Cosmo: a concurrent separation logic for Multicore OCaml

Glen Mével, Jacques-Henri Jourdan, François Pottier

September 21, 2020

Nomadic Labs, Paris

LRI & Inria, Paris, France

This talk

Our aim:

- Verifying
- fine-grained concurrent programs
- in the setting of Multicore OCaml's memory model.

Our contribution: A concurrent separation logic with views.

Multicore OCaml

Multicore OCaml: OCaml language with multicore programming.

Weak memory model for Multicore OCaml:

- Formalized in PLDI 2018.
- Two flavours of locations: “atomic”, “non-atomic”.

In traditional fine-grained concurrent separation logics...

We can assert ownership of a location and specify its value:

$$\{x \mapsto 42\}$$

$$x := 44$$

$$\{x \mapsto 44\}$$

Ownership can be shared between all threads via an invariant:

$$\boxed{\exists n \in \mathbb{N},\ x \mapsto n\ *\ n \text{ is even}} \vdash$$

$$\{\text{True}\}$$

$$x := 44$$

$$\{\text{True}\}$$

The challenge: subjectivity

With weak memory, each thread has its own **view** of memory.

Some assertions are **subjective**:

- Their validity relies on the thread's view.

Invariants are **objective**:

- They cannot share subjective assertions.

How to keep a simple and powerful enough logic?

Key idea: modeling subjectivity with views

A thread knows a subset $\mathcal{U}$ of all writes to the memory.

- Affects how the thread interacts with memory.
- $\mathcal{U}$ is the thread's **view**.

New assertions:

- $\uparrow\mathcal{U}$: we **have seen** $\mathcal{U}$, i.e. we know all writes in $\mathcal{U}$.
- $P \mathbin{@} \mathcal{U}$: having seen $\mathcal{U}$ is **objectively** enough for P to hold.

Key idea: decomposing subjective assertions

Decompose subjective assertions:

$$P \iff \exists \mathcal{U}.\ \underbrace{P \mathbin{@} \mathcal{U}}_{\text{objective}} \ * \ \underbrace{\uparrow \mathcal{U}}_{\text{subjective}}$$

Share parts via distinct mechanisms:

- $P \mathbin{@} \mathcal{U}$: via **objective invariants**, as usual.
- $\uparrow \mathcal{U}$: via **synchronization** offered by the memory model.

Our program logic

Rules of atomic locations, simplified

$x \mapsto_{\mathsf{at}} v$: the **atomic** location x **stores** the value v.

- **Sequentially consistent**.
- **Objective**.
- Standard rules:

$$\{x \mapsto_{\mathsf{at}} v\}\ x :=_{\mathsf{at}} v'\ \{\lambda(). x \mapsto_{\mathsf{at}} v'\}$$

$$\{x \mapsto_{\mathsf{at}} v\}\ !_{\mathsf{at}}\, x\ \{\lambda v'. v' = v * x \mapsto_{\mathsf{at}} v\}$$

Rules of non-atomic locations

$x \mapsto v$: **we know** the latest value v of the **non-atomic** location x.

- **Relaxed**.
- **Subjective** — cannot appear in an invariant.
- Standard rules too!

$$\{x \mapsto v\} \quad x := v' \quad \{\lambda(). \, x \mapsto v'\}$$

$$\{x \mapsto v\} \quad !\, x \quad \{\lambda v'. \, v' = v * x \mapsto v\}$$

Example: transferring an assertion through a spin lock

```
// release lock:
lock :=at false
```

```
// acquire lock:
while CAS lock false true = false
do () done
```

Example: transferring an assertion through a spin lock

$\{P\}$

```
// release lock:
lock :=at false
```

||

```
// acquire lock:
while CAS lock false true = false
do () done
```

$\{P\}$

Example: transferring an assertion through a spin lock

$\{P\}$

```
// release lock:
lock :=at false
```

||

```
// acquire lock:
      CAS lock false true
// CAS succeeds
```

$\{P\}$

Example: transferring an assertion through a spin lock

$\{P\}$
$\{\exists \mathcal{U}.\ P\,@\,\mathcal{U} * \uparrow\mathcal{U}\}$
// release lock:
`lock` $:=_{\mathtt{at}}$ `false`

||

// acquire lock:
`CAS lock false true`
// CAS succeeds
$\{\exists \mathcal{U}.\ P\,@\,\mathcal{U} * \uparrow\mathcal{U}\}$
$\{P\}$

Example: transferring an assertion through a spin lock

$\{P\}$
$\{\exists \mathcal{U}.\ P\,@\,\mathcal{U} * \uparrow \mathcal{U}\}$
// release lock:
`lock` $:=_{\mathtt{at}}$ `false`

||

// acquire lock:
`CAS lock false true`
// CAS succeeds
$\{\exists \mathcal{U}.\ P\,@\,\mathcal{U} * \uparrow \mathcal{U}\}$
$\{P\}$

- $P\,@\,\mathcal{U}$: transferred via **objective invariants**, as usual.

Example: transferring an assertion through a spin lock

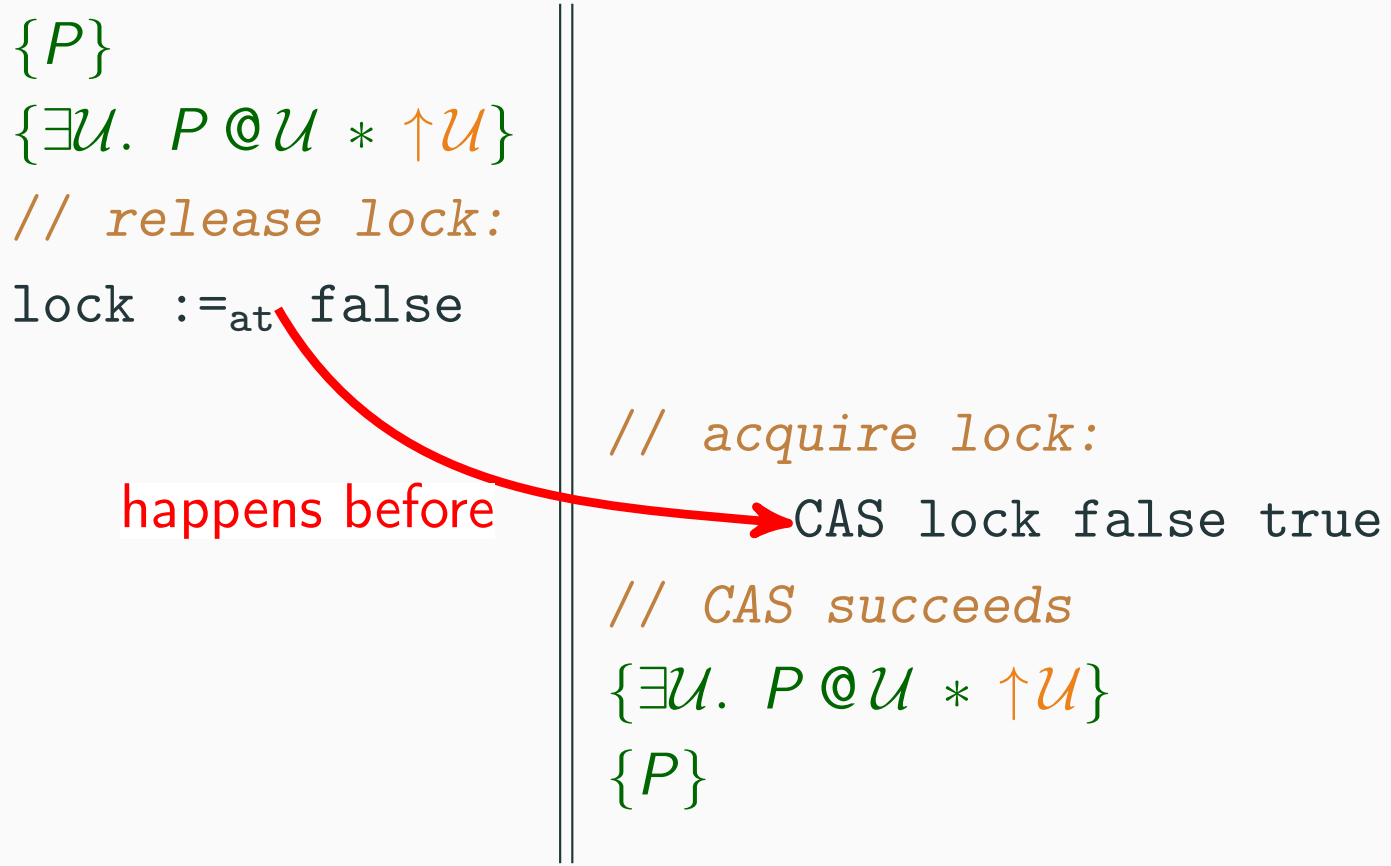
$\{P\}$
$\{\exists \mathcal{U}.\ P\,@\,\mathcal{U} * \uparrow\mathcal{U}\}$
// release lock:
`lock` $:=_{\mathtt{at}}$ `false`

‖

// acquire lock:
`CAS lock false true`
// CAS succeeds
$\{\exists \mathcal{U}.\ P\,@\,\mathcal{U} * \uparrow\mathcal{U}\}$
$\{P\}$

- $\uparrow\mathcal{U}$: transferred via synchronization.

Example: transferring an assertion through a spin lock



- $\uparrow\mathcal{U}$: transferred via **"atomic" accesses**.

Rules of atomic locations, simplified

$x \mapsto_{\mathsf{at}} v$: the atomic location x stores the value v.

- Sequentially consistent behavior for v.

- Objective.
- Rules:

$$\{x \mapsto_{\mathsf{at}} v\}\ x :=_{\mathsf{at}} v'\ \{\lambda().\ x \mapsto_{\mathsf{at}} v'\}$$

$$\{x \mapsto_{\mathsf{at}} v\}\ !_{\mathsf{at}}\, x\ \{\lambda v'.\ v' = v * x \mapsto_{\mathsf{at}} v\}$$

Rules of atomic locations

$x \mapsto_{\mathsf{at}} (v, \mathcal{U})$: the atomic location x stores the value v and a view (at least) $\mathcal{U}$.

- Sequentially consistent behavior for v.
- Release/acquire behavior for $\mathcal{U}$.
- Objective (still).
- Rules:

$$\{x \mapsto_{\mathsf{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}'\} \quad x :=_{\mathsf{at}} v' \quad \{\lambda(). \, x \mapsto_{\mathsf{at}} (v', \mathcal{U}')\}$$

$$\{x \mapsto_{\mathsf{at}} (v, \mathcal{U})\} \quad !_{\mathsf{at}} x \quad \{\lambda v'. \, v' = v * x \mapsto_{\mathsf{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}\}$$

Rules of atomic locations

$x \mapsto_{\mathsf{at}} (v, \mathcal{U})$: the atomic location x stores the value v and a view (at least) $\mathcal{U}$.

- Sequentially consistent behavior for v.
- Release/acquire behavior for $\mathcal{U}$.
- Objective (still).
- Rules:

$$\{x \mapsto_{\mathsf{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}'\} \qquad \{x \mapsto_{\mathsf{at}} (v, \mathcal{U})\}$$

$$x :=_{\mathsf{at}} v' \qquad !_{\mathsf{at}}\, x$$

$$\{\lambda(). \; x \mapsto_{\mathsf{at}} (v', \mathcal{U}')\} \qquad \{\lambda v'. \; v' = v * x \mapsto_{\mathsf{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}\}$$

release

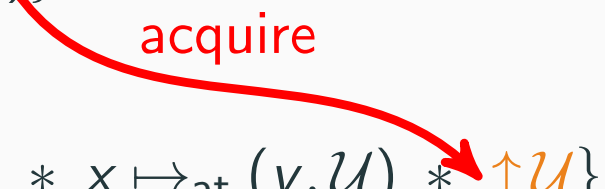
Rules of atomic locations

$x \mapsto_{\mathsf{at}} (v, \mathcal{U})$: the atomic location x stores the value v and a view (at least) $\mathcal{U}$.

- Sequentially consistent behavior for v.
- Release/acquire behavior for $\mathcal{U}$.
- Objective (still).
- Rules:

$$\{x \mapsto_{\mathsf{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}'\} \quad x :=_{\mathsf{at}} v' \quad \{\lambda(). \, x \mapsto_{\mathsf{at}} (v', \mathcal{U}')\}$$

$$\{x \mapsto_{\mathsf{at}} (v, \mathcal{U})\} \quad !_{\mathsf{at}} x \quad \{\lambda v'. \, v' = v * x \mapsto_{\mathsf{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}\}$$



Application: the spin lock

The spin lock

A spin lock implements a lock using an atomic boolean variable:

```
let release lk =
  lk :={at} false
```

```
let rec acquire lk =
  if CAS lk false true
  then ()
  else acquire lk
```

Interface:

$$\boxed{\texttt{isLock lk } P} \vdash \begin{cases} \{P\}\,\texttt{release lk}\,\{\mathsf{True}\} \\ \{\mathsf{True}\}\,\texttt{acquire lk}\,\{P\} \end{cases}$$

The spin lock

A spin lock implements a lock using an atomic boolean variable:

```
let release lk =
  lk :={at} false
```

```
let rec acquire lk =
  if CAS lk false true
  then ()
  else acquire lk
```

Invariant in traditional CSL:

$$\boxed{\texttt{lk} \mapsto_{\mathsf{at}} \texttt{true} \quad \vee \quad (\quad \texttt{lk} \mapsto_{\mathsf{at}} \texttt{false} \quad * \; P)} \vdash$$

$$\begin{cases} \{P\}\,\texttt{release lk}\,\{\mathsf{True}\} \\ \{\mathsf{True}\}\,\texttt{acquire lk}\,\{P\} \end{cases}$$

The spin lock

A spin lock implements a lock using an atomic boolean variable:

```
let release lk =
  lk :={at} false
```

```
let rec acquire lk =
  if CAS lk false true
  then ()
  else acquire lk
```

Invariant in our logic (where P is subjective!):

$$\boxed{\mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true} \quad \vee \quad (\exists \mathcal{U}.\ \mathtt{lk} \mapsto_{\mathsf{at}} (\mathtt{false}, \mathcal{U}) * P \mathbin{@} \mathcal{U})} \vdash$$

$$\begin{cases} \{P\}\ \mathtt{release\ lk}\ \{\mathsf{True}\} \\ \{\mathsf{True}\}\ \mathtt{acquire\ lk}\ \{P\} \end{cases}$$

The spin lock

A spin lock implements a lock using an atomic boolean variable:

```
let release lk =
  lk :={at} false
```

```
let rec acquire lk =
  if CAS lk false true
  then ()
  else acquire lk
```

Invariant in our logic (where P is subjective!):

$$\boxed{\texttt{lk} \mapsto_{\mathsf{at}} \texttt{true} \quad \vee \quad (\exists \mathcal{U}.\ \texttt{lk} \mapsto_{\mathsf{at}} (\texttt{false}, \mathcal{U}) * P\,@\,\mathcal{U})} \vdash$$

$$\begin{cases} \{P\}\,\texttt{release lk}\,\{\mathsf{True}\} \\ \{\mathsf{True}\}\,\texttt{acquire lk}\,\{P\} \end{cases}$$

Methodology

More case studies:

- Ticket lock
- Dekker mutual exclusion algorithm
- Peterson mutual exclusion algorithm

Method for proving correctness under weak memory:

1. Start with the invariant under sequential consistency;
2. Identify how information flows between threads;
    - i.e. where are the synchronization points;
3. Refine the invariant with corresponding views.

Conclusion and roadmap

Conclusion

Key idea: The logic of views enables concise and natural reasoning about how threads synchronize.

Proofs are fully mechanized in Coq with the Iris framework. 

What has been done:

- A high-level logic of views.
- Model of the logic on top of a lower-level logic.
- Case studies: locks, mutual exclusion.
- Proof mode *à la* Iris.
- Logical atomicity.

Future work:

- Formalize the `notify/wait` API.
- Verify more shared data structures.
- Allow data races on non-atomics.

Also done: logical atomicity

We can open invariants around **atomic** operations:

$$\frac{\{P * I\}\, e\, \{Q * I\} \quad e \text{ atomic}}{\boxed{I} \vdash \{P\}\, e\, \{Q\}}$$

Shortcoming: atomicity as a syntactical judgment.

- `acquire lk` is not **syntactically atomic**!

Done: introduce a new kind of Hoare triples such that:

$$\frac{\langle P * I\rangle\, e\, \langle Q * I\rangle}{\boxed{I} \vdash \langle P\rangle\, e\, \langle Q\rangle}$$

We can now specify operations which are **semantically atomic**, i.e. have a linearization point.

Ongoing work: more data structures

In progress: verify a concurrent FIFO with a circular buffer.

Challenges:

- Logical atomicity (done).
- Support for arrays (mostly done).

Future work: notify/wait

Multicore OCaml also provides blocking synchronization primitives:

`notify` Wake a waiting thread.

`wait` Sleep until waked.

Alternative to busy-waiting.

To do: support `notify/wait` in our logic.

We expect the following semantics (not in the PLDI 2018 paper):

`notify` Akin to a release write.

`wait` Akin to an acquire read.

Future work: data races

Our logic does not support **data races** on non-atomics.

- We cannot reason about racy programs.

However they have an operational semantics (PLDI 2018).

To do: design a high-level logic supporting data races on NAs.

Broad idea: non-atomic cells should store **monotonic** values.

Case study: parallel graph searching.

- Reuse the concurrent FIFO.

Questions?

Verifying the spin lock

```
// release lk:
```

$\{\quad \mathsf{isLock}\ \mathtt{lk}\ P\ *\ P\}$

$\{\quad \mathtt{lk} \mapsto_{\mathsf{at}} \_\quad *\ P\}$

$\left\{\exists \mathcal{U}.\ \mathtt{lk} \mapsto_{\mathsf{at}} \_\quad *\ \overbrace{\uparrow \mathcal{U}\ *\ P\ @\ \mathcal{U}}\right\}$

$\mathtt{lk} :=_{\mathsf{at}} \mathtt{false}$

$\{\exists \mathcal{U}.\ \mathtt{lk} \mapsto_{\mathsf{at}} (\mathtt{false}, \mathcal{U})\ *\ P\ @\ \mathcal{U}\}$

$\{\quad \mathsf{isLock}\ \mathtt{lk}\ P\}$

```
// acquire lk:
```

$\{\mathsf{isLock}\ \mathtt{lk}\ P\}$

$\left\{\begin{array}{l}(\exists \mathcal{U}.\ \mathtt{lk} \mapsto_{\mathsf{at}} (\mathtt{false}, \mathcal{U})\ *\ P\ @\ \mathcal{U}) \\ \vee \qquad \mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true}\end{array}\right\}$

```
if CAS lk false true
then
```

$\left\{\exists \mathcal{U}.\ \mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true}\ *\ \underbrace{\uparrow \mathcal{U}\ *\ P\ @\ \mathcal{U}}\right\}$

$\{\quad \mathtt{lk} \mapsto_{\mathsf{at}} \_\quad *\ P\}$

$\{\quad \mathsf{isLock}\ \mathtt{lk}\ P\ *\ P\}$

```
else
```

$\{\mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true}\}$

$\{\mathsf{isLock}\ \mathtt{lk}\ P\}$

```
acquire lk
```

$\{\mathsf{isLock}\ \mathtt{lk}\ P\ *\ P\}$

Model of the logic in Iris

Assertions are predicates on views:

$$\mathsf{vProp} \triangleq \mathsf{view} \longrightarrow \mathsf{iProp}$$

$$\uparrow\mathcal{U}_0 \triangleq \lambda\mathcal{U}.\ \mathcal{U}_0 \sqsubseteq \mathcal{U}$$

$$P * Q \triangleq \lambda\mathcal{U}.\ P\,\mathcal{U} * Q\,\mathcal{U}$$

$$P \mathrel{-\!\!*} Q \triangleq \lambda\mathcal{U}\ . \qquad\qquad P\,\mathcal{U} \mathrel{-\!\!*} Q\,\mathcal{U}$$

We equip a language-with-view with an operational semantics:

$$\mathsf{exprWithView} \triangleq \mathsf{expr} \times \mathsf{view}$$

Iris builds a WP calculus for exprWithView in iProp.

We derive a WP calculus for expr in vProp and prove adequacy:

$$\mathsf{WP}\ e\ \varphi \triangleq \lambda\mathcal{U}\ .$$

$$\mathsf{valid}\,\mathcal{U} \mathrel{-\!\!*} \mathsf{WP}\ \langle e, \mathcal{U}\rangle\ \left(\lambda\langle v, \mathcal{U}'\rangle.\ \mathsf{valid}\,\mathcal{U}' * \varphi\ v\ \mathcal{U}'\right)$$

where $\varphi : \mathsf{val} \longrightarrow \mathsf{vProp}$

Model of the logic in Iris

Assertions are monotonic predicates on views:

$$\mathsf{vProp} \triangleq \mathsf{view} \xrightarrow{\mathsf{mon}} \mathsf{iProp}$$

$$\uparrow \mathcal{U}_0 \triangleq \lambda \mathcal{U}.\ \mathcal{U}_0 \sqsubseteq \mathcal{U}$$

$$P * Q \triangleq \lambda \mathcal{U}.\ P\,\mathcal{U} * Q\,\mathcal{U}$$

$$P \mathrel{-\!\!*} Q \triangleq \lambda \mathcal{U}_1.\ \forall \mathcal{U} \sqsupseteq \mathcal{U}_1.\ P\,\mathcal{U} \mathrel{-\!\!*} Q\,\mathcal{U}$$

We equip a language-with-view with an operational semantics:

$$\mathsf{exprWithView} \triangleq \mathsf{expr} \times \mathsf{view}$$

Iris builds a WP calculus for exprWithView in iProp.

We derive a WP calculus for expr in vProp and prove adequacy:

$$\mathsf{WP}\ e\ \varphi \triangleq \lambda \mathcal{U}_1.\ \forall \mathcal{U} \sqsupseteq \mathcal{U}_1.$$
$$\mathsf{valid}\,\mathcal{U} \mathrel{-\!\!*} \mathsf{WP}\ \langle e, \mathcal{U} \rangle\ \left(\lambda \langle v, \mathcal{U}' \rangle.\ \mathsf{valid}\,\mathcal{U}' * \varphi\ v\ \mathcal{U}'\right)$$

where $\varphi : \mathsf{val} \to \mathsf{vProp}$

Model of the logic in Iris

Assertions are monotonic predicates on views:

$$\mathsf{vProp} \triangleq \mathsf{view} \xrightarrow{\mathsf{mon}} \mathsf{iProp}$$

$$\uparrow \mathcal{U}_0 \triangleq \lambda \mathcal{U}.\ \mathcal{U}_0 \sqsubseteq \mathcal{U}$$

$$P * Q \triangleq \lambda \mathcal{U}.\ P\,\mathcal{U} * Q\,\mathcal{U}$$

$$P \mathrel{-\!\!*} Q \triangleq \lambda \mathcal{U}_1.\ \forall \mathcal{U} \sqsupseteq \mathcal{U}_1.\ P\,\mathcal{U} \mathrel{-\!\!*} Q\,\mathcal{U}$$

We equip a language-with-view with an operational semantics:

$$\mathsf{exprWithView} \triangleq \mathsf{expr} \times \mathsf{view}$$

Iris builds a WP calculus for exprWithView in iProp.

We derive a WP calculus for expr in vProp and prove adequacy:

$$\mathsf{WP}\ e\ \varphi \triangleq \lambda \mathcal{U}_1.\ \forall \mathcal{U} \sqsupseteq \mathcal{U}_1.$$
$$\mathsf{valid}\,\mathcal{U} \mathrel{-\!\!*} \mathsf{WP}\ \langle e, \mathcal{U} \rangle\ \left(\lambda \langle v, \mathcal{U}' \rangle.\ \mathsf{valid}\,\mathcal{U}' * \varphi\ v\ \mathcal{U}'\right)$$

where $\varphi : \mathsf{val} \to \mathsf{vProp}$

Assertions are monotonic

Subjective assertions are **monotonic** w.r.t. the thread's view.

One reason is the frame rule:

$$\{x \mapsto v \ * \ P\}$$
$$\quad x := v'$$
$$\{\lambda(). \ x \mapsto v' \ * \ P\}$$

Assertions are monotonic

Subjective assertions are **monotonic** w.r.t. the thread's view.

One reason is the frame rule:

$$\{x \mapsto v \ * \ P \text{ — holds at the thread's current view}\}$$
$$x := v'$$
$$\{\lambda(). \ x \mapsto v' \ * \ P \text{ — holds at the thread's now extended view}\}$$

Decomposition of subjective assertions

This theorem allows us to decompose a subjective assertion P:

$$P \iff \exists \mathcal{U}.\ \underbrace{\uparrow \mathcal{U}}_{\text{subjective}} * \underbrace{P \mathbin{@} \mathcal{U}}_{\text{objective}}$$

We also have:

$$P \mathbin{@} \mathcal{U} \implies \text{Objectively}(\uparrow \mathcal{U} \mathrel{-\!\!*} P)$$

Decomposition of subjective assertions

This theorem allows us to decompose a subjective assertion P:

$$P \iff \exists \mathcal{U}.\ \underbrace{\uparrow \mathcal{U}}_{\text{subjective}} * \underbrace{P \mathbin{@} \mathcal{U}}_{\text{objective}}$$

We also have:

$$P \mathbin{@} \mathcal{U} \iff \text{Objectively}(\uparrow \mathcal{U} \mathrel{-\!\!*} P)$$

where $\text{Objectively}\, Q \iff (\forall \mathcal{U}.\ Q \mathbin{@} \mathcal{U}) \iff Q \mathbin{@} \varnothing$