Interactive Proofs in Higher-Order Concurrent Separation Logic

Robbert Krebbers *
Delft University of Technology,
The Netherlands
mail@robbertkrebbers.nl

Amin Timany
imec-Distrinet, KU Leuven, Belgium
amin.timany@cs.kuleuven.be

Lars Birkedal
Aarhus University, Denmark
birkedal@cs.au.dk

Abstract

When using a proof assistant to reason in an embedded logic – like separation logic – one cannot benefit from the proof contexts and basic tactics of the proof assistant. This results in proofs that are at a too low level of abstraction because they are cluttered with bookkeeping code related to manipulating the object logic.

In this paper, we introduce a so-called *proof mode* that extends the Coq proof assistant with (spatial and non-spatial) named proof contexts for the object logic. We show that thanks to these contexts we can implement high-level tactics for introduction and elimination of the connectives of the object logic, and thereby make reasoning in the embedded logic as seamless as reasoning in the meta logic of the proof assistant. We apply our method to Iris: a state of the art higher-order impredicative concurrent separation logic.

We show that our method is very general, and is not just limited to program verification. We demonstrate its generality by formalizing correctness proofs of fine-grained concurrent algorithms, derived constructs of the Iris logic, and a unary and binary logical relation for a language with concurrency, higher-order store, polymorphism, and recursive types. This is the first formalization of a binary logical relation for such an expressive language. We also show how to use the logical relation to prove contextual refinement of fine-grained concurrent algorithms.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Separation Logic, Interactive Theorem Proving, Coq, Fine-grained Concurrency, Logical Relations

1. Introduction

In the last decade, there has been tremendous progress on program logics for increasingly sophisticated programming languages [42, 17, 16, 13, 18, 41, 40, 11, 31, 24, 23, 26]. Part of the success of these logics stems from the fact that they have built-in support for reasoning about challenging programming language features. For instance, they include separating conjunction of separation logic for reasoning about mutable data structures, invariants for reasoning about sharing, guarded recursion for reasoning about various forms of recursion, and higher-order quantification for giving generic modular specifications to libraries.

Due to these built-in features, modern program logics are *very different* from the logics of general purpose proof assistants. Therefore, to use a proof assistant to formalize reasoning in a program logic, one needs to represent the program logic in that proof assistant, and then, to benefit from the built-in features of the program logic, use the proof assistant to reason *in* the embedded logic.

Reasoning in an embedded logic using a proof assistant traditionally results in a lot of overhead. Most of this overhead stems from the fact that when embedding a logic, one can no longer make use of the proof assistant's infrastructure for managing hypotheses. In separation logic this overhead is evident from the fact that propositions represent resources (they are *spatial*) and can thus be used at most once, which is very different from hypotheses in conventional logic that can be duplicated at will.

To remedy this situation, we present a so-called *proof mode* that extends the Coq proof assistant with (spatial and non-spatial) named contexts for managing the hypotheses of the object logic. We show that using our proof mode we can make reasoning in the embedded logic as seamless as reasoning in the meta logic of Coq. Although we believe that our proof mode is very generic, and can be applied to a variety of different embedded logics, we apply it to a specific logic in this paper, Iris: a state of the art impredicative higher-order separation logic for fine-grained concurrency [24, 23, 26]. We call the implementation on top of Iris IPM: *Iris Proof Mode*.

Iris is an interesting showcase for our proof mode, because unlike conventional program logics, it cannot only be used to reason about partial program correctness, but it also supports other kinds of reasoning. For starters, Iris differs from other (concurrent) program logics by not baking in particular reasoning principles, but by providing a minimal set of primitive constructs using which more advanced reasoning constructs can be defined *in* the logic. Furthermore, Iris can be used to define unary and binary relational interpretations of type systems and for proving theorems about those interpretations, *e.g.*, that if two terms are related in the relational interpretation of a type, then they are contextually equivalent. The type systems can range from ML-like type systems, such as $\mathsf{F}_{\mu,ref,conc}$ (System F with recursive types, references, and concurrency), to more expressive type-and-effect systems [27], or sophisticated ownership-based type systems such as the Rust type system [14]. We show that IPM supports all of these different kinds of reasoning.

One may wonder why we develop a reasoning tool for a logic like Iris in a general purpose proof assistant, instead of building a standalone tool. The main reason for using a proof assistant is that

* This research was carried out while this author was at Aarhus University.

POPL '17, January 18 - 20, 2017, Paris, France

ACM 978-1-4503-4660-3/17/01…$15.00.
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3009837.3009855

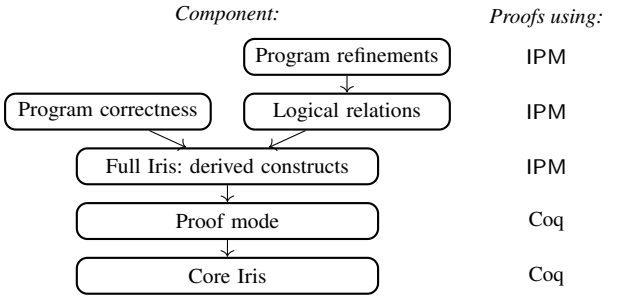

Figure 1. A formally verified stack of abstractions.

it is *foundational*. That means, correctness can be reduced to the adequacy result of the program logic, putting Iris and IPM outside of the trusted computing base. Moreover, by developing a reasoning tool in a proof assistant, we can piggy back on many of its features, instead of having to implement these features ourselves.

Contributions. We present a method for extending Coq with proof contexts and tactics for reasoning in embedded logics, and implement our method on top of the Iris logic. We show that our method and implementation – called IPM: Iris Proof Mode – are modular and widely applicable by verifying a stack of abstractions as shown in Figure 1. During the course of this paper we present the following contributions:

1. We use IPM to implement general purpose tactics for interactive proofs in higher-order separation logic. These tactics are partly implemented using reflection to ensure efficiency.
2. We show how Coq's type class machinery can be used to make these tactics modular. In particular, we show how additional logical connectives can be supported without the need to modify the implementation of the tactics.
3. We show how IPM can be used to prove the correctness of fine-grained concurrent algorithms.
4. We show that IPM can be used to prove soundness of a binary logical relation for a rich language with concurrency, higher-order store, polymorphism and recursive types. This is the first formalization of a binary logical relation for such a language in a proof assistant.
5. We use IPM to prove refinements of coarse- and fine-grained concurrent algorithms using the aforementioned logical relation.

Outline. We discuss the challenges involved in reasoning in an embedded logic using a proof assistant and outline the methodology of this paper in §2. Then, in §3, we give a tutorial-style introduction to IPM, and discuss the implementation in §4. In §5 we discuss how IPM is used for reasoning about concurrency, and in §6, we show how IPM can be used to prove the fundamental theorem and soundness of unary and binary logical relations for $\mathsf{F}_{\mu,ref,conc}$. In §6.4, we show how to use the logical relation to prove contextual refinement of fine-grained concurrent programs. Finally, we evaluate IPM in §7, discuss related work in §8, and conclude in §9.

Coq sources. The Coq sources can be found at:

`http://iris-project.org`

2. Embedding a Logic into a Proof Assistant

The most frequently used way of embedding an object logic into the meta logic of a proof assistant is through a *shallow embedding* [43]. That way, one represents the propositions of the object logic as semantic objects in the meta logic.

This surprisingly simple approach scales well to formalize *the meta-theory of (quite sophisticated) object logics*, for example [36, 5, 6, 22, 37, 24, 23, 25]. Unfortunately, as we will show in the remaining part of this section, this approach does not provide well-suited reasoning principles for doing proofs *in the object logic*. We use traditional separation logic as our running example.

For traditional intuitionistic separation logic, one would define its propositions *iProp* as follows:

$$\sigma \in \mathit{State} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \mathit{Val}$$
$$P, Q \in \mathit{iProp} \triangleq \mathit{State} \xrightarrow{\text{mon}} \mathsf{Prop}$$

The connectives of the object logic are defined via their semantic interpretation:

$$\begin{aligned}
\blacksquare \phi &\triangleq \lambda\sigma.\, \phi \\
\ell \mapsto v &\triangleq \lambda\sigma.\, \sigma(\ell) = v \\
P \wedge Q &\triangleq \lambda\sigma.\, P\sigma \wedge Q\sigma \\
P * Q &\triangleq \lambda\sigma.\, \exists\sigma_1\sigma_2.\, \sigma = \sigma_1 \uplus \sigma_2 \wedge P\sigma_1 \wedge Q\sigma_2 \\
(\exists x : A.\, P) &\triangleq \lambda\sigma.\, \exists x : A.\, P\sigma
\end{aligned}$$

Note that for some connectives (here $\wedge$ and $\exists$) the definition is given simply by lifting those of the meta logic into the object logic, whereas for others (here $*$), the definition is slightly more involved.

A shallow embedding has a couple of advantages over a *deep embedding*, which involves the extra step of defining an explicit syntax for all the connectives of the object logic:

- One can piggy back on the binders of the meta logic. For example, the predicate P in the existential quantifier $\exists x : A.\, P$ is modeled as a function $P : A \to \mathit{iProp}$ in the meta logic.
- One can piggy back on higher-order quantification of the meta logic. Using a proof assistant based on higher-order logic, one thus gets higher-order quantification in the object logic for free.
- One can easily embed propositions of the meta logic into the object logic, as done above using the $\blacksquare$ operator.

The entailment relation $\vdash$ of our simple separation logic can be defined as follows:

$$P \vdash Q \triangleq \forall\sigma.\, P\sigma \Rightarrow Q\sigma$$

The naive approach to proving an entailment is to work directly with the semantic interpretation of the logic by expanding the definitions of $\vdash$ and all logical connectives. However, for separating conjunction, this results in having to reason explicitly about disjointness $\uplus$ of states, which leads to an excessive number of proof obligations. Indeed, this approach is in direct opposition to the whole purpose of separation logic, which is to *hide* reasoning about disjointness. The situation becomes worse for logics with a step-indexed model (such as iCAP [40], CaReSL [41], VST [5] and Iris [24, 23, 26]), because then one needs to reason explicitly about steps too.

A more viable approach is to prove lemmas that correspond to the inference rules of the object logic. Examples of inference rules for separation logic are (where $P \dashv\vdash Q$ iff $P \vdash Q \wedge Q \vdash P$):

$$P * Q \dashv\vdash Q * P \qquad \text{(SEP-COMM)}$$
$$P * (Q * R) \dashv\vdash (P * Q) * R \qquad \text{(SEP-ASSOC)}$$
$$(P_1 \vdash Q_1) \text{ and } (P_2 \vdash Q_2) \;\Rightarrow\; P_1 * P_2 \vdash Q_1 * Q_2 \qquad \text{(SEP-MONO)}$$
$$P * (\exists x.\, Q) \dashv\vdash \exists x.\, (P * Q) \qquad \text{(SEP-EXIST-DISTR)}$$
$$(\forall x.\, (P \vdash Q)) \;\Rightarrow\; (\exists x.\, P) \vdash Q \qquad \text{(EXIST-ELIM)}$$

However, using just those rules, and without supporting infrastructure, it is very tedious to reason in the object logic.

```
1  Lemma and_exist A (P R: Prop) (Ψ: A → Prop) :
2    P ∧ (∃ a, Ψ a) ∧ R → ∃ a, Ψ a ∧ P.
3  Proof.
4    intros [HP [HΨ HR]].
5    destruct HΨ as [x HΨ].
6    exists x.
7    split. assumption. assumption.
8  Qed.
```

```
1  Lemma sep_exist A (P R: iProp) (Ψ: A → iProp) :
2    P ∗ (∃ a, Ψ a) ∗ R ⊢ ∃ a, Ψ a ∗ P.
3  Proof.
4    iIntros "[HP [HΨ HR]]".
5    iDestruct "HΨ" as (x) "HΨ".
6    iExists x.
7    iSplitL "HΨ". iAssumption. iAssumption.
8  Qed.
```

Figure 2. An example proof script in the Coq meta logic (left) and separation logic using IPM (right).

2.1 Reasoning in the Meta Logic of a Proof Assistant

Coq follows the model of tactic based goal-directed proofs, which is also used in many other proof assistants such as HOL [19] and LCF [20]. In this model, one states the proposition that one wishes to prove as the initial *goal*, and in turn, uses *tactics* to decompose the goal step by step into simpler subgoals. The initial goal is proven when all generated subgoals are proven.

Coq goals are sequents (the Coq-level entailment $\vdash_{\text{🐓}}$ should not be confused with the entailment $\vdash$ of the object logic):

$$H_1 : \phi_1, \ldots, H_n : \phi_n \vdash_{\text{🐓}} \psi$$

The left hand side of this sequent is called the *context* and the right hand side is called the *conclusion*.

Examples of basic Coq tactics are:

- The `intros H` tactic, which turns a goal with conclusion $\phi \to \psi$ intro a goal with conclusion ψ and a new hypothesis $\texttt{H} : \phi$.
- The `split` tactic, which turns a goal with conclusion $\psi_1 \wedge \psi_2$ into two subgoals whose conclusions are ψ_1 and ψ_2, and whose contexts are identical to the original goal.
- The `assumption` tactic, which closes a goal when its conclusion appears in the context.

An important feature of goal-directed proof assistants is context management. Using tactics one can manipulate parts of the context locally while the system keeps track of the rest. For example, when having $\texttt{H} : \psi_1 \wedge \psi_2$, the tactic `destruct H as` $[\texttt{H}_1\ \texttt{H}_2]$ generates a new goal in which the hypothesis `H` is removed and new hypotheses $\texttt{H}_1 : \psi_1$ and $\texttt{H}_2 : \psi_2$ are added in its place.

Introduction patterns [10, 8.3.2] are a powerful feature of Coq to perform basic context management in a concise way. For example, the tactic `intros [HP [HΦ HR]]` will introduce hypotheses and eliminate nested conjunctions on the fly. The left side of Figure 2 shows a proof script with the aforementioned tactics in action.

2.2 Reasoning in an Embedded Logic

When using a proof assistant to reason in an object logic, one's job involves manipulating goals of the shape $\vdash_{\text{🐓}} (Q \vdash R)$. As such, the premise Q of the entailment in the object logic appear in the conclusion of the goal on the meta logic, which means that one can no longer use the proof assistant's infrastructure for context management and tactics for introduction and elimination. Consider:

$$\ldots \vdash_{\text{🐓}} (P_1 * (\exists x.\, P_2) * P_3 \vdash Q)$$

In order to eliminate the existential, one has to use a combination of the inference rules SEP-COMM, SEP-EXIST-DISTR and EXIST-ELIM to first push the existential to the outside before one can actually eliminate it, whereas this operation would only take a single proof step when reasoning in the meta logic. This simple example already shows that the look and feel of reasoning in the object logic is very different from the look and feel of reasoning in the meta logic.

The main problem is that the left-hand side of the object logic entailment is an unstructured proposition, and as such, one cannot easily refer to individual parts of it. Our solution to this problem is surprisingly simple: represent entailment in the object logic also as a sequent with a named context:

$$H_1 : P_1, \ldots, H_n : P_n \vdash Q \quad \triangleq \quad P_1 * \cdots * P_n \vdash Q$$

Turning this simple idea into a usable method for efficiently proving properties in an expressive object logic using Coq is non-trivial. In Sections 4 and 5 we present our implementation on top of the Iris logic, called IPM: *Iris Proof Mode*, and show that:

- We can visualize the sequents of the object logic in Coq with the same look and feel as ordinary Coq goals.
- We can implement variants of the basic Coq tactics for introduction and elimination of all connectives of separation logic.
- In addition, we can implement custom tactics for Iris. Iris has a quite sophisticated model involving solutions to recursive domain equations in a category of metric spaces [23], but using IPM we can hide these internals.
- Coq's type class machinery can be used to make IPM modular. We show that additional logical connectives can be supported without the need to modify the implementation of the tactics.
- Our approach is efficient. Using reflection we make sure that single invocations of our basic tactics on the object logic correspond to a constant number of proof steps in the meta logic.

None of these efforts required us to modify the sources of Coq and IPM thus works with an off-the-shelf version of Coq.

3. IPM Tutorial

We give an introduction to IPM by proving a basic law of separation logic and functional correctness of in-place list reversal.

3.1 A Basic Law of Separation Logic

We consider the following law of affine separation logic:[1]

```
Lemma sep_exist A (P R: iProp) (Ψ: A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ R ⊢ ∃ a, Ψ a ∗ P.
```

Coq is able to recognize that this is a lemma in the object logic (*i.e.*, separation logic) since we are using the entailment relation $\vdash$. The connectives on the left and right hand side of $\vdash$ are thus parsed as those of the object logic instead of those of the Coq logic.

Figure 2 displays the proof script of this lemma, and compares it to an analogous lemma in the Coq meta logic. The goal outputted by IPM at the end of line 6 is as follows:

[1] Contrary to linear (or classical) separation logic, affine (or intuitionistic) separation logic enjoys weakening of separating conjunction, *i.e.* $P * Q \vdash P$. IPM uses an affine separation logic because Iris is affine. Nonetheless we believe that our method could be implemented on top of a linear separation logic as well, which would require a different implementation of some tactics because these should not be allowed to weaken the spatial context.

```
x : A
__________________________________(1/1)
"HP" : P
"HΨ" : Ψ x
"HR" : R
----------------------------------*
Ψ x ∗ P
```

IPM displays two proof contexts: the first is the Coq context, which contains variables and hypotheses at the meta level, whereas the second is the context of the object logic. So, the Coq goal is actually:

$$x : A \vdash_{\text{🐓}} (\texttt{HP} : P,\ \texttt{HΨ} : \Psi\, x,\ \texttt{HR} : R \vdash \Psi\, x * P)$$

In order to make reasoning in the object logic as seamless as possible, we render the context of the object logic in the same way as the context of the meta logic. Such aesthetics are important: a good overview of one's hypotheses is essential in large proofs.

Continuing the proof, at the end of line 6, we have to prove the separating conjunction $\Psi\, x * P$. Unlike conjunction, the introduction rule for separating conjunction is not so easy. Since separation logic is spatial, one has to split the context of the object logic into two parts: a part for the left conjunct, and a part for the right.

Using the basic rules of separation logic this is a tedious task: one has to rearrange the hypotheses up to associativity and commutativity in such a way that monotonicity of separating conjunction (see SEP-MONO) can be used to perform the introduction.

IPM makes introduction of separating conjunction $P * Q$ a single step. The tactic `iSplitL` $\vec{H}$ transforms the goal into two goals: one in which the hypotheses $\vec{H}$ are available to prove P, and another in which the remaining hypotheses are available to prove Q. Running `iSplitL "HΨ"` in line 6 of the example gives:

```
x : A
____________ (1/1)
"HΨ" : Ψ x
------------*
Ψ x
```

```
x : A
____________ (1/1)
"HP" : P
"HR" : R
------------*
P
```

Symmetrically, there is, of course, also a tactic `iSplitR` $\vec{H}$. So, in the example we could have written `iSplitR "HP HR"` too.

As shown in the example, a structured representation of the context of the object logic facilitates smooth interactive reasoning because it allows one to refer to hypotheses by name. Throughout this section, we show various tactics for separation logic for which it is essential to refer to hypotheses of the object logic by name.

A powerful proof method in separation logic is *framing*, which is the process of "canceling" a spatial hypothesis in the conclusion. Our tactic `iFrame` $\vec{H}$ automatically cancels the hypotheses $\vec{H}$ in the conclusion. Since separation logic is spatial, the hypotheses $\vec{H}$ will disappear in the resulting goal. Consider:

```
"HP" : P
"HΨ" : ∃ a : A, Ψ a
"HR" : R
----------------------------------------*
∃ a : A, Ψ a ∗ P
```

By writing `iFrame "HP"` the hypothesis `HP` disappears and the goal becomes `∃ a : A, Ψ a`. The proof of the lemma `sep_exist` in Figure 2 can thus be shortened to:

```
iIntros "[HP [HΨ HR]]". iFrame "HP". iAssumption.
```

As shown in this example, the `iFrame` tactic is able to cancel hypotheses under quantifiers. This is not hard-coded into the implementation of the tactic, but instead, the tactic can be extended by declaring type class instances, as we will show in Section 4.

3.2 In Place List Reversal

We will prove functional correctness of in-place reversal of linked lists to demonstrate more advanced features of IPM. The algorithm is written in an ML-like language that is deeply embedded in Coq:

```
Definition rev : val :=
  rec: "rev" "hd" "acc" :=
    match: "hd" with
      NONE => "acc"
    | SOME "l" =>
       let: "tmp1" := Fst !"l" in
       let: "tmp2" := Snd !"l" in
       "l" <- ("tmp1", "acc");;
       "rev" "tmp2" "hd"
    end.
```

As the above code shows, we make heavy use of Coq's expressive notation mechanism to obtain human readable notations. Variables are represented using strings.

In order to state functional correctness of this program, we relate mathematical lists (defined as an inductive data type in Coq) to linked-lists in our ML-like language:

```
Fixpoint is_list (hd: val) (xs: list val) : iProp :=
  match xs with
  | [] => hd = NONEV
  | x :: xs => ∃ l hd',
     hd = SOMEV #l ∗ l ↦ (x,hd') ∗ is_list hd' xs
  end%I.
```

The predicate `is_list` $hd\ \vec{x}$ states that the mathematical list $\vec{x}$ is represented as a linked-list with root pointer hd in memory. The use of separating conjunction in the `is_list` predicate is essential: it ensures that all pointers are disjoint.

Using the `is_list` predicate we can relate in-place reversal to the mathematical operation of reversing a list (called reverse). The Hoare logic specification thus looks as follows:

$$\{\texttt{is\_list}\ hd\ \vec{x} * \texttt{is\_list}\ acc\ \vec{y}\}\ \texttt{rev}\ hd\ acc$$
$$\{w.\, \texttt{is\_list}\ w\ (\mathsf{reverse}(\vec{x}) +\!\!+ \vec{y})\}$$

As is common for Hoare triples for effectful functional programs, the postcondition has a binder to refer to the return value. In Coq, a postcondition is thus modeled using a function type *Val* → *iProp*, and we thus often use functional notation for postconditions.

A common way of doing proofs in separation logic is by symbolic execution [7]. The key idea of symbolic execution is to treat the pre- and postconditions as symbolic representations of the heap, so that one can do proofs by symbolically executing the program with respect to the precondition.

For example, in order to prove $\{P\}\, \ell_1 \leftarrow !\, \ell_2 + 1; e'\, \{\Phi\}$, one looks for a maps-to predicate $\ell_2 \mapsto v$ in P, and continues proving $\{P\}\, \ell_1 \leftarrow v + 1; e'\, \{\Phi\}$. Subsequently, one looks for $\ell_1 \mapsto -$ in P and continues with $\{P'\}\, e'\, \{\Phi\}$ where P' is obtained by replacing $\ell_1 \mapsto -$ with $\ell_1 \mapsto v + 1$ in P.

In order to use IPM to perform (interactive) proofs by symbolic execution, we wish to leverage its facilities for context management to organize the precondition of the Hoare triple. To this end, we will do proofs in *weakest precondition* style instead of Hoare style, which intuitively enables us to "decouple" the precondition from the Hoare triple. This "decoupling" becomes evident from the way Hoare triples are defined in Iris:

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(P \to \mathsf{wp}\, e\, \{\Phi\})$$

Ignoring the $\Box$ modality for a moment, we can prove $\{P\}\, e\, \{\Phi\}$ by introducing P into the spatial context. We can then use the tactics of IPM to manipulate P, and the rules in Figure 3 to prove the weakest precondition in a goal directed style.

$$\textsc{Wp-frame}\ \frac{Q * \mathsf{wp}\, e\, \{\Phi\}}{\mathsf{wp}\, e\, \{x.\, Q * \Phi\, x\}} \qquad \textsc{Wp-val}\ \frac{\Phi\, v}{\mathsf{wp}\, v\, \{\Phi\}} \qquad \textsc{Wp-bind}\ \frac{\mathsf{wp}\, e\, \{v.\ \mathsf{wp}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}\, K[e]\, \{\Phi\}}$$

$$\textsc{Wp-lam}\ \frac{\triangleright \mathsf{wp}\, e[v/x]\, \{\Phi\}}{\mathsf{wp}\, (\lambda x.\, e) v\, \{\Phi\}} \qquad \textsc{Wp-rec}\ \frac{\triangleright \mathsf{wp}\, e[\texttt{rec}\, f(x) = e/f][v/x]\, \{\Phi\}}{\mathsf{wp}\, (\texttt{rec}\, f(x) = e) v\, \{\Phi\}}$$

$$\textsc{Wp-alloc}\ \frac{\triangleright(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi\, \ell)}{\mathsf{wp}\, \texttt{ref}(v)\, \{\Phi\}} \qquad \textsc{Wp-store}\ \frac{\triangleright\, \ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} \Phi\, ())}{\mathsf{wp}\, \ell \leftarrow w\, \{\Phi\}}$$

Figure 3. Selected rules of weakest preconditions.

The purpose of the *always modality* $\Box$ in the above definition is to ensure that the Hoare triple *itself* does not assert ownership of any resources. In terms of our toy separation logic from Section 2, one would model this modality as[2] $\Box P \triangleq \lambda\sigma.\, P\, \emptyset$. Elimination of $\Box$ is easy, as we have $\Box P \vdash P$. However, introduction of $\Box$ is only possible if any hypothesis $H : P$ in the context is *persistent* [23]. That means that no hypothesis asserts ownership of any resource, or, more formally, that all hypotheses satisfy $P \vdash \Box P$.

Persistent propositions play a very important role when reasoning in Iris, notably because they are duplicable, *i.e.*, $P \dashv\vdash P * P$, and can thus be freely shared among threads. This property does not hold for spatial connectives like $\ell \mapsto v$.

To make reasoning with persistent propositions easy, IPM also features a context of persistent hypotheses (apart from the Coq context and the context for spatial hypotheses):

```
Hpure_i : φ_i      Variables and pure Coq hypotheses
--------------------------------------
Hpersistent_i : P_i      Persistent hypotheses in object logic
--------------------------------------□
Hspatial_i : Q_i      Spatial hypotheses in object logic
--------------------------------------*
R      Goal in object logic
```

The tactics of IPM have extensive support for persistent propositions. For example:

- The `iIntros "!#"` tactic introduces the $\Box$ modality. Introduction of $\Box$ is only allowed if the spatial context is empty.
- The `iDestruct` H `as #`H_p tactic moves a persistent hypothesis from the spatial context into the persistent context. Logic programming using type classes is used to determine whether the hypothesis is persistent. Type classes are discussed in Section 4.
- Separating conjunctions $P * Q$ with either P or Q persistent can be introduced using the `iSplit` tactic without having to split the context among the conjuncts.

Coming back to the functional correctness of in-place list reversal, the proof in Figure 4 shows many features of IPM in action. Let us highlight some novel features:

Introduction patterns. The tactics `iIntros` and `iDestruct` support *introduction patterns* similar to the ones supported by Coq's `intros` and `destruct` tactics (*cf.* Section 2.1). The syntax of these tactics is as follows:

$$\texttt{iIntros}\ (x_1 \ldots x_n)\ \texttt{"ipat}_1 \ldots \texttt{ipat}_n\texttt{"}$$
$$\texttt{iDestruct}\ H\ \texttt{as}\ (x_1 \ldots x_n)\ \texttt{"ipat"}$$

[2] The semantic interpretation of $\Box$ in Iris [23] is more complicated, but the precise definition is orthogonal to this paper.

```
1  Lemma rev_acc_ht hd acc xs ys :
2    heap_ctx ⊢
3    {{ is_list hd xs ∗ is_list acc ys }} rev hd acc
4    {{ w, is_list w (reverse xs ++ ys) }}.
5  Proof.
6    iIntros "#Hh !# [Hxs Hys]".
7    iLöb as "IH" forall (hd acc xs ys). wp_rec; wp_let.
8    destruct xs as [|x xs]; iSimplifyEq.
9    - (* nil *) by wp_match.
10   - (* cons *) iDestruct "Hxs"
11       as (l hd') "[% [Hx Hxs]]"; iSimplifyEq.
12     wp_match. wp_load. wp_proj. wp_let.
13     wp_load. wp_proj. wp_let. wp_store.
14     rewrite reverse_cons -assoc.
15     iApply ("IH" $! hd' (InjRV #l) xs (x :: ys)
16       with "Hxs [Hx Hys]").
17     iExists l, acc; by iFrame.
18 Qed.
```

Figure 4. Functional correctness of in-place list reversal.

The prefix $x_1 \ldots x_n$ can be used to introduce universal quantifiers, respectively eliminate (nested) existential quantifiers.[3] For example, given the goal $\forall x\, y.\, P \mathrel{-\!\!*} Q$, we may write `iIntros (x y) "HP"`.

In addition to the standard introduction patterns (such as `?` for creating an anonymous hypothesis, `[`$\texttt{ipat}_1$ $\texttt{ipat}_2$`]` for eliminating a (separating) conjunction, `[`$\texttt{ipat}_1$`|`$\texttt{ipat}_2$`]` for eliminating a disjunction, and `[]` for false elimination), IPM supports:

- `#` `ipat`: move the hypothesis into the persistent context.
- `%`: move the hypothesis into the pure Coq context.
- `!#`: introduce a $\Box$ (this pattern can only appear at the top-level, and can be used only if the spatial context is empty).

In line 6 of Figure 4, `iIntros "#Hh !# [Hxs Hys]"` is used to introduce `Hh : heap_ctx` (an invariant used to encode the $\mapsto$ connective, see [24, 3.6]) into the persistent context, and `Hxs : is_list hd xs` and `Hys : is_list acc ys` into the spatial context.

The later modality and Löb induction. Iris uses the *later modality* $\triangleright$ [30, 3] and Löb induction to prove properties of recursive functions. We now show how both are supported by IPM.

The *later modality* states that a property holds only after a step of computation. The proof principle associated with the later modality is Löb induction, which allows one to prove any proposition P under the assumption that it holds later (*i.e.* after a step of computation):

$$(\triangleright P \to P) \vdash P$$

The tactic `iLöb` performs Löb induction and takes care of associated bookkeeping. Let us demonstrate this by an example, invoking `iLöb as "IH" forall (hd acc xs ys)` in line 7 of Figure 4 yields:

```
"Hh" : heap_ctx
"IH" : ▷ (∀ hd acc xs ys,
  is_list hd xs -∗ is_list acc ys -∗
  WP rev hd acc {{ w, is_list w (reverse xs ++ ys) }})
--------------------------------------□
"Hxs" : is_list hd xs
"Hys" : is_list acc ys
--------------------------------------*
WP rev hd acc {{ w, is_list w (reverse xs ++ ys) }}
```

The tactic generates an induction hypothesis `IH`, which is generalized over the Coq-level variables `hd`, `acc`, `xs`, `ys` and all spatial

[3] Due to limitations of Ltac, these variables always have to appear at the beginning and cannot be mixed with the IPM introduction pattern.

hypotheses, allowing us to prove the correctness of the recursive call. The induction hypothesis is guarded by the *later modality* $\triangleright$ and can thus only be used after one step of symbolic execution.

As we will see in the subsequent sections, the later modality plays an even more important role in Iris: it can be used to define guarded recursive predicates, and it appears in the rules for invariants.

Symbolic execution. A large part of the proof in Figure 4 involves symbolic execution of the program. IPM provides tactics that apply the rules for weakest preconditions (see Figure 3) and perform basic bookkeeping.[4] For example, the tactic `wp_store` tries to find a store construct in evaluation position (using the rule WP-BIND), executes it (using the rule WP-STORE), and performs the introductions of the $\triangleright$, $-\!\!*$ and $*$ connectives.

The apply tactic. The typical way of dealing with implications in Coq is *backwards chaining* using the `apply` tactic. Given a goal with conclusion ϕ and a hypothesis $H : \psi_1 \to \cdots \to \psi_n \to \phi$, one can write `apply` H to turn the goal into new goals $\psi_1, \ldots, \psi_n$ whose hypotheses are the same as those of the initial goal.

In separation logic this no longer works that easily since propositions are spatial. To apply $H : \forall x_1 \ldots x_n.\, Q_1 -\!\!* \cdots -\!\!* Q_m -\!\!* P$, one has to specify which hypotheses are used for which premise. The syntax of `iApply` is thus somewhat different:

$$\texttt{iApply}\,(H\ \texttt{\$!}\ t_1 \ldots t_n\ \texttt{with "spat}_1 \ldots \texttt{spat}_m\texttt{"})$$

The prefix $t_1 \ldots t_n$ tells how to instantiate universal quantifiers, and the suffix $\texttt{spat}_1 \ldots \texttt{spat}_m$ is a sequence of, what we dub, *specialization patterns*, to specify how the spatial hypotheses should be split among the subgoals for the premises:

- `[`$H_1 \ldots H_n$`]`: generate a goal with spatial hypotheses $H_1 \ldots H_n$. These hypotheses will disappear in subsequent goals.
- `[#]`: generate a goal with all hypotheses, provided the premise is persistent. All spatial hypotheses will remain in subsequent goals.

Specialization patterns can be used as part of other tactics too, for example, to eliminate `H` : $\square P_1 -\!\!* P_2 -\!\!* Q_1 * Q_2$, one can write `iDestruct ("H" with "[#] [H1 H2]") as "[H4 H5]"`.

4. Implementation

IPM is implemented entirely in Coq and does not involve any OCaml plugins or modifications of the Coq source code. To achieve that, the implementation involves an interplay between many advanced Coq features. In order to ensure efficiency, we have implemented the primitive tactics performing introduction and elimination of the logical connectives of our object logic using *computational reflection*. Next to that, we have used *logic programming* using type classes [38, 39] to make our tactics very modular, and finally, we have used *the Ltac tactic definition language* [12] to combine these parts into high-level tactics for the end-user.

In this section we explain some of the key points of our implementation. Readers who are only interested in how IPM can be used may skip this section.

4.1 Embedding of Environments and Core Tactics

In order to implement tactics for introduction and elimination of nearly all logical connectives of separation logic, it is only needed to manipulate the shape of the contexts, without the need to "look into" individual hypotheses. So, in order to implement these tactics using computational reflection, it is not necessary to deeply embed the whole logic, but just the contexts of the object logic.

The contexts of our separation logic are represented as pairs of association lists `env` with strings as keys:[5]

```
Record envs :=
  Envs { env_persistent : env iProp;
         env_spatial : env iProp }.
```

We can now define the semantic interpretation of contexts:

```
Coercion of_envs (Δ : envs) : iProp :=
  (■ envs_wf Δ ∗ □ [∗] env_persistent Δ
               ∗ [∗] env_spatial Δ)%I.
```

Goals of IPM are Coq goals of the shape `of_envs` $\Delta \vdash Q$, which can be written as $\Delta \vdash Q$ since we declared `of_envs` as a coercion. The condition `envs_wf` Δ ensures that Δ is well-formed, *i.e.*, all hypotheses have unique names. The connective `[∗]` folds a separating conjunction over a list.

Most IPM tactics are defined as Coq lemmas justifying context manipulations. For example, the lemma corresponding to the tactics `iSplitL` and `iSplitR` (*cf.* Section 3.1) for introduction of separating conjunction is as follows:

```
Lemma tac_sep_split Δ Δ1 Δ2 lr js Q1 Q2 :
  envs_split lr js Δ = Some (Δ1,Δ2) →
  (Δ1 ⊢ Q1) → (Δ2 ⊢ Q2) → Δ ⊢ Q1 ∗ Q2 .
```

The first premise comprises the function `envs_split`, which splits the context Δ into contexts Δ_1 and Δ_2 for the new goals. The context Δ_1 has the spatial hypotheses named `js`, while Δ_2 has the remaining spatial hypotheses (or *vice versa*, depending on the value of the Boolean `lr`). This function is written in Coq, which enables us to prove the first premise by computation. The last two premises are the new goals for the conjuncts `Q1` and `Q2`.

The end-user tactics `iSplitL` and `iSplitR` can now simply be defined as wrappers that use the lemma `tac_sep_split`. Each individual occurrence of an `iSplitL` or `iSplitR` tactic thus results in a Coq proof term whose size is constant: it consists solely of the a single lemma whose side conditions are proven by computation (*i.e.*, by the constructor `eq_refl` of the identity type).

In order to prove `tac_sep_split`, we have to prove soundness of the `envs_split` function:

```
Lemma envs_split_sound Δ lr js Δ1 Δ2 :
  envs_split lr js Δ= Some (Δ1,Δ2) →Δ ⊢ Δ1 ∗ Δ2.
```

The approach of implementing IPM tactics by reflection can be applied widely and easily. It involves: implementation of a sound Coq function that performs the needed context manipulation (IPM provides many such functions), a statement of the tactic in terms of a Coq lemma, and an Ltac wrapper.

4.2 Tactics Implemented Using Logic Programming

Some tactics do not just manipulate the shape of the context of the object logic, but also manipulate the propositions in the context or the conclusion. In this section we discuss the `iFrame` H tactic (*cf.* Section 3.1), which cancels the hypothesis H in the conclusion.

Intuitively, one would implement framing of `H` : R in $\Delta \vdash Q$ by traversal through Q. However, since we use a shallow embedding to represent the propositions of our object logic, propositions are semantic objects, so we cannot just write a Coq function to perform

[4] Although our tactics perform symbolic execution in small steps, one can easily define a tactic that performs symbolic execution repeatedly. However, in concurrent separation logic, small step symbolic execution is often needed so as to be able to open invariants around atomic expressions, see Section 5.

[5] The Coq implementation of IPM is very generic: the core data structures and tactics are not specific to Iris, but can be instantiated with any intuitionistic logic that is defined in terms of step-indexed predicates over a monoid-like structure. For simplicity, we omit these details.

this traversal. Instead, we use logic programming using type classes to perform such manipulations on the meta level [38, 39].

We follow the approach introduced in Section 4.1 to represent tactics as Coq lemmas. The lemma corresponding to `iFrame` is:

```
Class Frame R P Q := frame : R ∗ Q ⊢ P.
Lemma tac_frame Δ Δ' i p R P Q :
  envs_lookup_delete i Δ = Some (p, R, Δ') →
  Frame R P Q →
  ((if p then Δ else Δ') ⊢ Q) → Δ ⊢ P.
```

The first premise is easy: it states that there is a hypothesis `i : R` in Δ, where the Boolean `p` denotes whether `i` is in the persistent part or the spatial part, and Δ' is the context in which `i` is removed. The function `envs_lookup_delete` is implemented in Coq, so this premise can be solved by computation.

The second premise is where the actual work occurs. Instances of the type class `Frame R P Q` should be considered as clauses of a logic program with inputs `R` (the proposition we wish to frame) and `P` (the initial conclusion), and output `Q` (the conclusion of the new goal, in which `R` is canceled). For example:

```
Class MakeSep P Q PQ := make_sep : P ∗ Q ⊣⊢ PQ.
Instance frame_here R : Frame R R True.
Instance frame_sep_l R P1 P2 Q Q' :
  Frame R P1 Q → MakeSep Q P2 Q' → Frame R (P1 ∗ P2) Q'.
Instance frame_sep_r R P1 P2 Q Q' :
  Frame R P2 Q → MakeSep P1 Q Q' → Frame R (P1 ∗ P2) Q'.
```

The `MakeSep` class is used to remove superfluous `True` connectives that were the result of a successful canceling attempt:

```
Instance make_sep_true_l P: MakeSep True P P.
Instance make_sep_true_r P: MakeSep P True P.
Instance make_sep_default P Q: MakeSep P Q (P∗Q).
```

Type classes provide a modular way of implementing tactics: to extend the behavior of a given tactic, one simply declares additional instances (*i.e.*, clauses in the logic program). For example, the following instance enables framing under weakest preconditions:

```
Instance frame_wp E e R Φ Ψ :
  (∀ v, Frame R (Φ v) (Ψ v)) →
  Frame R (WP e @ E {{ Φ }}) (WP e @ E {{ Ψ }}).
```

The above instance demonstrates that our approach is also powerful enough to enable framing under binders.

Our approach of using type classes can be used to implement tactics for many connectives and many purposes. For example, we have used it to implement a tactic called `iNext`, which introduces a later modality $\triangleright$ by stripping a later from each hypothesis.

4.3 End-user Tactics

The end-user tactics of IPM are simply implemented as wrappers around the basic building blocks described in this section. These wrappers are written in Ltac and are responsible for:

Input handling. Tactics like `iIntros` and `iDestruct` can perform a sequence of operations as specified by the introduction patterns that they are invoked with.

In order to handle introduction patterns, we have implemented a parser for introduction patterns in Coq. This parser takes a Coq string and yields an abstract syntax tree, which is then processed using Ltac to invoke various primitive tactics responsible for the necessary introductions and eliminations.

Error handling. Ltac is used to catch errors of the Coq tactics that are used in the implementation of the IPM tactics, and then, to turn these errors into higher-level error messages.

4.4 Rendering of Goals

We use Coq's notation machinery to render goals `of_envs` $\Delta \vdash Q$ in the same way as Coq goals, but with an additional context Δ (*cf.* Section 3.1). There is one shortcoming of Coq's notation machinery we had to circumvent: all Coq notations should be parsable, whereas we want the aforementioned notations to only *appear* in intermediate proof states. To circumvent this shortcoming, we have made use of *zero width Unicode* characters that function as terminals in the parser, and thereby avoid conflicts with other notations.[6]

4.5 Modularity

As shown in Section 4.2, Coq's type classes provide a powerful mechanism to implement modular tactics. In this section we will show that we can take this approach even further, so that existing tactics can be extended to handle new logical connectives.

In this section we will demonstrate how we have implemented the `iDestruct` H `as` $[H_1\ H_2]$ tactic, so that it can eliminate all kinds of "conjunction like" connectives. This tactic is not only able to handle conjunctions $P \wedge Q$ and separating conjunctions $P * Q$, but also Iris specific connectives. Furthermore, it can distribute laters, for example, when eliminating $\triangleright(P \wedge Q)$, it will turn it into $\triangleright P$ and $\triangleright Q$, and thereby makes tedious reasoning about steps implicit, as one would do on paper.

Let us take a look at the Coq lemma corresponding to this tactic:

```
Class IntoAnd (p : bool) (P Q1 Q2 : uPred M) :=
  into_and : P ⊢ if p then Q1 ∧ Q2 else Q1 ∗ Q2.
Instance into_and_sep p P Q : IntoAnd p (P ∗ Q) P Q.
Instance into_and_and P Q : IntoAnd true (P ∧ Q) P Q.

Lemma tac_and_destruct Δ Δ' i p j1 j2 P P1 P2 Q :
  envs_lookup i Δ = Some (p, P) → IntoAnd p P P1 P2 →
  envs_simple_replace i p
    (Esnoc (Esnoc Enil j1 P1) j2 P2) Δ = Some Δ' →
  (Δ' ⊢ Q) → Δ ⊢ Q.
```

The type class `IntoAnd` avoids tying the tactic to (separating) conjunction. Note that since conjunction elimination is only sound for persistent hypotheses, the class is parametrized by a Boolean `p` that describes whether the hypothesis is persistent or spatial. The instance `into_and_and` requires this Boolean to be true.

By defining an additional type class instance it is easy to instruct this tactic to distribute laters:

```
Instance into_and_later p P Q1 Q2 :
  IntoAnd p P Q1 Q2 → IntoAnd p (▷ P) (▷ Q1) (▷ Q2).
```

Notice that type class search is hereditary, which enables the tactic to distribute nested laters. This approach scales to all kinds of "conjunction like" connectives, for example:

```
Instance into_and_mapsto l q v :
  IntoAnd false (l ↦{q} v) (l ↦{q/2} v) (l ↦{q/2} v).
```

This instance makes it possible to eliminate the maps-to connective $\ell \overset{q}{\mapsto} v$ with fractional permission q into two halves $\ell \overset{0.5q}{\longmapsto} v$.

5. Proofs of Concurrent Programs in IPM

In this section, we explain how IPM is used to reason about partial program correctness of concurrent algorithms using Iris.[7] During the course of this section, we will verify the correctness of a fine-grained implementation of a monotonic counter [33]. The code of the counter is as follows:

[6] Support for *printing only* notations will be available in future Coq versions, see `https://github.com/coq/coq/pull/194`.

[7] We use the presentation of Iris as described in [26].

```
Definition newcounter : val := λ: <>, ref #0.
Definition incr : val :=
  rec: "incr" "l" :=
    let: "n" := !"l" in
    if: CAS "l" "n" (#1 + "n") then #() else "incr" "l".
Definition read : val := λ: "l", !"l".
```

The construct `#` represents the embedding of literals (integers, locations, and unit values `()`) into our ML-like language.

The counter is simply represented by a reference to an integer, which is initialized to 0. The `read` operation yields the value of the counter, and `incr` increases the counter by one. The counter is called *monotonic* because using just these operations, its value can only be incremented. Notice that `incr` is made thread safe using a `CAS` (compare and swap) loop.

5.1 The Counter Predicate

We will define a predicate $C : \mathit{Loc} \to \mathbb{N} \to \mathit{iProp}$ in Iris to capture that the counter is monotonic. The meaning of $C(\ell, n)$ is that ℓ is at least n, which is made formal by the following Hoare triples:

$$\{\mathsf{True}\}\,\texttt{newcounter}\;()\,\{\ell.\, C(\ell, 0)\}$$
$$\{C(\ell, n)\}\,\texttt{read}\;\ell\,\{m.\, n \le m \land C(\ell, m)\}$$
$$\{C(\ell, n)\}\,\texttt{incr}\;\ell\,\{C(\ell, 1+n)\}$$

Since we wish to use the counter in a concurrent setting, it is essential that C is duplicable, *i.e.*, $C(\ell, n) \dashv\vdash C(\ell, n) * C(\ell, n)$, so that it can be freely shared among threads. As a consequence, this means that we cannot simply define $C(\ell, n)$ as $l \mapsto n$. The connective $l \mapsto n$ expresses full ownership of a physical location, and thus cannot be shared (*i.e.*, is not duplicable).

In order to define the predicate C, we will make use of the two key features of Iris: *invariants*, to enable sharing of $C(\ell, n)$, and *ghost state*, to enforce that updates to ℓ are monotone.

Invariants. The core idea is to encode $C(\ell, n)$ using an invariant that governs $l \mapsto n$. An invariant is a property I that holds for some piece of shared state at all times: each thread accessing the state may assume that the invariant holds before each step of its computation, but it must also ensure that it continues to hold after each step. In Iris, invariants are provided by propositions of the form $\boxed{I}^{\mathcal{N}}$, for which there is the following proof rule[8]:

$$\frac{\{I * P\}\, e\, \{v.\, I * Q\} \qquad \mathsf{atomic(e)}}{\boxed{I}^{\mathcal{N}} \vdash \{P\}\, e\, \{v.\, Q\}}$$

Here, the proposition $\boxed{I}^{\mathcal{N}}$ states the knowledge that there exists an invariant I governing some shared state. Given this knowledge, the rule tells us that e may gain (exclusive) control of I, so long as it ensures that I continues to hold when it is finished executing. The side condition that e should be physically atomic is crucial, if it were not, then another thread could access the shared state governed by I during e's execution. Note that $\boxed{I}^{\mathcal{N}}$ is persistent since it merely expresses that there exists some shared state satisfying I, and does not represent exclusive ownership of I.

Ghost state. We could now define $C(\ell, n)$ as $\boxed{\exists c.\, \ell \mapsto c}^{\mathcal{N}}$, but this would not be sufficient. We also have to relate the lower bound n to the actual value c of the counter ℓ, which is existentially quantified in the invariant. In order to relate n to c, we will use a *ghost variable*

[8] To identify invariants, each invariant lives in a namespace $\mathcal{N}$. Updates (explained in the following) and weakest preconditions are annotated with a mask $\mathcal{E}$ to avoid *reentrancy*, *i.e.*, avoiding "opening" the same invariant twice in a nested fashion. Unlike this paper, in which we omit these masks, they are visible in Coq, but all bookkeeping related to them is fully automated.

UPD-INTRO $P \vdash \Rightarrow P$

UPD-TRANS $\Rightarrow\Rightarrow P \vdash \Rightarrow P$

UPD-FRAME $Q * \Rightarrow P \vdash \Rightarrow (Q * P)$

OWN-ALLOC $\dfrac{a \neq \bot}{\mathsf{True} \vdash \Rightarrow \exists \gamma.\, \boxed{a}^{\gamma}}$

OWN-OP $\boxed{a \cdot b}^{\gamma} \dashv\vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$

OWN-VALID $\boxed{a}^{\gamma} \vdash a \neq \bot$

UPD-OWN $\dfrac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \vdash \Rightarrow \boxed{b}^{\gamma}}$

UPD-WP $\dfrac{P \vdash \mathsf{wp}\, e\, \{\Phi\}}{\Rightarrow P \vdash \mathsf{wp}\, e\, \{\Phi\}}$

INV-ALLOC $\triangleright P \vdash \Rightarrow \boxed{P}^{\mathcal{N}}$

INV-OPEN $\dfrac{\triangleright P \vdash \mathsf{wp}\, e\, \{v.\, \triangleright P * \Phi\, v\} \qquad \mathsf{atomic(e)}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wp}\, e\, \{\Phi\}}$

(where $a \rightsquigarrow b \triangleq \forall a_{\mathrm{f}}.\, a \cdot a_{\mathrm{f}} \neq \bot \to b \cdot a_{\mathrm{f}} \neq \bot$)

Figure 5. Selected rules of Iris.

$\boxed{a}^{\gamma}$ that mirrors the value c. Ghost state in Iris differs from physical state (*i.e.*, the $\mapsto$ connective) in two ways. Firstly, it is "logical state", *i.e.*, state that exists solely in the proof but not during the execution of the actual program. Secondly, unlike physical state, we can control what kind of sharing of ghost state is possible.

In order to specify what kind of sharing is possible, Iris allows the end-user to define a partial commutative monoid (PCM)[9] for each ghost variable. Sharing of ghost ownership is expressed using the following proof rule:

$$\boxed{a \cdot b}^{\gamma} \dashv\vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

The PCM M for the monotone counter is as follows:

$$M \triangleq \bullet(c) \mid \circ(n) \mid \bot$$
$$\circ(n_1) \cdot \circ(n_2) \triangleq \circ(n_1 \max n_2)$$
$$\circ(n) \cdot \bullet(c) \triangleq \bullet(c) \cdot \circ(n) \triangleq \begin{cases} \bullet(c) & \text{if } n \le c \\ \bot & \text{otherwise} \end{cases}$$
$$\bullet(c_1) \cdot \bullet(c_2) \triangleq \bot$$

The *authoritative element* $\bullet c$ will be used to express that the counter has exactly value c, and will thus be placed in the invariant. The *fragmental element* $\circ n$ will be used to express that the counter has *at least* value n, and will be used in the definition of $C(\ell, n)$ to relate n to the actual value of the counter. Formally, this gives rise to the following definitions:

$$I(\gamma, \ell) \triangleq \exists c.\, (\ell \mapsto c) * \boxed{\bullet(c)}^{\gamma}$$
$$C(\ell, n) \triangleq \exists \gamma.\, \boxed{I(\gamma, \ell)}^{\mathcal{N}} * \boxed{\circ(n)}^{\gamma}$$

5.2 The Proof in IPM

Proof of `read`. With the definition of C set up, we will now use IPM to go through the proof of `read`. After expanding the Hoare triple into a weakest precondition and introducing the precondition $C(\ell, n)$ into the context, we have:

```
"Hinv" : inv N (I γ l)
--------------------------------------□
"Hγf" : own γ (Frag n)
--------------------------------------*
WP ! #l {{ v, ∃ m : nat, ■ (v = #m ∧ n ≤ m) ∧ C l m }}
```

[9] Ghost state in Iris has a slightly richer structure than that of a PCM, namely that of a *resource algebra* (RA) [23]. This richer structure makes it easy to compose compound RAs using simpler constructions like sums, but that is orthogonal to this paper.

Since the dereferencing operator ! is atomic, we are allowed to open the invariant N using the rule INV-OPEN in Figure 5. This rule is simply a lemma that can be applied using the `iApply` tactic.[10]

After using the lemma corresponding to INV-OPEN, and after symbolic execution of the dereferencing operator (using the `wp_load` tactic), we end up with the following goal:

```
"Hinv" : inv N (I γ l)
--------------------------------------□
"Hγf" : own γ (Frag n)
"Hl" : l ↦ #c
"Hγ" : own γ (Auth c)
--------------------------------------∗
▷ I γ l ∗ (∃ m : nat, ■ (#c = #m ∧ n ≤ m) ∧ C l m)
```

As shown, we now have temporary access to the physical location $\ell \mapsto c$ (which was needed to execute the dereference), as well as ownership of $\bullet\,(c)$. Instead of giving up these resources immediately for proving the invariant $I(\gamma, \ell)$, we will use them to establish $n \leq c$. This property follows from the rule OWN-VALID in Figure 5, which is again just a lemma. Since the rule is a lemma, we can use the `iDestruct` tactic to *eliminate* this pure fact into the Coq context:

```
iDestruct (own_valid γ (Frag n · Auth c) with "[#]") as %H
```

The suffix `as %H` denotes that the pure fact $n \leq c$ should be put in the Coq context as a named hypothesis `H`. We use the specialization pattern `[#]` because pure facts – such as $n \leq c$ – are persistent. We thus do not have to give up resources for proving their obligations. The proof obligation of the above tactic is $\boxed{\circ\,(n) \cdot \bullet\,(c)}^{\gamma}$, and can be proven using the rule OWN-OP. In IPM, this corresponds to the application of the lemma `own_op`.

In order to establish $C(\ell, c)$ in the postcondition, we also need ownership of $\circ\,(c)$, which can be obtained by using the rule OWN-OP once more. Having established $n \leq c$ and $\circ\,(c)$, we can now close the invariant, and finish the proof.

Proof of `incr`. The proof of `incr` is a bit more involved. Firstly, since it involves a loop implemented by recursion, we use Löb induction. However, what is more interesting, is that the value of ℓ will be changed if the `CAS` succeeds. The proof state corresponding to the part of the proof where the `CAS` succeeds is as follows:

```
"IH" : C l n -∗ WP incr #l {{ v, v = #() ∧ C l (1+n) }}
"Hinv" : inv N (I γ l)
--------------------------------------□
"Hl" : l ↦ #c
"Hγ" : own γ (Auth c)
--------------------------------------∗
WP CAS #l #c #(1 + c) @ ⊤ ∖ N {{ v, v, I γ l ∗ ... }}
```

What is happening here is that the `CAS` is going to change the physical value of ℓ into $1 + c$. Therefore, in order to reestablish the invariant $I(\gamma, \ell)$ in the postcondition, we have to update the ghost variable γ from $\bullet\,(c)$ into $\bullet\,(1 + c)$.

Updates to ghost variables are called *frame-preserving updates* and can be performed using the rule UPD-OWN. We can perform a frame-preserving update $a \rightsquigarrow b$ on a ghost variable γ, if the update ensures that no matter what assumptions the rest of the program is making about the state of γ, if these assumptions were compatible with a, they should also be compatible with b. See Figure 5 for the formal definition of $a \rightsquigarrow b$.

We only need one frame-preserving update for our example:

```
Lemma M_update n : Auth c ~~> Auth (1 + c).
```

[10] The lemma for opening invariants is proved using IPM in terms of more primitive rules of Iris [26].

This update holds because for any a_{f} with $\bullet\,(c) \cdot a_{\mathrm{f}} \neq \bot$ we have $a_{\mathrm{f}} = \circ\,(n)$ with $n \leq c$, so clearly we also have $\bullet\,(1+c) \cdot a_{\mathrm{f}} = \bullet\,(1 + c) \cdot \circ\,(n) \neq \bot$ because $n \leq 1 + c$.

All operations on the ghost state of Iris are performed through the *update modality* [26] $\Rightarrow$. The update $\Rightarrow P$ asserts ownership of resources that can be updated to resources satisfying P, as witnessed by the rule UPD-OWN. The rules UPD-INTRO and UPD-TRANS state that $\Rightarrow$ is a monad, while UPD-FRAME allows framing below $\Rightarrow$. The rule UPD-WP says that we can eliminate an update modality while proving a weakest precondition.

Since *all* operations on ghost state are performed through $\Rightarrow$, it is fairly easy to support reasoning about ghost state in IPM: we only need a tactic that corresponds to the rule UPD-WP. The syntax for the tactic for eliminating an update modality $H : \forall x_1 \ldots x_n.\, Q_1 \mathrel{-\!\!*} \cdots \mathrel{-\!\!*} Q_m \mathrel{-\!\!*} \Rightarrow P$ is:

`iMod (`H `$!` $t_1 \ldots t_n$ `with "`$\mathtt{spat}_1 \ldots \mathtt{spat}_m$`") as` `ipat`

The specialization patterns $\mathtt{spat}_i$ describe which hypotheses will be consumed by the preconditions Q_i, while the introduction pattern `ipat` describes how the postcondition P should be eliminated.

So, to update the ghost variable in our example, we can write `iMod (own_update with "Hγ") as "Hγ"`. This tactic yields the side condition $\bullet\,(c) \rightsquigarrow ?a$, in which we have to prove that there is a frame preserving update to some $?a$. This side condition can be solved using the already proven lemma `M_update`. The goal then becomes (the hypotheses `IH`, `Hinv` and `Hl` are omitted):

```
"Hγ" : own γ (Auth (1 + c))
--------------------------------------∗
WP CAS #l #c #(1 + c) @ ⊤ ∖ N {{ v, I γ l ∗ ... }}
```

At this moment we have finished the difficult part of the proof. The remaining part involves executing the `CAS`, reestablishing the invariant, and finally proving the postcondition.

6. Case Study: Logical Relations

In this section we describe a large case study using IPM: a formalization of logical relations for $\mathsf{F}_{\mu,ref,conc}$: a call-by-value System F-like language with recursive types, general references, and concurrency. The logical relations interpretation we formalize is defined in Iris, enabling us to make use of the expressive built-in features of Iris, such as guarded recursive predicates and invariants, which would not have been possible when performing the formalization directly in the Coq meta logic.

The idea of expressing a relational interpretation of a type system in a logic goes back to Plotkin and Abadi [34], who showed how to do it for System F, and Dreyer *et al.* [15], who showed how to do it for a language with recursive types. The definition of the logical relation that we consider is due to Krogh-Jespersen *et al.* [27], and their definition is, in turn, based on an earlier interpretation in the CaReSL logic [41]. The contribution of this case study is thus not a new logical relation, but rather showing that binary logical relations for a higher-order concurrent imperative programming language can be *formalized in a proof assistant*, which has not been done before. IPM makes it feasible to formalize this logical relation *in* Iris and to do the proofs (Theorem 6.1 and 6.2) *in* the Iris logic, without having to reason explicitly about steps, possible worlds, *etc.*

6.1 The Language $\mathsf{F}_{\mu,ref,conc}$

Iris, as well as IPM, is not tied to a fixed programming language, but can be instantiated with different concrete programming languages, which is crucial in this case study. Instead of the ML-like language with named variables that we have used before, we now instantiate Iris with $\mathsf{F}_{\mu,ref,conc}$. This language is formalized in Coq using the Autosubst library for De Bruijn terms [35], which takes care of

$$
\begin{aligned}
e &::= x \mid \ell \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, e \mid \texttt{unfold}\, e \mid e\ e \\
&\quad \mid e\ \_ \mid \texttt{fork}\ \{e\} \mid \texttt{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid \texttt{CAS}(e, e, e) \\
v &::= n \mid \ell \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, v \\
\tau &::= X \mid \mathbb{N} \mid \tau \to \tau \mid \forall X.\, \tau \mid \mu X.\, \tau \mid \texttt{ref}(\tau)
\end{aligned}
$$

Figure 6. The syntax of $\mathsf{F}_{\mu,ref,conc}$ (sums and products omitted).

Thread-local CBV head-reduction (omitted): $(e, \sigma) \to_{\mathsf{h}} (e', \sigma')$

Thread-pool reduction: $(\vec{e}, \sigma) \to_{\mathsf{tp}} (\vec{e}', \sigma')$

$$
\frac{(e, \sigma) \to_{\mathsf{h}} (e', \sigma')}{(\vec{e}_1\ K[\,e\,]\ \vec{e}_2, \sigma) \to_{\mathsf{tp}} (\vec{e}_1\ K[\,e'\,]\ \vec{e}_2, \sigma')}
$$

$$
(\vec{e}_1\ K[\,\texttt{fork}\ \{e\}\,]\ \vec{e}_2, \sigma) \to_{\mathsf{tp}} (\vec{e}_1\ K[\,()\,]\ \vec{e}_2\ e, \sigma)
$$

Figure 7. Operational semantics of $\mathsf{F}_{\mu,ref,conc}$.

synthesizing the substitution operation and substitution lemmas for our language. De Bruijn terms make it easier to deal with substitution on open terms, which is needed for the proofs in this section.

The terms and types of $\mathsf{F}_{\mu,ref,conc}$ can be found in Figure 6. Terms are untyped, so type-level abstraction is written as $\Lambda\, e$ and type application as $e\ \_$, as in [2]. The operational semantics is split into two parts: thread-local head reduction $\to_{\mathsf{h}}$ and thread-pool reduction $\to_{\mathsf{tp}}$, see Figure 7. Both are defined using standard call-by-value evaluation contexts K, whose definition is omitted. Thread-pool reduction is defined on configurations $\rho = (\vec{e}, \sigma)$ consisting of a state σ (a finite partial map from locations to values) and a thread-pool $\vec{e}$ (a list of expressions corresponding to the threads). The thread-pool reduction is defined by interleaving, *i.e.*, by picking a thread and executing it, thread-locally, for one step. The only special case is $\texttt{fork}\ \{e\}$, which spawns a new thread e, and reduces itself to the unit value $()$.

Typing judgments take the form $\Xi \mid \Gamma \vdash e : \tau$, where Ξ is a context of type variables, and Γ is a context assigning types to program variables. The inference rules for the typing judgment are mostly standard and hence omitted.

6.2 Unary Logical Relation

The unary logical relation for $\mathsf{F}_{\mu,ref,conc}$ is presented in Figure 8. The logical relation is defined by two relations for each type τ, a value interpretation $[\![\tau]\!]_\Delta : \mathit{Val} \to \mathit{iProp}$ and an expression interpretation $[\![\tau]\!]^{\mathcal{E}}_\Delta : \mathit{Expr} \to \mathit{iProp}$. Note that these relations are Iris relations! Formally, they are defined on types τ in context Ξ, but we omit the Ξ here for notational simplicity. We furthermore omit product, sum and base types, but these are present in the Coq formalization. The Δ is a mapping from type variables to value interpretations, *i.e.*, $\Delta : \mathit{Tvar} \to \mathit{Val} \to \mathit{iProp}$.

Experts on logical relations will recognize that this definition is extremely compact; this is because we express the relations in Iris. The case for function types expresses the usual requirement that a value is in the interpretation if it maps a value in the argument type to an expression in the result type. The $\Box$ modality (here and elsewhere) is used to ensure that the relation is persistent, which it must be since typing is intuitionistic (consider for example that the context Γ is copied in the usual typing rule for products). The definition for recursive types is given using a recursively defined predicate; this is well-defined in Iris since the recursion variable occurs under the later $\triangleright$ modality.

We use the Iris invariant $\boxed{\exists w.\, \ell \mapsto w * [\![\tau]\!]_\Delta(w)}^{\mathcal{N}.\ell}$ to express that a value is in the interpretation of a reference type $\texttt{ref}(\tau)$. That

$$
\begin{aligned}
[\![X]\!]_\Delta(v) &\triangleq \Delta(X)(v) \\
[\![\tau_1 \to \tau_2]\!]_\Delta(v) &\triangleq \Box(\forall w.\, [\![\tau_1]\!]_\Delta(w) \to [\![\tau_2]\!]^{\mathcal{E}}_\Delta(v\ w)) \\
[\![\forall X.\, \tau]\!]_\Delta(v) &\triangleq \forall f.\, \Box [\![\tau]\!]^{\mathcal{E}}_{\Delta[X \mapsto f]}(v\ \_) \\
[\![\mu X.\, \tau]\!]_\Delta(v) &\triangleq \mu\, f.\, \exists w.\, v = \texttt{fold}\, w \wedge \triangleright [\![\tau]\!]_{\Delta[X \mapsto f]}(w) \\
[\![\texttt{ref}(\tau)]\!]_\Delta(v) &\triangleq \exists \ell.\, v = \ell \wedge \boxed{\exists w.\, \ell \mapsto w * [\![\tau]\!]_\Delta(w)}^{\mathcal{N}.\ell} \\
[\![\tau]\!]^{\mathcal{E}}_\Delta(e) &\triangleq \mathsf{wp}\, e\, \{v.\, [\![\tau]\!]_\Delta(v)\}
\end{aligned}
$$

Figure 8. The unary logical relation for $\mathsf{F}_{\mu,ref,conc}$.

is, the value is a location ℓ and, invariantly, the location ℓ contains a value w in memory that is in the interpretation of τ. This use of Iris invariants dispels the need for explicit possible worlds and explicit treatment of the type-world circularity, which is otherwise typical for logical relations for reference types [1, 8].

Finally, the expression relation $[\![\tau]\!]^{\mathcal{E}}_\Delta$ says that e is in the semantic interpretation of τ, if it is a computation whose possibly resulting value v is in the semantic interpretation of τ. Using the expression relation, we can define the semantic interpretation of types as:[11]

$$
\Xi \mid \Gamma \vDash e : \tau \triangleq \forall \Delta\, \vec{v}.\ \left(\bigwedge\nolimits_i [\![\sigma_i]\!]_\Delta(v_i)\right) \vdash [\![\tau]\!]^{\mathcal{E}}_\Delta(e[\vec{v}/\vec{x}]) \qquad (1)
$$

where $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and the environments $\Delta : \mathit{Tvar} \to \mathit{Val} \to \mathit{iProp}$ map into persistent interpretations.

For this logical relation, we can now prove:

Theorem 6.1 (Fundamental theorem of logical relations)**.**

```
Theorem fundamental Γ e τ : Γ ⊢ₜ e : τ → Γ ⊨ e : τ.
```

Theorem 6.2 (Type soundness)**.** *Reduction of any well-typed expression can never get stuck:*

```
Corollary type_soundness e τ e' thp σ σ' :
  [] ⊢ₜ e : τ → rtc step ([e], σ) (e' :: thp, σ') →
  is_Some (to_val e') ∨ reducible e' σ'.
```

Theorem 6.1 is proven in Iris using IPM. Theorem 6.2 is formalized in plain Coq and relies on the fundamental theorem and the adequacy result for Iris, which formalizes that the weakest precondition predicate of Iris really is connected to the operational semantics of $\mathsf{F}_{\mu,ref,conc}$ in the way you would expect.

Note that the corollary `type_soundness` shows the true power of using a proof assistant instead of a standalone tool: we can compose a proof in Iris with the adequacy result of Iris into a corollary that only mentions the typing judgment and the operational semantics. So, one no longer has to trust Iris or IPM!

We now demonstrate how IPM is used to prove a case of the fundamental lemma. Below we display the proof goal for showing that a recursive function $\texttt{rec}\, f(x) = e$ is in the expression interpretation $[\![\tau_1 \to \tau_2]\!]^{\mathcal{E}}_\Delta$ for the arrow type. The `HΓ` hypothesis contains the assumptions from the semantic interpretation of types (1).

```
"HΓ" : ⟦ Γ ⟧* Δ vs
------------------------------------□
⟦ TArrow τ1 τ2 ⟧ₑ Δ (Rec e.[upn 2 (env_subst vs)])
```

Here the `upn 2` comes from the fact that we are using De Bruijn indices and the recursive function definition binds two variables. We proceed by Löb induction (since we have to prove that a *recursive* function is in logical relation), expand the definition of $[\![\tau_1 \to \tau_2]\!]^{\mathcal{E}}_\Delta$,

[11] Contexts Ξ do not appear in the Coq code since we use De Bruijn indices.

and symbolically execute the `rec` $f(x) = e$ construct. The proof goal then becomes:

```
"HΓ" : ⟦ Γ ⟧* Δ vs
"IH" : ∀ v, ⟦ τ1 ⟧ Δ v →
      WP App (Rec e.[upn 2 (env_subst vs)]) (of_val v)
      {{ v, ⟦ τ2 ⟧ Δ v }}
"Hw" : ⟦ τ1 ⟧ Δ w
--------------------------------------□
WP e.[env_subst (RecV e.[upn 2 (env_subst vs)] :: w :: vs)]
{{ v, ⟦ τ2 ⟧ Δ v }}
```

Here `IH` is the Löb induction hypothesis and `Hw` is the assumption that an argument w is in the interpretation of the domain type. We then have to show that the body of the function, with the recursive function substituted for f and w substituted for the argument x, is in the interpretation of the codomain type.

This example illustrates that the formal reasoning mirrors the argument that one would do on paper. The total length of the proof for this case is 7 lines of IPM tactics.

6.3 Binary Logical Relation

We have also defined a binary logical relation for $\mathsf{F}_{\mu,ref,conc}$ and proven that logical relatedness implies contextual approximation. We defer a detailed treatment to our IPM formalization, and just give a quick overview here. It is not too hard to generalize the unary logical *value* interpretation to a binary relation, but to generalize the *expression* interpretation from the unary logical relation to the binary logical relation, one needs to find some way of expressing a *relation* between two expressions e and e' using weakest precondition predicates, which are unary. This can be done as follows:

$$[\![\tau]\!]^{\mathcal{E}}_{\Delta}(e,e') \triangleq \forall jK.\, j \Rrightarrow K[\,e'\,] \to \mathsf{wp}\, e\, \{v.\, \exists w.\, j \Rrightarrow K[\,w\,] * [\![\tau]\!]_{\Delta}(v,w)\}$$

Here $j \Rrightarrow K[\,e'\,]$ is a predicate on ghost state, which expresses that the specification side e' is in some evaluation context K for some thread j before we run e. In the post-condition, we have $j \Rrightarrow K[\,w\,]$. Together with an appropriate invariant on ghost state, these predicates ensure that we really are relating the execution of e, which results in a value v to an execution of e', which results in a value w, and that those values are in the binary value interpretation of the type τ.

The ghost state and the appropriate predicates on ghost states can be defined in Iris, by combining various monoid constructions.

Logical relatedness of e and e' is then defined as:

$$\Xi \mid \Gamma \vdash e \le_{\log} e' : \tau \triangleq \forall \vec{v}\, \vec{v}'\, \Delta\, \rho.\ \boxed{\Psi(\rho)}^{\mathcal{N}'} * \left(\bigwedge\nolimits_i [\![\sigma_i]\!]_{\Delta}(v_i, v_i')\right) \vdash [\![\tau]\!]^{\mathcal{E}}_{\Delta}(e[\vec{v}/\vec{x}], e'[\vec{v}'/\vec{x}])$$

where $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and the environments $\Delta : \mathit{Tvar} \to \mathit{Val} \to \mathit{iProp}$ map into persistent interpretations. Here $\boxed{\Psi(\rho)}^{\mathcal{N}'}$ is the invariant on ghost states mentioned above. Note again that logical relatedness is an Iris predicate.

The fact that logical relatedness implies contextual approximation is shown by a series of congruence lemmas, corresponding to each of the typing rules (each on average about 10 lines of code). These lemmas are proved in Iris using IPM.

6.4 Proving Logical Refinements

We have used the binary logical relation to prove that two fine-grained concurrent implementations of modules contextually refines their coarse-grained counterparts. The first example is a counter module and the second is a stack module. The fine-grained implementations use optimistic concurrency and no locks, whereas the coarse-grained implementations use a spin lock (implemented using a `CAS` loop) to lock the data structure of the module before and after an operation is performed on the data structure.

For the counter, we use an Iris invariant relating the reference cells in the two implementations and the lock used in the coarse-grained implementation. The stack example comes from [41], where it was proved on paper using an invariant formulated using state transition system. State transition systems can be encoded in Iris [24], but in our experience, it is often easier to use direct monoid constructions when working in Coq.

7. Evaluation

The IPM implementation on top of Iris consists of only 3.086 lines of code, but it relies on the formalization of the Iris logic, which is 12.373 lines of code [23, 26], and a support library for basic data structures (*e.g.*, lists, sets, maps) by Krebbers [25].

In Figure 9 we present some data for a variety of examples.[12] The first table shows data for the spin lock and the counter, discussed earlier in the paper, and three examples (the spawn/join primitives, the parallel composition operator, and a barrier synchronization primitive), which were originally formalized by Jung *et al.* [23], and which we reproved using IPM. The second table shows data for the unary and binary logical relations, and the third table data for the refinement proofs. In these tables, the 'Monoids' column shows the number of lines for monoid definitions (in addition to those provided by the Iris formalization), the 'Program and proof' column shows the number of lines in Coq for program and proof, the 'Build' column shows the Coq build time, and the 'Build*' shows the Coq build time with a new universe cycle detection algorithm implemented by Jourdan.[13]

As shown in Figure 9, the program correctness proofs are rather small, as are the proofs for the logical relations. We have not put effort into optimizing the proofs for the program refinements. In particular, they contain lots of boilerplate code to avoid performance issues of the Autosubst library.[14]

It turns out that the majority of our compilation time is spent on universe checking, which is probably due to the way that Iris is parametrized by the user chosen monoids, an issue orthogonal to IPM. Using a version of Coq with the new universe cycle detection algorithm by Jourdan reduces times drastically! Out of the remaining 70s for the stack refinement proof, 28s is spent on Autosubst.

8. Related Work

Proofs in the model. Doing proofs by expanding the connectives of a shallowly embedded separation logic into their interpretation is troublesome. This is due to the fact that separating conjunction is only defined when states are disjoint. By working in the model, one thus has to reason explicitly about disjointness of states. When done naively, this results in an excessive number of proof obligations. In order to circumvent this issue, Nanevski *et al.* [32] have shown that by defining states and disjoint union of states in a different way, one can get rid of most proof obligations related to disjointness.

In more recent work, Sergey *et al.* have shown that this approach scales to concurrent separation logic [36]. Using their Coq formalization of the FCSL logic [31], they have verified several fine-grained concurrent algorithms, including lock implementations, a concurrent spanning tree algorithm and Treiber stack.

One drawback of this approach is that it is limited to particular kinds of object logics. Most importantly, in order to use the encoding

[12] Machine used: Intel Core i7, 3.40GHz, 16GB ram, running Debian.

[13] Available at `https://github.com/coq/coq/pull/178`.

[14] We contacted the authors of Autosubst and learnt that it is primarily meant for meta theory, and that its performance is known to be insufficient for verification of concrete programs.

Program correctness	Monoids	Program and proof	Build	Build*
Spin lock	0	90	6.4s	4.0s
Monotone counter	0	84	8.5s	5.3
Spawn/join	0	81	7.5s	4.5s
Par	0	47	6.4s	4.8s
Barrier	0	325	30.2s	24.0s
Program refinements	**Monoids**	**Program and proof**	**Build**	**Build***
Counter	0	370	45s	21s
Stack	273	1.284	4m23s	1m10s

	Unary	Binary
Monoids	0	340
Logical relation	192	255
Fundamental	143	416
Soundness	27	354
Total	362	1.365
Total build	1m4s	2m48s
Total build*	23s	1m11s

Figure 9. Line counts and compilation times of the Coq formalization.

of Nanevski *et al.*, it is crucial to have a procedure that *decides* whether states are disjoint. However, for many logics, for example VST [5] and Iris [23], this is not the case. These logics have an expressive notion of ghost state, that allows one to store arbitrary propositions in the (ghost) state. The monoidal composition on states is only defined (*i.e.* disjoint) if both propositions are the same, which is undecidable. Also, it is unclear if their approach is able to handle logics with a step-indexed model, *e.g.* iCAP [40], CaReSL [41], VST [5] or Iris [24, 23]. These logics use step-indexing to model higher-order storable procedures, which cannot be handled by FCSL.

Interactive proofs. Some of the earliest work in this area is by Appel [4], who created a family of tactics that help out with the basic bookkeeping that is involved while reasoning in a separation logic. This work has later been extended by McCreight [29], who managed to cut down proof sizes drastically. In later work, Bengtson *et al.* have continued this line of work by implementing similar tactics in a more language independent fashion [6]. Although the philosophy of these implementations is the same as ours – namely, being able to reason in the logic, instead of the model – there are a couple of essential differences.

Firstly, these implementations do not have a structured context for the hypotheses in the object logic. Hence, one cannot refer to individual hypotheses, which makes it impossible to implement tactics for introduction and elimination of all logical connectives. Instead, these implementations provide ad-hoc tactics to support bookkeeping during symbolic execution, like pulling out pure propositions and existential quantifiers into the Coq context, framing, and rearranging the goal up to commutativity and associativity.

Furthermore, most of these implementations consider a restricted fragment of separation logic that primarily consists of: the embedding of Coq level assertions $\blacksquare\,\phi$, separating conjunction $*$, existential quantification $\exists$, and the maps-to predicate $\ell \mapsto v$. IPM supports *all* connectives of higher-order separation logic, including magic wand $-\!\!*$ and higher-order quantification, but also the entire Iris logic. Iris contains many connectives that would be non-trivial to handle in these previous implementations: the always $\Box$ and later $\triangleright$ modality, ghost ownership $\boxed{a}^{\gamma}$, the update $\Rrightarrow$ modality, and invariants $\boxed{P}^{\mathcal{N}}$.

The Verifast tool by Jacobs *et al.* [21] is a standalone tool for interactive program verification in separation logic. The style of reasoning in Verifast is quite different from ours. In Verifast one instructs the system by writing assertions throughout the program, whereas reasoning in IPM is done through tactics. Since there are no tactics, there is no need to refer to hypotheses by name, and as such, Verifast displays the spatial context as a multiset. Furthermore, like all other implementations that we have discussed, Verifast is meant for proving Hoare triples, so it is not suited for formalization of a logical relation and using that logical relation for proving refinements (*cf.* Section 6).

Automated proofs. In order to avoid the need for the end-user to deal with the logic altogether, there has been a lot of work on tools for automated proofs in separation logic. However, all the tools that we are aware of are primarily focused on program verification, and are limited to restricted fragments of separation logic so as to make effective automation possible.

In future work we would like to integrate support for better proof automation into IPM, for which the work in this area will be useful. We are especially interested in Chlipala's Bedrock [9] and Malecha and Bengtson's RTac [28].

9. Conclusions

We have introduced a method to make interactive reasoning using Coq in embedded logics – like separation logic – easy. The key idea of our method is to use a named context to represent the hypotheses of the object logic. This has the following advantages:

- It makes it possible to render the proof state of the object logic in the same way as the proof state of the meta logic. This gives users the impression that they are *reasoning in* the object logic, instead of *manipulating* the object via the meta logic.
- It makes it possible to do proofs at the right level of abstractions. Unlike most previous work, there is no longer a need to reason up to associativity and commutativity, but instead we provide high-level tactics for introductions and eliminations of *all* connectives of higher-order separation logic.
- It enables us to deeply embed the contexts of the object logic, while using a shallow embedding of the logic. This gives the best of two worlds: we can borrow many features of the meta logic (like its support for binding and higher-order quantification to model $\forall$ and $\exists$) while being able to implement the aforementioned tactics efficiently using computational reflection.

We have successfully implemented our method on top of Iris to show its effectiveness. We have formalizing functional correctness of fine-grained concurrent algorithms, some of the meta theory of Iris, a unary and binary logical relation for a higher-order concurrent imperative programming language, and logical refinements of a fine- and coarse-grained counter and stack using this logical relation.

The aforementioned applications show that we are really using the object logic (in our case Iris) as a *logic*, and not just as a *program logic* for verifying Hoare triples, which emphasizes the need for *interactive* reasoning as provided by our method.

Acknowledgments

We wish to thank Ralf Jung, Jacques Henri-Jourdan, Aleš Bizjak, Morten Krogh-Jespersen, Jan-Oliver Kaiser, Zhen Zhang and Joseph Tassarotti for testing IPM by using it as part of their Coq developments. We are also grateful to the anonymous reviewers for their helpful comments and suggestions.

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU), and by the Flemish Research Fund (grant G.0058.13).

References

[1] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[2] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[3] A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.

[4] A. W. Appel. Tactics for Separation Logic, 2006. Available at `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`.

[5] A. W. Appel, editor. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[6] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - A Framework for Higher-Order Separation Logic in Coq. In *ITP*, volume 7406 of *LNCS*, pages 315–331, 2012.

[7] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68, 2005.

[8] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, 2011.

[9] A. Chlipala. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP*, pages 391–402, 2013.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2016. Available at `https://coq.inria.fr/doc/`.

[11] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, pages 207–231, 2014.

[12] D. Delahaye. A Tactic Language for the System Coq. In *LPAR*, volume 1955 of *LNCS*, pages 85–95, 2000.

[13] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[14] D. Dreyer. ERC Project "RustBelt", 2016. Available at `http://plv.mpi-sws.org/rustbelt/`.

[15] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *LMCS*, 7(2:16), 2011.

[16] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[17] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

[18] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, pages 388–402, 2010.

[19] M. Gordon and T. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[20] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979.

[21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM*, volume 6617 of *LNCS*, pages 41–55, 2011.

[22] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *POPL*, pages 301–314, 2013.

[23] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.

[24] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.

[25] R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University, 2015.

[26] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The Essence of Higher-Order Concurrent Separation Logic, 2016. Draft.

[27] M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. A Logical Account of a Type-and-Effect System. In *POPL*, 2017.

[28] G. Malecha and J. Bengtson. Extensible and Efficient Automation Through Reflective Tactics. In *ESOP*, volume 9632 of *LNCS*, pages 532–559, 2016.

[29] A. McCreight. Practical Tactics for Separation Logic. In *TPHOLs*, volume 5674 of *LNCS*, pages 343–358, 2009.

[30] H. Nakano. A modality for recursion. In *LICS*, 2000.

[31] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.

[32] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274, 2010.

[33] A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI*, 2011.

[34] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA*, 1993.

[35] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *ITP*, volume 9236 of *LNCS*, pages 359–374, 2015.

[36] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

[37] F. Sieczkowski, A. Bizjak, and L. Birkedal. ModuRes: A Coq library for modular reasoning about concurrent higher-order imperative programming languages. In *ITP*, volume 9236 of *LNCS*, pages 375–390, 2015.

[38] M. Sozeau and N. Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.

[39] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *MSCS*, 21(4):795–825, 2011.

[40] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.

[41] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[42] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[43] M. Wildmoser and T. Nipkow. Certifying Machine Code Safety: Shallow Versus Deep Embedding. In *TPHOLs*, volume 3223 of *LNCS*, pages 305–320, 2004.