Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic

IKE MULDER, Radboud University Nijmegen, The Netherlands
ŁUKASZ CZAJKA, Technical University of Dortmund, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

Concurrent separation logic has been responsible for major advances in the formal verification of fine-grained concurrent algorithms and data structures such as locks, barriers, queues, and reference counters. The key ingredient of the verification of a fine-grained program is an invariant, which relates the physical data representation (on the heap) to a logical representation (in mathematics) and to the state of the threads (using a form of ghost state). An invariant is typically represented as a disjunction of logical states, but this disjunctive nature makes invariants a difficult target for automated verification. Current approaches roughly suffer from two problems. They use backtracking to introduce disjunctions in an uninformed manner, which can lead to unprovable goals if an appropriate case analysis has not been made before choosing the disjunct. Moreover, they eliminate disjunctions too eagerly, which can cause poor efficiency.

While disjunctions are no problem for automated provers based on classical (*i.e.,* non-separating) logic, the challenges with disjunctions are prominent in the study of proof automation for intuitionistic logic. We take inspiration from that area—specifically, based on ideas from *connection calculus*, we design a simple multi-succedent calculus for separation logic with disjunctions featuring a novel concept of a *connection*. While our calculus is not complete, it has the advantage that it can be extended with features of the state-of-the-art concurrent separation logic Iris (such as modalities, higher-order quantification, ghost state, and invariants), and can be implemented effectively in the Coq proof assistant with little need for backtracking. We evaluate the practicality on 24 challenging benchmarks, 14 of which we can verify fully automatically.

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

Additional Key Words and Phrases: Separation logic, disjunctions, backtracking, fine-grained concurrency, proof automation, Iris, Coq

ACM Reference Format:
Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023. Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, PLDI, Article 161 (June 2023), 25 pages. https://doi.org/10.1145/3591275

1 INTRODUCTION

Separation logic [O'Hearn et al. 2001; Reynolds 2002] and its successor concurrent separation logic [Brookes 2007; O'Hearn 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

Authors' addresses: Ike Mulder, Radboud University Nijmegen, Nijmegen, The Netherlands, me@ikemulder.nl; Łukasz Czajka, Technical University of Dortmund, Dortmund, Germany, lukaszcz@mimuw.edu.pl; Robbert Krebbers, Radboud University Nijmegen, Nijmegen, The Netherlands, mail@robbertkrebbers.nl.

2475-1421/2023/6-ART161
https://doi.org/10.1145/3591275

in separation logic has been instrumental in the development of (mostly)-automated tools for proving functional correctness [Jacobs et al. 2011; Müller et al. 2016; Oortwijn et al. 2020; Piskac et al. 2014b], including foundational tools that are embedded and proved sound in general-purpose proof assistants [Chlipala 2011; Mulder et al. 2022; 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; Mulder et al. 2022; 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] it is needed 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 [Mulder et al. 2022] 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:

[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.

- **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.
- **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 [Otten and Kreitz 1995; Waaler 2001; Wallen 1990]. 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; Mulder et al. 2022; 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 [Mulder et al. 2022] for the Iris framework of higher-order concurrent separation logic in Coq [Jung et al. 2016, 2018, 2015; Krebbers et al. 2018, 2017a,b]. 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; Liang and Miller 2009; Simmons 2014] 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 (§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). 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 (§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. 2023] (§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 (§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 (§2), and conclude with a description of related work (§8).

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

```
1  Context (P : Qp → iProp) {HP : Fractional P}.
2  Definition mk_arc : val :=
3    λ: <>, ref #1.
4  Definition clone : val :=
5    λ: "a", FAA "a" #1;; #().
6  Definition drop : val :=
7    λ: "a",
8      let: "old_val" := FAA "a" #(-1) in
9      ("old_val" = #1).
10 Definition arc_inv (γ : gname) (l : loc) : iProp :=
11   ∃ (n : nat), l ↦ #n ∗ ((⌜n = 0⌝ ∗ no_tokens P γ)
12                          ∨ (⌜0 < n⌝ ∗ token_counter P γ (Pos.of_nat n))).
13 Definition is_arc (γ : gname) (v : val) : iProp :=
14   ∃ (l : loc), ⌜v = #l⌝ ∗ inv N (arc_inv γ l).
15 Program Instance mk_arc_spec :
16   SPEC {{ P 1 }}
17     mk_arc #()
18   {{ (v : val) (γ : gname), RET v; is_arc γ v ∗ token P γ }}.
19 Program Instance clone_arc_spec (γ : gname) (v : val) :
20   SPEC {{ token P γ ∗ is_arc γ v }}
21     clone v
22   {{ RET #(); token P γ ∗ token P γ }}.
23 Program Instance drop_arc_spec (γ : gname) (v : val) :
24   SPEC {{ token P γ ∗ is_arc γ v }}
25     drop v
26   {{ (b : bool), RET #b; (⌜b = true⌝ ∗ P 1) ∨ ⌜b = false⌝ }}.
```

Fig. 1. Automatic verification of an ARC without deallocation

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 (§2.1), and a more challenging version that performs an explicit deallocation on the last `drop` (§2.2). In prior work [Dinsdale-Young et al. 2017; Mulder et al. 2022], 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.

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 Fig. 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 Fig. 1. Each method is specified using a Hoare-style triple 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 Fig. 1 mention resources `is_arc` $\gamma\ v$, representing the knowledge that value v is an ARC with *ghost name* γ, and resources `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 `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` 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 Fig. 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 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]

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

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

[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.

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

Fig. 2. The rules for the 'token' ghost theory.

is that they satisfy the rules in Fig. 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` $P\ \gamma\ 1$ 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 [Mulder et al. 2022]. 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
$$\ulcorner n > 0 \urcorner, \texttt{counter}\ P\ \gamma\ n, \texttt{token}\ P\ \gamma \vdash \Rrightarrow \left(\begin{array}{l} (\ulcorner n - 1 = 0 \urcorner * \texttt{no\_tokens}\ P\ \gamma) \\ \vee\ (\ulcorner 0 < n - 1 \urcorner * \texttt{counter}\ P\ \gamma\ (n-1)) \end{array} \right) * R$$

The disjunction originates from the invariant definition on line 11–12 in Fig. 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 $\ulcorner n - 1 = 0 \urcorner$ (§5.3).

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

```
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⌝ }}.
```

Fig. 3. Automatic verification of an ARC with explicit deallocation.

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 Fig. 3. The `mk_arc` and `clone` methods have the same implementation and 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 §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` $P\ \gamma$ and `no_tokens` $P\ \gamma$ 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 [Mulder et al. 2022] 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 §2.1.

Reestablishing invariants with non-guarded disjunctions. The verification of `drop` 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

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

The disjunction behind the turnstile comes directly from the definition of the invariant `arc_inv_free` on line 8–9 in Fig. 3. The situation is the same as in §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 $\texttt{counter}\ P\ \gamma\ p$ to the $\texttt{no\_tokens}\ P\ \gamma$ resources (§5.3). This connection instructs the proof search that if we *may* want to prove $\texttt{no\_tokens}\ P\ \gamma$, and we have $\texttt{counter}\ P\ \gamma\ p$, 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.

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 [Otten and Kreitz 1995; Waaler 2001; Wallen 1990]. 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) and ultimately Iris (§5). A comparison between our simplified version and complete connection calculi can be found in §8.

We start by phrasing the problems we have seen in §2 in the context of propositional intuitionistic logic (§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 (§3.2). Finally, we work out a representative example (§3.3), and discuss how our calculus can support domain-specific knowledge via *ground connections* (§3.4),

3.1 Challenges and Key Idea

Recall from §2 that disjunctions pose two challenges for automated proof search. First, when proving $\Delta \vdash P \vee Q$, 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 Diaframe [Mulder et al. 2022]). 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 $\Delta \vdash_{\mathrm{p}} P$ denote an entailment in intuitionistic propositional logic):

$$A \to ((B \wedge C) \vee F), A \vdash_{\mathrm{p}} B \vee F.$$

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.

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

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 $P, [Q] \vdash^{\mathrm{c}}_{\mathrm{p}} A, [\Gamma']$. 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 Fig. 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 §6 for more details how this is done in Coq). Inspired by focusing [Andreoli 1992; Liang and Miller 2009; Simmons 2014], 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 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. Once this is done, one uses the main workhorse F-A.

Suppose our goal is $\Delta \vdash_{\mathrm{p}} \Gamma$ with $A \in \Gamma$ an atom. Rule F-A requires an hypothesis $P \in \Delta$ for which a connection judgment $P, [Q] \vdash^{\mathrm{c}}_{\mathrm{p}} A, [\Gamma']$ can be established. Once such a connection is established, it is sufficient to prove the side-condition Q—we continue with goal $\Delta \vdash_{\mathrm{p}} Q, \Gamma$. Additionally, we need to prove a new goal $\Delta, \bigvee \Gamma' \vdash_{\mathrm{p}} \Gamma$ 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 $\Delta \vdash_{\mathrm{p}} Q, \Gamma$ either establish $\Delta \vdash_{\mathrm{p}} \Gamma$ or $\Delta \vdash_{\mathrm{p}} Q$. In the first case, we are done. In the latter case, our connection tells us that $\Delta \vdash_{\mathrm{p}} A, \Gamma'$. Since for the remaining cases we have $\Delta, \bigvee \Gamma' \vdash_{\mathrm{p}} \Gamma$ the rule is sound.

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

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

Inversion phase

Focusing phase

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

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

Fig. 4. Our calculus for propositional logic based on connections.

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 conjunctions to establish a connection.

Sources of incompleteness. The rules in Fig. 4 are not sufficient for all goals in intuitionistic logic. For example, goal $A \vee B \vdash_\mathrm{p} \top \to A, \top \to B$ is not provable, nor is $A, A \to \bot \vdash_\mathrm{p} B$. 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.

3.3 Example

Fig. 5 contains a possible derivation of $A \to ((B \wedge C) \vee F), A, F \to E \vdash_\mathrm{p} B \vee E$ (this entailment is closely related to the one in §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.
- It is unclear when to use R$\to$, and using it might cause incompleteness.

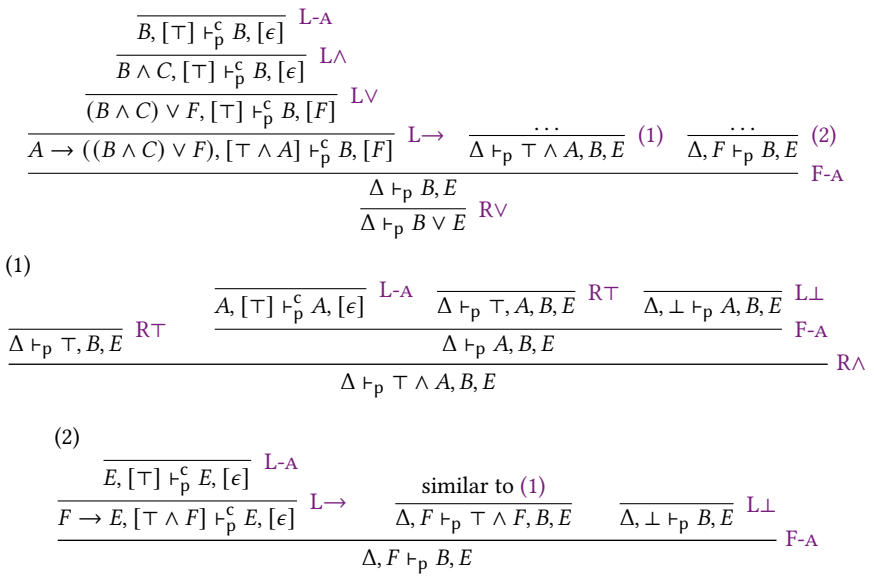

Fig. 5. Example derivation in our propositional calculus (we let $\Delta \triangleq A \to ((B \wedge C) \vee F), A, F \to E$).

- Rule F-A is applicable with the same atom A for two of its premises, which may 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.

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 $0 \le n, n \le 1 \vdash_{\mathrm{p}} (0 < n \wedge n \le 1) \vee n = 0$. 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 $0 \le n, [\top] \vdash^{\mathrm{c}}_{\mathrm{p}} 0 < n, [n = 0]$. 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 §5.3.

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 (§4.1). We then propose an extension of our calculus for separation logic (§4.2). This calculus can be turned into an algorithm, which we explain and apply on an example (§4.3).

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 ($\wedge$ and $\to$) in most proofs. We thus omit regular implication 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 §3, we consider multi-succedent entailments. These are now of the form $\Delta \vdash_s \Gamma$, and should be interpreted as $\ast\, \Delta \vdash_s \bigvee \Gamma$. Unlike §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 (§4.3).

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 $P \vdash_s P * P$ in general. The introduction rule for separating conjunction ($*$) thus differs in that of regular conjunction ($\wedge$) by having to subdivide resources over the conjuncts:

$$\frac{\Delta_1 \vdash_s L \qquad \Delta_2 \vdash_s G}{\Delta_1, \Delta_2 \vdash_s L * G}$$

(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 $A, A -\!\!* (B * C) \vdash_s B * C$, 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 [Mulder et al. 2022]. Essentially, the following two rules are applied eagerly to push an atom A to become the left-most conjunct (*i.e.,* $\Delta \vdash_s A * G$):

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

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.

4.2 Calculus

To adapt the proof rules from §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 $\Delta \vdash_s A * G, \Gamma$ will we look for a connection from some $H \in \Delta$ to A. The connection judgment $H, [L] \vdash_s^c A * [U], [\Lambda]$ also changes: it now contains a parameter U, which we dub the *residue*, that describes the resources from H that are not used to establish A. We interpret $H, [L] \vdash_s^c A * [U], [\Lambda]$ as $H * L \vdash_s (A * U) \vee \bigvee \Lambda$. Carrying around the residue U is not just for convenience or efficiency: unlike for propositional logic, goals can become

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

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

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

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

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

Inversion phase

Focusing phase

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

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

Grammar and definitions

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

$$\begin{aligned} \Delta \vdash_{\mathrm{s}} \Gamma &\triangleq \mathbin{⅋} \Delta \vdash_{\mathrm{s}} \bigvee \Gamma \\ H, [L] \vdash_{\mathrm{s}}^{\mathrm{c}} A * [U], [\Lambda] &\triangleq H * L \vdash_{\mathrm{s}} (A * U) \vee \bigvee \Lambda \\ \Delta &::= \epsilon \mid H, \Delta \\ \Lambda &::= \epsilon \mid U, \Lambda \\ \Gamma &::= \epsilon \mid G, \Gamma \end{aligned}$$

Fig. 6. Our calculus for separation logic based on connections.

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 Fig. 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 §3, yet it looks quite different. To see their relation, we restate F-A, along with an alternative:

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

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:

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

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

Similar to F-ALT, we first establish a connection judgment $H, [L] \vdash^{\mathsf{c}}_{\mathsf{s}} A * [U], [\Lambda]$. 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.

The final ingredient of the inversion phase is UNFOCUS, which should be used *only* when 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 R$*_A$, we will recover the used hypothesis.

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

4.3 Algorithmic Version

Unlike the system in §3, the system in Fig. 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 R$\wedge$, R$\vee$, R$\top$, R$*_\top$, R$*_*$, R$*_\vee$, R$\mathrel{-\!\!*}_\bot$, R$\mathrel{-\!\!*}_\top$, R$\mathrel{-\!\!*}_*$, R$\mathrel{-\!\!*}_\vee$; otherwise:
(2) If the left-most disjunct is a wand, apply R$\mathrel{-\!\!*}_H$; otherwise:
(3) If the left-most disjunct is a lone atom, apply RA; otherwise:
(4) Our goal has shape $\Delta \vdash_{\mathsf{s}} A * G, \Gamma$ with A an atom. Apply 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:

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

Fig. 7. Example derivation in our calculus for separation logic (we let $H \triangleq A \mathrel{-\!\!*} ((B * C) \vee F)$).

(5) Drop the left-most disjunct, by applying UNFOCUS.

Fig. 7 shows our algorithm in action on a slight modification of the example from §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 $H, [A] \vdash_{\mathsf{s}}^{\mathsf{c}} B * [C], [F]$ 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 $\mathrm{R}{*}_A$. 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 $\mathrm{R}{-\!\!*}_H$ 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 §2. Our implementation also provides tools to deal with problematic goals, which we will discuss in §6.

5 IRIS

Although Iris satisfies the rules in §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 [Mulder et al. 2022].

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 (§5.1). We then generalize the connection judgment to include bi-abduction hints (§5.2). We finally show some instances of the bi-abduction version of *ground connections* (introduced for propositional logic in §3.4), and how they are used in the automated verification of ARC (§5.3).

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:

TOKEN-DEALLOCATE-BIABD

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

TOKEN-MUTATE-DECR-BIABD

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

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.

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 $\Delta \vdash_{\mathsf{s}} \Rrightarrow ((\exists \vec{x}.\ L * G_1) \vee G_2)$. The additional disjunct G_2 makes the format multi-succedent, causing disjunctions to be introduced 'lazily' too.

The definition of the connection judgment CONNECTION-DEF, and its application rule $\mathrm{R}\exists *_A$ can be found in Fig. 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:

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

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 $\mathrm{R}\exists *_A$ with $G_2 = \bot$

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

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

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

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

Fig. 8. Bi-abduction hints and connection judgments for Iris's separation logic

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.

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 $A_1, [\vec{x}; L] \vdash^{\mathrm{c}}_{\mathrm{s}} \vec{y}; A_2 * [U], [\Lambda]$, 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 (§2.1), the original Diaframe gets stuck at the problematic entailment PROBLEM-GC-ARC-DROP, namely:

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

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

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

The ε_1 hypothesis is a technical trick from Mulder et al. [2022]. 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 (§2.2), the problematic entailment is PROBLEM-FREE-ARC-DROP, namely:

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

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 $\texttt{counter}\ P\ \gamma\ p$ to $\texttt{no\_tokens}\ P\ \gamma$.

MAYBE-TOKEN-DEALLOC

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

This states that if it is possible that $p = 1$, then $\texttt{counter}\ P\ \gamma\ p$ can *sometimes* be updated to prove $\texttt{no\_tokens}\ P\ \gamma$, if we additionally provide a $\texttt{token}\ P\ \gamma$. 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 $\texttt{counter}\ P\ \gamma\ p$ and $\texttt{token}\ P\ \gamma$, 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.

6 SOUNDNESS PROOF AND IMPLEMENTATION IN COQ

Soundness. In our artifact [Mulder et al. 2023] 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 (§3), propositions are interepreted as Coq's propositions `Prop`. For the separation-logic versions (§ 4 and 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. 2018, 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 §4.3, extended with the support for higher-order quantification and modalities described in §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* $\mathrm{R}{-\!\!*}_H$: $A \mathrel{-\!\!*} (B * C), A, F \vdash_{\mathsf{s}} (B \lor F) * A$. The `iSteps` tactic makes a bad choice here: after finding a connection from $A \mathrel{-\!\!*} (B * C)$ to 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 $\mathrm{R}{-\!\!*}_H$ 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 $\mathrm{R}{-\!\!*}_H$. The user can then manually pick the correct side of

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 [Calcagno et al. 2007; Vafeiadis 2008]	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

Fig. 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.

the disjunct using the standard Iris Proof Mode tactics. Second is the `iSmash` tactic, which *will* backtrack to try UNFOCUS if $\mathrm{R}\text{-}\!\!*_H$ failed to produce a proof. This is sufficient for Peterson [1981]'s algorithm and the simple example, but will still fail on $A \mathrel{-\!\!*} B \lor C, A \vdash_s F \mathrel{-\!\!*} B, F \mathrel{-\!\!*} C$.

- *The algorithm wrongfully drops a disjunct with UNFOCUS:* $A \land B, C \vdash_s (A \lor F) * C$. Our calculus has no support for regular conjunction in hypotheses, which means that no connection to A can be established. After UNFOCUS, we thus get stuck at the unprovable goal $A \land B, C \vdash_s F * C$. 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.

7 EVALUATION

We evaluate our approach by verifying the examples in Diaframe's original benchmark. Fig. 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 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 §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. Mulder et al. [2022] report stable verification times on their benchmark of failing examples in the original Diaframe, but 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 §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.

8 RELATED WORK

Focusing and connection calculi. Focusing [Andreoli 1992; Liang and Miller 2009; Simmons 2014] 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 [Otten and Kreitz 1995; Waaler 2001; Wallen 1990] 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 §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]; 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 [Lee and Park 2014; Piskac et al. 2014b; Reynolds et al. 2016], 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 [Mulder and Krebbers 2023] recently. 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.

ACKNOWLEDGMENTS

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

ARTIFACT AVAILABILITY

Our artifact [Mulder et al. 2023] contains a mechanization of our results in the proof assistant Coq. The artifact is persistently hosted on Zenodo, containing both a Virtual Machine with the dependencies pre-installed, and the complete source code along with compilation instructions. Instructions for evaluating the artifact are also available on Zenodo. The latest version of Diaframe can be found at https://gitlab.mpi-sws.org/iris/diaframe.

REFERENCES

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

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

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

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, 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

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

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

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

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

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

Albert Grigor'evich Dragalin. 1988. *Mathematical Intuitionism: Introduction to Proof Theory*. Translations of Mathematical Monographs, Vol. 67. American Mathematical Society.

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

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

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

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, 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

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. 2018. 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, 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
Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772
Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26
Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic *(POPL)*. 205–217. https://doi.org/10.1145/3009837.3009855
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
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
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
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
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
Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023. Artifact and Appendix of 'Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic'. https://doi.org/10.5281/zenodo.7799173
Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *PACMPL* 7, OOPSLA1 (2023), 91:462–91:491. https://doi.org/10.1145/3586043
Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris *(PLDI)*. 809–824. https://doi.org/10.1145/3519939.3523432
Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2
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. 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, 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
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
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
David J. Pym. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series, Vol. 26. Kluwer.
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, 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

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

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

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

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

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.

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

Received 2022-11-10; accepted 2023-03-31