RELEASE-R to symbolically execute the right-hand side. We conclude the proof of this case by closing the invariant with REFINES-FUPD and FUPD-ELIM, 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:

$$\boxed{R}^{\mathcal{N}}, \Box(\models (\mathrm{rec}\ldots)\ () \precsim (\lambda\ldots)\ () : \mathbb{Z}) \vdash \models (\mathrm{rec}\ldots)\ () \precsim (\lambda\ldots)\ () : \mathbb{Z}$$

3.2.3 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.
- Proof Phase 2: *Not* introducing the $\Box$ 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:

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

(L are 'easy' goals like $\ell \mapsto v$, described in §3.4.5). 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 Chapter 2 to handle invariants, which operates on precisely these goals.

Our proof strategy is the result of restating the original rules of ReLoC (Figure 3.3) so that they can be applied systematically. Our new rules can be found in Figure 3.4. 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,

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

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

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

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

Figure 3.4: Derived proof rules for ReLoC suitable for proof automation.

where $\forall \vec{x}.\, \{L\}\, e_1\, \{v.\, U\}$ is a Hoare triple for e_1. In Coq, $\forall \vec{x}.\, \{L\}\, e_1\, \{v.\, U\}$ 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 $\forall \vec{x}.\, \{L\}\, e_2\, \{v.\, U\}_{\mathsf{s}}$ 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 like VAL-Z and VAL-FUN, depending on 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, then start symbolic execution of the function with 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 to make progress.

Verification of the example in Figure 3.2. The strategy above is available using the `iStepsS` tactic in Coq. In the verification of the example in Figure 3.2, 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 §3.1. 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 Figure 3.2 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.

```
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.
```

Figure 3.5: Verification of a logically atomic triple for a fine-grained concurrent incrementer in Diaframe 2.0.

3.3 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 (§3.3.1). After discussing the formal proof rules in Iris (§3.3.2), we show our strategy for proof automation of these triples (§3.3.3).

3.3.1 Logical Atomicity in Iris

Consider the `inc` function defined in lines 1-7 of Figure 3.5. The pattern of recursively trying to `CAS` occurs in various concurrent programs: we have seen it in `fg_incrementer` in §3.2, 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 $\{\ell \mapsto z\}$ `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:

HOARE-LOAD

$$\frac{}{\{\ell \mapsto v\}\ !\ell\ \{w.\ulcorner w = v \urcorner * \ell \mapsto v\}_{\mathcal{E}}}$$

HOARE-INV-ACCESS

$$\frac{\text{atomic } e \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q\}_{\mathcal{E}\setminus\mathcal{N}}}{\left\{\boxed{R}^{\mathcal{N}} * P\right\}\ e\ \{v.\, Q\}_{\mathcal{E}}}$$

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 $(\texttt{inc}\,\ell)$', this rule is not applicable. But although `inc` is not *physically* atomic, the effects of `inc` appear to take place atomically for clients. That is, at a certain point during the execution of `inc`, namely, when the `CAS` succeeds, $\ell \mapsto z$ is atomically consumed to produce $\ell \mapsto (z+1)$. This gives us a characterization of linearizability: an operation is linearizable if it appears to take place atomically/instantly somewhere during its execution, and the precise moment when this happens place is called the *linearization point*. Inspired by the TaDA logic (da Rocha Pinto et al., 2014), Iris features a special kind of Hoare triple to specify this, called a logically atomic triple (Jung et al., 2015; Jung, 2019; Jung et al., 2020). We specify the behavior of `inc` using the logically atomic triple:

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

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:

LA-INV
$$\frac{\boxed{R}^{\mathcal{N}} \qquad \langle \vec{x}.\, \alpha * \triangleright R \rangle\ e\ \langle v.\, \beta * \triangleright R \rangle_{\mathcal{E} \setminus \mathcal{N}}}{\langle \vec{x}.\, \alpha \rangle\ e\ \langle v.\, \beta \rangle_{\mathcal{E}}}$$

LA-HOARE
$$\frac{\square\, \langle \vec{x}.\, \alpha \rangle\ e\ \langle v.\, \beta \rangle_{\mathcal{E}}}{\forall \vec{x}.\, \{\alpha\}\ e\ \{v.\, \beta\}_{\top}}$$

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 binder z in INC-LOGATOM is somewhat curious, being scoped over both the pre- and postcondition of the triple, but not over the expression. 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 $\vec{x}$.

3.3.2 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*:

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

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]

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

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 follows:

AU-ACCESS-IRIS
$$\langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \exists \vec{x}.\, \alpha * \Big(\big(\alpha \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}}\big) \wedge \big(\forall v.\, \beta \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} Q\big) \Big)$$

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 Figure 3.5 needs to account for the recursive call when the `CAS` fails. We will use Löb induction once more—Figure 3.6 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.

[1] The definition of logically atomic triples does not feature the $\Box$ 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.

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

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

Figure 3.6: Selection of Iris's proof rules for Löb induction on weakest preconditions.

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.

3.3.3 Proof Automation Strategy

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

- Weakest preconditions: $\Delta \vdash \mathsf{wp}\, e\, \{\Phi\}$. The definition of logically atomic triples LA-DEF features regular weakest preconditions as the goal.
- Atomic updates: $\Delta \vdash \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow Q \rangle_{\mathcal{E}}$. After applying a known logically atomic triple with SYM-EX-LOGATOM, the goal becomes an atomic update.
- Goals of the form $\Delta \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\, L * G$. These goals occur after introducing atomic updates with AU-INTRO or when establishing 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 by internalizing Löb induction, and by adding support for logically atomic triples.

Suppose our goal is $\Delta \vdash \mathtt{wp}\ e\ \{\Phi\}$. 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'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 $\Delta \vdash \left\langle \vec{x}.\ L \mid v.\ U \Rrightarrow \mathtt{wp}\ K[v]\ \{\Phi\}\right\rangle_{\top\setminus\mathcal{E}}$.
   (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 $\Delta \vdash \triangleright \mathtt{wp}\ b[v/x]\ \{\Phi\}$.
   (b) For recursive functions. Apply LÖB, then apply UNFOLD-REC. Continue with new goal $\Delta, \triangleright \square(\Delta \mathrel{-\!\!*} \mathtt{wp}\ e\ \{\Phi\}) \vdash \triangleright \mathtt{wp}\ b[e/f][v'/x]\ \{\Phi\}$.[2] Note that Δ may contain an atomic update, which will thus be needed to apply the induction hypothesis for recursive calls.

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

4. $G = \triangleright G'$. Apply the $\triangleright$-INTRO rule, which introduces the later from the goal and strips later modalities from hypotheses in 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 $\Delta \vdash {}^{\mathcal{E}}\!\Rrightarrow^{?\mathcal{E}'} \exists \vec{x}.\ L * \left(\left(L \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Delta\right) \wedge \left(\forall v.\ U \mathrel{-\!\!*} {}^{?\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} G\right)\right)$.
6. $G = {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \exists \vec{x}.\ L * G$. Use proof automation from Diaframe to make progress. If enabled and when relevant, Diaframe will backtrack to determine the linearization point.

This strategy can prove the logically atomic triple in Figure 3.5 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.

[2] In the Coq implementation we additionally generalize the Löb induction hypothesis over the arguments supplied to e.

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 automation needs to obtain ownership of α.

This can be done by allowing Diaframe 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 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, and achieve this by replacing the conjunction with a disjunction on the left-hand side of a wand:[3]

AU-ACCESS-DIAFRAME

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

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 Figure 3.5. 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 will access the atomic update with 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 $(\forall mv.\, \_ \vee \_ \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \_)$. Diaframe notices it can use this closing resource to restore the mask, so the goal becomes (note that mv is bound in the remaining proof obligation G):

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

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 (3-b) handles the situation in which the function is recursive and generates a Löb induction hypothesis. Item (3-a) is a specialized version that handles the case where there is no actual recursion. Omitting this specialized version would work, but would cause Item (3-b) to generate useless induction hypotheses that in turn increase the search space in Item (2-c), and thus slow down the automation. Omitting Item (3-a) 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

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

in Figure 3.5, and show a number of other simple examples in § 3.5. 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.

3.4 Implementation as Extensible Proof Strategy

In § 3.2 and 3.3 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 (§3.2), a weakest precondition, or an atomic update (§3.3). As we will see in §3.4.5, rules of these strategies cannot be represented by the automation of Diaframe. 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. 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 § 3.2 and 3.3 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 (§3.4.5). 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* (§ 3.4.1 and 3.4.2),

while rules of Category 3 and 4 fit into our *transformer hints* (§3.4.3). A combination of abduction hints and transformer hints (§3.4.4) can be used to implement composite procedures such as Löb induction.

3.4.1 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 from Chapter 2, 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.

3.4.2 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 §3.3.3 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.

3.4.3 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:

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

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

1. Find $H \in \Delta$ and $\mathcal{T}'$ such that $H, \mathcal{T} \overrightarrow{\sim}_{\mathsf{hyp}}\ [\mathcal{T}']$. 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.

[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 §3.4.2.

2. Otherwise, find G such that $\Delta, \mathcal{T} \overset{\rightarrow}{\sim}_{\mathsf{ctx}} [G]$. 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.

3.4.4 Transformer Hints for Other Rules

In §3.4.3, 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 §3.3.3:

- If $G = \langle \vec{x}.\, L \mid v.\, U \Rrightarrow \Phi \rangle_{\mathcal{E}}$, and G does not occur in our environment Δ. We introduce the atomic update with rule 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 $G * [\mathsf{True}] \Vdash G$, instead of applying AU-INTRO. We therefore add an intermediate form $\mathbf{AU}_{\mathsf{pre}}(\vec{x}, \alpha, v, \beta, \Phi, \mathcal{E}) \triangleq \langle \vec{x}.\, \alpha \mid v.\, \beta \Rrightarrow \Phi \rangle_{\mathcal{E}}$ and a combination of transformer and abduction hints:

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

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

Since AU-INTRO-PRE is a last-resort hint (indicated by ε_1), we ensure that the assumption hint $G * [\mathsf{True}] \Vdash G$ is preferred. After applying AU-INTRO-PRE, the proof search strategy tries to establish $\mathbf{AU}_{\mathsf{pre}}$. 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 $\mathbf{AU}_{\mathsf{pre}}$ 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 $\mathbf{löb}_{\mathsf{pre}}\,(G) \triangleq G$ and $\mathbf{löb}_{\mathsf{post}}\,(G) \triangleq G$, and the following hints:

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

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

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

By delegating Löb induction to the $\mathbf{löb}_{\mathsf{pre}}$ and $\mathbf{löb}_{\mathsf{post}}$ 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 artifact (Mulder and Krebbers, 2023a). The full-blown version generalizes over variables and thus has a more sophisticated version of the second hint.

3.4.5 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, '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 from Chapter 2 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 $H, \mathcal{T} \overrightarrow{\sim}_{\mathsf{hyp}} [\mathcal{T}']$. Continue with goal $\Delta \setminus H \vdash \mathcal{T}'$.
   (b) Otherwise, find G' such that $\Delta, \mathcal{T} \overrightarrow{\sim}_{\mathsf{ctx}} [G']$. Continue with goal $\Delta \vdash G'$.

Diaframe vs Diaframe 2.0. There are two main reasons why Diaframe's bi-abduction hints cannot express the proof search strategies from § 3.2.3 and 3.3.3. Firstly, context transformer hints (Item (5-b)) have a shape that is simply incompatible with Item (4-b). 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 (4-b), causing the proof search to get stuck.

Table 3.1: Data on examples with logical atomicity, in comparison with Voila. Rows correspond to files in the supplementary artifact (Mulder and Krebbers, 2023a). 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. Verification time includes both running the proof search strategy and checking the proof in Coq.

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

3.5 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 (§3.5.1). 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) (§3.5.2). Besides reverifying existing examples, we use our results to verify logical atomicity of the Michael-Scott queue (Michael and Scott, 1996) (§3.5.3). This queue is known to be linearizable—Vindum and Birkedal (2021) proved contextual refinement with ReLoC, and we take inspiration from their approach—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 (§3.5.4).

3.5.1 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 Table 3.1. 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 et al., 2014; da Rocha Pinto, 2016), 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 Table 3.1. On average, Diaframe 2.0 has ca. 0.4 lines of proof burden per line of implementation (63 lines of proof burden on 156 lines of implementation), while Voila has, in our count, 1.7 lines of proof burden per line of implementation.[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).

3.5.2 Comparison to Complex Interactive Proofs of Logical Atomicity 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 Table 3.2. 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

[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).

Table 3.2: Data on examples with logical atomicity, in comparison with Iris Proof Mode (IPM) proofs.

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		

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. 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.

3.5.3 Experiences Verifying Logical Atomicity of a Complex Data Structure: 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, 2023a)). Our proof reuses some of their techniques (the persistent points-to predicate), but represents the queue data structure invariant somewhat differently, thereby making it more amenable to automation. 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.

The queue data structure invariant we use differs from the invariant used by Vindum and Birkedal (2021) mainly in the way we represent the linked list of queued nodes. In this list of nodes, only the next-node pointer of the last node is mutable. For our invariant, we define a resource that contains all these next-node pointers, including the last one, while Vindum and Birkedal (2021) have a separate resource for the last node. We add appropriate hints to extract from our linked-list resource the points-to resources for arbitrary nodes in the list. This approach is more amenable for automation, since obtaining the points-to resource for a next-node pointer is always done in the same way: by using our linked list resource. The proof by Vindum and Birkedal (2021) instead needs to differentiate between last and non-last nodes to obtain this resource.

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, by letting the user specify from which hypothesis a hint should be found. This is described in more detail in the artifact's `README` file (Mulder and Krebbers, 2023a), under `solveStep with`.

3.5.4 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.[6] Statistics on the examples can be found in Table 3.3. The proof of ticket lock $\precsim$ spin lock differs slightly from the original proof: instead of relying on ReLoC's *logically atomic relational specifications* (Frumin et al., 2018), we use Iris's regular logically atomic specifications (§3.3) for the same effect.[7]

We summarize some aggregated data from Table 3.3. 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 coarse-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 §3.5.2. Additionally, it may be necessary to manually establish an invariant like in §3.2, or to manually perform right-hand side execution. A tactic `iStepR` is available for this last case.

3.6 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 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 §3.5.1. In summary, we show an average proof size reduction by a factor 4, and

[6] We think it would be difficult to adequately extend Diaframe to support helping in ReLoC. Helping in ReLoC is supported by *splitting* a refinement proof obligation (Vindum et al., 2022) into a resource for the right-hand side, and a proof obligation for the left-hand side. One completes the refinement proof by symbolically executing the right-hand side *resource* at the appropriate moment. Since such symbolic execution is not goal-directed, it is harder to automate with Diaframe.

[7] 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.

Table 3.3: 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.

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

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 §3.5.1 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 (Vafeiadis, 2010; Henzinger et al., 2013), 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., 2014b). 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.

Plankton and Nekton. Plankton (Meyer et al., 2022, 2023b) is another recent non-foundational tool for verifying linearizability of concurrent search structures, building on techniques similar to Krishna et al. (2020, 2021). Plankton uses a custom separation logic with proof automation in mind, with 'temporal interpolation' as a key innovation for verifying future-dependent linearization points. This approach sacrifices compositional client reasoning, but achieves an astounding degree of automation: it just needs a local 'node' invariant to prove that a program is linearizable.

Nekton (Meyer et al., 2023a) repurposes some of Plankton's innovations to check proof outlines of linearizability. By sacrificing some automation (the user has to provide proof outlines), Nekton gains generality: instead of just targeting concurrent search structures, it targets all programs expressible in the flow framework (Krishna et al., 2018). Nekton has been used to verify linearizability of the state-of-the-art FEMRS tree (Feldman et al., 2018).

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 (Jacobs and Piessens, 2011; Svendsen et al., 2013; da Rocha Pinto et al., 2014; Dreyer et al., 2010; 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 (Sergey et al., 2015; Nanevski et al., 2019) 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).

3.7 Future Work

We would like to improve the usability of Diaframe 2.0. As can be seen in Figure 3.2, 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 in Coq could be incorporated (Ekici et al., 2017; Besson, 2021; Czajka, 2020).

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 and Du (2024) have 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 (§ 3.5), our proof automation cannot always automatically determine the required case distinctions for a proof. Additionally, we rely on backtracking to determine linearization points. We will describe an extension of Diaframe with better support for disjunctions in Chapter 4.

Chapter 4

Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic

4.1 Introduction

Separation logic (O'Hearn et al., 2001; Reynolds, 2002) and its successor concurrent separation logic (O'Hearn, 2007; Brookes, 2007) have shown to be invaluable to modularly verify increasingly complicated programs that involve pointers and shared-memory concurrency. Already from the early days of separation logic, researchers have investigated how separation logic could be used for automated verification. The seminal work by Berdine et al. (2005, 2006) on symbolic execution in separation logic has been instrumental in the development of (mostly)-automated tools for proving functional correctness (Jacobs et al., 2011; Piskac et al., 2014b; Müller et al., 2016; Oortwijn et al., 2020), including foundational tools that are embedded and proved sound in general-purpose proof assistants (Chlipala, 2011; Sammler et al., 2021).

A particular subfield concerns the automated verification of fine-grained concurrent programs (Calcagno et al., 2007; Dinsdale-Young et al., 2017; Windsor et al., 2017). Such programs use low-level atomic operations (such as compare-and-swap) instead of high-level concurrency primitives (such as locks). The verification of these programs is particularly challenging because one needs to define an *invariant*, which describes how ownership is shared and transferred between threads, and prove this invariant holds after *every step* the program takes. The invariant usually consists of a disjunction of the logical states the program (or data structure) can be in, together with some form of ghost state (or a protocol) to relate each disjunct to the state of the threads (Dinsdale-Young et al., 2010). Depending on the shape of the invariant, state-of-the-art tools can automatically verify the program. In the ideal case, logical states correspond one-to-one to the physical states of the program (*i.e.,* the data on the heap). For example, to verify a spin lock, the invariant has two disjuncts that correspond to whether the lock (represented as a Boolean on the heap) is in the locked or unlocked state.

The more challenging case arises if the logical states do not correspond exactly to the physical states of the program. An example is an atomic reference counter (ARC), such as the one in Rust Language (2021), that keeps track of the number of readers of a shared resource. In the verification there are two logical states/disjuncts—either there exists a positive number of readers matching up with the integer value of the ARC, or there are no readers left and the ARC has been deallocated. In the last disjunct, there is no corresponding physical state (*i.e.,* points-to assertion) since the ARC has been deallocated. State-of-the-art automated tools need user guidance to verify some of ARC's methods w.r.t. this challenging invariant. In Starling (Windsor et al., 2017) one needs to add an annotation for each program statement. In Caper (Dinsdale-Young et al., 2017) one needs to insert a no-op `assert (cnt = 1 ? true : true)` into the code of one method to force the automation to perform a case split. In Diaframe (Chapter 2) one needs to fall back to an interactive proof in this same method, to perform a case split similar to the one in Caper.

Dealing with disjunctions $P \lor Q$ in separation logic is challenging because the disjuncts P and Q can be arbitrary formulas of separation logic. That is, they can contain $*$ and $\mapsto$: logical operators that are not part of the native logic of automated provers based on classical logic (*e.g.,* SMT). Logics such as Iris go beyond that by allowing P and Q to also contain higher-order quantification, invariants and modalities.[1] The aforementioned tools for automated verification in fine-grained concurrent separation logic deal with disjunctions (*i.e.,* the logical states of the invariant) roughly as follows. Before each program instruction, they make a case distinction on the logical states of the invariant. Some of these cases can discharged because they lead to a contradiction with other (ghost) resources that are owned by the thread. After symbolic execution of the program instruction, the invariant needs to be reestablished, which is done by *backtracking* on all logical states of the invariant. That is, the automation picks one of the states of the invariant, then attempts to finish the proof. If this fails, another state is tried until the proof succeeds. This approach works surprisingly well for the ideal cases described above (*e.g.,* the spin lock), but is insufficient for the more challenging ones (*e.g.,* the ARC). There are three problems:

- **Debugging/cooperation with interactive proofs.** It is difficult to determine why a proof by backtracking failed (bug in the specification, bug in the program, or the problem being too difficult for the automation) without investigating all choices the backtracking algorithm considered. Similarly, if a backtracking proof fails, there is no canonical sub-goal where the user could continue with an interactive proof.
- **Uninformed disjunction introduction.** When proving $\Delta \vdash P \lor Q$, it is often the case that neither $\Delta \vdash P$ nor $\Delta \vdash Q$ is true. Merely backtracking on the left/right introduction of $P \lor Q$ will thus never find a proof. Instead, one needs to perform some case analysis first. For example, in our ARC example, one needs to make a distinction on whether the reference counter had value 1 to decide if P or Q should be introduced.

[1] An additional challenge is that the standard way SMT-like solvers may approach disjunctions is by assuming the negation of all cases, and showing a contradiction: to establish $\vdash P \lor Q$, they will try to prove $\neg P, \neg Q \vdash \bot$. This approach is incompatible with intuitionistic/non-classical logics where the Law of Excluded Middle (LEM) does not hold. Cao et al. (2017) show that LEM does not hold in advanced separation logics such as Iris.

- **Overeager disjunction elimination.** To verify more complicated programs one often uses multiple invariants simultaneously. For example, to verify an MCS lock (Mellor-Crummey and Scott, 1991) or CLH lock (Magnusson et al., 1994), one typically uses a separate invariant for each location/node. This may cause up to 4 such invariants to be in scope, and it would be inefficient to consider all states/disjuncts (for each program instruction, there will be an exponential number of cases). Caper uses backtracking to try to perform case analysis on the least number of invariants, which works well for successful verification attempts, but leads to very slow times for failing verification attempts Wolf et al. (2021).

The last two problems are well-known in the context of proof search in intuitionistic logic. For instance, to show $P \lor Q \vdash Q \lor P$, one first needs to eliminate the disjunction on the left, and only afterwards, a disjunct on the right can be chosen. In this example, the disjunction on the left could be eagerly eliminated by a proof search procedure, but in general, this is impossible because the disjunction could be the conclusion of a hypothesis $\forall \vec{x}.\, R_1 \to \ldots \to R_n \to P \lor Q$. After all, there are infinitely many ways to instantiate $\vec{x}$. Indeed, procedures for proof search in intuitionistic logic available in proof assistants are either restricted to the propositional fragment (such as the `tauto` tactic in Coq), perform blind quantifier instantiation (such as the `firstorder` tactic in Coq) or are already incomplete for trivial goals (such as the `eauto` tactic in Coq). An efficient and complete solver for first-order intuitionistic logic is the ileanCop automated theorem prover (Otten, 2008), which is based on connection calculi (Waaler, 2001; Wallen, 1990; Otten and Kreitz, 1995). Intuitively, the idea behind these calculi is to establish a *connection* between parts of a hypothesis and the goal. This connection makes sure that the correct disjunction (on the left) is eliminated, and the correct disjunct (on the right) is introduced, thereby addressing the two problems we sketched.

The completeness of solvers based on connection calculus allows them to take an all-or-nothing approach: either the proof is finished, the goal is declared unprovable, or proof search does not terminate. Such an approach becomes untenable in more bespoke logics such as concurrent separation logic. However, useful proof automation for these logics does exist (Chlipala, 2011; Sammler et al., 2021), and usually relies on a tailor-made, *backward-chaining* (Prolog-style) proof search strategy. Such strategies can be applied in a step-wise fashion, where each step replaces an entailment by one or multiple simpler entailments. To do so, these strategies need a way to detect what hypotheses are relevant to what (part of the) goal. This is what we are looking for: a set of rules to determine appropriate disjunctions to eliminate and disjuncts to introduce.

To design such a set of rules, we take inspiration from connection calculus, which results in a simple calculus with connections for separation logics. Our approach is not tied to a specific model of separation logic (*e.g.,* Iris), we support any bunched implications (BI) logic (O'Hearn and Pym, 1999; Pym, 2002). We implement and evaluate the practicality of our calculus in the Diaframe automation tool (Chapter 2) for the Iris framework of higher-order concurrent separation logic in Coq (Jung et al., 2015, 2016; Krebbers et al., 2017b; Jung et al., 2018b; Krebbers et al., 2017a, 2018). Our simple calculus does not yield a complete proof search procedure, but our evaluation suggests that it significantly improves on the state-of-the-art on automated verification of practical examples in fine-grained concurrent separation logic—also for examples with coarse-grained concurrency. The key strength of our calculus is that it can be extended with many features outside

of first-order logic, such as Iris's modalities, higher-order quantification, ghost state mechanisms, and invariant assertions. For this purpose, theoretical completeness is not essential and simplicity is preferable.

We base our calculus on a *focused* version of intuitionistic *multi-succedent* calculus by Dragalin (1988). Focusing (Andreoli, 1992; Simmons, 2014; Liang and Miller, 2009) addresses the issue of rule non-permutability by dividing the proof search into two alternating phases: the inversion phase in which invertible rules are eagerly applied, and the focusing phase which groups sequences of non-invertible rules. This approach limits backtracking to the choice of a connection in the focusing phase, thereby significantly reducing the search space, and in turn increasing the efficiency and predictability of the proof automation. The multi-succedent aspect of our calculus allows it to delay choosing which disjunct to introduce.

A key feature of concurrent separation logics such as Iris is their support for ghost state, which is used to relate the logical state of the invariant to the state of the individual threads. Crucially, after each program instruction, the ghost state needs to be updated to match up with the effect of the instruction. There are often multiple ways in which the ghost state can be updated. This is a disjunctive pattern that influences the connections we should consider. Since Iris provides various forms of ghost theories, we do not want to hard-wire their rules into our calculus. We thus make our calculus parametric in *ground connections* to describe "domain-specific" rules for the atoms. This mechanism is suitable to, but not limited to, describe the rules of Iris's ghost theories.

Outline and contributions. Our contributions are as follows:

- We present our approach on a minimal calculus for regular (non-separating) propositional logic (§ 4.3). This calculus is based on a focused version of intuitionistic multi-succedent calculus by Dragalin (1988), incorporating ideas from focusing and connection calculi.
- We extend our approach to propositional separation logic (§ 4.4). To deal with the substructural aspects of separation logic, we combine our calculus with the goal-directed approach to handle separating conjunctions from RefinedC (Sammler et al., 2021).
- We extend our approach to the higher-order concurrent separation logic Iris by integrating it into the Diaframe proof automation tool (§ 4.5). To make our calculus parametric in domain-specific theories (such as those for ghost state), we combine our notion of *ground connections* with Diaframe's notion of *bi-abduction* hints.
- We implement our approach in the proof assistant Coq (Mulder et al., 2023a) (§ 4.6). The implementation contains machine-checked soundness proofs of both the minimal version for propositional logic and the full-blown version. The full-blown version provides various tactics that can execute the procedure automatically.
- We evaluate our implementation on 24 examples (§ 4.7). We can verify 14/24 examples fully automatically, and reduce the overall proof burden by 33% compared to the original Diaframe. We also compare to the SMT-based verification tool Caper (Dinsdale-Young et al., 2017), which supports 15/24 examples, and can do 13/24 examples fully automatically.

We start by describing the problem in the context of two versions of the aforementioned ARC example (§4.2), and conclude with a description of related work (§4.8).

4.2 Motivating Examples

We show challenging uses of disjunctions that occur when verifying an Atomic Reference Counter (ARC) inspired by Rust Language (2021). ARCs are employed to safely give multiple threads read access to a resource, and to recover write access when all threads are done. The ARC we consider accomplishes this by keeping a thread-safe tally of the number of active readers. ARCs usually come with at least three methods: `mk_arc` creates a new ARC for (initially) a single reader, `clone` registers an additional reader thereby 'duplicating' read access, while `drop` deregisters a reader. The `drop` method also returns whether the number of active readers is now zero, and gives back write access if that is the case. Precisely this case distinction is problematic for automatic verification.

We verify two ARCs: a version that leaves deallocation to a garbage collector (§4.2.1), and a more challenging version that performs an explicit deallocation on the last `drop` (§4.2.2). In prior work (Dinsdale-Young et al. (2017) and Chapter 2), verification of the `drop` method requires user guidance, but our approach can verify it fully automatically. For both versions of ARC, we show the precise subgoal in the `drop` verification at which prior work used to get stuck.

4.2.1 ARC Without Deallocation

An implementation of an ARC in Iris's default programming language HeapLang (Jung et al., 2016) is given in lines 2–9 of Figure 4.1. We represent an ARC using a location that counts the number of active readers. To create a new ARC, we return a new location that points to 1, indicating there is a single reader. To `clone` the ARC, we use fetch-and-add (`FAA`) to atomically increment the reference count by 1. Dually, to `drop` an ARC, we use `FAA` to atomically decrement the reference count by 1. The `FAA` method will return the old value of the location. We are the last reader precisely when the old value was 1, hence `drop` returns the result of this comparison. Note that `drop`ping the last reader does not deallocate the location: we rely on garbage collection to do so.

The specification of this ARC is given in lines 15–26 of Figure 4.1. Each method is specified using a Hoare-style triple $\text{SPEC}\ \{L\}\ e\ \{\vec{y},\ \text{RET}\ v;\ U\}$. Such a triple indicates that if one owns resources satisfying L, evaluating e is safe, and if e terminates, there exist instances of the logical variables $\vec{y}$ so that the return value is precisely v, and one owns resources satisfying U. (Both v and U may mention $\vec{y}$.) The specifications in Figure 4.1 mention resources $\texttt{is\_arc}\ \gamma\ v$, representing the knowledge that value v is an ARC with *ghost name* γ, and resources $\texttt{token}\ P\ \gamma$, representing read access to shareable assertion P, governed by an ARC with name γ. The ghost name γ is used to tie the $\texttt{token}\ P\ \gamma$ to a specific ARC. The definition of `is_arc` will be discussed shortly.

The entire specification of this ARC is parameterized by the shareable assertion P, as can be seen in line 1. Shareable assertions are represented as fractional permissions (Boyland, 2003). In Iris these are *well-behaved* predicates $P : \mathbb{Q}_p \to \mathit{iProp}$. Here *iProp* is the type of Iris assertions, and $\mathbb{Q}_p \triangleq \{q \in \mathbb{Q} \mid q > 0\}$. We call P well-behaved (`Fractional`

```
Context (P : Qp → iProp) {HP : Fractional P}.
Definition mk_arc : val :=
  λ: <>, ref #1.
Definition clone : val :=
  λ: "a", FAA "a" #1;; #().
Definition drop : val :=
  λ: "a",
    let: "old_val" := FAA "a" #(-1) in
    ("old_val" = #1).
Definition arc_inv (γ : gname) (l : loc) : iProp :=
  ∃ (n : nat), l ↦ #n ∗ ((⌜n = 0⌝ ∗ no_tokens P γ)
                      ∨ (⌜0 < n⌝ ∗ token_counter P γ (Pos.of_nat n))).
Definition is_arc (γ : gname) (v : val) : iProp :=
  ∃ (l : loc), ⌜v = #l⌝ ∗ inv N (arc_inv γ l).
Program Instance mk_arc_spec :
  SPEC {{ P 1 }}
    mk_arc #()
  {{ (v : val) (γ : gname), RET v; is_arc γ v ∗ token P γ }}.
Program Instance clone_arc_spec (γ : gname) (v : val) :
  SPEC {{ token P γ ∗ is_arc γ v }}
    clone v
  {{ RET #(); token P γ ∗ token P γ }}.
Program Instance drop_arc_spec (γ : gname) (v : val) :
  SPEC {{ token P γ ∗ is_arc γ v }}
    drop v
  {{ (b : bool), RET #b; (⌜b = true⌝ ∗ P 1) ∨ ⌜b = false⌝ }}.
```

Figure 4.1: Automatic verification of an ARC without deallocation

in Coq) if $P\ q_1 * P\ q_2 \dashv\vdash P\ (q_1 + q_2)$ for all q_1 and q_2. The resource $P\ 1$ denotes write access, while $P\ q$ with $0 < q < 1$ denotes read access. Read access $P\ q$ can be obtained from `token` $P\ \gamma$ resources (see TOKEN-ACCESS in Figure 4.2).

Let us now explain the specifications. The specification for `mk_arc` requires one to give up write access $P\ 1$, after which the function returns a ghost name γ and value v for which we learn `is_arc` $\gamma\ v$, and additionally obtain a read access `token` $P\ \gamma$. Tokens can be duplicated with `clone`: it requires one `token` $P\ \gamma$, but returns two. Finally, tokens can be destroyed with `drop`, which returns a Boolean. Only if this Boolean is true (in case we are the last reader) do we recover write access $P\ 1$.

Both `clone` and `drop` can be called concurrently on a given ARC, meaning that multiple threads can mutate the location storing the reference count. In separation logic, the *maps-to* resource $\ell \mapsto v$ represents the right to mutate a location ℓ. This is an exclusive resource: only one thread can hold it. However, in the case of ARC, we want to share this resource between multiple threads. To do so, we employ Iris's invariants mechanism. Iris's invariant assertion $\boxed{L}$ represents the knowledge that resources satisfying L hold invariantly. Invariant assertions are duplicable (*i.e.*, $\boxed{L} \dashv\vdash \boxed{L} * \boxed{L}$), so unlike maps-to resources, they can be shared between threads. The sharing via an invariant comes at a

TOKEN-ALLOCATE
$$P\,1 \vdash \Rrightarrow \exists \gamma.\, \texttt{counter}\ P\ \gamma\ 1 * \texttt{token}\ P\ \gamma$$

TOKEN-DEALLOCATE
$$\texttt{counter}\ P\ \gamma\ 1 * \texttt{token}\ P\ \gamma \vdash \Rrightarrow (\texttt{no\_tokens}\ P\ \gamma * P\,1)$$

TOKEN-MUTATE-INCR
$$\texttt{counter}\ P\ \gamma\ p \vdash \Rrightarrow (\texttt{counter}\ P\ \gamma\ (p+1) * \texttt{token}\ P\ \gamma)$$

TOKEN-INTERACT
$$\texttt{no\_tokens}\ P\ \gamma * \texttt{token}\ P\ \gamma \vdash \bot$$

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

TOKEN-MUTATE-DECR
$$\frac{p > 1}{\texttt{counter}\ P\ \gamma\ p * \texttt{token}\ P\ \gamma \vdash \Rrightarrow \texttt{counter}\ P\ \gamma\ (p-1)}$$

Figure 4.2: The rules for the 'token' ghost theory.

cost—resources L inside an invariant $\boxed{L}$ can be accessed only during *atomic* operations, such as a load or an `FAA` instruction. After the operation finishes, we must show that the invariant still holds. This is shown by Iris's invariant accessing rule (simplified for presentation purposes):[2]

$$\frac{\{L * R\}\ e\ \{L * Q\} \qquad \text{atomic } e}{\{\boxed{L} * R\}\ e\ \{Q\}}$$

Lines 10–14 contains the invariant we use to verify our ARC. We define a value v to be `is_arc` if it is a location ℓ, whose value is governed by an invariant (`inv N` in Coq). The resource `arc_inv` inside the invariant states that location ℓ points to a natural number n, *i.e.,* $\ell \mapsto n$. Additionally, depending on whether $n = 0$, the invariant contains the resource `no_tokens` $P\ \gamma$ or `counter` $P\ \gamma\ n$. (The notation $\ulcorner \phi \urcorner$ denotes the embedding of Coq proposition ϕ in Iris's separation logic.)

The `no_tokens` $P\ \gamma$, `counter` $P\ \gamma\ n$, and `token` $P\ \gamma$ resources, are instances of *ghost state*—non-physical resources that can express protocols. Besides their use for ARC, these resources are also used to verify other data structures (*e.g.,* readers-writer locks). These resources should be seen as atoms of the logic, and are interpreted using an appropriate ghost theory in Iris. What is important is that they satisfy the rules in Figure 4.2. Intuitively, the `counter` $P\ \gamma\ n$ resource states that there are exactly $n > 0$ copies of `token` $P\ \gamma$, while `no_tokens` $P\ \gamma$ states that no `token` $P\ \gamma$ resources exist. Most of these rules mention Iris's update modality $\Rrightarrow$. This modality can be eliminated at every program step, and can be ignored for our purposes. During the verification of `mk_arc`, the TOKEN-ALLOCATE rule is used to allocate a fresh γ for which the `counter` $P\ \gamma\ 1$

[2] Invariant $\boxed{L}^{\mathcal{N}}$ carry a namespace $\mathcal{N}$. The namespace is used to ensure that invariants cannot be accessed twice, which would be unsound. Since Iris's invariants are impredicative Svendsen and Birkedal (2014), one only obtains the resources L under a *later* modality, *i.e.,* $\triangleright L$. These technicalities require additional bookkeeping but are orthogonal to disjunctions.

resource is put in the invariant, and the `token` $P\ \gamma$ resource is returned. The verification of `clone` uses TOKEN-INTERACT to establish that the $n = 0$ disjunct of the invariant is contradictory. After incrementing the stored value to $n + 1$, TOKEN-MUTATE-INCR is used to create an additional `token` $P\ \gamma$. The verification of `mk_arc` and `clone` poses no problems for state-of-the-art tools like Caper (Dinsdale-Young et al., 2017) and Diaframe (Chapter 2). The problem lies with `drop`, where both tools require guidance from the user. With our approach, `drop` can now be verified completely automatically.

Informed disjunction introduction in the verification of drop. Let us consider what happens during the verification of `drop`. When we execute the `FAA` instruction, we access our invariant to obtain some n with $\ell \mapsto n$. Similar to the proof of `clone`, the rule TOKEN-INTERACT states that the $n = 0$ disjunct in our invariant is contradictory, so we only need to consider the second clause, *i.e.,* that $0 < n$ and additionally we have `counter` $P\ \gamma\ n$. After symbolic execution of `FAA`, we have $\ell \mapsto (n - 1)$ and need to reestablish the invariant. It is easy to reestablish the first separating conjunct of the invariant: we just need to relinquish the $\ell \mapsto (n-1)$ resource. Reestablishing the second part of the invariant requires us to prove the following separation logic entailment:

PROBLEM-GC-ARC-DROP

$$\ulcorner n > 0 \urcorner, \texttt{counter}\ P\ \gamma\ n, \texttt{token}\ P\ \gamma \vdash \Rrightarrow \begin{pmatrix} (\ulcorner n - 1 = 0 \urcorner * \texttt{no\_tokens}\ P\ \gamma) \\ \vee\ (\ulcorner 0 < n - 1 \urcorner * \texttt{counter}\ P\ \gamma\ (n-1)) \end{pmatrix} * R$$

The disjunction originates from the invariant definition on line 11–12 in Figure 4.1. The $* R$ represents the verification of the remaining body of `drop`:[3] resources not required for restoring the invariant may be needed in this remaining verification. Indeed, to verify the remaining body we need $P\ 1$ whenever `drop` returns `true`, and we can only obtain $P\ 1$ with TOKEN-DEALLOCATE.

Proving the disjunction in PROBLEM-GC-ARC-DROP is challenging—with the current proof context, neither disjunct is provable. Both Caper and Diaframe try to *backtrack* on the choice of disjunct, but backtracking is hopeless—a correct proof needs to perform a *case analysis* on whether $n = 1$. In other words, it should consider whether we are the last reader, or there are more readers left. If $n = 1$, the first disjunct is provable, while if $n \neq 1$, the second disjunct is provable. Our approach automatically detects the need for this case split by the presence of a *ground connection* from Diaframe's dummy hypothesis $\varepsilon_1 \triangleq \top$ to the pure guard $\ulcorner n - 1 = 0 \urcorner$ (§4.5.3).

4.2.2 ARC with Explicit Deallocation

The problematic disjunction in the previous subsection had a specific shape: $(\ulcorner \phi \urcorner * L_1) \vee L_2$, *i.e.,* the left disjunct is guarded by a pure condition. One could imagine an ad-hoc approach that only handles disjunctions guarded by pure conditions, but other proofs involve disjunctions where the required case analysis is not this apparent from the syntax. We demonstrate this with a version of the ARC where the location is deallocated when the last reader is `drop`ped.

The changes in the implementation and verification of the modified ARC can be found in Figure 4.3. The `mk_arc` and `clone` methods have the same implementation and

[3] To be precise, $R \triangleq \texttt{wp}\ (\#n = \#1)\ \{v.\ \exists (b : \mathbb{B}).\ \ulcorner v = \#b \urcorner * ((\ulcorner b = \texttt{true} \urcorner * P\ 1) \vee \ulcorner b = \texttt{false} \urcorner)\}$.

```
1  Definition drop_free : val :=
2    λ: "a",
3      let: "old_val" := FAA "a" #(-1) in
4      if: "old_val" = #1 then
5        Free "a";; #true
6      else
7        #false.
8  Definition arc_inv_free (γ : gname) (l : loc) : iProp :=
9    no_tokens P γ ∨ (∃ p : positive, l ↦ #(Zpos p) ∗ token_counter P γ p).
10 Definition is_arc_free (γ : gname) (v : val) : iProp :=
11   ∃ (l : loc), ⌜v = #l⌝ ∗ inv N (arc_inv_free γ l).
12 Program Instance drop_arc_free_spec (γ : gname) (v : val) :
13   SPEC {{ token P γ ∗ is_arc_free γ v }}
14     drop v
15   {{ (b : bool), RET #b; (⌜b = true⌝ ∗ P 1) ∨ ⌜b = false⌝ }}.
```

Figure 4.3: Automatic verification of an ARC with explicit deallocation.

specification, and are thus omitted. The difference lies in the implementation of `drop`, which deallocates the location if called with the last reader. In the logic, symbolic execution of `Free` will consume the maps-to resource $\ell \mapsto n$, after which this resource can no longer appear in the invariant. Accordingly, the invariant `arc_inv_free` in line 8 now features a top-level disjunction. This disjunction states that either we know that no tokens remain, or some tokens remain and location ℓ keeps a record of how many.

Accessing invariants with disjunctions. The changed invariant puts us in a bit of a pickle: to symbolically execute the `FAA` in `clone` and `drop_free`, we require an $\ell \mapsto n$ resource. In the verification in §4.2.1 we are sure to get $\ell \mapsto n$ when opening the invariant, but this is no longer a given in the current version. In the proof of `clone` and `drop_free`, we need the TOKEN-INTERACT rule (which states that `token` P γ and `no_tokens` P γ are contradictory) to even establish $\ell \mapsto n$.

Two different approaches to deal with disjunction elimination have been considered in prior work. Caper (Dinsdale-Young et al., 2017) handles them by trying to open any invariant (called a *region* in Caper) in the proof context, finding contradictions if possible, and hoping to get the relevant resource out of some of those invariants. This approach is sufficient to access the invariant in this example, but it is very inefficient on goals with multiple invariants. Diaframe (Chapter 2) only accesses an invariant when it is sure the invariant is relevant for the goal—that is, the invariant can be used to discharge the left-most separating conjunct. To symbolically execute the `FAA`, it detects that it needs to establish a maps-to resource $\ell \mapsto n$ for some n. It thus looks for invariants containing a maps-to resource for ℓ. Unfortunately, the maps-to resource appears beneath a disjunction in our invariant, so Diaframe just gives up. To complete the proof, Diaframe requires the user to guide the proof search. Our new approach is capable of looking beneath disjunctions to determine whether an invariant is relevant or not. It will establish a *connection* between `arc_inv_free` and the goal $\ell \mapsto n$, allowing it to automatically determine which invariant to access. After finding the connection, the proof of `clone` proceeds largely the same as in §4.2.1.

Reestablishing invariants with non-guarded disjunctions. The verification of `drop_free` poses another challenge. To reestablish the invariant after symbolic execution of the `FAA` instruction, we need to prove the following entailment (where p is a positive natural number):

PROBLEM-FREE-ARC-DROP

$$\texttt{counter}\ P\ \gamma\ p, \texttt{token}\ P\ \gamma, \ell \mapsto (p-1) \vdash \Rrightarrow \begin{pmatrix} \texttt{no\_tokens}\ P\ \gamma \\ \lor\ (\exists p'.\ \ell \mapsto p' * \texttt{counter}\ P\ \gamma\ p') \end{pmatrix} * R$$

The disjunction behind the turnstile comes directly from the definition of the invariant `arc_inv_free` on line 8–9 in Figure 4.3. The situation is the same as in §4.2.1: neither side of the disjunction is provable, so directly backtracking on the choice of disjunct is hopeless. The $* R$ again represents the verification of the remaining body of `drop_free`, so it is crucial to know *now* which resources are necessary for restoring the invariant: they might be needed to prove R. The resources for the invariant and for the remaining verification are more entwined than before. If $p = 1$, we need both $\ell \mapsto (p-1)$ and $P\,1$ to finish the verification, since the `Free` ℓ operation will consume the maps-to resource.

We need to perform a case analysis between $p = 1$ and $p \neq 1$ to continue, but this is not apparent from the goal. Our approach can figure out this case distinction automatically, since an appropriate *ground connection* is found from the `counter` $P\ \gamma\ p$ to the `no_tokens` $P\ \gamma$ resources (§4.5.3). This connection instructs the proof search that if we *may* want to prove `no_tokens` $P\ \gamma$, and we have `counter` $P\ \gamma\ p$, it should perform a case analysis on $p = 1$ and $p \neq 1$ to proceed. This ground connection allows us to verify this ARC automatically, but it is more generic than that. It is also used in verifications of readers-writer locks, which also use 'token' ghost theory.

4.3 Propositional Logic

We present the basic idea behind our approach for informed disjunction introduction and elimination. The basic idea can be seen as relying on a significant simplification of the notion of *connection* from connection calculi (Waaler, 2001; Wallen, 1990; Otten and Kreitz, 1995). These calculi are complete for their specific logics, but not easy to extend to other formalisms, particularly separation logic (Galmiche and Méry, 2002). We thus do not attempt to formulate a complete calculus, but instead present a method that can be extended to separation logic (§4.4) and ultimately Iris (§4.5). A comparison between our simplified version and complete connection calculi can be found in §4.8.

We start by phrasing the problems we have seen in §4.2 in the context of propositional intuitionistic logic (§4.3.1). We illustrate our solution in this simple setting to give the reader an intuition, and to highlight its essential proof-theoretic features, without getting distracted by specific features of separation logic (*e.g.,* substructural aspects) and Iris (*e.g.,* invariants, ghost state, and modalities). Our solution for this logic consists of a set of rules that enables goal-directed elimination and introduction of disjunctions (§4.3.2). Finally, we work out a representative example (§4.3.3), and discuss how our calculus can support domain-specific knowledge via *ground connections* (§4.3.4),

4.3.1 Challenges and Key Idea

Recall from §4.2 that disjunctions pose two challenges for automated proof search. First, when proving $\Delta \vdash P \lor Q$, neither $\Delta \vdash P$ nor $\Delta \vdash Q$ might be true. In that case, uninformed disjunction introduction by backtracking will never find a proof. Second, we consider disjunctions inside invariants, for which it is often unclear if their elimination is helpful. In existing tools such disjunctions are either eliminated overeagerly (in Caper (Dinsdale-Young et al., 2017)) or never (in Chapter 2). These challenges with disjunction introduction and elimination are not specific to concurrent separation logic. To see that, let us consider the following example (we let $\Delta \vdash_{\mathsf{p}} P$ denote an entailment in intuitionistic propositional logic):

$$A \to ((B \land C) \lor F), A \vdash_{\mathsf{p}} B \lor F.$$

A naive approach would be just to 'try everything', which at one point would eliminate the implication and the disjunction underneath. As argued before, this naive approach is inefficient and does not scale when considering quantifiers. Our approach instead notices that B occurs both in the first hypothesis and in the first disjunct. Establishing a *connection* from the first hypothesis to the first disjunct allows us to continue by proving two easier goals. Contrary to the naive approach, we know *upfront* that eliminating the implication could help prove our goal, but we do not yet know if we can prove the left-hand side of the implication.

4.3.2 Calculus

We now put the intuitive idea behind connections on a formal footing. We let A, B, and C range over atoms, and let P and Q range over formulas of intuitionistic propositional logic. We let the contexts Δ and Γ be sets of formulas. We consider a formal system with two logical judgments:

- The *entailment judgment* $\Delta \vdash_{\mathsf{p}} \Gamma$, which we interpret as $\bigwedge \Delta \vdash_{\mathsf{p}} \bigvee \Gamma$.
- The *connection judgment* $P, [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']$, which we interpret as $P \land Q \vdash_{\mathsf{p}} A \lor \bigvee \Gamma'$.

The idea of having a context Γ on the right is inspired by the multi-succedent calculus for intuitionistic logic by Dragalin (1988). While Dragalin only has the entailment judgment, we extend the system with a connection judgment $P, [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']$. This judgment establishes a *connection* from formula P to atom A, with *side-conditions* Q and *remaining cases* Γ'. The terms between brackets can be seen as outputs: given a hypothesis $P \in \Delta$ and goal A, the rules of the judgment try to establish a connection by determining appropriate Q and Γ'. Our rules ensure that the only way to establish a connection from P to A is to find A occurring strictly positively in P, *i.e.,* A occurs on the right-hand side of any implication, but possibly under conjunctions and disjunctions.

The rules of our system can be found in Figure 4.4. It is trivial to establish that these rules are sound w.r.t. the semantic interpretation—they can simply be derived from the rules of intuitionistic propositional logic (see §4.6 for more details how this is done in Coq). Inspired by focusing (Andreoli, 1992; Simmons, 2014; Liang and Miller, 2009), the rules are divided into two groups: the inversion phase, and the focusing phase. Eventually, we want to turn our rules into an algorithm. We intend to apply inversion rules eagerly, without backtracking on the options. This limits the search space and improves efficiency. One of the inversion rules will require a connection, and connections

$$\mathrm{R}\wedge\ \frac{\Delta \vdash_{\mathsf{p}} P, \Gamma \qquad \Delta \vdash_{\mathsf{p}} Q, \Gamma}{\Delta \vdash_{\mathsf{p}} P \wedge Q, \Gamma} \qquad \mathrm{R}\vee\ \frac{\Delta \vdash_{\mathsf{p}} P, Q, \Gamma}{\Delta \vdash_{\mathsf{p}} P \vee Q, \Gamma} \qquad \mathrm{R}\top\ \frac{}{\Delta \vdash_{\mathsf{p}} \top, \Gamma} \qquad \mathrm{R}\to\ \frac{\Delta, Q \vdash_{\mathsf{p}} P}{\Delta \vdash_{\mathsf{p}} Q \to P, \Gamma}$$

$$\mathrm{L}\bot\ \frac{}{\Delta, \bot \vdash_{\mathsf{p}} \Gamma} \qquad \text{F-A}\ \frac{P \in \Delta \qquad A \in \Gamma \qquad P, [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma'] \qquad \Delta \vdash_{\mathsf{p}} Q, \Gamma \qquad \Delta, \bigvee \Gamma' \vdash_{\mathsf{p}} \Gamma}{\Delta \vdash_{\mathsf{p}} \Gamma}$$

Inversion phase

Focusing phase

$$\text{L-A}\ \frac{}{A, [\top] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\epsilon]} \qquad \mathrm{L}\wedge\ \frac{P_i, [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']}{(P_1 \wedge P_2), [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']}$$

$$\mathrm{L}\to\ \frac{P, [Q_2] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']}{(Q_1 \to P), [Q_2 \wedge Q_1] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']} \qquad \mathrm{L}\vee\ \frac{P_i, [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']}{(P_1 \vee P_2), [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [P_{3-i}, \Gamma']}$$

Figure 4.4: Our calculus for propositional logic based on connections.

can only be established with focusing rules. On these rules we *do* intend to backtrack. The premises of focusing rules can only be established by other focusing rules.

Inversion phase. Rules R$\wedge$, R$\vee$, R$\top$ and L$\bot$ are all straightforward. The R$\to$ rule chooses one disjunct for implication introduction, after which the other disjuncts are no longer available. In a classical logic, it would be valid to keep Γ around, but we consider an intuitionistic system. Eager application of the above rules effectively digs up all atoms under conjunctions and disjunctions. One then uses the main workhorse F-A.

Suppose our goal is $\Delta \vdash_{\mathsf{p}} \Gamma$ with $A \in \Gamma$ an atom. Rule F-A requires an hypothesis $P \in \Delta$ for which a connection judgment $P, [Q] \vdash^{\mathsf{c}}_{\mathsf{p}} A, [\Gamma']$ can be established. Once such a connection is established, it is sufficient to prove the side-condition Q—we continue with goal $\Delta \vdash_{\mathsf{p}} Q, \Gamma$. Additionally, we need to prove a new goal $\Delta, \bigvee \Gamma' \vdash_{\mathsf{p}} \Gamma$ to cover the remaining cases. Here we define $\bigvee \epsilon \triangleq \bot$ and $\bigvee (P_1, \ldots, P_n) \triangleq P_1 \vee \ldots \vee P_n$. To see why F-A is sound, note that proofs of $\Delta \vdash_{\mathsf{p}} Q, \Gamma$ either establish $\Delta \vdash_{\mathsf{p}} \Gamma$ or $\Delta \vdash_{\mathsf{p}} Q$. In the first case, we are done. In the latter case, our connection tells us that $\Delta \vdash_{\mathsf{p}} A, \Gamma'$. Since for the remaining cases we have $\Delta, \bigvee \Gamma' \vdash_{\mathsf{p}} \Gamma$ the rule is sound.

Focusing phase. The rules in this phase are responsible for finding a connection, *i.e.,* decomposing the hypothesis P to a strictly positive occurrence of the atom A. The rules basically perform structural recursion on the focused hypothesis P. Rule L-A states that A proves A with a trivial side-condition and no remaining cases. Rule L$\to$ establishes a connection for an implication $Q \to P$, by adding side-condition Q to an established connection from P to A. Rule L$\vee$ establishes a connection for a disjunction $P_1 \vee P_2$, by adding the unused disjunct to the remaining cases. Finally, rule L$\wedge$ simply looks beneath

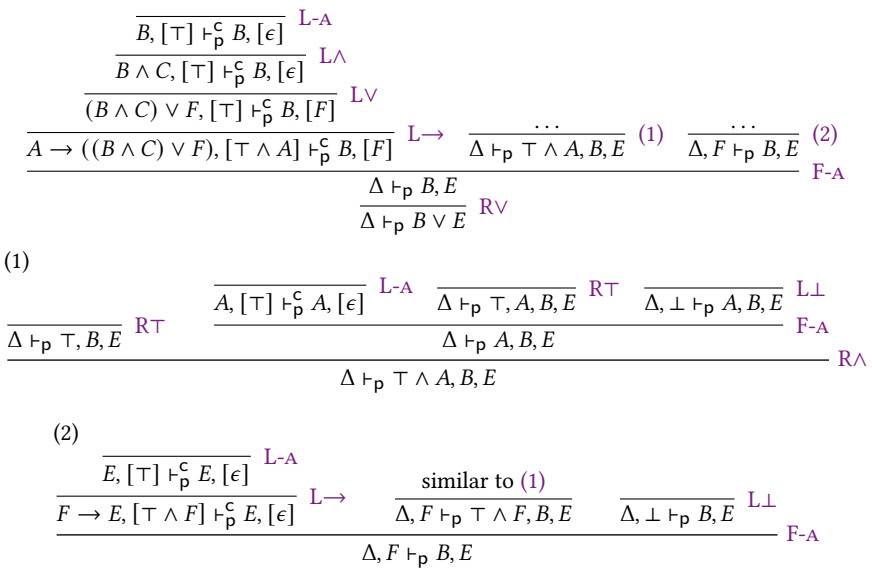

Figure 4.5: Example derivation in our propositional calculus (we let $\Delta \triangleq A \to ((B \wedge C) \vee F), A, F \to E$).

conjunctions to establish a connection.

Sources of incompleteness. The rules in Figure 4.4 are not sufficient for all goals in intuitionistic logic. For example, goal $A \vee B \vdash_{\mathsf{p}} \top \to A, \top \to B$ is not provable, nor is $A, A \to \bot \vdash_{\mathsf{p}} B$. This would require an appropriate way to 'look under' an implication with focusing rules. At the cost of complicating the system, our method may be extended to approach completeness. The trade-off between completeness and complexity may be chosen separately for each practical adaptation. For our purposes, incompleteness for the above goals is acceptable. We are primarily looking for a way to make progress on the kind of disjunctions that appear in invariants.

4.3.3 Example

Figure 4.5 contains a possible derivation of $A \to ((B \wedge C) \vee F), A, F \to E \vdash_{\mathsf{p}} B \vee E$ (this entailment is closely related to the one in §4.3.1). For the first application of F-A, we find (and prove) a connection from the implication to B, with side-condition A. This means we now have to show that side-condition A holds, and this is done in (1). However, we also have to deal with the remaining case F. As (2) shows, for F we will actually pick a different disjunct to prove, namely E.

This demonstration shows that our rules are capable of goal-directed disjunction introduction and elimination. However, they do not yet constitute an *algorithm*. This has a couple of reasons:

- The disjuncts Γ are a set, so it is unclear what disjunct to use for F-A.

- The conjuncts Δ are a set, so it is unclear what conjunct to use for F-A, if multiple apply.
- Using R$\rightarrow$ will irrevocably pick a disjunct to prove. If all disjuncts are implications, one can only proceed with this rule—but how to pick the correct disjunct?
- If rule F-A is applicable on goal $\Delta \vdash_{\mathsf{p}} \Gamma$ for some $P \in \Delta$, $A \in \Gamma$, and some Q and Γ', it is also applicable for the same P, Q and Γ' for the spawned subgoals $\Delta \vdash_{\mathsf{p}} Q, \Gamma$ and $\Delta, \bigvee \Gamma' \vdash_{\mathsf{p}} \Gamma$. This can cause loops.

Since intuitionistic propositional logic is not our target logic, we leave the system as is. In the next section, we show how our approach scales to separation logic, and how we *can* turn the approach into an algorithm in that setting.

4.3.4 Ground Connections

The system in this section deals with uninterpreted atoms. When we use it for program verification, the atoms will represent domain-specific propositions about data types (natural numbers, maps, lists, sets) and Iris's ghost state. Some atoms will be distinct, but related in some way. Consider the entailment $0 \le n, n \le 1 \vdash_{\mathsf{p}} (0 < n \land n \le 1) \lor n = 0$. Without domain-specific information, the system will never be able to show its validity. *Ground connections* provide a way to incorporate domain-specific relations between atoms in the system. In this case, we could provide a ground connection from $0 \le n$ to $0 < n$ by proving the connection judgment $0 \le n, [\top] \vdash^{\mathsf{c}}_{\mathsf{p}} 0 < n, [n = 0]$. By adding this rule as an axiom in the focusing phase, we have extended our system with domain-specific knowledge. We will revisit our idea of ground connections for ghost resources in §4.5.3.

4.4 Separation Logic

We now develop our connection-based approach for propositional (*i.e.,* without quantifiers) separation logic. We first provide background on separation logic and outline the challenges with its substructural nature (§4.4.1). We then propose an extension of our calculus for separation logic (§4.4.2). This calculus can be turned into an algorithm, which we explain and apply on an example (§4.4.3).

4.4.1 Background and Challenges

We do not base our approach on a specific model of separation logic (*e.g.,* Iris), but make it parametric over models of Bunched Implication (BI) logic (O'Hearn and Pym, 1999; Pym, 2002). That is, our approach is parametric over a structure with the usual logical connectives, along with the *separating conjunction* ($*$), and *magic wand* ($-\!\!*$). The separating conjunction ($*$) is assumed to be associative and commutative, have a neutral element ($\top * P \dashv\vdash P$),[4] and be the adjoint of the magic wand ($P \vdash Q -\!\!* R$ iff $P * Q \vdash R$). In program verification frameworks such as Iris, the separating connectives are used instead of the regular ones ($\land$ and $\rightarrow$) in most proofs. We thus omit regular implication

[4] To ease presentation, we consider *affine* BIs, where $\land$ and $*$ have the same neutral element $\top = \mathsf{True} = \mathsf{Emp}$. Such BIs satisfy $P * Q \vdash P$. Our full version in Coq supports general BIs where $\land$ and $*$ have different neutral elements.

from the fragment of separation logic we consider. To represent intermediate goals, we allow regular conjunction on the right-hand side of the turnstile, but do not support it in hypotheses. Similar to §4.3, we consider multi-succedent entailments. These are now of the form $\Delta \vdash_{\mathsf{s}} \Gamma$, and should be interpreted as $\ast\, \Delta \vdash_{\mathsf{s}} \bigvee \Gamma$. Unlike §4.3, we consider Δ and Γ to be a *list* of formulas instead of a set. This will help us turning our rules into an algorithm (§4.4.3).

Introduction of separating conjunction. The main additional challenge of separation logic is its substructural nature—resources may only be used once, and we will need to accommodate for this in our calculus. The most striking consequence of substructurality is that we do not have $P \vdash_{\mathsf{s}} P * P$ in general. The introduction rule for separating conjunction ($*$) thus differs in that of regular conjunction ($\wedge$) by having to subdivide resources over the conjuncts:

$$\frac{\Delta_1 \vdash_{\mathsf{s}} L \qquad \Delta_2 \vdash_{\mathsf{s}} G}{\Delta_1, \Delta_2 \vdash_{\mathsf{s}} L * G}$$

(Remember that the resources in Δ are conjuncted with $*$, not $\wedge$.) Choosing how to subdivide the resources with this rule is challenging. We are aware of two approaches to deal with this problem, both of which rely on restricting the grammar of the logic. The classical restriction is to the (linear) hereditary Harrop fragment (Miller et al., 1991), where the final conclusion of a magic wand must always be a single atom. In this system, the proper distribution to Δ_1 and Δ_2 can be determined by annotating the entailment with an input and output environment (Cervesato et al., 2000; Hodas and Miller, 1991), or by annotating every hypotheses in Δ with a Boolean constraint (Harland and Pym, 1997). However, this fragment does not contain goals like $A, A \mathrel{-\!\!*} (B * C) \vdash_{\mathsf{s}} B * C$, which we do wish to consider. We thus follow a recent approach that instead restricts the grammar of the left conjunct L. This allows the separating conjunction to be proven 'in place', *i.e.,* without explicitly splitting the context. This approach first appeared in RefinedC (Sammler et al., 2021), and was later adapted by Diaframe (Chapter 2). Essentially, the following two rules are applied eagerly to push an atom A to become the left-most conjunct (*i.e.,* $\Delta \vdash_{\mathsf{s}} A * G$):

$$\frac{\Delta \vdash_{\mathsf{s}} L_1 * (L_2 * G), \Gamma}{\Delta \vdash_{\mathsf{s}} (L_1 * L_2) * G, \Gamma} \qquad \frac{\Delta \vdash_{\mathsf{s}} G, \Gamma}{\Delta \vdash_{\mathsf{s}} \top * G, \Gamma}$$

Here, we let $L ::= \top \mid A \mid L * L$. Limiting the grammar of the left conjunct enables a deterministic rule for introducing the separating conjunction, and—as RefinedC and Diaframe have demonstrated—this limitation is acceptable: interesting verification goals remain expressible.

4.4.2 Calculus

To adapt the proof rules from §4.3 to separation logic, we adopt the approach mentioned in the previous subsection, limiting the grammar of the left conjunct L and adding rules to push atoms to be the left-most conjunct. Only for goals of shape $\Delta \vdash_{\mathsf{s}} A * G, \Gamma$ will we look for a connection from some $H \in \Delta$ to A. The connection judgment $H, [L] \vdash^{\mathsf{c}}_{\mathsf{s}} A * [U], [\Lambda]$ also changes: it now contains a parameter U, which we dub the *residue*, that describes the

$$\mathrm{R}\top\ \frac{}{\Delta \vdash_{\mathsf{s}} \top, \Gamma} \qquad \mathrm{R}A\ \frac{\Delta \vdash_{\mathsf{s}} A * \top, \Gamma}{\Delta \vdash_{\mathsf{s}} A, \Gamma} \qquad \mathrm{R}\wedge\ \frac{\Delta \vdash_{\mathsf{s}} G_1, \Gamma \qquad \Delta \vdash_{\mathsf{s}} G_2, \Gamma}{\Delta \vdash_{\mathsf{s}} G_1 \wedge G_2, \Gamma} \qquad \mathrm{R}\vee\ \frac{\Delta \vdash_{\mathsf{s}} G_1, G_2, \Gamma}{\Delta \vdash_{\mathsf{s}} G_1 \vee G_2, \Gamma}$$

$$\mathrm{R}{-\!\!*}_{\bot}\ \frac{}{\Delta \vdash_{\mathsf{s}} \bot -\!\!* G, \Gamma} \qquad \mathrm{R}{-\!\!*}_{\top}\ \frac{\Delta \vdash_{\mathsf{s}} G, \Gamma}{\Delta \vdash_{\mathsf{s}} \top -\!\!* G, \Gamma} \qquad \mathrm{R}{-\!\!*}_{*}\ \frac{\Delta \vdash_{\mathsf{s}} U_1 -\!\!* U_2 -\!\!* G, \Gamma}{\Delta \vdash_{\mathsf{s}} (U_1 * U_2) -\!\!* G, \Gamma} \qquad \mathrm{R}{-\!\!*}_{\vee}\ \frac{\Delta \vdash_{\mathsf{s}} (U_1 -\!\!* G) \wedge (U_2 -\!\!* G), \Gamma}{\Delta \vdash_{\mathsf{s}} (U_1 \vee U_2) -\!\!* G, \Gamma}$$

$$\mathrm{R}{-\!\!*}_{H}\ \frac{\Delta, H \vdash_{\mathsf{s}} G}{\Delta \vdash_{\mathsf{s}} H -\!\!* G, \Gamma} \qquad \mathrm{R}{*}_{\top}\ \frac{\Delta \vdash_{\mathsf{s}} G, \Gamma}{\Delta \vdash_{\mathsf{s}} \top * G, \Gamma} \qquad \mathrm{R}{*}_{*}\ \frac{\Delta \vdash_{\mathsf{s}} L_1 * (L_2 * G), \Gamma}{\Delta \vdash_{\mathsf{s}} (L_1 * L_2) * G, \Gamma} \qquad \mathrm{R}{*}_{\vee}\ \frac{\Delta \vdash_{\mathsf{s}} L_1 * G, L_2 * G, \Gamma}{\Delta \vdash_{\mathsf{s}} (L_1 \vee L_2) * G, \Gamma}$$

$$\mathrm{R}{*}_{A}\ \frac{H \in \Delta \qquad H, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U], [\Lambda]}{\Delta \setminus H \vdash_{\mathsf{s}} L * \left((U -\!\!* G) \wedge \left(\bigvee \Lambda -\!\!* (A * G \vee \bigvee \Gamma)\right)\right), H -\!\!* \bigvee \Gamma} \qquad \textsc{unfocus}\ \frac{\Delta \vdash_{\mathsf{s}} \Gamma}{\Delta \vdash_{\mathsf{s}} A * G, \Gamma}$$

Inversion phase

Focusing phase

$$\mathrm{L}\text{-}\mathrm{A}\ \frac{}{A, [\top] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [\top], [\epsilon]} \qquad \mathrm{L}{*}\ \frac{U_i, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * \left[U'\right], [D_1, \ldots, D_n]}{(U_1 * U_2), [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * \left[U' * U_{3-i}\right], [D_1 * U_{3-i}, \ldots, D_n * U_{3-i}]}$$

$$\mathrm{L}{-\!\!*}\ \frac{U_1, [L_2] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U_2], [\Lambda]}{(L_1 -\!\!* U_1), [L_2 * L_1] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U_2], [\Lambda]} \qquad \mathrm{L}{\vee}\ \frac{U_i, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * \left[U'\right], [\Lambda]}{(U_1 \vee U_2), [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * \left[U'\right], [U_{3-i} * L, \Lambda]}$$

Grammar

$$\begin{aligned} A &::= \text{atoms} \\ L &::= \top \mid A \mid L * L \mid L \vee L \\ H &::= A \mid L -\!\!* U \\ U, D &::= \bot \mid \top \mid H \mid U * U \mid U \vee U \\ G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid U -\!\!* G \mid L * G \end{aligned} \qquad \begin{aligned} \Delta &::= \epsilon \mid H, \Delta \\ \Lambda &::= \epsilon \mid U, \Lambda \\ \Gamma &::= \epsilon \mid G, \Gamma \end{aligned}$$

Definitions

$$\Delta \vdash_{\mathsf{s}} \Gamma \triangleq \ast\!\!\!\ast\ \Delta \vdash_{\mathsf{s}} \bigvee \Gamma$$

$$H, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U], [\Lambda] \triangleq H * L \vdash_{\mathsf{s}} (A * U) \vee \bigvee \Lambda$$

Figure 4.6: Our calculus for separation logic based on connections.

resources from H that are not used to establish A. We interpret $H, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U], [\Lambda]$ as $H * L \vdash_{\mathsf{s}} (A * U) \vee \bigvee \Lambda$. Carrying around the residue U is not just for convenience or efficiency: unlike for propositional logic, goals can become unprovable if the resources U are dropped, since they may be needed to prove G. The rules of the system, along with the grammar, can be found in Figure 4.6.

Inversion phase. Let us first consider the rules in the inversion phase. The rules for introducing $\top$, $\wedge$ and $\vee$ are the same as in the system for propositional logic. Rule RA puts a lone atom into a separating conjunction, whose introduction rules we will discuss shortly. The wand introduction rule R$-\!\!*_H$ is the same as that of $\to$, but there are additional wand introduction rules for each entry in the grammar of U. These rules enforce that hypotheses are simplified upon adding them to Δ. We have an introduction rule for $*$ for every entry in the grammar of L, which (using associativity of $*$ and distributivity of $*$ over $\vee$) pushes a lone atom A to be the left conjunct. This is all to prepare for the R$*_A$ rule, which requires a connection to make progress. The rule R$*_A$ for finding a connection is quite a mouthful, so we will go over it in more detail. It plays the same role as F-A in §4.3, yet it looks quite different. To see their relation, we restate F-A, along with an alternative:

$$\text{F-A}\quad \frac{H \in \Delta \qquad A \in \Gamma \qquad H, [L] \vdash_{\mathsf{p}}^{\mathsf{c}} A, [\Lambda] \qquad \Delta \vdash_{\mathsf{p}} L, \Gamma \qquad \Delta, \bigvee \Lambda \vdash_{\mathsf{p}} \Gamma}{\Delta \vdash_{\mathsf{p}} \Gamma} \qquad \text{F-ALT}\quad \frac{H \in \Delta \qquad A \in \Gamma \qquad H, [L] \vdash_{\mathsf{p}}^{\mathsf{c}} A, [\Lambda] \qquad \Delta \vdash_{\mathsf{p}} L \wedge \left(\bigvee \Lambda \to \bigvee \Gamma\right), \Gamma}{\Delta \vdash_{\mathsf{p}} \Gamma}$$

One can check that F-ALT implies F-A, by using R$\wedge$, R$\to$ and R$\vee$. The rule R$*_A$ is the separation logic cousin of F-ALT, but it has to deal with the remaining goal $* G$. Assuming $H \in \Delta$, it states:

$$\frac{H, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U], [\Lambda] \qquad \Delta \setminus H \vdash_{\mathsf{s}} L * \Big(\overbrace{(U \mathrel{-\!\!*} G)}^{(1)} \wedge \overbrace{\Big(\bigvee \Lambda \mathrel{-\!\!*} (A * G \vee \bigvee \Gamma)\Big)}^{(2)}\Big), \overbrace{H \mathrel{-\!\!*} \bigvee \Gamma}^{(3)}}{\Delta \vdash_{\mathsf{s}} A * G, \Gamma}\ (\mathrm{R}*_A)$$

(side-conditions: $H, [L]$; residue: $[U]$; remaining cases: $[\Lambda]$)

Similar to F-ALT, we first establish a connection judgment $H, [L] \vdash_{\mathsf{s}}^{\mathsf{c}} A * [U], [\Lambda]$. The entailment we need to prove has the side-condition L as the left conjunct—but the right conjunct of R$*_A$ is more complicated than in F-ALT. It consists of a regular conjunction of (1) with (2). To see why, note that from the connection we learn $(A * U) \vee \bigvee \Lambda$, while we need to prove $(A * G) \vee \bigvee \Gamma$. In the $A * U$ case, we commit to proving $A * G$, since we have already learned A. What remains is (1): try to prove goal G with our additional resources U, *i.e.,* $U \mathrel{-\!\!*} G$. In the $\bigvee \Lambda$ case, the goal remains unchanged: this is precisely obligation (2), which we also encounter in F-ALT.

It could also be that we cannot establish L. Unlike for propositional logic, that would be a problem: we did consume H and remove it from our context Δ, but we may need it to prove Γ. To cover this case, we have component (3) of shape $H \mathrel{-\!\!*} \bigvee \Gamma$. Essentially, we keep a fall-back disjunct around: should we fail to prove the first disjunct, we can still try (3), with which we will recover hypothesis H. This additionally removes $A * G$ from the goal: we failed to prove it earlier, interpret this as 'the left-most disjunct is unprovable', and remove it from the goal accordingly.

$$\dfrac{\dfrac{H, [A] \vdash^{\mathsf{c}}_{\mathsf{s}} B * [C], [F] \qquad \dfrac{A, [\top] \vdash^{\mathsf{c}}_{\mathsf{s}} A * [\top], [\epsilon] \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{F \vdash_{\mathsf{s}} F}\ \text{follows easily}}{F \vdash_{\mathsf{s}} B * \top, F}\ \textsc{unfocus}}{F \vdash_{\mathsf{s}} (B * \top) \vee F}\ \mathrm{R}\vee}{\vdash_{\mathsf{s}} F \mathrel{-\!\!*} ((B * \top) \vee F), A \mathrel{-\!\!*} H \mathrel{-\!\!*} F}\ \mathrm{R}\mathrel{-\!\!*}_H \qquad \vdots}{A \vdash_{\mathsf{s}} A * \big((C \mathrel{-\!\!*} \top) \wedge \big(F \mathrel{-\!\!*} ((B * \top) \vee F)\big)\big), H \mathrel{-\!\!*} F}\ \mathrm{R}*_A}{H, A, \vdash_{\mathsf{s}} B * \top, F}\ \mathrm{R}A}{H, A, \vdash_{\mathsf{s}} B, F}\ \mathrm{R}\vee}{H, A, \vdash_{\mathsf{s}} B \vee F}\ \mathrm{R}\vee$$

Figure 4.7: Example derivation in our calculus for separation logic (we let $H \triangleq A \mathrel{-\!\!*} ((B * C) \vee F)$).

The final ingredient of the inversion phase is UNFOCUS, which should be used *only* when $\mathrm{R}*_A$ is not applicable—that is, if we cannot find a connection from any $H \in \Delta$ to A. If that is the case, we give up trying to prove this disjunct and continue with the left-most disjunct of Γ. If UNFOCUS is needed directly after an application of $\mathrm{R}*_A$, we will recover the used hypothesis.

Focusing phase. The rules of this phase are used to establish a connection in $\mathrm{R}*_A$. Contrary to their counterparts in §4.3, they need to ensure resources are not dropped, as that may make our goal unprovable. Rule $\mathrm{L}\mathrel{-\!\!*}$ is nearly identical, but rule $\mathrm{L}\vee$ includes the side-condition L in the remaining cases. In $\mathrm{L}*$, we obtain the unused conjunct U_{3-i} in both the residue, and in every disjunct in Λ.

4.4.3 Algorithmic Version

Unlike the system in § 4.3, the system in Figure 4.6 can be turned into an algorithm. Substructurality helps here: since hypotheses must be removed after usage, we will never have trivial loops. On goal $\Delta \vdash_{\mathsf{s}} \Gamma$, the algorithm tries the following repeatedly, in order:

1. Eagerly apply $\mathrm{R}\wedge$, $\mathrm{R}\vee$, $\mathrm{R}\top$, $\mathrm{R}*_\top$, $\mathrm{R}*_*$, $\mathrm{R}*_\vee$, $\mathrm{R}\mathrel{-\!\!*}_\bot$, $\mathrm{R}\mathrel{-\!\!*}_\top$, $\mathrm{R}\mathrel{-\!\!*}_*$, $\mathrm{R}\mathrel{-\!\!*}_\vee$; otherwise:
2. If the left-most disjunct is a wand, apply $\mathrm{R}\mathrel{-\!\!*}_H$; otherwise:
3. If the left-most disjunct is a lone atom, apply $\mathrm{R}A$; otherwise:
4. Our goal has shape $\Delta \vdash_{\mathsf{s}} A * G, \Gamma$ with A an atom. Apply $\mathrm{R}*_A$ to find a connection from $H \in \Delta$ to A. Connections are established by backtracking on rules in the focusing phase. If no connection for any $H \in \Delta$ can be found, then:
5. Drop the left-most disjunct, by applying UNFOCUS.

Figure 4.7 shows our algorithm in action on a slight modification of the example from §4.3.3. We define $H \triangleq A \mathrel{-\!\!*} ((B * C) \vee F)$, and prove $H, A, \vdash_{\mathsf{s}} B \vee F$. We omit derivations of connections, and show the important steps. The connection $H, [A] \vdash^{\mathsf{c}}_{\mathsf{s}} B * [C], [F]$ forces the elimination of the disjunction in H *before* choosing a disjunct, and at the same time makes an informed choice of disjunct: H can sometimes produce B, and precisely in this case we choose the B disjunct in the goal.

The role of backtracking. The only source of backtracking in the algorithm is Item 4, which applies R$*_A$. To apply this rule, we backtrack over the hypothesis $H \in \Delta$ to find a connection to goal A. Determining whether a specific $H \in \Delta$ has a connection to A also involves some backtracking, since we need to choose one of the conjuncts or disjuncts in L$*$ and L$\vee$, respectively. Once we find a connection from some $H \in \Delta$ to A, we *do not* backtrack to find different connections. This is almost never a problem because of the substructural nature of separation logic: a given atomic goal A usually only occurs once in the hypotheses.

Sources of incompleteness. The algorithm is not complete, even with the limited grammar of L. A key source of incompleteness is Item 2, which applies rule R$-\!\!*_H$ when the left-most disjunct is a wand, thereby removing the option to consider another disjunct. On goal $A \mathrel{-\!\!*} B \vee C, A \vdash_{\mathsf{s}} F \mathrel{-\!\!*} B, F \mathrel{-\!\!*} C$, the system has no choice but to commit to the first disjunct, since we do not have a focusing rule to look beneath wands. Another problematic goal is $A \mathrel{-\!\!*} (B * C), A, F \vdash_{\mathsf{s}} (B \vee F) * A$, for which the system will (wrongfully) commit to prove B, whereas F should be proven. Despite this, the system can fully automatically solve practical examples, such as the ones from §4.2. Our implementation also provides tools to deal with problematic goals, which we will discuss in §4.6.

4.5 Iris

Although Iris satisfies the rules in §4.4, it comes with additional connectives and rules that complicate the situation, *e.g.,* modalities, higher-order quantification, invariants, and ghost state. To handle disjunctions as well as these additional features, we marry our connection-based calculus with the existing proof search strategy for Iris provided by Diaframe (Chapter 2).

The key ingredient of Diaframe's strategy is its notion of *bi-abduction hints*, which makes Diaframe parametric in domain-specific knowledge about ghost state. We provide background about bi-abduction hints, and explain why they do not address the problems with disjunctions (§4.5.1). We then generalize the connection judgment to include bi-abduction hints (§4.5.2). We finally show some instances of the bi-abduction version of *ground connections* (introduced for propositional logic in §4.3.4), and how they are used in the automated verification of ARC (§4.5.3).

4.5.1 Background on Diaframe

Diaframe accounts for Iris's higher-order quantification and modalities with bi-abduction hints, which are defined as follows (eliding Iris's *masks* on update modalities):

$$H * [\vec{y}; L] \Vdash [\Rrightarrow] \vec{x}; A * [U] \triangleq \forall \vec{y}.\, \big(H * L \vdash_{\mathsf{s}} \Rrightarrow (\exists \vec{x}.\, A * U)\big).$$

Such hints are used to update hypotheses H to prove goal A, where H and A are some resources. For example, the TOKEN-MUTATE-INCR rule used for the verification of ARC is provided as the following hint by Diaframe's 'token' ghost theory library:

$$\texttt{counter}\ P\ \gamma\ p * [\_; \top] \Vdash [\Rrightarrow]\ \_; \texttt{counter}\ P\ \gamma\ (p+1) * [\texttt{token}\ P\ \gamma]$$

During program verification, this hint can be used to update a hypothesis $\texttt{counter}\ P\ \gamma\ p$ to atomic goal $\texttt{counter}\ P\ \gamma\ (p+1)$, after which we receive a $\texttt{token}\ P\ \gamma$ to help prove the remaining goal. Additionally, Diaframe has a procedure to construct hints recursively. This is used to determine whether opening an invariant is relevant, with a procedure similar to the focusing rules of our calculus (these recursive hints go below quantifiers, hence the binders $\vec{x}$ and $\vec{y}$ in hints).

Bi-abduction hints have two weaknesses. Firstly, they cannot express disjunctive reasoning patterns of ghost state. The following two hints are available for $\texttt{counter}\ P\ \gamma\ p$, but they do not specify what to do for general p, nor can they:

$$\textsc{token-deallocate-biabd}\quad \frac{p = 1}{\texttt{counter}\ P\ \gamma\ p * [\_; \texttt{token}\ P\ \gamma] \Vdash [\Rrightarrow]\ \_; \texttt{no\_tokens}\ P\ \gamma * [P\,1]}$$

$$\textsc{token-mutate-decr-biabd}\quad \frac{p > 1}{\texttt{counter}\ P\ \gamma\ p * [\_; \texttt{token}\ P\ \gamma] \Vdash [\Rrightarrow]\ \_; \texttt{counter}\ P\ \gamma\ (p-1) * [\top]}$$

The second weakness is that bi-abduction hints cannot look beneath disjunctions. If an invariant features a top-level disjunction, no bi-abduction hints can be found without user guidance. We show how connections generalize bi-abduction hints and address both weaknesses.

4.5.2 Connections in Iris

To retain Diaframe's support for ghost resources, while extending it with our connection-based approach for disjunctions, we describe a generalization of bi-abduction hints and connection judgments. The new connection judgment has the modalities and quantifiers from bi-abduction, and the remaining cases from the propositional connection judgment. The challenge is getting the *scoping* right, *i.e.,* determining under which quantification and modality to put the remaining cases.

Diaframe translates verification goals (such as Hoare triples) into an *entailment format* of shape $\Delta \vdash_{\mathsf{s}} \Rrightarrow \exists \vec{x}.\, L * G$, and uses bi-abduction hints to make progress on these entailments. A key component of this approach is the 'lazy' introduction of existentials and modalities. We extend this format to $\Delta \vdash_{\mathsf{s}} \Rrightarrow ((\exists \vec{x}.\, L * G_1) \lor G_2)$. The additional disjunct G_2 makes the format multi-succedent, causing disjunctions to be introduced 'lazily' too.[5]

The definition of the connection judgment CONNECTION-DEF, and its application rule $\mathrm{R}\exists *_A$ can be found in Figure 4.8, along with Diaframe's original definition of bi-abduction hints BIABD-DEF and the corresponding application rule BIABD-APPLY. Arrows signify the scope of variables $\vec{x}$ and $\vec{y}$. The definitions CONNECTION-DEF and BIABD-DEF mainly differ in the Λ parameter, hence:

$$H * [\vec{y}; L] \Vdash [\Rrightarrow]\ \vec{x}; A * [U] \quad \text{if and only if} \quad H, [\vec{y}; L] \vdash^{\mathsf{c}}_{\mathsf{s}} \vec{x}; A * [U], [\epsilon]$$

[5] Even though a disjunction is a special case of an existential quantifier, this format is more general: when G_2 is not a separating conjunction, one cannot directly fit $\Rrightarrow ((\exists \vec{x}.\, L * G_1) \lor G_2)$ into $\Rrightarrow \exists \vec{x}.\, L * G$.

$$H * [\vec{y}; L] \Vdash [\Rrightarrow]\ \vec{x}; A * [U] \triangleq \forall \vec{y}.\ \big(H * L \vdash_{\mathsf{s}} \Rrightarrow (\exists \vec{x}.\ A * U)\big) \quad \textsc{(biabd-def)}$$

BIABD-APPLY
$$\frac{H \in \Delta \qquad H * [\vec{y}; L] \Vdash [\Rrightarrow]\ \vec{x}; A * [U] \qquad \Delta \setminus H \vdash_{\mathsf{s}} \Rrightarrow \exists \vec{y}.\ L * (\forall \vec{x}.\ U \mathrel{-\!\!*} G)}{\Delta \vdash_{\mathsf{s}} \Rrightarrow \exists \vec{x}.\ A * G}$$

$$H, [\vec{y}; L] \vdash^{\mathsf{c}}_{\mathsf{s}} \vec{x}; A * [U], [\Lambda] \triangleq \forall \vec{y}.\ \Big(H * L \vdash_{\mathsf{s}} \Rrightarrow \Big((\exists \vec{x}.\ A * U) \vee \bigvee \Lambda\Big)\Big) \quad \textsc{(connection-def)}$$

R$\exists *_A$
$$\frac{H \in \Delta \qquad H, [\vec{y}; L] \vdash^{\mathsf{c}}_{\mathsf{s}} \vec{x}; A * [U], [\Lambda]}{\Delta \setminus H \vdash_{\mathsf{s}} \Rrightarrow \begin{pmatrix} \big(\exists \vec{y}.\ L * \big((\forall \vec{x}.\ U \mathrel{-\!\!*} \Rrightarrow G_1) \wedge \big(\bigvee \Lambda \mathrel{-\!\!*} \Rrightarrow ((\exists \vec{x}.\ A * G_1) \vee G_2)\big)\big)\big) \\ \vee\ (H \mathrel{-\!\!*} \Rrightarrow G_2) \end{pmatrix}}{\Delta \vdash_{\mathsf{s}} \Rrightarrow ((\exists \vec{x}.\ A * G_1) \vee G_2)}$$

Figure 4.8: Bi-abduction hints and connection judgments for Iris's separation logic

By above equivalence, existing bi-abduction hints become instances of the connection judgment, allowing us to reuse all of Diaframe's ghost libraries. Note that when applying R$\exists *_A$ with $G_2 = \bot$ and $\Lambda = \epsilon$, the resulting goal is quite similar to that of BIABD-APPLY. When establishing these generalized connections, a focusing rule similar to L$\vee$ allows us to look beneath disjunctions in hypotheses. Disjunctive ghost-state reasoning patterns can be expressed by adding domain-specific *ground connections*, which we now discuss.

4.5.3 Ground Connections in Iris

To apply our system to the verification of actual programs, it should be instructed about the domain-specific knowledge on the relation between ghost resources (which are considered atoms in the formal system). We do so by adding *ground connections*, *i.e.*, by adding 'axioms' to the focusing judgment. A *ground connection* from A_1 to A_2 is an axiom $A_1, [\vec{x}; L] \vdash^{\mathsf{c}}_{\mathsf{s}} \vec{y}; A_2 * [U], [\Lambda]$, usually provided (and proved sound) by a ghost resource library. This states that A_1 can *possibly* be updated to A_2, with side-condition L, residue U, and remaining cases Λ. We discuss some examples.

Pure ground connection. In the verification of `drop` in ARC without deallocation (§4.2.1), the original version of Diaframe gets stuck at the problematic entailment PROBLEM-GC-ARC-DROP, namely:

$$\ulcorner n > 0 \urcorner, \texttt{counter}\ P\ \gamma\ n, \texttt{token}\ P\ \gamma \vdash \Rrightarrow \begin{pmatrix} \ulcorner n - 1 = 0 \urcorner * \texttt{no\_tokens}\ P\ \gamma) \\ \vee\ \ulcorner 0 < n - 1 \urcorner * \texttt{counter}\ P\ \gamma\ (n-1)) \end{pmatrix} * R$$

In our new system, the proof strategy looks for a connection to $\ulcorner n - 1 = 0 \urcorner$, and finds the following ground connection:

$$\frac{\phi \text{ is decidable} \qquad \neg\phi \text{ is not provable}}{\varepsilon_1, [\_; \top] \vdash^{\mathsf{c}}_{\mathsf{s}} \_; \ulcorner \phi \urcorner * [\ulcorner \phi \urcorner], [\ulcorner \neg \phi \urcorner]}$$

The ε_1 hypothesis is a technical trick from Chapter 2. It is a syntactic marker with $\varepsilon_1 \triangleq \top$, and always said to be the last hypothesis in the context (sound since $\Delta \vdash \top$). The connection states that for decidable ϕ, there is a connection from ε_1 to $\ulcorner\phi\urcorner$, *i.e.,* that ε_1 can sometimes be updated to $\ulcorner\phi\urcorner$. Whenever it can, we learn $\ulcorner\phi\urcorner$ additionally. Whenever it cannot, we learn $\ulcorner\neg\phi\urcorner$. The proof automation will thus perform case analysis on pure goals ϕ whenever a disjunct is guarded by ϕ. The '$\neg\phi$ is not provable' condition is met whenever the pure automation cannot establish $\neg\phi$. This is necessary to prevent loops: in the remaining case we learn $\neg\phi$, but are still faced with a disjunct containing ϕ. Since we can now establish $\neg\phi$, this ground connection will not be found.

Token ground connection. In the verification of `drop` in ARC with explicit deallocation (§4.2.2), the problematic entailment is PROBLEM-FREE-ARC-DROP, namely:

$$\texttt{counter}\ P\ \gamma\ p, \texttt{token}\ P\ \gamma, \ell \mapsto (p-1) \vdash \Rrightarrow \begin{pmatrix} \texttt{no\_tokens}\ P\ \gamma \\ \lor\ \exists p'.\ \ell \mapsto p' * \texttt{counter}\ P\ \gamma\ p' \end{pmatrix} * R$$

Note that TOKEN-DEALLOCATE-BIABD is not applicable, since we only know $p > 0$. The proof search strategy will find the following ground connection from `counter` $P\ \gamma\ p$ to `no_tokens` $P\ \gamma$.

MAYBE-TOKEN-DEALLOC

$$\frac{p \neq 1 \text{ is not provable}}{\texttt{counter}\ P\ \gamma\ p, [\_; \texttt{token}\ P\ \gamma] \vdash^{\mathsf{c}}_{\mathsf{s}} \_; \quad \texttt{no\_tokens}\ P\ \gamma * [\ulcorner p = 1 \urcorner * P\,1], [\texttt{counter}\ P\ \gamma\ p * \texttt{token}\ P\ \gamma * \ulcorner p \neq 1 \urcorner]}$$

This states that if it is possible that $p = 1$, then `counter` $P\ \gamma\ p$ can *sometimes* be updated to prove `no_tokens` $P\ \gamma$, if we additionally provide a `token` $P\ \gamma$. Since it does not require $p = 1$, it is stronger than TOKEN-DEALLOCATE-BIABD. If the update is succesful, we learn $p = 1$ and $P\,1$. Otherwise, we recover both `counter` $P\ \gamma\ p$ and `token` $P\ \gamma$, and additionally learn $p \neq 1$. This ground connection performs the desired case distinction in the verification of ARC, but is also used in the verification of readers-writer locks. The '$p \neq 1$ is not provable' condition is again necessary to prevent loops.

4.6 Soundness Proof and Implementation in Coq

Soundness. In our artifact (Mulder et al., 2023a) we prove soundness of our calculi. We do so by giving a semantic interpretation of the propositions and judgments, and prove the derivation rules as lemmas about the semantic interpretation. For the calculus for propositional logic (§4.3), propositions are interepreted as Coq's propositions `Prop`. For the separation-logic versions (§ 4.4 and 4.5), we interpret propositions within an arbitrary BI logic (O'Hearn and Pym, 1999; Pym, 2002) (we use the type classes from the MoSeL framework (Krebbers et al., 2018) for BIs in Coq). Since Iris is an instance of a BI, and Iris is sound (Jung et al., 2018b, Thm. 7), this means our proof automation constructs closed Coq proofs w.r.t. the operational semantics of the programming language.

Algorithm. The algorithm from §4.4.3, extended with the support for higher-order quantification and modalities described in §4.5, has been implemented as a fork of the Diaframe library. It consists of ca. 22.000 lines of Coq code, about 7.000 more than the

original library. The algorithm works by applying the proof rules from our calculus, and is thus trivially sound. We rely on type classes (Sozeau and Oury, 2008), for instance, to register ground connections. The proof search strategy is available as a tactic `iSteps`, implemented with Ltac (Delahaye, 2000).

Debugging and cooperation with interactive proofs. Since our calculi and algorithms are incomplete, they will fail to prove some goals that might nevertheless be provable. In such cases, we aim to provide better information and cooperation with interactive proofs than traditional backtracking proof search. We discuss some common patterns where the proof search gets stuck, and discuss the additional tactics we provide to investigate or complete such proofs.

- *Unprovable part of goal is not inside a disjunction:* $A_i \vdash_\mathsf{s} (A_1 \lor A_2) * B$. For this example, the `iSteps` tactic will make partial progress: it proves the appropriate side of the disjunction, then stops at the remaining goal $\vdash_\mathsf{s} B$. This is precisely the problematic part of the original goal.

- *The algorithm commits to a wrong disjunct with* R$\mathrel{-\!\!*}_H$: $A \mathrel{-\!\!*} (B * C), A, F \vdash_\mathsf{s} (B \lor F) * A$. The `iSteps` tactic makes a bad choice here: after finding a connection from hypothesis $A \mathrel{-\!\!*} (B * C)$ to goal B and some further steps, the goal becomes $F \vdash_\mathsf{s} C \mathrel{-\!\!*} A, A \mathrel{-\!\!*} (A \mathrel{-\!\!*} (B * C)) \mathrel{-\!\!*} (F * A)$. Rule R$\mathrel{-\!\!*}_H$ then commits to the wrong disjunct: we get stuck at $F, C \vdash_\mathsf{s} A$. We encountered a situation like this during the verification of Peterson (1981)'s algorithm. Our implementation provides two options to proceed. First, there is the `iStepsSafe` tactic, which stops the algorithm just before dropping disjuncts with R$\mathrel{-\!\!*}_H$. The user can then manually pick the correct side of the disjunct using the standard Iris Proof Mode tactics. Second is the `iSmash` tactic, which *will* backtrack to try UNFOCUS if R$\mathrel{-\!\!*}_H$ failed to produce a proof. This is sufficient for Peterson (1981)'s algorithm and the simple example, but will still fail on $A \mathrel{-\!\!*} B \lor C, A \vdash_\mathsf{s} F \mathrel{-\!\!*} B, F \mathrel{-\!\!*} C$.

- *The algorithm wrongfully drops a disjunct with* UNFOCUS: $A \land B, C \vdash_\mathsf{s} (A \lor F) * C$. Our calculus has no support for regular conjunction in hypotheses, which means that no connection to A can be established. (Support could be added—this example serves to demonstrate what happens when the algorithm misses a desirable connection.) After UNFOCUS, we thus get stuck at the unprovable goal $A \land B, C \vdash_\mathsf{s} F * C$. We provide the `iStepsSafest` tactic for such cases: it behaves like `iStepsSafe`, but also stops the algorithm just before UNFOCUS would drop the left-most disjunct. `iStepsSafest` may still turn provable goals into unprovable goals, by finding and using connections for some pathological resources. Resources that feature a '$\bot \mathrel{-\!\!*} A$' pattern may cause problems, for example. We have not seen this happen in practice.

4.7 Evaluation

We evaluate our approach by verifying the examples in Diaframe's original benchmark. Figure 4.9 contains a comparison between our connection-based approach, the original Diaframe, and Caper.

Comparison with Diaframe 1.0. Our connection-based handling of disjunctions increases the number of examples that can be verified fully automatically from 7/24 to

name	impl	total	time	proof	old proof	old time	caper total	caper proof
arc (Rust Language, 2021)	18	53	0:13	0	7	0:10	70	1
bag_stack (Treiber, 1986)	29	119	0:25	38	36	0:17	70	0
barrier	58	183	16:30	7	38	13:22	102	0
barrier_client	58	150	1:10	21	44	0:50	189	0
bounded_counter	20	72	0:17	6	7	0:11	50	2
cas_counter	14	56	0:11	0	0	0:08	40	0
cas_counter_client	16	36	0:07	0	0	0:06	94	0
clh_lock (Magnusson et al., 1994)	30	86	0:34	0	3	0:22		
fork_join	14	57	0:09	0	0	0:08	38	0
fork_join_client	13	30	0:04	0	0	0:04	70	0
inc_dec	23	78	0:44	0	0	0:31	54	0
lclist (Vafeiadis, 2008; Calcagno et al., 2007)	28	83	0:42	15	18	0:27		
lclist_extra	119	187	2:29	7	2	1:31		
mcs_lock (Mellor-Crummey and Scott, 1991)	54	131	1:38	0	11	1:11		
msc_queue (Michael and Scott, 1996)	36	156	2:54	43	46	1:42		
peterson (Peterson, 1981)	46	164	14:19	25	28	7:51		
queue	42	163	1:42	46	46	1:17	99	0
rwlock_duolock (Courtois et al., 1971)	45	101	0:23	0	10	0:21		
rwlock_lockless_faa	27	74	0:34	0	1	0:20	68	1
rwlock_ticket_bounded	40	117	1:15	5	12	0:54		
rwlock_ticket_unbounded	38	111	0:25	0	5	0:21		
spin_lock	13	59	0:08	0	0	0:06	39	0
ticket_lock	23	84	0:32	0	6	0:23	59	0
ticket_lock_client	18	39	0:07	0	0	0:06	79	0
total	822	2389	47:24	213	320	32:30	1121	4

Figure 4.9: Data on verified examples. Rows correspond to files in the supplementary material. Columns show number of lines of *impl*ementation of the program, lines in *total*, and lines of *proof* burden. The time column displays the average verification time in minutes:seconds. Proof burden for the *old* proof burden of Diaframe is also shown, as well as the proof burden of Caper. **Bolded** examples have reduced proof burden.

14/24. Larger examples tend to have a larger reduction of proof burden: for example, the proof burden of ARC (18 lines of implementation) is reduced by 7 lines, while that of the barrier (58 lines of implementation) is reduced by 31 lines. The remaining proof burden is due to existing orthogonal problems with Diaframe's automation (its poor support for recursive representation predicates, and its limited solver for pure goals). The verification time over all examples did go up by around 50%, from 32 to 47 minutes. We believe the increased verification time is acceptable, as for 22/24 examples the verification time remains below 3 minutes. Exceptions are the barrier and Peterson (1981)'s algorithm. Both of these examples feature invariants with n-ary disjunctions, where $n \geq 10$. We believe the slowdown is a side-effect of our proper support for disjunctions. In the original Diaframe, disjunctions in invariants were 'opaque', whereas all the atoms inside a disjunction are now checked for ground connections to a goal. The verification of Peterson's algorithm suffers an additional slowdown due to the use of backtracking, which we will discuss shortly.

Comparison with Caper. For most examples, Caper (Dinsdale-Young et al., 2017) still has the lowest proof burden. This is due to aforementioned orthogonal problems with Diaframe (recursive representation predicates and pure goals). Since Caper has superior support for recursive representation predicates, it can verify the queue and the Treiber (1986) stack without user guidance. Caper also employs SMT solvers as trusted oracles, making it better at solving pure sideconditions. However, as noted by Windsor et al. (2017), Caper likely cannot handle examples such as the CLH lock (Magnusson et al., 1994) and the MCS lock (Mellor-Crummey and Scott, 1991), making our work the first that fully automatically verifies these examples. Additionally, for both the verification of the ARC from § 4.2.2 and an `FAA`-based readers-writer lock, we outperform Caper. Precisely these verifications feature the problematic disjunction pattern that motivated this work.

Backtracking. Caper, the original Diaframe, and our fork of Diaframe, use some form of backtracking. We distinguish between *local backtracking* (*i.e.,* determining an appropriate rule to apply) and *global backtracking* (*i.e.,* going back to try a different rule, if verification fails). Caper uses global backtracking for all its examples. As noted by Wolf et al. (2021), this causes unstable verification times for failing verification attempts. The original Diaframe uses local backtracking for all its examples, and global backtracking in 12/24 examples. Although the verification times were stable when benchmarking failing examples in the original Diaframe, global backtracking still makes it harder to debug failing verification attempts. In our fork of Diaframe, the verification times of these failing examples are still stable, yet global backtracking is only required for 1/24 examples: Peterson's algorithm. The verification of this example also suffers from the biggest slowdown. It is worth noting that the proofs of the MCS lock (Mellor-Crummey and Scott, 1991) and the CLH lock (Magnusson et al., 1994) involve goals with 4 invariants in the proof context, but due to our connection-based approach, no global backtracking is needed to determine which invariant to access. The verification times for these examples are well below 2 minutes.

Ground connections. The verifications use Diaframe's 5 existing hint libraries for ghost resources, to which we added a modest amount of 3 ground connections (*i.e.,* disjunctive patterns in ghost-state reasoning that were not expressible in the original

Diaframe). Namely, the rules for the pure and token ground connections from §4.5.3, and an additional ground connection for the ticket ghost theory that is used in the verification of the ticket lock and barrier.

Coarse-grained concurrency. Although we target verification of programs with fine-grained concurrency, our techniques are also applicable for coarse-grained concurrency. In general, the coarse-grained concurrent setting is easier: invariants in fine-grained concurrency need to be established after every program step, while (lock) invariants in coarse-grained concurrency only need to be established when the lock is released. Our benchmark includes three examples with coarse-grained concurrency: lclist, lclist_extra and rwlock_duolock. These verifications have been carried out in a modular fashion, on top of independently verified lock specifications.

4.8 Related Work

Focusing and connection calculi. Focusing (Andreoli, 1992; Simmons, 2014; Liang and Miller, 2009) reduces the search space of proofs by limiting backtracking to the choice of focus. Focusing by itself does not directly address the issue of the irrelevance of rule applications, in particular of unnecessary disjunction elimination. Ordinarily, focusing stops at a disjunction and switches to the inversion phase in which the disjunction is eagerly split. There is no explicit mechanism to limit the choice of focus to formulas whose decomposition contributes to the final derivation.

Connection calculi (Waaler, 2001; Wallen, 1990; Otten and Kreitz, 1995) address both the issues of non-permutability and irrelevance with connection-guided proof search. They identify a connection first, then in essence directly reduce the sequent to obtain the initial sequent (Wallen, 1990, §3). Somewhat similarly, our approach relies on identifying a *connection* between a positive atom occurrence on the right and a (unifiable) atom occurring in the target of a hypothesis. Our connection-based method is a simplified version of the original concept, retaining only a general analogy. Connection calculi develop this basic idea much further and achieve completeness for specific logics, such as intuitionistic propositional or first-order logic. The cost is substantial complication of the systems and difficulty in transporting the idea to other frameworks. The calculus from §4.3 could be made complete by labeling the formulas with intuitionistic prefixes like in the free variable calculi (Waaler, 2001, §4) related to connection calculi. The prefixes would describe where each formula occurrence "originates from", so the implication-introduction rule does not need to "forget" the other disjuncts: the prefixes enforce that hypotheses can be used only for "their" disjuncts.

Sequent calculi for BI. Separation logic may be seen as a particular theory in (a variant of) the logic of Bunched Implications (BI), which was introduced in proof-theoretic sequent-calculus formulation by O'Hearn and Pym (1999) and Pym (2002). A focused sequent calculus for BI was devised and shown complete by Gheorghiu and Marin (2021). They handle disjunction on the left using an unfocused left-elimination rule, which is not connection-driven nor goal-directed in the sense of connection calculi. There exist complete connection calculi for propositional BI (Galmiche and Méry, 2002) and multiplicative intuitionistic linear logic (Galmiche and Méry, 2018), but it is unclear if they can be extended to more complex logics such as concurrent separation logic, nor if

they can be implemented efficiently in a general-purpose proof assistant such as Coq.

First-order/symbolic heap separation logic solvers. The literature provides various complete solvers for fragments of separation logic (Piskac et al., 2014b; Reynolds et al., 2016; Lee and Park, 2014), also with inductive representation predicates (Le et al., 2018; Piskac et al., 2014a). Proper support for inductive representation predicates is lacking in our work. The mentioned solvers enjoy excellent proof automation on their fragments, but none of their fragments cover all the connectives we consider—namely, the combination of magic wand, disjunction and quantifiers.

Fine-grained concurrency. Voila (Wolf et al., 2021) and Starling (Windsor et al., 2017) are other tools for semi-automated verification of fine-grained concurrent programs. Both are *proof outline checkers* and require annotations before and after each program instruction—these annotations ensure the required case distinctions are made. Such annotations are not required in our work. Voila can, however, prove the stronger notion of *logical atomicity*, which was added to Diaframe recently (Chapter 3). A notable exception to the lower proof burden is in the verification of Peterson's algorithm, where Starling's constraint-based approach seems a better fit.

Chapter 5

Unification for Subformula Linking under Quantifiers

5.1 Introduction

Suppose you are faced with the following proof obligation:

$$(\forall x.\, Px \to \exists y.\, Qxy) \vdash \exists y.\, \exists z.\, Q(fz)y. \tag{5.1}$$

What is the easiest simpler proof obligation with which you could prove this entailment? After some inspection, one can see that $\exists z.\, P(fz)$ suffices. But how does one compute this in general, given a single hypothesis and goal? In particular, interdependencies of variables can be quite challenging.

We follow Chaudhuri (2013) and call this problem *subformula linking*. Subformula linking is used for recent work on 'gestural' theorem proving for intuitionistic first-order logic, which continues work on proof by pointing by Bertot et al. (1994). One gestural prover is ProfInt (Chaudhuri, 2021, 2023), where one can link two subformulas by selecting them with a mouseclick. Another gestural prover is Actema (Donato et al., 2022), where one can drag and drop a hypothesis on a goal, prompting the system to link subformulas, and compute a remaining proof obligation.

A variant of subformula linking also shows up in other logics. The *framing* problem (Berdine et al., 2005; Kassios, 2006) in separation logic is about canceling occurrences of the same atom in the hypothesis and goal. For example, we may wish to cancel out (or 'frame') the atom R in $R * Q \vdash \exists x.\, S\, x * R$, and continue by proving $Q \vdash \exists x.\, S\, x$. The Iris framework for concurrent separation logic in Coq (Jung et al., 2015, 2016; Krebbers et al., 2017b; Jung et al., 2018b; Krebbers et al., 2017a, 2018) has a tactic called `iFrame`, which can perform the above framing. The implementation of this tactic essentially solves a subformula linking problem.

The previous examples all originate from interactive theorem proving. However, subformula linking is also useful in the setting of automated theorem proving. Diaframe—a recent tool for automated proofs of concurrent programs using Iris in Coq, presented in Chapters 2 to 4—also makes (implicit) use of subformula linking. Consider a (slightly

simplified)[1] verification goal in Diaframe that occurs in the verification of Courtois et al. (1971)'s classic readers-writer lock:

$$(\forall R.\ R \mathrel{-\!\!*} \exists \gamma.\ \mathsf{is\_lock}\ \gamma\ v\ R) \vdash \exists \gamma_1 \gamma_2.\ \mathsf{is\_lock}\ \gamma_1\ v\ (S\ \gamma_2). \tag{5.2}$$

This entailment is similar to the previous example, but it uses higher-order quantification and the magic wand ($\mathrel{-\!\!*}$), which is the substructural version of implication ($\to$) in separation logic. The is_lock predicate (Hobor et al., 2008; Dinsdale-Young et al., 2010) states that value v is a lock with name γ, and guards resource R—although the precise semantics does not matter for now: it should be viewed as an abstract predicate. The key characteristic of Equation (5.2) is that we want to simplify it to $\exists \gamma_2.\ S\ \gamma_2$, a goal that Diaframe can prove automatically.

Unfortunately, previous work on subformula linking does not produce satisfactory solutions for these examples. In the realm of first-order logic, Actema (Donato et al., 2022) cannot establish a link for Equation (5.1) and thus fails. While ProfInt (Chaudhuri, 2021, 2023) can establish links, there are many candidate links, and not all of them are provable (*i.e.,* some are equivalent to $\bot$). We could backtrack on all candidate links, but that would be detrimental for performance when applied to proof automation for concurrent programs (as argued in Chapter 4). In the realm of separation logic, the examples are simply out of scope of Iris's `iFrame` because it does not consider subterms of the hypothesis.

Quantifiers pose problems for existing approaches to subformula linking. To understand why, we briefly discuss the setup of existing approaches for linking under quantifiers. ProfInt and Actema have recursive procedures that traverse the hypothesis and goal to find a shared atom. When both the hypothesis and goal are a logical connective (*i.e.,* not an atom), one needs to choose to either proceed in the hypothesis or goal. While this choice is unspecified in the mathematical presentation of these systems (as a non-deterministic inductive relation), an implementation needs to make a concrete choice. This choice matters—just like it matters in which order the ordinary proof rules for introduction and elimination are used—and might result in finding different links, or no links at all. As the paper on ProfInt (Chaudhuri, 2021, §2.1) remarks, this is a challenging problem:

> Resolving this ambiguity is going to be as hard as fully automated proof search, which will therefore not be recursively solvable as soon as we introduce quantifiers.

Nonetheless, to make subformula linking usable—for example to develop better approaches for automated program verification—one should try to rule out as many useless or blatantly false linkings. We describe how ProfInt and Actema deal with this problem in the context of quantifiers, their limitations, and how we address these.

ProfInt establishes unwanted links due to scoping issues. ProfInt (Chaudhuri, 2021) links atoms by posing *equality* constraints. Syntactically equal predicates are linked by requiring the user to prove that all their arguments are equal, *e.g.,* $P\ x \vdash P\ y$ will be

[1] Iris's update modality $\Rrightarrow$ is omitted from Equation (5.2) and the remaining proof obligation $\Rrightarrow \exists \gamma_2.\ S\ \gamma_2$. Because $\Rrightarrow S\ \gamma_2'$ is not provable for any constant γ_2', it is crucial that we keep the existential quantification, and do not prematurely instantiate γ_2 with an evar.

reduced to $x \doteq y$. Although this reduction seems innocent, it means that ProfInt can establish links under quantifiers regardless of whether the order of traversal respects variable scoping. For example, ProfInt can establish two links for $\exists x.\, P\, x \vdash \exists y.\, P\, y$. The first link produces the tautology $\forall x.\, \exists y.\, x \doteq y$ as its simplification. The second link produces $\exists y.\, \forall x.\, x \doteq y$, which is logically equivalent to false (if the domain of quantification is non-trivial). To use subformula linking in automated theorem proving, it would be helpful if such unwanted links are simply ruled out altogether. In particular, if there are no sensible links at all, this means that the automation can immediately move on to the next hypothesis without having to backtrack.

Actema cannot establish desired links. Actema (Donato et al., 2022) rules out some of ProfInt's unwanted links, but it actually rules out too many links. Actema uses *unification* to determine the appropriate order to traverse below quantifiers. If two atoms are not unifiable, they cannot be linked: $P\, x \vdash P\, y$ will thus not be reduced to $x \doteq y$ if x and y are different variables. When linking the previous example $\exists x.\, P\, x \vdash \exists y.\, P\, y$, Actema will unify y on the right-hand side (*i.e.,* introduce the $\exists$ in the goal) with the x obtained from the left-hand side (*i.e.,* by eliminating the $\exists$ in the hypothesis). This can *only* be done if the $\exists$ in the hypothesis is eliminated first—which rules out the unwanted/ill-scoped linking ProfInt finds.

Actema uses unification whenever quantifiers need to be instantiated with a specific term (*i.e.,* $\forall$-quantifiers in hypotheses/the 'left', and $\exists$-quantifiers in goals/the 'right'). For example, Actema finds that x in Equation (5.1) must be of shape $f\, ?z$ for some unknown z, since this causes the arguments of Q to match syntactically. Actema has two rules available to derive a link: one for when x is unified with a concrete term t, and one for when x does not get unified at all. Unfortunately, in Equation (5.1), neither rule is applicable—although x has been unified with $f\, ?z$, it contains the uninstantiated $?z$, meaning $f\, ?z$ is not a concrete term. As such, Actema cannot establish a link for Equation (5.1).

Our approach: Quantifying on the Uninstantiated. We propose a new system called **QU** for linking subformulas under quantifiers. Like Actema, we use unification to rule out unwanted links due to scoping errors. However, QU improves upon Actema by being able to link subformulas with non-trivial quantifier instantiation, such as the examples in this section. Our approach is to *Quantify on the Uninstantiated* (QU). Consider Equation (5.1), where the x gets unified with $f\, ?z$, and $?z$ is uninstantiated. QU quantifies precisely on this z, *i.e.,* we produce the simplification $\exists z.\, P(f z)$. On the implementation level, we use evars (existential variables) and convert these back into existential quantifiers.

Anti goals. QU does not specify if hypothesis-rules ('left' rules) or goal-rules ('right' rules) should be given priority when mixing quantifiers with the propositional connectives (conjunction, disjunction, implication)—this remains an open problem in general subformula linking. Both the Actema (Donato et al., 2022) and ProfInt (Chaudhuri, 2023) implementation use heuristics and/or backtracking to make this choice, and so do we. QU nevertheless helps in eliminating unwanted links from the search space.

Applications. Quantifiers are problematic for subformula linking regardless of the logic of the system—*i.e.,* first-order, higher-order, and separation logic systems face essentially the same problem. The idea of QU is not tied to a specific logic, however. To

demonstrate this, we use QU to improve Iris's `iFrame` tactic for higher-order separation logic—making it strictly more powerful, with comparable performance. We also explain how Diaframe makes use of QU.

Contributions and artifacts.

- We present the QU rules for linking subformulas under quantifiers (§ 5.3). We formally prove in Coq that our system lies between Actema and ProfInt, using a deep embedding of first-order logic by Kirst et al. (2022).
- We present a simple, shallowly embedded subformula linking procedure in Coq (§ 5.4). This demonstrates that QU can be implemented and used inside Coq.
- We extend and improve Iris's `iFrame` tactic with the QU rules (§5.5.1), demonstrating the practical applicability of our approach.
- We describe how Diaframe uses the QU rules to verify Courtois et al. (1971)'s readers-writer lock (§ 5.5.2).

We start with some background on subformula linking (§ 5.2). We conclude with an evaluation of the improved `iFrame` (§ 5.6) and a discussion of related work (§ 5.7). The Coq sources of these artifacts are in the supplementary material (Mulder and Krebbers, 2023b).

5.2 Background on Subformula Linking

To provide background on subformula linking and to compare existing systems, we give a uniform presentation of subformula linking (§ 5.2.1), and formulate ProfInt (§ 5.2.2) and Actema (§ 5.2.3) as instances.[2] Although many rules are shared by ProfInt and Actema, their differences are significant for the links they can derive. We give examples where ProfInt establishes unwanted links due to scoping issues with quantifiers, and where Actema cannot establish desired links, before presenting our system QU (§ 5.3).

5.2.1 Subformula Linking Judgment

We consider first-order intuitionistic logic with equality (our implementations in § 5.4 and 5.5 scale to higher-order separation logic). Terms, atoms, propositions, and proof contexts are inductively defined as:

$$\begin{aligned} t, s &::= x \mid f\vec{t} \\ A &::= \bot \mid \top \mid P\vec{t} \mid t \doteq s \\ H, G, O &::= A \mid H \wedge H \mid H \vee H \mid H \to H \mid \forall x.\, H \mid \exists x.\, H \\ \Delta &::= \cdot \mid H, \Delta \end{aligned}$$

Predicates P and functions f have an arity n and take a list of terms of length n. For two lists $\vec{t} = t_1, \ldots, t_n$ and $\vec{s} = s_1, \ldots, t_n$ of the same length, we will write $\vec{t} \doteq \vec{s}$ for $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$. If both lists are empty, $\vec{t} \doteq \vec{s}$ is just $\top$.

[2] We discuss differences between the original formulation of ProfInt and Actema and our unified presentation in §5.7.

$$\mathrm{L}\wedge\ \frac{H_i \wedge [O] \Vdash G \qquad i \in \{1,2\}}{(H_1 \wedge H_2) \wedge [O] \Vdash G} \qquad \mathrm{L}\vee\ \frac{H_i \wedge [O] \Vdash G \qquad i \in \{1,2\}}{(H_1 \vee H_2) \wedge [O \wedge (H_{3-i} \to G)] \Vdash G}$$

$$\mathrm{L}\to\ \frac{H_2 \wedge [O] \Vdash G}{(H_1 \to H_2) \wedge [H_1 \wedge O] \Vdash G} \qquad \mathrm{L}\exists\ \frac{\forall x. \quad H \wedge [O] \Vdash G}{(\exists x. H) \wedge [\forall x. O] \Vdash G} \qquad \mathrm{L}\forall\ \frac{\forall x. \quad H \wedge [O] \Vdash G}{(\forall x. H) \wedge [\exists x. O] \Vdash G}$$

$$\mathrm{R}\wedge\ \frac{H \wedge [O] \Vdash G_i \qquad i \in \{1,2\}}{H \wedge [O \wedge G_{3-i}] \Vdash G_1 \wedge G_2} \qquad \mathrm{R}\vee\ \frac{H \wedge [O] \Vdash G_i \qquad i \in \{1,2\}}{H \wedge [G_{3-i} \vee O] \Vdash G_1 \vee G_2}$$

$$\mathrm{R}\to\ \frac{H \wedge [O] \Vdash G_2}{H \wedge [G_1 \to O] \Vdash G_1 \to G_2} \qquad \mathrm{R}\forall\ \frac{\forall x. \quad H \wedge [O] \Vdash G}{H \wedge [\forall x. O] \Vdash \forall x. G} \qquad \mathrm{R}\exists\ \frac{\forall x. \quad H \wedge [O] \Vdash G}{H \wedge [\exists x. O] \Vdash \exists x. G}$$

(a) Rules that are shared by Actema and ProfInt.

$$\textsc{Asmp-actema}\ \frac{}{A \wedge [\top] \Vdash A}$$

$$\textsc{Cong-profint}\ \frac{}{P\vec{t} \wedge \left[\vec{t} \doteq \vec{s}\right] \Vdash P\vec{s}}$$

$$\mathrm{L}\forall_n\textsc{-actema}\ \frac{H[t/x] \wedge [O] \Vdash G}{(\forall x. H) \wedge [O] \Vdash G} \qquad \mathrm{R}\exists_n\textsc{-actema}\ \frac{H \wedge [O] \Vdash G[t/x]}{H \wedge [O] \Vdash \exists x. G}$$

(b) Subformula rules specific to ProfInt.

(c) Subformula rules specific to Actema.

Figure 5.1: Subformula rules in ProfInt and Actema.

We interpret the judgment $\Delta \vdash G$ as $\bigwedge_{H \in \Delta} H \vdash G$, where $H \vdash G$ is inductively defined using the usual rules for introduction and elimination of first-order intuitionistic logic.

To provide a uniform formulation of subformula linking, we consider the *linking judgment* $H \wedge [O] \Vdash G$, which says that given a *hypothesis* H and *goal* G, it suffices to prove the *simplification* O instead of G.

We call a derivation of $H \wedge [O] \Vdash G$ a *linkage*. Each linkage should satisfy $H, O \vdash G$, which trivially gives us the following derivable inference rule:

$$\textsc{link-apply}\ \frac{H \in \Delta \qquad H \wedge [O] \Vdash G \qquad \Delta \vdash O}{\Delta \vdash G}$$

In ProfInt and Actema, the user initiates this rule graphically by pointing out the common subformulas in H and G, or by dragging and dropping. ProfInt and Actema are then responsible for automatically finding a linkage with simplification O, allowing the user to continue with obligation $\Delta \vdash O$.

The inference rules for establishing $H \wedge [O] \Vdash G$ in Profint and Actema will be given as inductively-defined relations in § 5.2.2 and 5.2.3. These relations should be interpreted

as (non-deterministic) recursive 'procedures' that compute an appropriate O for a given H and G. That is, the rules will follow the structure of the hypothesis H and goal G. We use the convention to put 'outputs' of relations, such as O, between brackets [and]; other parameters are 'inputs'.

5.2.2 Rules for ProfInt

The rules for ProfInt's linking judgment $H \curlywedge [O] \Vdash G$ are given in Figures 5.1a and 5.1b. Their purpose is to find a shared atom in H and G that can be linked. The base case is Cong-profint in Figure 5.1b. This rule applies if the hypothesis and goal are the same predicate P, resulting in a simplification that says their arguments should be equal.

The recursive rules in Figure 5.1a traverse the formula in a non-deterministic fashion to reach the base case. They come in two categories: 'left' rules for the hypothesis H and 'right' rules for the goal G. All rules compute a simplification O based on the simplification for the recursive call.

If the hypothesis is a conjunction $H_1 \wedge H_2$, rule L$\wedge$ proceeds in either H_1 or H_2 and leaves the simplification O unchanged. If the hypothesis is a disjunction $H_1 \vee H_2$, rule L$\vee$ again proceeds in either H_1 or H_2 but adds the conjunct $H_1 \to G$ or $H_2 \to G$ to the simplification O in order to account for the other disjunct. If the hypothesis is an implication $H_1 \to H_2$, rule L$\to$ adds the premise H_1 as a conjunct to the simplification O. If the hypothesis is a quantifier, rules L$\exists$ and L$\forall$ proceed under the quantifier. They add the opposite quantifier to the output O, since we are in a negative position. The 'right' rules are mostly dual to the 'left' rules.

Let us demonstrate these rules on an example. Suppose we have the hypothesis $(A \to (B \wedge C))$ and goal $B \wedge D$, where A, B, C and D are atoms (0-ary predicates), then:

$$\mathrm{L}{\to}\ \dfrac{\mathrm{R}{\wedge_1}\ \dfrac{\mathrm{L}{\wedge_1}\ \dfrac{\textsc{Cong-profint}\ \dfrac{}{B \curlywedge [\top] \Vdash B}}{B \curlywedge [\top \wedge D] \Vdash B \wedge D}}{(B \wedge C) \curlywedge [\top \wedge D] \Vdash B \wedge D}}{A \to (B \wedge C) \curlywedge [A \wedge (\top \wedge D)] \Vdash B \wedge D}$$

When using ProfInt's implementation (Chaudhuri, 2023), the user does not have to construct this derivation by hand. Instead, the user clicks on the occurrences of B in H and G. This click instructs the implementation to derive the linking judgment, and to transform goal $B \wedge D$ into $A \wedge (\top \wedge D)$ with link-apply. (Both ProfInt and our implementation in §5.5 remove the superfluous occurrence of $\top$, *i.e.,* give $A \wedge D$. We ignore these simplifications for brevity's sake.) This example shows that by clicking on the occurrences of B in H and G, ProfInt essentially eliminates an implication and a conjunction.

Non-determinism. Linking is non-deterministic, *i.e.,* for the same hypothesis H and goal G one can find different simplifications O_1 and O_2 with $H \curlywedge [O_1] \Vdash G$ and $H \curlywedge [O_2] \Vdash G$. In the above derivation, we could have used R$\wedge_1$ first, resulting in the simplification $((A \wedge \top) \wedge D)$. In this case, the choice is immaterial, since the simplifications are equiderivable, *i.e.,* $(A \wedge \top) \wedge D \dashv\vdash A \wedge (\top \wedge D)$.

In general, the rule order matters. Consider finding an O with $(A \to B) \curlywedge [O] \Vdash (A \to B) \vee C$. By prioritizing the 'left' rules, we find $O_1 = A \wedge (C \vee (A \to \top))$ using L$\to$,

R$\vee_1$, R$\to$. By prioritizing the right rules, we find $O_2 = C \vee (A \to (A \wedge \top))$ using R$\vee_1$, R$\to$, L$\to$. The first simplification results in information loss: O_1 is equivalent to A. The second simplification O_2 is equivalent to $\top$, and thus desired.

Picking the right order of rules is non-trivial. In this example we see that R$\vee$ (disjunction introduction) should take priority over L$\to$ (implication elimination), but this does not hold in general. For many examples, one also wants to prioritize L$\vee$ (disjunction elimination) over R$\vee$ (disjunction introduction)—but what if the disjunction to eliminate resides in the conclusion of an implication? (For example, consider $((A \to A) \to (B \vee C)) \wedge [O] \Vdash B \vee C$.)

The implementations of Profint, Actema and QU use heuristics to determine the rule order. Our goal is not to improve these heuristics, but to design rules for quantifiers that exclude linkages that are blatantly false due to scoping issues.

Problem: Profint establishes unwanted links due to scoping issues. Suppose we want to construct a linkage $(\forall x.\, \exists y.\, Qxy) \wedge [O] \Vdash \exists z.\, Qtz$. Using the following derivation we find $O = \exists x.\, \forall y.\, \exists z.\, x \doteq t \wedge y \doteq z$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall z.\quad Qxy \wedge [x \doteq t \wedge y \doteq z] \Vdash Qtz}\,\textsc{Cong-profint}}{\forall y.\quad Qxy \wedge [\exists z.\, x \doteq t \wedge y \doteq z] \Vdash \exists z.\, Qtz}\,\mathrm{R}\exists}{\forall x.\quad (\exists y.\, Qxy) \wedge [\forall y.\, \exists z.\, x \doteq t \wedge y \doteq z] \Vdash \exists z.\, Qtz}\,\mathrm{L}\exists}{(\forall x.\, \exists y.\, Qxy) \wedge [\exists x.\, \forall y.\, \exists z.\, x \doteq t \wedge y \doteq z] \Vdash \exists z.\, Qtz}\,\mathrm{L}\forall$$

This is the desired simplification, since it is a tautology (pick $x = t$ and $z = y$). In fact, ProfInt has additional simplification rules that can reduce O to just $\top$.

Unfortunately, the heuristics of ProfInt's implementation prioritize the 'right' rules, resulting in:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall y.\quad Qxy \wedge [x \doteq t \wedge y \doteq z] \Vdash Qtz}\,\textsc{Cong-profint}}{\forall x.\quad (\exists y.\, Qxy) \wedge [\forall y.\, x \doteq t \wedge y \doteq z] \Vdash Qtz}\,\mathrm{L}\exists}{\forall z.\quad (\forall x.\, \exists y.\, Qxy) \wedge [\exists x.\, \forall y.\, x \doteq t \wedge y \doteq z] \Vdash Qtz}\,\mathrm{L}\forall}{(\forall x.\, \exists y.\, Qxy) \wedge [\exists z.\, \exists x.\, \forall y.\, x \doteq t \wedge y \doteq z] \Vdash \exists z.\, Qtz}\,\mathrm{R}\exists$$

This is problematic, since this simplification is logically equivalent to $\bot$ (assuming the domain is non-trivial). There is no way we could pick a z that is equal to every y. This is the result of a blatant scoping issue: we have mistakenly used existential introduction (R$\exists$) before existential elimination (L$\exists$). We would like this derivation to be ruled out.

5.2.3 Rules for Actema

Actema takes a different approach for linking subformulas under quantifiers than Profint. This approach avoids scoping issues, but results in a failure to find other desired links.

The rules for Actema's linking judgment $H \wedge [O] \Vdash G$ are given in Figures 5.1a and 5.1c. Instead of generating a simplification that involves equalities, Actema uses *unification* to determine appropriate ways to eliminate universal quantifiers, and introduce existential quantifiers. The base case Asmp-actema requires the atoms to match *exactly*.

This requirement can be met under quantifiers since Actema has the rules L$\forall_n$-ACTEMA and R$\exists_n$-ACTEMA. The premises of these rules allow one to instantiate the quantifier with a specific term t, which we are free to choose. For example, Actema can directly find $P\,t \land [\top] \vDash \exists x.\, P\,x$ with R$\exists_n$-ACTEMA and ASMP-ACTEMA by instantiating x with t.

This begs the question: how does one automatically find appropriate terms for R$\exists_n$-ACTEMA? This can be done using *evars* (existential variables), which we denote as $?t$. Instead of choosing a concrete term t in R$\exists_n$-ACTEMA upfront, evars allow one to postpone this choice. As soon as we learn an appropriate concrete term s for $?t$, we instantiate $?t$ with s—and the derivation behaves as if we had chosen s all along.

The ASMP-ACTEMA rule prompts appropriate instantiations of evars. Crucially, an evar $?t$ can only be instantiated with term s if the variables mentioned by s were in scope when $?t$ was created—instantiation fails otherwise. Such failures are crucial for determining an appropriate rule order. It means that ill-scoped linkages are ruled out by construction.

Let us reconsider the example of finding a linkage $(\forall x.\, \exists y.\, Qxy) \land [O] \vDash \exists z.\, Qtz$ from § 5.2.2. If one were to start with R$\exists_n$-ACTEMA, one needs to instantiate the evar $?z$ with a variable y, which is not in scope when R$\exists_n$-ACTEMA was used. This fails, prompting Actema to try the correct rule order, *i.e.,* prioritizing the 'left' rules, resulting in the desired simplification $O = \forall y.\, \top$.

Unification guides Actema in the search for an appropriate rule order. In some cases, unification rules out a bad linkage completely. Consider $(\forall x.\, \exists y.\, R\,x\,y) \land [O] \vDash \exists z.\, \forall x'.\, R\,x'\,z$. ProfInt would propose an unprovable simplification O, while Actema fails to establish a linkage. From a proof automation point of view (*e.g.,* for our applications in § 5.5), Actema's behavior of failing instead of finding an unprovable linkage is preferable. It prompts the automation to consider another hypothesis for finding a linkage.

Problem: Actema cannot establish desired links. We have seen that unification allows Actema to rule out inappropriate linkages. Unfortunately, it rules out linkages too aggressively—some linkages that one might expect to obtain are also ruled out. This can happen when the order of related quantifiers in hypothesis and goal do not match, like in:

$$(\forall x.\, Px \to \exists y.\, Qxy) \land [O] \vDash \exists z.\, \exists x'.\, Q(f\,x')\,z \tag{5.3}$$

One would expect to find $O = \exists x.\, P(f\,x)$ for Equation (5.3), but this linkage is not derivable in Actema, and neither is any other O. To see why, note that the desired z in the goal is obtained from the existentially quantified y in the hypothesis. This means we must start with a 'left' rule for the universal quantifier, choosing between L$\forall_n$-ACTEMA and L$\forall$. A linkage does not exist for every x, so L$\forall$ fails. We would like to use L$\forall_n$-ACTEMA with some $t = f\,?x'$, but no appropriate instance for $?x'$ is in scope. We can only get access to such an instance by first using R$\exists$ twice—which means we have a circular dependency.

Note that ProfInt is able to derive a linkage for this example. However, similar to the example in the previous section, it might use rules in the wrong order and find unprovable simplifications O (*i.e.,* a simplification that is logically equivalent to $\bot$ if the domain is non-trivial).

$$\mathrm{L}\forall_{QU}(\vec{y} = [u], t = fu)\ \dfrac{\forall u.\ \dfrac{\mathrm{L}\to\ \dfrac{\mathrm{L}\exists\ \dfrac{\forall y.\ \dfrac{\mathrm{R}\exists_{QU}(\vec{y} = [], t = y)\ \dfrac{\mathrm{R}\exists_{QU}(\vec{y} = [], t = u)\ \dfrac{\textsc{Asmp-actema}}{Q\,(fu)\,y \wedge [\top] \Vdash Q(fu)\,y}}{Q\,(fu)\,y \wedge [\top] \Vdash \exists x'.\, Q(fx')\,y}}{Q\,(fu)\,y \wedge [\top] \Vdash \exists z.\, \exists x'.\, Q(fx')\,z}}{(\exists y.\, Q\,(fu)\,y) \wedge [\forall y.\, \top] \Vdash \exists z.\, \exists x'.\, Q(fx')\,z}}{(P\,(fu) \to \exists y.\, Q\,(fu)\,y) \wedge [P(fu) \wedge \forall y.\, \top] \Vdash \exists z.\, \exists x'.\, Q(fx')\,z}}{(\forall x.\, Px \to \exists y.\, Qxy) \wedge [\exists u.\, P(fu) \wedge \forall y.\, \top] \Vdash \exists z.\, \exists x'.\, Q(fx')\,z}$$

Figure 5.2: An example linkage in QU.

5.3 Quantifying on the Uninstantiated

We present our system Quantifying on the Uninstantiated (QU). Compared to Actema, we do not choose between instantiation or quantification—rather, we quantify precisely on the parts that remain uninstantiated. We start by discussing the rules of QU (§5.3.1) and explain how QU is used on examples (§5.3.2). We finally discuss the proof-theoretic properties of QU (§5.3.3)—we show that the strength of the linkages from QU lies between Actema and ProfInt.

5.3.1 Rules for QU

QU features the rules from Figure 5.1a, except L∀ and R∃. We have Asmp-actema as a base case, and the following rules for existential and universal quantifiers:

$$\mathrm{R}\exists_{QU}\ \frac{\forall \vec{y}.\quad H \wedge [O] \Vdash G[t/x]}{H \wedge [\exists \vec{y}.\, O] \Vdash \exists x.\, G} \qquad \mathrm{L}\forall_{QU}\ \frac{\forall \vec{y}.\quad H[t/x] \wedge [O] \Vdash G}{(\forall x.\, H) \wedge [\exists \vec{y}.\, O] \Vdash G}$$

We write $\vec{y}$ for a (possibly empty) list of variables, and the term t can mention these variables.

To provide an intuition for these rules, let us show that our new rule $\mathrm{L}\forall_{QU}$ generalizes Actema's L∀ and $\mathrm{L}\forall_n$-actema (dually, $\mathrm{R}\exists_{QU}$ generalizes R∃ and $\mathrm{R}\exists_n$-actema):

$$\mathrm{L}\forall\ \frac{\forall x.\quad H \wedge [O] \Vdash G}{(\forall x.\, H) \wedge [\exists x.\, O] \Vdash G} \qquad \mathrm{L}\forall_n\text{-actema}\ \frac{H[t/x] \wedge [O] \Vdash G}{(\forall x.\, H) \wedge [O] \Vdash G}$$

Similar to L∀, the premise of $\mathrm{L}\forall_{QU}$ is quantified. Similar to $\mathrm{L}\forall_n$-actema, we instantiate the quantifier with a term t. We retain the expressivity of Actema. If we take $\vec{y}$ to be the empty list, $\mathrm{L}\forall_{QU}$ reduces directly to $\mathrm{L}\forall_n$-actema. If we take $\vec{y}$ to be the list with just x', and $t = x'$, $\mathrm{L}\forall_{QU}$ is precisely L∀.

The other quantifier rules L∃ and R∀ stay the same, since these correspond to reversible inference rules.

5.3.2 QU by Example

We show how QU goes beyond Actema by deriving the linkage from the example in §5.2.3:

$$(\forall x.\, Px \to \exists y.\, Qxy) \wedge [O] \Vdash \exists z.\, \exists x'.\, Q(fx')\, z$$

The full derivation is included in Figure 5.2. The key step is the use of $\mathrm{L}\forall_{QU}$, where we pick $\vec{y} = [u]$ and $t = fu$. Intuitively, this choice makes the arguments of Q in hypothesis and goal match precisely, so that the Asmp-actema can be applied in the base case. We also use $\mathrm{R}\exists_{QU}$ twice with empty quantifier list (*i.e.,* $\vec{y} = []$), which simplifies to Actema's $\mathrm{R}\exists_n$-actema.

The application of $\mathrm{L}\forall_{QU}$ with $\vec{y} = [u]$ and $t = fu$ in Figure 5.2 is not expressible in Actema, and crucial for getting the desired linkage in this example. However, it is reliant upon somehow making the correct choice for $\vec{y}$ and t. Additionally, so far we have only seen cases where $\vec{y}$ is a list of length at most 1. We will consider another example to demonstrate how we can determine appropriate choices for $\vec{y}$ and t, and that we sometimes need $\vec{y}$ to be a longer list. We will discuss the actual implementation that chooses $\vec{y}$ and t in Coq in §5.4. The example is as follows:

$$(\forall x.\, \exists y.\, Q\, x\, y) \wedge [O] \Vdash \exists z.\, \exists u.\, \exists v.\, Q(g\, u\, v)\, z \tag{5.4}$$

Consider the following partial derivation of a linkage:

$$\mathrm{L}\forall_{QU}\ \frac{\forall \vec{w}.\ \mathrm{L}\exists\ \dfrac{\forall y.\quad Q\ ?t\ y \wedge [\ldots] \Vdash \exists z.\, \exists u.\, \exists v.\, Q(g\, u\, v)\, z}{(\exists y.\, Q\ ?t\ y) \wedge [\ldots] \Vdash \exists z.\, \exists u.\, \exists v.\, Q(g\, u\, v)\, z}}{(\forall x.\, \exists y.\, Q\, x\, y) \wedge [\ldots] \Vdash \exists z.\, \exists u.\, \exists v.\, Q(g\, u\, v)\, z}$$

We have chosen to instantiate t in $\mathrm{L}\forall_{QU}$ with an evar $?t$, delaying the choice for a concrete term. We still have to choose the $\vec{w}$ over which $\mathrm{L}\forall_{QU}$ should quantify. Whatever our choice, the next steps in the derivation of the linkage would be to apply $\mathrm{R}\exists_{QU}$ three times. In the base case, Asmp-actema will produce two unification problems for the arguments of Q, the easy $?z[\vec{w}, y] = y$, and the harder:

$$?t[\vec{w}] = g\ ?u[\vec{w}, y]\ ?v[\vec{w}, y]$$

Here, we write $?s[\vec{x}]$ for an evar $?s$ which has variables $\vec{x}$ in scope. We cannot instantiate $?t$ to be $g\ ?u[\vec{w}, y]\ ?v[\vec{w}, y]$, since the evars $?u$ and $?v$ have y in scope, and $?t$ does not. To proceed, the unification algorithm now *prunes* the term on the right-hand side (Ziliani and Sozeau, 2015, §4.3.1). This comes down to first creating new evars $?u'[\vec{w}]$ and $?v'[\vec{w}]$, then instantiating $?u[\vec{w}, y] = ?u'[\vec{w}]$ and $?v[\vec{w}, y] = ?v'[\vec{w}]$. At that point, the unification algorithm instantiates $?t[\vec{w}] = g\ ?u'[\vec{w}]\ ?v'[\vec{w}]$. Let us return to our derivation, with this $?t$ filled in:

$$\mathrm{L}\forall_{QU}\ \frac{\forall \vec{w}.\quad (\exists y.\, Q\ (g\ ?u'\ ?v')\ y) \wedge [\forall y.\, \top] \Vdash \ldots}{(\forall x.\, \exists y.\, Q\, x\, y) \wedge [\exists \vec{w}.\, \forall y.\, \top] \Vdash \ldots}$$

We now want to *quantify on the uninstantiated.* That is, $\vec{w}$ will contain a variable for each evar that remained uninstantiated in t.[3] Here we pick $\vec{w}$ to be the two-element

[3] A reviewer pointed out that this idea is similar in spirit to the let generalization step in Hindley-Milner type inference. Indeed, type inference for 'let $f := (\lambda x.\, x)$ in e' will infer the type scheme $\forall \alpha.\, \alpha \to \alpha$ for f, by quantifying over all the uninstantiated/free type variables. It would be interesting to find a more formal connection between the two.

list $[u''; v'']$ and instantiate $?u'[u'', v''] = u''$ and $?v'[u'', v''] = v''$. This results in the following derivation:

$$\mathrm{L}\forall_{QU}\ \frac{\forall u'', v''.\quad (\exists y.\, Q\,(g\,u''\,v'')\,y) \wedge [\forall y.\, \top] \Vdash \ldots}{(\forall x.\, \exists y.\, Q\,x\,y) \wedge [\exists u''.\, \exists v''.\, \forall y.\, \top] \Vdash \ldots}$$

We will explain how this can be done automatically in §5.4.3.

5.3.3 Comparison to ProfInt and Actema

We prove some results about the relative strength of Profint, Actema and QU. We have mechanized these results in Coq using the library for first-order logic by Kirst et al. (2022). This library provides a deep embedding of terms, connectives, propositions, and proofs. We inductively define the three linking judgments, which we disambiguate using subscripts. For example, $H \wedge [O] \Vdash_{\text{ACTEMA}} G$ is the inductively-defined linking judgment of Actema.

First, we prove that all linkage systems are sound:

Theorem 5.3.1 (Soundness)

(a) If $H \wedge [O] \Vdash_{\text{ACTEMA}} G$*, then* $H, O \vdash G$*.*

(b) If $H \wedge [O] \Vdash_{\text{PROFINT}} G$*, then* $H, O \vdash G$*.*

(c) If $H \wedge [O] \Vdash_{\text{QU}} G$*, then* $H, O \vdash G$*.*

Next, we prove that all linkages that can be established by Actema, can also be established by our system QU.

Theorem 5.3.2 (Actema vs. QU) *If* $H \wedge [O] \Vdash_{\text{ACTEMA}} G$*, then* $H \wedge [O] \Vdash_{\text{QU}} G$*.*

This theorem holds because the quantifier rules $\mathrm{R}\exists_{QU}$ and $\mathrm{L}\forall_{QU}$ of QU generalize those of Actema (§5.3.1).

Note that this theorem states that Actema linkages are expressible in QU, which does not guarantee that the procedure we informally describe in §5.3.2 actually finds Actema's solution. This would be harder to formalize because it depends on Coq's unification algorithm. We nevertheless think our solutions would agree with Actema. Actema only uses $\mathrm{L}\forall_n$-ACTEMA if it can unify term t with a concrete term. Concrete terms do not contain evars/uninstantiated terms, so we find the same solution. If instead Actema uses $\mathrm{L}\forall$, the term t must have remained an evar, and so our approach chooses precisely that evar to quantify on.

The relation between QU and ProfInt is more difficult to formalize. Simplifications O produced by a ProfInt linkage $H \wedge [O] \Vdash_{\text{PROFINT}} G$ involve equalities, which are absent in the simplifications produced by QU. This means that the linkage systems do not produce syntactically equal simplifications O. To properly relate two linkages with different simplifications, we furthermore need to ensure that ProfInt and QU apply rules in the same order. We will write a superscript p to indicate that a linking judgment is derived by applying the rules in p in order. For example:

$$(A \wedge B) \wedge [\top \wedge C] \Vdash_{\text{PROFINT}}^{[\mathrm{L}\wedge_1; \mathrm{R}\wedge_1]} (A \wedge C).$$

We can then relate the linkages from QU and ProfInt.

Theorem 5.3.3 (QU vs. ProfInt) *Let p be a sequence of linking rules. If $H \wedge [O] \Vdash^{p}_{\text{QU}} G$, then there is a unique O' for which $H \wedge [O'] \Vdash^{p}_{\text{PROFINT}} G$, and additionally $O \vdash O'$.*

In other words, for a given rule order, if QU can derive a linkage, then ProfInt can also derive a linkage. Additionally, QU's simplification O is at least as hard to prove as ProfInt's simplification O'. Another way to read Theorem 5.3.3 is that the instantiations made by QU are guaranteed to satisfy the equalities from ProfInt.

However, we would rather have something stronger: that the simplifications produced by QU are not harder than those produced by ProfInt, *i.e.*, $O \dashv\vdash O'$. When $\text{L}\forall_{QU}$ and $\text{R}\exists_{QU}$ are used as intended (*i.e.*, they quantify *precisely* on the uninstantiated terms), we conjecture that this is indeed the case. The way that $\text{L}\forall_{QU}$ is currently stated does not guarantee this. Indeed, by picking a particular constant for the term t, we could derive a linkage that is too specialized, and thus harder to prove. Formally proving this conjecture would require a verified unification algorithm for the deeply embedded logic we consider, and a corresponding restriction on the terms t in $\text{L}\forall_{QU}$. We leave this endeavor for future work.

Completeness. Chaudhuri (2021) showed that the full ProfInt linkage system is complete. The full ProfInt system differs from the presentation in §5.2.2 in two regards: one can link two hypotheses, and one can also link within formulas. Subformula linking within formulas allows ProfInt to prove entailments such as

$$A \vee B \vdash ((A \to \bot) \to \bot) \vee ((B \to \bot) \to \bot)$$

since $A \to \bot$ and $\bot$ can be linked *within* the left disjunct. This entailment is out of reach for our linkage system, since no subformula of the hypothesis $A \vee B$ can be linked to a subformula of the goal (note that our system only considers subformulas of the right-hand side of an implication). This incompleteness is acceptable for the application we have in mind, *i.e.*, proof automation for (higher-order) separation logic—implications do not frequently occur inside disjunctions in this setting.

5.4 Implementation

We demonstrate that QU can be effectively implemented in the Coq proof assistant. We start by defining linkages and linkage rules for the propositions `Prop` of Coq's higher-order logic (§ 5.4.1). (This makes it different from § 5.3.3 where we performed meta-theoretic reasoning on a deep embedding of first-order logic.) We then define simple *telescopes* (§ 5.4.2), which form a building block for representing n-ary functions and n-ary quantification. Telescopes allow us to state the $\text{L}\forall_{QU}$ and $\text{R}\exists_{QU}$ rules properly and to implement custom Ltac (Delahaye, 2000) code that assists in solving the unification problems arising in QU (§5.4.3). Finally, we make the LINK-APPLY inference rule available as a Coq tactic (§5.4.4).

5.4.1 Linkages in Coq

We start by defining linkages *semantically* in Coq:

```
Class Link (H O G : Prop) :=
  link_sound : H ∧ O ⟶ G.
```

This defines a type class (Sozeau and Oury, 2008) called `Link`, for which we will define the notation `LINK H ∧ [O] ⊨ G`. To construct an instance `LINK H ∧ [O] ⊨ G`, one must prove `H ∧ O ⟶ G` (*i.e.,* soundness). A semantic definition like this makes it particularly easy to define linkage rules as type class instances:

```
Instance link_asmp_actema A :
  LINK A ∧ [True] ⊨ A.
Proof. unfold Link; firstorder eauto. Qed.
Instance link_l_and_1 H1 H2 O G :
  LINK H1 ∧ [O] ⊨ G ⟶
  LINK (H1 ∧ H2) ∧ [O] ⊨ G.
Proof. unfold Link; firstorder eauto. Qed.
Instance link_l_exists {A} (H O : A ⟶ Prop) G :
  (∀ a, LINK (H a) ∧ [O a] ⊨ G) ⟶
  LINK (∃ a, H a) ∧ [∀ a, O a] ⊨ G.
Proof. unfold Link; firstorder eauto. Qed.
```

Instances for the other rules from Figure 5.1b are similar. The key step is to define an appropriate instance for our new rule $\mathrm{L}\forall_{QU}$. Let us repeat the statement from §5.3.1:

$$\frac{\forall \vec{y}. \quad H[t/x] \land [O] \models G}{(\forall x.\, H) \land [\exists \vec{y}.\, O] \models G}$$

Remember that the term t can mention the variables $\vec{y}$. We will first express the rule in a form where we allow t to depend on exactly one variable y:

```
Lemma link_l_forall_qu_v1 {A} (H : A ⟶ Prop) G
      {Y} (t : Y ⟶ A) (O : Y ⟶ Prop) :
  (∀ (y : Y), LINK (H (t y)) ∧ [O y] ⊨ G) ⟶
  LINK (∀ a, H a) ∧ [∃ (y : Y), O y] ⊨ G.
Proof. unfold Link; firstorder eauto. Qed.
```

We have to address two issues to turn this 1-ary lemma into an instance that produces good simplifications and can be applied automatically by type class search.

The first issue is that while the n-ary version can be derived from 1-ary function through currying, this results in a complicated simplification. We want the resulting simplification to be an n-ary existential quantification, instead of a unary quantification on a product Y. In particular, we want to avoid a useless quantification over `u : unit` if no variables are needed. Generating an n-ary existential quantification is important to show readable goals to the user and to aid automation in making further progress. We address this issue using *telescopes* to write an n-ary rule (§5.4.2).

The second issue is that of determining appropriate terms for `Y`, `t` and `O`. We can use evars `?Y`, `?t` and `?O` when applying this lemma, but instantiating these evars is challenging. At some point we want to unify *e.g.,* `?t y` with a concrete term—while also instantiating the type `?Y` of `y`. We do not know of any existing unification algorithm that supports this kind of problem. Indeed, Coq's default unification algorithm rightfully refuses to solve this problem. We address this issue using a custom tactic written in Ltac (§5.4.3).

5.4.2 Simple Telescopes in Coq

Telescopes (de Bruijn, 1991) can represent (the type of) sequences of variables with possibly dependent types. For the applications in § 5.5, we use the formalization of (dependent) telescopes provided by the coq-std++ library (The Coq-std++ Team, 2023).[4] For brevity, the telescopes in this section do not allow dependent types.

We use telescopes to formalize n-ary existential quantification in QU. For example, for $\exists(x : X)(y : Y).\, P\, x\, y$, we use '$[X; Y]$: tele'. Telescopes give us a generic uncurried version $\overline{P} : [X_1; \ldots; X_n] \to$ `Prop` of $P : X_1 \to \cdots \to X_n \to$ `Prop`, and a generic telescopic quantifier $\vec{\exists}\overline{P}$ that simplifies (*i.e.*, is definitionally equal) to the n-ary existential quantification.

In the non-dependent setting, we can represent telescopes as a simple list of `Type`s:

```
Definition tele : Type := list Type.
Definition teleS : Type → tele → tele := cons.
Definition teleO : tele := nil.
Notation "[tele X ; .. ; Z ]" :=
  ((teleS X (.. (teleS Z teleO) .. ))).
```

We can treat such lists themselves as a `Type`, by taking the product of all the `Type`s in the list. We can construct this product by folding over the list:

```
Definition tele_arg (T : tele) : Type :=
  fold_right prod unit T.
Coercion tele_arg : tele >-> Sortclass.
Check ((1, (false, tt)) : [tele nat; bool]).
```

After registering `tele_arg` as a `Coercion`, the preceding `Check` indeed goes through. This relies on Coq doing a type-level computation: it checks that `tele_arg [nat; bool]` is convertible to the type `nat * (bool * unit)`. Therefore, we have that `(1, (false, tt))` is of type `tele_arg [nat; bool]`.

The function type `T → Prop` with `T : tele` corresponds to an uncurried n-ary function. We can do n-ary quantification on such functions by recursion on the list `T`:

```
Fixpoint tele_ex {T : tele} : (T → Prop) → Prop :=
  match T with
  | [] ⇒ fun g : unit → Prop ⇒ g tt
  | X :: T' ⇒ fun g : (X × tele_arg T') → Prop ⇒
      ∃ (x : X), tele_ex (fun r ⇒ g (x, r))
  end.

Lemma tele_ex_exists {T : tele} (g : T → Prop) :
  tele_ex g ↔ ∃ (a : T), g a.
```

We have included some type annotations in `tele_ex` to illuminate what is going on. In the 0-ary case, we simply pass the unit element to the function of type `unit → Prop`. In the $(n + 1)$-ary case, we existentially quantify on the first projection of the pair, and recursively call `tele_ex` to existentially quantify on the second projection.

[4] The coq-std++ library (The Coq-std++ Team, 2023) defines `tele` as a custom inductive type to handle dependent types. Furthermore, by making this inductive type universe polymorphic (Sozeau and Tabareau, 2014), coq-std++ avoids universe constraints that would otherwise restrict the usage of telescopes in larger developments.

Lemma `tele_ex_exists` shows that `tele_ex` is equivalent to regular existential quantification. Remember that to formalize $\mathrm{R}\exists_{QU}$ and $\mathrm{L}\forall_{QU}$ we want to avoid a regular (unary) existential quantification on a complicated type like `T : tele`, and instead generate n nested existential quantifiers.

5.4.3 Quantifying on the Uninstantiated with Ltac

We are now ready to state a version of $\mathrm{L}\forall_{QU}$ with proper n-ary quantification:

```
Lemma link_l_forall_qu_v2 {A} (H : A → Prop) G
    {Y : tele} (t : Y → A) (O : Y → Prop) :
  (∀ (y : Y), ∃ (t' : A) (O' : Prop),
     LINK (H t') ∧ [O'] ⊫ G
       ∧ t' = t y ∧ O' = O y) →
  LINK (∀ a, H a) ∧ [ tele_ex O ] ⊫ G.
```

This lemma differs from `link_l_forall_qu_v1` in §5.4.1 in its use of `Y : tele` and `tele_ex`.

Furthermore, to address the problem of unifying `?t y` with a concrete term, we swap `t y` with a new variable `t'`, and require these two to be equal (and similarly for `O`). This means that when we use Coq's type class search to establish the `LINK` premise, it does not need to worry about (and is in fact oblivious of) the fact that `t'` and `t y` should be equal. This change also allows us to solve the unification problem for `Y` manually: we can determine an appropriate value for `Y` with some meta-programming in Ltac when proving `t' = t y`.

Let us consider the proof obligations spawned by applying `link_l_forall_qu_v2`. Suppose we would like to prove `LINK (∀ a, H a) ∧ [ ?O ] ⊫ G`. To proceed, we apply the lemma, introduce `y` and make fresh evars for `t'` and `O'` with tactic:

```
eapply link_l_forall_qu_v2; intros; do 2 eexists.
```

After the application of this tactic, our goal is:

```
LINK (H ?t') ∧ [?O'] ⊫ G  ∧  ?t' = ?t y ∧ ?O' = ?O y
```

Since the argument to `H` is a simple evar `?t'`, the first conjunct can be handled by a recursive call to the linking procedure (*i.e.,* type class search). In particular, in the base case Asmp-actema can instantiate `?t'` with an appropriate term if necessary. This would not be possible for `?t y`.

Let us now repeat Equation (5.4) from §5.3.2 and consider our proof obligations. We are trying to derive:

$$(\forall x.\, \exists y.\, Q\, x\, y) \wedge [\ldots] \vDash \exists z.\, \exists u.\, \exists v.\, Q(g\, u\, v)\, z.$$

The combination of tactics discussed previously will fill in an evar `?t'` for x. The linking procedure will (recursively) establish `LINK (H ?t') ∧ [?O'] ⊫ G` and in the process unify `?t'` with g `?u` `?v`. The equality we thus wish to prove is

$$g\ ?\mathtt{u}\ ?\mathtt{v} = ?\mathtt{t}\ \mathtt{y}, \tag{5.5}$$

where `y` is of type `?Y`. The full unification problem we face is

$$\exists (Y : \mathtt{Type}).\, \exists (t : Y \to A).\, \forall (y : Y).\, \exists\, u\, v.\, g\, u\, v = t\, y.$$

```
1  Ltac solve_evar_tele_equality :=
2    lazymatch goal with
3    | ⊢ ?l = ?f ?arg ⇒
4      let rec retcon_tele the_arg T :=    (* we receive the_arg : tele_arg T *)
5        match l with
6        | context [?term] ⇒               (* look through all subterms of l *)
7          is_evar term;                  (* check that the subterm term of l is an evar *)
8          let X := type of term in
9          let T' := open_constr:(_) in   (* creates a new evar T' *)
10         unify T (teleS X T');          (* instantiates T to be X :: T' *)
11         unify term (fst the_arg);      (* instantiates term to be fst arg, of type X *)
12         retcon_tele (snd the_arg) T'   (* .. now repeat this for other evars in l *)
13       | _ ⇒ unify T teleO               (* if l has no evars, unify T with nil *)
14       end
15     in
16     let T' := lazymatch type of arg with tele_arg ?T ⇒ T end in
17     retcon_tele arg T';
18     exact (eq_refl _)                  (* unification can now instantiate f *)
19   end.
```

Figure 5.3: Ltac code for QU.

We want to quantify on the uninstantiated, so our goal is to infer `Y = [tele U; V]`. Evars ?u and ?v should be unified with projections of `y`, so that the remaining unification problem can be solved by Coq. This is what the Ltac script `solve_evar_tele_equality` from Figure 5.3 does.

If we call `solve_evar_tele_equality` on Equation (5.5), line 3 in Figure 5.3 will store the left-hand side of the equality in `l`, and `y` in `arg`. We then read the telescope `Y` into variable `T'` (line 16), and call `retcon_tele arg T'`. This will 'retcon' (for retroactive continuity) the telescope `T'` to be a list of the types of all uninstantiated evars in `l`. Additionally, it will unify all evars with projections of `arg`. The recursive `retcon_tele` achieves this by scanning `l` for evars (lines 4–6),[5] and then unifying `T'` to be a list that starts with the type of this evar (lines 8–10).

We then unify the evar with a projection of `arg` (line 11), and repeat the process until no more evars are found (line 12 and 13). If we find an evar that does not have `arg` in scope, the unification on line 11 will fail and cause Ltac to backtrack and continue with the next evar. This is desired: it means such evars should either not be quantified on, or be quantified on by some earlier application of $\mathrm{L}\forall_{QU}$.

We still need to prove the equality once this is finished. The equality in Equation (5.5) has been reduced to

$$g\ \texttt{(fst y) (fst (snd y)) = ?t y.}$$

and '`exact (eq_refl _)`'[6] can make quick work of this: Coq's unification algorithm (Ziliani

[5] The `match context` combination with `is_evar` in lines 4–6 may seem to be an inefficient way of finding all evars in a term `l`, since it traverses all subterms of `l`. However, our experiments showed it to be more efficient than an alternative implementation using `unshelve`.

[6] It is crucial to use `exact` or `refine`, instead of `reflexivity` or `apply`. As mentioned here, `exact` uses Coq's newer 'evar_conv' unification algorithm, which often performs better than the older 'w_unify' unification

and Sozeau, 2015, step 4 on page 186) is able to infer an appropriate value for `t` now.

The technique of 'retroactively' determining an appropriate list of types is similar in spirit to those in the 'Procrastination' library (Guéneau, 2018). The Procrastination library instead collects side conditions on an evar, *i.e.,* propositions.

5.4.4 Linkage Tactic

We now have all ingredients in place to construct an instance for $\mathrm{L}\forall_{QU}$ that Coq's type class search can understand.

To call the custom Ltac from §5.4.3 we do not register `link_l_forall_qu_v2` as a regular instance. Instead we add an external hint to the type class database:

```
Hint Extern 4 (LINK (∀ a, _) ⊼ [ _ ] ⊫ _) ⇒
  eapply link_l_forall_qu_v2; intros ?; do 2 eexists;
    split; [ solve [typeclasses eauto] | ];
    split; [ solve_evar_tele_equality | ];
    exact (eq_refl _) : typeclass_instances.
```

This hint applies our specially crafted lemma, runs type class search on `LINK H ?t ⊼ [?0] ⊫ G`, and then quantifies on uninstantiated evars in `?t`. The `exact (eq_refl _)` tactic takes care of the remaining equality on `?0`.

Putting it all together. With type class instances for all the linkage rules in place, we obtain a very simple implementation of a linkage system in Coq—in about 250 lines of code in total. This includes the straightforward implementation of a tactic `link_to` that performs LINK-APPLY, omitted here. All linkages that were discussed before have the desired result. In the supplementary material (Mulder and Krebbers, 2023b), we have included solutions to some of the exercises in Actema's course on first-order logic, to demonstrate our linkage system.

Non-determinism. We have not specified a heuristic that determines in what order the linkage rules should be applied. By implementing the linkage rules as type class `Instance`s, we implicitly rely on the backtracking semantics of type class search—all orders will be considered. This means linking only fails after all possible orders of linking rules have been considered, which is not great from a performance perspective. Similar to ProfInt and Actema, we use heuristics to determine the rule order in our applications in §5.5, and thereby avoid this inefficiency. We either have no 'left' rules (`iFrame` in §5.5.1), or we prioritize 'left' rules (Diaframe in §5.5.2).

5.5 Applications

We demonstrate that the QU approach has practical applications outside pure intuitionistic logic. First, we apply QU to the *framing* problem (Berdine et al., 2005; Kassios, 2006) from separation logic in the context of the Iris framework for concurrent separation logic in Coq (Jung et al., 2015, 2016; Krebbers et al., 2017b; Jung et al., 2018b; Krebbers et al., 2017a, 2018) (§ 5.5.1). Second, we apply QU to the Iris-based proof automation framework Diaframe described in Chapters 2 to 4 (§ 5.5.2), where we show that the

algorithm.

automatic verification of a classical readers-writer lock crucially relies on the QU rules for subformula linking under quantifiers.

5.5.1 Framing under Quantifiers in Separation Logic

Separation logic (O'Hearn et al., 2001) is an extension of Hoare logic that allows one to reason modularly about the correctness of stateful programs. We focus on the assertion language of separation logic, which extends ordinary logic with two logical connectives that enable this modular reasoning: the separating conjunction ($*$) and magic wand ($-\!\!*$). Separating conjunction can be seen as a *substructural* version of conjunction ($\wedge$), which means that we cannot use separation logic propositions P more than once—in particular, $P \vdash P * P$ does not hold in general. The introduction rule of separating conjunction thus requires one to split the list of hypotheses over the conjuncts:

$$\frac{\Delta_1 \vdash P \qquad \Delta_2 \vdash Q}{\Delta_1, \Delta_2 \vdash P * Q}$$

During program verification with separation logic, one often faces proof obligations of the form $\Delta, P \vdash P * G$. In this case, there is an obvious choice for splitting the environment: one 'frames' P away, and continues with $\Delta \vdash G$ (*i.e.,* take $\Delta_1 = P$ and $\Delta_2 = \Delta$). In an interactive proof setting, one should not have to spell out the precise environments.

Iris comes with an interactive proof mode and accompanying tactics (Krebbers et al., 2017a, 2018), whose `iFrame` tactic can be used to frame away hypotheses in the goal. This tactic is implemented with a type class `Frame`, exactly like `Link` from §5.4.1. The (slightly simplified) definition of `Frame` is:

```
Class Frame {PROP : bi} (H G O : PROP) :=
  frame : H ∗ O ⊢ G.
```

Compared to `Link`, the `Frame` class involves the separating conjunction ($*$) instead of the regular conjunction ($\wedge$). Furthermore, `Frame` works in a generic object logic `PROP` of type `bi`. This makes the `Frame` type class applicable in any Bunched Implication logic (O'Hearn and Pym, 1999; Pym, 2002), *i.e.,* logics that satisfy the relevant axioms for $*$ and $-\!\!*$.

When trying to frame resource `H` in goal `G`, Iris runs type class search for `Frame H G O`. If successful, it removes resource `H` from the environment, and replaces the goal with `O`. Instances of the `Frame` type class are 'just' subformula linking rules in separation logic. This is evident by comparing the following instances to Asmp-actema and R$\wedge$:

```
Global Instance frame_here A : Frame A A emp.
Global Instance frame_sep_l H G1 G2 O :
  Frame H G1 O →
  Frame H (G1 ∗ G2) (O ∗ G2).
```

Framing under existential quantification. To frame beneath quantifiers, one faces problems similar to those described in §5.2. However, there are also some differences that we discuss first. When framing, we only look for hypotheses that appear (nearly) verbatim in the goal—meaning there are only right rules for `Frame`. The existing implementation for framing under existential quantifiers only provides R$\exists$ and not R$\exists_n$-actema. This means that framing could not instantiate quantifiers, and so it fails on *e.g.,* $P\,1 \vdash (\exists n.\, P\,n) * Q$.

Having a single instance was a conscious design choice of the Iris Proof Mode: by having two applicable `Instance`s when the goal is existentially quantified, (failing) type class search would run twice on very similar subgoals. An n-ary existential quantification would do this 2^n times, which becomes unacceptably slow.

By quantifying on the uninstantiated, we can allow framing to instantiate quantifiers without this exponential slowdown. Additionally, remember that the QU rule R$\exists_{QU}$ is strictly more general than having both R$\exists_n$-ACTEMA and R$\exists$. We use the following `Frame` instance, similar to the linking instance `link_l_forall_qu_v2` from §5.4.3.

```
Lemma frame_exist_qu {A : Type} (G : A → PROP) H
    {Y : tele} (t : Y → A) (O : Y → PROP) :
  (∀ (y : Y), ∃ (t' : A) (O' : PROP),
      Frame H (G t') O'
        ∧ t' = t y ∧ O' = O y) →
  Frame H (∃ a, G a) (bi_texist O).
```

The main difference is the use of `bi_texist`, which does n-ary existential quantification in `PROP`. We also reuse the tactics from §5.4.3 for proving the equalities on `t'` and `O'`.

5.5.2 Automatic Verification of a Readers-Writer Lock

The proof automation provided by Diaframe also relies on subformula linking, and as such faces the same problems regarding quantifiers. Diaframe has been using the QU approach for a while, but the technique and its use were not described anywhere.

Let us consider the automatic verification of the classic readers-writer lock by Courtois et al. (1971) in Diaframe. A lock is a data structure from concurrent programming, in charge of sharing access to a resource R among multiple threads. It guarantees that at all times, at most a single thread can access resource R. A readers-writer lock generalizes a regular lock: it guarantees that either there are zero or more 'readers', *i.e.,* threads with read-only access to R, or there is a single 'writer' thread that can mutate R.

The classic readers-writer lock implementation by Courtois et al. (1971) is built from two regular locks. We shall consider the verification of *allocating* a new readers-writer lock, which first allocates two regular locks. Let us first consider two specifications for allocating a regular (spin) lock:[7]

$$\{R\}\ \mathsf{new\_lock}()\ \{v.\ \exists \gamma.\ \mathrm{is\_lock}\ \gamma\ v\ R\} \tag{5.6a}$$

$$\{\mathsf{True}\}\ \mathsf{new\_lock}()\ \{v.\ (\forall R.\ R \mathrel{-\!\!*} \exists \gamma.\ \mathrm{is\_lock}\ \gamma\ v\ R)\} \tag{5.6b}$$

Specification (5.6a) states that executing $\mathsf{new\_lock}()$ is safe, and returns a value v for which $\exists \gamma.\ \mathrm{is\_lock}\ \gamma\ v\ R$ holds—if we have given up resource R before executing $\mathsf{new\_lock}$. Parameter γ ensures we can distinguish between different locks. The $\mathrm{is\_lock}$ predicate is part of the precondition of the other lock methods.

Specification (5.6b) differs from (5.6a) in that one does not have to give up resource R directly. Rather, it returns a more complicated proposition, which allows clients to choose and hand in R at a later point in the program execution.

Although (5.6a) is the standard lock specification (Hobor et al., 2008; Dinsdale-Young et al., 2010), (5.6b) is strictly stronger. One can (manually) verify the readers-writer lock

[7] We omit Iris's update modality $\Rightarrow$ from (5.6b) since it poses orthogonal problems with automation that are addressed in Chapter 2.

Table 5.1: Evaluation data of `iFrame`. For each project, we list the total number of lines, the number of lines that use `iFrame`, and the number of lines that needed to be changed for the new `iFrame`. We also list the total compilation time of the project, and the change in compilation time with the new `iFrame`.

repository	total lines	`iFrame` lines	lines changed	total time	time changed %
Iris	60156	543	- 2	8:29	-1.0%
iris-examples	22912	800	-14	10:03	-1.2%
ReLoC	14092	505	- 5	4:54	-2.0%
RustBelt	19889	840	-33	14:09	+0.3%
total	117049	2688	-54	37:37	-0.7%

with (5.6a), but (5.6b) is more useful for proof automation. When an automated verifier symbolically executes new_lock with specification (5.6a), it does not have any syntactic indication for an appropriate choice for resource R. A wrong choice can easily lead to a failing verification (Dardinier et al., 2023). With specification (5.6b), the automation can wait for a proof obligation with shape $\mathsf{is\_lock}\,\gamma\,v\,S$ to choose R equal to S.

This is precisely what happens when verifying the allocation of Courtois et al. (1971)'s readers-writer lock. The allocation function first allocates two regular locks, for which we will use specification (5.6b). To prove that the readers-writer lock is successfully allocated, we are faced with goal:

$$(\forall R.\ R \mathrel{-\!\!*} \exists \gamma.\ \mathsf{is\_lock}\,\gamma\,v\,R) \vdash \exists \gamma_1 \gamma_2.\ \mathsf{is\_lock}\,\gamma_1\,v\,(P\,\gamma_2).$$

Here, $P\,\gamma_2$ encodes the protocol for accessing the readers-writer lock. Since Diaframe uses the QU rules, it can simplify this entailment to $\exists \gamma_2.\ P\,\gamma_2$, which Diaframe's automation can subsequently discharge. The QU rules are crucial: during subformula linking, R will be unified with $P\,?\gamma_2$, where $?\gamma_2$ remains uninstantiated. By quantifying precisely on this γ_2, we obtain the desired linkage.

Finally, note that the readers-writer lock verification involves quantification over propositions R. This shows that the QU approach scales to higher-order logic.

5.6 Evaluation of iFrame

We evaluate the scalability of QU by testing our improved `iFrame` tactic. We test the improved tactic on four Iris projects to verify it does not cause performance regressions or introduce failures where it succeeded before (§5.6.1). We report on the results of a more artificial benchmark that compares the performance of the new $\exists$ rule to the rules for other connectives (§5.6.2). This benchmark shows that the Ltac implementation from §5.4.3 has acceptable complexity. We cannot conduct an evaluation of Diaframe, since there exists no baseline version of Diaframe without QU.

5.6.1 Subformula Linking with iFrame in Practice

Table 5.1 contains the results of our evaluation of the improved `iFrame` tactic. We investigate four Iris-based repositories of significant size: Iris itself, ReLoC (Frumin et al., 2018,

2021b), RustBelt (Jung et al., 2018a), and Iris's 'examples' repository. We compare the total compilation time of each repository with and without the improved `iFrame`. We also report the total number of changed lines that were required to patch these repositories. Existing proofs might break because `iFrame` is more powerful in the sense that it can frame more hypotheses and even solve a goal entirely that was not solved before.

We find that overall, the effect on compilation time (0.7% faster) is hardly distinguishable from noise. This is a positive result because the improved `iFrame` is strictly stronger, without being noticeably slower. The 2% speedup in ReLoC may be due to the fact that the QU instance uses a `Hint Extern` with a pattern, meaning the framed goal must be an existential quantification *syntactically*. The previous `Instance` for framing under existential quantifiers would also trigger on goals that are not syntactically an existential quantification, but can be unfolded to one.

The lines of code reduction by the improved `iFrame` are modest, but not insignificant: a reduction of 54 lines on a total of about 2700 lines using `iFrame`. (Note that we have omitted the addition of ±150 lines of implementation of the QU rule in Iris.) These numbers can be improved—we have only fixed broken proofs, not optimized existing proofs. In some of the changed lines, a single call to the improved `iFrame` has replaced a combination of five tactics.

We have not investigated the effect of the improved `iFrame` on writing new proofs. However, our impression is that the additional strength of the tactic makes it easier for users to write proofs: more parts can be automatically discharged. A frequently occurring pattern is a proof obligation of shape $\Delta, P\,x \vdash \exists y.\, P\,y * G$ with $\Delta \vdash G$ requiring a manual proof. Such situations were typically tackled with the combination '`iExists _;iFrame`', but a single call to the improved `iFrame` now suffices.

5.6.2 Comparing Performance of Linkage Rules

We conduct a more artificial benchmark to check that the QU rule for $\exists$ performs acceptably in comparison to the linkage rules for other connectives—even in the presence of large terms or many quantifiers. We consider the following framing problem in which we vary the number n:

$$P * Q * R * S\,t\,\ldots\,t \vdash \exists^n \vec{x}.\, L * R * Q * S\,\vec{x} * P.$$

Here, $\exists^n$ is an n-ary existential quantification ($\vec{x}$ has length n), and S has n arguments. We consider two variants (1) frame (small) P away, which preserves all n existential quantifiers in the goal; and (2) frame (large) $S\,t\,\ldots\,t$ away, which instantiates all quantifiers in the goal. These cover the frequent use cases of keeping a quantifier and instantiating it.

To compare the rule for existentials to other connectives, we consider the following variant:

$$P * Q * R * S\,t\,\ldots\,t \vdash O^n(L * R * Q * S\,?\vec{x} * P),$$

Here, O is an element of $\{\forall \_.\, \cdot, (R \mathrel{-\!\!*} \cdot), (R * \cdot), (R \wedge \cdot)\}$.[8] We consider the same variants as before (frame P or frame S). In the second variant we must also unify $S\,?\vec{x}$ with $S\,t\,\ldots\,t$. The previous problem is thus very similar to this one, apart from the connectives being framed under.

[8] We do not consider disjunction, since its framing rule in Iris requires a link on both sides.

We pick t to be a term of significant size, and test the performance with L being a large and small term. This means we compare the performance of four framing problems in total: framing (small) hypothesis P *or* (large) hypothesis S in a goal with a small *or* large L.

Results. The benchmarks show that the performance of the QU rules for $\exists$ are as fast as those for other connectives in 3/4 problems, namely when framing large S and/or large P. When framing small P in large L and $n < 60$, the performance is about equal to that of $\wedge$, and up to four times as slow as $-\!\!*$ (the fastest connective). At $n = 150$, the QU rules for $\exists$ are twice as slow as $\wedge$, and seven times slower than $-\!\!*$. We have included some of the running times for the small hypothesis, small goal framing under $\exists$ and $\wedge$ below:

n	6	12	30	60	150
runtime for $\exists$ (seconds)	0.07	0.15	0.33	0.9	5.5
runtime for $\wedge$ (seconds)	0.07	0.13	0.32	0.76	2.8

This shows that the QU rules perform acceptably, even with n-ary existential quantification for large n (the authors have not seen $n > 10$ in existing Iris projects). There is an observable difference in performance only when $n \geq 60$, but we conjecture that the time Coq spends on proof checking is a much more influential factor than the runtime of `iFrame`.

5.7 Related Work

Subformula linking. The notion of subformula linking has been introduced as part the Profound system by Chaudhuri (2013). Profound is a predecessor of ProfInt (Chaudhuri, 2021, 2023) that involves first-order classical linear logic instead of first-order intuitionistic logic.

Prior works on subformula linking differ mostly from this work in their application. The ProfInt (Chaudhuri, 2021, 2023) and Actema (Donato et al., 2022) gesture-based interactive theorem provers use subformula linking to simplify proof states by letting the user graphically indicate what subformulas to link. (A further predecessor of gesture-based theorem proving is 'proof by pointing' by Bertot et al. (1994).) Aside from the difference in applications, there are some notable differences in the formal systems:

- Besides links between hypothesis and goal, ProfInt and Actema also consider links between two hypotheses. ProfInt even considers links within a single formula. We do not consider such links, because hypothesis-goal links are the only relevant links for our application of *backward-chaining proof search*—which always takes the goal as starting point. Hypothesis-hypothesis links are essential for the completeness of ProfInt.

- The original presentation of Profint (Chaudhuri, 2021) uses a weaker subformula linking rule than R$\vee$. This weaker rule makes the rule order immaterial for the propositional fragment, but sometimes produces links that are harder to prove than links with R$\vee$. The rule order still matters when two hypotheses are linked.

- Actema supports 'rewriting' through subformula linking with an additional linkage rule $x \doteq y \land [P\,x] \vDash P\,y$. Such a rule can be added as an instance to a QU-based system. It would be interesting to explore applications of such linkages, especially in combination Coq's generalized rewriting (Sozeau, 2009), which is used extensively in Iris to rewrite with relations other than equality.

Framing in separation logic. Various approaches have been proposed to (automatically) solve the 'framing' or 'frame inference' problem (Kassios, 2006; Berdine et al., 2005) in separation logic.

The VST framework in Coq (Cao et al., 2018) comes with a `cancel` tactic for frame inference, which contrary to Iris's `iFrame` does not proceed below existential quantifiers in the goal. VST also provides the more powerful `entailer!` tactic (Appel, 2023, §42), which uses `cancel` after suitably normalizing and rewriting the proof goal. These can both be used to fully solve an entailment or make partial progress in an interactive proof.

The automation of the Bedrock (Chlipala, 2011, §3, step 6) and RefinedC (Sammler et al., 2021, §4, step 5) frameworks in Coq instantiates existentially quantified goals with evars after having eliminated existentials in hypotheses. This is an effective approach for the verification of sequential problems. However, in the context of the verification of concurrent programs in Iris, existentials require a more careful treatment, as also argued in Chapter 2. We achieve this through subformula linking.

Calcagno et al. (2009b) describe a shape-analysis based procedure for inferring *e.g.,* pre- and postconditions of Hoare triples. They provide recursive rules to solve the framing problem ('abductive inference') in a specialized separation logic. Their $\mapsto$-`match` rule (Calcagno et al., 2009b, Figure 1) handles existential quantifiers on points-to resources, in a way similar to ProfInt's R$\exists$. This is less general than R$\exists_{QU}$, but works appropriately since there can only be one points-to resource for a given location.

5.8 Future Work

We have presented QU, an approach for subformula linking under quantifiers using unification, and demonstrated its use by improving Iris's `iFrame` tactic for resource framing. We see several possible directions for future work.

Both Actema and ProfInt come with prototypes for graphical interactive theorem proving. It would be interesting to build such a prototype for QU, or to change an existing prototype to use the QU rules for quantifiers.

All systems we have considered here (Actema, ProfInt, and QU) leave the order of linking rules unspecified, while these are crucial for the quality of the resulting simplification. In particular, the systems use a heuristic to decide whether to prioritize left- or right-rules. It would be interesting to design a formal inference system that rules out information loss entirely.

Chapter 6

Conclusions and Future Work

In this thesis, we studied the verification of fine-grained concurrent programs—crucial building blocks of modern operating systems. Fine-grained concurrent programs are notoriously hard to get right, and play an important role in the software stack, so verifying them is of utmost importance. Existing research has focused either on *automatic* verification of these programs, or on *foundational* (*i.e.,* highly trustworthy) verification of these programs. This thesis presents the first work that allows both automatic and foundational verification of fine-grained concurrent programs.

The main contributions of this thesis are proof search strategies for Iris's concurrent separation logic, and implementations of these in the Diaframe library for proof automation. We used Diaframe to prove functional correctness and linearizability of various examples, showing that the proof burden of verification with Diaframe is competitive to existing automated tools while adding foundational guarantees.

The proof search strategies we developed share four key ideas. That is, they all do goal-directed proof search, try to avoid backtracking entirely, are extensible with rules fitting a general format, and use subformula linking to detect cases where a rule is almost applicable. The resulting strategies show that at least for the verification of fine-grained concurrent programs, foundational methods need not be less automated than non-foundational (*i.e.,* SMT-solver based) methods.

Nevertheless, verifying fine-grained concurrent programs remains challenging. We see several directions for future work.

Fundamental improvements. So far, we have focused on automating proofs of given specifications in existing program logics. We see two possibilities for fundamentally improving this workflow.

Firstly, coming up with good specifications is often one of the hardest parts of concurrent program verification. Methods for automatically inferring program invariants could make verification significantly less labor-intensive. Calcagno et al. (2009a) describe a method to infer lock invariants in a relatively simple concurrent separation logic. Scaling inference to invariants for fine-grained concurrent programs could be very useful.

Secondly, the program logic we have used still has room for improvement. One improvement relates to programs that feature so-called 'future dependent linearization points'. Examples are the Michael–Scott queue (Michael and Scott, 1996) and the RDCSS

algorithm (Harris et al., 2002). The verification of these programs in Iris is tricky since certain information is required at one point, while only becoming available later. Jung et al. (2020) successfully verified the RDCSS algorithm by introducing *prophecy variables* to Iris. Although this does the job, the required reasoning appears to be fundamentally not goal-directed, and so not amenable to our current proof automation strategies. Meyer et al. (2023b, 2022) have developed a separation logic that can perform this kind of reasoning without prophecy variables—using 'temporal interpolation'. Their tool `plankton` achieves an impressive degree of automation using this technique. In its current form, temporal interpolation sacrifices some compositional reasoning. It would be very interesting to see if this restriction could be lifted, and to get access to temporal interpolation in Iris.

Domain-specific improvements to automation. One of the key features of our proof automation strategies is its extensibility: support for new resources or new goals can be added by users. However, ideally one should not need to add explicit support for relatively simple resources and goals—they should be supported out-of-the-box.

For example, Diaframe's support for proving 'pure' sideconditions (*i.e.,* non-separation logic propositions, such as equalities) is limited and relatively ad-hoc. It would be useful to improve this solver, or to replace it with a stronger solver. Recent literature provides several candidates: tools for reconstructing proofs from SMT solvers like SMTCoq (Ekici et al., 2017), but also proof-assistant based approaches such as `itauto` (Besson, 2021), `sauto` (Czajka, 2020), and domain-specific solvers for *e.g.,* lists (Wang and Appel, 2023).

Another possible area of improvement is Diaframe's support for resources that describe recursive data structures. For non-recursive data structures, Diaframe can infer most proof rules automatically by unfolding definitions. This is no longer possible when the definitions are recursive, but we suspect one could come up with other techniques. Perhaps the work of Ta et al. (2018) could be further investigated, which describes techniques to automatically derive proof rules for inductive heap predicates.

Ease of use. User friendliness is crucial in order for proof automation to be an effective tool. We see a couple of directions in which this could be improved. We will focus on improving ease of use of the automation *inside* the proof assistant, and thereby mostly on the ease of use for researchers/other experts. One could also explore creating a front-end tool for Diaframe in the style of auto-active verification (Leino and Moskal, 2010), as also done in RefinedC (Sammler et al., 2021).

When verifying a program, the automation proceeds by symbolically executing it. If the automation gets stuck at one point, it can be unclear what program path caused the stuck verification. Reporting a summary of this could be helpful for debugging.

We have designed our proof search strategies so that proofs can be developed with a mix of interactive and automatic steps. Interactive proofs usually give explicit names to variables and hypotheses, to improve readability. This is somewhat more difficult for the automatic steps, since the generated variables and hypotheses are harder to predict. We have some initial experiments that do this, which should be further evaluated.

Diaframe's tutorial material could also be expanded and improved. These are currently focused on verifying fine-grained concurrent programs, while some researchers may want to use the automation for other ends.

Applications to other Iris projects. There have been a couple of research projects that use Diaframe to obtain some degree of automation. For example, Moine et al. (2023)

used Diaframe to implement symbolic execution tactics for their custom separation logic for reasoning about garbage collection. Moine et al. (2024) did similarly for their custom separation logic for reasoning about disentanglement. Jung et al. (2023) have experiments showing that their modular specifications for hazard pointers can be semi-automatically verified with Diaframe. Finally, Lorenzen et al. (2024) used Diaframe to semi-automatically verify that functional and imperative (sequential) algorithms on trees are equivalent.

Diaframe's proof search strategy is in principle applicable to any Bunched Implication logic (O'Hearn and Pym, 1999; Pym, 2002) that satisfies the laws of the Iris Proof Mode (Krebbers et al., 2018). This has been demonstrated in Park et al. (2024) (work by the author not included in this thesis), where Diaframe was used to help verify a form of linearizability under weak memory. Recent work recasting VST's program logic for C programs (Appel et al., 2014) into an IPM compatible Bunched Implication logic (Mansky and Du, 2024) could therefore also make use of our proof search strategies. Another exciting advancement is an Iris-inspired separation logic that guarantees deadlock-freedom (Jacobs et al., 2024). We expect that Diaframe could be of use here, and in other works that build on the Iris Proof Mode. It would be interesting to investigate this further.

Applications to general proof automation. We conjecture that the key ideas behind Diaframe could also be of use to improve automation for proof assistants in general, *i.e.,* in higher-order (intuitionistic) logics instead of higher-order separation logics. We expect, perhaps counterintuitively, that switching to the 'simpler' intuitionistic setting will make it harder to automate. Substructurality is of help to the automation: since resources do not usually appear twice, any way to obtain a required resource is usually the correct way. Since this is no longer true in an intuitionistic setting, any form of proof automation needs an approach for when required propositions are obtainable in multiple ways.

Finally, as discussed in Chapter 5, there are still some open questions relating to using subformula linking in proof automation. In particular, when both hypothesis and goal are composite formulas with a shared atom, what is the 'best' order of applying subformula linking rules? In other words, what order of applying subformula linking rules results in a link that is easiest to prove? Knowing such an order in advance would prevent needless (and unwanted) backtracking on all the possible orderings.

Bibliography

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

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

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

Andrew W. Appel. 2023. VST Manual. https://raw.githubusercontent.com/PrincetonUniversity/VST/c3ef3da3c21632306878dda6c0a20103cbc828a1/doc/VC.pdf

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

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

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

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

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

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

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Programming Languages and Systems (LNCS)*. 52–68. https://doi.org/10.1007/11575467_5

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)*. 115–137. https://doi.org/10.1007/11804192_6

Yves Bertot, Gilles Kahn, and Laurent Théry. 1994. Proof by Pointing. In *Theoretical Aspects of Computer Software (LNCS)*. 141–160. https://doi.org/10.1007/3-540-57887-0_94

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

Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *FM*. Springer International Publishing, 127–131. https://doi.org/10.1007/978-3-319-06410-9_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

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

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

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

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent Separation Logic. *CACM* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

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

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

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

Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. 2009a. Bi-Abductive Resource Invariant Synthesis. In *APLAS (LNCS)*. 259–274. https://doi.org/10.1007/978-3-642-10672-9_19

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

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

Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. Bringing Order to the Separation Logic Jungle. In *APLAS (LNCS)*. 190–211. https://doi.org/10.1007/978-3-319-71237-6_10

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

CBC. 2010. CBC News Indepth: Power Outage. https://web.archive.org/web/20100812125832/http://www.cbc.ca/news/background/poweroutage/numbers.html

Iliano Cervesato, Joshua Seth 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

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

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

Kaustuv Chaudhuri. 2013. Subformula Linking as an Interaction Method. In *ITP (LNCS)*. 386–401. https://doi.org/10.1007/978-3-642-39634-2_28

Kaustuv Chaudhuri. 2021. Subformula Linking for Intuitionistic Logic with Application to Type Theory. In *CADE (LNCS)*. 200–216. https://doi.org/10.1007/978-3-030-79876-5_12

Kaustuv Chaudhuri. 2023. ProfInt Prototype.
https://chaudhuri.info/research/profint/

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

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

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

Coq Development Team. 2024. The Coq Proof Assistant Reference Manual.
https://coq.inria.fr/doc/

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

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, 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

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

Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 2023. Verification-Preserving Inlining in Automatic Separation Logic Verifiers. *PACMPL* 7, OOPSLA1 (2023), 102:789–102:818. https://doi.org/10.1145/3586054

Nicolaas Govert de Bruijn. 1991. Telescopic Mappings in Typed Lambda Calculus. *Information and Computation* 91, 2 (1991), 189–204.
https://doi.org/10.1016/0890-5401(91)90066-B

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

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE*. 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Willem-Paul de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. 2001. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press.

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

Edsger W. Dijkstra. 1968. Cooperating Sequential Processes. In *Programming Languages : NATO Advanced Study Institute : lectures given at three weeks Summer School held in Villard-le-Lans, 1966*. Academic Press, 43–112. https://doi.org/10.1007/978-1-4757-3472-0_2

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

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

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

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

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

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

Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A Drag-and-Drop Proof Tactic. In *CPP*. 197–209. https://doi.org/10.1145/3497775.3503692

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

Albert Grigor'evich Dragalin. 1988. *Mathematical Intuitionism: Introduction to Proof Theory*. Translations of Mathematical Monographs, Vol. 67. American Mathematical Society.

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

Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC (LIPIcs)*. https://doi.org/10.4230/LIPIcs.DISC.2018.23

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

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

Robert W Floyd. 1967. Assigning Meaning to Programs. In *Mathematical Aspects of Computer Science*. Number 19 in Proceedings of Symposia in Applied Mathematics. 19–32. https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf

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

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

Didier Galmiche and Daniel Méry. 2002. Connection-Based Proof Search in Propositional BI Logic. In *CADE (LNCS)*. 111–128. https://doi.org/10.1007/3-540-45620-1_8

Didier Galmiche and Daniel Méry. 2018. Labelled Connection-Based Proof Search for Multiplicative Intuitionistic Linear Logic. In *ARQNL (IJCAR, Vol. 2095)*. 49–63. http://ceur-ws.org/Vol-2095/paper3.pdf

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

H. Geuvers. 2009. Proof Assistants: History, Ideas and Future. *Sadhana* 34, 1 (2009), 3–25. https://doi.org/10.1007/s12046-009-0001-5

Alexander Gheorghiu and Sonia Marin. 2021. Focused Proof-search in the Logic of Bunched Implications. In *FoSSaCS (LNCS)*. 247–267. https://doi.org/10.1007/978-3-030-71995-1_13

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

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *APLAS*. LNCS, Vol. 4807. 19–37. https://doi.org/10.1007/978-3-540-76637-7_3

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

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

Armaël Guéneau. 2018. Procrastination: A Proof Engineering Technique. In *Coq Workshop*. https://inria.hal.science/hal-01962659

James Harland and David Pym. 1997. Resource-Distribution via Boolean Constraints. In *CADE (LNCS)*. 222–236. https://doi.org/10.1007/3-540-63104-6_21

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

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

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

C. A. R. Hoare. 1978. Communicating Sequential Processes. *CACM* 21, 8 (1978), 666–677. https://doi.org/10.1145/359576.359585

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

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

Joshua Seth Hodas and Dale Miller. 1991. Logic Programming in a Fragment of Intuitionistic Linear Logic. In *LICS*. 32–42. https://doi.org/10.1109/LICS.1991.151628

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

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

Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *PACMPL* 8, POPL (2024), 47:1385–47:1417. https://doi.org/10.1145/3632889

Cliff B. Jones. 1981. *Developing Methods for Computer Programs Including a Notion of Interference*. Ph. D. Dissertation. Oxford University. http://www.cs.ox.ac.uk/files/9025/PRG-25.pdf

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *IFIP*. 321–332. https://www.researchgate.net/publication/221331334

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *PACMPL* 7, OOPSLA2 (2023), 251:828–251:856. https://doi.org/10.1145/3622827

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

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

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)*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM (LNCS)*. 268–283. https://doi.org/10.1007/11813040_19

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

Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Mück, Benjamin Peters, Gert Smolka, and Wehr, Dominik. 2022. A Coq Library for Mechanised First-Order Logic. In *Coq Workshop*. https://github.com/uds-psl/coq-library-fol

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. 2017b. 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. 2017a. 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

Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2018. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *PACMPL* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html

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

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

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

K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *ESOP (LNCS)*, Giuseppe Castagna (Ed.). 378–393. https://doi.org/10.1007/978-3-642-00590-9_27

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *CACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

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

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

Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. *PACMPL* 8, PLDI (2024), 168:518–168:542. https://doi.org/10.1145/3656398

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

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

William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *PACMPL* 8, POPL (2024), 6:148–6:174. https://doi.org/10.1145/3632848

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

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

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

Roland Meyer, Anton Opaterny, Thomas Wies, and Sebastian Wolff. 2023a. Nekton: A Linearizability Proof Checker. In *CAV*. Springer Nature Switzerland, 170–183. https://doi.org/10.1007/978-3-031-37706-8_9

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *PACMPL* 6, OOPSLA2 (2022), 174:1378–174:1407. https://doi.org/10.1145/3563337

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023b. Embedding Hindsight Reasoning in Separation Logic. *PACMPL* 7, PLDI (2023), 182:1848–182:1871. https://doi.org/10.1145/3591296

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

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1991. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic* 51, 1 (1991), 125–157. https://doi.org/10.1016/0168-0072(91)90068-W

Robin Milner. 1980. *A Calculus of Communicating Systems*. LNCS, Vol. 92. Springer. https://doi.org/10.1007/3-540-10235-3

Mitre. 2014. CVE - CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *PACMPL* 7, POPL (2023), 25:718–25:747. https://doi.org/10.1145/3571218

Alexandre Moine, Sam Westrick, and Stephanie Balzer. 2024. DisLog: A Separation Logic for Disentanglement. *PACMPL* 8, POPL (2024), 11:302–11:331. https://doi.org/10.1145/3632853

Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023a. Artifact and Appendix of 'Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic'. Zenodo. https://doi.org/10.5281/zenodo.7799173

Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023b. Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. *PACMPL* 7, PLDI (2023), 161:1340–161:1364. https://doi.org/10.1145/3591275

Ike Mulder and Robbert Krebbers. 2023a. Artifact of 'Proof Automation for Linearizability in Separation Logic'. Zenodo. https://doi.org/10.5281/zenodo.7712620

Ike Mulder and Robbert Krebbers. 2023b. Artifact of 'Unification for Subformula Linking under Quantifiers'. Zenodo. https://doi.org/10.5281/zenodo.10364816

Ike Mulder and Robbert Krebbers. 2023c. Proof Automation for Linearizability in Separation Logic. *PACMPL* 7, OOPSLA1 (2023), 91:462–91:491. https://doi.org/10.1145/3586043

Ike Mulder and Robbert Krebbers. 2024. Unification for Subformula Linking under Quantifiers. In *CPP*. 75–88. https://doi.org/10.1145/3636501.3636950

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022a. Artifact and Appendix of 'Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris'. https://doi.org/10.5281/zenodo.6330596 Project webpage: https://gitlab.mpi-sws.org/iris/diaframe.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022b. 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

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

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs That Alter Data Structures. In *CSL (LNCS)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR*. 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

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

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. https://doi.org/10.2307/421090

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

Wytse Oortwijn, Dilian Gurov, and Marieke Huisman. 2020. Practical Abstractions for Automated Verification of Shared-Memory Concurrency. In *VMCAI (LNCS)*. 401–425. https://doi.org/10.1007/978-3-030-39322-9_19

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

Jens Otten. 2008. leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic (System Descriptions). In *IJCAR (LNCS)*. 283–291. https://doi.org/10.1007/978-3-540-71070-7_23

Jens Otten and Christoph Kreitz. 1995. A Connection Based Proof Method for Intuitionistic Logic. In *TABLEAUX (LNCS)*. 122–137. https://doi.org/10.1007/3-540-59338-1_32

Susan S. Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Ph. D. Dissertation. Cornell. https://ecommons.cornell.edu/handle/1813/6393

Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6, 4 (1976), 319–340. https://doi.org/10.1007/BF00268134

Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. A Proof Recipe for Linearizability in Relaxed Memory Separation Logic. *PACMPL* 8, PLDI (2024), 154:175–154:198. https://doi.org/10.1145/3656384

Matthew J. Parkinson, Richard Bornat, and Peter O'Hearn. 2007. Modular Verification of a Non-Blocking Stack *(POPL)*. 297–302. https://doi.org/10.1145/1190216.1190261

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

Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. In *CoqPL*. https://www.pédrot.fr/articles/coqpl2019.pdf

Gary Lynn 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

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

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014b. 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.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. 46–57. https://doi.org/10.1109/SFCS.1977.32

David J. Pym. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series, Vol. 26. Kluwer.

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

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

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

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

Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: Verification of Machine Code against Authoritative ISA Semantics *(PLDI)*. 825–840. https://doi.org/10.1145/3519939.3523434

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

Robert J. Simmons. 2014. Structural Focalization. *ACM Transactions on Computational Logic* 15, 3 (2014), 21:1–21:33. https://doi.org/10.1145/2629678

Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning* 2, 1 (2009), 41–62. https://doi.org/10.6092/issn.1972-5787/1574

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

Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *ITP (LNCS)*. 499–514. https://doi.org/10.1007/978-3-319-08970-6_32

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

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

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 J. 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

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

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

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

The Coq-std++ Team. 2023. An Extended "Standard Library" for Coq. https://gitlab.mpi-sws.org/iris/stdpp

Richard Kent Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. https://dominoweb.draco.res.ibm.com/58319a2ed2b1078985257003004617ef.html

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

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

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

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

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

Viktor Vafeiadis and Matthew J. 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

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

Arild Waaler. 2001. Chapter 22 - Connections in Nonclassical Logics. In *Handbook of Automated Reasoning*. MIT Press, 1487–1578. https://doi.org/10.1016/B978-044450813-3/50024-2

Lincoln A. Wallen. 1990. *Automated Proof Search in Non-Classical Logics - Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* MIT Press.

Qinshi Wang and Andrew W. Appel. 2023. A Solver for Arrays with Concatenation. *J Autom Reasoning* 67, 1 (2023), 4. https://doi.org/10.1007/s10817-022-09654-y

Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. 2008. The Isabelle Framework. In *TPHOLs (LNCS)*. 33–38. https://doi.org/10.1007/978-3-540-71067-7_7

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

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

Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading *(ICFP)*. 179–191. https://doi.org/10.1145/2784731.2784751

Summary

Concurrent programs are notoriously hard to get right, and play an important role in the modern software stack. This is especially true for *fine-grained* concurrent programs, which allow a lot of interference from threads running in parallel. It is therefore of utmost importance to verify that fine-grained concurrent programs are correct. Existing research has focused either on *automatic* verification of these programs, or on *foundational* (*i.e.,* highly trustworthy) verification of these programs. This thesis presents the first work that does both: automatic and foundational verification of fine-grained concurrent programs.

To verify these programs, we use *concurrent separation logic*. This program logic excels in compositional reasoning about fine-grained concurrency. Concurrent separation logic can be used to establish functional correctness of a program (*i.e.,* that its input-output behavior is correct), and to establish linearizability of a program (*i.e.,* that its effects appear to take place instantaneously). Moreover, compositionality allows us to use the verification of a concurrent library to verify client programs *abstractly*: without having to look at the implementation details of the library.

Specifically, we use the Iris framework for higher-order concurrent separation logic. Iris has been used to successfully verify various complicated fine-grained concurrent programs. Furthermore, Iris is foundational: it is embedded in the proof assistant Coq, and therefore highly trustworthy. However, verifying fine-grained concurrent programs in Iris is labor-intensive. Users have to provide detailed proofs to convince the system that a program is indeed correct.

The main contributions of this thesis are proof search strategies for Iris's concurrent separation logic, and implementations of these in the newly developed Diaframe library for proof automation. Diaframe can (semi-)automatically construct proofs of functional correctness and linearizability for various examples, while retaining the strong guarantees of working in a proof assistant. Moreover, Diaframe's proof automation is explicitly designed to be predictable, flexible, extensible, and to allow partial progress. This ensures that the automation is as helpful as possible for the user, even for failing verifications.

We evaluated Diaframe by comparing its proof burden to that of existing automated tools. The evaluations show that Diaframe's automation is competitive with the state of the art, while adding foundational guarantees.

Samenvatting

Gelijktijdige ('concurrent') programma's bevatten nogal eens subtiele fouten, terwijl ze een belangrijk element zijn van moderne software-systemen. Dit geldt des te meer voor *fijnmazige* ('fine-grained') gelijktijdige programma's: programma's waar gelijktijdig draaiende subprogramma's voor veel interferentie kunnen zorgen. Het is daarom van belang om de correctheid van fijnmazige gelijktijdige programma's te verifiëren. Eerder onderzoek heeft zich ofwel gewijd aan de *automatische* verificatie van zulke programma's, ofwel aan *zeer betrouwbare* ('foundational') verificatie van zulke programma's. In dit proefschrift presenteren we het eerste onderzoek dat beide doet: automatische en zeer betrouwbare verificatie van fijnmazige gelijktijdige programma's.

Om deze programma's te verifiëren gebruiken we *gelijktijdige scheidingslogica* ('concurrent separation logic'). Deze programma-logica blinkt uit in het op een compositionele manier redeneren over fijnmazige gelijktijdige programma's. Met behulp van gelijktijdige scheidingslogica kunnen we laten zien dat een programma functioneel correct is (*i.e.,* dat het invoer-uitvoer gedrag van het programma klopt), en dat een programma lineariseerbaar is (*i.e.,* dat de effecten van het programma op een instantaan moment lijken plaats te vinden). De compositionaliteit van de logica stelt ons verder in staat om programma's te verifiëren die gebruik maken van geverifieerde bibliotheken ('libraries') van gelijktijdige programma's, *op een abstracte manier*: dat wil zeggen, zonder te hoeven kijken naar de implementatie van de bibliotheek.

Om precies te zijn gebruiken we het Iris raamwerk voor hogere-orde gelijktijdige scheidingslogica. Met behulp van Iris is de correctheid van verschillende lastige fijnmazige gelijktijdige programma's vastgesteld. Verder is Iris zeer betrouwbaar, aangezien het raamwerk is ingebed in de bewijsassistent Coq. Jammer genoeg is het verifiëren van fijnmazige gelijktijdige programma's in Iris veel werk. Gebruikers moeten gedetailleerde bewijzen aanleveren om het systeem te overtuigen dat een programma correct is.

De hoofdbijdragen van dit proefschrift zijn strategieën om bewijzen te zoeken in de gelijktijdige scheidingslogica van Iris, en implementaties van deze strategieën in Diaframe, een nieuw ontwikkelde bibliotheek voor bewijsautomatisering. Diaframe kan (semi-)automatisch bewijzen dat diverse programma's functioneel correct of lineariseerbaar zijn, terwijl het de sterke garanties behoudt van het werken in een bewijsassistent. De bewijsautomatisering van Diaframe is ontworpen om voorspelbaar, flexibel en uitbreidbaar te zijn, en om gedeeltelijke voortgang te kunnen maken in een bewijs. Dit zorgt er voor dat de automatisering zo behulpzaam is als mogelijk, zelfs als verificatie faalt.

We hebben Diaframe geëvalueerd door de bewijslast van Diaframe te vergelijken met die van bestaande geautomatiseerde tools. Dit laat zien dat de automatisering van

Diaframe competitief is met recent onderzoek, terwijl het sterkere garanties geeft wat betreft betrouwbaarheid.

Titles in the IPA Dissertation Series since 2022

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways.* Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering*. Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2025-02

Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.[1]

The following research code repositories have been produced during this research:

- Chapter 2: Ike Mulder, Robbert Krebbers, and Herman Geuvers (2022).
  Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris.
  Coq code in a Git repository, permanently available on Zenodo.
  https://doi.org/10.5281/zenodo.6330596
- Chapter 3: Ike Mulder and Robbert Krebbers (2023).
  Proof Automation for Linearizability in Separation Logic.
  Coq code in a Git repository, permanently available on Zenodo.
  https://doi.org/10.5281/zenodo.7712620
- Chapter 4: Ike Mulder, Łukasz Czajka, and Robbert Krebbers (2023).
  Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic.
  Coq code in a Git repository, permanently available on Zenodo.
  https://doi.org/10.5281/zenodo.7799173
- Chapter 5: Ike Mulder and Robbert Krebbers (2023).
  Unification for Subformula Linking under Quantifiers.
  Coq code in a Git repository, permanently available on Zenodo.
  https://doi.org/10.5281/zenodo.10364816

The development version of Diaframe (Coq code in a Git repository) is available at https://gitlab.mpi-sws.org/iris/diaframe. A frozen version of Diaframe is also available at https://doi.org/10.5281/zenodo.14317899.

[1] https://www.ru.nl/en/institute-for-computing-and-information-sciences/research, bottom of page, last accessed April 18th, 2024

Curriculum Vitae

Ike Mulder was born in Leiden on September 1st, 1995. He displayed an interest in the exact sciences from an early age, and went on to study Applied Mathematics at Delft University of Technology in 2012. He completed his bachelor's degree in 2017, then a master's degree specializing in Analysis in 2019. During the master's thesis, Ike discovered a passion for formalized reasoning. He consequently started a PhD in this area in 2020, at the Software Science department of the Radboud University, where he was supervised by Robbert Krebbers and Herman Geuvers.

In his free time, Ike enjoys having drinks with friends, playing board-games and reading science-fiction books. He practices some judo, and loves to go skiing & snowboarding in the winter holidays.