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: 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:

\[
\exists n \in \mathbb{N}, \ x \mapsto n \times n \text{ is even} \quad \vdash
\]

\[
\{ \text{True} \}
\]

\[
x := 44
\]

\[
\{ \text{True} \} 
\]
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?
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 $\mathcal{U}$.
- $P \odot \mathcal{U}$: having seen $\mathcal{U}$ is objectively enough for $P$ to hold.
Decompose subjective assertions:

\[ P \iff \exists U. \ (P \land U) \ast \uparrow U \]

Share parts via distinct mechanisms:

- \( P \land U \): via \textit{objective invariants}, as usual.
- \( \uparrow U \): via \textit{synchronization} offered by the memory model.
Our program logic
Rules of atomic locations, simplified

\( x \mapsto_{\text{at}} v \) : the atomic location \( x \) stores the value \( v \).

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

\[
\begin{align*}
\{ x & \mapsto_{\text{at}} v \} \\
\{ x & :=_{\text{at}} v' \} \\
\{ \lambda(). \ x & \mapsto_{\text{at}} v' \} & \quad \{ x & \mapsto_{\text{at}} v \} \\
\{ !_{\text{at}} x \} & \quad \{ \lambda v'. \ v' = v \ & \ast \ & x & \mapsto_{\text{at}} v \} 
\end{align*}
\]
**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!**

\[
\begin{align*}
\{ x \mapsto v \} & \quad \{ x \mapsto v \} \\
\ x := v' & \quad !x \\
\{ \lambda(). \ x \mapsto v' \} & \quad \{ \lambda v'. \ v' = v * x \mapsto v \}
\end{align*}
\]
Example: transferring an assertion through a spin lock

// release lock:
lock := \_at \_false

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

\_Mével, Jourdan, Pottier: Cosmo: a concurrent separation logic for Multicore OCaml
Example: transferring an assertion through a spin lock

\{P\}

// release lock:
lock := \text{at} \; \text{false}

// acquire lock:
while \text{CAS} \; \text{lock} \; \text{false} \; \text{true} = \text{false}
do () done

\{P\}

\bullet \; \text{P at } U
\bullet \; \uparrow U: transferred via objective invariants, as usual.

\text{"atomic" accesses.}
Example: transferring an assertion through a spin lock

\[ \{P\} \]

// release lock:
lock := \texttt{at} false

// acquire lock:
CAS lock false true

// CAS succeeds

\[ \{P\} \]

\[ \exists U. P \land \neg U \nuparrow U \]

Mével, Jourdan, Pottier: \textit{Cosmo: a concurrent separation logic for Multicore OCaml}
Example: transferring an assertion through a spin lock

\{
{P}
\{∃U. P @ U \ast \uparrow U}\}

// release lock:
lock := \texttt{at false}

// acquire lock:
\texttt{CAS lock false true}

// CAS succeeds
\{∃U. P @ U \ast \uparrow U\}
\{P\}
Example: transferring an assertion through a spin lock

\{P\}
\{\exists U. P @ U \ast \uparrow U\}
// release lock:
lock :=_{at} \text{false}

// acquire lock:
\text{CAS lock false true = false}
// CAS succeeds
\{\exists U. P @ U \ast \uparrow U\}
\{P\}

• $P @ U$ : transferred via \textbf{objective invariants}, as usual.
Example: transferring an assertion through a spin lock

\[
\{P\}
\{\exists U. \; P @ U \; \ast \; \uparrow U\}
\]

// release lock:
lock := \texttt{at} false

// acquire lock:
CAS lock false true

// CAS succeeds
\[
\{\exists U. \; P @ U \; \ast \; \uparrow U\}
\]

\[
\{P\}
\]

- \(\uparrow U\) : transferred via synchronization.
Example: transferring an assertion through a spin lock

\[
\{P\} \\
\{\exists U. \; P @ U \; \ast \; \uparrow U\}
\]

// release lock:
\[
\text{lock} :=_{at} \text{false}
\]

\[
\text{happens before}
\]

// acquire lock:
\[
\text{CAS lock false true = false}
\]

// CAS succeeds
\[
\{\exists U. \; P @ U \; \ast \; \uparrow U\}
\]

\[
\{P\}
\]

• \(\uparrow U\): transferred via "atomic" accesses.
Rules of atomic locations, simplified

\( x \mapsto_{at} v \) : the atomic location \( x \) stores the value \( v \).

- Sequentially consistent behavior for \( v \).
- Objective.
- Rules:

\[
\begin{align*}
\{ x \mapsto_{at} v \} & \quad x :=_{at} v' \\
\{ \lambda(). \; x \mapsto_{at} v' \} & \quad \{ \lambda v'. \; v' = v * x \mapsto_{at} v \}
\end{align*}
\]
Rules of atomic locations

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

- Sequentially consistent behavior for \( v \).
- **Release/acquire** behavior for \( U \).
- Objective (still).

**Rules:**

\[
\begin{align*}
\{ x \mapsto_{\text{at}} (v, U) \} & \quad \{ x \mapsto_{\text{at}} (v, U') \} \\
\{ \lambda(). \ x \mapsto_{\text{at}} (v', U') \} & \quad \{ \lambda v'. \ v' = v \} \\
& \quad \{ x \mapsto_{\text{at}} (v, U) \} \\
\} \quad \{ x \mapsto_{\text{at}} (v, U') \}
\]

\]
$x \mapsto_{\text{at}} (v, \mathcal{U})$: 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:

\[
\begin{align*}
\{ x \mapsto_{\text{at}} (v, \mathcal{U}) \ast \uparrow \mathcal{U}' \} & \quad \{ x \mapsto_{\text{at}} (v, \mathcal{U}) \} \\
\text{release} & \\
\{ \lambda (). \ x \mapsto_{\text{at}} (v', \mathcal{U}') \} & \quad \{ \lambda v'. \ v' = v \ast x \mapsto_{\text{at}} (v, \mathcal{U}) \ast \uparrow \mathcal{U} \}
\end{align*}
\]
Rules of atomic locations

\( x \mapsto_{\text{at}} (v, \mathcal{U}) \) : 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:

\[
\begin{align*}
\{ x \mapsto_{\text{at}} (v, \mathcal{U}) \} & \downarrow \mathcal{U}' \\
\{ \lambda(). \ x \mapsto_{\text{at}} (v', \mathcal{U}') \} \\
\{ \lambda v'. \ v' = v \} & \downarrow \mathcal{U} \\
\{ \lambda v'. \ v' \neq v \} & \downarrow \mathcal{U} \\
\{ \lambda v'. \ v' = v \} & \downarrow \mathcal{U} \\
\{ \lambda v'. \ v' \neq v \} & \downarrow \mathcal{U} \\
\end{align*}
\]
Application: the spin lock
The spin lock

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

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

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

Interface:

```
isLock lk P ⊨

{P} release lk {True}
{True} acquire lk {P}
```
A spin lock implements a lock using an atomic boolean variable:

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

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

Invariant in traditional CSL:

$$lk \mapsto_{at} true \lor (lk \mapsto_{at} false \star P) \vdash$$

$${\{P\}} \text{release } lk \{\text{True}\}$$

$${\{\text{True}\}} \text{acquire } lk \{P\}$$
A spin lock implements a lock using an atomic boolean variable:

\[
\text{let release lk =} \quad \text{let rec acquire lk =} \\
\quad \text{lk :=\{at\} false} \quad \text{if CAS lk false true} \\
\quad \text{then ()} \quad \text{then ()} \quad \text{else acquire lk}
\]

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

\[
lk \mapsto_{\text{at}} \text{true} \quad \lor \quad (\exists U. lk \mapsto_{\text{at}} (\text{false}, U) \star P @ U) \\
\begin{cases} 
\{P\} \text{release lk \{True\}} \\
\{\text{True}\} \text{acquire lk \{P\}}
\end{cases}
\]
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
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:
\{ isLock \ lk \ P \ \ast \ P \} \\
\{ \ lk \xrightarrow{\text{at } \_} \ \ast \ P \} \\
\{ \exists \ U. \ lk \xrightarrow{\text{at } \_} \ \ast \ \uparrow U \ \ast \ P \ \ii \ U \} \\
lk := \xrightarrow{\text{at } \false}
\{ \exists \ U. \ lk \xrightarrow{\text{at } \false, U} \ \ast \ P \ \ii \ U \} \\
\{ \ \text{isLock} \ lk \ P \} \\

// acquire lk:
\{ isLock \ lk \ P \} \\
\{ \ (\exists U. \ lk \xrightarrow{\text{at } \false, U} \ \ast \ P \ \ii \ U) \} \\
\lor \lk \xrightarrow{\text{at } \true} \\
\ \text{if CAS} \ lk \ false \ true
\ \text{then}
\{ \exists U. \ lk \xrightarrow{\text{at true, } U} \ \ast \ \uparrow U \ \ast \ P \ \ii \ U \} \\
\{ \ lk \xrightarrow{\text{at } \_} \ \ast \ P \} \\
\{ \ \text{isLock} \ lk \ P \ \ast \ P \} \\
\ \text{else}
\{ lk \xrightarrow{\text{at } \true} \} \\
\{ \ \text{isLock} \ lk \ P \} \\
\text{acquire} \ lk \\
\{ \text{isLock} \ lk \ P \ \ast \ P \}
Model of the logic in Iris

Assertions are predicates on views:

\[ \text{vProp} \triangleq \text{view} \rightarrow \text{iProp} \]

\[ \uparrow U_0 \triangleq \lambda U. U_0 \sqsubseteq U \]

\[ P \ast Q \triangleq \lambda U. P U \ast Q U \]

\[ P \star Q \triangleq \lambda U . \quad P U \star Q U \]

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:

\[ \text{WP} e \varphi \triangleq \lambda U . \]

\[ \text{valid} U \dashv \ast \text{WP} \langle e, U \rangle \left( \lambda \langle v, U' \rangle . \text{valid} U' \ast \varphi v U' \right) \]

where \( \varphi : \text{val} \rightarrow \text{vProp} \)
Model of the logic in Iris

Assertions are **monotonic** predicates on views:

\[
\begin{align*}
\mathsf{vProp} & \triangleq \text{view } \xrightarrow{\text{mon}} \mathsf{iProp} \\
\U_0 & \triangleq \lambda U. \ U_0 \sqsubseteq U \\
P \star Q & \triangleq \lambda U. \ P U \star Q U \\
P \triangleright Q & \triangleq \lambda U_1. \ \forall U \sqsubseteq U_1. \ P U \triangleright Q U
\end{align*}
\]

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

\[
\mathsf{exprWithView} \triangleq \mathsf{expr} \times \text{view}
\]

Iris builds a WP calculus for \(\mathsf{exprWithView}\) in \(\mathsf{iProp}\).

We derive a WP calculus for \(\mathsf{expr}\) in \(\mathsf{vProp}\) and prove adequacy:

\[
\mathsf{WP} \ e \ \varphi \ \triangleq \ \lambda U_1. \ \forall U \sqsubseteq U_1. \ \text{valid} U \triangleright \mathsf{WP} \ \langle e, U \rangle \ (\lambda \langle v, U' \rangle. \ \text{valid} U' \star \varphi \ v \ U')
\]

where \(\varphi : \mathsf{val} \to \mathsf{vProp}\).
Model of the logic in Iris

Assertions are **monotonic** predicates on views:

\[ \forall \lambda_U. \forall U_0 \subseteq U \]

\[ \forall \lambda_U. P U \rightarrow Q U \]

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:

\[ \forall \lambda_U_1. \forall U \subseteq U_1. \text{valid} U \rightarrow \text{WP} \langle e, U \rangle \left( \lambda \langle v, U' \rangle. \text{valid} U' \rightarrow \varphi v U' \right) \]

where \( \varphi : \text{val} \rightarrow \text{vProp} \)
Subjective assertions are **monotonic** w.r.t. the thread’s view.

One reason is the frame rule:

\[
\begin{align*}
\{ x \mapsto v \mid P \} \\
 x := v' \\
\{ \lambda(). \ x \mapsto v' \mid P \}
\end{align*}
\]
Subjective assertions are **monotonic** w.r.t. the thread’s view.

One reason is the frame rule:

\[
\begin{align*}
\{ x \mapsto v \land P \quad &\text{— holds at the thread’s current view} \} \\
\quad \quad \quad x := v' \\
\{ \lambda().\ x \mapsto v' \land P \quad &\text{— holds at the thread’s now extended view} \}
\end{align*}
\]
This theorem allows us to decompose a subjective assertion $P$:

$$ P \iff \exists U. \uparrow U \ast P \circ U $$

subjective

objective

We also have:

$$ P \circ U \implies \text{Objectively}(\uparrow U \ast P) $$
Decomposition of subjective assertions

This theorem allows us to decompose a subjective assertion $P$:

$$ P \iff \exists U. \uparrow U \; \ast \; P \odot U $$

subjective

objective

We also have:

$$ P \odot U \iff \text{Objectively}(\uparrow U \; \ast \; P) $$

where Objectively $Q \iff (\forall U. \; Q \odot U) \iff Q \odot \emptyset$