

# An introduction to Iris

Jean-Marie Madiot & **François Pottier**

JFLA 2026

- 1 What is Iris (About)?
- 2 Basic Connectives
- 3 Mutable State
- 4 Locks (Primitive)
- 5 Invariants
- 6 Locks (User-Defined)

To prove the *safety* and *correctness* of programs,

To prove the *safety* and *correctness* of programs,

- in the beginning there was Floyd-Hoare logic (1967–1969)
  - *propositions* about the machine's state

To prove the *safety* and *correctness* of programs,

- in the beginning there was Floyd-Hoare logic (1967–1969)
  - *propositions* about the machine's state
- then there was Separation Logic (1999–2002)
  - *assertions* about *fragments* of the machine's state
  - *separation* and *ownership*
  - *[reasoning should be] confined to the cells that the program actually accesses — O'Hearn, Reynolds, Yang (2001)*

Then it became apparent that SL could be pushed much further.

Then it became apparent that SL could be pushed much further.

- Concurrent Separation Logic (2004–2007)
  - shared *locks* mediating access to exclusive assertions
  - guaranteed *data race freedom*

Then it became apparent that SL could be pushed much further.

- Concurrent Separation Logic (2004–2007)
  - shared *locks* mediating access to exclusive assertions
  - guaranteed *data race freedom*
- Iris (2015–2017)
  - separation never truly exists; a *fiction* of separation suffices
  - *stability* of assertions is key
  - monolithic machine state, separable *ghost state*, and *invariants*

Iris is a large and complex system ([paper](#); [lecture notes](#); [tutorial](#)).

- As of today, [145 Iris-related papers](#) listed

We wish to

- introduce just the key ideas
- give demonstrations of Iris at work

Two lectures:

- #1 (FP): basic concepts; locks; invariants
- #2 (JMM): user-defined separable ghost state

Logic involves *propositions* about an unchanging mathematical world.

A proposition has a *truth value*: it is either *true* or *false*, and forever so.

$even(1)$  — *false*

$even(2)$  — *true*

$\forall n : \mathbb{N}. \exists p : \mathbb{N}. n \leq p \wedge prime(p)$  — *true*

$\forall x : \mathbb{N}. even(x) \rightarrow odd(x + 1)$  — *true*

The rules of logic ensure that only true propositions have proofs.

# What Logic for a Changing World?

Can one make *assertions* about a changing world?

There is nobody in the street.

# What Logic for a Changing World?

Can one make *assertions* about a changing world?

There is nobody in the street.

— *may be true now*

# What Logic for a Changing World?

Can one make *assertions* about a changing world?

There is nobody in the street.

— *may be true now*

— *could become false at any time*

— *somebody could turn the corner*

— *an **unstable** assertion about a changing world*

# What Logic for a Changing World?

Can one make *stable, local* assertions about a changing world?

My room is painted white.

— *true now*

# What Logic for a Changing World?

Can one make *stable, local* assertions about a changing world?

My room is painted white.

— *true now*

— *perhaps not true forever*

— *I might decide to paint it a different color*

# What Logic for a Changing World?

Can one make *stable, local* assertions about a changing world?

My room is painted white.

— *true now*

— *perhaps not true forever*

— *I might decide to paint it a different color*

— *but no one else may do so (I own this room)*

# What Logic for a Changing World?

Can one make *stable, local* assertions about a changing world?

My room is painted white.

- *true now*
- *perhaps not true forever*
- *I might decide to paint it a different color*
- *but no one else may do so (I own this room)*
- *a **stable** assertion*
- *expressing **knowledge** about the world,*
- ***permission** to change the world,*
- *and **absence of permission** for others to change it*

Can one make *stable, local* assertions about a changing world?

I was born on a Monday.

Can one make *stable, local* assertions about a changing world?

I was born on a Monday.

— *true*

Can one make *stable, local* assertions about a changing world?

I was born on a Monday.

— *true*

— *was not true 60 years ago*

Can one make *stable, local* assertions about a changing world?

I was born on a Monday.

— *true*

— *was not true 60 years ago*

— *nobody can change this fact*

Can one make *stable, local* assertions about a changing world?

I was born on a Monday.

— *true*

— *was not true 60 years ago*

— *nobody can change this fact*

— *a **stable** assertion about a changing world*

An example of an assertion that *becomes true* at some point in time and thereafter *persists* forever.

Can one make *stable, local* assertions about a changing world?

Over 129,864,880 books have been published.

- *true*
- *was not true 60 years ago*
- *nobody can invalidate this fact*
- *a **stable** assertion about a changing world*

Can one make *stable, local* assertions about a changing world?

Over 129,864,880 books have been published.

- *true*
- *was not true 60 years ago*
- *nobody can invalidate this fact*
- *a **stable** assertion about a changing world*
- *though anyone has **permission** to publish new books*

Stable because this aspect of the world evolves in a *monotonic* way.

An assertion should

- express *knowledge* about (a fragment of) the world
- represent *permission* to change (this fragment of) the world
- represent *interdiction* for others to make incompatible changes

An assertion is *stable* if it contains *enough interdiction* to justify the knowledge and permission that it offers.

Separation Logic (SL) is a logic where *every assertion is stable*.

- SL = Stability Logic?

Separation Logic enables *local reasoning* about a composite system.

- each participant has *partial knowledge* of the world and *partial permission* to change the world
- one participant's knowledge is never invalidated by another participant's actions
- the share (knowledge and permissions) of one participant is compatible with the share of every other participant
- at all times, *the conjunction of all shares* is consistent

① What is Iris (About)?

② Basic Connectives

Conjunction

Implication

Persistence

Update

Execution

③ Mutable State

④ Locks (Primitive)

⑤ Invariants

⑥ Locks (User-Defined)

The world is partly *physical*, partly *ghost*.

Typical examples of basic assertions:

- a *physical memory cell*,  $x \mapsto v$ 
  - the points-to assertion (Reynolds, 2002)
- an *immutable* physical memory cell,  $x \mapsto_{\square} v$ 
  - the persistent points-to assertion (Friis Vindum and Birkedal, 2021)
- a *ghost memory cell*,  $\boxed{a}^{\gamma}$ 
  - new in Iris 1 (Jung et al., 2015)

I want to describe five fundamental connectives:

- *conjunction*,  $A * B$ 
  - decomposes a view of the world into several parts
- *implication*,  $A \multimap B$ 
  - change one's view of the world – not the world itself
- *persistence*,  $\Box A$ 
  - means “forever  $A$ ”
- *update*,  $\boxplus B$ 
  - changes the ghost world
  - the binary form  $A \boxRightarrow B$  is sugar for  $\Box(A * \boxplus B)$
- *execution*,  $\text{ex } s \{B\}$ 
  - changes the ghost and physical world
  - the Hoare triple  $\{A\} s \{B\}$  is sugar for  $\Box(A \multimap \text{ex } s \{B\})$

I will not discuss today:

- *pure* assertions  $\lceil P \rceil$  where  $P$  is a proposition
- *quantifiers*  $\forall x.A, \exists x.A$
- the *later* modality  $\triangleright A$
- user-defined assertions, which can
  - *inductive*: linked list (segment), tree, iterated conjunction
  - *co-inductive*
  - *guarded recursive*: `ex`

I will discuss later today:

- *locks*, first considered primitive, then user-defined
- *invariants*

## ② Basic Connectives

Conjunction

Implication

Persistence

Update

Execution

Conjunction  $A * B$  means

- $A$  holds and  $B$  holds
- and one can act on one side *without disturbing* the other—*stability*.

This is visible in the way  $\dashv$  and  $\Rightarrow$  and `ex` interact with  $*$ .

It is sometimes called “separating” conjunction

- because  $x \mapsto v * y \mapsto v'$  implies  $\lceil x \neq y \rceil$

but the key point is stability.

Conjunction is associative and commutative. *True* is its unit.

It is *not idempotent*:

- Some assertions are not duplicable: in general,  $A \not\vdash A * A$
- Every persistent assertion is duplicable:  $\Box A \vdash \Box A * \Box A$

The logic is *affine*, as opposed to linear:  $A \vdash \text{True}$ .

## ② Basic Connectives

Conjunction

**Implication**

Persistence

Update

Execution

Implication  $A \multimap B$  means:

- by *consuming*  $A$
- and by *consuming*  $A \multimap B$  as well
- you can get  $B$ .

Think of *two puzzle pieces* that fit together.

Implication changes *your view* of the world, not the world itself.

- $x \mapsto 0 \multimap \exists y. x \mapsto y \multimap \lceil 0 \leq y \rceil$
- $x \mapsto y \multimap y \mapsto z \multimap \text{listseg}(x, z)$
- $x \mapsto y \multimap (y \mapsto z \multimap \text{listseg}(x, z))$

Here is one formulation (Ishtiaq and O'Hearn, 2001):

$$\frac{R * A \vdash B}{R \vdash A \multimap B}$$

$$\frac{R \vdash A \multimap B \quad R' \vdash A}{R * R' \vdash B}$$

$$\frac{A \vdash A' \quad B \vdash B'}{A * B \vdash A' * B'}$$

Here is one formulation (Ishtiaq and O'Hearn, 2001):

$$\frac{R * A \vdash B}{R \vdash A \multimap B}$$

$$\frac{R \vdash A \multimap B \quad R' \vdash A}{R * R' \vdash B}$$

$$\frac{A \vdash A' \quad B \vdash B'}{A * B \vdash A' * B'}$$

This should be easier to read:

$$\begin{aligned} & R \multimap A \multimap B \\ \equiv & (R * A) \multimap B \end{aligned}$$

currying/uncurrying

$$(A \multimap B) * A \multimap B$$

application

$$\begin{aligned} & (A \multimap B) \\ \multimap & (A * R \multimap B * R) \end{aligned}$$

*stability* (frame)

$$\begin{aligned} & (R * A \multimap B) * R \\ \multimap & A \multimap B \end{aligned}$$

partial application

$A \wedge B$  is an external choice:

- you can have  $A$  *and* you can have  $B$
- but you can have *only one* of them.

$A \wedge B$  is equivalent to

$$\exists S. S * (S \multimap A) * (S \multimap B)$$

Here is a proof.

## 2 Basic Connectives

Conjunction

Implication

**Persistence**

Update

Execution

$\Box A$  means that  $A$  is forever true.

An assertion is *persistent* if it can be written in the form  $\Box A$ .

- by definition,  $Persistent(P)$  means  $P \vdash \Box P$

Intuitively, a proof of  $\Box A$  is a proof of  $A$  that uses persistent facts only.

$$\frac{\Box A \vdash B}{\Box A \vdash \Box B}$$

introduction

$$\frac{\Box A}{\neg^* A}$$

elimination

## ② Basic Connectives

Conjunction

Implication

Persistence

**Update**

Execution

A binary update  $A \Rightarrow B$  means:

- by consuming  $A$
- and by *changing the world*
- you can get  $B$ .

It is sugar for  $\Box(A \multimap \Rightarrow B)$ .

A unary update  $\Rightarrow B$  means

- permission to *change the world* to get  $B$ .

Caveat: there are several notions of update; I am blurring the distinction.

An update is used to allocate a new ghost cell:

- $True \Rightarrow \exists \gamma. [a]^\gamma$

and to update a ghost cell (simplified rule—see JMM's lecture):

- $[a]^\gamma \Rightarrow [b]^\gamma$

An update is used to open and close an invariant (later today).

Binary update behaves very much like implication:

$$R \Rightarrow A \Rightarrow B \\ \equiv (R * A) \Rightarrow B$$

currying/uncurrying

$$(A \Rightarrow B) * A \Rightarrow B$$

application

$$(A \Rightarrow B) \\ -* (A * R \Rightarrow B * R)$$

*stability* (frame)

$$(R * A \Rightarrow B) * R \\ -* A \Rightarrow B$$

partial application

It is easier to remember just the laws of unary update:

$$A \rightarrow * \Rightarrow A$$

return (reflexivity)

$$\Rightarrow \Rightarrow A \rightarrow * \Rightarrow A$$

join (transitivity)

$$A \rightarrow * B \rightarrow * (\Rightarrow A) \rightarrow * (\Rightarrow B)$$

covariance (map)

$$A \rightarrow * (\Rightarrow B) \rightarrow * \Rightarrow (A \rightarrow * B)$$

*stability* (strength)

One sums up these laws by saying: unary update is a strong monad.

Update *does not commute* with universal quantification:

$$\forall x. \boxplus A \not\vdash \boxplus \forall x. A$$

An intuitive explanation is: a ghost cell can be updated *in any way* you wish but not *in all ways* at once:

$$\begin{array}{l} \boxed{a}^\gamma \quad \text{entails} \quad \forall b. \boxplus \boxed{b}^\gamma \\ \boxed{a}^\gamma \quad \text{does not entail} \quad \boxplus \forall b. \boxed{b}^\gamma \end{array}$$

This is the same reason why the *value restriction* exists in ML.

This also explains the lack of an *intersection rule* in Iris:

$$\forall x. \text{ex } e \{A\} \not\vdash \text{ex } e \{\forall x. A\}$$

## ② Basic Connectives

Conjunction

Implication

Persistence

Update

Execution

So far everything has been about logic, not about programming.

Now assume that *a programming language* (syntax, semantics) is given.

For example, it might be

- a WHILE language, whose statements return no result;
- a  $\lambda$ -calculus, whose expressions return a value.

The assertion  $\text{ex } s \{B\}$  means

- there is *permission* to *execute* the statement  $s$  *once*
- this is safe (execution won't crash)
- this may change the (physical and ghost) world
- and if/once execution terminates,  $B$  will hold.

In the Iris literature,  $\text{ex}$  is named  $wp$  for “weakest precondition”.

In **dynamic logic** (Pratt, 1974) it is written  $[s]B$ .

I like  $\text{ex } s \{B\}$  because it can seem to mean

- “out of  $s$  one gets  $B$ ”
- “executing  $s$  establishes  $B$ ”

The Hoare triple, or Separation Logic triple,

$$\{A\} s \{B\}$$

is sugar for

$$\Box(A \ast \text{ex } s \{B\})$$

An update is a special case of an execution assertion.

$$\begin{aligned} & \models B \\ \equiv & \text{ex skip } \{B\} \\ & \text{skip (return)} \end{aligned}$$

$$\begin{aligned} & \text{ex } s_1 \{ \text{ex } s_2 \{B\} \} \\ \equiv & \text{ex } (s_1; s_2) \{B\} \\ & \text{sequencing (join)} \end{aligned}$$

$$\begin{aligned} & A \multimap B \\ \multimap & \text{ex } s \{A\} \multimap \text{ex } s \{B\} \\ & \text{weakening (map)} \end{aligned}$$

$$\begin{aligned} & A \multimap \text{ex } s \{B\} \\ \multimap & \text{ex } s \{A \multimap B\} \\ & \textit{stability} / \textit{frame} \text{ (strength)} \end{aligned}$$

Execution absorbs updates:

$$\begin{array}{c} \Rightarrow \text{ex } s \{B\} \\ -* \text{ex } s \{B\} \end{array}$$

update before execution

$$\begin{array}{c} \text{ex } s \{\Rightarrow B\} \\ -* \text{ex } s \{B\} \end{array}$$

update after execution

These laws because the *definition* of `ex` involves  $\Rightarrow$ .

*Structured parallel composition* and *thread creation* are easy to describe:

$$\begin{array}{l}
 \text{ex } s_1 \{B_1\} * \text{ex } s_2 \{B_2\} \\
 \rightarrow * \text{ex } (s_1 \parallel s_2) \{B_1 * B_2\} \\
 \text{fork / join}
 \end{array}
 \qquad
 \begin{array}{l}
 \text{ex } s \{B\} \\
 \rightarrow * \text{ex } (\text{fork } s) \{True\} \\
 \text{fork}
 \end{array}$$

The second rule offers no way of waiting for the child thread to finish so as to obtain  $B$ . It is up to the user to implement such a mechanism using channels, references, etc.

If the programming language has *expressions* (which return values) then one uses  $\text{ex } e \{ \psi \}$  where  $\psi : Val \rightarrow iProp$ .

$\text{ex } e \{ y.B \}$  means

- there is *permission* to *execute* the expression *e* *once*
- and (if it terminates then) it returns a value *y* such that *B* holds.

The skip and sequencing rules become

$$\begin{array}{l}
 \Rightarrow \psi \ v \\
 \equiv \text{ex } v \ \{\psi\} \\
 \text{return}
 \end{array}
 \qquad
 \begin{array}{l}
 \text{ex } e_1 \ \{v. \ \text{ex } [v/x]e_2 \ \{\psi\}\} \\
 \equiv \text{ex } (\text{let } x = e_1 \ \text{in } e_2) \ \{\psi\} \\
 \text{bind}
 \end{array}$$

These rules are used to reason *step by step* about a program.

They allow *symbolic execution* inside the proof assistant.

- ① What is Iris (About)?
- ② Basic Connectives
- ③ Mutable State**
- ④ Locks (Primitive)
- ⑤ Invariants
- ⑥ Locks (User-Defined)

The assertion  $x \mapsto v$  describes a *reference* (a mutable memory block).

This assertion means:

- the reference at address  $x$  *currently contains* the value  $v$
- *permission* to read and write this reference
- interdiction for anyone else to read or write this reference

This assertion is *exclusive*:  $x \mapsto v * x \mapsto v \vdash \text{False}$ .

More generally, there is *separation*:  $x \mapsto v * y \mapsto v' \vdash \lceil x \neq y \rceil$ .

This assertion is *not duplicable*:  $x \mapsto v \not\vdash x \mapsto v * x \mapsto v$ .

References can be *allocated*, *read*, and *written*.

$\text{ex } (\text{ref } v) \{v'. \exists l. \lceil v' = l \rceil * l \mapsto v\}$   
create

$l \mapsto v$   
 $* \text{ex } (!l) \{v'. \lceil v' = v \rceil * l \mapsto v\}$   
read

$l \mapsto v$   
 $* \text{ex } (l := v') \{_. l \mapsto v'\}$   
write

In a language without GC, there would be a *deallocation* operation.

# Operations on References, Texan Style

This postcondition-passing style makes the rules easier to apply:

$$\begin{array}{ccc} \text{True} * (\forall l. l \mapsto v * \psi l) & & l \mapsto v * (l \mapsto v * \psi v) \\ \text{* ex (ref } v \text{) } \{ \psi \} & & \text{* ex (!} l \text{) } \{ \psi \} \\ \text{create} & & \text{read} \\ \\ & & l \mapsto v * (l \mapsto v' * \psi ()) \\ \text{* ex (} l := v' \text{) } \{ \psi \} & & \\ & & \text{write} \end{array}$$

Instead of saying: the postcondition of `write` is  $l \mapsto v'$ ,  
say: it is anything you want, provided it is implied by  $l \mapsto v'$ .

# Making a Reference Immutable

A mutable reference can be forever turned into an immutable one.

$$\begin{array}{ccc} l \mapsto v & & l \mapsto_{\square} v \\ \Rightarrow l \mapsto_{\square} v & \rightarrow * & \text{ex } (!l) \{v'. \lceil v' = v \rceil\} \\ \text{freeze} & & \text{read frozen} \end{array}$$

The two views cannot co-exist:  $l \mapsto v * l \mapsto_{\square} v'$  implies *False*.

A write access to a reference requires a mutable points-to assertion.

A read access requires a (mutable or immutable) points-to assertion.

Therefore a write and a read *can never* be simultaneously enabled!

- *Data-race freedom* is guaranteed. (Good!)
- Communication between threads is *impossible*. (Bad!)

These points hold even if *read-modify-write* operations (FAA, CAS, etc.) are allowed, as they also require an exclusive points-to assertion.

To allow threads to interact, one must introduce

- synchronisation primitives: for example, *locks*; or
- shared *invariants*.

- ① What is Iris (About)?
- ② Basic Connectives
- ③ Mutable State
- ④ Locks (Primitive)**
- ⑤ Invariants
- ⑥ Locks (User-Defined)

In OCaml, an abstract type of locks could look like this:

```
type lock
(* a lock can be shared between several threads *)
val newlock : unit -> lock
val acquire : lock -> unit (* acquire access permission *)
val release : lock -> unit (* release access permission *)
```

The type-checker does not know *what data structure* a lock protects, so cannot check that acquire and release are correctly used.

A stack, protected by a lock, could look like this:

```
type 'a stack =  
  { data: 'a list ref; lock: lock } (* lock protects data *)  
  
let make () =  
  let data = ref [] in  
  let lock = newlock() in  
  { data; lock }  
  
let push x stack =  
  acquire stack.lock;           (* acquire permission *)  
  stack.data := x :: !stack.data; (* access the data *)  
  release stack.lock           (* release permission *)
```

To verify the safety of this code, *reasoning rules for locks* are needed.

There exists  $isLock : Val \rightarrow iProp \rightarrow iProp$  such that:

$$\begin{array}{l}
 \text{Persistent}(isLock \ v \ R) \\
 \text{share} \\
 \end{array}
 \quad
 \begin{array}{l}
 R \\
 \rightarrow * \text{ ex } (\text{newlock}()) \{v. isLock \ v \ R\} \\
 \text{create} \\
 \end{array}$$
  

$$\begin{array}{l}
 isLock \ v \ R \\
 \rightarrow * \text{ ex } (\text{acquire } v) \{R\} \\
 \text{acquire} \\
 \end{array}
 \quad
 \begin{array}{l}
 isLock \ v \ R * R \\
 \rightarrow * \text{ ex } (\text{release } v) \{True\} \\
 \text{release} \\
 \end{array}$$

From the user's point of view, acquire *produces*  $R$ ; release *consumes*  $R$ .

A stack, protected by a lock, could look like this:

```
type 'a stack =  
  { data: 'a list ref; lock: lock } (* lock protects data *)  
  
let make () =  
  let data = ref [] in  
  let lock = newlock() in  
  { data; lock }  
  
let push x stack =  
  acquire stack.lock;           (* acquire permission *)  
  stack.data := x :: stack.data; (* access the data *)  
  release stack.lock           (* release permission *)
```

See [a proof of safety](#) of this code.

The permission to access the data appears only within critical sections

- between release and acquire

so *data-race freedom* is still guaranteed

- even though interactions between threads are now possible

One could improve this Lock API in several ways:

- separating the *creation* of the lock and the *initialization* of the assertion  $R$
- use *fractions* to keep track of sharing and allow *canceling* a lock whose fraction is 1
- introduce an assertion  $isLocked\ v$  to *prevent releasing a lock that one does not hold*
  - under our API, such a mistake is possible if  $R * R \not\vdash False$
- view  $isLocked\ v$  as an *obligation* to eventually release the lock
  - current Iris does not allow this, as it is affine
  - current Iris does not guarantee absence of deadlocks

In this formulation, acquire yields a *unique permission* to release:

$$\begin{array}{ccc}
 \text{Persistent}(isLock \ v \ R) & \xrightarrow{R} & \text{ex} \ (\text{newlock}()) \ \{v. \ isLock \ v \ R\} \\
 \text{share} & & \text{create}
 \end{array}$$

$$\begin{array}{c}
 isLock \ v \ R \\
 \xrightarrow{*} \ \text{ex} \ (\text{acquire } v) \ \{R \ * \ (R \ \xrightarrow{*} \ \text{ex} \ (\text{release } v) \ \{True\})\} \\
 \text{acquire then release}
 \end{array}$$

This prevents releasing a lock that one does not hold.

The proof is left as an *exercise*.

In this formulation, the assertion  $isLock\ v\ R$  is not needed.

$$\begin{array}{l}
 R \\
 \text{ex}(\text{newlock}()) \\
 \rightarrow * \left\{ v. \square \left( \begin{array}{l} \text{ex}(\text{acquire } v) \\ \{ R * (R \rightarrow * \text{ex}(\text{release } v) \{ True \}) \} \end{array} \right) \right\} \\
 \text{create then forever (acquire then release)}
 \end{array}$$

The entire Lock API is described by just one rule!

The proof is left as an **exercise**.

- ① What is Iris (About)?
- ② Basic Connectives
- ③ Mutable State
- ④ Locks (Primitive)
- ⑤ Invariants**
- ⑥ Locks (User-Defined)

We have described *locks* as primitive objects that allow synchronization.

But there are many more:

- semaphores,
- barriers,
- condition variables,
- channels (concurrent FIFO queues),
- concurrent data structures of all kinds.

We want to *construct* and *verify* them, not view them all as primitive.

To describe a runtime mechanism that involves multiple participant threads and transfers of permissions,

- ① define custom *ghost state* to represent each participant's view
- ② prove *ghost update* lemmas describing how the participants' views can evolve
- ③ install an *invariant* to relate the physical state and the ghost state

Points 1 and 2 will be covered by JMM. Now what is an invariant?

I am *not* talking about

- a data structure invariant
  - a user-defined assertion such as *isLinkedList*  $\ell$  vs
- a loop invariant
  - the precondition of a recursive function

An Iris *invariant* is an assertion that *everyone agrees to maintain, forever*.

An invariant can be

- *created* (established) at a certain point in time
  - an invariant is part of the ghost state
- *opened* (temporarily violated), then *closed* (established) again
  - everyone can *depend* on the invariant
  - everyone must *preserve* the invariant
  - violations must be short-lived: at most *one atomic instruction*
- *shared* between participants
  - an invariant is never destroyed
  - its existence can be advertised to all participants

One can dynamically *create*, *share*, *open*, and *close* invariants.

$$\begin{array}{ccc}
 P \Rightarrow \boxed{P} & \text{Persistent}(\boxed{P}) & \boxed{P} \\
 \text{create} & \text{share} & \text{--}^* \text{--}^* (P * (P \text{--}^* \text{--}^* \text{True})) \\
 & & \text{open / close}
 \end{array}$$

This is analogous to creating, sharing, acquiring, releasing a lock, but the whole thing is *ghost*—there is no runtime machinery.

These rules are *unsound*.

With these rules,  
an invariant can be *opened twice* simultaneously,  
by the same thread or by two distinct threads,  
duplicating  $P$ .

This simplified presentation differs from Iris and is not machine-checked.

Introduce a (ghost) assertion  $W$ , for *world satisfaction*.

- $W$  is a witness that all invariants in the world are satisfied (closed)
- $W$  can also be viewed as *permission to open* and exploit invariants

Restrict the rule `open / close` :

$P \Rightarrow \boxed{P}$   
creation

*Persistent*( $\boxed{P}$ )  
share

$\boxed{P} \text{ } * \text{ } W \Rightarrow (P * (P * \text{ } \Rightarrow \text{ } W))$   
open / close

Now, the question is,

- how can the token  $W$  be *obtained*?
- when and how must it be *surrendered*?

Now, the question is,

- how can the token  $W$  be *obtained*?
- when and how must it be *surrendered*?

We want  $W$  to appear/disappear before/after every *atomic expression*.

One can think of  $W$  as a token that is

- given by the scheduler to the active thread
- taken from the active thread by the scheduler

Parameterize the *execution* assertion with a *mask*  $m \in \{0, 1\}$ .

- $ex_0$   $e \{ \psi \}$  means  $e$  is safe even if some invariants are violated
  - interleaving with other threads forbidden
  - $e$  must be atomic
- $ex_1$   $e \{ \psi \}$  means  $e$  is safe provided all invariants hold
  - the proof can exploit (open and close) invariants
  - interleaving with other threads permitted

All of the rules for  $ex_m$  are polymorphic in  $m$  *except sequencing*, which requires  $m = 1$ :

$$\begin{array}{l} \lceil m = 1 \rceil \\ \rightarrow * \quad ex_m \ e_1 \ \{v. \ ex_m \ [v/x]e_2 \ \{\psi\}\} \\ \rightarrow * \quad ex_m \ (\text{let } x = e_1 \ \text{in } e_2) \ \{\psi\} \\ \text{bind} \end{array}$$

In other words,  $ex_0$  cannot reason about composite expressions; it is restricted to *atomic expressions*.

$ex_0$  and  $ex_1$  are related as follows:

$$\begin{array}{ccc}
 \begin{array}{c}
 \boxed{ex_0} \in \{\psi\} \\
 \rightarrow * \boxed{ex_1} \in \{\psi\} \\
 \text{weaken}
 \end{array}
 &
 &
 \begin{array}{c}
 (W \rightarrow * \boxed{ex_0} \in \{W * \psi\}) \\
 \rightarrow * \boxed{ex_1} \in \{\psi\} \\
 \text{atomic}
 \end{array}
 \end{array}$$

The second rule states that *during the execution of an atomic expression* the token  $W$  appears out of thin air.

By combining the previous rules, we obtain a simpler **open / close** rule, which does not mention  $W$ .

$$\begin{array}{l} \boxed{P} \\ \rightarrow * (P \rightarrow * \text{ex}_0 \text{ e } \{P * \psi\}) \\ \rightarrow * \text{ex}_1 \text{ e } \{\psi\} \\ \text{open / close} \end{array}$$

By combining the previous rules, we obtain a simpler **open / close** rule, which does not mention  $W$ .

$$\begin{array}{l}
 \boxed{P} \\
 \text{--}^* (P \text{ --}^* \text{ex}_0 \text{ e } \{P * \psi\}) \\
 \text{--}^* \text{ex}_1 \text{ e } \{\psi\} \\
 \text{open / close}
 \end{array}$$

Imagine  $e$  is a memory access (read, write, CAS, etc.). Then

- $P$  can be exploited to obtain  $\ell \mapsto v$  and *justify this access*
- the updated assertion  $\ell \mapsto v'$  must be used to reconstruct  $P$  thereby *proving that the invariant is preserved*

For example, specializing the rule for a write:

$$\begin{array}{l}
 \ell \mapsto v \\
 \rightarrow * \text{ex}_0 (\ell := v') \{ \_ . \ell \mapsto v' \} \\
 \text{write}
 \end{array}
 \qquad
 \begin{array}{l}
 \boxed{P} \\
 \rightarrow * (P \rightarrow * \text{ex}_0 e \{P * \psi\}) \\
 \rightarrow * \text{ex}_1 e \{ \psi \} \\
 \text{open / close}
 \end{array}$$

yields the following rule:

$$\begin{array}{l}
 \boxed{P} \\
 \rightarrow * (P \rightarrow * \exists v. \ell \mapsto v * (\ell \mapsto v' \rightarrow * P * \psi ())) \\
 \rightarrow * \text{ex}_1 (\ell := v') \{ \psi \} \\
 \text{open / close across a write}
 \end{array}$$

Invariants are a form of *higher-order ghost state*:

- assertions about the (physical and ghost) heap
- stored inside the (ghost) heap

In combination with ghost state,  
the rules that I have sketched are still *unsound*.

Two known paradoxes involve (roughly)

- storing at ghost address  $\gamma$  the proposition:  
“the proposition stored at address  $\gamma$  is false”
- creating an invariant whose content is the proposition:  
“it is impossible to initialize all invariants”

To avoid these paradoxes, the invariant opening rule must be weakened:

$$\begin{array}{ccc}
 P \Rightarrow \boxed{P} & \text{Persistent}(\boxed{P}) & \text{* } \boxed{P} \\
 \text{create} & \text{share} & W \Rightarrow (\triangleright P * (\triangleright P \text{* } \vdash W)) \\
 & & \text{open / close}
 \end{array}$$

$P$  implies  $\triangleright P$ . The converse is false.

This prevents circular arguments where an invariant is exploited as part of its own initialization.

One defines `ex` so that every time one step of computation is taken  $\triangleright P$  can be transformed into  $P$ .

The rules that I have sketched can open *only one invariant at a time*.

- Opening an invariant consumes  $W$ .
- But, to open a second invariant,  $W$  is needed.

Iris has more complex rules, where a mask is not just one bit but a function of an infinite set of *names* to bits.

Then one can open two invariants simultaneously provided they have distinct names.

- ① What is Iris (About)?
- ② Basic Connectives
- ③ Mutable State
- ④ Locks (Primitive)
- ⑤ Invariants
- ⑥ Locks (User-Defined)

A spin lock can be implemented as follows:

```
type lock = bool Atomic.t (* true if lock is held *)  
let newlock() = Atomic.make false  
let try_acquire lock = Atomic.compare_and_set lock false true  
let rec acquire lock = if not (try_acquire lock) then acquire lock  
let release lock = Atomic.set lock false
```

This data structure can be described by an invariant:

$$isLock\ v\ R \quad \triangleq \quad \exists l. \lceil v = l \rceil * \boxed{l \mapsto true \vee (l \mapsto false * R)}$$

Based on this definition of *isLock*,  
one can *prove* that the code satisfies the API shown earlier:

$$\begin{array}{l}
 \text{Persistent}(isLock \ v \ R) \\
 \text{share} \\
 \\
 \text{isLock } v \ R \\
 -* \text{ ex } (\text{acquire } v) \{R\} \\
 \text{acquire} \\
 \\
 isLock \ v \ R * R \\
 -* \text{ ex } (\text{release } v) \{True\} \\
 \text{release}
 \end{array}
 \quad
 \begin{array}{l}
 R \\
 -* \text{ ex } (\text{newlock}()) \{v. isLock \ v \ R\} \\
 \text{create}
 \end{array}$$

See *the proof* in all of its glory.

# That's all, folks!

Coming up next:

Everything you always wanted to know  
about ghost state but were afraid to ask