

Interactive Proofs in Higher-Order Concurrent Separation Logic

Robbert Krebbers¹ Amin Timany² Lars Birkedal³

¹Delft University of Technology, The Netherlands

²imec-Distrinet, KU Leuven, Belgium

³Aarhus University, Denmark

January 18, 2017 @ POPL, Paris, France

Goal of this talk

Many POPL papers about complicated program logics come with mechanized soundness proofs, but how to reason in these logics?

Goal: reasoning in an object logic in the same style as reasoning in Coq

Goal of this talk

Many POPL papers about complicated program logics come with mechanized soundness proofs, but how to reason in these logics?

Goal: reasoning in an object logic in the same style as reasoning in Coq

Goal of this talk

Many POPL papers about complicated program logics come with mechanized soundness proofs, but how to reason in these logics?

Goal: reasoning in an object logic in the same style as reasoning in Coq

How?

- ▶ Extend Coq with (spatial and non-spatial) named proof contexts for an object logic
- ▶ Tactics for introduction and elimination of all connectives of the object logic
- ▶ Entirely implemented using reflection, type classes and Ltac (no OCaml plugin needed)



Goal of this talk

Many POPL papers about complicated program logics come with mechanized soundness proofs, but how to reason in these logics?

Goal: reasoning in *Iris* in the same style as reasoning in Coq

How?

- ▶ Extend Coq with (spatial and non-spatial) named proof contexts for *Iris*
- ▶ Tactics for introduction and elimination of all connectives of *Iris*
- ▶ Entirely implemented using reflection, type classes and Ltac (no OCaml plugin needed)



Iris: language independent higher-order separation logic for modular reasoning about fine-grained concurrency in Coq

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R -* ∃ a, Ψ a * P.`

Proof.

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

-----^(1/1)

`P * (∃ a : A, Ψ a) * R -* ∃ a : A, Ψ a * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R -* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`P * (∃ a : A, Ψ a) * R -* ∃ a : A, Ψ a * P` (1/1)

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

-----(1/1)

`"HP" : P`

`"HΨ" : ∃ a : A, Ψ a`

`"HR" : R`

-----*

`∃ a : A, Ψ a * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

-----(1/1)

`"HP" : P`

`"HΨ" : ∃ a : A, Ψ a`

`"HR" : R`

-----*

`∃ a : A, Ψ a * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----(1/1)

`"HP" : P`

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`∃ a : A, Ψ a * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

----- (1/1)

`"HP" : P`

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`∃ a : A, Ψ a * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

----- (1/1)

`"HP" : P`

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`Ψ x * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----(1/1)

`"HP" : P`

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`Ψ x * P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

2 subgoals

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

----- (1/2)

`"HΨ" : Ψ x`

----- *

`Ψ x`

----- (2/2)

`"HP" : P`

`"HR" : R`

----- *

`P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

`-`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----(1/1)

`"HΨ" : Ψ x`

-----*

`Ψ x`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

`- iAssumption.`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----(1/1)

`"HΨ" : Ψ x`

-----*

`Ψ x`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`
`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

```
iIntros "[HP [HΨ HR]]".  
iDestruct "HΨ" as (x) "HΨ".  
iExists x.  
iSplitL "HΨ".  
- iAssumption.
```

This subproof is complete, but there are some unfocused goals:

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

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

`- iAssumption.`

`-`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----^(1/1)

`"HP" : P`

`"HR" : R`

-----*

`P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

`- iAssumption.`

`- iAssumption.`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

----- (1/1)

`"HP" : P`

`"HR" : R`

-----*

`P`

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :` `No more subgoals.`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

```
iIntros "[HP [HΨ HR]]".
iDestruct "HΨ" as (x) "HΨ".
iExists x.
iSplitL "HΨ".
- iAssumption.
- iAssumption.
```

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :` `No more subgoals.`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

```
iIntros "[HP [HΨ HR]]".  
iDestruct "HΨ" as (x) "HΨ".  
iExists x.  
iSplitL "HΨ".  
- iAssumption.  
- iAssumption.
```

Qed.

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iExistential_intro (λ a, Ψ a) HR.`

Logical notations overridden in scope for Iris

`- iAssumption.`

`- iAssumption.`

Qed.

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

-----(1/1)

`"HP" : P`

`"HΨ" : ∃ a : A, Ψ a`

`"HR" : R`

Notation for deeply embedded context

P

*

Iris Proof Mode (IPM) demo

```
Lemma and_exist_sep {A} P R (Ψ: A → iProp) :  
  P * (∃ a, Ψ a) * R →* ∃ a, Ψ a * P.
```

Proof.

```
iIntros "[HP [HΨ HR]]".  
Unset Printing Notations.
```

```
1 subgoal  
M : ucmraT  
A : Type  
P, R : iProp  
Ψ : A → iProp
```

-----(1/1)

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

Notation for deeply embedded context

P

*

Iris Proof Mode (IPM) demo

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`Unset Printing Notations.`

`1 subgoal`

`M : ucMraT`

`A : Type@{Top.105}`

`P, R : uPred M`

`Ψ : forall _ : A, uPred M`

`(1/1)`

`@uPred_entails M`

`(@of_envs M`

`(@Envs M (@Enil (uPred M))`

`(@Esnoc (uPred M)`

`(@Esnoc (uPred M)`

`(@Esnoc (uPred M) (@Enil (uPred M))`

`(String`

`(Ascii false false false true false false true
false)`

`(String`

`(Ascii false false false false true false true
false) EmptyString)) P)`

`(String`

`(Ascii false false false true false false true false)`

`(String`

`(Ascii false true true true false false true true)`

`(String`

`(Ascii false false false true false true false
true) EmptyString)))`

`(@uPred_exist M A (fun a : A => Ψ a)))`

`(String`

`(Ascii false false false true false false true false)`

Why should we care about interactive proofs? Why not automate everything?

Infeasible to automate everything, for example:

- ▶ Concurrent algorithms in Iris (Jung, Krebbers, Swasey, Timany)
- ▶ The Rust type system in Iris (Jung, Jourdan, Dreyer, Krebbers)
- ▶ Logical relations in Iris (Krogh-Jespersen, Svendsen, Timany, Birkedal, Tassarotti, Jung, Krebbers)
- ▶ Weak memory concurrency in Iris (Kaiser, Dang, Dreyer, Lahav, Vafeiadis)
- ▶ Object calculi in Iris (Swasey, Dreyer, Garg)
- ▶ Logical atomicity in Iris (Krogh-Jespersen, Zhang, Jung)
- ▶ Defining Iris in Iris (Krebbers, Jung, Jourdan, Bizjak, Dreyer, Birkedal)

Most of these projects are formalized in IPM

How to do such proofs in a proof assistant?

Current proof assistant support is limited to **basic** separation logic:

- ▶ **Macros for manipulating Hoare triples:** Appel, Wright, Charge!, ...
- ▶ **Heavy automation:** Bedrock, Rtac, ...

Iris has many complicated connectives that are beyond basic separation logic

How to embed a logic into a proof assistant

Deep embedding

```
Inductive form : Type :=  
  | iAnd: form → form → form  
  | iForall: string → form → form → form
```

Shallow embedding

```
Definition iProp : Type :=  
  (* predicates over states *).  
Definition iAnd : iProp → iProp → iProp :=  
  (* semantic interpretation *).  
Definition iForall : ∀ A, (A → iProp) → iProp :=  
  (* semantic interpretation *).
```

How to embed a logic into a proof assistant

Deep embedding

```
Inductive form : Type :=  
  | iAnd: form → form → form  
  | iForall: string → form → form → form
```

Traverse formulas using Coq functions (fast)

Reflective tactics (fast)

Shallow embedding

```
Definition iProp : Type :=  
  (* predicates over states *).  
Definition iAnd : iProp → iProp → iProp :=  
  (* semantic interpretation *).  
Definition iForall : ∀ A, (A → iProp) → iProp :=  
  (* semantic interpretation *).
```

Traverse formulas on the meta level (slow)

Tactics on the meta level (slow)

How to embed a logic into a proof assistant

Deep embedding

```
Inductive form : Type :=  
  | iAnd: form → form → form  
  | iForall: string → form → form → form
```

Traverse formulas using Coq functions (fast)

Reflective tactics (fast)

Need to explicitly encode binders

Need to embed features like lists

Shallow embedding

```
Definition iProp : Type :=  
  (* predicates over states *).  
Definition iAnd : iProp → iProp → iProp :=  
  (* semantic interpretation *).  
Definition iForall : ∀ A, (A → iProp) → iProp :=  
  (* semantic interpretation *).
```

Traverse formulas on the meta level (slow)

Tactics on the meta level (slow)

Reuse binders of Coq

Piggy-back on features like lists from Coq

How to embed a logic into a proof assistant

Deep embedding

```
Inductive form : Type :=  
  | iAnd: form → form → form  
  | iForall: string → form → form → form
```

Traverse formulas using Coq functions (fast)

Reflective tactics (fast)

Need to explicitly encode binders

Need to embed features like lists

Grammar of formulas fixed once and forall

Shallow embedding

```
Definition iProp : Type :=  
  (* predicates over states *).  
Definition iAnd : iProp → iProp → iProp :=  
  (* semantic interpretation *).  
Definition iForall : ∀ A, (A → iProp) → iProp :=  
  (* semantic interpretation *).
```

Traverse formulas on the meta level (slow)

Tactics on the meta level (slow)

Reuse binders of Coq

Piggy-back on features like lists from Coq

Easily extensible with new connectives

How to embed a logic into a proof assistant

Deep embedding

```
Inductive form : Type :=  
| iAnd: form → form → form  
| iForall: string → form → form → form
```

Traverse formulas using Coq functions (fast)

Reflective tactics (fast)

Need to explicitly encode binders

Need to embed features like lists

Grammar of formulas fixed once and forall

Shallow embedding

```
Definition iProp : Type :=  
(* predicates over states *).  
Definition iAnd : iProp → iProp → iProp :=  
(* semantic interpretation *).  
Definition iForall : ∀ A, (A → iProp) → iProp :=  
(* semantic interpretation *).
```

Traverse formulas on the meta level (slow)

Tactics on the meta level (slow)

Reuse binders of Coq

Piggy-back on features like lists from Coq

Easily extensible with new connectives

Context manipulation is the prime task of tactics:

Deeply embed contexts, shallowly embed the logic

Deeply embedded contexts in IPM

Visible goal in IPM:

$\vec{x} : \vec{\phi}$ Variables and pure Coq hypotheses

$\vec{H}_{\text{persistent}} : \vec{P}$ Persistent hypotheses in object logic

$\vec{H}_{\text{spatial}} : \vec{Q}$ Spatial hypotheses in object logic

R Goal in object logic

Deeply embedded contexts in IPM

Visible goal in IPM:

Propositions that enjoy $P \Leftrightarrow P * P$

$\vec{x} : \vec{\phi}$ Variables and pure Coq hypotheses

$\vec{H}_{\text{persistent}} : \vec{P}$ Persistent hypotheses in object logic

$\vec{H}_{\text{spatial}} : \vec{Q}$ Spatial hypotheses in object logic

R Goal in object logic

Deeply embedded contexts in IPM

Visible goal in IPM:

Propositions that enjoy $P \Leftrightarrow P * P$

$\vec{x} : \vec{\phi}$ Variables and pure Coq hypotheses

$\vec{H}_{\text{persistent}} : \vec{P}$ Persistent hypotheses in object logic

$\vec{H}_{\text{spatial}} : \vec{Q}$ Spatial hypotheses in object logic

R Goal in object logic

Actual Coq goal (without pretty printing):

$\vec{x}_i : \vec{\phi}_i$

$\text{of_envs } (\text{Envs } \dots \dots) \vdash R$

where:

Record envs :=

Envs { env_persistent : env iProp; env_spatial : env iProp }.

Coercion of_envs ($\Delta : \text{envs}$) : iProp :=

($\ulcorner \text{envs_wf } \Delta \urcorner * \square [*] \text{env_persistent } \Delta * [*] \text{env_spatial } \Delta \urcorner$)%I.

Deeply embedded contexts in IPM

Visible goal in IPM:

Propositions that enjoy $P \Leftrightarrow P * P$

$\vec{x} : \vec{\phi}$ Variables and pure Coq hypotheses

$\vec{H}_{\text{persistent}} : \vec{P}$ Persistent hypotheses in object logic

$\vec{H}_{\text{spatial}} : \vec{Q}$ Spatial hypotheses in object logic

R Goal in object logic

Actual Coq goal (without pretty printing):

$\vec{x}_i : \vec{\phi}_i$

$\text{of_envs } (\text{Envs } \dots \dots) \vdash R$

where:

Association list of shallowly embedded propositions

Record envs :=

Envs { env_persistent : env iProp; env_spatial : env iProp }.

Coercion of_envs ($\Delta : \text{envs}$) : iProp :=

($\ulcorner \text{envs_wf } \Delta \urcorner * \square [*] \text{env_persistent } \Delta * [*] \text{env_spatial } \Delta \urcorner$)%I.

Deeply embedded contexts in IPM

Visible goal in IPM:

$\vec{x} : \vec{\phi}$ Variables and pure Coq hypotheses

$\vec{H}_{\text{persistent}} : \vec{P}$ Persistent hypotheses in object logic

$\vec{H}_{\text{spatial}} : \vec{Q}$ Spatial hypotheses in object logic

R Goal in object logic

Propositions that enjoy $P \Leftrightarrow P * P$

Actual Coq goal (without pretty printing):

$\vec{x}_i : \vec{\phi}_i$

of_envs (Envs) $\vdash R$

where:

Association list of shallowly embedded propositions

```
Record envs :=  
  Envs { env_persistent : env iProp; env_spatial : env iProp }.  
Coercion of_envs ( $\Delta$  : envs) : iProp :=  
  (  $\ulcorner$  envs_wf  $\Delta$   $\urcorner$  *  $\square$  [*] env_persistent  $\Delta$  * [*] env_spatial  $\Delta$  )%I.
```

Folded separating conjunction

The iSplit tactic

Lemma `and_exist_sep` {A} P R (Ψ : A \rightarrow iProp) :

P * (\exists a, Ψ a) * R \rightarrow \exists a, Ψ a * P.

Proof.

`iIntros` "[HP [H Ψ HR]]".

`iDestruct` "H Ψ " as (x) "H Ψ ".

`iExists` x.

1 subgoal

M : ucmraT

A : Type

P, R : iProp

Ψ : A \rightarrow iProp

x : A

-----(1/1)

"HP" : P

"H Ψ " : Ψ x

"HR" : R

-----*

Ψ x * P

The iSplit tactic

Lemma `and_exist_sep` {A} P R (Ψ : A \rightarrow iProp) :

P * (\exists a, Ψ a) * R \rightarrow \exists a, Ψ a * P.

Proof.

`iIntros` "[HP [H Ψ HR]]".

`iDestruct` "H Ψ " as (x) "H Ψ ".

`iExists` x.

`iSplitL` "H Ψ ".

1 subgoal

M : ucmraT

A : Type

P, R : iProp

Ψ : A \rightarrow iProp

x : A

-----(1/1)

"HP" : P

"H Ψ " : Ψ x

"HR" : R

-----*

Ψ x * P

The iSplit tactic

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iExists x.`

`iSplitL "HΨ".`

2 subgoals

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

----- (1/2)

`"HΨ" : Ψ x`

-----*

`Ψ x`

----- (2/2)

`"HP" : P`

`"HR" : R`

-----*

`P`

Implementation of the iSplit tactic

Tactics implemented by reflection as mere lemmas:

Lemma `tac_sep_split` $\Delta \Delta_1 \Delta_2$ `lr js Q1 Q2` :
 `envs_split lr js Δ = Some (Δ_1, Δ_2)` \rightarrow
 $(\Delta_1 \vdash Q1) \rightarrow (\Delta_2 \vdash Q2) \rightarrow \Delta \vdash Q1 * Q2$.

Implementation of the `iSplit` tactic

Tactics implemented by reflection as mere lemmas:

Lemma `tac_sep_split` $\Delta \Delta_1 \Delta_2$ `lr js` `Q1 Q2` :
`envs_split` `lr js` $\Delta = \text{Some } (\Delta_1, \Delta_2) \rightarrow$
 $(\Delta_1 \vdash Q1) \rightarrow (\Delta_2 \vdash Q2) \rightarrow \Delta \vdash Q1 * Q2.$

Context splitting implemented as a computable Coq function

Implementation of the `iSplit` tactic

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split  $\Delta \Delta_1 \Delta_2$  lr js Q1 Q2 :  
  envs_split lr js  $\Delta = \text{Some } (\Delta_1, \Delta_2) \rightarrow$   
  ( $\Delta_1 \vdash Q1$ )  $\rightarrow (\Delta_2 \vdash Q2) \rightarrow \Delta \vdash Q1 * Q2$ .
```

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=  
  let Hs := words Hs in  
  eapply tac_sep_split with _ _ false Hs _ _;  
  [env_cbv; reflexivity ||  
    fail "iSplitL: hypotheses" Hs "not found in the context"  
    | (* goal 1 *)  
    | (* goal 2 *) ].
```

Implementation of the `iSplit` tactic

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split  $\Delta \Delta_1 \Delta_2$  lr js Q1 Q2 :  
  envs_split lr js  $\Delta = \text{Some } (\Delta_1, \Delta_2) \rightarrow$   
   $(\Delta_1 \vdash Q1) \rightarrow (\Delta_2 \vdash Q2) \rightarrow \Delta \vdash Q1 * Q2.$ 
```

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=  
  let Hs := words Hs in  
  eapply tac_sep_split with _ _ false Hs _ _;  
  [env_cbv; reflexivity ||  
    fail "iSplitL: hypotheses" Hs "not found in the context"  
    | (* goal 1 *)  
    | (* goal 2 *) ].
```

Report sensible error to the user

The iFrame tactic

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----(1/1)

`"HP" : P`

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`∃ a : A, Ψ a * P`

The iFrame tactic

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iFrame "HP".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----(1/1)

`"HP" : P`

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`∃ a : A, Ψ a * P`

The iFrame tactic

Lemma `and_exist_sep {A} P R (Ψ: A → iProp) :`

`P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.`

Proof.

`iIntros "[HP [HΨ HR]]".`

`iDestruct "HΨ" as (x) "HΨ".`

`iFrame "HP".`

`1 subgoal`

`M : ucmraT`

`A : Type`

`P, R : iProp`

`Ψ : A → iProp`

`x : A`

-----^(1/1)

`"HΨ" : Ψ x`

`"HR" : R`

-----*

`∃ a : A, Ψ a`

Implementation of the iFrame tactic

Problem: the goal is not deeply embedded, how to manipulate it?

Implementation of the iFrame tactic

Problem: the goal is not deeply embedded, how to manipulate it?

Solution: logic programming using type classes

The lemma corresponding to the tactic in Coq:

```
Class Frame (R P Q : iProp) := frame : R * Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which R is framed

Initial conclusion

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

Implementation of the `iFrame` tactic

Problem: the goal is not deeply embedded, how to manipulate it?

Solution: logic programming using type classes

The lemma corresponding to the tactic in Coq:

```
Class Frame (R P Q : iProp) := frame : R * Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which R is framed

Initial conclusion

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

Note: we support framing under binders (\exists , \forall , ...) and user defined connectives

Implementation of the iFrame tactic (2)

Consider the type class:

```
Class Frame (R P Q : iProp) := frame : R * Q ⊢ P.
```

What we want to frame



Initial conclusion

Conclusion of the new goal in which R is framed

Implementation of the iFrame tactic (2)

Consider the type class:

```
Class Frame (R P Q : iProp) := frame : R * Q ⊢ P.
```

What we want to frame

Conclusion of the new goal in which R is framed

Initial conclusion

Instances (rules of the logic program):

```
Instance frame_here R : Frame R R True.
```

```
Instance frame_sep_l R P1 P2 Q :  
  Frame R P1 Q → Frame R (P1 * P2) (Q * P2).
```

```
Instance frame_sep_r R P1 P2 Q :  
  Frame R P2 Q → Frame R (P1 * P2) (P1 * Q).
```

Implementation of the iFrame tactic (2)

Consider the type class:

```
Class Frame (R P Q : iProp) := frame : R * Q ⊢ P.
```

What we want to frame

Initial conclusion

Conclusion of the new goal in which R is framed

Instances (rules of the logic program):

```
Class MakeSep P Q PQ := make_sep : P * Q ⊢ PQ.
```

```
Instance frame_here R : Frame R R True.
```

```
Instance frame_sep_l R P1 P2 Q Q' :
```

```
Frame R P1 Q → MakeSep Q P2 Q' → Frame R (P1 * P2) Q'.
```

```
Instance frame_sep_r R P1 P2 Q Q' :
```

```
Frame R P2 Q → MakeSep P1 Q Q' → Frame R (P1 * P2) Q'.
```

```
Instance make_sep_true_l P : MakeSep True P P | 1.
```

```
Instance make_sep_true_r P : MakeSep P True P | 1.
```

```
Instance make_sep_default P Q : MakeSep P Q (P * Q) | 2.
```

Proving Hoare triples

Consider:

$$\{x \mapsto v_1 * y \mapsto v_2\} \text{swap}(x, y) \{x \mapsto v_2 * y \mapsto v_1\}$$

How to use IPM to manipulate the precondition?

Proving Hoare triples

Consider:

$$\{x \mapsto v_1 * y \mapsto v_2\} \text{swap}(x, y) \{x \mapsto v_2 * y \mapsto v_1\}$$

How to use IPM to manipulate the precondition?

Solution: define Hoare triple in terms of weakest preconditions

We let:

$$\{P\} e \{Q\} \triangleq \Box(P \multimap \text{wp } e \{Q\})$$

where $\text{wp } e \{Q\}$ gives the *weakest precondition* under which:

- ▶ all executions of e are safe
- ▶ the final state of e satisfies the postcondition Q

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
----- (1/1)  
{{ l1 ↦ v1 * l2 ↦ v2 }} (swap #l1) #l2 {{ -, l1 ↦ v2 * l2 ↦ v1 }}
```


Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

```
Proof.  
  iIntros "!# [Hl1 Hl2]".
```

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
----- (1/1)  
"Hl1" : l1 ↦ v1  
"Hl2" : l2 ↦ v2  
-----*  
WP (swap #l1) #l2 {{ -, l1 ↦ v2 * l2 ↦ v1 }}
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
iIntros "![Hl1 Hl2]".  
do 2 wp_let.
```

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
----- (1/1)  
"Hl1" : l1 ↦ v1  
"Hl2" : l2 ↦ v2  
-----*  
WP  
let: "tmp" := ! #l1 in  
#l1 ← ! #l2 ;;  
#l2 ← "tmp" {{ -, l1 ↦ v2 * l2 ↦ v1 }}
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
iIntros "![Hl1 Hl2]".  
do 2 wp_let.  
wp_load; wp_let.
```

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
----- (1/1)  
"Hl1" : l1 ↦ v1  
"Hl2" : l2 ↦ v2  
-----*  
WP #l1 ← ! #l2 ;; #l2 ← v1 {{ -, l1 ↦ v2 * l2 ↦ v1 }}
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ _, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
iIntros "![Hl1 Hl2]".  
do 2 wp_let.  
wp_load; wp_let.  
wp_load.
```

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
----- (1/1)  
"H11" : l1 ↦ v1  
"H12" : l2 ↦ v2  
-----*  
WP #l1 ← v2 ;; #l2 ← v1 {{ _, l1 ↦ v2 * l2 ↦ v1 }}
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
iIntros "![Hl1 Hl2]".  
do 2 wp_let.  
wp_load; wp_let.  
wp_load.  
wp_store.
```

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
----- (1/1)  
"Hl1" : l1 ↦ v2  
"Hl2" : l2 ↦ v2  
-----*  
WP #l2 ← v1 {{ -, l1 ↦ v2 * l2 ↦ v1 }}
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
iIntros "![Hl1 Hl2]".  
do 2 wp_let.  
wp_load; wp_let.  
wp_load.  
wp_store.  
wp_store.
```

```
1 subgoal  
Σ : gFunctors  
H : heapG Σ  
l1, l2 : loc  
v1, v2 : val  
-----(1/1)  
"Hl1" : l1 ↦ v2  
"Hl2" : l2 ↦ v1  
-----*  
l1 ↦ v2 * l2 ↦ v1
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
  "x" ← !"y";;  
  "y" ← "tmp".
```

No more subgoals.

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ -, l1 ↦ v2 * l2 ↦ v1 }}.
```

```
Proof.  
  iIntros "!# [Hl1 Hl2]".  
  do 2 wp_let.  
  wp_load; wp_let.  
  wp_load.  
  wp_store.  
  wp_store.  
  iFrame.
```

Proving swap using symbolic execution

```
Definition swap : val := λ: "x" "y",  
  let: "tmp" := !"x" in  
    "x" ← !"y";;  
    "y" ← "tmp".
```

```
Lemma swap_spec l1 l2 v1 v2 :  
  {{ l1 ↦ v1 * l2 ↦ v2 }} swap #l1 #l2  
  {{ _, l1 ↦ v2 * l2 ↦ v1 }}.
```

Proof.

```
iIntros "!# [Hl1 Hl2]".  
do 2 wp_let.  
wp_load; wp_let.  
wp_load.  
wp_store.  
wp_store.  
iFrame.
```

Qed.

Making IPM tactics modular using type classes

We want `iDestruct` "H" as "[H1 H2]" to:

- ▶ turn $H : P * Q$ into $H1 : P$ and $H2 : Q$
- ▶ turn $H : \triangleright(P * Q)$ into $H1 : \triangleright P$ and $H2 : \triangleright Q$
- ▶ turn $H : 1 \mapsto v$ into $H1 : 1 \xrightarrow{1/2} v$ and $H2 : 1 \xrightarrow{1/2} v$

Making IPM tactics modular using type classes

We want `iDestruct` "H" as "[H1 H2]" to:

- ▶ turn $H : P * Q$ into $H1 : P$ and $H2 : Q$
- ▶ turn $H : \triangleright(P * Q)$ into $H1 : \triangleright P$ and $H2 : \triangleright Q$
- ▶ turn $H : l \mapsto v$ into $H1 : l \xrightarrow{1/2} v$ and $H2 : l \xrightarrow{1/2} v$

We use type classes to achieve that:

```
Class IntoAnd (p : bool) (P Q1 Q2 : uPred M) :=
  into_and : P ⊢ if p then Q1 ∧ Q2 else Q1 * Q2.
Instance into_and_sep p P Q : IntoAnd p (P * Q) P Q.
Instance into_and_and P Q : IntoAnd true (P ∧ Q) P Q.
Instance into_and_later p P Q1 Q2 : IntoAnd p P Q1 Q2 → IntoAnd p (▷ P) (▷ Q1) (▷ Q2).
Instance into_and_mapsto l q v : IntoAnd false (l ↦{q} v) (l ↦{q/2} v) (l ↦{q/2} v).

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

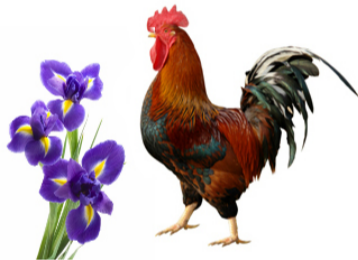
IPM in summary

- ▶ Contexts are deeply embedded
- ▶ Context manipulation is done via **computational reflection**
- ▶ IPM tactics are just Coq lemmas
- ▶ **Type classes** are used to make the tactics more general
- ▶ **Ltac** is used to provide an end-user syntax and error reporting



IPM in summary

- ▶ Contexts are deeply embedded
- ▶ Context manipulation is done via **computational reflection**
- ▶ IPM tactics are just Coq lemmas
- ▶ **Type classes** are used to make the tactics more general
- ▶ **Ltac** is used to provide an end-user syntax and error reporting



These ideas are hopefully applicable to other object logics

In the paper and Coq formalization

- ▶ Detailed description of the implementation
 - ▶ Verification of concurrent algorithms using IPM
 - ▶ Formalization of unary and binary logical relations
 - ▶ Proving logical refinements
- } Shows that IPM scales
-

Thank you!

Want a 'proof mode' for another logic, talk to us!

Download Iris at <http://iris-project.org/>

Talks about Iris this week:

- ▶ Wed 15:35 @ POPL: Krogh-Jespersen, *Svendsen* and Birkedal
A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic
- ▶ Sat 9:00 @ CoqPL: *Krebbers*
Demonstration of the Iris separation logic in Coq
- ▶ Sat 10:30 @ CoqPL: *Timany*, *Krebbers* and Birkedal
Logical Relations in Iris

Coq wish list

- ▶ Data types in Ltac
- ▶ Side-effecting tactics that can return a value
- ▶ More expressive parsing mechanism of tactic notations
- ▶ Exception handling in Ltac to enable better error message generation
- ▶ Opt-out from backtracking Ltac semantics

