Mechanized proofs in higher-order separation logic

Robbert Krebbers

Delft University of Technology, The Netherlands

February 5, 2019 @ Vrije Universiteit, Amsterdam, The Netherlands

Tactic-style proofs (as in LCF/Coq/HOL/*etc.*) have shown to be effective in large-scale proof developments (CompCert, Four color, Feit-Thompson, Kepler, . . .)

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

```
1 subgoal
A : Type
P, Q : Prop
Ψ : A → Prop
H1 : P
H2 : ∃ a : A, Ψ a
H3 : Q
____________________________(1/1)
Q ∧ (∃ a : A, P ∧ Ψ a)
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

Context

```
1 subgoal
A : Type
P, Q : Prop
Ψ : A → Prop
H1 : P
H2 : ∃ a : A, Ψ a
H3 : Q
____________________(1/1)
Q ∧ (∃ a : A, P ∧ Ψ a)
```

Goal

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

```
1 subgoal
A : Type
P, Q : Prop
Ψ : A → Prop
H1 : P
H2 : ∃ a : A, Ψ a
H3 : Q
____________________________(1/1)
Q ∧ (∃ a : A, P ∧ Ψ a)
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

```
1 subgoal
A : Type
P, Q : Prop
Ψ : A → Prop
H1 : P
x : A
H2 : Ψ x
H3 : Q
_______________________(1/1)
Q ∧ (∃ a : A, P ∧ Ψ a)
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

```
1 subgoal
A : Type
P, Q : Prop
Ψ : A → Prop
H1 : P
x : A
H2 : Ψ x
H3 : Q
____________________________(1/1)
Q ∧ (∃ a : A, P ∧ Ψ a)
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

```
2 subgoals
A : Type
P, Q : Prop
Ψ : A → Prop
H1 : P
x : A
H2 : Ψ x
H3 : Q
____________________(1/2)
Q
____________________(2/2)
∃ a : A, P ∧ Ψ a
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  destruct H2 as [x H2].
  split.
  - assumption.
  - exists x.
    auto.
Qed.
```

```
No more subgoals.
```

Basic example of tactic-style proofs in Coq

```
Lemma test {A} (P Q : Prop) (Ψ : A → Prop) :
  P ∧ (∃ a, Ψ a) ∧ Q → Q ∧ ∃ a, P ∧ Ψ a.
Proof.
  intros [H1 [H2 H3]].
  by firstorder (* automate this *).
Qed.
```

Scales in practice

- High-level tactics for arithmetic, Prolog-style search, algebra, ...
- Compact syntax for combining tactics (ssreflect)
- Tactic programming (using ML, Ltac, ...)

Tactic-style proofs for other logics,
like **separation logic**

Separation logic [O'Hearn, Reynolds, Yang; CSL'01]

Propositions P, Q denote ownership of resources

Separating conjunction $P * Q$:
The resources consists of separate parts satisfying P and Q

Basic example:

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

the $*$ ensures that x and y are different memory locations

Why is separation logic useful?

Separation logic is very useful:

- It provides a high level of modularity
- It scales to fancy PL features like concurrency

Just in Coq, there is an ever growing collection of separation logics:

- Bedrock
- CFML
- Charge!
- CHL
- FCSL
- Iris
- VST
- ...

Problem:

Cannot reuse the tactics/context of the proof assistant when reasoning in an embedded logic like separation logic

Goal of this talk

Enable tactic-style proofs in separation logic

- Extend Coq with named proof contexts for separation logic
- Tactics for introduction and elimination of all connectives of separation logic ...
- ...that can be used in Coq's mechanisms for automation/tactic programming
- Implemented without modifying Coq (using reflection, type classes and Ltac)

Papers: Iris Proof Mode / MoSeL

For the Iris logic

Interactive Proofs in Higher-Order Concurrent Separation Logic

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

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

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

Abstract

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

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

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

instance, they include separating conjunction of separation logic for reasoning about mutable data structures, invariants for reasoning about sharing, guarded recursion for reasoning about various forms of recursion, and higher-order quantification for giving generic modular specifications to libraries.

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

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

To remedy this situation, we present a so-called *proof mode* that extends the Coq proof assistant with (spatial and non-spatial) named contexts for managing the hypotheses of the object logic. We show that using our proof mode we can make reasoning in the embedded

POPL'17

For many separation logics

MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic

ROBBERT KREBBERS, Delft University of Technology, The Netherlands
JACQUES-HENRI JOURDAN, LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay, France
RALF JUNG, MPI-SWS, Germany
JOSEPH TASSAROTTI, Carnegie Mellon University, USA
JAN-OLIVER KAISER, MPI-SWS, Germany
AMIN TIMANY, imec-Distrinet, KU Leuven, Belgium
ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France
DEREK DREYER, MPI-SWS, Germany

A number of tools have been developed for carrying out separation-logic proofs mechanically using an interactive proof assistant. One of the most advanced such tools is the Iris Proof Mode (IPM) for Coq, which offers a rich set of tactics for making separation-logic proofs look and feel like ordinary Coq proofs. However, IPM is tied to a particular separation logic (namely, Iris), thus limiting its applicability.

In this paper, we propose MoSeL, a general and extensible Coq framework that brings the benefits of IPM to a much larger class of separation logics. Unlike IPM, MoSeL is applicable to both affine and linear separation logics (and combinations thereof), and provides generic tactics that can be easily extended to account for the bespoke connectives of the logics with which it is instantiated. To demonstrate the effectiveness of MoSeL, we have instantiated it to provide effective tactical support for interactive and semi-automated proofs in six very

ICFP'18

Iris [Jung, Krebbers *et al.*; POPL'15, ICFP'16, ESOP'17, JFP'18]

A general, language-independent, framework for modeling your own domain specific higher-order separation logics

Iris [Jung, Krebbers *et al.*; POPL'15, ICFP'16, ESOP'17, JFP'18]

A general, language-independent, framework for modeling your own domain specific higher-order separation logics

- **General:** unifies the reasoning principles in many other logics

Iris [Jung, Krebbers *et al.*; POPL'15, ICFP'16, ESOP'17, JFP'18]

A general, language-independent, framework for modeling your own domain specific higher-order separation logics

- **General:** unifies the reasoning principles in many other logics
- **Language-independent:** parameterized by the language

Iris [Jung, Krebbers *et al.*; POPL'15, ICFP'16, ESOP'17, JFP'18]

A general, language-independent, framework for modeling your own domain specific higher-order separation logics

- **General:** unifies the reasoning principles in many other logics
- **Language-independent:** parameterized by the language
- **Modeling logics:** can be used to model domain specific logics
  - iGPS for weak memory [ECOOP'17]
  - RustBelt's lifetime logic [POPL'18]
  - ReLoC for program refinements [LICS'18]
  - Iron for resource management [POPL'19]

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

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

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iInt
  iDes
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Lemma in the Iris logic

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ 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
Ψ : A → iProp
x : A
____________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
____________________________∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ 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
Ψ : A → iProp
x : A
____________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
____________________________∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

∗ means: resources should be split

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ 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
Ψ : A → iProp
x : A
____________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
____________________________∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

∗ means: resources should be split

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

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

The hypotheses for the left conjunct

```
Qed.
```

```
2 subgoals
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
____________________________(1/2)
"H3" : Q
____________________________∗
Q

____________________________(2/2)
"H1" : P
"H2" : Ψ x
____________________________∗
∃ a : A, P ∗ Ψ a
```

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

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

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

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

We can also solve this lemma automatically

```
No more subgoals.
```

Scaling up

Iris Proof Mode scales and is used for any project involving Iris today because:

- **Proofs have the look and feel of ordinary Coq proofs**
  For many Coq tactics `tac`, it has a variant `iTac`

Scaling up

Iris Proof Mode scales and is used for any project involving Iris today because:

- **Proofs have the look and feel of ordinary Coq proofs**
  For many Coq tactics `tac`, it has a variant `iTac`
- **Support for advanced features of separation logic**
  Higher-order quantification, invariants, ghost state, later $\triangleright$ modality, ...

Scaling up

Iris Proof Mode scales and is used for any project involving Iris today because:

- **Proofs have the look and feel of ordinary Coq proofs**
  For many Coq tactics `tac`, it has a variant `iTac`
- **Support for advanced features of separation logic**
  Higher-order quantification, invariants, ghost state, later $\triangleright$ modality, ...
- **Integration with tactics for proving programs**
  Symbolic execution tactics for weakest preconditions

Scaling up

Iris Proof Mode scales and is used for any project involving Iris today because:

- **Proofs have the look and feel of ordinary Coq proofs**
  For many Coq tactics `tac`, it has a variant `iTac`
- **Support for advanced features of separation logic**
  Higher-order quantification, invariants, ghost state, later $\triangleright$ modality, ...
- **Integration with tactics for proving programs**
  Symbolic execution tactics for weakest preconditions
- **Tactic programming**
  One can combine/program with IPM tactics using Coq's Ltac like ordinary Coq tactics

Implementation of separation logic tactics

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	Shallow embedding

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

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)

How to embed a logic into a proof assistant?

Deep embedding	Shallow embedding

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

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)
Need to explicitly encode binders	Reuse binders of Coq
Need to embed features like lists	Piggy-back on features like lists from Coq

How to embed a logic into a proof assistant?

Deep embedding	Shallow embedding

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

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)
Need to explicitly encode binders	Reuse binders of Coq
Need to embed features like lists	Piggy-back on features like lists from Coq
Grammar of formulas fixed once and forall	Easily extensible with new connectives

How to embed a logic into a proof assistant?

Deep embedding	Shallow embedding

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

Traverse formulas using Coq functions (fast)	Traverse formulas on the meta level (slow)
Reflective tactics (fast)	Tactics on the meta level (slow)
Need to explicitly encode binders	Reuse binders of Coq
Need to embed features like lists	Piggy-back on features like lists from Coq
Grammar of formulas fixed once and forall	Easily extensible with new connectives

Context manipulation is the prime task of tactics:
Deeply embedded contexts, shallowly embedded logic ⇒ Best of both worlds

Deeply embedded contexts in IPM (1)

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

Deeply embedded contexts in IPM (1)

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ 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
Ψ : A → iProp
x : A
____________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
____________________________∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

Deeply embedded contexts in IPM (1)

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  Unset Printing Notations.
```

```
1 subgoal
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
____________________________(1/1)
"H1" : P
"H2" : Ψ x
____________________________∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

Notation for deeply embedded context

Deeply embedded contexts in IPM (1)

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  Unset Printing Notations.
```

```
1 subgoal
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
_____________________(1/1)
envs_entails (Envs Enil
 (Esnoc (Esnoc (Esnoc Enil
   (String (Ascii false
     false false true false
     false true false)
   (String (Ascii true
     false false false true
     true false false)
   EmptyString)) P)
   ...
```

Deeply embedded contexts in IPM (2)

Visible goal (with pretty printing):

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

Π Spatial separation logic hypotheses

---$*$

R Separation logic goal

Deeply embedded contexts in IPM (2)

Visible goal (with pretty printing):

$$\begin{array}{ll} \vec{\mathtt{x}} : \vec{\phi} & \text{Variables and pure Coq hypotheses} \\ \hline \Pi & \text{Spatial separation logic hypotheses} \\ \hline R & \text{Separation logic goal} \end{array} *$$

Actual Coq goal (without pretty printing):

$$\begin{array}{l} \vec{\mathtt{x}} : \vec{\phi} \\ \hline \Pi \Vdash Q \end{array}$$

Where:

$$\Pi \Vdash Q \triangleq \ast \Pi \vdash Q$$

Implementation of the iSplitL/iSplitR tactic

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ lr js Q₁ Q₂ :
  envs_split lr js Π = Some (Π₁,Π₂) →
  (Π₁ ⊢ Q₁) → (Π₂ ⊢ Q₂) → Π ⊢ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Implementation of the iSplitL/iSplitR tactic

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π₁ Π₂ lr js Q₁ Q₂ :
  envs_split lr js Π = Some (Π₁,Π₂) →
  (Π₁ ⊩ Q₁) → (Π₂ ⊩ Q₂) → Π ⊩ Q₁ ∗ Q₂.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Implementation of the iSplitL/iSplitR tactic

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π1 Π2 lr js Q1 Q2 :
  envs_split lr js Π = Some (Π1,Π2) →
  (Π1 ⊢ Q1) → (Π2 ⊢ Q2) → Π ⊢ Q1 ∗ Q2.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=
  let Hs := words Hs in
  let Hs := eval vm_compute in (INamed <$> Hs) in
  eapply tac_sep_split with _ _ Left Hs _ _;
    [pm_reflexivity ||
      fail "iSplitL: hypotheses" Hs "not found"
    | (* goal 1 *)
    | (* goal 2 *) ].
```

Implementation of the iSplitL/iSplitR tactic

Tactics implemented by reflection as mere lemmas:

```
Lemma tac_sep_split Π Π1 Π2 lr js Q1 Q2 :
  envs_split lr js Π = Some (Π1,Π2) →
  (Π1 ⊢ Q1) → (Π2 ⊢ Q2) → Π ⊢ Q1 ∗ Q2.
```

$$\frac{\Pi_1 \Vdash Q_1 \qquad \Pi_2 \Vdash Q_2}{\Pi_1, \Pi_2 \Vdash Q_1 * Q_2}$$

Context splitting implemented as a computable Coq function

Ltac wrappers around the reflective tactic:

```
Tactic Notation "iSplitL" constr(Hs) :=
  let Hs := words Hs in
  let Hs := eval vm_compute in (INamed <$> Hs) in
  eapply tac_sep_split with _ _ Left Hs _ _;
    [pm_reflexivity ||
      fail "iSplitL: hypotheses" Hs "not found"
    | (* goal 1 *)
    | (* goal 2 *) ].
```

Report sensible error to the user

Proving program specifications

Proving program specifications

Consider:

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

How to use IPM to manipulate the precondition?

Proving program specifications

Consider:

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

How to use IPM to manipulate the precondition?

Solution: define Hoare triple in terms of weakest preconditions

We let:

$$\{P\}\ e\ \{Q\} \triangleq \square(P \mathrel{-\!\!*} \mathrm{wp}\ e\ \{Q\})$$

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

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.
```

```
1 subgoal
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]".
  do 2 wp_let.
  wp_load; wp_let.
  wp_load.
  wp_store.
  wp_store.
  iFrame.
Qed.
```

```
1 subgoal
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.
  wp_load; wp_let.
  wp_load.
  wp_store.
  wp_store.
  iFrame.
Qed.
```

```
1 subgoal
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.
  wp_load.
  wp_store.
  wp_store.
  iFrame.
Qed.
```

```
1 subgoal
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.
  wp_store.
  wp_store.
  iFrame.
Qed.
```

```
1 subgoal
l1, l2 : loc
v1, v2 : val
____________________________(1/1)
"Hl1" : l1 ↦ v1
"Hl2" : 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.
  wp_store.
  iFrame.
Qed.
```

```
1 subgoal
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.
  iFrame.
Qed.
```

```
1 subgoal
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".
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.
```

```
No more subgoals.
```

Projects that use IPM

The approach in Iris Proof Mode actually scales beyond `swap`

- Non-determinism in C [Frumin *et al.*; ESOP'19]
- Time complexity [Mével *et al.*; ESOP'19]
- Resource obligations [Bizjak *et al.*; POPL'19]
- The Rust type system [Jung *et al.*; POPL'18]
- Haskell's ST monad [Timany *et al.*; POPL'18]
- Program refinements [Tassarotti *et al.*; ESOP'17 & Frumin *et al.*; LICS'18]
- Iris's meta theory [Krebbers *et al.*; ESOP'17]
- Weak memory [Kaiser *et al.*; ECOOP'17]
- Object capabilities [Swasey *et al.*; OOPSLA'17]

Next step: Going beyond Iris

Making Iris Proof Mode independent of Iris

It sounds easy [Krebbers *et al.*; POPL'17]:

[. . .] we believe that our proof mode is very generic, and can be applied to a variety of different embedded logics [. . .]

Making Iris Proof Mode independent of Iris

It sounds easy [Krebbers *et al.*; POPL'17]:

[. . .] we believe that our proof mode is very generic, and can be applied to a variety of different embedded logics [. . .]

But doing it generally turned out to be be more challenging

Problem #1: Iris propositions are affine

In Iris you may "forget" about resources:

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

Problem #1: Iris propositions are affine

In Iris you may "forget" about resources:

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

Due to the **affinity axiom** $P * Q \vdash Q$, which is hard-wired into many tactics:

`iClear`

$$\frac{\Pi \Vdash Q}{\Pi, P \Vdash Q}$$

`iAssumption`

$$\Pi, P \Vdash P$$

Problem #1: Iris propositions are affine

In Iris you may "forget" about resources:

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

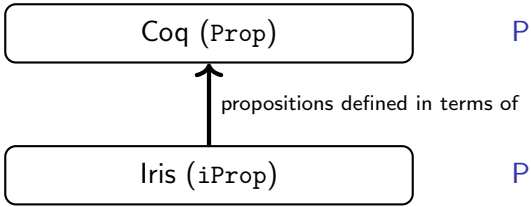
Due to the **affinity axiom** $P * Q \vdash Q$, which is hard-wired into many tactics:

$$\frac{\texttt{iClear}\quad \Pi \Vdash Q}{\Pi, P \Vdash Q} \qquad \frac{\texttt{iAssumption}}{\Pi, P \Vdash P}$$

Not having the affinity axiom is useful: precise accounting of resources

Challenge: How to disentangle the affinity axiom from the Iris tactics?

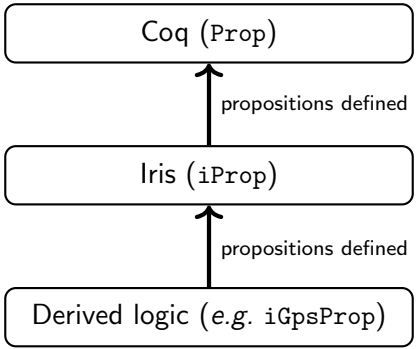
Problem #2: No tactical support for derived logics



Proof using standard Coq tactics

Proof using Iris tactics

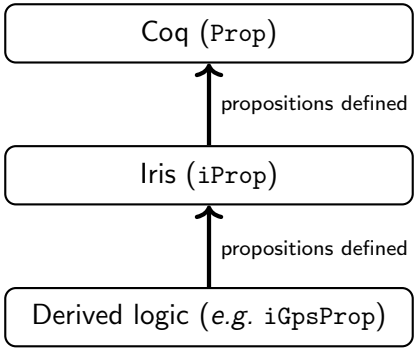
Problem #2: No tactical support for derived logics



Proof using standard Coq tactics

Proof using Iris tactics

Problem #2: No tactical support for derived logics

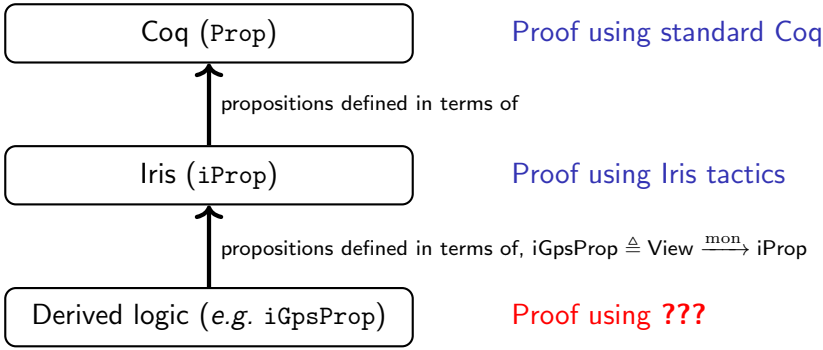


Proof using standard Coq tactics

Proof using Iris tactics

Proof using **???**

Problem #2: No tactical support for derived logics



Challenge: How to reason in logics defined in terms of another

MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic in Coq [Krebbers *et al.*; ICFP'18]

New features w.r.t. Iris Proof Mode:

- MoSeL is parameterized by a general abstraction of separation logic
- MoSeL supports general and affine separation logics, and combinations thereof
- MoSeL supports reasoning in derived separation logics
- MoSeL can be fine-tuned for each logic using type classes

MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic in Coq [Krebbers *et al.*; ICFP'18]

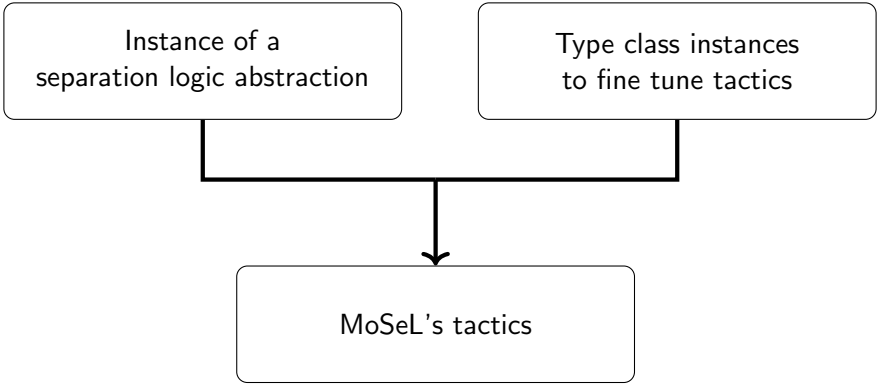
New features w.r.t. Iris Proof Mode:

- MoSeL is parameterized by a general abstraction of separation logic
- MoSeL supports general and affine separation logics, and combinations thereof
- MoSeL supports reasoning in derived separation logics
- MoSeL can be fine-tuned for each logic using type classes

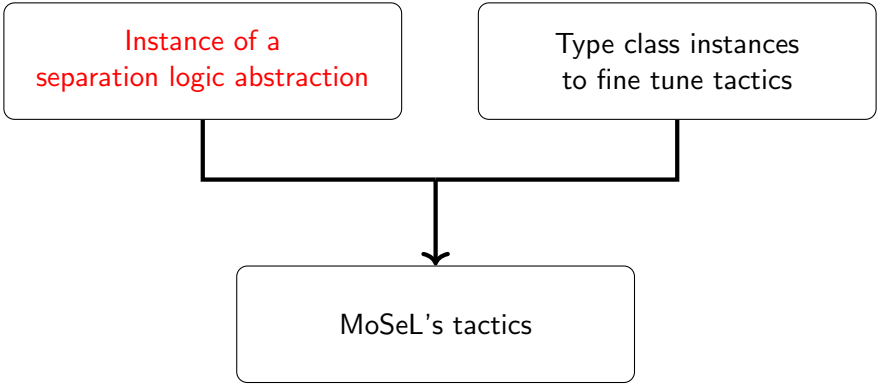
MoSeL is usable in practice: we used it on very different existing separation logics

CFML CHL Fairis iGPS Iris Iron

Setup of MoSeL



Setup of MoSeL



How to abstract over separation logics?

BI logics: An abstract interface for separation logics

A **Bunched Implications (BI) logic** [O'Hearn&Pym,99] is a preorder $(Prop, \vdash)$ with:

- Operations $\mathsf{True}, \mathsf{False}, \wedge, \vee, \Rightarrow, \forall, \exists$ satisfying the axioms of intuitionistic logic
- Operations $\mathsf{emp}, *, \mathrel{-\!\!*}$ satisfying:

$$\begin{aligned} \mathsf{emp} * P &\dashv\vdash P \\ P * Q &\vdash Q * P \\ (P * Q) * R &\vdash P * (Q * R) \end{aligned}$$

$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

BI logics: An abstract interface for separation logics

A **Bunched Implications (BI) logic** [O'Hearn&Pym,99] is a preorder $(Prop, \vdash)$ with:

- Operations $\mathsf{True}, \mathsf{False}, \wedge, \vee, \Rightarrow, \forall, \exists$ satisfying the axioms of intuitionistic logic
- Operations $\mathsf{emp}, *, -\!\!*$ satisfying:

$$\begin{aligned} \mathsf{emp} * P &\dashv\vdash P \\ P * Q &\vdash Q * P \\ (P * Q) * R &\vdash P * (Q * R) \end{aligned}$$

$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \frac{P * Q \vdash R}{\overline{P \vdash Q -\!\!* R}}$$

```
Structure bi := Bi {
  bi_car     :> Type;
  bi_entails : bi_car → bi_car → Prop;
  bi_forall  : ∀ A, (A → bi_car) → bi_car;
  bi_sep     : bi_car → bi_car → bi_car;
  (* other separation logic operators and axioms *)
}.
```

Proofs in MoSeL

Proofs in a specific logic:

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

Proofs for all logics:

```
Lemma test {PROP : bi} {A} (P Q : PROP) (Ψ : A → PROP) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  – iAssumption.
  – iExists x.
    iFrame.
Qed.
```

Proofs in MoSeL

Proofs in a specific logic:

```
Lemma test {A} (P Q : iGpsProp) (Ψ : A → iGpsProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
```

Lemma for another logic than Iris

```
    iAssumption.
  − iExists x.
    iFrame.
Qed.
```

Proofs for all logics:

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

Proofs in MoSeL

Proofs in a specific logic:

```
Lemma test {A} (P Q : iGpsProp) (Ψ : A → iGpsProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
```

Lemma for another logic than Iris

```
    iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Proofs for all logics:

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

Lemma universally quantified in the BI logic

Addressing challenge #1:
Disentangling the affinity axiom

$$P * Q \vdash Q$$

A poor man's solution

Make two versions of the tactics

1. For affine logics (like Iris and iGPS)
2. For non-affine logics (like CFML and CHL)

A poor man's solution

Make two versions of the tactics

1. For affine logics (like Iris and iGPS)
2. For non-affine logics (like CFML and CHL)

Problems:

- Duplicate work/maintenance
- Some logics mix affine and non-affine propositions, for example:

GC locations (affine)	**Non-GC locations** (not affine)
$\ell \mapsto_{\mathsf{gc}} v$	$\ell \mapsto v$

(Another example in [Tassarotti *et al.*; ESOP'17])

Key idea

- **Don't**: classify whether the whole logic is affine
- **Do:** classify whether individual propositions are affine

Classifying whether propositions are affine

Affine propositions:

$$\mathrm{affine}(P) \triangleq P \vdash \mathrm{emp}$$

(propositions that can be "thrown away")

The new tactics:

$$\frac{\texttt{iClear}\quad \Pi \Vdash Q \qquad \mathrm{affine}(P)}{\Pi, P \Vdash Q}$$

$$\frac{\texttt{iAssumption}\quad \mathrm{affine}(\Pi)}{\Pi, Q \Vdash Q}$$

Classifying whether propositions are affine in Coq

A new type class:

```
Class Affine {PROP : bi} (Q : PROP) := affine : Q ⊢ emp.
```

Instances:

- Tell MoSeL that specific connectives are affine:

```
Instance mapsto_gc_affine l v : Affine (l ↦gc v).
```

- Capture that affine propositions are closed under most connectives:

```
Instance sep_affine {PROP : bi} (P Q : bi) :
  Affine P → Affine Q → Affine (P ∗ Q).
```

MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic in Coq

What about modalities?

The affine modality

The **affine modality**:

$$\begin{aligned} \langle \mathit{affine} \rangle\, P \quad &\triangleq \quad P \wedge \mathsf{emp} \\ &\approx \quad \text{“}P \text{ holds using just affine resources”} \end{aligned}$$

This modality is useful for reasoning about affinity

The affine modality

The **affine modality**:

$$\begin{aligned} \langle \mathit{affine} \rangle\, P &\triangleq P \wedge \mathsf{emp} \\ &\approx \text{“}P \text{ holds using just affine resources”} \end{aligned}$$

This modality is useful for reasoning about affinity

- Can be used to turn any proposition into an affine version, *e.g.*
  A wand that can be dropped $\langle \mathit{affine} \rangle\, (P \mathrel{-\!\!*} Q)$

The affine modality

The **affine modality**:

$$\begin{aligned} \langle \mathit{affine} \rangle\, P &\triangleq P \wedge \mathsf{emp} \\ &\approx \text{“}P \text{ holds using just affine resources”} \end{aligned}$$

This modality is useful for reasoning about affinity

- Can be used to turn any proposition into an affine version, *e.g.*
  A wand that can be dropped $\langle \mathit{affine} \rangle\, (P \mathrel{-\!\!*} Q)$
- Commutes with most operators, *e.g.*
  $\langle \mathit{affine} \rangle\, (P \vee Q) \dashv\vdash \langle \mathit{affine} \rangle\, P \vee \langle \mathit{affine} \rangle\, Q$

The affine modality

The **affine modality**:

$$\begin{aligned} \langle \mathit{affine} \rangle\, P &\triangleq P \wedge \mathsf{emp} \\ &\approx \text{“}P \text{ holds using just affine resources”} \end{aligned}$$

This modality is useful for reasoning about affinity

- Can be used to turn any proposition into an affine version, *e.g.*
  A wand that can be dropped $\langle \mathit{affine} \rangle\, (P \mathrel{-\!\!*} Q)$
- Commutes with most operators, *e.g.*
  $\langle \mathit{affine} \rangle\, (P \vee Q) \dashv\vdash \langle \mathit{affine} \rangle\, P \vee \langle \mathit{affine} \rangle\, Q$
- Gives rise to an alternative classification of affine propositions
  $\mathsf{affine}(P) \quad \text{iff} \quad P \vdash \langle \mathit{affine} \rangle\, P$

The idea of carving out classes of propositions and defining their corresponding modalities is widely applicable:

- Persistent propositions $\boxdot$
- Intuitionistic propositions $\Box$
- Absorbing propositions $\langle absorb \rangle$
- Timeless propositions (in step-indexed logics) $\triangleright, \diamond$
- Objective propositions (in iGPS) $\langle obj \rangle, \langle subj \rangle$
- Normal propositions (in CFML) $\langle normal \rangle$
- ...

The ICFP'18 paper shows how to modularly deal with such classes and use them in general tactics

Thanks to

My coauthors on the IPM/MoSeL papers:

Ralf Jung

Jacques-Henri Jourdan

Amin Timany

Joseph Tassarotti

Jan-Oliver Kaiser

Arthur Charguéraud

Derek Dreyer

Lars Birkedal

And all that have contributed by using IPM/MoSeL in one way or another

Thank you!

Download MoSeL/Iris at `http://iris-project.org/`

Advertisement. I am currently looking for a PhD student (4 years)

Topics: Separation logic for multilingual programs, asynchronous I/O, non-functional properties, verified compilation, ...

Interested/Know someone? Get in touch!