An introduction to Iris

Jean-Marie Madiot & **François Pottier**

JFLA 2026

Outline

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

A (Very Partial) History

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

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

A (Very Partial) History

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*

A (Very Partial) History

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*

This Introduction to Iris

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.

$$\begin{aligned} \mathit{even}(1) &\quad \text{— } \mathit{false} \\ \mathit{even}(2) &\quad \text{— } \mathit{true} \\ \forall n : \mathbb{N}.\ \exists p : \mathbb{N}.\ n \leq p \land \mathit{prime}(p) &\quad \text{— } \mathit{true} \\ \forall x : \mathbb{N}.\ \mathit{even}(x) \to \mathit{odd}(x+1) &\quad \text{— } \mathit{true} \end{aligned}$$

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

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*

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

My room is painted white.

— *true now*

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*

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*

More Examples of Stable Assertions

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

I was born on a Monday.

More Examples of Stable Assertions

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

I was born on a Monday.

— *true*

More Examples of Stable Assertions

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

I was born on a Monday.

— *true*

— *was not true 60 years ago*

More Examples of Stable Assertions

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.

More Examples of Stable Assertions

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*

More Examples of Stable Assertions

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.

What is a Stable Assertion?

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.

What is Separation Logic?

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

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*, $A \mathrel{-\!\!*} B$
  - change one's view of the world – not the world itself
- *persistence*, $\Box A$
  - means "forever A"
- *update*, $\models\!\!\!\Rrightarrow B$
  - changes the ghost world
  - the binary form $A \Rrightarrow B$ is sugar for $\Box(A \mathrel{-\!\!*} \models\!\!\!\Rrightarrow B)$
- *execution*, $\mathit{ex}\ s\ \{B\}$
  - changes the ghost and physical world
  - the Hoare triple $\{A\}\, s\, \{B\}$ is sugar for $\Box(A \mathrel{-\!\!*} \mathit{ex}\ s\ \{B\})$

And a Lot More

I will not discuss today:

- *pure* assertions $\ulcorner P \urcorner$ 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*

Conjunction

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 $-\!*$ and $\Rrightarrow$ and *ex* interact with $*$.

It is sometimes called "separating" conjunction

- because $x \mapsto v * y \mapsto v'$ implies $\ulcorner x \neq y \urcorner$

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: $\square A \vdash \square A * \square A$

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

Implication $A \mathrel{-\!\!*} B$ means:

- by *consuming* A
- and by *consuming* $A \mathrel{-\!\!*} B$ 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.

- $x \mapsto 0 \ \mathrel{-\!\!*}\ \exists y.\, x \mapsto y * \ulcorner 0 \leq y \urcorner$
- $x \mapsto y * y \mapsto z \ \mathrel{-\!\!*}\ \mathit{listseg}(x, z)$
- $x \mapsto y \ \mathrel{-\!\!*}\ (y \mapsto z \mathrel{-\!\!*} \mathit{listseg}(x, z))$

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

$$\frac{R * A \vdash B}{R \vdash A \mathrel{-\!\!*} B} \qquad \frac{R \vdash A \mathrel{-\!\!*} B \qquad R' \vdash A}{R * R' \vdash B} \qquad \frac{A \vdash A' \qquad B \vdash B'}{A * B \vdash A' * B'}$$

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

$$\frac{R * A \vdash B}{R \vdash A \mathrel{-\!\!*} B} \qquad \frac{R \vdash A \mathrel{-\!\!*} B \qquad R' \vdash A}{R * R' \vdash B} \qquad \frac{A \vdash A' \qquad B \vdash B'}{A * B \vdash A' * B'}$$

This should be easier to read:

$$\begin{aligned} & R \mathrel{-\!\!*} A \mathrel{-\!\!*} B \\ \equiv\ & (R * A) \mathrel{-\!\!*} B \end{aligned}$$

currying/uncurrying

$$(A \mathrel{-\!\!*} B) * A \mathrel{-\!\!*} B$$

application

$$\begin{aligned} & (A \mathrel{-\!\!*} B) \\ \mathrel{-\!\!*}\ & (A * R \mathrel{-\!\!*} B * R) \end{aligned}$$

stability (frame)

$$\begin{aligned} & (R * A \mathrel{-\!\!*} B) * R \\ \mathrel{-\!\!*}\ & A \mathrel{-\!\!*} B \end{aligned}$$

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

$$\exists S. \quad S * (S \mathrel{-\!\!*} A) * (S \mathrel{-\!\!*} B)$$

Here is a proof.

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

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

- by definition, $\mathit{Persistent}(P)$ means $P \vdash \square P$

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

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

introduction

$$\square A \twoheadrightarrow A$$

elimination

A binary update $A \Rrightarrow B$ means:

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

It is sugar for $\Box(A \twoheadrightarrow \,\models\!\!\!\Rrightarrow B)$.

A unary update $\models\!\!\!\Rrightarrow B$ 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:

- $\mathit{True} \Rrightarrow \exists \gamma.\ \boxed{a}^{\gamma}$

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

- $\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}$

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

Binary update behaves very much like implication:

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

currying/uncurrying

$$(A \Rrightarrow B) * A \Rrightarrow B$$

application

$$(A \Rrightarrow B) \mathrel{-\!\!*} (A * R \Rrightarrow B * R)$$

stability (frame)

$$(R * A \Rrightarrow B) * R \mathrel{-\!\!*} A \Rrightarrow B$$

partial application

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

$$A \twoheadrightarrow* \ \Rrightarrow A$$

return (reflexivity)

$$\Rrightarrow\Rrightarrow A \twoheadrightarrow* \ \Rrightarrow A$$

join (transitivity)

$$A \mathrel{-\!\!*} B \mathrel{-\!\!*} (\Rrightarrow A) \mathrel{-\!\!*} (\Rrightarrow B)$$

covariance (map)

$$A * (\Rrightarrow B) \mathrel{-\!\!*} \Rrightarrow (A * B)$$

stability (strength)

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

An Interesting Non-Law

Update *does not commute* with universal quantification:

$$\forall x. \Rrightarrow A \quad \not\vdash \quad \Rrightarrow \forall x. A$$

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

$$\begin{array}{ccc} \boxed{a}^{\gamma} & \text{entails} & \forall b. \Rrightarrow \boxed{b}^{\gamma} \\ \boxed{a}^{\gamma} & \text{does not entail} & \Rrightarrow \forall b. \boxed{b}^{\gamma} \end{array}$$

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

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

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

2 Basic Connectives

A Programming Language

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 λ-calculus, whose expressions return a value.

The assertion $ex\ s\ \{B\}$ 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, *ex* is named *wp* for "weakest precondition".

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

I like *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

$$\Box(A \twoheadrightarrow \mathit{ex}\ s\ \{B\})$$

An update is a special case of an execution assertion.

$$\Rrightarrow B \equiv \boxed{ex}\ skip\ \{B\}$$

skip (return)

$$\boxed{ex}\ s_1\ \{\boxed{ex}\ s_2\ \{B\}\} \equiv \boxed{ex}\ (s_1; s_2)\ \{B\}$$

sequencing (join)

$$A \mathrel{-\!\!*} B \mathrel{-\!\!*} \boxed{ex}\ s\ \{A\} \mathrel{-\!\!*} \boxed{ex}\ s\ \{B\}$$

weakening (map)

$$A * \boxed{ex}\ s\ \{B\} \mathrel{-\!\!*} \boxed{ex}\ s\ \{A * B\}$$

stability / *frame* (strength)

Execution absorbs updates:

$$\begin{array}{rl} & \Rrightarrow \boxed{ex}\ s\ \{B\} \\ \twoheadrightarrow & \boxed{ex}\ s\ \{B\} \end{array}$$

update before execution

$$\begin{array}{rl} & \boxed{ex}\ s\ \{\Rrightarrow B\} \\ \twoheadrightarrow & \boxed{ex}\ s\ \{B\} \end{array}$$

update after execution

These laws because the *definition* of $\boxed{ex}$ involves $\Rrightarrow$.

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

$$\boxed{ex}\ s_1\ \{B_1\} \ast \boxed{ex}\ s_2\ \{B_2\} \twoheadrightarrow\!\ast\ \boxed{ex}\ (s_1 \mid\mid s_2)\ \{B_1 \ast B_2\}$$

fork / join

$$\boxed{ex}\ s\ \{B\} \twoheadrightarrow\!\ast\ \boxed{ex}\ (\text{fork}\, s)\ \{True\}$$

fork

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 *ex* $e\ \{\psi\}$ where $\psi : \mathit{Val} \to \mathit{iProp}$.

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

$$\begin{array}{rl} & \Rrightarrow \psi\ v \\ \equiv & \boxed{ex}\ v\ \{\psi\} \\ & \quad \text{return} \end{array} \qquad \begin{array}{rl} & \boxed{ex}\ e_1\ \{v.\ \boxed{ex}\ [v/x]e_2\ \{\psi\}\} \\ \equiv & \boxed{ex}\ (\text{let}\, x = e_1\, \text{in}\, e_2)\ \{\psi\} \\ & \qquad \text{bind} \end{array}$$

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

They allow *symbolic execution* inside the proof assistant.

Outline

Points-To Assertions

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*: $x \mapsto v * x \mapsto v \vdash \mathit{False}$.

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

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

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

$$\boxed{ex}\ (\mathsf{ref}\,v)\ \{v'.\,\exists\ell.\,\ulcorner v' = \ell \urcorner * \underbrace{\ell \mapsto v}_{\text{create}}\}$$

$$\ell \mapsto v \quad -\!\!* \quad \boxed{ex}\ (!\ell)\ \{v'.\,\ulcorner v' = v \urcorner * \underbrace{\ell \mapsto v}_{\text{read}}\}$$

$$\ell \mapsto v \quad -\!\!* \quad \boxed{ex}\ (\ell := v')\ \{\_.\,\underbrace{\ell \mapsto v'}_{\text{write}}\}$$

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:

$$\begin{array}{l} \quad \mathit{True} * (\forall \ell.\ \ell \mapsto v \mathrel{-\!\!*} \psi\,\ell) \\ \mathrel{-\!\!*}\ \ \boxed{ex}\ (\mathsf{ref}\, v)\ \{\psi\} \end{array} \quad \text{create}$$

$$\begin{array}{l} \quad \ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \psi\, v) \\ \mathrel{-\!\!*}\ \ \boxed{ex}\ (!\ell)\ \{\psi\} \end{array} \quad \text{read}$$

$$\begin{array}{l} \quad \ell \mapsto v * (\ell \mapsto v' \mathrel{-\!\!*} \psi\,()) \\ \mathrel{-\!\!*}\ \ \boxed{ex}\ (\ell := v')\ \{\psi\} \end{array} \quad \text{write}$$

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

Making a Reference Immutable

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

$$\begin{array}{lll} & \ell \mapsto v & \qquad \ell \mapsto_{\square} v \\ \Rrightarrow & \ell \mapsto_{\square} v & \qquad -\!\!* \;\; ex \; (!\ell) \; \{v'. \ulcorner v' = v \urcorner\} \\ & \text{freeze} & \qquad \qquad \text{read frozen} \end{array}$$

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

Data-Race Freedom

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*.

Outline

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

Lock API (OCaml)

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.

Lock Usage (OCaml)

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:

$$Persistent(isLock\ v\ R) \quad \text{share}$$

$$R \twoheadrightarrow \boxed{ex}\ (\text{newlock}())\ \{v.\ isLock\ v\ R\} \quad \text{create}$$

$$isLock\ v\ R \twoheadrightarrow \boxed{ex}\ (\text{acquire}\ v)\ \{R\} \quad \text{acquire}$$

$$isLock\ v\ R * R \twoheadrightarrow \boxed{ex}\ (\text{release}\ v)\ \{True\} \quad \text{release}$$

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

Lock Usage (OCaml)

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 $R * R \nvdash \mathit{False}$
- 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

Alternate Formulation 1

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

$$Persistent(isLock\ v\ R) \quad \text{share}$$

$$R \mathrel{-\!\!*} \boxed{ex}\ (\text{newlock}())\ \{v.\ isLock\ v\ R\} \quad \text{create}$$

$$isLock\ v\ R \mathrel{-\!\!*} \boxed{ex}\ (\text{acquire}\ v)\ \{R * (R \mathrel{-\!\!*} \boxed{ex}\ (\text{release}\ v)\ \{True\})\} \quad \text{acquire then release}$$

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.

$$R \mathrel{-\!\!*} \boxed{ex}\ (\text{newlock}()) \left\{ v.\ \Box \left(\boxed{ex}\ (\text{acquire}\ v) \ \{R * (R \mathrel{-\!\!*} \boxed{ex}\ (\text{release}\ v)\ \{\mathit{True}\})\} \right) \right\}$$

create then forever (acquire then release)

The entire Lock API is described by just one rule!

The proof is left as an exercise.

Many Synchronization Operations

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* ℓ *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.

$$\underbrace{P \Rrightarrow \boxed{P}}_{\text{create}} \qquad \underbrace{\mathit{Persistent}(\boxed{P})}_{\text{share}} \qquad \boxed{P} \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow} (P * \underbrace{(P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow} \mathit{True})}_{\text{open / close}})$$

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*.

Why 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:

$$\underbrace{P \Rrightarrow \boxed{P}}_{\text{creation}} \qquad \underbrace{\mathit{Persistent}(\boxed{P})}_{\text{share}} \qquad \underbrace{\boxed{P} \twoheadrightarrow\!\!* \; {\color{blue}W} \Rrightarrow (P * (P \twoheadrightarrow\!\!* \; \models\!\!\Rrightarrow {\color{blue}W}))}_{\text{open / close}}$$

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 *ex*ecution 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 $m = 1$:

$$\ulcorner m = 1 \urcorner$$
$$-\!\!* \; ex_m \; e_1 \; \{v.\; ex_m \; [v/x]e_2 \; \{\psi\}\}$$
$$-\!\!* \; ex_m \; (\text{let } x = e_1 \text{ in } e_2) \; \{\psi\}$$

bind

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:

$$\begin{array}{rl} & ex_0\ e\ \{\psi\} \\ \twoheadrightarrow\!\!* & ex_1\ e\ \{\psi\} \end{array}$$

weaken

$$\begin{array}{rl} & (W \twoheadrightarrow\!\!* \ ex_0\ e\ \{W * \psi\}) \\ \twoheadrightarrow\!\!* & ex_1\ e\ \{\psi\} \end{array}$$

atomic

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.

$$\begin{array}{ll} & \boxed{P} \\ \twoheadrightarrow & (P \twoheadrightarrow \mathit{ex}_0\ e\ \{P * \psi\}) \\ \twoheadrightarrow & \mathit{ex}_1\ e\ \{\psi\} \end{array}$$

open / close

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

$$\boxed{P} \twoheadrightarrow (P \twoheadrightarrow ex_0\ e\ \{P * \psi\}) \twoheadrightarrow ex_1\ e\ \{\psi\}$$

open / close

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:

$$\ell \mapsto v \;-\!\!*\; \boxed{ex_0}\; (\ell := v')\; \{\_.\, \ell \mapsto v'\}$$

write

$$\boxed{P} \;-\!\!*\; (P -\!\!*\; \boxed{ex_0}\; e\; \{P * \psi\}) \;-\!\!*\; \boxed{ex_1}\; e\; \{\psi\}$$

open / close

yields the following rule:

$$\boxed{P} \;-\!\!*\; (P -\!\!*\; \exists v.\, \ell \mapsto v * (\ell \mapsto v' -\!\!*\; P * \psi\,())) \;-\!\!*\; \boxed{ex_1}\; (\ell := v')\; \{\psi\}$$

open / close across a write

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 γ the proposition:
  "the proposition stored at address γ is false"
- creating an invariant whose content is the proposition:
  "it is impossible to initialize all invariants"

Last Attempt (Sound)

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

$$P \Rrightarrow \boxed{P} \qquad \mathit{Persistent}(\boxed{P}) \qquad \boxed{P} \twoheadrightarrow W \Rrightarrow (\triangleright P * (\triangleright P \twoheadrightarrow \Rrightarrow W))$$

create — share — open / close

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.

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
```

Locks as a User-Defined Data Structure

This data structure can be described by an invariant:

$$\textit{isLock}\ v\ R \quad \triangleq \quad \exists \ell.\ \ulcorner v = \ell \urcorner * \boxed{\ell \mapsto \textit{true} \vee (\ell \mapsto \textit{false} * R)}$$

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

$$Persistent(isLock\ v\ R) \quad \text{share}$$

$$R \twoheadrightarrow \boxed{ex}\ (\text{newlock}())\ \{v.\ isLock\ v\ R\} \quad \text{create}$$

$$isLock\ v\ R \twoheadrightarrow \boxed{ex}\ (\text{acquire}\ v)\ \{R\} \quad \text{acquire}$$

$$isLock\ v\ R * R \twoheadrightarrow \boxed{ex}\ (\text{release}\ v)\ \{True\} \quad \text{release}$$

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