Unification for Subformula Linking under Quantifiers

Ike Mulder
Radboud University Nijmegen
Netherlands
me@ikemulder.nl

Robbert Krebbers
Radboud University Nijmegen
Netherlands
mail@robbertkrebbers.nl

Abstract

Subformula linking is a technique that allows one to simplify proof goals by identifying subformulas of hypotheses that share atoms with the goal. It has been used by recent prototypes for gesture-based interactive theorem proving, but also for theorem proving in separation logic.

When linking formulas, we should avoid information loss, *i.e.,* subformula linking should succeed precisely when a provable simplification can be generated. Avoiding information loss is challenging when quantifiers are involved. Existing approaches either generate simplifications that involve equalities, or determine substitutions for variables via unification. The first approach can produce unprovable simplifications, while the second approach can fail to find desired links.

We propose a third approach, called *Quantifying on the Uninstantiated (QU)*, which is also based on unification and lies between the two existing approaches. We show that QU has practical applications for proof automation, by improving tactics for resource framing in the Iris framework for separation logic in Coq.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Logic and verification**; *Separation logic*; *Higher order logic*.

Keywords: Coq, subformula linking, unification, higher-order logic, separation logic, program verification

ACM Reference Format:
Ike Mulder and Robbert Krebbers. 2024. Unification for Subformula Linking under Quantifiers. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3636501.3636950

1 Introduction

Suppose you are faced with the following proof obligation:

$$(\forall x.\, Px \to \exists y.\, Qxy) \vdash \exists y.\, \exists z.\, Q(fz)y. \tag{1}$$

This work is licensed under a Creative Commons Attribution 4.0 International License.

CPP '24, January 15–16, 2024, London, UK
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0488-8/24/01
https://doi.org/10.1145/3636501.3636950

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

We follow Chaudhuri [4] and call this problem *subformula linking*. Subformula linking is used for recent work on 'gestural' theorem proving for intuitionistic first-order logic, which continues work on proof by pointing by Bertot et al. [2]. One gestural prover is ProfInt [5, 6], where one can link two subformulas by selecting them with a mouseclick. Another gestural prover is Actema [13], where one can drag and drop a hypothesis on a goal, prompting the system to link subformulas, and compute a remaining proof obligation.

A variant of subformula linking also shows up in other logics. The *framing* problem [1, 22] in separation logic is about canceling occurrences of the same atom in the hypothesis and goal. For example, we may wish to cancel out (or 'frame') the atom R in $R * Q \vdash \exists x.\, S\,x * R$, and continue by proving $Q \vdash \exists x.\, S\,x$. The Iris framework for concurrent separation logic in Coq [18, 19, 21, 24–26] has a tactic called `iFrame`, which can perform the above framing. The implementation of this tactic essentially solves a subformula linking problem.

The previous examples all originate from interactive theorem proving. However, subformula linking is also useful in the setting of automated theorem proving. Diaframe [27, 29, 30]—a recent tool for automated proofs of concurrent programs using Iris in Coq—also makes (implicit) use of subformula linking. Consider a (slightly simplified)[1] verification goal in Diaframe that occurs in the verification of Courtois et al. [8]'s classic readers-writer lock:

$$(\forall R.\, R \mathrel{-\!\!*} \exists \gamma.\, \mathsf{is\_lock}\, \gamma\, v\, R) \vdash \exists \gamma_1 \gamma_2.\, \mathsf{is\_lock}\, \gamma_1\, v\, (S\, \gamma_2). \tag{2}$$

This entailment is similar to the previous example, but it uses higher-order quantification and the magic wand ($\mathrel{-\!\!*}$), which is the substructural version of implication ($\to$) in separation logic. The is_lock predicate [12, 16] states that value v is a lock with name γ, and guards resource R—although the precise semantics does not matter for now: it should be viewed as an abstract predicate. The key characteristic of Equation (2) is that we want to simplify it to $\exists \gamma_2.\, S\,\gamma_2$, a goal that Diaframe can prove automatically.

[1] Iris's update modality $\Rrightarrow$ is omitted from Equation (2) and the remaining proof obligation $\Rrightarrow \exists \gamma_2.\, S\,\gamma_2$. Because $\Rrightarrow S\,\gamma_2'$ is not provable for any constant γ_2', it is crucial that we keep the existential quantification, and do not prematurely instantiate γ_2 with an evar.

Unfortunately, previous work on subformula linking does not produce satisfactory solutions for these examples. In the realm of first-order logic, Actema [13] cannot establish a link for Equation (1) and thus fails. While ProfInt [5, 6] can establish links, there are many candidate links, and not all of them are provable (*i.e.,* some are equivalent to $\bot$). We could backtrack on all candidate links, but that would be detrimental for performance when applied to proof automation for concurrent programs [27]. In the realm of separation logic, the examples are simply out of scope of Iris's `iFrame` because it does not consider subterms of the hypothesis.

Quantifiers pose problems for existing approaches to subformula linking. To understand why, we briefly discuss the setup of existing approaches for linking under quantifiers. ProfInt and Actema have recursive procedures that traverse the hypothesis and goal to find a shared atom. When both the hypothesis and goal are a logical connective (*i.e.,* not an atom), one needs to choose to either proceed in the hypothesis or goal. While this choice is unspecified in the mathematical presentation of these systems (as a non-deterministic inductive relation), an implementation needs to make a concrete choice. This choice matters—just like it matters in which order the ordinary proof rules for introduction and elimination are used—and might result in finding different links, or no links at all. As the paper on ProfInt [5, §2.1] remarks, this is a challenging problem:

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

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

ProfInt establishes unwanted links due to scoping issues. ProfInt [5] links atoms by posing *equality* constraints. Syntactically equal predicates are linked by requiring the user to prove that all their arguments are equal, *e.g.,* $P\,x \vdash P\,y$ will be reduced to $x \doteq y$. Although this reduction seems innocent, it means that ProfInt can establish links under quantifiers regardless of whether the order of traversal respects variable scoping. For example, ProfInt can establish two links for $\exists x.\, P\,x \vdash \exists y.\, P\,y$. The first link produces the tautology $\forall x.\, \exists y.\, x \doteq y$ as its simplification. The second link produces $\exists y.\, \forall x.\, x \doteq y$, which is logically equivalent to false (if the domain of quantification is non-trivial). To use subformula linking in automated theorem proving, it would be helpful if such unwanted links are simply ruled out altogether. In particular, if there are no sensible links at all, this means that the automation can immediately move on to the next hypothesis without having to backtrack.

Actema cannot establish desired links. Actema [13] rules out some of ProfInt's unwanted links, but it actually rules out too many links. Actema uses *unification* to determine the appropriate order to traverse below quantifiers. If two atoms are not unifiable, they cannot be linked: $P\,x \vdash P\,y$ will thus not be reduced to $x \doteq y$ if x and y are different variables. When linking the previous example $\exists x.\, P\,x \vdash \exists y.\, P\,y$, Actema will unify y on the right-hand side (*i.e.,* introduce the $\exists$ in the goal) with the x obtained from the left-hand side (*i.e.,* by eliminating the $\exists$ in the hypothesis). This can *only* be done if the $\exists$ in the hypothesis is eliminated first—which rules out the unwanted/ill-scoped linking ProfInt finds.

Actema uses unification whenever quantifiers need to be instantiated with a specific term (*i.e.,* $\forall$-quantifiers in hypotheses/the 'left', and $\exists$-quantifiers in goals/the 'right'). For example, Actema finds that x in Equation (1) must be of shape $f\,?z$ for some unknown z, since this causes the arguments of Q to match syntactically. Actema has two rules available to derive a link: one for when x is unified with a concrete term t, and one for when x does not get unified at all. Unfortunately, in Equation (1), neither rule is applicable—although x has been unified with $f\,?z$, it contains the uninstantiated $?z$, meaning $f\,?z$ is not a concrete term. As such, Actema cannot establish a link for Equation (1).

Our approach: Quantifying on the Uninstantiated. We propose a new system called **QU** for linking subformulas under quantifiers. Like Actema, we use unification to rule out unwanted links due to scoping errors. However, QU improves upon Actema by being able to link subformulas with non-trivial quantifier instantiation, such as the examples in this section. Our approach is to *Quantify on the Uninstantiated* (QU). Consider Equation (1), where the x gets unified with $f\,?z$, and $?z$ is uninstantiated. QU quantifies precisely on this z, *i.e.,* we produce the simplification $\exists z.\, P(f\,z)$. On the implementation level, we use evars (existential variables) and convert these back into existential quantifiers.

Anti goals. QU does not specify if hypothesis-rules ('left' rules) or goal-rules ('right' rules) should be given priority when mixing quantifiers with the propositional connectives (conjunction, disjunction, implication)—this remains an open problem in general subformula linking. Both the Actema [13] and ProfInt [6] implementation use heuristics and/or backtracking to make this choice, and so do we. QU nevertheless helps in eliminating unwanted links from the search space.

Applications. Quantifiers are problematic for subformula linking regardless of the logic of the system—*i.e.,* first-order, higher-order, and separation logic systems face essentially the same problem. The idea of QU is not tied to a specific logic, however. To demonstrate this, we use QU to improve Iris's `iFrame` tactic for higher-order separation logic—making it strictly more powerful, with comparable performance. We also explain how Diaframe makes use of QU.

Contributions and artifacts.

- We present the QU rules for linking subformulas under quantifiers (§ 3). We formally prove in Coq that our system lies between Actema and ProfInt, using a deep embedding of first-order logic by Kirst et al. [23].
- We present a simple, shallowly embedded subformula linking procedure in Coq (§ 4). This demonstrates that QU can be implemented and used inside Coq.
- We extend and improve Iris's `iFrame` tactic with the QU rules (§ 5.1), demonstrating the practical applicability of our approach.
- We describe how Diaframe [30] uses the QU rules to verify Courtois et al. [8]'s readers-writer lock (§ 5.2).

We start with some background on subformula linking (§ 2). We conclude with an evaluation of the improved `iFrame` (§ 6) and a discussion of related work (§ 7). The Coq sources of these artifacts are in the supplementary material [28].

2 Background on Subformula Linking

To provide background on subformula linking and to compare existing systems, we give a uniform presentation of subformula linking (§ 2.1), and formulate ProfInt (§ 2.2) and Actema (§ 2.3) as instances.[2] Although many rules are shared by ProfInt and Actema, their differences are significant for the links they can derive. We give examples where ProfInt establishes unwanted links due to scoping issues with quantifiers, and where Actema cannot establish desired links, before presenting our system QU (§ 3).

2.1 Subformula Linking Judgment

We consider first-order intuitionistic logic with equality (our implementations in § 4 and 5 scale to higher-order separation logic). Terms, atoms, propositions, and proof contexts are inductively defined as:

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

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

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

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

[2]We discuss differences between the original formulation of ProfInt and Actema and our unified presentation in § 7.

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

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

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

The inference rules for establishing $H \curlywedge [O] \Vdash G$ in Profint and Actema will be given as inductively-defined relations in § 2.2 and 2.3. These relations should be interpreted as (non-deterministic) recursive 'procedures' that compute an appropriate O for a given H and G. That is, the rules will follow the structure of the hypothesis H and goal G. We use the convention to put 'outputs' of relations, such as O, between brackets [and]; other parameters are 'inputs'.

2.2 Rules for ProfInt

The rules for ProfInt's linking judgment $H \curlywedge [O] \Vdash G$ are given in Fig. 1a and 1b. Their purpose is to find a shared atom in H and G that can be linked. The base case is Cong-profint in Fig. 1b. This rule applies if the hypothesis and goal are the same predicate P, resulting in a simplification that says their arguments should be equal.

The recursive rules in Fig. 1a traverse the formula in a non-deterministic fashion to reach the base case. They come in two categories: 'left' rules for the hypothesis H and 'right' rules for the goal G. All rules compute a simplification O based on the simplification for the recursive call.

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

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

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

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

$$\text{L}\exists\ \frac{\forall x.\quad H \curlywedge [O] \Vdash G}{(\exists x.\, H) \curlywedge [\forall x.\, O] \Vdash G} \qquad \text{L}\forall\ \frac{\forall x.\quad H \curlywedge [O] \Vdash G}{(\forall x.\, H) \curlywedge [\exists x.\, O] \Vdash G} \qquad \text{R}\wedge\ \frac{H \curlywedge [O] \Vdash G_i \qquad i \in \{1,2\}}{H \curlywedge [O \wedge G_{3-i}] \Vdash G_1 \wedge G_2} \qquad \text{R}\vee\ \frac{H \curlywedge [O] \Vdash G_i \qquad i \in \{1,2\}}{H \curlywedge [G_{3-i} \vee O] \Vdash G_1 \vee G_2}$$

$$\text{R}\rightarrow\ \frac{H \curlywedge [O] \Vdash G_2}{H \curlywedge [G_1 \rightarrow O] \Vdash G_1 \rightarrow G_2} \qquad \text{R}\forall\ \frac{\forall x.\quad H \curlywedge [O] \Vdash G}{H \curlywedge [\forall x.\, O] \Vdash \forall x.\, G} \qquad \text{R}\exists\ \frac{\forall x.\quad H \curlywedge [O] \Vdash G}{H \curlywedge [\exists x.\, O] \Vdash \exists x.\, G}$$

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

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

(b) Subformula rules specific to ProfInt.

$$\textsc{Asmp-actema}\ \frac{}{A \curlywedge [\top] \Vdash A} \qquad \text{L}\forall_n\textsc{-actema}\ \frac{H[t/x] \curlywedge [O] \Vdash G}{(\forall x.\, H) \curlywedge [O] \Vdash G} \qquad \text{R}\exists_n\textsc{-actema}\ \frac{H \curlywedge [O] \Vdash G[t/x]}{H \curlywedge [O] \Vdash \exists x.\, G}$$

(c) Subformula rules specific to Actema.

Figure 1. Subformula rules in ProfInt and Actema.

When using ProfInt's implementation [6], the user does not have to construct this derivation by hand. Instead, the user clicks on the occurrences of B in H and G. This click instructs the implementation to derive the linking judgment, and to transform goal $B \wedge D$ into $A \wedge (\top \wedge D)$ with link-apply. (Both ProfInt and our implementation in § 5 remove the superfluous occurrence of $\top$, *i.e.,* give $A \wedge D$. We ignore these simplifications for brevity's sake.) This example shows that by clicking on the occurrences of B in H and G, ProfInt essentially eliminates an implication and a conjunction.

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

In general, the rule order matters. Consider finding an O with $(A \rightarrow B) \curlywedge [O] \Vdash (A \rightarrow B) \vee C$. By prioritizing the 'left' rules, we find $O_1 = A \wedge (C \vee (A \rightarrow \top))$ using L$\rightarrow$, R$\vee_1$, R$\rightarrow$. By prioritizing the right rules, we find $O_2 = C \vee (A \rightarrow (A \wedge \top))$ using R$\vee_1$, R$\rightarrow$, L$\rightarrow$. The first simplification results in information loss: O_1 is equivalent to A. The second simplification O_2 is equivalent to $\top$, and thus desired.

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

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

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

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

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

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

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

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

2.3 Rules for Actema

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

The rules for Actema's linking judgment $H \wedge [O] \Vdash G$ are given in Fig. 1a and 1c. Instead of generating a simplification that involves equalities, Actema uses *unification* to determine appropriate ways to eliminate universal quantifiers, and introduce existential quantifiers. The base case ASMP-ACTEMA requires the atoms to match *exactly*. This requirement can be met under quantifiers since Actema has the rules $\mathrm{L}\forall_n$-ACTEMA and $\mathrm{R}\exists_n$-ACTEMA. The premises of these rules allow one to instantiate the quantifier with a specific term t, which we are free to choose. For example, Actema can directly find $P\,t \wedge [\top] \Vdash \exists x.\, P\,x$ with $\mathrm{R}\exists_n$-ACTEMA and ASMP-ACTEMA by instantiating x with t.

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

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

Let us reconsider the example from § 2.2 of finding a linkage $(\forall x.\, \exists y.\, Qxy) \wedge [O] \Vdash \exists z.\, Qtz$. If one were to start with $\mathrm{R}\exists_n$-ACTEMA, one needs to instantiate the evar $?z$ with a variable y, which is not in scope when $\mathrm{R}\exists_n$-ACTEMA was used. This fails, prompting Actema to try the correct rule order, *i.e.,* prioritizing the 'left' rules, resulting in the desired simplification $O = \forall y.\, \top$.

Unification guides Actema in the search for an appropriate rule order. In some cases, unification rules out a bad linkage completely. Consider $(\forall x.\, \exists y.\, R\,x\,y) \wedge [O] \Vdash \exists z.\, \forall x'.\, R\,x'\,z$. ProfInt would propose an unprovable simplification O, while Actema fails to establish a linkage. From a proof automation point of view (*e.g.,* for our applications in § 5), Actema's behavior of failing instead of finding an unprovable linkage is preferable. It prompts the automation to consider another hypothesis for finding a linkage.

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

$$(\forall x.\, Px \to \exists y.\, Qxy) \wedge [O] \Vdash \exists z.\, \exists x'.\, Q(f x')\, z \qquad (3)$$

One would expect to find $O = \exists x.\, P(fx)$ for Equation (3), but this linkage is not derivable in Actema, and neither is any other O. To see why, note that the desired z in the goal is obtained from the existentially quantified y in the hypothesis. This means we must start with a 'left' rule for the universal quantifier, choosing between $\mathrm{L}\forall_n$-ACTEMA and $\mathrm{L}\forall$. A linkage does not exist for every x, so $\mathrm{L}\forall$ fails. We would like to use $\mathrm{L}\forall_n$-ACTEMA with some $t = f\,?x'$, but no appropriate instance for $?x'$ is in scope. We can only get access to such an instance by first using $\mathrm{R}\exists$ twice—which means we have a circular dependency.

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

3 Quantifying on the Uninstantiated

We present our system Quantifying on the Uninstantiated (QU). Compared to Actema, we do not choose between instantiation or quantification—rather, we quantify precisely on the parts that remain uninstantiated. We start by discussing the rules of QU (§ 3.1) and explain how QU is used on examples (§ 3.2). We finally discuss the proof-theoretic properties of QU (§ 3.3)—we show that the strength of the linkages from QU lies between Actema and ProfInt.

3.1 Rules for QU

QU features the rules from Fig. 1a, except $\mathrm{L}\forall$ and $\mathrm{R}\exists$. We have ASMP-ACTEMA as a base case, and the following rules for existential and universal quantifiers:

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

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

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

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

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

The other quantifier rules $\mathrm{L}\exists$ and $\mathrm{R}\forall$ stay the same, since these correspond to reversible inference rules.

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

Figure 2. An example linkage in QU.

3.2 QU by Example

We show how QU goes beyond Actema by deriving the linkage from the example in §2.3:

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

The full derivation is included in Fig. 2. The key step is the use of $\textsc{L}\forall_{QU}$, where we pick $\vec{y} = [u]$ and $t = fu$. Intuitively, this choice makes the arguments of Q in hypothesis and goal match precisely, so that the Asmp-actema can be applied in the base case. We also use $\textsc{R}\exists_{QU}$ twice with empty quantifier list (*i.e.,* $\vec{y} = []$), which simplifies to Actema's $\textsc{R}\exists_n$-actema.

The application of $\textsc{L}\forall_{QU}$ with $\vec{y} = [u]$ and $t = fu$ in Fig. 2 is not expressible in Actema, and crucial for getting the desired linkage in this example. However, it is reliant upon somehow making the correct choice for $\vec{y}$ and t. Additionally, so far we have only seen cases where $\vec{y}$ is a list of length at most 1. We will consider another example to demonstrate how we can determine appropriate choices for $\vec{y}$ and t, and that we sometimes need $\vec{y}$ to be a longer list. We will discuss the actual implementation that chooses $\vec{y}$ and t in Coq in §4. The example is as follows:

$$(\forall x.\,\exists y.\,Q\,x\,y) \wedge [O] \Vdash \exists z.\,\exists u.\,\exists v.\,Q(g\,u\,v)\,z \qquad (4)$$

Consider the following partial derivation of a linkage:

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

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

$$?t[] = g\,?u[y]\,?v[y]$$

Here, we write $?s[\vec{x}]$ for an evar $?s$ which has variables $\vec{x}$ in scope. We cannot instantiate $?t$ to be $g\,?u[y]\,?v[y]$, since the evars $?u$ and $?v$ have y in scope, and $?t$ does not. To proceed, the unification algorithm now *prunes* the term on the right-hand side [39, §4.3.1]. This comes down to first creating new evars $?u'[]$ and $?v'[]$, then instantiating $?u[y] = ?u'[]$ and $?v[y] = ?v'[]$. At that point, the unification algorithm instantiates $?t[] = g\,?u'[]\,?v'[]$. Let us return to our derivation, with $?t$ filled in:

$$\textsc{L}\forall_{QU}\ \dfrac{\forall \vec{y}.\ \ \ (\exists y.\,Q\,(g\,?u\,?v)\,y) \wedge [\forall y.\,\top] \Vdash \ldots}{(\forall x.\,\exists y.\,Q\,x\,y) \wedge [\exists \vec{y}.\,\forall y.\,\top] \Vdash \ldots}$$

We now want to *quantify on the uninstantiated.* That is, $\vec{y}$ will contain a variable for each evar that remained uninstantiated in t.[3] Here we pick $\vec{y}$ to be the two-element list $[u''; v'']$ and instantiate $?u'[] = u''$ and $?v'[] = v''$. This results in the following derivation:

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

We will explain how this can be done automatically in §4.3.

3.3 Comparison to ProfInt and Actema

We prove some results about the relative strength of Profint, Actema and QU. We have mechanized these results in Coq using the library for first-order logic by Kirst et al. [23]. This library provides a deep embedding of terms, connectives, propositions, and proofs. We inductively define the three linking judgments, which we disambiguate using subscripts. For example, $H \wedge [O] \Vdash_{\text{ACTEMA}} G$ is the inductively-defined linking judgment of Actema.

First, we prove that all linkage systems are sound:

Theorem 3.1 (Soundness)**.**

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

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

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

Theorem 3.2 (Actema vs. QU)**.** *If $H \curlywedge [O] \Vdash_{\text{ACTEMA}} G$, then $H \curlywedge [O] \Vdash_{\text{QU}} G$.*

This theorem holds because the quantifier rules R$\exists_{QU}$ and L$\forall_{QU}$ of QU generalize those of Actema (§ 3.1).

Note that this theorem states that Actema linkages are expressible in QU, which does not guarantee that the procedure we informally describe in § 3.2 actually finds Actema's solution. This would be harder to formalize because it depends on Coq's unification algorithm. We nevertheless think our solutions would agree with Actema. Actema only uses L$\forall_n$-ACTEMA if it can unify term t with a concrete term. Concrete terms do not contain evars/uninstantiated terms, so we find the same solution. If instead Actema uses L$\forall$, the term t must have remained an evar, and so our approach chooses precisely that evar to quantify on.

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

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

We can then relate the linkages from QU and ProfInt.

Theorem 3.3 (QU vs. ProfInt)**.** *Let p be a sequence of linking rules. If $H \curlywedge [O] \Vdash_{\text{QU}}^{p} G$, then there is a unique O' for which $H \curlywedge [O'] \Vdash_{\text{PROFINT}}^{p} G$, and additionally $O \vdash O'$.*

In other words, for a given rule order, if QU can derive a linkage, then ProfInt can also derive a linkage. Additionally, QU's simplification O is at least as hard to prove as ProfInt's simplification O'. Another way to read Theorem 3.3 is that the instantiations made by QU are guaranteed to satisfy the equalities from ProfInt.

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

Completeness. Chaudhuri [5] showed that the full ProfInt linkage system is complete. The full ProfInt system differs from the presentation in § 2.2 in two regards: one can link two hypotheses, and one can also link within formulas. Subformula linking within formulas allows ProfInt to prove entailments such as

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

since $A \rightarrow \bot$ and $\bot$ can be linked *within* the left disjunct. This entailment is out of reach for our linkage system, since no subformula of the hypothesis $A \vee B$ can be linked to a subformula of the goal (note that our system only considers subformulas of the right-hand side of an implication). This incompleteness is acceptable for the application we have in mind, *i.e.,* proof automation for (higher-order) separation logic—implications do not frequently occur inside disjunctions in this setting [27].

4 Implementation

We demonstrate that QU can be effectively implemented in the Coq proof assistant. We start by defining linkages and linkage rules for the propositions `Prop` of Coq's higher-order logic (§ 4.1). (This makes it different from § 3.3 where we performed meta-theoretic reasoning on a deep embedding of first-order logic.) We then define simple *telescopes* (§ 4.2), which form a building block for representing n-ary functions and n-ary quantification. Telescopes allow us to state the L$\forall_{QU}$ and R$\exists_{QU}$ rules properly and to implement custom Ltac [11] code that assists in solving the unification problems arising in QU (§ 4.3). Finally, we make the LINK-APPLY inference rule available as a Coq tactic (§ 4.4).

4.1 Linkages in Coq

We start by defining linkages *semantically* in Coq:

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

This defines a type class [36] called `Link`, for which we will define the notation `LINK H ⋏ [O] ⊫ G`. To construct an instance `LINK H ⋏ [O] ⊫ G`, one must prove `H ∧ O → G` (*i.e.,* soundness). A semantic definition like this makes it particularly easy to define linkage rules as type class instances:

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

Instances for the other rules from Fig. 1b are similar. The key step is to define an appropriate instance for our new rule L$\forall_{QU}$. Let us repeat the statement from §3.1:

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

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

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

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

The first issue is that while the n-ary version can be derived from 1-ary function through currying, this results in a complicated simplification. We want the resulting simplification to be an n-ary existential quantification, instead of a unary quantification on a product Y. In particular, we want to avoid a useless quantification over `u : unit` if no variables are needed. Generating an n-ary existential quantification is important to show readable goals to the user and to aid automation in making further progress. We address this issue using *telescopes* to write an n-ary rule (§4.2).

The second issue is that of determining appropriate terms for `Y`, `t` and `O`. We can use evars ?`Y`, ?`t` and ?`O` when applying this lemma, but instantiating these evars is challenging. At some point we want to unify *e.g.,* ?`t y` with a concrete term—while also instantiating the type ?`Y` of `y`. We do not know of any existing unification algorithm that supports this kind of problem. Indeed, Coq's default unification algorithm rightfully refuses to solve this problem. We address this issue using a custom tactic written in Ltac (§4.3).

4.2 Simple Telescopes in Coq

Telescopes [10] can represent (the type of) sequences of variables with possibly dependent types. For the applications in §5, we use the formalization of (dependent) telescopes provided by the coq-std++ library [38].[4] For brevity, the telescopes in this section do not allow dependent types.

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

[4] The coq-std++ library [38] defines `tele` as a custom inductive type to handle dependent types. Furthermore, by making this inductive type universe polymorphic [37], coq-std++ avoids universe constraints that would otherwise restrict the usage of telescopes in larger developments.

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

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

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

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

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

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

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

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

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

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

4.3 Quantifying on the Uninstantiated with Ltac

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

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

This lemma differs from `link_l_forall_qu_v1` in §4.1 in its use of `Y : tele` and `tele_ex`.

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

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

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

After the application of this tactic, our goal is:

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

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

Let us now repeat Equation (4) from §3.2 and consider our proof obligations. We are trying to derive:

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

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

$$g\, ?\mathtt{u}\, ?\mathtt{v} = ?\mathtt{t}\, \mathtt{y}, \tag{5}$$

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

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

We want to quantify on the uninstantiated, so our goal is to infer `Y = [tele U; V]`. Evars `?u` and `?v` should be unified with projections of `y`, so that the remaining unification problem can be solved by Coq. This is what the Ltac script `solve_evar_tele_equality` from Fig. 3 does.

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

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

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

We still need to prove the equality once this is finished. The equality in Equation (5) has been reduced to

```
g (fst y) (fst (snd y)) = ?t y.
```

and 'exact (eq_refl _)'[6] can make quick work of this: Coq's unification algorithm [39, step 4 on page 186] is able to infer an appropriate value for `t` now.

4.4 Linkage Tactic

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

To call the custom Ltac from § 4.3 we do not register `link_l_forall_qu_v2` as a regular instance. Instead we add an external hint to the type class database:

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

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

Putting it all together. With type class instances for all the linkage rules in place, we obtain a very simple implementation of a linkage system in Coq—in about 250 lines of code in total. This includes the straightforward implementation of a tactic `link_to` that performs link-apply, omitted here. All linkages that were discussed before have the desired result. In the supplementary material [28], we have included solutions to some of the exercises in Actema's course on first-order logic, to demonstrate our linkage system.

Non-determinism. We have not specified a heuristic that determines in what order the linkage rules should be applied. By implementing the linkage rules as type class `Instance`s, we implicitly rely on the backtracking semantics of type class search—all orders will be considered. This means linking only fails after all possible orders of linking rules have been considered, which is not great from a performance perspective. Similar to ProfInt and Actema, we use heuristics to determine the rule order in our applications, and thereby avoid this inefficiency. We either have no 'left' rules (`iFrame` in §5.1), or we prioritize 'left' rules (Diaframe in §5.2).

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

```
Ltac solve_evar_tele_equality :=
  lazymatch goal with
  | ⊢ ?l = ?f ?arg ⇒
    let rec retcon_tele the_arg T :=          (* we receive the_arg : tele_arg T *)
      match l with
      | context [?term] ⇒                     (* look through all subterms of l *)
        is_evar term;                          (* check that the subterm term of l is an evar *)
        let X := type of term in
        let T' := open_constr:(_) in           (* creates a new evar T' *)
        unify T (teleS X T');                  (* instantiates T to be X :: T' *)
        unify term (fst the_arg);              (* instantiates term to be fst arg, which now has type X *)
        retcon_tele (snd the_arg) T'           (* .. now repeat this for other evar subterms of l *)
      | _ ⇒ unify T tele0                     (* if l has no remaining evars, instantiate T to be the empty list [] *)
      end
    in
    let T' := lazymatch type of arg with tele_arg ?T ⇒ T end in
    retcon_tele arg T';
    exact (eq_refl _)                          (* unification can now instantiate f *)
  end.
```

Figure 3. Ltac code for QU.

5 Applications

We demonstrate that the QU approach has practical applications outside pure intuitionistic logic. First, we apply QU to the *framing* problem [1, 22] from separation logic in the context of the Iris framework for concurrent separation logic in Coq [18, 19, 21, 24–26] (§5.1). Second, we apply QU to the Iris-based proof automation framework Diaframe [27, 29, 30] (§5.2), where we show that the automatic verification of a classical readers-writer lock crucially relies on the QU rules for subformula linking under quantifiers.

5.1 Framing under Quantifiers in Separation Logic

Separation logic [31] is an extension of Hoare logic that allows one to reason modularly about the correctness of stateful programs. We focus on the assertion language of separation logic, which extends ordinary logic with two logical connectives that enable this modular reasoning: the separating conjunction ($*$) and magic wand ($-\!\!*$). Separating conjunction can be seen as a *substructural* version of conjunction ($\land$), which means that we cannot use separation logic propositions P more than once—in particular, $P \vdash P * P$ does not hold in general. The introduction rule of separating conjunction thus requires one to split the list of hypotheses over the conjuncts:

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

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

Iris comes with an interactive proof mode and accompanying tactics [24, 26], whose `iFrame` tactic can be used to frame away hypotheses in the goal. This tactic is implemented with a type class `Frame`, exactly like `Link` from §4.1. The (slightly simplified) definition of `Frame` is:

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

Compared to `Link`, the `Frame` class involves the separating conjunction ($*$) instead of the regular conjunction ($\land$). Furthermore, `Frame` works in a generic object logic `PROP` of type `bi`. This makes the `Frame` type class applicable in any Bunched Implication logic [32, 33], *i.e.,* logics that satisfy the relevant axioms for $*$ and $-\!\!*$.

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

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

Framing under existential quantification. To frame beneath quantifiers, one faces problems similar to those described in §2. However, there are also some differences that we discuss first. When framing, we only look for hypotheses that appear (nearly) verbatim in the goal—meaning there are only right rules for `Frame`. The existing implementation for framing under existential quantifiers only provides R$\exists$ and not R$\exists_n$-ACTEMA. This means that framing could not instantiate quantifiers, and so framing fails on *e.g.,* $P\ 1 \vdash (\exists n.\ P\ n) * Q$.

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

By quantifying on the uninstantiated, we can allow framing to instantiate quantifiers without this exponential slowdown. Additionally, remember that the QU rule $R\exists_{QU}$ is strictly more general than having both $R\exists_n$-ACTEMA and $R\exists$. We use the following `Frame` instance, similar to the linking instance `link_l_forall_qu_v2` from §4.3.

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

The main difference is the use of `bi_texist`, which does n-ary existential quantification in `PROP`. We also reuse the tactics from §4.3 for proving the equalities on `t'` and `O'`.

5.2 Automatic Verification of a Readers-Writer Lock

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

Let us consider the automatic verification of the classic readers-writer lock by Courtois et al. [8] in Diaframe. A lock is a data structure from concurrent programming, in charge of sharing access to a resource R among multiple threads. It guarantees that at all times, at most a single thread can access resource R. A readers-writer lock generalizes a regular lock: it guarantees that either there are zero or more 'readers', *i.e.*, threads with read-only access to R, or there is a single 'writer' thread that can mutate R.

The classic readers-writer lock implementation by Courtois et al. [8] is built from two regular locks. We shall consider the verification of *allocating* a new readers-writer lock, which first allocates two regular locks. Let us first consider two specifications for allocating a regular (spin) lock:[7]

$$\{R\}\ \mathsf{new\_lock}()\ \{v.\ \exists\gamma.\ \mathsf{is\_lock}\,\gamma\,v\,R\} \tag{6a}$$

$$\{\mathsf{True}\}\ \mathsf{new\_lock}()\ \{v.\ (\forall R.\ R \mathrel{-\!\!*} \exists\gamma.\ \mathsf{is\_lock}\,\gamma\,v\,R)\} \tag{6b}$$

Specification (6a) states that executing $\mathsf{new\_lock}()$ is safe, and returns a value v for which $\exists\gamma.\ \mathsf{is\_lock}\,\gamma\,v\,R$ holds—if we have given up resource R before executing $\mathsf{new\_lock}$. Parameter γ ensures we can distinguish between different locks. The $\mathsf{is\_lock}$ predicate is part of the precondition of the other lock methods.

[7] We omit Iris's update modality $\Rrightarrow$ from (6b) since it poses orthogonal problems with automation that are addressed by Mulder et al. [30] in Diaframe.

Specification (6b) differs from (6a) in that one does not have to give up resource R directly. Rather, it returns a more complicated proposition, which allows clients to choose and hand in R at a later point in the program execution.

Although (6a) is the standard lock specification [12, 16], (6b) is strictly stronger. One can (manually) verify the readers-writer lock with (6a), but (6b) is more useful for proof automation. When an automated verifier symbolically executes $\mathsf{new\_lock}$ with specification (6a), it does not have any syntactic indication for an appropriate choice for resource R. A wrong choice can easily lead to a failing verification [9]. With specification (6b), the automation can wait for a proof obligation with shape $\mathsf{is\_lock}\,\gamma\,v\,S$ to choose R equal to S.

This is precisely what happens when verifying the allocation of Courtois et al. [8]'s readers-writer lock. The allocation function first allocates two regular locks, for which we will use specification (6b). To prove that the readers-writer lock is successfully allocated, we are faced with goal:

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

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

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

6 Evaluation of iFrame

We evaluate the scalability of QU by testing our improved `iFrame` tactic. We test the improved tactic on four Iris projects to verify it does not cause performance regressions or introduce failures where it succeeded before (§6.1). We report on the results of a more artificial benchmark that compares the performance of the new $\exists$ rule to the rules for other connectives (§ 6.2). This benchmark shows that the Ltac implementation from §4.3 has acceptable complexity. We cannot conduct an evaluation of Diaframe, since there exists no baseline version of Diaframe without QU.

6.1 Subformula Linking with iFrame in Practice

Table 1 contains the results of our evaluation of the improved `iFrame` tactic. We investigate four Iris-based repositories of significant size: Iris itself, ReLoC [14, 15], RustBelt [17], and Iris's 'examples' repository. We compare the total compilation time of each repository with and without the improved `iFrame`. We also report the total number of changed lines that were required to patch these repositories. Existing proofs might break because `iFrame` is more powerful in the sense that it can frame more hypotheses and even solve a goal entirely that was not solved before.

Table 1. Evaluation data of `iFrame`. For each project, we list the total number of lines, the number of lines that use `iFrame`, and the number of lines that needed to be changed for the new `iFrame`. We also list the total compilation time of the project, and the change in compilation time with the new `iFrame`.

repository	total lines	`iFrame` lines	lines changed	total time	time changed %
Iris [18]	60156	543	- 2	8:29	-1.0%
iris-examples [20]	22912	800	-14	10:03	-1.2%
ReLoC [14, 15]	14092	505	- 5	4:54	-2.0%
RustBelt [17]	19889	840	-33	14:09	+0.3%
total	117049	2688	-54	37:37	-0.7%

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

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

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

6.2 Comparing Performance of Linkage Rules

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

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

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

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

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

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

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

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

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

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

7 Related Work

Subformula linking. The notion of subformula linking has been introduced introduced as part the Profound system by Chaudhuri [4]. Profound is a predecessor of ProfInt [5, 6] that involves first-order classical linear logic instead of first-order intuitionistic logic.

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

Prior works on subformula linking differ mostly from this work in their application. The ProfInt [5, 6] and Actema [13] gesture-based interactive theorem provers use subformula linking to simplify proof states by letting the user graphically indicate what subformulas to link. (A further predecessor of gesture-based theorem proving is 'proof by pointing' by Bertot et al. [2].) Aside from the difference in applications, there are some notable differences in the formal systems:

- Besides hypothesis-goal links, ProfInt and Actema also consider hypothesis-hypothesis links. ProfInt even considers links within a single formula. We do not consider such links, because hypothesis-goal links are the only relevant links for our application of *backward-chaining proof search*—which always takes the goal as starting point. Hypothesis-hypothesis links are essential for the completeness of ProfInt.
- The original presentation of Profint [5] uses a weaker subformula linking rule than R$\vee$. This weaker rule makes the rule order immaterial for the propositional fragment, but sometimes produces links that are harder to prove than links with R$\vee$. The rule order still matters when two hypotheses are linked.
- Actema supports 'rewriting' through subformula linking with an additional linkage rule $x \doteq y \wedge [P\,x] \Vdash P\,y$. Such a rule can be added as an instance to a QU-based system. It would be interesting to explore applications of such linkages, especially in combination Coq's generalized rewriting [35], which is used extensively in Iris to rewrite with relations other than equality.

Framing in separation logic. Various approaches have been proposed to (automatically) solve the 'framing' or 'frame inference' problem [1, 22] in separation logic. We focus on approaches that are implemented in proof assistants.

The VST framework in Coq [3] comes with a `cancel` tactic for frame inference, which contrary to Iris's `iFrame` does not proceed below existential quantifiers in the goal. VST also provides the more powerful `entailer` tactic, which is aimed at fully solving an entailment rather than making partial progress in an interactive proof.

The automation of the Bedrock [7, §3, step 6] and RefinedC [34, §4, step 5] frameworks in Coq instantiates existentially quantified goals with evars after having eliminated existentials in hypotheses. This is an effective approach for the verification of sequential problems. However, in the context of the verification of concurrent programs in Iris, existentials require a more careful treatment, as also argued in Diaframe [30]. We achieve this through subformula linking.

8 Future Work

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

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

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

Acknowledgments

We thank the reviewers for their useful feedback.

References

[1] 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

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

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

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

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

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

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

[8] 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

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

[10] Nicolaas Govert de Bruijn. 1991. Telescopic Mappings in Typed Lambda Calculus. *Information and Computation* 91, 2 (1991), 189–204. https://doi.org/10.1016/0890-5401(91)90066-B

[11] 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

[12] 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

[13] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A Drag-and-Drop Proof Tactic. In *CPP*. 197–209. https://doi.org/10.1145/3497775.3503692

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

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

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

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

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

[19] 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

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

[21] 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

[22] Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM (LNCS)*. 268–283. https://doi.org/10.1007/11813040_19

[23] Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Mück, Benjamin Peters, Gert Smolka, and Wehr, Dominik. 2022. A Coq Library for Mechanised First-Order Logic. In *Coq Workshop*. https://github.com/uds-psl/coq-library-fol

[24] 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

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

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

[27] Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023. Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. *PACMPL* 7, PLDI (2023), 161:1340–161:1364. https://doi.org/10.1145/3591275

[28] Ike Mulder and Robbert Krebbers. 2023. Artifact of 'Unification for Subformula Linking under Quantifiers'. Zenodo. https://doi.org/10.5281/zenodo.10364816

[29] 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

[30] 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

[31] 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

[32] 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

[33] David J. Pym. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series, Vol. 26. Kluwer.

[34] 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

[35] Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning* 2, 1 (2009), 41–62. https://doi.org/10.6092/issn.1972-5787/1574

[36] 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

[37] Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *ITP (LNCS)*. 499–514. https://doi.org/10.1007/978-3-319-08970-6_32

[38] The Coq-std++ Team. 2023. An Extended "Standard Library" for Coq. https://gitlab.mpi-sws.org/iris/stdpp

[39] Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading *(ICFP)*. 179–191. https://doi.org/10.1145/2784731.2784751

Received 2023-09-19; accepted 2023-11-25