Chapter 1

Background on Iris

- Iris is …
- We will use a central example to explain Iris
- We start by defining the example
- Then show a program to proof about
- Show parts of a proof
- Explain why this does not work for more complicated versions

Iris is a separation logic [Jun+15; Jun+16; Kre+17; Jun+18]. Propositions can be seen as predicates over resources, *e.g.,* heaps. Thus, there are a number of extra logical connectives such as `P * Q`, which represents that `P` and `Q` split up the resources into two disjoints in which they respectively hold. Moreover, hypotheses in our logic can often be used only once when proving something, they represent resources that we consume when used. To be able to reason in this logic in Coq a tactics' language has been added to Coq called the Iris Proof Mode (IPM) [KTB17; Kre+18].

1.1 Separation logic

- We want to be able to reason about memory, heaps
- Use a logic that has extra connectives for talking about memory
- Most important, points to, $\mapsto$
- Picture of memory with $l \mapsto x$ next to it
- $l \mapsto x$ means there is only one location in memory, l and it has value x

- $\wedge$ now no longer works
- introduce $\star$
- Another picture with logic next to it
- Describe rules of $\star$

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

$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \; *\text{-MONO}$$

- This does not include $P \vdash P * P$
- This is called affine
- Note about linear logics

1.2 Writing specifications of programs

- We will write our programs in heaplang, …
- Start by verifying simple program

```
Definition copy_singleton : val :=
  λ: "l", let: "x" := !"l" in
          SOME (Alloc "x").
```

- We now want to specify what happens to the memory when the program executes
- Use texan triples, similar to hoare triples

```
Lemma singleton_spec (x : val) (l : loc) :
  {{{ l ↦ (x, NONE) }}}
    singleton #l
  {{{ v, RET v; v ↦ (x, NONE) * l ↦ (x, NONE) }}}.
```

- Start with a description of the memory before copy singleton, precondition
- program to execute,
- `#l` transforms a location into a value in heaplang
- describe post condition

1.3 Proof rules

- Describe more parts of the logic, like persistent, later and magic wand

1.4 Proofs

- Maybe show a proof in Coq?

1.5 How does Iris actually work

- I don't really know yet

1.6 Contexts

- Iris uses a named context instead of the entailment
- `env A` is a list of pairs from identifiers to `A`.

```
Inductive ident :=
  | IAnon : positive → ident
  | INamed :> string → ident.

Record envs (PROP : bi) := Envs {
  env_persistent : env PROP;
  env_spatial : env PROP;
  env_counter : positive;
}.
```

TODO: Explain *Prop*somewhere or just use *iProp*

TODO: I am simplifying the environments here, should I do that?

- Identifiers are either anonymous or named
- Environments are maps from identifiers to values
- The final context is two environments of propositions and a counter
- The first environment is the persistent context
- The second environment is the spatial context
- The two environments can't have overlapping identifiers
- The counter is used to always be able to generate a fresh anonymous identifier
- Semantics of the contexts is

```
Definition of_envs {PROP : bi}
    (Γp Γs : env PROP) : PROP :=
  (□ [∧] Γp ∧ [∗] Γs)%I.
```

Question: Should I differentiate between □ and <pers>

- The persistent environment is combined with $\wedge$ and surrounded by a $\square$.
- The spatial environment is combined with $*$
- We can now write our entailment as

```
Definition envs_entails {PROP : bi}
    (Δ : envs PROP) (Q : PROP) : Prop :=
  of_envs (env_intuitionistic Δ) (env_spatial Δ) ⊢ Q.
```

- This is represented in the proof state as

```
P, Q, R: iProp
============
"HP" : P
------------□
"HR" : Q
------------*
R
```

- P is a persistent proposition
- Q is a spatial proposition
- We need to proof R

1.7 Tactics

- The proof rules are hard to use with the context
- Define Lemma's that work with the context
- These allow us to define our tactics easily

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

- Introduces a magic wand
- Add introduced proposition to the spatial context
- The condition we need to satisfy is a Coq function that resolves to Q with P added to the context
- If `i` already exists in the context, we have to proof False
- Tactics now just process the arguments and call necessary Lemma's

The tactics the IPM adds are build to replicate many of the behaviors of the Coq tactics while manipulating the Iris contexts. In the next section we will show how the Iris variant of the **`intros`** tactic works.

1.8 `iIntros` example

`iIntros` is based on the Coq **`intros`** tactic. The Coq **`intros`** tactic makes use of a domain specific language (DSL) for quickly introducing different logical connective. In Iris this concept was adopted for the `iIntros` tactic, but adopted to the Iris contexts. Also, a few expansions, as inspired by ssreflect [HKP97; GMT16], were added to perform other common initial proof steps such as **`simpl`**, **`done`** and others. We will show a few examples of how `iIntros` can be used to help prove lemmas.

We begin with a lemma about the magic wand. The magic wand can be seen as the implication of separation logic which also takes into account the separation of resources.

$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R} \mathrel{-\!\!*}\text{-Intro} \qquad \frac{P \wedge Q \vdash R}{P \vdash Q \to R} \to\text{-Intro}$$

Thus, where a normal implication introduction adds the left-hand side to the Coq context, the magic wand adds the left-hand side to the spatial resource context.

TODO: Rewrite when I have a solid explanation of the Iris contexts

```
P, R: iProp
============
------------*
P -* R -* P
```

When using `iIntros "HP HR"`, the proof state is transformed into the following state.

```
P, R: iProp
============
"HP" : P
"HR" : R
------------*
P
```

We have introduced the two separation logic propositions into the spatial context. This does not only work on the magic wand, we can also use this to introduce more complicated statements. Take the following proof state,

```
P: nat → iProp
============================================
--------------------------------------------*
∀ x : nat, (∃ y : nat, P x ∗ P y) ∨ P 0 -∗ P 1
```

It consists of a universal quantification, an existential quantification, a conjunction and a disjunction. We can again use one application of `iIntros` to introduce and eliminate the premise. `iIntros "%x [[%y [Hx Hy]] | H0]"` takes the proof to the following state of two goals

```
(1/2)
P: nat → iProp
x, y: nat
==================
"Hx" : P x
"Hy" : P y
------------------*
P 1

(2/2)
P: nat → iProp
x: nat
==================
"H0" : P 0
------------------*
P 1
```

The intro pattern consists of multiple sub intro patterns. Each sub intro pattern starts with a forall introduction or wand introduction. We then interpret the intro pattern for the introduced hypothesis. They can have the following interpretations:

- `"H"` represents renaming a hypothesis. The name given is used as the name of the hypothesis in the spatial context.
- `"%H"` represents pure elimination. The introduced hypothesis is interpreted as a Coq hypothesis, and added to the Coq context.
- `"[IPL | IPR]"` represents disjunction elimination. We perform a disjunction elimination on the introduced hypothesis. Then, we apply the two included intro patterns two the two cases created by the disjunction elimination.
- `"[IPL IPR]"` represents separating conjunction elimination. We perform a separating conjunction elimination. Then, we apply the two included intro patterns two the two hypotheses by the separating conjunction elimination.
- `"[%x IP]"` represents existential elimination. If first element of a separating conjunction pattern is a pure elimination we first try to eliminate an exists in the hypothesis and apply the included intro pattern on the resulting hypothesis. If that does not succeed we do a conjunction elimination.

Thus, we can break down `iIntros "%x [[%y [Hx Hy]] | H0]"` into its components. We first forall introduce or first sub intro pattern `"%x"` and then perform the second case, introduce a pure Coq variable for the `∀ x : nat`. Next we wand introduce for the second sub intro pattern, `"[[%y [Hx Hy]] | H0]"` and interpret the outer pattern. it is the third case and eliminates the disjunction, resulting in two goals. The left patterns of the seperating conjunction pattern eliminates the exists and adds the `y` to the Coq context. Lastly, `"[Hx Hy]"` is the fourth case and eliminates the seperating conjunction in the Iris context by splitting it into two assumptions `"Hx"` and `"Hy"`.

There are more patterns available to introduce more complicated goals, these can be found in a paper written by Krebbers, Timany, and Birkedal [KTB17].

1.9 Inductive predicates

- Sometimes you want to define a statement about the memory that has some finite size