# Demonstration of the Iris separation logic in Coq

Robbert Krebbers[1]

Delft University of Technology, The Netherlands

January 21, 2017 @ Coq PL, Paris, France

[1] Iris is joint work with: Ralf Jung, Jacques-Hendri Jourdan, Aleš Bizjak, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Amin Timany, Derek Dreyer, and Lars Birkedal

# Iris Proof Mode (IPM)

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

# Iris Proof Mode (IPM)

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

# Iris Proof Mode (IPM)

**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)

# Iris Proof Mode (IPM)

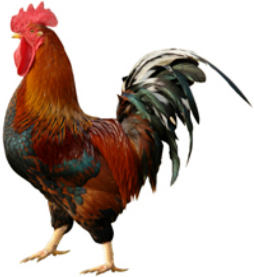
**Goal:** reasoning in Iris in the same style as reasoning in Coq

**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

# Demo



# The setup of IPM in a nutshell

- Deep embedding of contexts as association lists:

```
Record envs :=
  Envs { env_persistent : env iProp; env_spatial : env iProp }.
Coercion of_envs (Δ : envs) : iProp :=
  (⌜ envs_wf Δ ⌝ ∗ □ [∗] env_persistent Δ ∗ [∗] env_spatial Δ)%I.
```

# The setup of IPM in a nutshell

- Deep embedding of contexts as association lists:

```
Record envs :=
  Envs { env_persistent : env iProp; env_spatial : env iProp }.
Coercion of_envs (Δ : envs) : iProp :=
  (⌜ envs_wf Δ ⌝ ∗ □ [∗] env_persistent Δ ∗ [∗] env_spatial Δ)%I.
```

Propositions that enjoy $P \Leftrightarrow P * P$

# The setup of IPM in a nutshell

- Deep embedding of contexts as association lists:

```
Record envs :=
  Envs { env_persistent : env iProp; env_spatial : env iProp }.
Coercion of_envs (Δ : envs) : iProp :=
  (⌜envs_wf Δ⌝ ∗ □ [∗] env_persistent Δ ∗ [∗] env_spatial Δ)%I.
```

Propositions that enjoy $P \Leftrightarrow P * P$

- Tactics implemented by reflection:

```
Lemma tac_sep_split Δ Δ1 Δ2 lr js Q1 Q2 :
  envs_split lr js Δ = Some (Δ1,Δ2) →
  (Δ1 ⊢ Q1) → (Δ2 ⊢ Q2) → Δ ⊢ Q1 ∗ Q2.
```

# The setup of IPM in a nutshell

- Deep embedding of contexts as association lists:

```
Record envs :=
  Envs { env_persistent : env iProp; env_spatial : env iProp }.
Coercion of_envs (Δ : envs) : iProp :=
  (⌜envs_wf Δ⌝ ∗ □ [∗] env_persistent Δ ∗ [∗] env_spatial Δ)%I.
```

Propositions that enjoy $P \Leftrightarrow P * P$

- Tactics implemented by reflection:

```
Lemma tac_sep_split Δ Δ1 Δ2 lr js Q1 Q2 :
  envs_split lr js Δ = Some (Δ1,Δ2) →
  (Δ1 ⊢ Q1) → (Δ2 ⊢ Q2) → Δ ⊢ Q1 ∗ Q2.
```

Context splitting implemented in Gallina

# The setup of IPM in a nutshell

- Deep embedding of contexts as association lists:

```
Record envs :=
  Envs { env_persistent : env iProp; env_spatial : env iProp }.
Coercion of_envs (Δ : envs) : iProp :=
  (⌜envs_wf Δ⌝ ∗ □ [∗] env_persistent Δ ∗ [∗] env_spatial Δ)%I.
```

Propositions that enjoy $P \Leftrightarrow P * P$

- Tactics implemented by reflection:

```
Lemma tac_sep_split Δ Δ1 Δ2 lr js Q1 Q2 :
  envs_split lr js Δ = Some (Δ1,Δ2) →
  (Δ1 ⊢ Q1) → (Δ2 ⊢ Q2) → Δ ⊢ Q1 ∗ Q2.
```

Context splitting implemented in Gallina

- Ltac wrappers around reflective tactics:

```
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 *) ].
```

# This talk

Demonstrate some uses of IPM:

- Symbolic execution
- Lock based concurrency
- Verification of a spin lock

**Part #1:** symbolic execution

# Hoare triples

**Hoare triples** for partial program correctness:

$$\{P\}\, e\, \{w.\, Q\}$$

Precondition

Postcondition

Binder for return value

If the initial state satisfies $P$, then:

- $e$ does not get stuck/crash
- if $e$ terminates with value $v$, the final state satisfies $Q[v/w]$

# Separation logic [O'Hearn, Reynolds, Yang]

**The points-to connective** $x \mapsto v$

- provides the knowledge that location $x$ has value $v$, and
- provides exclusive ownership of $x$

**Separating conjunction** $P * Q$:
the state consists of *disjoint parts* satisfying $P$ and $Q$

# Separation logic [O'Hearn, Reynolds, Yang]

**The points-to connective** $x \mapsto v$

- provides the knowledge that location $x$ has value $v$, and
- provides exclusive ownership of $x$

**Separating conjunction** $P * Q$:
the state consists of *disjoint parts* satisfying $P$ and $Q$

Example:

$$\{x \mapsto v_1 * y \mapsto v_2\}\,\mathit{swap}(x, y)\,\{w.\, w = () \wedge x \mapsto v_2 * y \mapsto v_1\}$$

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

# Proving Hoare triples using IPM

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 Hoare triples using IPM

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:

$$\{P\}\, e\, \{w.\, Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{w.\, Q\})$$

where $\mathsf{wp}\, e\, \{w.\, Q\}$ gives the *weakest precondition* under which:

- all executions of $e$ are safe
- if $e$ terminates with value $v$, the final state satisfies $Q[v/w]$

# Rules for weakest precondition

Rule of ‘consequence’:

$$\mathsf{wp}\ \ e\ \ \{v.\, \mathsf{wp}\, K[\,v\,]\, \{\Phi\}\} \quad -\!\!* \quad \mathsf{wp}\ \ K[\,e\,]\ \ \{\Phi\}$$

Value rule:

$$\Phi\, v \quad -\!\!* \quad \mathsf{wp}\ v\, \{\Phi\}$$

Lamda rule:

$$\mathsf{wp}\ \ e[v/x]\ \ \{\Phi\} \quad -\!\!* \quad \mathsf{wp}\ \ (\lambda x.e)v\ \ \{\Phi\}$$

# Stateful rules for weakest preconditions

Let us just translate the Hoare rules naively:

$$\begin{array}{rcl} \ell \mapsto v & -\!\!* & \mathsf{wp}\ \ !\,\ell\ \{w.\, w = v * \ell \mapsto v\} \\ \ell \mapsto v_1 & -\!\!* & \mathsf{wp}\ \ \ell := v_2\ \{w.\, w = () * \ell \mapsto v_2\} \end{array}$$

Problems:

- Having to frame and weaken to apply these rules
- Equalities in the postconditions

# Stateful rules for weakest preconditions

Let us just translate the Hoare rules naively:

$$\begin{aligned}
\ell \mapsto v \quad &\twoheadrightarrow \quad \mathsf{wp}\ \ !\ell\ \ \{w.\, w = v * \ell \mapsto v\} \\
\ell \mapsto v_1 \quad &\twoheadrightarrow \quad \mathsf{wp}\ \ \ell := v_2\ \ \{w.\, w = () * \ell \mapsto v_2\}
\end{aligned}$$

Problems:

- Having to frame and weaken to apply these rules
- Equalities in the postconditions

'Backwards' or 'predicate transformer' formulation:

$$\begin{aligned}
\ell \mapsto v \quad * \quad (\ell \mapsto v \ \twoheadrightarrow\ \Phi\, v) \quad &\twoheadrightarrow \quad \mathsf{wp}\ \ !\ell\ \ \{\Phi\} \\
\ell \mapsto v_1 \quad * \quad (\ell \mapsto v_2 \ \twoheadrightarrow\ \Phi\, ()) \quad &\twoheadrightarrow \quad \mathsf{wp}\ \ \ell := v_2\ \ \{\Phi\}
\end{aligned}$$

Resources that have to be given up

Resources that are given back

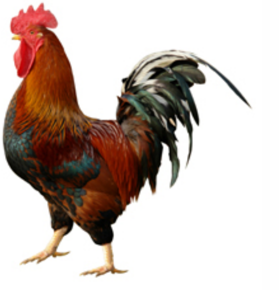
# Nicer definition of the Hoare triple [J-O. Kaiser]

$$\{P\}\, e\, \{\vec{x}\ \mathsf{RET}\, w.\ Q\} \triangleq \square(\forall \Phi.\ P \twoheadrightarrow (\forall \vec{x}.\ Q \twoheadrightarrow \Phi\ w) \twoheadrightarrow \mathsf{wp}\ e\ \{\Phi\})$$

Resources that have to be given up

Resources that are given back

# Demo



**Part #2:** lock based concurrency

# Concurrent separation logic [O'Hearn]

The *par* rule:

$$\frac{\{P_1\}\, e_1 \,\{Q_1\} \qquad \{P_2\}\, e_2 \,\{Q_2\}}{\{P_1 * P_2\}\, e_1 || e_2 \,\{Q_1 * Q_2\}}$$

# Concurrent separation logic [O'Hearn]

The *par* rule:

$$\frac{\{P_1\}e_1\{Q_1\} \qquad \{P_2\}e_2\{Q_2\}}{\{P_1 * P_2\}e_1||e_2\{Q_1 * Q_2\}}$$

For example:

$$\{x \mapsto 4 * y \mapsto 6\}$$

$$x := !\,x + 2 \;\Big\|\; y := !\,y + 2$$

$$\{x \mapsto 6 * y \mapsto 8\}$$

# Concurrent separation logic [O'Hearn]

The *par* rule:

$$\frac{\{P_1\}\, e_1 \,\{Q_1\} \qquad \{P_2\}\, e_2 \,\{Q_2\}}{\{P_1 * P_2\}\, e_1 || e_2 \,\{Q_1 * Q_2\}}$$

For example:

$$\begin{array}{c}
\{x \mapsto 4 * y \mapsto 6\} \\
\begin{array}{l|l}
\{x \mapsto 4\} & \{y \mapsto 6\} \\
x := !\,x + 2 & y := !\,y + 2 \\
\end{array} \\
\{x \mapsto 6 * y \mapsto 8\}
\end{array}$$

# Concurrent separation logic [O'Hearn]

The *par* rule:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 || e_2\, \{Q_1 * Q_2\}}$$

For example:

$$\begin{array}{c}
\{x \mapsto 4 * y \mapsto 6\} \\
\begin{array}{l|l}
\{x \mapsto 4\} & \{y \mapsto 6\} \\
x := !\, x + 2 & y := !\, y + 2 \\
\{x \mapsto 6\} & \{y \mapsto 8\}
\end{array} \\
\{x \mapsto 6 * y \mapsto 8\}
\end{array}$$

# Concurrent separation logic [O'Hearn]

The *par* rule:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 || e_2\, \{Q_1 * Q_2\}}$$

For example:

$$\begin{array}{c}
\{x \mapsto 4 * y \mapsto 6\} \\
\begin{array}{l|l}
\{x \mapsto 4\} & \{y \mapsto 6\} \\
x := !\, x + 2 & y := !\, y + 2 \\
\{x \mapsto 6\} & \{y \mapsto 8\}
\end{array} \\
\{x \mapsto 6 * y \mapsto 8\}
\end{array}$$

Works great for concurrent programs without shared memory: concurrent quick sort, concurrent merge sort, ...

# What about shared state?

A simple problem:

$$\{\mathsf{True}\}$$

$$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$$

$$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$$

$$\mathit{swap}\ x\ y \;\Big\|\; \texttt{assert}\,(!\,x + !\,y = 4)$$

$$\{\mathsf{True}\}$$

# What about shared state?

A simple problem:

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in} \\
\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in} \\
\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in} \\
\begin{array}{l||l}
\mathit{acquire}\ z & \mathit{acquire}\ z \\
\mathit{swap}\ x\ y & \texttt{assert}\,(!\,x + !\,y = 4) \\
\mathit{release}\ z & \mathit{release}\ z
\end{array} \\
\{\mathsf{True}\}
\end{array}
$$

# Specification of a lock

Specifications of the operations:

Lock invariant

$$\{P\}\, new\_lock()\{l.\, \mathsf{IsLock}(l, P)\}$$

$$\{\mathsf{IsLock}(l, P)\}\, acquire\ l\{P \qquad\qquad\}$$

$$\{\mathsf{IsLock}(l, P) * \qquad\qquad P\}\, release\ l\{\mathsf{True}\}$$

The IsLock predicate can be shared among threads:

$$\mathsf{IsLock}(l, P) \Leftrightarrow \mathsf{IsLock}(l, P) * \mathsf{IsLock}(l, P)$$

# Specification of a lock

Specifications of the operations:

Lock invariant

$$\{P\}\, \mathit{new\_lock}()\, \{l.\ \mathsf{IsLock}(l, P)\}$$

$$\{\mathsf{IsLock}(l, P)\}\, \mathit{acquire}\ l\, \{P * \mathsf{Locked}(l)\}$$

$$\{\mathsf{IsLock}(l, P) * \mathsf{Locked}(l) * P\}\, \mathit{release}\ l\, \{\mathsf{True}\}$$

The IsLock predicate can be shared among threads:

$$\mathsf{IsLock}(l, P) \Leftrightarrow \mathsf{IsLock}(l, P) * \mathsf{IsLock}(l, P)$$

# The proof of our program

$\{\mathsf{True}\}$

$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$

$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$

$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$

$$
\begin{array}{l|l}
\mathit{acquire}\ z & \mathit{acquire}\ z \\
\mathit{swap}\ x\ y & \texttt{assert}\ (!\,x + !\,y = 4) \\
\mathit{release}\ z & \mathit{release}\ z
\end{array}
$$

$\{\mathsf{True}\}$

# The proof of our program

$\{\text{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$

$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$

| | |
|---|---|
| $\mathit{acquire}\ z$ | $\mathit{acquire}\ z$ |
| $\mathit{swap}\ x\ y$ | $\texttt{assert}\ (!\,x + !\,y = 4)$ |
| $\mathit{release}\ z$ | $\mathit{release}\ z$ |

$\{\text{True}\}$

# The proof of our program

$\{\mathsf{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$

| | | |
|---|---|---|
| *acquire z* | ‖ | *acquire z* |
| *swap x y* | ‖ | $\texttt{assert}\ (!\,x + !\,y = 4)$ |
| *release z* | ‖ | *release z* |

$\{\mathsf{True}\}$

# The proof of our program

$\{\text{True}\}$

$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$

$\{x \mapsto 0\}$

$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$

$\{x \mapsto 0 * y \mapsto 4\}$

$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$

allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

| | |
|---|---|
| *acquire z* | *acquire z* |
| *swap x y* | $\texttt{assert}\ (!\,x + !\,y = 4)$ |
| *release z* | *release z* |

$\{\text{True}\}$

# The proof of our program

$\{\mathsf{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$
allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

| | |
|---|---|
| $\{\mathsf{True}\}$ | $\{\mathsf{True}\}$ |
| *acquire z* | *acquire z* |
| *swap x y* | $\texttt{assert}\ (!\,x + !\,y = 4)$ |
| *release z* | *release z* |

$\{\mathsf{True}\}$

# The proof of our program

$\{\text{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$
allocate $\text{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

$$
\begin{array}{l||l}
\{\text{True}\} & \{\text{True}\} \\
\mathit{acquire}\ z & \mathit{acquire}\ z \\
\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} & \\
\mathit{swap}\ x\ y & \texttt{assert}\ (!\,x + !\,y = 4) \\
 & \\
\mathit{release}\ z & \mathit{release}\ z \\
 & \\
\{\text{True}\} &
\end{array}
$$

# The proof of our program

$\{\mathsf{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\texttt{let}\ z = \textit{new\_lock}()\ \texttt{in}$
allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

$$\begin{array}{l||l}
\{\mathsf{True}\} & \{\mathsf{True}\} \\
\textit{acquire } z & \textit{acquire } z \\
\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} & \\
\textit{swap } x\ y & \texttt{assert}\ (!\,x + !\,y = 4) \\
\{x \mapsto n_2 * y \mapsto n_1 * n_1 + n2 = 4\} & \\
\textit{release } z & \textit{release } z
\end{array}$$

$\{\mathsf{True}\}$

# The proof of our program

$\{\mathsf{True}\}$

$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$

$\{x \mapsto 0\}$

$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$

$\{x \mapsto 0 * y \mapsto 4\}$

$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$

allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

| | | |
|---|---|---|
| $\{\mathsf{True}\}$ | $\Vert$ | $\{\mathsf{True}\}$ |
| *acquire z* | $\Vert$ | *acquire z* |
| $\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\}$ | $\Vert$ | |
| *swap x y* | $\Vert$ | $\texttt{assert}\ (!\,x + !\,y = 4)$ |
| $\{x \mapsto n_2 * y \mapsto n_1 * n_1 + n2 = 4\}$ | $\Vert$ | |
| *release z* | $\Vert$ | *release z* |
| $\{\mathsf{True}\}$ | $\Vert$ | |

$\{\mathsf{True}\}$

# The proof of our program

$\{\mathsf{True}\}$
$\mathtt{let}\ x = \mathtt{ref}(0)\ \mathtt{in}$
$\{x \mapsto 0\}$
$\mathtt{let}\ y = \mathtt{ref}(4)\ \mathtt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\mathtt{let}\ z = \mathit{new\_lock}()\ \mathtt{in}$
allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

$$
\begin{array}{l||l}
\{\mathsf{True}\} & \{\mathsf{True}\} \\
\mathit{acquire}\ z & \mathit{acquire}\ z \\
\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} & \{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} \\
\mathit{swap}\ x\ y & \mathtt{assert}\ (!\, x + !\, y = 4) \\
\{x \mapsto n_2 * y \mapsto n_1 * n_1 + n2 = 4\} & \\
\mathit{release}\ z & \mathit{release}\ z \\
\{\mathsf{True}\} &
\end{array}
$$

$\{\mathsf{True}\}$

# The proof of our program

$\{\mathsf{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$
allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

| | |
|---|---|
| $\{\mathsf{True}\}$ | $\{\mathsf{True}\}$ |
| *acquire z* | *acquire z* |
| $\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\}$ | $\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\}$ |
| *swap x y* | $\texttt{assert}\ (!\,x + !\,y = 4)$ |
| $\{x \mapsto n_2 * y \mapsto n_1 * n_1 + n2 = 4\}$ | $\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\}$ |
| *release z* | *release z* |
| $\{\mathsf{True}\}$ | |

$\{\mathsf{True}\}$

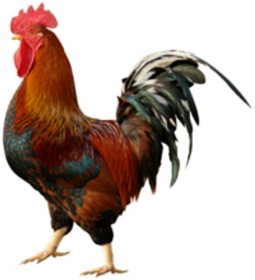
# The proof of our program

$\{\mathsf{True}\}$
$\texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}$
$\{x \mapsto 0\}$
$\texttt{let}\ y = \texttt{ref}(4)\ \texttt{in}$
$\{x \mapsto 0 * y \mapsto 4\}$
$\texttt{let}\ z = \mathit{new\_lock}()\ \texttt{in}$
allocate $\mathsf{IsLock}(z, \exists n_1, n_2.\, x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4)$

$$
\begin{array}{l||l}
\{\mathsf{True}\} & \{\mathsf{True}\} \\
\mathit{acquire}\ z & \mathit{acquire}\ z \\
\{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} & \{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} \\
\mathit{swap}\ x\ y & \texttt{assert}\ (!\,x + !\,y = 4) \\
\{x \mapsto n_2 * y \mapsto n_1 * n_1 + n2 = 4\} & \{x \mapsto n_1 * y \mapsto n_2 * n_1 + n2 = 4\} \\
\mathit{release}\ z & \mathit{release}\ z \\
\{\mathsf{True}\} & \{\mathsf{True}\}
\end{array}
$$

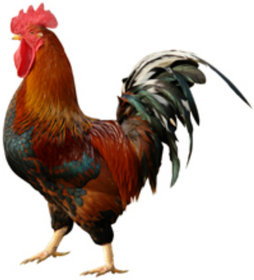
$\{\mathsf{True}\}$

# Demo



**Part #3:** verification of a spin lock

# Demo



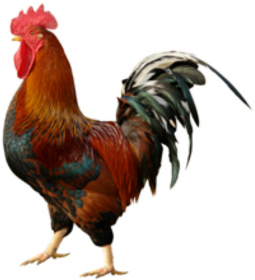
**Part #4:** conclusions

# Current Iris projects

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

# Coq wish list

- Data types in Ltac
- Side-effecting tactics that can return a value
- More expressive parsing mechanism for tactic notations
- Exception handling in Ltac to generate better error messages
- Opt-out from backtracking Ltac semantics
- Better ways to seal-off definitions



# Thank you!

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