

# Program Verification with Types and Logic

## Concurrent separation logic in Iris (optional lecture)

Original version of the slides by Robbert Krebbers (for the Type Theory Based Tools workshop, 2017, Paris, France), with improvements by Derek Dreyer (for MFPS, 2017, Ljubljana, Slovenia), Jacques-Henri Jourdan and Robbert Krebbers (for the POPL'18 tutorial, Los Angeles, USA), Joseph Tassarotti, Ralf Jung, and Tej Chajed (for the POPL'21 tutorial, Online)

# What is Iris?

Language-independent higher-order separation logic with simple foundations for verifying fine-grained concurrent programs in Coq



# What is Iris?

Language-independent **higher-order separation logic** with simple foundations for verifying fine-grained concurrent programs in Coq



- ▶ **Higher-order separation logic:** Supports modular reasoning about higher-order stateful programs

# What is Iris?

Language-independent higher-order separation logic with simple foundations for verifying **fine-grained concurrent programs** in Coq



- ▶ **Higher-order separation logic:** Supports modular reasoning about higher-order stateful programs
- ▶ **Fine-grained concurrent programs:** Programs that use low-level synchronization primitives for more parallelism

# What is Iris?

**Language-independent** higher-order separation logic with simple foundations for verifying fine-grained concurrent programs in Coq



- ▶ **Higher-order separation logic:** Supports modular reasoning about higher-order stateful programs
- ▶ **Fine-grained concurrent programs:** Programs that use low-level synchronization primitives for more parallelism
- ▶ **Language-independent:** Parameterized by the language

# What is Iris?

Language-independent higher-order separation logic with **simple foundations** for verifying fine-grained concurrent programs in Coq



- ▶ **Higher-order separation logic:** Supports modular reasoning about higher-order stateful programs
- ▶ **Fine-grained concurrent programs:** Programs that use low-level synchronization primitives for more parallelism
- ▶ **Language-independent:** Parameterized by the language
- ▶ **Simple foundations:** Small, “canonical” set of primitive rules

# What is Iris?

Language-independent higher-order separation logic with simple foundations for verifying fine-grained concurrent programs **in Coq**



- ▶ **Higher-order separation logic:** Supports modular reasoning about higher-order stateful programs
- ▶ **Fine-grained concurrent programs:** Programs that use low-level synchronization primitives for more parallelism
- ▶ **Language-independent:** Parameterized by the language
- ▶ **Simple foundations:** Small, “canonical” set of primitive rules
- ▶ **Coq:** Support for machine-checked proof (*already used in this course*)

# The versatility of Iris

Iris has been used to formalize many projects, ranging from custom program logics to semantic models of type systems to program proofs, for example:

- ▶ RustBelt
- ▶ Perennial
- ▶ Many other examples



# RustBelt: formalizing the Rust type system

**Rust** is a safe systems programming language with a sophisticated type system based on **ownership**, **borrowing**, and **lifetimes**



- ▶ Safety of high-level Rust code relies on **safe encapsulation** of **unsafe code** in the lower layers
- ▶ **RustBelt** uses Iris to build a **logical relation** for the Rust type system, formalizing the invariants encoded by the types
- ▶ Borrowing and lifetimes are formalized by the **lifetime logic**, which puts Iris's flexibility to the test
- ▶ RustBelt is able to verify the safety of Mutex and other Rust standard library abstractions

## Perennial: logic for crash-safety reasoning

Storage systems need proofs of correctness both under failures (due to kernel panic or disconnecting disk) and normal execution

- ▶ **Perennial** uses Iris to build a variant of Hoare logic with a **crash condition** that holds at all intermediate points, even on failure
- ▶ Iris gives the Perennial logic the flexibility to combine concurrency and failure reasoning
- ▶ Perennial is built on top of a custom language which models the executable code written in **Go**

## Many other diverse projects using Iris

- ▶ **Concurrent Search Templates** uses Iris to prove some data structures correct
- ▶ **Aneris** is a program logic for distributed systems built using Iris
- ▶ **Scala Step-by-Step** formalizes soundness of the Scala type system using Iris to handle step-indexing
- ▶ **RefinedC** is an automated verification tool for C programs based on Iris

# Outline

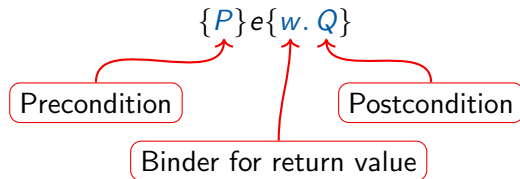
- ▶ **Part 1:** Working with invariants and ghost state
- ▶ **Part 2:** Modeling ghost state via “PCMs”
- ▶ **Hands-on Iris:** Work on the exercises at  
<https://gitlab.mpi-sws.org/iris/tutorial-pop121>

## **Part 1:**

Working with invariants and ghost state

# Hoare triples

**Hoare triples** for partial program correctness:



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 2001]

**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 2001]

**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:

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

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



## Concurrent separation logic [O'Hearn, Brookes 2004]

The *par* rule:

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

## Concurrent separation logic [O'Hearn, Brookes 2004]

The *par* rule:

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

For example:

$$\frac{\{x \mapsto 4 * y \mapsto 6\}}{x := !x + 2 \parallel y := !y + 2} \{x \mapsto 6 * y \mapsto 8\}$$

## Concurrent separation logic [O'Hearn, Brookes 2004]

The *par* rule:

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

For example:

$$\frac{\begin{array}{c} \{x \mapsto 4 * y \mapsto 6\} \\ \{x \mapsto 4\} \quad \{y \mapsto 6\} \\ x := !x + 2 \quad y := !y + 2 \end{array}}{\{x \mapsto 6 * y \mapsto 8\}}$$

## Concurrent separation logic [O'Hearn, Brookes 2004]

The *par* rule:

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

For example:

$$\frac{\begin{array}{c} \{x \mapsto 4 * y \mapsto 6\} \\ \{x \mapsto 4\} \quad \parallel \quad \{y \mapsto 6\} \\ x := !x + 2 \quad y := !y + 2 \\ \{x \mapsto 6\} \quad \parallel \quad \{y \mapsto 8\} \\ \{x \mapsto 6 * y \mapsto 8\} \end{array}}{\{x \mapsto 4 * y \mapsto 6\}}$$

## Concurrent separation logic [O'Hearn, Brookes 2004]

The *par* rule:

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

For example:

$$\frac{\begin{array}{c} \{x \mapsto 4 * y \mapsto 6\} \\ \{x \mapsto 4\} \quad \parallel \quad \{y \mapsto 6\} \\ x := !x + 2 \quad y := !y + 2 \\ \{x \mapsto 6\} \quad \parallel \quad \{y \mapsto 8\} \\ \{x \mapsto 6 * y \mapsto 8\} \end{array}}{\{x \mapsto 4 * y \mapsto 6\}}$$

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

## What about shared state/racy programs?

A classic problem:

```
let x = ref(0) in  
  
fetchandadd(x, 2) || fetchandadd(x, 2)  
  
!x
```

Where `fetchandadd(x, y)` is the atomic version of  $x := !x + y$ .

## What about shared state/racy programs?

A classic problem:

```
{True}
let x = ref(0) in

fetchandadd(x, 2) || fetchandadd(x, 2)

!x
{w. w = 4}
```

Where `fetchandadd(x, y)` is the atomic version of  $x := !x + y$ .

## What about shared state/racy programs?

A classic problem:

```
{True}
let x = ref(0) in
{x ↦ 0}

fetchandadd(x, 2) || fetchandadd(x, 2)

!x
{w. w = 4}
```

Where `fetchandadd(x, y)` is the atomic version of  $x := !x + y$ .



# What about shared state/racy programs?

A classic problem:

```
{True}
let x = ref(0) in
{x ↦ 0}
{??}
fetchandadd(x, 2) || {??}
{??} fetchandadd(x, 2)
!x
{w. w = 4}
```

Where `fetchandadd(x, y)` is the atomic version of `x := !x + y`.

Problem: can only give ownership of `x` to one thread

## Invariants

**The invariant assertion**  $\boxed{R}$  expresses that  $R$  is maintained as an invariant on the state

# Invariants

**The invariant assertion**  $\boxed{R}$  expresses that  $R$  is maintained as an invariant on the state

Invariant opening:

$$\frac{\{R * P\} e \{R * Q\} \quad e \text{ atomic}}{\{\boxed{R} * P\} e \{\boxed{R} * Q\}}$$

# Invariants

**The invariant assertion**  $\boxed{R}$  expresses that  $R$  is maintained as an invariant on the state

Invariant opening:

$$\frac{\{R * P\} e \{R * Q\} \quad e \text{ atomic}}{\{\boxed{R} * P\} e \{\boxed{R} * Q\}}$$

Invariant allocation:

$$\frac{\{\boxed{R} * P\} e \{Q\}}{\{R * P\} e \{Q\}}$$

# Invariants

**The invariant assertion**  $\boxed{R}^{\mathcal{N}}$  expresses that  $R$  is maintained as an invariant on the state

Invariant opening:

$$\frac{\{R * P\} e \{R * Q\} \quad e \text{ atomic}}{\{\boxed{R} * P\} e \{\boxed{R} * Q\}}$$

Invariant allocation:

$$\frac{\{\boxed{R} * P\} e \{Q\}}{\{R * P\} e \{Q\}}$$

Invariant duplication:  $\boxed{R} \vdash \boxed{R} * \boxed{R}$

# Invariants

**The invariant assertion**  $\boxed{R}^{\mathcal{N}}$  expresses that  $R$  is maintained as an invariant on the state

Invariant opening:

$$\frac{\{R * P\} e \{R * Q\}_{\mathcal{E}} \quad e \text{ atomic}}{\{\boxed{R}^{\mathcal{N}} * P\} e \{\boxed{R}^{\mathcal{N}} * Q\}_{\mathcal{E} \uplus \mathcal{N}}}$$

Invariant allocation:

$$\frac{\{\boxed{R}^{\mathcal{N}} * P\} e \{Q\}_{\mathcal{E}}}{\{R * P\} e \{Q\}_{\mathcal{E}}}$$

Invariant duplication:  $\boxed{R}^{\mathcal{N}} \vdash \boxed{R}^{\mathcal{N}} * \boxed{R}^{\mathcal{N}}$

Technicalities: **names** prevent opening the same invariant twice

# Invariants

**The invariant assertion**  $\boxed{R}^{\mathcal{N}}$  expresses that  $R$  is maintained as an invariant on the state

Invariant opening:

$$\frac{\{\triangleright R * P\} e \{\triangleright R * Q\}_{\mathcal{E}} \quad e \text{ atomic}}{\{\boxed{R}^{\mathcal{N}} * P\} e \{\boxed{R}^{\mathcal{N}} * Q\}_{\mathcal{E} \uplus \mathcal{N}}}$$

Invariant allocation:

$$\frac{\{\boxed{R}^{\mathcal{N}} * P\} e \{Q\}_{\mathcal{E}}}{\{\triangleright R * P\} e \{Q\}}$$

Invariant duplication:  $\boxed{R}^{\mathcal{N}} \vdash \boxed{R}^{\mathcal{N}} * \boxed{R}^{\mathcal{N}}$

Technicalities: **names** prevent opening the same invariant twice and the **later**  $\triangleright$  is

needed for impredicativity, i.e.,  $\boxed{\dots \boxed{R}^{\mathcal{N}_2} \dots}^{\mathcal{N}_1}$

## Invariants in action

Let us consider a simpler problem first:

```
{True}  
let x = ref(0) in
```

```
  fetchandadd(x, 2)
```

```
!x
```

```
{n. even(n)}
```

```
  fetchandadd(x, 2)
```



## Invariants in action

Let us consider a simpler problem first:

```
{True}  
let x = ref(0) in  
{x ↦ 0}
```

```
fetchandadd(x, 2)
```

```
!x
```

```
{n. even(n)}
```

```
fetchandadd(x, 2)
```

## Invariants in action

Let us consider a simpler problem first:

```
{True}  
let x = ref(0) in  
{x ↦ 0}  
allocate  $\exists n. x \mapsto n * \text{even}(n)$ 
```

```
fetchandadd(x, 2)
```

```
!x
```

```
{n. even(n)}
```

```
fetchandadd(x, 2)
```

## Invariants in action

Let us consider a simpler problem first:

$\{\text{True}\}$	
<code>let</code> $x = \text{ref}(0)$ <code>in</code>	
$\{x \mapsto 0\}$	
allocate $\boxed{\exists n. x \mapsto n * \text{even}(n)}$	
$\{\boxed{\exists n. x \mapsto n * \text{even}(n)}\}$	$\{\boxed{\exists n. x \mapsto n * \text{even}(n)}\}$
 <code>fetchandadd</code> ( $x, 2$ )	 <code>fetchandadd</code> ( $x, 2$ )
$\{\boxed{\exists n. x \mapsto n * \text{even}(n)}\}$	$\{\boxed{\exists n. x \mapsto n * \text{even}(n)}\}$
 $!x$	
 $\{n. \text{even}(n)\}$	

## Invariants in action

Let us consider a simpler problem first:

```
{True}
let x = ref(0) in
{x ↦ 0}
allocate [∃n. x ↦ n * even(n)]
{ [∃n. x ↦ n * even(n)] }
  { x ↦ n * even(n) }
  fetchandadd(x, 2)
  { x ↦ n + 2 * even(n + 2) }
{ [∃n. x ↦ n * even(n)] }
|
{ [∃n. x ↦ n * even(n)] }
  fetchandadd(x, 2)
{ [∃n. x ↦ n * even(n)] }

!x

{n. even(n)}
```

## Invariants in action

Let us consider a simpler problem first:

```
{True}
let x = ref(0) in
{x ↦ 0}
allocate  $\exists n. x \mapsto n * \text{even}(n)$ 
 $\{ \exists n. x \mapsto n * \text{even}(n) \}$ 
 $\{ x \mapsto n * \text{even}(n) \}$ 
fetchandadd(x, 2)
 $\{ x \mapsto n + 2 * \text{even}(n + 2) \}$ 
 $\{ \exists n. x \mapsto n * \text{even}(n) \}$ 
 $\{ \exists n. x \mapsto n * \text{even}(n) \}$ 
 $\{ x \mapsto n * \text{even}(n) \}$ 
fetchandadd(x, 2)
 $\{ x \mapsto n + 2 * \text{even}(n + 2) \}$ 
 $\{ \exists n. x \mapsto n * \text{even}(n) \}$ 

!x

{n. even(n)}
```

## Invariants in action

Let us consider a simpler problem first:

```
{True}
let x = ref(0) in
{x ↦ 0}
allocate  $\exists n. x \mapsto n * \text{even}(n)$ 

 $\exists n. x \mapsto n * \text{even}(n)$ 
    {x ↦ n * even(n)}
    fetchandadd(x, 2)
    {x ↦ n + 2 * even(n + 2)}
     $\exists n. x \mapsto n * \text{even}(n)$ 
    {x ↦ n * even(n)}
    !x
    {n. x ↦ n * even(n)}
  
{n. even(n)}
```

||

$\exists n. x \mapsto n * \text{even}(n)$   
{x ↦ n \* even(n)}  
fetchandadd(x, 2)  
{x ↦ n + 2 \* even(n + 2)}  
 $\exists n. x \mapsto n * \text{even}(n)$

## Invariants in action

Let us consider a simpler problem first:

$\{\text{True}\}$ $\text{let } x = \text{ref}(0) \text{ in}$ $\{x \mapsto 0\}$ $\text{allocate } \boxed{\exists n. x \mapsto n * \text{even}(n)}$ $\left\{ \boxed{\exists n. x \mapsto n * \text{even}(n)} \right\}$ $\quad \{x \mapsto n * \text{even}(n)\}$ $\quad \text{fetchandadd}(x, 2)$ $\quad \{x \mapsto n + 2 * \text{even}(n + 2)\}$ $\left\{ \boxed{\exists n. x \mapsto n * \text{even}(n)} \right\}$ $\quad \{x \mapsto n * \text{even}(n)\}$ $\quad !x$ $\quad \{n. x \mapsto n * \text{even}(n)\}$ $\{n. \text{even}(n)\}$	$\left\{ \boxed{\exists n. x \mapsto n * \text{even}(n)} \right\}$ $\quad \{x \mapsto n * \text{even}(n)\}$ $\quad \text{fetchandadd}(x, 2)$ $\quad \{x \mapsto n + 2 * \text{even}(n + 2)\}$ $\left\{ \boxed{\exists n. x \mapsto n * \text{even}(n)} \right\}$
--	--

Problem: still cannot prove it returns 4

## Ghost variables

Consider the invariant:

$$\exists n. x \mapsto n * \dots$$

How to avoid **information loss** due to existential quantification?



## Ghost variables

Consider the invariant:

$$\exists n. x \mapsto n * \dots$$

How to avoid **information loss** due to existential quantification?



Solution: ghost variables



## Ghost variables

Consider the invariant:

$$\boxed{\exists n_1, n_2. x \mapsto (n_1 + n_2) * \gamma_1 \hookrightarrow_{\bullet} n_1 * \gamma_2 \hookrightarrow_{\bullet} n_2}$$

How to avoid **information loss** due to existential quantification?



**Solution: ghost variables**



**Ghost variables** come in “entangled” pairs:

$$\underbrace{\gamma \hookrightarrow_{\bullet} n}_{\text{in the invariant ("authoritative")}} * \underbrace{\gamma \hookrightarrow_{\circ} n}_{\text{in the Hoare triple ("fragment")}}$$

## Ghost variables

Consider the invariant:

$$\boxed{\exists n_1, n_2. x \mapsto (n_1 + n_2) * \gamma_1 \hookrightarrow_{\bullet} n_1 * \gamma_2 \hookrightarrow_{\bullet} n_2}$$

How to avoid **information loss** due to existential quantification?



**Solution: ghost variables**



**Ghost variables** come in “entangled” pairs:

$$\text{True} \quad \equiv * \quad \exists \gamma. \quad \underbrace{\gamma \hookrightarrow_{\bullet} n}_{\substack{\text{in the invariant} \\ \text{("authoritative")}}} \quad * \quad \underbrace{\gamma \hookrightarrow_{\circ} n}_{\substack{\text{in the Hoare triple} \\ \text{("fragment")}}}$$

## Ghost variables

Consider the invariant:

$$\boxed{\exists n_1, n_2. x \mapsto (n_1 + n_2) * \gamma_1 \hookrightarrow_\bullet n_1 * \gamma_2 \hookrightarrow_\bullet n_2}$$

How to avoid **information loss** due to existential quantification?



**Solution: ghost variables**



**Ghost variables** come in “entangled” pairs:

$$\text{True} \quad \equiv * \quad \exists \gamma. \quad \underbrace{\gamma \hookrightarrow_\bullet n}_{\substack{\text{in the invariant} \\ \text{“authoritative”}}} \quad * \quad \underbrace{\gamma \hookrightarrow_\circ n}_{\substack{\text{in the Hoare triple} \\ \text{“fragment”}}}$$

When you own both parts you obtain that the values are equal and can update both parts:

$$\begin{aligned} \gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ m &\Rightarrow n = m \\ \gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ m &\equiv * \quad \gamma \hookrightarrow_\bullet n' * \gamma \hookrightarrow_\circ n' \end{aligned}$$

## Aside: Where do these rules come from?

We introduced invariants and now ghost variables, with rules for using them

Where do these rules come from?

## Aside: Where do these rules come from?

We introduced invariants and now ghost variables, with rules for using them

Where do these rules come from?

We will get to that in **Part 2** of this lecture

## Ghost variables in action

$\{\text{True}\}$   
`let  $x = \text{ref}(0)$  in`

`fetchandadd( $x$ , 2)`

`!  $x$`

$\{n. n = 4\}$

`fetchandadd( $x$ , 2)`

## Ghost variables in action

```
{True}  
let x = ref(0) in  
{x ↦ 0}
```

```
fetchandadd(x, 2)
```

```
!x
```

```
{n. n = 4}
```

```
fetchandadd(x, 2)
```



# Ghost variables in action

$\{\text{True}\}$   
`let`  $x = \text{ref}(0)$  `in`  
 $\{x \mapsto 0\}$   
 $\{x \mapsto 0 * \gamma_1 \hookrightarrow_{\bullet} 0 * \gamma_1 \hookrightarrow_o 0 * \gamma_2 \hookrightarrow_{\bullet} 0 * \gamma_2 \hookrightarrow_o 0\}$

`fetchandadd`( $x, 2$ )

$!x$

$\{n. n = 4\}$

`fetchandadd`( $x, 2$ )

## Ghost variables in action

```
{True}  
let x = ref(0) in  
{x ↦ 0}  
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦o 0 * γ2 ↦• 0 * γ2 ↦o 0}  
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
```

fetchandadd(x, 2)

fetchandadd(x, 2)

!x

{n. n = 4}

## Ghost variables in action

```
{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦◦ 0 * γ2 ↦◦ 0}
```

fetchandadd(x, 2)

!x

{n. n = 4}

fetchandadd(x, 2)

## Ghost variables in action

```
{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦o 0 * γ2 ↦• 0 * γ2 ↦o 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦o 0 * γ2 ↦o 0}
{γ1 ↦o 0}
```

fetchandadd(x, 2)

{γ<sub>2</sub> ↦<sub>o</sub> 0}

fetchandadd(x, 2)

!x

{n. n = 4}

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 

```

```

{γ1 ↦◦ 0 * γ2 ↦◦ 0}
{γ1 ↦◦ 0}

```

```

fetchandadd(x, 2)

```

```

{γ1 ↦◦ 2}
{γ1 ↦◦ 2 * γ2 ↦◦ 2}

```

```

!x

```

```

{n. n = 4}

```

```

|| {γ2 ↦◦ 0}

||
||
|| fetchandadd(x, 2)
||
|| {γ2 ↦◦ 2}

```

## Ghost variables in action

<pre> {True} let x = ref(0) in {x ↦ 0} {x ↦ 0 * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} allocate [∃ n<sub>1</sub>, n<sub>2</sub>. x ↦ n<sub>1</sub> + n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>] {γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} {γ<sub>1</sub> ↦<sub>◦</sub> 0}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (n<sub>1</sub> + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}     fetchandadd(x, 2)       {γ<sub>1</sub> ↦<sub>◦</sub> 2}       {γ<sub>1</sub> ↦<sub>◦</sub> 2 * γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>	<pre>    {γ<sub>2</sub> ↦<sub>◦</sub> 0}       fetchandadd(x, 2)       {γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>
--	---

!x

{n. n = 4}

# Ghost variables in action

<pre> {True} let x = ref(0) in {x ↦ 0} {x ↦ 0 * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} allocate [∃ n<sub>1</sub>, n<sub>2</sub>. x ↦ n<sub>1</sub> + n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>] {γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} {γ<sub>1</sub> ↦<sub>◦</sub> 0}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (n<sub>1</sub> + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   fetchandadd(x, 2) {γ<sub>1</sub> ↦<sub>◦</sub> 2} {γ<sub>1</sub> ↦<sub>◦</sub> 2 * γ<sub>2</sub> ↦<sub>◦</sub> 2}  !x  {n. n = 4} </pre>	<pre>    {γ<sub>2</sub> ↦<sub>◦</sub> 0}    fetchandadd(x, 2)     {γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>
--	--

# Ghost variables in action

<pre> {True} let x = ref(0) in {x ↦ 0} {x ↦ 0 * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} allocate <span style="border: 1px solid black; padding: 2px;">∃ n<sub>1</sub>, n<sub>2</sub>. x ↦ n<sub>1</sub> + n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub></span> {γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} {γ<sub>1</sub> ↦<sub>◦</sub> 0}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (n<sub>1</sub> + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   fetchandadd(x, 2) {γ<sub>1</sub> ↦<sub>◦</sub> 2} {γ<sub>1</sub> ↦<sub>◦</sub> 2 * γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>	<pre>    {γ<sub>2</sub> ↦<sub>◦</sub> 0}      fetchandadd(x, 2)     {γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>
--	--

!x

{n. n = 4}



# Ghost variables in action

<pre> {True} let x = ref(0) in {x ↦ 0} {x ↦ 0 * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} allocate <span style="border: 1px solid black; padding: 2px;">∃ n<sub>1</sub>, n<sub>2</sub>. x ↦ n<sub>1</sub> + n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub></span> {γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} {γ<sub>1</sub> ↦<sub>◦</sub> 0}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (n<sub>1</sub> + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   fetchandadd(x, 2)   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (2 + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   {γ<sub>1</sub> ↦<sub>◦</sub> 2}   {γ<sub>1</sub> ↦<sub>◦</sub> 2 * γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>	<pre> {γ<sub>2</sub> ↦<sub>◦</sub> 0}  fetchandadd(x, 2)  {γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>
--	--

!x

{n. n = 4}

# Ghost variables in action

<pre> {True} let x = ref(0) in {x ↦ 0} {x ↦ 0 * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} allocate <span style="border: 1px solid black; padding: 2px;">     ∃ n<sub>1</sub>, n<sub>2</sub>. x ↦ n<sub>1</sub> + n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub> </span> {γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} {γ<sub>1</sub> ↦<sub>◦</sub> 0}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (n<sub>1</sub> + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>} {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>} fetchandadd(x, 2)   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (2 + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   {γ<sub>1</sub> ↦<sub>◦</sub> 2} {γ<sub>1</sub> ↦<sub>◦</sub> 2 * γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>	<pre> {γ<sub>2</sub> ↦<sub>◦</sub> 0}  fetchandadd(x, 2)  {γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>
--	--

!x

{n. n = 4}

# Ghost variables in action

<pre> {True} let x = ref(0) in {x ↦ 0} {x ↦ 0 * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} allocate <span style="border: 1px solid black; padding: 2px;">∃ n<sub>1</sub>, n<sub>2</sub>. x ↦ n<sub>1</sub> + n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub></span> {γ<sub>1</sub> ↦<sub>◦</sub> 0 * γ<sub>2</sub> ↦<sub>◦</sub> 0} {γ<sub>1</sub> ↦<sub>◦</sub> 0} {   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (n<sub>1</sub> + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> n<sub>1</sub> * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ n<sub>2</sub> * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   fetchandadd(x, 2)   {γ<sub>1</sub> ↦<sub>◦</sub> 0 * x ↦ (2 + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> 0 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>}   {γ<sub>1</sub> ↦<sub>◦</sub> 2 * x ↦ (2 + n<sub>2</sub>) * γ<sub>1</sub> ↦<sub>•</sub> 2 * γ<sub>2</sub> ↦<sub>•</sub> n<sub>2</sub>} } {γ<sub>1</sub> ↦<sub>◦</sub> 2} {γ<sub>1</sub> ↦<sub>◦</sub> 2 * γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>	<pre> {γ<sub>2</sub> ↦<sub>◦</sub> 0}  fetchandadd(x, 2)  {γ<sub>2</sub> ↦<sub>◦</sub> 2} </pre>
--	--

!x

{n. n = 4}

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦◦ 0 * γ2 ↦◦ 0}
{γ1 ↦◦ 0}
| {γ1 ↦◦ 0 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 0 * x ↦ n2 * γ1 ↦• 0 * γ2 ↦• n2}
| fetchandadd(x, 2)
| {γ1 ↦◦ 0 * x ↦ (2 + n2) * γ1 ↦• 0 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * x ↦ (2 + n2) * γ1 ↦• 2 * γ2 ↦• n2}
{γ1 ↦◦ 2}
{γ1 ↦◦ 2 * γ2 ↦◦ 2}

```

!x

{n. n = 4}

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
| {γ1 ↦◦ 0 * γ2 ↦◦ 0}
| {γ1 ↦◦ 0}
| {γ1 ↦◦ 0 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 0 * x ↦ n2 * γ1 ↦• 0 * γ2 ↦• n2}
| fetchandadd(x, 2)
| {γ1 ↦◦ 0 * x ↦ (2 + n2) * γ1 ↦• 0 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * x ↦ (2 + n2) * γ1 ↦• 2 * γ2 ↦• n2}
| {γ1 ↦◦ 2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| !x
| {n. n = 4}
| {γ2 ↦◦ 0}
| {...}
| fetchandadd(x, 2)
| {...}
| {γ2 ↦◦ 2}

```

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ₁ ↦• 0 * γ₁ ↦₀ 0 * γ₂ ↦• 0 * γ₂ ↦₀ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
| {γ₁ ↦₀ 0 * γ₂ ↦₀ 0}
| {γ₁ ↦₀ 0}
| {γ₁ ↦₀ 0 * x ↦ (n₁ + n₂) * γ₁ ↦• n₁ * γ₂ ↦• n₂}
| {γ₁ ↦₀ 0 * x ↦ n₂ * γ₁ ↦• 0 * γ₂ ↦• n₂}
| fetchandadd(x, 2)
| {γ₁ ↦₀ 0 * x ↦ (2 + n₂) * γ₁ ↦• 0 * γ₂ ↦• n₂}
| {γ₁ ↦₀ 2 * x ↦ (2 + n₂) * γ₁ ↦• 2 * γ₂ ↦• n₂}
| {γ₁ ↦₀ 2}
| {γ₁ ↦₀ 2 * γ₂ ↦₀ 2}
| {γ₁ ↦₀ 2 * γ₂ ↦₀ 2 * x ↦ (n₁ + n₂) * γ₁ ↦• n₁ * γ₂ ↦• n₂}
| !x
| {n. n = 4}

```

```

|| {γ₂ ↦₀ 0}
| {...}
| fetchandadd(x, 2)
| {...}
|| {γ₂ ↦₀ 2}

```

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦◦ 0 * γ2 ↦◦ 0}
{γ1 ↦◦ 0}
| {γ1 ↦◦ 0 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 0 * x ↦ n2 * γ1 ↦• 0 * γ2 ↦• n2}
| fetchandadd(x, 2)
| {γ1 ↦◦ 0 * x ↦ (2 + n2) * γ1 ↦• 0 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * x ↦ (2 + n2) * γ1 ↦• 2 * γ2 ↦• n2}
{γ1 ↦◦ 2}
{γ1 ↦◦ 2 * γ2 ↦◦ 2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ 4 * γ1 ↦• 2 * γ2 ↦• 2}
|x
|
{n. n = 4}

```

```

{γ2 ↦◦ 0}
| {...}
| fetchandadd(x, 2)
| {...}
{γ2 ↦◦ 2}

```

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦◦ 0 * γ2 ↦◦ 0}
{γ1 ↦◦ 0}
| {γ1 ↦◦ 0 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 0 * x ↦ n2 * γ1 ↦• 0 * γ2 ↦• n2}
| fetchandadd(x, 2)
| {γ1 ↦◦ 0 * x ↦ (2 + n2) * γ1 ↦• 0 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * x ↦ (2 + n2) * γ1 ↦• 2 * γ2 ↦• n2}
{γ1 ↦◦ 2}
{γ1 ↦◦ 2 * γ2 ↦◦ 2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ 4 * γ1 ↦• 2 * γ2 ↦• 2}
|x
|
{n. n = 4}

```

```

{γ2 ↦◦ 0}
| {...}
| fetchandadd(x, 2)
| {...}
{γ2 ↦◦ 2}

```



# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦◦ 0 * γ2 ↦◦ 0}
{γ1 ↦◦ 0}
| {γ1 ↦◦ 0 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 0 * x ↦ n2 * γ1 ↦• 0 * γ2 ↦• n2}
| fetchandadd(x, 2)
| {γ1 ↦◦ 0 * x ↦ (2 + n2) * γ1 ↦• 0 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * x ↦ (2 + n2) * γ1 ↦• 2 * γ2 ↦• n2}
{γ1 ↦◦ 2}
{γ1 ↦◦ 2 * γ2 ↦◦ 2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ 4 * γ1 ↦• 2 * γ2 ↦• 2}
!x
| {n. n = 4 * γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ 4 * γ1 ↦• 2 * γ2 ↦• 2}
{n. n = 4}

```

{γ<sub>2</sub> ↦<sub>◦</sub> 0}  
{...}  
fetchandadd(x, 2)  
{...}  
{γ<sub>2</sub> ↦<sub>◦</sub> 2}

# Ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ1 ↦• 0 * γ1 ↦◦ 0 * γ2 ↦• 0 * γ2 ↦◦ 0}
allocate  $\exists n_1, n_2. x \mapsto n_1 + n_2 * \gamma_1 \mapsto_{\bullet} n_1 * \gamma_2 \mapsto_{\bullet} n_2$ 
{γ1 ↦◦ 0 * γ2 ↦◦ 0}
{γ1 ↦◦ 0}
| {γ1 ↦◦ 0 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 0 * x ↦ n2 * γ1 ↦• 0 * γ2 ↦• n2}
| fetchandadd(x, 2)
| {γ1 ↦◦ 0 * x ↦ (2 + n2) * γ1 ↦• 0 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * x ↦ (2 + n2) * γ1 ↦• 2 * γ2 ↦• n2}
{γ1 ↦◦ 2}
{γ1 ↦◦ 2 * γ2 ↦◦ 2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ (n1 + n2) * γ1 ↦• n1 * γ2 ↦• n2}
| {γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ 4 * γ1 ↦• 2 * γ2 ↦• 2}
| !x
| {n. n = 4 * γ1 ↦◦ 2 * γ2 ↦◦ 2 * x ↦ 4 * γ1 ↦• 2 * γ2 ↦• 2}
{n. n = 4}

```

{γ<sub>2</sub> ↦<sub>◦</sub> 0}  
| {...}  
| fetchandadd(x, 2)  
| {...}  
{γ<sub>2</sub> ↦<sub>◦</sub> 2}

## Ghost variables with fractional permissions [Boyland 2003]

What if we have  $n$  threads? Using  $n$  different ghost variables, results in different proofs for each thread. *That is not modular.*

Better way: ghost variables with a *fractional permission*  $(0, 1]_{\mathbb{Q}}$ :

$$\gamma \xrightarrow[\circ]{\pi_1 + \pi_2} (n_1 + n_2) \quad \Leftrightarrow \quad \gamma \xrightarrow[\circ]{\pi_1} n_1 * \gamma \xrightarrow[\circ]{\pi_2} n_2$$

When allocating you get full ownership ( $\pi = 1$ ):

$$\text{True} \equiv * \quad \exists \gamma. \gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow[\circ]{1} n$$

## Ghost variables with fractional permissions [Boyland 2003]

What if we have  $n$  threads? Using  $n$  different ghost variables, results in different proofs for each thread. *That is not modular.*

Better way: ghost variables with a *fractional permission*  $(0, 1]_{\mathbb{Q}}$ :

$$\gamma \xrightarrow[\circ]{\pi_1 + \pi_2} (n_1 + n_2) \quad \Leftrightarrow \quad \gamma \xrightarrow[\circ]{\pi_1} n_1 * \gamma \xrightarrow[\circ]{\pi_2} n_2$$

When allocating you get full ownership ( $\pi = 1$ ):

$$\text{True} \Rightarrow * \quad \exists \gamma. \gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow[\circ]{1} n$$

You only get the equality when you have *full ownership* ( $\pi = 1$ ):

$$\gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow[\circ]{1} m \quad \Rightarrow \quad n = m$$

## Ghost variables with fractional permissions [Boyland 2003]

What if we have  $n$  threads? Using  $n$  different ghost variables, results in different proofs for each thread. *That is not modular.*

Better way: ghost variables with a *fractional permission*  $(0, 1]_{\mathbb{Q}}$ :

$$\gamma \xrightarrow{\pi_1 + \pi_2}_{\circ} (n_1 + n_2) \quad \Leftrightarrow \quad \gamma \xrightarrow{\pi_1}_{\circ} n_1 * \gamma \xrightarrow{\pi_2}_{\circ} n_2$$

When allocating you get full ownership ( $\pi = 1$ ):

$$\text{True} \equiv * \quad \exists \gamma. \gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow{1}_{\circ} n$$

You only get the equality when you have *full ownership* ( $\pi = 1$ ):

$$\gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow{1}_{\circ} m \quad \Rightarrow \quad n = m$$

Updating is possible with *partial ownership* ( $0 < \pi \leq 1$ ):

$$\gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow{\pi}_{\circ} m \quad \equiv * \quad \gamma \hookrightarrow_{\bullet} (n + i) * \gamma \xrightarrow{\pi}_{\circ} (m + i)$$

## Ghost variables with fractional permissions [Boyland 2003]

What if we have  $n$  threads? Using  $n$  different ghost variables, results in different proofs for each thread. *That is not modular.*

Better way: ghost variables with a *fractional permission*  $(0, 1]_{\mathbb{Q}}$ :

$$\gamma \xrightarrow{\pi_1 + \pi_2}_{\circ} (n_1 + n_2) \quad \Leftrightarrow \quad \gamma \xrightarrow{\pi_1}_{\circ} n_1 * \gamma \xrightarrow{\pi_2}_{\circ} n_2$$

When allocating you get full ownership ( $\pi = 1$ ):

$$\text{True} \equiv * \quad \exists \gamma. \gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow{1}_{\circ} n$$

You only get the equality when you have *full ownership* ( $\pi = 1$ ):

$$\gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow{1}_{\circ} m \quad \Rightarrow \quad n = m$$

Updating is possible with *partial ownership* ( $0 < \pi \leq 1$ ):

$$\gamma \hookrightarrow_{\bullet} n * \gamma \xrightarrow{\pi}_{\circ} m \quad \equiv * \quad \gamma \hookrightarrow_{\bullet} (n + i) * \gamma \xrightarrow{\pi}_{\circ} (m + i)$$

Keeps the invariant that all  $\gamma \xrightarrow{\pi_i}_{\circ} n_i$  sum up to  $\gamma \hookrightarrow_{\bullet} \sum n_i$

# Fractional ghost variables in action

```
{True}  
let x = ref(0) in
```

```
fetchandadd(x, 2)
```

```
!x
```

```
{n. n = 2k}
```

```
fetchandadd(x, 2) ...
```

# Fractional ghost variables in action

```
{True}  
let x = ref(0) in  
{x ↦ 0}
```

fetchandadd(x, 2)

!x

{n. n = 2k}

||  
fetchandadd(x, 2)  
|| ...



# Fractional ghost variables in action

```
{True}  
let x = ref(0) in  
{x ↦ 0}  
{x ↦ 0 * γ ↦• 0 * γ ↦o1 0}
```

fetchandadd(x, 2)

!x

{n. n = 2k}

|| fetchandadd(x, 2) || ...

# Fractional ghost variables in action

```
{True}  
let x = ref(0) in  
{x ↦ 0}  
{x ↦ 0 * γ ↦• 0 * γ ↦01 0}  
allocate  $\exists n. x \mapsto n * \gamma \mapsto_{\bullet} n$ 
```

fetchandadd(x, 2)

!x

{n. n = 2k}

||  
fetchandadd(x, 2)  
|| ...

# Fractional ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ ↦• 0 * γ ↦o1 0}
allocate  $\exists n. x \mapsto n * \gamma \mapsto_{\bullet} n$ 
{γ ↦o1/k 0}

```

fetchandadd(x, 2)

{γ ↦<sub>o</sub><sup>1/k</sup> 2}

!x

{n. n = 2k}

$$\left\| \begin{array}{c} \left\{ \gamma \mapsto_o^{1/k} 0 \right\} \\ \\ \text{fetchandadd}(x, 2) \\ \\ \left\{ \gamma \mapsto_o^{1/k} 2 \right\} \end{array} \right\| \dots$$

# Fractional ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ ↦• 0 * γ ↦o1 0}
allocate  $\exists n. x \mapsto n * \gamma \mapsto_{\bullet} n$ 
{
  γ ↦o1/k 0
  {
    γ ↦o1/k 0 * x ↦ n * γ ↦• n
    fetchandadd(x, 2)
  }
  γ ↦o1/k 2
}

```

!x

{n. n = 2k}

$$\left\| \begin{array}{c} \left\{ \gamma \xrightarrow{o}^{1/k} 0 \right\} \\ \text{fetchandadd}(x, 2) \\ \left\{ \gamma \xrightarrow{o}^{1/k} 2 \right\} \end{array} \right\| \dots$$

# Fractional ghost variables in action

$$\begin{array}{l}
 \{\text{True}\} \\
 \text{let } x = \text{ref}(0) \text{ in} \\
 \{x \mapsto 0\} \\
 \{x \mapsto 0 * \gamma \hookrightarrow_{\bullet} 0 * \gamma \xrightarrow{1}_{\circ} 0\} \\
 \text{allocate } \boxed{\exists n. x \mapsto n * \gamma \hookrightarrow_{\bullet} n} \\
 \left\{ \gamma \xrightarrow{1/k}_{\circ} 0 \right\} \\
 \left\{ \gamma \xrightarrow{1/k}_{\circ} 0 * x \mapsto n * \gamma \hookrightarrow_{\bullet} n \right\} \\
 \text{fetchandadd}(x, 2) \\
 \left\{ \gamma \xrightarrow{1/k}_{\circ} 2 * x \mapsto (2 + n) * \gamma \hookrightarrow_{\bullet} (2 + n) \right\} \\
 \left\{ \gamma \xrightarrow{1/k}_{\circ} 2 \right\}
 \end{array}
 \parallel \parallel
 \begin{array}{l}
 \left\{ \gamma \xrightarrow{1/k}_{\circ} 0 \right\} \\
 \text{fetchandadd}(x, 2) \\
 \left\{ \gamma \xrightarrow{1/k}_{\circ} 2 \right\}
 \end{array}
 \parallel \parallel \dots$$

!x

$\{n. n = 2k\}$

# Fractional ghost variables in action

$\{\text{True}\}$

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

$\{x \mapsto 0\}$

$\{x \mapsto 0 * \gamma \hookrightarrow_{\bullet} 0 * \gamma \xrightarrow{1}_{\circ} 0\}$

**allocate**  $\boxed{\exists n. x \mapsto n * \gamma \hookrightarrow_{\bullet} n}$

$\{\gamma \xrightarrow{1/k}_{\circ} 0\}$

$\left\{ \gamma \xrightarrow{1/k}_{\circ} 0 * x \mapsto n * \gamma \hookrightarrow_{\bullet} n \right\}$

**fetchandadd**( $x, 2$ )

$\left\{ \gamma \xrightarrow{1/k}_{\circ} 2 * x \mapsto (2 + n) * \gamma \hookrightarrow_{\bullet} (2 + n) \right\}$

$\{\gamma \xrightarrow{1/k}_{\circ} 2\}$

$\left\{ \gamma \xrightarrow{1/k}_{\circ} 0 \right\}$

$\{\dots\}$

**fetchandadd**( $x, 2$ )

$\{\dots\}$

$\left\{ \gamma \xrightarrow{1/k}_{\circ} 2 \right\}$

$\dots$

$!x$

$\{n. n = 2k\}$

# Fractional ghost variables in action

$\{\text{True}\}$ <code>let</code> $x = \text{ref}(0)$ <code>in</code> $\{x \mapsto 0\}$ $\{x \mapsto 0 * \gamma \hookrightarrow_{\bullet} 0 * \gamma \xrightarrow{1}_{\circ} 0\}$ <code>allocate</code> $\boxed{\exists n. x \mapsto n * \gamma \hookrightarrow_{\bullet} n}$ $\{\gamma \xrightarrow{1/k}_{\circ} 0\}$ $\left\{ \begin{array}{l} \gamma \xrightarrow{1/k}_{\circ} 0 * x \mapsto n * \gamma \hookrightarrow_{\bullet} n \\ \text{fetchandadd}(x, 2) \\ \gamma \xrightarrow{1/k}_{\circ} 2 * x \mapsto (2 + n) * \gamma \hookrightarrow_{\bullet} (2 + n) \end{array} \right\}$ $\{\gamma \xrightarrow{1/k}_{\circ} 2\}$ $\left\{ \begin{array}{l} \gamma \xrightarrow{1}_{\circ} 2k * x \mapsto n * \gamma \hookrightarrow_{\bullet} n \\ !x \end{array} \right\}$ $\{n. n = 2k\}$	$\left\  \begin{array}{l} \{\gamma \xrightarrow{1/k}_{\circ} 0\} \\ \{\dots\} \\ \text{fetchandadd}(x, 2) \\ \{\dots\} \\ \{\gamma \xrightarrow{1/k}_{\circ} 2\} \end{array} \right\  \dots$
---	--

# Fractional ghost variables in action

```

{True}
let x = ref(0) in
{x ↦ 0}
{x ↦ 0 * γ ↦o 0 * γ ↦o 0}
allocate  $\exists n. x \mapsto n * \gamma \mapsto_{\bullet} n$ 
{
  γ ↦o 0
  {
    γ ↦o 0 * x ↦ n * γ ↦o n
    fetchandadd(x, 2)
    {
      γ ↦o 2 * x ↦ (2 + n) * γ ↦o (2 + n)
    }
    γ ↦o 2
    {
      γ ↦o 2k * x ↦ n * γ ↦o n
    }
    !x
    {
      n.n = 2k ∧ γ ↦o 2k * x ↦ 2k * γ ↦o 2k
    }
  }
  {n.n = 2k}
}

```

$\left\{ \gamma \mapsto_{\circ} 0 \right\}$

$\left\{ \gamma \mapsto_{\circ} 0 * x \mapsto n * \gamma \mapsto_{\circ} n \right\}$

$\left\{ \gamma \mapsto_{\circ} 2 * x \mapsto (2 + n) * \gamma \mapsto_{\circ} (2 + n) \right\}$

$\left\{ \gamma \mapsto_{\circ} 2 \right\}$

$\left\{ \gamma \mapsto_{\circ} 2k * x \mapsto n * \gamma \mapsto_{\circ} n \right\}$

$\left\{ n.n = 2k \wedge \gamma \mapsto_{\circ} 2k * x \mapsto 2k * \gamma \mapsto_{\circ} 2k \right\}$

$\{n.n = 2k\}$

$\parallel$

$\left\{ \gamma \mapsto_{\circ} 0 \right\}$

$\left\{ \dots \right\}$

$\text{fetchandadd}(x, 2)$

$\left\{ \dots \right\}$

$\left\{ \gamma \mapsto_{\circ} 2 \right\}$

$\parallel$

$\dots$

19



**Part 2:**  
Modeling ghost state via “PCMs”

# Mechanisms for concurrent reasoning

We have seen so far:

- ▶ Invariants  $\boxed{R}^{\mathcal{N}}$
- ▶ Ghost variables  $\gamma \hookrightarrow_{\bullet} n$  and  $\gamma \hookrightarrow_{\circ} n$
- ▶ Fractional ghost variables  $\gamma \hookrightarrow_{\bullet} n$  and  $\gamma \xrightarrow{\pi}_{\circ} n$

How can we make sure we have all the mechanisms we will need?

# Mechanisms for concurrent reasoning

We have seen so far:

- ▶ Invariants  $\boxed{R}^{\mathcal{N}}$
- ▶ Ghost variables  $\gamma \hookrightarrow_{\bullet} n$  and  $\gamma \hookrightarrow_{\circ} n$
- ▶ Fractional ghost variables  $\gamma \hookrightarrow_{\bullet} n$  and  $\gamma \xrightarrow{\pi}_{\circ} n$

How can we make sure we have all the mechanisms we will need?

**The Iris approach:** these mechanisms can be **encoded** using a simple mechanism of *ghost resource ownership*

## Resource algebras (RAs): A generalization of PCMs

*Resource algebra* (RA) with carrier  $M$ :

- ▶ Composition  $(\cdot) : M \rightarrow M \rightarrow M$
- ▶ Validity predicate  $\mathcal{V} \subseteq M$

Satisfying:

$$a \cdot b = b \cdot a \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c \quad (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V}$$

## Resource algebras (RAs): A generalization of PCMs

*Resource algebra* (RA) with carrier  $M$ :

- ▶ Composition  $(\cdot) : M \rightarrow M \rightarrow M$
- ▶ Validity predicate  $\mathcal{V} \subseteq M$

Satisfying:

$$a \cdot b = b \cdot a \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c \quad (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V}$$

Iris provides  $\boxed{a : M}^\gamma$  expressing ownership of an element  $a$  of resource algebra  $M$  (with name  $\gamma$ )

## Ghost variables revisited

Resource algebra for ghost variables:

$$M \triangleq \bullet n \mid \circ n \mid \bullet \circ n \mid \perp$$

$$\mathcal{V} \triangleq \{a \neq \perp \mid a \in M\}$$

$$\bullet n \cdot \circ n' = \circ n' \cdot \bullet n \triangleq \begin{cases} \bullet \circ n & \text{if } n = n' \\ \perp & \text{otherwise} \end{cases}$$

$$\text{other combinations} \triangleq \perp$$

## Ghost variables revisited

Resource algebra for ghost variables:

$$\begin{aligned}M &\triangleq \bullet n \mid \circ n \mid \bullet \circ n \mid \perp \\ \mathcal{V} &\triangleq \{a \neq \perp \mid a \in M\} \\ \bullet n \cdot \circ n' &= \circ n' \cdot \bullet n \triangleq \begin{cases} \bullet \circ n & \text{if } n = n' \\ \perp & \text{otherwise} \end{cases} \\ \text{other combinations} &\triangleq \perp\end{aligned}$$

And define:

$$\gamma \hookrightarrow_{\bullet} n \triangleq [\bullet n]^{\gamma}$$

$$\gamma \hookrightarrow_{\circ} n \triangleq [\circ n]^{\gamma}$$

## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv * \exists \gamma. [a]^\gamma \quad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \quad [a]^\gamma \Rightarrow a \in \mathcal{V}$$



## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv \bigstar \exists \gamma. [a]^\gamma \qquad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \qquad [a]^\gamma \Rightarrow a \in \mathcal{V}$$

The ghost variable laws follow from these:

$$\text{True} \equiv \bigstar \exists \gamma. [\bullet n]^\gamma * [\circ n]^\gamma$$

## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv \star \exists \gamma. [a]^\gamma \quad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \quad [a]^\gamma \Rightarrow a \in \mathcal{V}$$

The ghost variable laws follow from these:

$$\text{True} \equiv \star \exists \gamma. [\bullet \circ n]^\gamma \equiv \star \exists \gamma. [\bullet n]^\gamma * [\circ n]^\gamma$$

## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv \bigstar \exists \gamma. [a]^\gamma \qquad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \qquad [a]^\gamma \Rightarrow a \in \mathcal{V}$$

The ghost variable laws follow from these:

$$\text{True} \equiv \bigstar \exists \gamma. [\bullet \circ n]^\gamma \equiv \bigstar \exists \gamma. [\bullet n]^\gamma * [\circ n]^\gamma$$

Remember:

$$\bullet n \cdot \circ n' = \circ n' \cdot \bullet n \triangleq \begin{cases} \bullet \circ n & \text{if } n = n' \\ \perp & \text{otherwise} \end{cases}$$

## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv \bigstar \exists \gamma. [a]^\gamma \qquad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \qquad [a]^\gamma \Rightarrow a \in \mathcal{V}$$

The ghost variable laws follow from these:

$$\begin{aligned} \text{True} &\equiv \bigstar \exists \gamma. [\bullet \circ n]^\gamma \equiv \bigstar \exists \gamma. [\bullet n]^\gamma * [\circ n]^\gamma \\ [\bullet n]^\gamma * [\circ m]^\gamma &\Rightarrow n = m \end{aligned}$$

Remember:

$$\bullet n \cdot \circ n' = \circ n' \cdot \bullet n \triangleq \begin{cases} \bullet \circ n & \text{if } n = n' \\ \perp & \text{otherwise} \end{cases}$$

## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv \bigstar \exists \gamma. [a]^\gamma \quad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \quad [a]^\gamma \Rightarrow a \in \mathcal{V}$$

The ghost variable laws follow from these:

$$\begin{aligned} \text{True} &\equiv \bigstar \exists \gamma. [\bullet \circ n]^\gamma \equiv \bigstar \exists \gamma. [\bullet n]^\gamma * [\circ n]^\gamma \\ &\quad [\bullet n]^\gamma * [\circ m]^\gamma \Rightarrow (\bullet n \cdot \circ m) \in \mathcal{V} \Rightarrow n = m \end{aligned}$$

Remember:

$$\begin{aligned} \bullet n \cdot \circ n' &= \circ n' \cdot \bullet n \triangleq \begin{cases} \bullet \circ n & \text{if } n = n' \\ \perp & \text{otherwise} \end{cases} \\ \mathcal{V} &\triangleq \{a \neq \perp \mid a \in M\} \end{aligned}$$

## Ghost resource laws

Iris provides general laws for ghost resources:

$$a \in \mathcal{V} \equiv \bigstar \exists \gamma. [a]^\gamma \quad [a \cdot b]^\gamma \Leftrightarrow [a]^\gamma * [b]^\gamma \quad [a]^\gamma \Rightarrow a \in \mathcal{V}$$

The ghost variable laws follow from these:

$$\begin{aligned} \text{True} &\equiv \bigstar \exists \gamma. [\bullet \circ n]^\gamma \equiv \bigstar \exists \gamma. [\bullet n]^\gamma * [\circ n]^\gamma \\ [\bullet n]^\gamma * [\circ m]^\gamma &\Rightarrow (\bullet n \cdot \circ m) \in \mathcal{V} \Rightarrow n = m \end{aligned}$$

Remember:

$$\bullet n \cdot \circ n' = \circ n' \cdot \bullet n \triangleq \begin{cases} \bullet \circ n & \text{if } n = n' \\ \perp & \text{otherwise} \end{cases}$$
$$\mathcal{V} \triangleq \{a \neq \perp \mid a \in M\}$$

## Updating resources

Resources can be *updated* using **frame-preserving updates**:

$$\frac{a \rightsquigarrow b}{\boxed{a}^\gamma \equiv * \boxed{b}^\gamma} \quad a \rightsquigarrow b \triangleq \forall a_f. a \cdot a_f \in \mathcal{V} \Rightarrow b \cdot a_f \in \mathcal{V}$$

# Updating resources

Resources can be *updated* using **frame-preserving updates**:

$$\frac{a \rightsquigarrow b}{\boxed{a}^\gamma \equiv \star \boxed{b}^\gamma} \quad a \rightsquigarrow b \triangleq \forall a_f. a \cdot a_f \in \mathcal{V} \Rightarrow b \cdot a_f \in \mathcal{V}$$

Key idea: a resource can be updated if the update does not invalidate the resources of concurrently-running threads

Thread 1		Thread 2		...		Thread n	
$a$	$\cdot$	$a_2$	$\cdot$	...	$\cdot$	$a_n$	$\in \mathcal{V}$
$\Downarrow$							
$b$	$\cdot$	$a_2$	$\cdot$	...	$\cdot$	$a_n$	$\in \mathcal{V}$



# Updating resources

Resources can be *updated* using **frame-preserving updates**:

$$\frac{a \rightsquigarrow b}{\boxed{a}^\gamma \equiv_* \boxed{b}^\gamma} \quad a \rightsquigarrow b \triangleq \forall a_f. a \cdot a_f \in \mathcal{V} \Rightarrow b \cdot a_f \in \mathcal{V}$$

For ghost variables:

$$\frac{\bullet \circ n \rightsquigarrow \bullet \circ n'}{\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ n \equiv_* \gamma \hookrightarrow_\bullet n' * \gamma \hookrightarrow_\circ n'}$$

## Updating resources

Resources can be *updated* using **frame-preserving updates**:

$$\frac{a \rightsquigarrow b}{\boxed{a}^\gamma \equiv * \boxed{b}^\gamma} \quad a \rightsquigarrow b \triangleq \forall a_f. a \cdot a_f \in \mathcal{V} \Rightarrow b \cdot a_f \in \mathcal{V}$$

For ghost variables:

$$\frac{\bullet n \rightsquigarrow \bullet n'}{\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ n \equiv * \gamma \hookrightarrow_\bullet n' * \gamma \hookrightarrow_\circ n'}$$

$$\bullet n \rightsquigarrow \bullet n' = \forall a_f. \bullet n \cdot a_f \in \mathcal{V} \Rightarrow \bullet n' \cdot a_f \in \mathcal{V}$$

## Updating resources

Resources can be *updated* using **frame-preserving updates**:

$$\frac{a \rightsquigarrow b}{\boxed{a}^\gamma \equiv * \boxed{b}^\gamma} \quad a \rightsquigarrow b \triangleq \forall a_f. a \cdot a_f \in \mathcal{V} \Rightarrow b \cdot a_f \in \mathcal{V}$$

For ghost variables:

$$\frac{\bullet n \rightsquigarrow \bullet n'}{\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ n \equiv * \gamma \hookrightarrow_\bullet n' * \gamma \hookrightarrow_\circ n'}$$

$$\bullet n \rightsquigarrow \bullet n' = \forall a_f. \bullet n \cdot a_f \in \mathcal{V} \Rightarrow \bullet n' \cdot a_f \in \mathcal{V}$$

$\bullet n \cdot a_f = \perp$ , so the premise holds vacuously.

## Generalizing to a library of RA combinators

Iris comes with a library of useful RA combinators

- ▶  $\text{AUTH}(M)$ : Generalizes the  $\bullet$ ,  $\circ$ ,  $\bullet\circ$  construction over an arbitrary RA  $M$  – we call it the “authoritative” RA.
- ▶  $\text{EXCL}(X)$ : The “exclusive” RA, whose valid elements are the elements of  $X$ , and where composition is always undefined.
- ▶  $\text{FRAC}$ : The RA for fractions in  $(0, 1]$  with addition.
- ▶ The expected RA liftings of products, sums, etc.

Using these combinators, we can easily construct the necessary models of many desired forms of ghost state:

- ▶ Ghost variables:  $\text{AUTH}(\text{EXCL NAT})$
- ▶ Fractional ghost variables:  $\text{AUTH}(\text{FRAC} \times \text{NAT}_+)$

# Many things I have not covered

Modal basis of Iris:  $\Box, \triangleright, \Vdash$

- ▶ **Persistent** modality  $\Box P$ : Says  $P$  holds forever, i.e., only relying on duplicable resources, such as invariants
- ▶ **Later** modality  $\triangleright P$ : Says  $P$  holds one step-index later (lower); needed to model impredicative invariants
- ▶ **Update** modality  $\Vdash P$ : Says  $P$  holds after some frame-preserving update to ghost state

Higher-order ghost state, e.g., named propositions  $\gamma \mapsto P$

- ▶  $\gamma \mapsto P$  means  $\gamma$  is a name for proposition  $P$
- ▶  $\gamma \mapsto P * \gamma \mapsto Q \Rightarrow \triangleright(P = Q)$
- ▶ Sounds arcane, but turns out to be surprisingly useful!
- ▶ Achieved by equipping RAs with a step-indexing structure

Encoding of Iris program logic (including invariants)  
within the modal base logic (with higher-order ghost state)

# Homework

- ▶ Clone the tutorial lecture material  
`https://gitlab.mpi-sws.org/iris/tutorial-pop121`
- ▶ Read the README.md and install `coq-iris-heap-lang`
- ▶ Exercise 1 and 2 are about sequential programs (so you can get used to the `wp_` tactics and style of Hoare triples in Iris)
- ▶ Exercise 3, 4 and 5 are about concurrency