Fixpoints in Higher-Order Separation Logic

Robbert Krebbers[1]   Luko van der Maas[1]   Enrico Tassi[2]

[1]Radboud University Nijmegen, The Netherlands

[2]Université Côte d'Azur, Inria, France

September 28, 2025 @ ITP, Reykjavík, Iceland

The overaching goal

A full-fledged ITP for separation logic

A full-fledged ITP for separation logic

Why separation logic?

- A form of substructural logic (no contraction)
- Widely used in program verification

A full-fledged ITP for separation logic embedded in an off-the-shelf ITP

Why separation logic?

- A form of substructural logic (no contraction)
- Widely used in program verification

The overaching goal

A full-fledged ITP for separation logic embedded in an off-the-shelf ITP

Why separation logic?

- A form of substructural logic (no contraction)
- Widely used in program verification

Why an embedding?

- Prove soundness of the embedded ITP (reduces TCB to host ITP)
- Reuse infrastructure of the host ITP
- Users do not need to learn new tool

The goal of this paper

Support for inductive predicates

Support for inductive predicates

I will explain:

- Why you need inductive predicates in separation logic
- Why you do not get them for free from the host ITP
- How to encode them using a least fixpoint theorem
- How to automate this process using Iris Proof Mode (IPM) and Rocq-Elpi

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iI
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

lemma in separation logic

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iD
  iS
  -
  -
    iFrame.
Qed.
```

instead of abstract P, Q, Φ we could also use concrete assertions:

- $\ell \mapsto v$: location ℓ contains value v
- wp $e\,\{\Phi\}$: program e is safe and has postcondition Φ

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
______________________(1/1)
P ∗ (∃ a : A, Φ a) ∗ Q
−∗ Q ∗ (∃ a : A, P ∗ Φ a)
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
______________________(1/1)
"H1" : P
"H2" : ∃ a : A, Φ a
"H3" : Q
--------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
--------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
-------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

$*$ means: resources should be split

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
```

the hypotheses for the left conjunct

```
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
"H1" : P
"H2" : Φ x
"H3" : Q
---------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

∗ means: resources should be split

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q -∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
2 subgoals
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/2)
"H3" : Q
-----------------------∗
Q

______________________(2/2)
"H1" : P
"H2" : Φ x
-----------------------∗
∃ a : A, P ∗ Φ a
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  by iFrame.
Qed.
```

we can also solve this goal automatically

```
1 subgoal
A : Type
P, Q : iProp
Φ : A → iProp
x : A
______________________(1/1)
"H1" : P
"H2" : ∃ a, Φ a
"H3" : Q
-------------------------∗
Q ∗ (∃ a : A, P ∗ Φ a)
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  by iFrame.
Qed.
```

we can also solve this goal automatically

```
No more subgoals.
```

Separation logic and Iris Proof Mode 101

```
Lemma demo {A} (P Q : iProp) (Φ : A → iProp) :
  P ∗ (∃ a, Φ a) ∗ Q -∗ Q ∗ ∃ a, P ∗ Φ a.
Proof.
  iIntros "[$ [? $]] //".
Qed.
```

or use intro patterns

This paper in a nutshell

```
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.
```

Add the `Iris` keyword to define an inductive predicate in separation logic

You can do `iInduction` on the derivation

Ways to define inductive predicates in separation logic

Method	**Restriction**	**Ease of use**
Rocq's `Fixpoint`	Structural recursion	☺ Trivial
Banach fixpoint	Guarded by $\triangleright$	☺ Prove `Contractive`
Least and greatest fixpoint	Monotonicity/positivity	☹ Prior to this paper

Ways to define inductive predicates in separation logic

Metho
Rocq's
Banach
Least a

Least and greatest fixpoint:

Prior to this paper, the following needs to be done manually:

- Define fixpoint function as disjunction of cases
- Curry/uncurry
- Prove monotonicity
- Lift folding/unfolding lemmas
- Lift constructors and induction principle

Finally, applying the induction principle is aweful

Least fixpoints in separation logic

[Folklore result, akin to Baelde/Miller 2007 in linear logic, mechanized in Iris by Jung/Krebbers 2017]

Theorem

Given a pre-fixpoint function $F : (A \to iProp) \to (A \to iProp)$ that is monotone:

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

There exists a least fixpoint $\mu F : A \to iProp$ with:

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

Least fixpoints in separation logic

[Folklore result, akin to Baelde/Miller 2007 in linear logic, mechanized in Iris by Jung/Krebbers 2017]

Theorem

Given a pre-fixpoint function $F : (A \to iProp) \to (A \to iProp)$ that is monotone:

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

There exists a least fixpoint $\mu F : A \to iProp$ with:

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

$\Box\,P$ means P can be used multiple times (akin to “bang” ! in linear logic)

- $\Box\,P \mathrel{*\!\!*} \Box\,P * \Box\,P$
- $\Box\,P \mathrel{-\!\!*} P$ **holds unconditionally**
- $P \mathrel{-\!\!*} \Box\,P$ **holds only for specific** P

Least fixpoints in separation logic

[Folklore result, akin to Baelde/Miller 2007 in linear logic, mechanized in Iris by Jung/Krebbers 2017]

Theorem

Given a pre-fixpoint function $F : (A \to iProp) \to (A \to iProp)$ that is monotone:

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

There exists a least fixpoint $\mu\, F : A \to iProp$ with:

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

Proof.

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

Least fixpoints in separation logic

[Folklore result, akin to Baelde/Miller 2007 in linear logic, mechanized in Iris by Jung/Krebbers 2017]

Theorem

Given a pre-fixpoint function $F : (A \to iProp) \to (A \to iProp)$ that is monotone:

$$\forall(\Phi_1, \Phi_2 : A \to iProp).\ \Box(\forall x : A.\, \Phi_1\, x \twoheadrightarrow \Phi_2\, x) \twoheadrightarrow \forall x : A.\, F\,\Phi_1\, x \twoheadrightarrow F\,\Phi_2\, x$$

There exists a least fixpoint $\mu\, F : A \to iProp$ with:

1. *(Fixpoint equations)* $\forall x.\, F\,(\mu\, F)\, x \ast\!\!\ast\ \mu\, F\, x$
2. *(Iteration principle)* $\Box(\forall x.\, F\,\Phi\, x \twoheadrightarrow \Phi\, x) \twoheadrightarrow \forall x.\, \mu\, F\, x \twoheadrightarrow \Phi\, x$

Proof.

Define $\mu\, F \triangleq \lambda x.\, \forall(\Phi : A \to \mathrm{iProp}).\ \Box(\forall y.\, F\,\Phi\, y \twoheadrightarrow \Phi\, y) \twoheadrightarrow \Phi\, x$ $\square$

needs higher-order impredicative separation logic (Iris gets that for free from Rocq)

Least fixpoints in separation logic

[Folklore result, akin to Baelde/Miller 2007 in linear logic, mechanized in Iris by Jung/Krebbers 2017]

Theorem

Given a pre-fixpoint function $F : (A \to iProp) \to (A \to iProp)$ that is monotone:

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

There ex

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

Greatest fixpoints are dual:

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

Proof.

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

needs higher-order impredicative separation logic (Iris gets that for free from Rocq)

Example 1: The simplest list predicate

```
Iris Inductive is_list : loc → list val → iProp :=
  | is_list_nil l :
     l ↦ NIL -∗
     is_list_with l []
  | is_list_cons v vs l l' :
     l ↦ CONS (v,#l') -∗
     is_list l' vs -∗
     is_list l (v :: vs).
```

Example 1: The simplest list predicate

```
Iris Inductive is_list : loc → list val → iProp :=
  | is_list_nil l :
     l ↦ NIL -∗
     is_list_with l []
  | is_list_cons v vs l l' :
     l ↦ CONS (v,#l') -∗
     is_list l' vs -∗
     is_list l (v :: vs).
```

iProp is a complicated Σ-type (*i.e.*, not a sort), so ordinary Rocq Inductive would not accept this

Example 1: The simplest list predicate

```
Iris Inductive is_list : loc → list val → iProp :=
  | is_list_nil l :
     l ↦ NIL -∗
     is_list_with l []
  | is_list_cons v vs l l' :
     l ↦ CONS (v,#l') -∗
     is_list l' vs -∗
     is_list l (v :: vs).
```

$$F_{\mathsf{isList}} : (\mathsf{loc} \times \mathsf{list\ val} \to \mathsf{iProp}) \to \mathsf{loc} \times \mathsf{list\ val} \to \mathsf{iProp}$$

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

$$\mathsf{isList}\ \ell\ \vec{v} \triangleq \mu\, F_{\mathsf{isList}}\,(\ell, \vec{v})$$

Example 1: The simplest list predicate

```
Iris Inductive is_list : loc → list val → iProp :=
  | is_list_nil l :
    l ↦ NIL -∗
    is_list_with l []
  |
```

Iteration:

$$\frac{\begin{array}{l} \square(\forall \ell.\quad \ell \mapsto \texttt{NIL} \quad -\!\!* \ \Phi\,\ell\,[\,]) \ * \\ \square(\forall \ell, \ell', w, \vec{w}.\ \ell \mapsto \texttt{CONS}\,(w,\ \ell') \ -\!\!*\ \Phi\,\ell'\,\vec{w} \quad -\!\!*\ \Phi\,\ell\,(w :: \vec{w})) \end{array}}{\forall \ell, \vec{v}.\ \mathsf{isList}\ \ell\ \vec{v} \ -\!\!*\ \Phi\,\ell\,\vec{v}}$$

F_{isList} : (loc × list val → iProp) → loc × list val → iProp

$$F_{\mathsf{isList}}\ \mathit{rec}\ (\ell, \vec{v}) \triangleq (\ell \mapsto \texttt{NIL} * \vec{v} = [\,]) \lor (\exists \ell', w, \vec{w}.\ \ell \mapsto \texttt{CONS}\,(w,\ \ell') * \mathit{rec}\,(\ell', \vec{w}) * \vec{v} = w :: \vec{w})$$

$$\mathsf{isList}\ \ell\ \vec{v} \triangleq \mu\, F_{\mathsf{isList}}\,(\ell, \vec{v})$$

Example 1: The simplest list predicate

```
Iris Inductive is_list : loc → list val → iProp :=
  | is_list_nil l :
    l ↦ NIL -∗
    is_list_with l []
  |
```

Iteration and induction:

$$\frac{\begin{array}{l}\square(\forall \ell.\quad \ell \mapsto \mathtt{NIL} \twoheadrightarrow \Phi\,\ell\,[\,]) \ast \\ \square(\forall \ell, \ell', w, \vec{w}.\, \ell \mapsto \mathtt{CONS}\,(w,\ \ell') \twoheadrightarrow (\Phi\,\ell'\,\vec{w} \wedge \mathsf{isList}\ \ell'\ \vec{w}) \twoheadrightarrow \Phi\,\ell\,(w :: \vec{w}))\end{array}}{\forall \ell, \vec{v}.\ \mathsf{isList}\ \ell\ \vec{v} \twoheadrightarrow \Phi\,\ell\,\vec{v}}$$

$$F_{\mathsf{isList}}\ \mathit{rec}\ (\ell, \vec{v}) \triangleq (\ell \mapsto \mathtt{NIL} \ast \vec{v} = [\,]) \vee$$
$$(\exists \ell', w, \vec{w}.\, \ell \mapsto \mathtt{CONS}\,(w,\ \ell') \ast \mathit{rec}\,(\ell', \vec{w}) \ast \vec{v} = w :: \vec{w})$$

$$\mathsf{isList}\ \ell\ \vec{v} \triangleq \mu\, F_{\mathsf{isList}}\,(\ell, \vec{v})$$

Example 1: The simplest list predicate

```
Iris Inductive is_list : loc → list val → iProp :=
  | is_list_nil l :
     l ↦ NIL -∗
     is_list_with l []
  | is_list_cons v vs l l' :
     l ↦ CONS (v,#l') -∗
     is_list l' vs -∗
     is_list l (v :: vs).
```

decreasing argument, so `Fixpoint` would have worked too

$$F_{\mathsf{isList}} : (\mathsf{loc} \times \mathsf{list\ val} \to \mathsf{iProp}) \to \mathsf{loc} \times \mathsf{list\ val} \to \mathsf{iProp}$$

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

$$\mathsf{isList}\ \ell\ \vec{v} \triangleq \mu\, F_{\mathsf{isList}}\,(\ell, \vec{v})$$

Example 2: No decreasing argument

```
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.
```

Example 2: No decreasing argument

```
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.
```

node marked as ‘deleted’, common in concurrent data structures

Example 2: No decreasing argument

```
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.
```

no decreasing argument, `Fixpoint` would **not** work

node marked as ‘deleted’, common in concurrent data structures

Example 2: No decreasing argument

```
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.
```

parameters are supported similarly to Rocq

no decreasing argument, `Fixpoint` would **not** work

node marked as 'deleted', common in concurrent data structures

Example 2: No decreasing argument

```
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.
```

$$F_{\text{isListWithTl}} : \text{loc} \to (\text{loc} \times \text{list val} \to \text{iProp}) \to \text{loc} \times \text{list val} \to \text{iProp}$$

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

$$\text{isListWithTl}\ tl\ \ell\ \vec{v} \triangleq \mu\,(F_{\text{isListWithTl}}\ tl)\ (\ell, \vec{v})$$

Example 2: No decreasing argument

```
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' :
```

Iteration and induction:

$$\frac{\begin{array}{l} \Box(\qquad tl \mapsto \texttt{NIL} \qquad \twoheadrightarrow \Phi\ tl\ [\,]) \ast \\ \Box(\forall \ell, \ell', \vec{w}.\ \ell \mapsto \texttt{DEL}\ \ell' \twoheadrightarrow (\Phi\ \ell'\ \vec{w} \wedge \mathsf{isListWithTl}\ tl\ \ell'\ \vec{w}) \twoheadrightarrow \Phi\ \ell\ \vec{w}) \ast \\ \Box(\forall \ell, \ell', w, \vec{w}.\ \ell \mapsto \texttt{CONS}\,(w,\ \ell') \twoheadrightarrow (\Phi\ \ell'\ \vec{w} \wedge \mathsf{isListWithTl}\ tl\ \ell'\ \vec{w}) \twoheadrightarrow \Phi\ \ell\,(w :: \vec{w})) \end{array}}{\forall \ell, \vec{v}.\ \mathsf{isListWithTl}\ tl\ \ell\ \vec{v} \twoheadrightarrow \Phi\ \ell\ \vec{v}}$$

$$F_{\mathsf{isListWithTl}} : \mathsf{loc} \to (\mathsf{loc} \times \mathsf{list\ val} \to \mathsf{iProp}) \to \mathsf{loc} \times \mathsf{list\ val} \to \mathsf{iProp}$$

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

$$\mathsf{isListWithTl}\ tl\ \ell\ \vec{v} \triangleq \mu\,(F_{\mathsf{isListWithTl}}\ tl)\,(\ell, \vec{v})$$

Example 3: Multiple recursion

```
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)
```

Example 3: Multiple recursion

```
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)
```

`Fixpoint` would not recognize the sets `Xl` and `Xr` are smaller

Example 3: Multiple recursion

```
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)
```

To prove monotonicity (the premise of the fixpoint theorem), we really need the persistence modality in the definition of monotonicity:

$$\forall(\Phi_1, \Phi_2 : A \to \mathrm{iProp}).\ \Box(\forall x : A.\, \Phi_1\, x \twoheadrightarrow \Phi_2\, x) \twoheadrightarrow \forall x : A.\, F\, \Phi_1\, x \twoheadrightarrow F\, \Phi_2\, x$$

Example 4: Higher-order representation predicates [Charguéraud, 2015]

```
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.
```

Example 4: Higher-order representation predicates [Charguéraud, 2015]

```
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.
```

parameters can be higher-order

Example 5: Nested recursion

```
Inductive rose_tree :=
  | Node : list rose_tree → rose_tree.

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).

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).
```

Example 5: Nested recursion

```
Inductive rose_tree :=
  | Node : list rose_tree → rose_tree.

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).

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).
```

nested recursion

Example 5: Nested recursion

```
Inductive rose_tree :=
  | Node : list rose_tree → rose_tree.

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).

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).
```

nested recursion

`is_rose_tree` is well-defined because `is_ho_list_loc` is monotone in Φ

Example 6: Total weakest precondition

$$\mathsf{wp}\ e\,[\,\Phi\,] \triangleq \text{“}e \text{ terminates and postcondition } \Phi \text{ holds for all results”}$$

Example 6: Total weakest precondition

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

Example 6: Total weakest precondition

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

compared to vanilla/partial Iris, no $\triangleright$ here

Example 6: Total weakest precondition

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

compared to vanilla/partial Iris, no $\triangleright$ here

well-defined as least fixpoint because recursion is positive

Example 6: Total weakest precondition

$$\mathrm{wp}\ e\,[\Phi] \triangleq \begin{cases} \Rrightarrow \Phi\ e & \text{if } e \in \mathrm{val} \\ \forall h.\, \mathcal{S}\ h \twoheadrightarrow{*} \Rrightarrow \mathrm{reducible}(e, h) * & \text{if } e \notin \mathrm{val} \\ \qquad \forall e_2, h_2.\, ((e, h) \to (e_2, h_2)) \twoheadrightarrow{*} & \\ \qquad\qquad \Rrightarrow \mathcal{S}\ h_2 * \mathrm{wp}\ e_2\,[\Phi] & \end{cases}$$

compared to vanilla/partial Iris, no $\triangleright$ here

well-defined as least fixpoint because recursion is positive

☺ As expressive as other program logics for total correctness (*e.g.*, CFML)

- Weaker "stepping rules" than vanilla/partial Iris, so you cannot use Löb induction
- Prove termination by induction on Rocq type or `Iris Inductive`

Example 6: Total weakest precondition

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

compared to vanilla/partial Iris, no $\triangleright$ here

well-defined as least fixpoint because recursion is positive

- ☺ As expressive as other program logics for total correctness (*e.g.*, CFML)
  - Weaker "stepping rules" than vanilla/partial Iris, so you cannot use Löb induction
  - Prove termination by induction on Rocq type or `Iris Inductive`
- ☹ Limited support for concurrency and Iris-style invariants
  - Only open timeless invariants
  - Termination for every scheduling (including unfair ones)

Example 6: Total weakest precondition in Rocq

With concurrency and other bells and whistles

```
Iris Inductive twp (s : stuckness) : coPset → expr → (val -d> iProp) -n> iProp :=
  | twp_some E v e1 Φ :
     (|={E}=> Φ v) -∗
     ⌜ to_val e1 = Some v ⌝ -∗
     twp s E e1 Φ
  | twp_none E e1 Φ :
     (∀ σ1 ns κs nt,
       state_interp σ1 ns κs nt ={E,∅}=∗
         ⌜ if s is NotStuck then reducible_no_obs e1 σ1 else True ⌝ ∗
        ∀ κ e2 σ2 efs, ⌜ prim_step e1 σ1 κ e2 σ2 efs ⌝ ={∅,E}=∗
          ⌜ κ = [] ⌝ ∗
          state_interp σ2 (S ns) κs (length efs + nt) ∗
          twp s E e2 Φ ∗
          [∗ list] ef ∈ efs, twp s ⊤ ef fork_post) -∗
     ⌜ to_val e1 = None ⌝ -∗
     twp s E e1 Φ.
```

Now let us discuss the implementation

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
2. Prove that the pre-fixpoint function is monotone
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺ Rocq-Elpi allows to use Rocq's `Inductive` syntax
2. Prove that the pre-fixpoint function is monotone
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺ Rocq-Elpi allows to use Rocq's `Inductive` syntax
   - ☺ Rocq-Elpi has good support to manipulate untyped terms involving binders
2. Prove that the pre-fixpoint function is monotone
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺ Rocq-Elpi allows to use Rocq's `Inductive` syntax
   - ☺ Rocq-Elpi has good support to manipulate untyped terms involving binders
   - ☹ Making the fixpoint theorem variadic is tricky, we specialize the theorem
2. Prove that the pre-fixpoint function is monotone
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺ Rocq-Elpi allows to use Rocq's `Inductive` syntax
   - ☺ Rocq-Elpi has good support to manipulate untyped terms involving binders
   - ☹ Making the fixpoint theorem variadic is tricky, we specialize the theorem
2. Prove that the pre-fixpoint function is monotone
   - ☺ Port Rocq's `Proper` mechanism to Iris to specify variadic monotonicity
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺ Rocq-Elpi allows to use Rocq's `Inductive` syntax
   - ☺ Rocq-Elpi has good support to manipulate untyped terms involving binders
   - ☹ Making the fixpoint theorem variadic is tricky, we specialize the theorem
2. Prove that the pre-fixpoint function is monotone
   - ☺ Port Rocq's `Proper` mechanism to Iris to specify variadic monotonicity
   - ☹ Interfacing between Rocq-Elpi and Ltac is no good
     Need to port some Iris Proof Mode (IPM) tactics to Rocq-Elpi
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic

Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺ Rocq-Elpi allows to use Rocq's `Inductive` syntax
   - ☺ Rocq-Elpi has good support to manipulate untyped terms involving binders
   - ☹ Making the fixpoint theorem variadic is tricky, we specialize the theorem
2. Prove that the pre-fixpoint function is monotone
   - ☺ Port Rocq's `Proper` mechanism to Iris to specify variadic monotonicity
   - ☹ Interfacing between Rocq-Elpi and Ltac is no good
     Need to port some Iris Proof Mode (IPM) tactics to Rocq-Elpi
3. Generate fixpoint equations, constructors, induction principle
4. Hook for `iInduction` tactic
   - ☺ Databases in Rocq-Elpi are great

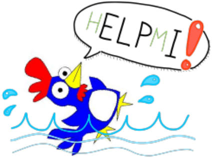
Rocq-Elpi command

Steps:

1. Generate the pre-fixpoint function and fixpoint
   - ☺
   - ☺
   - ☹
2. Pro
   - ☺
   - ☹
3. Gen
4. Hook for `iInduction` tactic
   - ☺ Databases in Rocq-Elpi are great

What is Rocq-Elpi anyway?

- Modern meta-programming language for Rocq
- Based on λ-Prolog
- HOAS for manipulating binders
- Many bindings to the Rocq API
- Actively developed



Step 1: Generate the pre-fixpoint function and fixpoint

```
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.
```

```
Definition is_list_with_tl_pre (tl : loc) :=
  λ (rec : loc → list val → iProp),
    λ (l : loc) (vs : list val),
      tl ↦ NIL ∗ ⌜vs = []⌝ ∗ ⌜l = tl⌝
    ∨ (∃ (v : val) (vs' : list val) (l' : loc),
         l ↦ CONS (v, #l') ∗ rec l' vs ∗ ⌜vs = v :: vs'⌝)
    ∨ ∃ (l' : loc), l ↦ DEL #l' ∗ rec l' vs.
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 Iris command takes the Inductive as an argument
- The Inductive is not type checked/elaborated, Rocq-Elpi gives an untyped AST
- Rocq-Elpi allows processing that AST and controlling when to type check

Step 1: Generate the pre-fixpoint function and fixpoint

```
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.
```

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

specialized for variadic case to avoid currying/telescopes

- The Iris command takes the Inductive as an argument
- The Inductive is not type checked/elaborated, Rocq-Elpi gives an untyped AST
- Rocq-Elpi allows processing that AST and controlling when to type check

Step 2: Prove that the pre-fixpoint function is monotone

Goal: Specify and prove that a variadic function is monotone

We port `Proper` [Sozeau, 2009] to separation logic:

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

Step 2: Prove that the pre-fixpoint function is monotone

Goal: Specify and prove that a variadic function is monotone

We port `Proper` [Sozeau, 2009] to separation logic:

	Type	Signature
$*$	$\mathrm{iProp} \to \mathrm{iProp} \to \mathrm{iProp}$	$(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$
$\mathrel{-\!\!*}$	$\mathrm{iProp} \to \mathrm{iProp} \to \mathrm{iProp}$	$\mathrm{flip}\,(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$
$\square$	$\mathrm{iProp} \to \mathrm{iProp}$	$\square\,(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$
$\exists$	$(A \to \mathrm{iProp}) \to \mathrm{iProp}$	$((=)_A \Longrightarrow (\mathrel{-\!\!*})) \Longrightarrow (\mathrel{-\!\!*})$

Proving that pre-fixpoint functions are monotone is done by goal-directed proof search, similar to std++'s `solve_proper` (but without non-determinism)

Interlude: Porting IPM tactics to Rocq-Elpi

To generate the proofs of monotonicity, fixpoint equations, constructors, induction principles, *etc.* we need to generate proofs in Iris Proof Mode (IPM)

Problem: Interfacing between Rocq-Elpi and Ltac1 is brittle
Solution: Port selected IPM tactics to Rocq-Elpi

Interlude: Porting IPM tactics to Rocq-Elpi

To generate the proofs of monotonicity, fixpoint equations, constructors, induction principles, *etc.* we need to generate proofs in Iris Proof Mode (IPM)

Problem: Interfacing between Rocq-Elpi and Ltac1 is brittle
Solution: Port selected IPM tactics to Rocq-Elpi

```
pred eiIntro-ident i:ident, i:igoal, o:igoal.
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".
```

Step 4: Hook for `iInduction` tactic

To perform `iInduction` we need to lookup the right induction principle

Solution: Define Rocq-Elpi database to register information about inductives:

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

Add entries as Prolog-style clauses:

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

Summary of the paper

- The folklore method of encoding least/greatest fixpoints in higher-order separation logic
- Many examples
- The Iris total weakest precondition (which I defined in 2017, but until now did not describe in a paper)
- A prototype `Iris Inductive` command implemented using Rocq-Elpi

Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic

Robbert Krebbers
Radboud University, Nijmegen, The Netherlands

Luko van der Maas
Radboud University, Nijmegen, The Netherlands

Enrico Tassi
Université Côte d'Azur, Inria, Nice, 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`

Funding 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.

Summary of the paper

- The folklore method of encoding least/greatest fixpoints in higher-order sepa…
- Mar…
- The… I de… describe in a paper)
- A prototype `Iris Inductive` command implemented using Rocq-Elpi

No more need for such footnotes ☺

[10] This part of Iris was never described in a paper; it can be found at https://gitlab.mpi-sws.org/iris/iris/-/blob/iris-4.2.0/iris/program_logic/total_weakestpre.v.

Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic

Robbert Krebbers
Radboud University, Nijmegen, The Netherlands
Luko van der Maas
Radboud University, Nijmegen, The Netherlands
Enrico Tassi
Université Côte d'Azur, Inria, Nice, France

… 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

Funding 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.

Ltac versus Rocq-Elpi

- Ltac1 will remain the primary language for users to write interactive Rocq proofs (between `Proof` and `Qed`)
- Rocq-Elpi is much better for meta programming
- Maintaining a version of each IPM tactic in Rocq-Elpi and Ltac1 is awful
- Not clear what is the best way forward

Ltac versus Rocq-Elpi

- Ltac1 will remain the primary language for users to write interactive Rocq proofs (between `Proof` and `Qed`)
- Rocq-Elpi is much better for meta programming
- Maintaining a version of each IPM tactic in Rocq-Elpi and Ltac1 is awful
- Not clear what is the best way forward

The elephant in the room: what about Ltac2?

Future work

- **(easy)** Support for greatest fixpoints, *i.e.*, `Iris CoInductive`
- **(engineering)** Improve quality of life (*e.g.*, sealing, error messages, simplify code)
- **(hard)** Investigate how to interface between Ltac1 and Rocq-Elpi, needed to upstream `Iris Inductive` into Iris