Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic

Robbert Krebbers
Radboud University Nijmegen, Netherlands

Luko van der Maas
Radboud University Nijmegen, Netherlands

Enrico Tassi
Université Côte d'Azur, Inria, France

Abstract

Inductive predicates play a key role in program verification using separation logic. There are many methods for defining such predicates in separation logic, which all have different conditions and thus support different classes of predicates. The most common methods are: (1) through a structurally-recursive definition (commonly used to define representation predicates for the verification of data structures), and (2) through step-indexing (commonly used to give a semantics of Hoare triples for partial program correctness). A lesser-known method is to define such inductive predicates *internally* in higher-order separation logic through a least fixpoint of a monotone function.

The contributions of this paper are fourfold. First, we present the folklore result (from the Iris library) that one can define least (and greatest) fixpoints internally in separation logic by extending the standard second-order impredicative encoding with some modalities. Second, we show that these fixpoints are useful to define representation predicates where the mathematical and in-memory data structures do not correspond. Third, we show that these fixpoints can be used to define Hoare triples and weakest preconditions for *total* program correctness in Iris. Fourth, we present a prototype command (akin to Rocq's `Inductive`), written in Rocq-Elpi, to generate the least fixpoint and its reasoning principles (constructors and induction principles) from a high-level specification.

2012 ACM Subject Classification Theory of computation → Programming logic

Keywords and phrases Separation Logic, Program Verification, Data Structures, Iris, Rocq prover

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.27

Supplementary Material *Software (Rocq code)*: `https://doi.org/10.5281/zenodo.15727403`

Acknowledgements We thank Ralf Jung and Jacques-Henri Jourdan for early discussions about fixpoints in Iris, Aleš Bizjak for suggesting to define weakest preconditions using a least fixpoint, and the anonymous reviewers for their feedback. This work is supported in part by ERC grant COCONUT (grant no. 101171349), funded by the European Union, and NWO grant FuRoRe (OCENW.M.22.282). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Separation logic [54, 59] and its extension concurrent separation logic [52, 11] are widely used methods for the verification of imperative and concurrent programs. A key ingredient of separation logic is the use of *representation predicates* to specify data structures. Consider the following (slightly adapted) example from the seminal paper by Reynolds [59]:

$$\mathsf{isList}\ \ell\ [\,] \triangleq \ell \mapsto \mathtt{NIL} \qquad \mathsf{isList}\ \ell\ (v :: \vec{v}) \triangleq \exists \ell'.\ \ell \mapsto \mathtt{CONS}\ (\ell', v) * \mathsf{isList}\ \ell'\ \vec{v}$$

The predicate isList $\ell\ \vec{v}$ should be read as: the location ℓ in the heap contains a pointer-based linked list described by the mathematical list $\vec{v}$. Importantly, isList : loc → list val → iProp is not a predicate in the meta-logic (*e.g.,* Prop in Rocq), but a predicate in separation logic (*e.g.,* iProp in Iris) which describes a set of heaps. The definition of isList can hence make use of the *points-to assertion* $\ell \mapsto v$ (location ℓ contains value v) and the *separating conjunction* $P * Q$ (propositions P and Q hold in disjoint heaps).

With separation logic frameworks in proof assistants, such as BedRock [19], CFML [15], Charge [8], Iris [45, 37] and VST [12], one commonly defines these predicates by structural recursion on the mathematical data type. The list predicate is defined in Rocq using Iris as:

```
Fixpoint is_list (l : loc) (vs : list val) : iProp :=
  match vs with
  | [] => l ↦ NIL
  | v :: vs => ∃ l', l ↦ CONS (#l',v) ∗ is_list l' vs
  end.
```

This method is applicable to many (tree-like) data structures and is widely used in the literature because such predicates are easy to define and use. One obtains the recursive equations as definitional equalities, and one can perform structural induction on the mathematical list.

Problem But what if it is impossible to define the predicate by structural recursion, or it is too cumbersome to convince the guardedness/termination checker? When working in the meta logic, one would define the predicate inductively (`Inductive` in Rocq) instead of by structural recursion (`Fixpoint` in Rocq)—but *a priori* there is no analogue for defining inductive predicates in separation logic. Particularly, the standard `Inductive` command in Rocq cannot be used because `iProp` is not an *arity* of a sort [72] (see § 6 for details).

What is lesser known is that one can define inductive predicates internally in higher-order separation logic through a second-order impredicative encoding of least fixpoints, akin to the folklore encoding in second-order predicate logic and linear logic [7], which in turn is inspired by the Tarski-Knaster theorem [40, 66]. The Iris framework for separation logic in Rocq has a small library for least fixpoints (whose first version was written by Jung [35] in 2017), but it has seen less use in practice than the `Fixpoint` approach. We believe that is for at least two reasons. First, there is a lack of intuitive descriptions of applications where the least fixpoint approach shines. Second, there is no support for making the least fixpoint encoding usable in practice—the user is required to write an abundance of boilerplate.

We describe two applications of inductive predicates in separation logic. First, in the context of data structure verification, we show that inductive predicates are useful to define representation predicates if the mathematical and in-memory structure do not correspond. Second, we show that inductive predicates in separation logic are useful to develop the meta theory of separation logic itself. Specifically we show how to define Hoare triples and weakest preconditions for *total* program correctness internally in the Iris base logic. The first author added this construction as part of Iris in 2017 [42], and variants have been used in practice, *e.g.,* [28, 1, 79, 78], but the details have never been spelled out in a published paper.

To make the least fixpoint encoding practical, we present a prototype `Iris Inductive` command for inductively-defined predicates in separation logic, written in Rocq-Elpi [25, 69, 71], built on top of the Iris Proof Mode (IPM) [45, 43]. Inspired by proof assistants in the HOL family [49, 56, 33], our command transforms a high-level specification of an inductive predicate into low-level definitions—with the crucial difference that we are working in an *embedded separation logic* instead of the *unrestricted meta logic* of the proof assistant.

Application #1: Data structure verification Consider a variation of the representation predicate for lists, defined using our `Iris Inductive` command:

```
Iris Inductive is_del_list : loc → list val → iProp :=
  | is_del_list_nil l : l ↦ NIL -∗ is_del_list l []
  | is_del_list_cons l l' v vs :
     l ↦ CONS (#l',v) -∗ is_del_list l' vs -∗ is_del_list l (v :: vs)
  | is_del_list_del l l' vs : l ↦ DEL #l' -∗ is_del_list l' vs -∗ is_del_list l vs.
```

The in-memory representation contains an additional *deleted* node `DEL`. Similar to `CONS`, the `DEL` node contains a link to the next node, but does not hold a value. Such `DEL` nodes are useful in concurrent programming, where a node is first 'marked' as deleted and is only removed from the list later [32], or to atomically append queues [62]. Crucially, this representation predicate cannot be defined by structural recursion because the size of `vs` does not decrease in the `DEL` case. An alternative, as used by Somers and Krebbers [62] (in Iris), is to define the predicate by structural recursion on `list (option A)`, where `None` accounts for deleted nodes. However, that approach involves additional boilerplate and is not applicable to more advanced use cases such as our program logic for total correctness.

The `Iris` prefix indicates that the `Inductive` should be processed by our Rocq-Elpi prototype using the following steps. **1.** We define the fixpoint using a second-order impredicative encoding by transforming the constructors into a disjunction of existentially quantified cases. Here, we exploit Elpi's support for HOAS [57] to perform term surgery with binders. **2.** We prove monotonicity (*i.e.,* positivity) of the recursive argument using a goal-directed proof search. To support a variadic notion of monotonicity, we port the notion of *signatures* from Rocq's generalized rewriting framework [63] to separation logic. **3.** We generate the constructors and induction principles, making use of monotonicity. Here, we reimplement some of the IPM tactics in Elpi to avoid interfacing with the original IPM Ltac code. Finally, we provide a tactic akin to Rocq's `induction` that applies the induction principle.

Application #2: Total program correctness Iris employs the reductionist methodology to develop a large fragment of its meta theory in separation logic itself [44]. At its core, Iris features a *base logic*, which is a higher-order intuitionistic logic, with separating conjunction ($*$), magic wand ($-\!\!*$), a customizable notion of resource ownership, and a handful of modalities. The *program logic* (which includes weakest preconditions and Hoare triples) is encoded in the base logic. This methodology ensures that the definition of the program logic is concise and and can easily be adapted to other domains such as continuations [74], effect handlers [23], non-interference [27, 29], randomized algorithms [67], and crash safety [13].

The standard Iris program logic is, however, limited to partial program correctness and lacks support for total program correctness (*i.e.,* termination). Semantically speaking this limitation stems from the fact that the program logic is defined using step-indexing [4, 2, 5], which is coinductive in nature. We show that through a simple modification to the definition of the Iris program logic, we obtain a version for total correctness: we remove a modality, and instead of the step-indexed Banach's fixpoint, we use a least fixpoint. The key difference w.r.t. the original Iris program logic for partial correctness is that we cannot introduce a later modality during computation steps. Consequently we cannot use Löb induction to reason about loops and recursive functions, and are forced to use induction on a data type in the Rocq meta logic or an inductively-defined predicate in Iris to ensure termination.

Contributions and outline We show that inductive predicates, defined using a standard second-order encoding of least fixpoints in separation logic (§2), are useful assets for program

$$\mathsf{True} * P \dashv\vdash P \qquad P * Q \dashv\vdash Q * P \qquad (P * Q) * R \dashv\vdash P * (Q * R)$$

$$*\text{-MONO}\ \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad -\!\!*\text{-INTRO}\ \frac{P * Q \vdash R}{P \vdash Q -\!\!* R} \qquad -\!\!*\text{-ELIM}\ \frac{P \vdash Q -\!\!* R}{P * Q \vdash R}$$

$$\square\text{-DUP}\ \square P \dashv\vdash \square P * \square P \qquad \square\text{-ELIM}\ \square P \vdash P \qquad \square\text{-IDEM}\ \square P \dashv\vdash \square\square P$$

$$\square\text{-MONO}\ \frac{P \vdash Q}{\square P \vdash \square Q} \qquad \square\text{-True}\ \square\,\mathsf{True} \dashv\vdash \mathsf{True}$$

Figure 1 Selected laws of the Iris base logic. (We use $\dashv\vdash$ as notation for bidirectional entailment.)

verification. Both for the verification of data structures (§ 3) and for developing the meta theory of a program logic for total program correctness (§ 4). We present a prototype `Iris Inductive` command, written in Rocq-Elpi, which given a high-level specification generates the low-level definitions (§ 5). We conclude by discussing related (§ 6) and future work (§ 7). All our results are mechanized in the Rocq prover using the Iris framework [46]. An extensive description of the command is in the MSc thesis of the second author [77].

Limitations **1.** We consider our `Iris Inductive` command to be a *prototype* because it is unclear how how to avoid the duplication of IPM tactics in Ltac and Elpi, and thus integrate it into the production version of Iris (see § 7). **2.** While our program logic is as least as powerful as variants of 'traditional' separation logic for total correctness, *e.g.,* CFML [14, 15, 16], it is too weak for blocking concurrency as it does not integrate the assumption of fair scheduling. We consider reasoning about the scheduler orthogonal and leave that for future work.

2 Fixpoints in Higher-Order Separation Logic

In this section we give a brief introduction to higher-order separation logic, and describe the folklore result of defining fixpoints using a second-order impredicative encoding. This construction works for any higher-order predicate BI [53, 10] with a persistence modality, *i.e.,* a Modal BI (MoBI) [43]. For brevity's sake, we present the construction specifically for the Iris base logic, which is an instance of a MoBI.[1] The fixpoint construction in the Iris base logic was mechanized by Jung [35], and has been ported to MoBIs by Krebbers [41].

The Iris base logic The propositions of the Iris base logic iProp include the standard connectives of higher-order intuitionistic logic with equality (*i.e.,* $\mathsf{True}, \mathsf{False}, \wedge, \vee, \Rightarrow, \forall, \exists, =$). The key feature of a higher-order logic is that the domain A in $\forall x : A.\, P$ and $\exists x : A.\, P$ can be any type, and can even include the type of propositions iProp (impredicativity). Particularly, one can quantify over predicates $\forall \Phi : A \to \mathsf{iProp}.\, P$, which is crucial for the fixpoint encoding.

The Iris base logic includes the separating conjunction ($*$), magic wand ($-\!\!*$), and persistence modality ($\square$). Selected laws are shown in Figure 1. To obtain an intuition for these connectives, it is useful to think of the model of separation logic, where propositions denote ownership of resources. The Iris base logic supports user-defined (higher-order ghost)

[1] (Mo)BIs have two unit elements—True for ordinary conjunction ($\wedge$) and Emp for separating conjunction ($*$)—while the Iris base logic has a singular unit element True. Similarly, (Mo)BIs have have a *persistence* ($\boxdot$) and *intuitionistic* ($\square$) modality, which coincide for the Iris base logic [43, §2.4].

resources [39, 36, 37], but for simplicity we consider the resources to be heaps (finite maps from locations to values). This means we have a primitive $\ell \mapsto v$, which holds for a heap h, if it contains a location ℓ with value v. The separating conjunction $P * Q$ holds for a heap h, if it can be split into disjoint heaps h_1 and h_2 in which P and Q hold, respectively. The magic wand $P \mathrel{-\!\!*} Q$ holds for a heap h, if for any disjoint heap h' that satisfies P, we have that Q holds for the disjoint union of h and h'. The separating conjunction is associative, commutative and monotone, and has True as its unit. The laws $\mathrel{-\!\!*}$-INTRO and $\mathrel{-\!\!*}$-ELIM express that $*$ and $\mathrel{-\!\!*}$ interact in the same way as $\wedge$ and $\Rightarrow$ (*i.e.,* they are adjoints).

Separation logic is substructural, hence we *do not* have $P * P \dashv\vdash P$. However, for some propositions P that law does hold, particularly the *persistent* propositions—those that do not assert (exclusive) ownership of resources. Simple examples of persistent propositions are True, False and equality, but Iris features many more (*e.g.,* invariant assertions $\boxed{P}$). To reason about persistent propositions, Iris employs the *persistence modality* ($\Box$), and says that a proposition P is persistent iff $P \vdash \Box P$. In the heap model, $\Box P$ holds for a heap h, if P holds for the empty heap $\emptyset$. Persistent propositions are duplicable ($\Box$-DUP) and the $\Box$ modality can be eliminated ($\Box$-ELIM). The $\Box$ modality is idempotent and monotone, and commutes with (separating) conjunction, disjunction and quantifiers. The $\Box$ modality can be introduced if the hypotheses are persistent, as expressed by the following derived law:

$$\frac{P \vdash Q \qquad \mathsf{persistent}(P)}{P \vdash \Box Q}\ \Box\text{-INTRO}$$

Fixpoints We define the least (μ) and greatest (ν) fixpoint operators, which take a *pre-fixpoint function* $F : (A \to \mathsf{iProp}) \to (A \to \mathsf{iProp})$, and return a fixpoint $A \to \mathsf{iProp}$. The argument $\mathit{rec} : A \to \mathsf{iProp}$ of F corresponds to the recursive occurrence. For example, for the list representation predicate with *deleted* nodes from § 1 (which cannot be defined directly by structural recursion), we use the following pre-fixpoint function (with $A \triangleq \mathsf{loc} \times \mathsf{list\ val}$):

$$\begin{aligned} F_{\mathsf{isDelList}}\ \mathit{rec}\ (\ell, \vec{v}) \triangleq\ & (\ell \mapsto \mathtt{NIL} * \vec{v} = [\,]) \lor \\ & (\exists \ell', w, \vec{w}.\ \ell \mapsto \mathtt{CONS}\ (w,\ \ell') * \mathit{rec}\ (\ell', \vec{w}) * \vec{v} = w :: \vec{w}) \lor \\ & (\exists \ell'.\ \ell \mapsto \mathtt{DEL}\ \ell' * \mathit{rec}\ (\ell', \vec{v})) \end{aligned}$$

We now define $\mathsf{isDelList} : \mathsf{loc} \to \mathsf{list\ val} \to \mathsf{iProp}$ as $\lambda \ell, \vec{v}.\ \mu\, F_{\mathsf{isDelList}}\ (\ell, \vec{v})$. A fixpoint of F is well-defined if all occurrences of rec are in positive position, *i.e.,* if F is monotone:

$$\forall (\Phi_1, \Phi_2 : A \to \mathsf{iProp}).\ \Box(\forall x : A.\ \Phi_1\, x \mathrel{-\!\!*} \Phi_2\, x) \mathrel{-\!\!*} \forall x : A.\ F\, \Phi_1\, x \mathrel{-\!\!*} F\, \Phi_2\, x$$

Since we work in separation logic, we use the magic wand ($\mathrel{-\!\!*}$) instead of implication ($\Rightarrow$). The persistence modality ($\Box$) allows $\forall x : A.\ \Phi_1\, x \mathrel{-\!\!*} \Phi_2\, x$ to be used multiple times, which is needed for predicates with multiple recursive occurrences (`is_search_tree` in § 3). To our knowledge, dropping the modality does not give benefits if there is one recursive occurence.

▶ **Theorem 1.** *Given a monotone pre-fixpoint function $F : (A \to \mathit{iProp}) \to (A \to \mathit{iProp})$, there exists a least fixpoint $\mu\, F : A \to \mathit{iProp}$ that satisfies:*

1. *(Fixpoint equations) $F\,(\mu\, F)\, x \mathrel{-\!\!*} \mu\, F\, x$ and $\mu\, F\, x \mathrel{-\!\!*} F\,(\mu\, F)\, x$, and,*
2. *(Iteration principle) $\Box(\forall x.\ F\, \Phi\, x \mathrel{-\!\!*} \Phi\, x) \mathrel{-\!\!*} \forall x.\ \mu\, F\, x \mathrel{-\!\!*} \Phi\, x$.*

Proof. We define the least fixpoint as:

$$\mu\, F \triangleq \lambda x.\ \forall (\Phi : A \to \mathsf{iProp}).\ \Box(\forall y.\ F\, \Phi\, y \mathrel{-\!\!*} \Phi\, y) \mathrel{-\!\!*} \Phi\, x$$

This definition is analogue to the folklore encoding in higher-order logic, but uses magic wand ($-\!\!*$) instead of implication ($\Rightarrow$). Similar to the definition in second-order linear logic [7], which uses the bang operator (!), it uses Iris's persistence modality ($\Box$).

1. Proof of $F\,(\mu\,F)\,x \mathrel{-\!\!*} \mu\,F\,x$. Assume $H_F : F\,(\mu\,F)\,x$. Now given an arbitrary predicate $\Phi : A \to \mathsf{iProp}$ that satisfies $H_\Phi : \Box(\forall y.\, F\,\Phi\,y \mathrel{-\!\!*} \Phi\,y)$, we should prove $\Phi\,x$. We use $\Box$-DUP to duplicate H_Φ and apply it to our goal, which means we should prove $F\,\Phi\,x$. By monotonicity of F and the assumption H_F, it suffices to prove $\Box(\forall x.\, \mu\,F\,x \mathrel{-\!\!*} \Phi\,x)$. As H_Φ is persistent, we can use $\Box$-INTRO to introduce the $\Box$ in our goal. So assume $H_\mu : \mu\,F\,x$, we should prove $\Phi\,x$. This follows by unfolding μ in H_μ and our copy of H_Φ.
2. Proof of $\mu\,F\,x \mathrel{-\!\!*} F\,(\mu\,F)\,x$. Assume $H_\mu : \mu\,F\,x$. We specialize H_μ with $\Phi = F\,(\mu\,F)$, and apply it to our goal. It now remains to prove $\Box(\forall y.\, F\,(F\,(\mu\,F))\,y \mathrel{-\!\!*} F\,(\mu\,F)\,y)$. This goal follows from monotonicity of F and Item 1 (*i.e.*, $F\,(\mu\,F)\,x \mathrel{-\!\!*} \mu\,F\,x$).
3. The iteration principle follows immediately by definition of $\mu\,F$. ◀

The iteration principle from Theorem 1 is often inconvenient in practice. It only gives the induction hypothesis, but forgets that the fixpoint predicate holds for the recursive occurrence. From the iteration principle, we derive the following *induction principle*:

$$\Box(\forall x.\, F\,(\lambda y.\, \Phi\,y \wedge \mu\,F\,y)\,x \mathrel{-\!\!*} \Phi\,x) \mathrel{-\!\!*} \forall x.\, \mu\,F\,x \mathrel{-\!\!*} \Phi\,x$$

To obtain an intuition for the iteration and induction principle, it useful to specialize them to the isDelList predicate. After expanding $F_{\mathsf{isDelList}}$ and some logical simplification we get:

$$\frac{\begin{array}{lll}\Box(\forall \ell. & \ell \mapsto \mathtt{NIL} & \mathrel{-\!\!*} \Phi\;\ell\;[\,]) * \\ \Box(\forall \ell, \ell', w, \vec{w}.\; \ell \mapsto \mathtt{CONS}\,(w,\,\ell') & \mathrel{-\!\!*} (\Phi\,\ell'\,\vec{w} \wedge \mathsf{isDelList}\;\ell'\;\vec{w}) & \mathrel{-\!\!*} \Phi\,\ell\,(w :: \vec{w})) * \\ \Box(\forall \ell, \ell', \vec{w}. & \ell \mapsto \mathtt{DEL}\;\ell' & \mathrel{-\!\!*} (\Phi\,\ell'\,\vec{w} \wedge \mathsf{isDelList}\;\ell'\;\vec{w}) \qquad \mathrel{-\!\!*} \Phi\,\ell\,\vec{w})\end{array}}{\forall \ell, \vec{v}.\; \mathsf{isDelList}\;\ell\;\vec{v} \mathrel{-\!\!*} \Phi\,\ell\,\vec{v}}$$

The part in red is only present in the induction principle. We need a conjunction ($\wedge$) instead of separating conjunction ($*$) because the latter would be unsound if $\Phi\,\ell'\,\vec{w}$ and $\mathsf{isDelList}\;\ell'\;\vec{w}$ describe overlapping resources (take $\Phi\,\ell\,\vec{v} \triangleq \mathsf{isDelList}\;\ell\;\vec{v} * \vec{v} = [\,]$). (Unlike standard inductive types in Rocq, there is no dependent eliminator/induction principle as `iProp` is irrelevant.)

One can dually encode coinductive predicates using the greatest fixpoint operator $\nu\,F \triangleq \lambda x.\, \exists(\Phi : A \to \mathsf{iProp}).\, \Box(\forall y.\, \Phi\,y \mathrel{-\!\!*} F\,\Phi\,y) * \Phi\,x$. This definition is analogue to the standard encoding in second-order logic with $*$ and $\Box$ instead of $\wedge$. The greatest fixpoint is used in Iris for logical atomicity [38], and a nested least/greatest fixpoint is used for termination-preserving program refinements [28]. We focus on least fixpoints, but believe it is easy to generalize our Rocq-Elpi prototype to generate greatest fixpoints too (see § 7).

3 Data Structure Verification

In this section we show how least fixpoints are used to define representation predicates for data structure verification. We focus on examples where the in-memory and mathematical structure do not correspond, and structural recursion cannot be used, or is too cumbersome.

Consider a variant of the list predicate from the induction with 'deleted' nodes:

```
Iris Inductive is_list_with_tl (tl : loc) : loc → list val → iProp :=
  | is_list_with_tl_nil : tl ↦ NIL -∗ is_list_with_tl tl tl []
  | is_list_with_tl_cons v vs l l' :
      l ↦ CONS (v,#l') -∗ is_list_with_tl tl l' vs -∗ is_list_with_tl tl l (v :: vs)
  | is_list_with_tl_del vs l l' :
      l ↦ DEL #l' -∗ is_list_with_tl tl l' vs -∗ is_list_with_tl tl l vs.
```

Similar to the example in the introduction, this representation predicate cannot be defined by structural recursion because the size of the list `vs` does not decrease in `is_list_with_tl_del`. Compared to the example from the introduction, we add a parameter `tl` for the tail pointer. This parameter is necessary if we want to verify a constant-time 'push back' operation, which inserts an element at the end using the tail pointer, instead of by traversing the list. We define $\mathsf{isListWithTl}\ tl\ \ell\ \vec{v} \triangleq \mu\,(F_{\mathsf{isListWithTl}}\ tl)\ (\ell, \vec{v})$, with the following pre-fixpoint function:

$$\begin{aligned} F_{\mathsf{isListWithTl}}\ tl\ rec\ (\ell, \vec{v}) \triangleq\ & (\ell \mapsto \mathtt{NIL} * \vec{v} = [\,] * tl = \ell) \lor \\ & (\exists \ell'.\ \ell \mapsto \mathtt{DEL}\ \ell' * rec\,(\ell', \vec{v})) \lor \\ & (\exists \ell', w, \vec{w}.\ \ell \mapsto \mathtt{CONS}\,(w,\ \ell') * rec\,(\ell', \vec{w}) * \vec{v} = w :: \vec{w}) \end{aligned}$$

We derive the constructors from the fixpoint equations. The induction principle is:

$$\frac{\begin{array}{llll} \mathsf{isListWithTl}\text{-}\textsc{ind} & & & \\ \square(& tl \mapsto \mathtt{NIL} & & \mathrel{-\!\!*}\ \Phi\ tl\ [\,]) * \\ \square(\forall \ell, \ell', \vec{v}. & \ell \mapsto \mathtt{DEL}\ \ell' & \mathrel{-\!\!*}\ (\Phi\,\ell'\,\vec{v} \land \mathsf{isListWithTl}\ tl\ \ell'\ \vec{v}) & \mathrel{-\!\!*}\ \Phi\,\ell\,\vec{v}) * \\ \square(\forall \ell, \ell', w, \vec{w}.\ \ell \mapsto \mathtt{CONS}\,(w,\ \ell') & & \mathrel{-\!\!*}\ (\Phi\,\ell'\,\vec{w} \land \mathsf{isListWithTl}\ tl\ \ell'\ \vec{w}) & \mathrel{-\!\!*}\ \Phi\,\ell\,(w :: \vec{w})) \end{array}}{\forall \ell, \vec{v}.\ \mathsf{isListWithTl}\ tl\ \ell\ \vec{v} \mathrel{-\!\!*} \Phi\,\ell\,\vec{v}}$$

Like Rocq's native inductive types, the induction predicate Φ does not take the parameters (here, the tail pointer) as its argument because these remain constant during the induction. That is, we have $\Phi : \mathsf{loc} \to \mathsf{list\ val} \to \mathsf{iProp}$ instead of $\Phi : \mathsf{loc} \to \mathsf{loc} \to \mathsf{list\ val} \to \mathsf{iProp}$.

Consider a representation predicate for sets implemented using binary search trees (here, ⌜ ϕ ⌝ is the embedding of a Rocq proposition ϕ : `Prop` in Iris):

```
Iris Inductive is_search_tree : loc → gset Z → iProp :=
  | is_search_tree_empty l :
     l ↦ LEAF -∗ is_search_tree l ∅
  | is_search_tree_node l n ll lr Xl Xr :
     l ↦ NODE (#n, #ll, #lr) -∗ is_search_tree ll Xl -∗ is_search_tree lr Xr -∗
     ⌜ set_Forall (λ n', n' < n) Xl ⌝ -∗ ⌜ set_Forall (λ n', n < n') Xr ⌝ -∗
     is_search_tree l ({[ n ]} ∪ Xl ∪ Xr)
```

We use the type of finite sets `gset A` with elements of type `A` from the Rocq-std++ library [73]. This predicate could in principle be defined by structural recursion on the size of the set, but doing so would be cumbersome. Rocq's termination checker does not know that `Xl` and `Xr` are structurally smaller than `{[ n ]} ∪ Xl ∪ Xr`, so one needs to work around that, *e.g.,* by adding the size as an explicit index or through well-founded recursion. Defining the predicate as a least fixpoint requires no such workarounds. We simply create a pre-fixpoint function $F_{\mathsf{isSearchTree}}$ following the same pattern as before. It is worth noting that to prove monotonicity of $F_{\mathsf{isSearchTree}}$, the $\square$ modality in the definition of monotonicity is crucial due to the two recursive occurrences of `is_search_tree` in `is_search_tree_node`.

Parameters of `Iris Inductive` predicates are not restricted to first-order data. Due to the higher-order nature of Rocq and Iris, they can even be types or predicates themselves. For example, consider a higher-order representation predicate [15] with 'deleted' nodes:

```
Iris Inductive is_ho_list {A} (Φ : val → A → iProp) : loc → list A → iProp :=
  | is_ho_list_nil l : l ↦ NIL -∗ is_ho_list Φ l []
  | is_ho_list_cons v x xs l l' :
     l ↦ CONS (v,#l') -∗ Φ v x -∗ is_ho_list Φ l' xs -∗ is_ho_list Φ l (x :: xs)
  | is_ho_list_del xs l l' :
     l ↦ DEL #l' -∗ is_ho_list Φ l' xs -∗ is_ho_list Φ l xs.
Definition is_ho_list_loc {A} (Φ : loc → A → iProp) : loc → list A → iProp :=
  is_ho_list (λ v x, ∃ l : loc, ⌜ v = #l ⌝ ∗ Φ l x).
```

$$\Phi\, v \vdash \mathsf{wp}\, v\, [\Phi] \qquad \text{(TWP-VAL)}$$
$$\mathsf{True} \vdash \mathsf{wp}\, (\mathtt{ref}\ v)\, [\ell.\, \ell \mapsto v] \qquad \text{(TWP-ALLOC)}$$
$$\ell \mapsto v \vdash \mathsf{wp}\, !\ell\, [w.\, w = v * \ell \mapsto v] \qquad \text{(TWP-LOAD)}$$
$$\ell \mapsto v \vdash \mathsf{wp}\, (\ell \leftarrow v')\, [w.\, w = () * \ell \mapsto v'] \qquad \text{(TWP-STORE)}$$
$$\mathsf{wp}\, e\, [v.\, \mathsf{wp}\, K[v]\, [\Phi]] \vdash \mathsf{wp}\, K[e]\, [\Phi] \qquad \text{(TWP-BIND)}$$
$$(\forall v.\, \Phi\, v \mathrel{-\!\!*} \Psi\, v) * \mathsf{wp}\, e\, [\Phi] \vdash \mathsf{wp}\, e\, [\Psi] \qquad \text{(TWP-WAND)}$$
$$\mathsf{wp}\, e[(\mathtt{rec}\ f\ x = e)/f][v/x]\, [\Phi] \vdash \mathsf{wp}\, (\mathtt{rec}\ f\ x = e)\, v\, [\Phi] \qquad \text{(TWP-REC)}$$

Figure 2 Proof rules for the total weakest precondition connective.

Higher-order representation predicate make it possible to specify nested data structures, *e.g.*, `is_ho_list_loc (is_ho_list (=)) : loc → list (list val) → iProp` expresses that a location contains a lists of lists. (The predicate `is_ho_list` is more general than `is_ho_list_loc` since it allows one to talk about lists with unboxed values.)

Least fixpoints provide support for predicates that are defined in a nested-recursive way. Let us consider a representation predicate for simple rose trees:

```
Inductive rose_tree := Node : list rose_tree → rose_tree.
Iris Inductive is_rose_tree : loc → rose_tree → iProp :=
  | is_tree_node l ts : is_ho_list_loc is_rose_tree l ts -* is_rose_tree l (Node ts).
```

The least fixpoint is well-defined because `is_ho_list` Φ is monotone in Φ. This monotonity property is proved by induction, and declared as a `Proper` instance allowing our monotonicity solver (§ 5.2) to use it in the monotonicity proof of $F_{\mathsf{isRoseTree}}$. We note that monotonicy is more general than the syntactic strict positivity condition for ordinary `Inductive` predicates in Rocq. If we prove that a predicate is monotone (and declare the corresponding `Proper` instance), we can use it in a nested recursive fashion to define other inductive predicates.

4 Total Program Correctness

In this section we present a program logic for total program correctness, defined internally in the Iris base logic using a least fixpoint. Before presenting the definition, we explain the proof rules for program specifications. We conclude by comparing to the standard (step-indexed) Iris program logic for partial correctness.

Total weakest preconditions and its proof rules At the heart of our program logic we have the *total weakest precondition* connective $\mathsf{wp}\, e\, [\Phi]$. (We use square $[\Phi]$ and curly $\{\Phi\}$ brackets to distinguish total and partial correctness weakest preconditions and Hoare triples, respectively.) Given a postcondition $\Phi : \mathsf{val} \to \mathsf{iProp}$, the connective $\mathsf{wp}\, e\, [\Phi] : \mathsf{iProp}$ gives the weakest precondition under which all executions of e are *terminating* and *safe*, and all return values v satisfy $\Phi\, v$. A program execution is safe if it does not get stuck in the operational semantics, which particularly means that operators are never applied to wrong operands (*e.g.*, $3 + \mathtt{true}$) and no invalid memory accesses take place. Hoare triples are defined as $[P]\, e\, [\Phi] \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, [\Phi])$. The magic wand ($-\!\!*$) encodes the precondition P, and the $\Box$ modality ensures that triples can be used multiple times.

The proof rules are given in Figure 2. Rule TWP-VAL says that it suffices to prove the postcondition if a program is already a value. Rules TWP-ALLOC, TWP-LOAD and TWP-STORE

are the quintessential reasoning rules for mutable references. Rule TWP-BIND is used to reason about an expression nested inside a (call-by-value) evaluation context K. Rule TWP-WAND is used to weaken the postcondition, as well as to frame away parts of the precondition.

What might be surprising is the fact that the rule TWP-REC for recursive functions talks about a single unfolding, and lacks a loop variant/measure to reason about recursive calls. Similar to CFML [14] we can reason about recursive programs through induction at the meta-level (using Rocq's `induction`). But there is another option—we can perform induction on an inductive predicate defined as a least fixpoint in separation logic. Consider $[\,\mathsf{isListWithTl}\ tl\ \ell\ \vec{v} * i < \mathsf{length}\ \vec{v}\,]\ \texttt{lookup}\ \ell\ i\ [\,w.\, w = \vec{v}_i * \mathsf{isListWithTl}\ tl\ \ell\ \vec{v}\,]$. The function $\texttt{lookup}\ \ell\ i$ traverses the list and returns the value at index i, skipping 'deleted' nodes. It is futile to perform (meta-level) induction on i or $\vec{v}$ since there might be an arbitrary number of 'deleted' nodes that should be skipped. Instead we perform induction on $\mathsf{isListWithTl}\ tl\ \ell\ \vec{v}$. Note that we truly need induction instead of iteration. When making a recursive call, we use the induction hypothesis (first conjunct in isListWithTl-IND), and when we reach the desired value, we use $\mathsf{isListWithTl}\ tl\ \ell'\ \vec{v}'$ (second conjunct) to prove the postcondition.

Definition of the total weakest precondition Similar to the standard Iris weakest precondition connective, we define our total weakest precondition connective as a derived form in the Iris base logic (for brevity's sake, we omit concurrency and Iris's invariant masks):

$$\mathsf{wp}\ e\ [\,\Phi\,] \triangleq \begin{cases} \Rrightarrow \Phi\, e & \text{if } e \in \mathsf{val} \\ \forall h.\, \mathcal{S}\, h \mathrel{-\!\!*} \Rrightarrow \mathsf{red}(e,h) * & \text{if } e \notin \mathsf{val} \\ \qquad \forall e_2, h_2.\, \big((e,h) \to (e_2,h_2)\big) \mathrel{-\!\!*} \Rrightarrow \mathcal{S}\, h_2 * \mathsf{wp}\ e_2\ [\,\Phi\,] \end{cases}$$

This definition has a solution as a least fixpoint because the recursive occurrence appears in positive position. This definition is quite a mouthful, so let us at first ignore Iris's update modality ($\Rrightarrow$) and state interpretation ($\mathcal{S}$). In the base case ($e \in \mathsf{val}$), the definition simply requires the postcondition Φ to hold. The inductive case ($e \notin \mathsf{val}$) has a safety and preservation part. The safety part requires that e is not stuck ($\mathsf{red}(e,h) \triangleq \exists e', h'.\, (e,h) \to (e',h')$). The preservation part requires that for any step $(e,h) \to (e_2,h_2)$ in the operational semantics, the weakest precondition holds recursively for e_2. The least fixpoint guarantees that derivations of $\mathsf{wp}\ e\ [\,\Phi\,]$ are finite, and therefore that e terminates—without explicit step counting. The state interpretation $\mathcal{S}\, h$ [37, §7.4] links the heap h used in the operational semantics to the points-to assertion $\ell \mapsto v$. The update modality ($\Rrightarrow$) is used to support Iris's ghost state.

The proof rules from Figure 2 are proved as lemmas in separation logic. For TWP-BIND and TWP-WAND we perform induction on the least fixpoint in the premise. To ensure that the total weakest precondition is doing its job, we prove *adequacy*, which given a closed proof (*i.e.,* precondition True), allows us to obtain strong normalization at the meta-level.

▶ **Theorem 2** (Adequacy). *If* $[\,\mathsf{True}\,]\, e\, [\,\Phi\,]$, *then* (e,h) *is strongly normalizing for any heap* h.

Comparison to the partial weakest precondition Our total weakest precondition is almost identical to the Iris weakest precondition for partial correctness (written $\{\Phi\}$ instead of $[\,\Phi\,]$):

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \begin{cases} \Rrightarrow \Phi\, e & \text{if } e \in \mathsf{val} \\ \forall h.\, \mathcal{S}\, h \mathrel{-\!\!*} \Rrightarrow \mathsf{red}(e,h) * & \text{if } e \notin \mathsf{val} \\ \qquad \triangleright\, \forall e_2, h_2.\, \big((e,h) \to (e_2,h_2)\big) \mathrel{-\!\!*} \Rrightarrow \mathcal{S}\, h_2 * \mathsf{wp}\ e_2\ \{\Phi\} \end{cases}$$

The sole difference is the later modality ($\triangleright$) [51] in the recursive case (marked in red), which is part of the support for *step-indexing* [4, 2, 5] by the Iris base logic. While the presence of

the modality might appear innocent, it has major consequences. To understand that, we should explain the semantics of step-indexing. The semantics of Iris propositions is indexed by a natural number n, called the *step-index*. The proposition $\triangleright P$ is defined to hold at step n iff P holds at step $n-1$, and $\triangleright P$ holds vacuously at step 0. Since each recursive occurrence of the weakest precondition is below a later modality, this means that $\mathsf{wp}\, e\, \{\Phi\}$ holds at step-index n if the program is safe for n steps of computation. In other words, the presence of the later modality brings us to the realm of partial program correctness.

Now let us review the consequences to the proof rules. Since the weakest precondition connective contains a later modality, that modality also shows up in the proof rules for program operations that perform a computation step, *e.g.,* recursive functions calls:

WP-REC
$$\triangleright\, (\mathsf{wp}\, e[(\texttt{rec}\ f\ x = e)/f][v/x]\, \{\Phi\}) \vdash \mathsf{wp}\, (\texttt{rec}\ f\ x = e)\, v\, \{\Phi\}$$

$\triangleright$-INTRO
$$P \vdash \triangleright P$$

LÖB
$$(\triangleright P \Rightarrow P) \vdash P$$

The $\triangleright$ modality makes WP-REC stronger since we can always introduce the $\triangleright$ using $\triangleright$-INTRO. It also means we can trivially verify looping programs, *e.g.,* we can prove $\mathsf{wp}\, \texttt{diverge}\, ()\, \{\mathsf{False}\}$ with $\texttt{diverge} \triangleq \texttt{rec}\ f\ x = f\ x$. This is done using LÖB induction, which corresponds to induction on the step-index, *i.e.,* the number of remaining computation steps.

Another difference is that Iris defines its partial weakest precondition connective using Banach's fixpoint [37, §5.6], which is a step-indexed fixpoint operator that has no restrictions on positivity, but requires recursion to be guarded by a later modality. Since guarded fixpoints are unique [37, Thm 4] and the definition of the weakest precondition does not involve a negative occurrence, one could use the least and Banach fixpoint interchangeably to define the partial weakest precondition. Based on that observation, Vistrup *et al.* [79] extended our total weakest precondition connective with a parameter that determines whether a later modality is emitted, and thereby provide a uniform treatment of partial and total correctness.

Invariants and concurrency Like the Iris weakest precondition, we extend our total weakest precondition with Iris's mask-changing update modality and a threadpool semantics to support invariants and concurrency, but these extensions come with some limitations.

First, since no later modality is emitted for computation steps, the opening of invariants $\boxed{P}$ is limited to *timeless propositions* [37, §5.7], *i.e.,* propositions P that do *not* include nested invariants or weakest preconditions. This restriction is needed for adequacy. If we allow unrestricted Iris invariants (that can be opened without a later modality), we could prove a total Hoare triple for Landin's Knot [47], which is clearly not terminating. That is, given $\texttt{landin} \triangleq \texttt{let}\ r = \texttt{ref}\ (\lambda x.\, x)\ \texttt{in}\ r \leftarrow (\lambda x.\, !\, r\, ());\, !\, r\, ()$, we could prove $[\,\mathsf{True}\,]\, \texttt{landin}\, [\,\mathsf{True}\,]$ using the invariant $\boxed{\exists f.\, r \mapsto f * \Box\, \mathsf{wp}\, f\, ()\, [\,\mathsf{True}\,]}$.

Second, our total weakest precondition enforces a very strong notion of termination in the context of concurrency: $\mathsf{wp}\, e\, [\,\Phi\,]$ says that e terminates for *any* scheduler. This means that we inherently cannot verify blocking programs. For example, $\texttt{fork}\ \{\texttt{while}\, (!\, r = 0);\}\, ; r \leftarrow 1$ does not terminate if the scheduler only picks the forked-off thread. Program logics for total correctness of concurrency thus integrate an assumption of fair scheduling [68, 48, 24, 75].

Despite these limitations, we should emphasize that our logic is at least as strong as 'traditional' separation logic for total correctness of sequential programs, *e.g.,* CFML [14, 15, 16]. That means, we have all the proof rules of sequential separation logic, but additionally support a limited form of ghost state, invariant assertions and concurrency. Additionally, we obtain a smooth integration between partial and total correctness: we have $\mathsf{wp}\, e\, [\,\Phi\,] \vdash \mathsf{wp}\, e\, \{\Phi\}$, *i.e.,* total correctness implies partial correctness. It is an open question how to define an fully-fledged Iris-like logic for total correctness (see § 6).

5 Prototype Command and Tactic in Rocq-Elpi

In this section we discuss our Rocq-Elpi prototype for transforming high-level `Iris Inductive` specifications into an encoding using a least fixpoint. We show how to generate the pre-fixpoint function and least fixpoint definition (§ 5.1), specify and prove monotonicity of variadic (pre-fixpoint) functions (§ 5.2), port the IPM tactics to Rocq-Elpi as needed to generate the constructors and induction principles (§ 5.3), and our `iInduction` tactic (§ 5.4). We conclude with a short evaluation of our implementation (§ 5.5).

5.1 Generating the Fixpoint

Inductive specifications are at the core of Rocq, and their user-facing syntax is well known among all Rocq users. As shown in § 3, our `Iris Inductive` command takes advantage of that very same syntax, allowing the user to declare inductive predicates in separation logic similarly to native inductive predicates in Rocq. The keyword `Iris` is the invocation of the Rocq-Elpi command, while its argument spans from the `Inductive` keyword to the end of the text. Taken alone, the argument of `Iris` is a syntactically valid inductive specification, but it would be immediately rejected by Rocq because the types of the constructors are expected to be implications in the Rocq meta logic and not magic wands in Iris.

Rocq-Elpi commands can receive arguments as raw syntax trees, *i.e.,* after Rocq has parsed them, unfolded notations and performed lexical analysis to identify bound variables, and resolved free variables to (existing) global identifiers. A Rocq-Elpi command can elaborate these syntax trees into proper declarations and submit them to the Rocq type checker for validation. For example, the inductive specification `is_list_with_tl` from § 3 is transformed into the following pre-fixpoint function:

```
Definition is_list_with_tl_pre (tl : loc) :=                    (* parameters *)
  λ (rec : loc → list val → iProp),                               (* fixpoint *)
    λ (l : loc) (vs : list val),                                (* fixpoint arguments *)
      tl ↦ NIL ∗ ⌜vs = []⌝ ∗ ⌜l = tl⌝                                    (* nil *)
    ∨ (∃ (v : val) (vs' : list val) (l' : loc),                          (* cons *)
         l ↦ CONS (v, #l') ∗ rec l' vs ∗ ⌜vs = v :: vs'⌝)
    ∨ ∃ (l' : loc), l ↦ DEL #l' ∗ rec l' vs.                            (* del *)
```

This definition closely follows the definition of $F_{\mathsf{isListWithTl}}$ in § 3, but is in curried form. Note that the parameter `tl` appears before the `rec` argument, similar to $F_{\mathsf{isListWithTl}}$.

This definition is generated by applying a number of transformation steps. **1.** In each inductive constructor type we replace the top-level wands with separating conjunctions, and we transform the binders of the constructors into existential quantifiers in Iris. **2.** We combine all constructors with disjunctions. **3.** We wrap the entire disjunction into a function that takes the parameters (`tl`), the recursive argument (`rec`) and the indices (`l` and `vs`). **4.** We replace all recursive occurrences of `is_list_with_tl` with `rec`. **5.** We turn the right-most use of `rec` into a series of equalities on its argument. Note that the transformations do not necessarily create sensible intermediary terms. Particularly, the meaning of the recursive occurrence in Steps 1–4 is context dependent. The right-most recursive occurrence is a placeholder for the equalities on the variables, which are only generated in Step 5. In that step, we take good care to minimize the number of existential quantifiers and equalities by substituting trivial equalities instead of adding them. Consider the constructor `is_list_with_tl_del`, for which we would naively generate `∃ vs' l'' l', l'' ↦ DEL #l' ∗ rec l' vs' ∗ ⌜ l = l'' ⌝ ∗ ⌜ vs = vs' ⌝`. In Step 5 we do not add the quantifiers `l''` and `vs'`, and substitute the trivial equalities.

The implementation of these elaboration steps takes great advantage of Elpi's automatic handling of binders using HOAS [57] and the natural support for open recursion provided by its rule-based nature. In particular term replacement, as in Step 5, can be easily implemented atop of a pre-defined (deep) copy function by adding, at run-time, ad-hoc rules for the terms to be replaced. For example, to turn `UglyTerm` into `NiceTerm` by replacing all occurrences of `l''` with `l`, one can just write "`copy {{ l'' }} {{ l }} => copy UglyTerm NiceTerm`". Last, Rocq-Elpi allows us to generate incomplete syntax trees, *i.e.,* containing Rocq's placeholders for implicit arguments, as long as the missing information can be reconstructed by the Rocq elaborator, which we call at judicious moments.

The next step is to define the inductive predicate by constructing the least fixpoint of the pre-fixpoint function. In § 2 we used the least fixpoint operator μF, which takes a *unary* pre-fixpoint function $F : (A \to \mathsf{iProp}) \to (A \to \mathsf{iProp})$. To avoid (un)currying, we generate a specialized version for the given arity in our Rocq-Elpi implementation. For instance:

```
Definition is_list_with_tl (tl l : loc) (vs : list val) :=
  ∀ Φ : loc → list val → iProp,
    □ (∀ l' vs', is_list_with_tl_pre tl Φ l' vs' -∗ Φ l' vs') -∗ Φ l vs.
```

The generated definition closely follows the unary version in § 2, but expands the predicate Φ to the given arity. Generalizing specialized versions has another benefit. For technical reasons due to step-indexing, Iris requires higher-order predicates, *i.e.,* predicates with predicates as their arguments (such as our total weakest precondition), to be *morphisms* in the category of OFEs [37, §4.2], denoted `-n>` in Rocq. Our prototype recognizes these morphisms, *e.g.,*

```
Iris Inductive twp : expr → (val -d> iProp Σ) -n> iProp Σ := ..
```

Here, it generates a fixpoint definition in which the quantified predicate is also a morphism:

```
Definition twp (e : expr) (Ψ : val → iProp Σ) : iProp Σ :=
  ∀ Φ : expr → (val -d> iProp Σ) -n> iProp Σ,
    □ (∀ e' Ψ', twp_pre Φ e' Ψ' -∗ Φ e' Ψ') -∗ Φ e Ψ.
```

5.2 Variadic Monotonicity

The fixpoint μF only satisfies the desired properties (fixpoint equations and iteration/induction) if the pre-fixpoint function F is monotone. To express that a variadic (uncurried) pre-fixpoint function is monotone, and to enable a goal-directed proof search, we develop a variadic notion of monotonicity by porting the notion of *signatures* from Rocq's generalized rewriting framework [63] to separation logic. To each function $f : A$ we assign a signature $S_A : A \to A \to \mathsf{iProp}$ (using iProp instead of Prop compared to Rocq's generalized rewriting). A function f is monotone iff $S_A\, f\, f$, *i.e.,* if f is a *proper element* of S_A, denoted $\texttt{Proper}\, S_A\, f$. The signature S_A of $f : A$ is defined using combinators that follow the structure of A:[2]

$$S_A \Longrightarrow S_B \triangleq \lambda f\, g : A \to B.\, \forall x, y.\, S_A\, x\, y \twoheadrightarrow S_B\, (f\, x)\, (g\, y) \qquad (=)_A \triangleq \lambda x\, y : A.\, x = y$$

$$\square\, S_A \triangleq \lambda x\, y : A.\, \square\, (S_A\, x\, y) \qquad \mathsf{flip}\, S_A \triangleq \lambda x\, y : A.\, S_A\, y\, x$$

$$(\twoheadrightarrow) \triangleq \lambda P\, Q : \mathsf{iProp}.\, P \twoheadrightarrow Q$$

Some example signatures for the separation logic connectives are as follows:

[2] Rocq's generalized rewriting framework and our actual Rocq implementation feature the signature $\mathsf{pointwise}_A\, S_B \triangleq \lambda f\, g : A \to B.\, \forall x.\, S_B\, (f\, x)\, (g\, x)$, which is logically equivalent to $(=)_A \Longrightarrow S_B$, and is often used to obtain simpler proofs.

	Type	Signature
$*$	$\mathsf{iProp} \to \mathsf{iProp} \to \mathsf{iProp}$	$(-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*)$
$-\!\!*$	$\mathsf{iProp} \to \mathsf{iProp} \to \mathsf{iProp}$	$\mathsf{flip}\,(-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*)$
$\Box$	$\mathsf{iProp} \to \mathsf{iProp}$	$\Box\,(-\!\!*) \Longrightarrow (-\!\!*)$
$\exists$	$(A \to \mathsf{iProp}) \to \mathsf{iProp}$	$\big((=)_A \Longrightarrow (-\!\!*)\big) \Longrightarrow (-\!\!*)$

We use flip to express that $-\!\!*$ is antitone in its first argument. To understand the signature of the existential, it is useful to expand the combinators, then we see that $\exists$ is a proper element iff $\forall \Phi\, \Psi.\, (\forall x\, y.\, x = y \to \Phi\, x -\!\!* \Psi\, y) -\!\!* (\exists x.\, \Phi\, x) -\!\!* (\exists x.\, \Psi\, x)$. A variadic pre-fixpoint function of type $(A_1 \to \cdots \to A_n \to \mathsf{iProp}) \to A_1 \to \cdots \to A_n \to \mathsf{iProp}$ has signature: $\Box\big((=)_{A_1} \Longrightarrow \cdots \Longrightarrow (=)_{A_n} \Longrightarrow (-\!\!*)\big) \Longrightarrow (=)_{A_1} \Longrightarrow \cdots \Longrightarrow (=)_{A_n} \Longrightarrow (-\!\!*)$. Similar to Rocq's generalized rewriting framework, we use type classes [64] to register proper elements, which makes it possible for users to add custom monotonicity rules.[3]

Once we have assigned a signature S_A to a pre-fixpoint function $f : A$ we should construct a proof of $\texttt{Proper}\, S_A\, f$. We construct this proof in a goal-directed fashion by traversing the syntax tree of f. For each node $g : B$ in the syntax tree of f, we identify the appropriate signature S_B and apply the proof of $\texttt{Proper}\, S_B\, g$. Suppose we have to solve $\Box(\cdots) \vdash (\cdots * \cdots) -\!\!* (\cdots * \cdots)$. We use type classes to search for a signature whose rightmost symbol is $-\!\!*$, and which has $*$ as its proper element, and find $\texttt{Proper}\,((-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*))\,(*)$.

The strategy of the proof search consists of two steps: normalization and application. We start with normalization, which introduces universal quantifiers, magic wands and modalities. We then perform an application, which tries in given order: **1.** If the left- and right-hand side of the top-level relation are equal, we are done. **2.** If the goal follows from a hypothesis (in the IPM context), we are done. **3.** Otherwise, search for an applicable signature and proper proof and apply it. Perform normalization and continue until all subgoals are closed.

5.3 Reimplementing IPM tactics in Rocq-Elpi

The monotonicity properties, as well as the constructors and induction principles, are lemmas in separation logic (iProp) instead of lemmas in the Rocq meta logic (Prop). The (automatic) construction of proofs in separation logic is much harder than its counterpart in plain Rocq. For example, proving an implication $\phi \Rightarrow \psi$ with $\phi, \psi : \mathsf{Prop}$ is easy—we simply introduce ϕ into the Rocq context and construct a proof of ψ. In Rocq-Elpi this means we generate the proof term $\lambda i : \phi.\, \textit{proof-of-}\psi$, where we construct $\textit{proof-of-}\psi$ with $i : \phi$ in its context.

Constructing a proof of $P -\!\!* Q$ with $P, Q : \mathsf{iProp}$ is much more challenging. Since P is a separation logic proposition, we cannot introduce it into the Rocq context, and likewise a λ is not a proof term of $P -\!\!* Q$. Instead we need to use the proof rules of separation logic from Figure 1, such as $-\!\!*$-INTRO. Using these proof rules directly is tedious as we have to manipulate an entailment $P \vdash Q$ where the *context* P has an arbitrary structure.

To make it feasible to construct proofs in separation logic, the Iris Proof Mode (IPM) represents a separation logic goal as a structured judgment $\Pi;\, \Sigma \Vdash Q \triangleq \Box(\bigwedge \Pi) \wedge (\mathbin{\ast\!\!\ast} \Sigma) \vdash Q$, where Π is the *persistent context* and Σ is the *spatial context*. Contexts are association lists, mapping hypothesis names to separation logic propositions. IPM provides notations so that separation logic goals are rendered in a human-readable way, and provides `i`-variants of most Rocq tactics, *e.g.*, `iIntros`, `iDestruct` and `iExists`. At their core, these tactics are derived rules for the $\Vdash$ judgment, represented as lemmas in Rocq. For instance, for `iIntros` we have:

[3] We use type classes instead of an Elpi database because they allow users to define the property and hint using a single `Instance` command. There is ongoing work to unify type classes and Elpi databases [26].

$$\frac{\Pi;\ \Sigma, (i : P) \Vdash Q \qquad i \notin \Pi; \Sigma}{\Pi;\ \Sigma \Vdash (P \mathrel{-\!\!*} Q)}$$

```
Lemma tac_wand_intro Δ i P Q R :
  FromWand R P Q → (* means R ⊢ P -∗ Q *)
  match envs_app false (Esnoc Enil i P) Δ with
  | None => False
  | Some Δ' => envs_entails Δ' Q
  end →
  envs_entails Δ R.
```

We let $\Delta = \Pi;\ \Sigma$ in the lemma. The lemma is derived from $-\!\!*$-INTRO and rules to reassociate $\wedge$, $*$, and $\Box$. The IPM tactic `iIntros` is an Ltac1 wrapper that applies `tac_wand_intro` and performs some processing to reduce the goal and generate error messages. (Similar to Rocq's `intros`, the full `iIntros` and our Rocq-Elpi port support an *introduction pattern* instead of a single identifier as its argument. We omit details about introduction patterns.)

(Interactive) proofs in IPM are constructed by chaining tactics written in Ltac1, but this works poorly when generating IPM proofs using Rocq-Elpi because the Ltac1 bridge is brittle.[4] Instead, we reimplement the Ltac wrappers of many IPM tactics in Rocq-Elpi to enable convenient chaining of IPM tactics in Rocq-Elpi functions.

To enable the incremental construction of proofs, Rocq terms contain typed holes that represent the open goals. A tactic makes progress by *refining* a hole in a term, and when it generates new holes, these correspond to new subgoals. Holes are a primitive notion in Rocq-Elpi. The Rocq-Elpi runtime manages their eventual assignment and compatibility with the hole type, and thus relieves the tactic programmer of this burden. Moreover, Rocq-Elpi provides quotations and anti-quotations to build terms using Rocq's syntax: A Rocq term is enclosed in double curly braces, while `lp:` allows escaping back to Elpi.

The Elpi type `igoal` of *Iris goals* bundles a goal `envs_entails Δ R` with a proof term. Rocq-Elpi tactics are (logic programming) rules that take an Iris goal (whose proof term is a hole) as input, and produce new Iris subgoal(s) as output. The tactic `eiIntro-ident ID GOAL SUBGOAL` for wand introduction takes a `GOAL` of type `envs_entails Δ (P -∗ Q)`, and generates a `SUBGOAL` of type `envs_entails Δ' Q`, where `P` has been inserted with name `ID` in `Δ'`:

```
pred eiIntro-ident i:ident, i:igoal, o:igoal.  % takes an id and a goal to a subgoal
eiIntro-ident ID GOAL (igoal IType IProof) :-
  ident->term ID T,                            % data conversion
  (@no-tc! ==>                                 % refine H disabling TC resolution
    refine-igoal-with {{ tac_wand_intro _ lp:T _ _ _ lp:FromWand lp:IProof }} GOAL),
  tc-solve-term FromWand, !,                   % run TC resolution on FromWand
  coq.typecheck IProof IType' ok,              % inspect subgoal
  pm-reduce IType' IType,                      % normalize subgoal
  std.assert! (not (IType = {{ False }})) "eiIntro: not fresh".
```

We convert the Elpi representation of the name `ID` into a Rocq representation. We then use the helper `refine-igoal-with` to refine the Iris goal `GOAL` with the lemma `tac_wand_intro`. (We disable the implicit type class resolution performed by unification through `@no-tc! ==>`, so we can control how `FromWand` is resolved ourselves.) We use the helper `pm-reduce` to reduce functions that manipulate IPM contexts such as `envs_app`. Finally, we use `std.assert!` to test that the new subgoal is not `False` (which happens if the name `ID` is not fresh).

With the reimplemention of IPM tactics at hand, the generation of a proof (*e.g.,* of the monotonicity property, constructors, or induction principle) boils down to chaining these

[4] Since Ltac tactics can change the Rocq context arbitrarily, this makes interfacing inherently hard. After execution of an Ltac1 tactic, the current Ltac1 bridge assumes the worst case and adds the entire Ltac context to the Elpi context. This makes chaining tactics inefficient and inconvenient.

tactics. Compared to Ltac, we explicitly pass around the goals and subgoals as arguments, instead of making use of tacticals to compose tactics.

5.4 The iInduction Tactic

Rocq's `induction` tactic receives a term (of an inductive type in Rocq), finds the corresponding induction principle and applies it. Our prototype `iInduction` tactic performs the same job for inductive predicates defined using `Iris Inductive`. To find the induction principle, we let the `Iris Inductive` command store information in an Elpi database, which can be queried by the `iInduction` tactic. An Elpi database is a collection of rules (in the logic programming sense) that can be programmatically extended or queried. Our database is declared as:

```
Elpi Db induction.db lp:{{
  pred inductive-ind o:gref, o:gref.
  (* more predicate signatures *)
}}.
```

The omitted predicates store additional information, such as the number of parameters and constructors. A rule `inductive-ind I P` associates the induction principle `P` (of the Elpi type `gref` representing global Rocq names) to the inductive predicate `I`. For example, by invoking the command `Iris Inductive is_list_with_tl` ... (§3), the following entry is added to the database (omitting fully qualified names):

```
inductive-ind (const "is_list_with_tl") (const "is_list_with_tl_ind").
```

The `iInduction` tactic applies the induction principle with the correct parameters. This information is obtained by querying the database. Subsequently, `iInduction` introduces the separating conjunctions/modalities/quantifiers to generate separate subgoals for each inductive case. This step uses a database query to obtain the number of constructors.

5.5 Evaluation

The total size of our `Iris Inductive` command and `iInduction` tactic consists of 345 lines of code (LOC) in Rocq and 1761 LOC in Elpi. Detailed line counts are as follows:

Category	LOC Rocq	LOC Elpi
Library with Elpi utilities	0	100
Generation of pre-fixpoint function and fixpoint definition (§5.1)	24	243
Reimplementation of IPM tactics (§5.3)	154	874
Signatures and `Proper` search (§5.2)	134	86
Generation of constructors and induction principles (§5.3)	0	411
`iInduction` tactic (§5.4)	33	47

We have not reimplemented all IPM tactics, only the relevant parts of the tactics needed to implement our command and tactic, *e.g.,* `iIntros`, `iModIntro`, `iClear`, `iPoseProof`, `iDestruct`, `iSpecialize` and `iApply`. Our reimplementation works for any Modal BI (MoBI) instead of just the Iris base logic, and supports Iris's introduction patterns, for which we reimplemented the parser and evaluator. (Since our reimplementation of IPM is partial, it is impossible to provide a meaningful comparison of the LOC between the Elpi and Ltac versions.)

To evaluate our prototype we reimplemented the total weakest precondition (originally defined through a manual least fixpoint encoding by the first author in 2017 [42]) using our `Iris Inductive` command. We used the constructors and `iInduction` tactic to derive the proof rules. Since the signature of the total weakest precondition and its proof rules are not changed, no changes are needed for Iris users who already use the total weakest precondition.

6 Related Work

The representation of (co)inductive data types and predicates received plenty of attention in the literature. The history is too rich to survey, but we see there are roughly two approaches: add (co)inductives as a primitive construct (as done in Rocq [22]) or encode them (as done in the HOL family [49, 31, 56, 33, 9, 76]). In this paper we use an encoding, but there are some key differences to prior work. First, we work in an *embedded* logic, instead of the native logic of the proof assistant. We thus cannot use the native tactics for introduction and elimination of the logical connectives and have to manage the proof context ourselves (with help of the Iris Proof Mode [45]). Second, we work in *separation* logic, which is a substructural logic with restrictions on the number of times each hypothesis can be used. We are not aware of a prior implementation of inductive types or predicates in a proof assistant based on substructural logic. Third, we only focus on predicates, not data types, so there is no notion of computation. For data types we rely on the machinery provided by Rocq.

The steps performed by our `Iris Inductive` command have some similarity with those performed by the automation of Harrison [33] for HOL. Particularly, Harrison already emphasizes the importance of a variadic notion of monotonicity, and allows users to add monotonicity rules. We achieve these goals through signatures [63] and Rocq's type classes [64].

In early versions of Rocq (v4.10) based on the Calculus of Constructions (CoC) [21], inductive types (and predicates) were represented using a second-order encoding [58]. Following the work on the Calculus of Inductive Constructions (CIC) [22, 55], inductive types were added as a primitive to remedy the lack of dependent induction principles and discrimination of constructors (*e.g.,* $0 \neq 1$ not being provable). Since we only consider inductive predicates and `iProp` is irrelevant, these issues of the second-order encoding are not applicable to us. Compared to native inductive predicates in Rocq, we use a semantic condition (extensible through `Proper` instances) instead of a syntactic check. We require mere positivity instead of strict positivity (but are not aware of real-life uses cases of the former).

As discussed throughout this paper, there are multiple ways of defining (co)inductive predicates in separation logic, which all have different conditions and thus support different classes of predicates. With structural recursion, the size of the recursive argument should be decreasing. With Banach's fixpoint, recursive occurrences should be below a later modality (*i.e.,* the pre-fixpoint function should be contractive). With least/greatest fixpoints, recursive occurrences should be positive (*i.e.,* the pre-fixpoint function should be monotone). Yet another approach has been used in the literature: if the type of separation logic propositions is 'simple enough', one can use the native `Inductive` command of the proof assistant. For instance, when using simple heap predicates, *i.e.,* `heapProp := heap → Prop`, Rocq accepts the definition `Inductive is_list : loc → list val → heapProp` because it unfolds `heapProp` and simply considers the predicate to have the heap as an additional index (formally, `heapProp` is an arity of sort `Prop` [72]). Up to our knowledge, this technique dates back to Appel [3] in 2006, who used it to define the list predicate. More recently, this technique has been used to define typing rules internally in separation logic, using Agda [60] and Rocq [34]. Without automation like our `Iris Inductive` command, this technique avoids a tedious manual encoding. However, it is not applicable to Iris, whose type of propositions `iProp` is a Σ-type that bundles a predicate with some properties (and is thus not an arity of sort `Prop`).

Other separation logics for total correctness have been developed. CFML [14, 15, 16] (in Rocq) uses standard heap predicates and defines Hoare triples/weakest preconditions in terms of a big-step operational semantics. Due to the use of a big-step operational semantics, non-determinism is not supported, but this is remedied by later work on Omnisemantics [17].

Compared to CFML, our logic for total correctness is defined in Iris, and therefore inherits a limited form of the Iris mechanisms for ghost state, invariant assertions and concurrency.

It is an open question how to define a fully-fledged Iris-like logic for total correctness. Time credits [6] (mechanized and extended by Mevel *et al.* [50] in Iris) provide a mechanism to prove *bounded termination* using separation logic. The logic is extended with a resource $\$n$, which gives the permission to perform n computation steps. The Hoare triple $\{\$n\}\, e\, \{\Phi\}$ means that e terminates in at most n steps. Compared to our total weakest preconditions, time credits require program specifications to mention explicit step counts. This makes it possible to establish (amortized) complexity bounds [18]. Spies *et al.* [65] observe that the explicit counting of steps limits compositionality of termination proofs, and thus generalize time credits $\$n$ from n being a natural number to an ordinal number. The trade-offs between transfinite time credits and our weakest preconditions remain to be investigated. We use vanilla Iris, whereas Spies *et al.* use Transfinite Iris, a variant of Iris with transfinite/ordinal step-indexing that violates some of Iris's commuting rules for the later modality [65, §7].

One can also prove termination using separation logic by showing a refinement with a source program [68, 28] or a labeled transition system (LTS) [75], and then use another method to prove termination of the source program/LTS. Finally, there exist specialized logics for proving termination of blocking concurrent programs (*i.e.,* with busy loops) [48, 24]. These logics are however first-order and are not mechanized in a proof assistant.

Rocq-Elpi [69, 71] has been used to implement commands similar to our `Iris Inductive` command. Hierarchy Builder [20] takes advantage of raw syntax trees to use the familiar `Record` syntax to describe algebraic interfaces. The `derive` code synthesis framework [70, 30] can automatically prove monotonicity properties for containers specifications akin to our pre-fixpoint functions, but does so in an ad-hoc way rather than by composing proofs using signatures. The Algebra Tactics framework [61] provides `ring`, `field`, `lra`, `nra` and `psatz` tactics for the Mathematical Components library, via a sophisticated term pre-processing. None of the aforementioned frameworks stress the Rocq-Elpi tactic API or the Ltac1 bridge, since they provide leaf tactics (that close the goal), whereas our IPM tactics generate subgoals.

7 Conclusions and Future Work

We showed that inductive predicates, encoded internally using a least fixpoint in higher-order separation logic, are useful assets for program verification. Our prototype `Iris Inductive` command provides a first step towards making this encoding practical, but future work remains to be done. We could add `Iris CoInductive` (greatest fixpoint), for which we can reuse the generation of the pre-fixpoint function and monotonicy proofs, but have to write a dual version of the generation of the variadic fixpoint and coinduction principle. Low-hanging fruit includes support for improved error messages and better automation for monotonicity proofs (*e.g.,* to declare nested inductive predicates without manual `Proper` instances).

More fundamentally, one should investigate what is the best way to provide both Ltac1 and Rocq-Elpi interfaces for tactics (such as IPM). While Ltac1 is considered legacy by the Rocq developers (no improvements, but also no breaking changes), IPM cannot drop Ltac1 support as it remains the primary language for end-users to write Rocq proofs interactively (*i.e.,* the commands between `Proof` and `Qed`). One could improve the bridge for calling Ltac1 from Elpi, or rewrite IPM in Elpi and provide Ltac1 wrappers. We hope our work constitutes a valuable case study for the Rocq-Elpi and IPM teams to investigate these options.

References

1 Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Error credits: Resourceful reasoning about error bounds for higher-order probabilistic programs. *PACMPL*, 8(ICFP):284–316, 2024. doi:10.1145/3674635.

2 Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

3 Andrew W. Appel. Tactics for separation logic, 2006. Unpublished manuscript. URL: http://www.cs.princeton.edu/~appel/papers/septacs.pdf.

4 Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001. doi:10.1145/504709.504712.

5 Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 109–122, 2007. doi:10.1145/1190216.1190235.

6 Robert Atkey. Amortised resource analysis with separation logic. *LMCS*, 7(2), 2011. doi:10.2168/LMCS-7(2:17)2011.

7 David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In *LPAR*, volume 4790 of *LNCS*, pages 92–106, 2007. doi:10.1007/978-3-540-75560-9_9.

8 Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A framework for higher-order separation logic in Coq. In *ITP*, volume 7406 of *LNCS*, pages 315–331, 2012. doi:10.1007/978-3-642-32347-8_21.

9 Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *TPHOLs*, volume 1690 of *LNCS*, pages 19–36, 1999. doi:10.1007/3-540-48256-3_3.

10 Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *TOPLAS*, 29(5):24, 2007. doi:10.1145/1275497.1275499.

11 Stephen Brookes. A semantics for concurrent separation logic. *TCS*, 375(1-3):227–270, 2007. doi:10.1016/J.TCS.2006.12.034.

12 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *JAR*, 61(1-4):367–422, 2018. doi:10.1007/S10817-018-9457-5.

13 Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *SOSP*, pages 243–258, 2019. doi:10.1145/3341301.3359632.

14 Arthur Charguéraud. Program verification through characteristic formulae. In *ICFP*, pages 321–332, 2010. doi:10.1145/1863543.1863590.

15 Arthur Charguéraud. Higher-order representation predicates in separation logic. In *CPP*, pages 3–14, 2016. doi:10.1145/2854065.2854068.

16 Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *PACMPL*, 4(ICFP):116:1–116:34, 2020. doi:10.1145/3408998.

17 Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth handling of nondeterminism. *TOPLAS*, 45(1):5:1–5:43, 2023. doi:10.1145/3579834.

18 Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *JAR*, 62(3):331–365, 2019. doi:10.1007/S10817-017-9431-7.

19 Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP*, pages 391–402, 2013. doi:10.1145/2500365.2500592.

20 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *FSCD*, volume 167 of *LIPIcs*, pages 34:1–34:21, 2020. doi:10.4230/LIPICS.FSCD.2020.34.

21 Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *I&C*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

22 Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *COLOG-88*, pages 50–66, 1990. doi:10.1007/3-540-52335-9_47.

23 Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *PACMPL*, 5(POPL):1–28, 2021. doi:10.1145/3434314.

24 Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. *TOPLAS*, 43(4):16:1–16:134, 2021. doi:10.1145/3477082.

25 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λprolog interpreter. In *LPAR*, volume 9450 of *LNCS*, pages 460–468, 2015. doi:10.1007/978-3-662-48899-7_32.

26 David Fissore and Enrico Tassi. A new type-class solver for Coq in Elpi. In *Coq Workshop*, 2023. URL: https://inria.hal.science/hal-04467855.

27 Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs. In *S&P*, pages 1416–1433, 2021. doi:10.1109/SP40001.2021.00003.

28 Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: A separation logic framework for verifying concurrent program optimizations. *PACMPL*, 6(POPL):1–31, 2022. doi:10.1145/3498689.

29 Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *PACMPL*, 5(POPL):1–29, 2021. doi:10.1145/3434291.

30 Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality tests, automatically: Deriving eqType instances for Jasmin's data types with Coq-Elpi. In *CPP*, pages 167–181, 2023. doi:10.1145/3573105.3575683.

31 Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In *HUG*, volume 780 of *LNCS*, pages 141–154, 1993. doi:10.1007/3-540-57826-9_131.

32 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *LNCS*, pages 300–314, 2001. doi:10.1007/3-540-45414-4_21.

33 John Harrison. Inductive definitions: Automation and application. In *TPHOL*, volume 971 of *LNCS*, pages 200–213, 1995. doi:10.1007/3-540-60275-5_66.

34 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *PACMPL*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

35 Ralf Jung. Iris Merge Request 60: "Implement greatest fixed point inside the logic", 2017. URL: https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/60.

36 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016. doi:10.1145/2951913.2951943.

37 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 28:e20, 2018. doi:10.1017/S0956796818000151.

38 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: Prophecy variables in separation logic. *PACMPL*, 4(POPL):45:1–45:32, 2020. doi:10.1145/3371113.

39 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015. doi:10.1145/2676726.2676980.

40 Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1928.

41 Robbert Krebbers. Iris commit 1e8054db: "Port fixpoints to BIs", 2017. URL: https://gitlab.mpi-sws.org/iris/iris/-/commit/1e8054db.

42 Robbert Krebbers. Iris Merge Request: "Weakest preconditions for total program correctness", 2017. URL: https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/65.

43 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):77:1–77:30, 2018. doi:10.1145/3236772.

44 Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, volume 10201 of *LNCS*, pages 696–723, 2017. doi:10.1007/978-3-662-54434-1_26.

45 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017. doi:10.1145/3009837.3009855.

46 Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Rocq development of "Inductive predicates via least fixpoints in higher-order separation logic", 2025. doi:10.5281/zenodo.15727403.

47 Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. doi:10.1093/COMJNL/6.4.308.

48 Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *PACMPL*, 2(POPL):20:1–20:31, 2018. doi:10.1145/3158108.

49 Thomas F. Melham. A package for inductive relation definitions in HOL. In *TPHOL*, pages 350–357, 1991.

50 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In *ESOP*, volume 11423 of *LNCS*, pages 3–29, 2019. doi:10.1007/978-3-030-17184-1_1.

51 Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000. doi:10.1109/LICS.2000.855774.

52 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1-3):271–307, 2007. doi:10.1016/J.TCS.2006.12.035.

53 Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999. doi:10.2307/421090.

54 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001. doi:10.1007/3-540-44802-0_1.

55 Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA*, volume 664 of *LNCS*, pages 328–345, 1993. doi:10.1007/BFB0037116.

56 Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *CADE*, volume 814 of *LNCS*, pages 148–161, 1994. doi:10.1007/3-540-58156-1_11.

57 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988. doi:10.1145/53990.54010.

58 Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *MFPS*, volume 442 of *LNCS*, pages 209–228, 1989. doi:10.1007/BFB0040259.

59 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

60 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP*, pages 284–298, 2020. doi:10.1145/3372885.3373818.

61 Kazuhiko Sakaguchi. Reflexive tactics for algebra, revisited. In *ITP*, volume 237 of *LIPIcs*, pages 29:1–29:22, 2022. doi:10.4230/LIPICS.ITP.2022.29.

62 Thomas Somers and Robbert Krebbers. Verified lock-free session channels with linking. *PACMPL*, 8(OOPSLA2):588–617, 2024. doi:10.1145/3689732.

63 Matthieu Sozeau. A new look at generalized rewriting in type theory. *JFR*, 2(1):41–62, 2009. doi:10.6092/ISSN.1972-5787/1574.

64 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008. doi:10.1007/978-3-540-71067-7_23.

65 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*, pages 80–95, 2021. doi:10.1145/3453483.3454031.

66 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 2:285–309, 1955. doi:10.2140/pjm.1955.5.285.

67 Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *PACMPL*, 3(POPL):64:1–64:30, 2019. doi:10.1145/3290377.

68 Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In *ESOP*, volume 10201 of *LNCS*, pages 909–936, 2017. doi:10.1007/978-3-662-54434-1_34.

69 Enrico Tassi. Elpi: An extension language for Coq (metaprogramming Coq in the Elpi λProlog dialect). In *CoqPL*, 2018. URL: https://inria.hal.science/hal-01637063.

70 Enrico Tassi. Deriving proved equality tests in Coq-Elpi: Stronger induction principles for containers in Coq. In *ITP*, volume 141 of *LIPIcs*, pages 29:1–29:18, 2019. doi:10.4230/LIPICS.ITP.2019.29.

71 Enrico Tassi. Elpi: Rule-based meta-language for Rocq. In *CoqPL*, 2025. URL: https://inria.hal.science/hal-04990628.

72 The Rocq Prover development team. Rocq 9.0.0 reference manual, section inductive types and recursive functions, 2026. URL: https://rocq-prover.org/doc/V9.0.0/refman/language/core/inductive.html.

73 The std++ developers and contributors. Rocq-std++: An extended "Standard Library" for Rocq, 2024. URL: https://gitlab.mpi-sws.org/iris/stdpp/.

74 Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *PACMPL*, 3(ICFP):105:1–105:28, 2019. doi:10.1145/3341709.

75 Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *PACMPL*, 8(POPL):241–272, 2024. doi:10.1145/3632851.

76 Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS*, pages 596–605, 2012. doi:10.1109/LICS.2012.75.

77 Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi. Master's thesis, Radboud University Nijmegen, 2024. doi:10.5281/zenodo.15727560.

78 Orpheas van Rooij and Robbert Krebbers. Affect: An affine type and effect system. *PACMPL*, 9(POPL):126–154, 2025. doi:10.1145/3704841.

79 Max Vistrup, Michael Sammler, and Ralf Jung. Program logics à la carte. *PACMPL*, 9(POPL):300–331, 2025. doi:10.1145/3704847.