Cosmo: a concurrent separation logic for Multicore OCaml

Glen Mével, Jacques-Henri Jourdan, François Pottier

August, 2020
ICFP, "New York"

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".

(Also at ICFP 2020: *Retrofitting Parallelism onto OCaml*)

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:

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

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

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

$\{P\}$

$\{\exists \mathcal{U}.\ P \mathbin{@} \mathcal{U} * \uparrow\mathcal{U}\}$

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

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

$\{\exists \mathcal{U}.\ P \mathbin{@} \mathcal{U} * \uparrow\mathcal{U}\}$

$\{P\}$

Example: transferring an assertion through a spin lock

$\{P\}$

$\{\exists \mathcal{U}.\ P \,@\, \mathcal{U} * \uparrow\mathcal{U}\}$

```
// release lock:
lock :=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

$\{P\}$

$\{\exists\mathcal{U}.\ P \mathbin{@} \mathcal{U} * \uparrow\mathcal{U}\}$

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

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

$\{\exists\mathcal{U}.\ P \mathbin{@} \mathcal{U} * \uparrow\mathcal{U}\}$

$\{P\}$

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

Example: transferring an assertion through a spin lock

$\{P\}$

$\{\exists \mathcal{U}.\ P \,@\, \mathcal{U} * \uparrow\mathcal{U}\}$

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

happens before

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

$\{\exists \mathcal{U}.\ P \,@\, \mathcal{U} * \uparrow\mathcal{U}\}$

$\{P\}$

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

Rules of atomic locations, simplified

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

- Sequentially consistent behavior for v.
- Objective.
- Rules:

$$\begin{array}{ll} \{x \mapsto_{\mathrm{at}} v\} & \{x \mapsto_{\mathrm{at}} v\} \\ \quad x :=_{\mathrm{at}} v' & \quad !_{\mathrm{at}}\, x \\ \{\lambda().\ x \mapsto_{\mathrm{at}} v'\} & \{\lambda v'.\ v' = v \ast x \mapsto_{\mathrm{at}} v\} \end{array}$$

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:

$$\begin{array}{ll} \{x \mapsto_{\mathsf{at}} (v, \mathcal{U}) \ast \uparrow \mathcal{U}'\} & \{x \mapsto_{\mathsf{at}} (v, \mathcal{U})\} \\ \quad x :=_{\mathsf{at}} v' & \quad !_{\mathsf{at}}\, x \\ \{\lambda().\ x \mapsto_{\mathsf{at}} (v', \mathcal{U}')\} & \{\lambda v'.\ v' = v \ast x \mapsto_{\mathsf{at}} (v, \mathcal{U}) \ast \uparrow \mathcal{U}\} \end{array}$$

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:

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

release

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

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:

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

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

acquire

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:

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

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 traditional CSL:

$$\boxed{\mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true} \quad \vee \quad (\quad \mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{false} \quad * \; P)} \vdash \begin{cases} \{P\}\, \mathtt{release\ lk}\, \{\mathsf{True}\} \\ \{\mathsf{True}\}\, \mathtt{acquire\ lk}\, \{P\} \end{cases}$$

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!):

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

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

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

Conclusion

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

In the paper:

- Model of the logic.
- A lower-level logic.
- More case studies.

Fully mechanized in Coq with the Iris framework.

Future work:

- Verify more shared data structures.
- Allow data races on non-atomics?

Questions?

Verifying the spin lock

// release lk:

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

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

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

```
lk :=at false
```

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

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

// acquire lk:

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

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

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

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

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

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

```
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:

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

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

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

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

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

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

Iris builds a WP calculus for exprWithView in iProp.

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

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

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

Model of the logic in Iris

Assertions are monotonic predicates on views:

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

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

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

$$P \twoheadrightarrow Q \triangleq \lambda\mathcal{U}_1.\, \forall\mathcal{U} \sqsupseteq \mathcal{U}_1.\, P\,\mathcal{U} \twoheadrightarrow Q\,\mathcal{U}$$

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

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

Iris builds a WP calculus for exprWithView in iProp.

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

$$\mathrm{WP}\ e\ \varphi \triangleq \lambda\mathcal{U}_1.\, \forall\mathcal{U} \sqsupseteq \mathcal{U}_1.$$
$$\mathrm{valid}\,\mathcal{U} \twoheadrightarrow \mathrm{WP}\ \langle e, \mathcal{U}\rangle\ \left(\lambda\langle v, \mathcal{U}'\rangle.\, \mathrm{valid}\,\mathcal{U}' \ast \varphi\ v\ \mathcal{U}'\right)$$

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

Model of the logic in Iris

Assertions are monotonic predicates on views:

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

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

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

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

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

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

Iris builds a WP calculus for exprWithView in iProp.

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

$$\begin{aligned}\text{WP}\; e\; \varphi \triangleq \lambda\mathcal{U}_1.\, \forall\mathcal{U} \sqsupseteq \mathcal{U}_1.\\ \text{valid}\,\mathcal{U} \mathrel{-\!\!\ast} \text{WP}\;\langle e, \mathcal{U}\rangle\; \left(\lambda\langle v, \mathcal{U}'\rangle.\, \text{valid}\,\mathcal{U}' \ast \varphi\; v\; \mathcal{U}'\right)\end{aligned}$$

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

Assertions are monotonic

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

One reason is the frame rule:

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

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:

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

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