Towards a separation logic for Multicore OCaml

Glen Mével, Jacques-Henri Jourdan, François Pottier
November 18, 2019
Gallium seminar, Inria, Paris

CNRS & Inria, Paris, France
The weak memory model
Multicore OCaml

Extension of the OCaml language with **multicore programming**. Research project at OCaml Labs (Cambridge), will be merged eventually.

**Strengths:**

- brings multicore abilities to a functional, statically typed, memory-safe programming language;
- (gives the programmer a simpler memory model than that of C11, hopefully);
- limited performance drop for sequential code.

**Goals of this PhD:**

- Build a proof system for Multicore OCaml programs.
- Prove interesting concurrent data structures.
Weak memory models

Sequential consistency is unrealistic.

We need a **weaker memory model**, where different threads have different **views** of the shared state.

The model should be specific to our language.

Existing works: Java (2000s), C11 (2010s; also Rust).

**Candidate model for Multicore OCaml:**

Dolan, Sivaramakrishnan, Madhavapeddy.  
*Bounding Data Races in Space and Time.*  
PLDI 2018.

Two access modes: non-atomic, atomic.
An operational model for Multicore OCaml: non-atomics

\[ x := x_1 \quad \| \quad y := y_1 \]
\[ A := y \quad \| \quad B := x \]
Each non-atomic location has a history, i.e. a map from timestamps to values (timestamps are per location).
x := x₁ \parallel y := y₁
A := !y \parallel B := !x

Each **non-atomic** location has a **history**, i.e. a map from timestamps to values (timestamps are per location).

Each thread has its own **view** of the non-atomic store, i.e. a map from non-atomic locations to timestamps.
An operational model for Multicore OCaml: non-atomics

\[ x := x_1 \quad | \quad y := y_1 \]
\[ A := !y \quad | \quad B := !x \]

Each non-atomic location has a history, i.e., a map from timestamps to values (timestamps are per location).

Each thread has its own view of the non-atomic store, i.e., a map from non-atomic locations to timestamps.

**non-atomic write**

- Timestamp must be fresh.
- Timestamp must be newer than current thread’s view.
- Current thread’s view is updated.
An operational model for Multicore OCaml: non-atomics

Each non-atomic location has a history, i.e., a map from timestamps to values (timestamps are per location).

Each thread has its own view of the non-atomic store, i.e., a map from non-atomic locations to timestamps.

Non-atomic write

- Timestamp must be fresh.
- Timestamp must be newer than current thread’s view.
- Current thread’s view is updated.

\[
x := x_1 \quad y := y_1 \\
A := !y \quad B := !x
\]
An operational model for Multicore OCaml: non-atomics

Each non-atomic location has a history, i.e. a map from timestamps to values (timestamps are per location).

Each thread has its own view of the non-atomic store, i.e. a map from non-atomic locations to timestamps.

non-atomic read

- Returns any value at least as recent as current thread’s view.
- Current thread’s view is unchanged.
An operational model for Multicore OCaml: non-atomics

\[
x := x_1 \quad y := y_1 \\
A := !y \quad B := !x
\]

Each non-atomic location has a history, i.e. a map from timestamps to values (timestamps are per location).

Each thread has its own view of the non-atomic store, i.e. a map from non-atomic locations to timestamps.

**non-atomic read**

- Returns **any** value at least as recent as current thread’s view.
- Current thread’s view is unchanged.
Non-atomic locations are useful for updating the state locally, but they don’t provide synchronization.

Atomic locations allow the message-passing idiom.
An operational model for Multicore OCaml: atomics

\[
\begin{align*}
  x & := x_1 \\
  a & :=_{\text{at}} \text{true} \\
  \text{REPEAT} & \\
  C & := !_{\text{at}} a \\
  \text{UNTIL} & \ C = \text{true} \\
  B & := !x
\end{align*}
\]

thread B    thread A

\[
\begin{align*}
  x & : x_0 \\
  a & : \text{false}
\end{align*}
\]
An operational model for Multicore OCaml: atomics

\[ \begin{align*}
x & := x_1 \\
\text{a} & := \text{at} \text{ true} \\
\text{REPEAT} & \\
\text{C} & := !\text{at} \text{ a} \\
\text{UNTIL} & \text{ C } = \text{ true} \\
\text{B} & := !x
\end{align*} \]

Each **atomic** location stores **one** value, and **one view** of the non-atomic store.
An operational model for Multicore OCaml: atomics

\[
x := x_1 \\
a :=_{at} true
\]

\[
\text{REPEAT} \\
\quad C := !_a a \\
\quad \text{UNTIL } C = true \\
\quad B := !_x x
\]

Each atomic location stores one value, and one view of the non-atomic store.
An operational model for Multicore OCaml: atomics

\[
x := x_1 \\
a :=_{at} \text{true}
\]

\[
\begin{align*}
&\text{REPEAT} \\
&\quad C := !_{at} a \\
&\quad \text{UNTIL} \ C = \text{true} \\
&\quad B := !x
\end{align*}
\]
An operational model for Multicore OCaml: atomics

\[
x := x_1 \\
a :=_{at} \text{true}
\]

\[
\text{REPEAT} \\
\quad C := !_{at} a \\
\text{UNTIL } C = \text{true} \\
B := !x
\]

Each atomic location stores one value, and one view of the non-atomic store.

**atomic write**

Merges the writer’s view into the atomic location’s view.
An operational model for Multicore OCaml: atomics

Each atomic location stores one value, and one view of the non-atomic store.

atomic read

Merges the atomic location’s view into the reader’s view.
An operational model for Multicore OCaml: atomics

\[
\begin{align*}
  x &:= x_1 \\
  a &:=_{\text{at}} \text{true} \\
  \text{REPEAT} \\
  C &:= !_{\text{at}} a \\
  \text{UNTIL} C = \text{true} \\
  B &:= !x
\end{align*}
\]

Each atomic location stores one value, and one view of the non-atomic store.
Our program logic
The predicate $x \mapsto v$ means that we own the non-atomic location $x$ and that we have seen its latest value, which is $v$.

**Non-atomic write:**

\[
\{x \mapsto v\} \\
{x} := v' \\
\{\lambda(). \ x \mapsto v'\}
\]

**Non-atomic read:**

\[
\{x \mapsto v\} \\
!x \\
\{\lambda v'. \ v' = v * x \mapsto v\}
\]
Invariants are the mechanism by which threads can share propositions in a Concurrent Separation Logic such as Iris:

\[
\begin{align*}
\{P \times I\} & \longrightarrow \{Q \times I\} & \text{e atomic} \\
\{I\} & \vdash \{P\} \text{ e } \{Q\}
\end{align*}
\]

The proposition \(x \mapsto v\) is subjective: its truth depends on the thread’s view of memory.

It is unsound to share it via an invariant.

Propositions which are true in all threads are called objective:

- “pure” facts, such as \(v = 5\)
- ghost state, such as \(\gamma \mapsto \circ 5\)
- atomic state, such as \(a \mapsto_{\text{at}} (v, V)\)

Only objective propositions can be put in an invariant.
The predicate $a \mapsto_{\text{at}} v$ means that we own the atomic location $a$, which stores the value $v$.

It is **objective**.

**Atomic write:**

$$\{a \mapsto_{\text{at}} v\}$$

$$a :=_{\text{at}} v'$$

$$\{\lambda().\ a \mapsto_{\text{at}} v'\}$$

**Atomic read:**

$$\{a \mapsto_{\text{at}} v\}$$

$$\!_{\text{at}} a$$

$$\{\lambda v'.\ v' = v * a \mapsto_{\text{at}} v\}$$
Rules of atomic locations (simplified)

The predicate $a \mapsto_{\text{at}} v$ means that we own the atomic location $a$, which stores the value $v$.

It is **objective**.

**Atomic write:**

\[
\{a \mapsto_{\text{at}} v\} \\
\text{a :=}_{\text{at}} v' \\
\{\lambda().\ a \mapsto_{\text{at}} v'\}
\]

**Atomic read:**

\[
\{a \mapsto_{\text{at}} v\} \\
\text{!}_{\text{at}} a \\
\{\lambda v'.\ v' = v * a \mapsto_{\text{at}} v\}
\]

**views**

Views are ordered by inclusion.

The predicate $\uparrow V$ means “the current thread's view includes $V$”.

8
The predicate \( a \mapsto_{at} (v, V) \) means that we own the atomic location \( a \), which stores the value \( v \) and a view (at least) \( V \).

It is objective.

**Atomic write:**

\[
\{ a \mapsto_{at} (v, V) \} \star \uparrow V'
\]

\( a :=_{at} v' \)

\[
\{ \lambda(). \ a \mapsto_{at} (v', V') \}
\]

**Atomic read:**

\[
\{ a \mapsto_{at} (v, V) \}
\]

\( \!_{at} a \)

\[
\{ \lambda v'. \ v' = v \star a \mapsto_{at} (v, V) \star \uparrow V' \}
\]
Rules of atomic locations

The predicate $a \mapsto_{at} (v, V)$ means that we own the atomic location $a$, which stores the value $v$ and a view (at least) $V$.

It is **objective**.

**Atomic write:**

$$\{ a \mapsto_{at} (v, V) \} \quad \text{and} \quad \text{↑}_{V'}$$

$$a :=_{at} v'$$

$$\{ \lambda(). \ a \mapsto_{at} (v', V' \sqcup V) \} \quad \text{and} \quad \text{↑}_{V}$$

**Atomic read:**

$$\{ a \mapsto_{at} (v, V) \}$$

$$\text{!}_{at} a$$

$$\{ \lambda v'. \ v' = v \} \quad \text{and} \quad \{ a \mapsto_{at} (v, V) \} \quad \text{and} \quad \text{↑}_{V}$$

**views**

Views are ordered by inclusion.

The predicate $\text{↑}_{V}$ means “the current thread’s view includes $V$”.

8
The objective proposition “$P$ at $\mathcal{V}$” is the subjective proposition $P$ seen at a fixed view $\mathcal{V}$.

$$P \iff \exists \mathcal{V}. \ (\uparrow \mathcal{V}) \ast (P \text{ at } \mathcal{V})$$

$(\Rightarrow)$ If $P$ holds now, then it holds at the current view.

$(\Leftarrow)$ If $P$ holds at some earlier view, then it holds now.
The message passing idiom

\[ P \iff \exists V. (\uparrow V) \ast (P \text{ at } V) \]
The message passing idiom

\[ P \iff \exists \mathcal{V}. (\uparrow \mathcal{V}) \ast (P \text{ at } \mathcal{V}) \]

\[
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \} \\
\{ a \mapsto_{\text{at}} (\text{true}, \mathcal{V}) \ast P \}
\]
The message passing idiom

\[ P \iff \exists \mathcal{V}. (\uparrow \mathcal{V}) \ast (P \text{ at } \mathcal{V}) \]

\[
\begin{align*}
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \} \\
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \text{ at } \mathcal{V} \ast \uparrow \mathcal{V} \}
\end{align*}
\]

\[
\begin{align*}
\{ a \mapsto_{\text{at}} (\text{true}, \mathcal{V}) \ast P \}
\end{align*}
\]
The message passing idiom

\[ P \iff \exists V. (\uparrow V) \ast (P \text{ at } V) \]

\[
\begin{align*}
\{ a \mapsto \text{at}(\text{false}, \emptyset) \ast P \} \\
\{ a \mapsto \text{at}(\text{false}, \emptyset) \ast P \text{ at } V \ast \uparrow V \}
\end{align*}
\]

\[
\begin{align*}
a &:= \text{at} \; \text{true} \\
\{ a \mapsto \text{at}(\text{true}, V) \ast P \text{ at } V \}
\end{align*}
\]

\[
\begin{align*}
\{ a \mapsto \text{at}(\text{true}, V) \ast P \}
\end{align*}
\]
The message passing idiom

\[ P \iff \exists \mathcal{V}. (\uparrow \mathcal{V}) \ast (P \text{ at } \mathcal{V}) \]

\[
\begin{align*}
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \} \\
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \text{ at } \mathcal{V} \ast \uparrow \mathcal{V} \} \\
a :=_{\text{at}} \text{true} \\
\{ a \mapsto_{\text{at}} (\text{true}, \mathcal{V}) \ast P \text{ at } \mathcal{V} \}
\end{align*}
\]

This proposition is objective: it can be put in an invariant.

\[
\{ a \mapsto_{\text{at}} (\text{true}, \mathcal{V}) \ast P \}
\]
The message passing idiom

\[ P \iff \exists V. (\uparrow V) \ast (P \text{ at } V) \]

\[
\begin{align*}
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \} \\
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \text{ at } V \ast \uparrow V \} \\
a :=_{\text{at}} \text{true} \\
\{ a \mapsto_{\text{at}} (\text{true}, V) \ast P \text{ at } V \} \\
\{ a \mapsto_{\text{at}} (\text{true}, V) \ast P \text{ at } V \ast \uparrow V \}
\end{align*}
\]

This proposition is objective: it can be put in an invariant.
The message passing idiom

\[ P \iff \exists V. (\uparrow V) \ast (P \text{ at } V) \]

\{a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P\}
\{a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \text{ at } V \ast \uparrow V\}
\text{a :=}_{\text{at}} \text{ true}
\{a \mapsto_{\text{at}} (\text{true}, V) \ast P \text{ at } V\}
\text{inv}
\{a \mapsto_{\text{at}} (\text{true}, V) \ast P \text{ at } V \ast \uparrow V\}
\text{C := } !_{\text{at}} a
\{a \mapsto_{\text{at}} (\text{true}, V) \ast P \text{ at } V \ast \uparrow V\}
\{a \mapsto_{\text{at}} (\text{true}, V) \ast P\}

This proposition is objective: it can be put in an invariant.
Example: spin lock

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

```ml
let rec acquire lk = if CAS lk false true then () else acquire lk
let release lk = lk :=\{at\} false
```

The invariant in the sequentially consistent model is:

\[
\text{lockInv } lk \ P \triangleq \begin{array}{l}
lk \mapsto_{at} \text{true} \lor ( \, \begin{array}{l}
lk \mapsto_{at} \text{false} \\
\ast \, P
\end{array} \, )
\end{array}
\]
A spin lock implements a lock using an atomic boolean variable.

```ocaml
define acquire lk =
    if CAS lk false true
    then ()
    else acquire lk

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

The invariant in the weak model is:

\[
\text{lockInv} \; \text{lk} \; P \triangleq
\begin{align*}
\text{lk} & \rightarrow_{\text{at}} \text{true} \\
\text{lk} & \rightarrow_{\text{at}} (\exists V. \text{lk} \rightarrow_{\text{at}} \text{(false, V)} \star P \; \text{at} \; V)
\end{align*}
\]
A ticket lock implements a lock using two atomic integer variables.

The invariant in the sequentially consistent model is:

\[ \exists t, n. \]

\[ \text{turn} \xrightarrow{\text{at}} t \]

\[ \text{next} \xrightarrow{\text{at}} n \]

\[ \gamma \text{ } (\text{ticket } \gamma \text{ } t \lor (\text{locked } \gamma \text{ } * \text{ } P)) \]

\[ \gamma \xleftarrow{\text{..}} \]

\[ \gamma \xleftarrow{\text{..}} \]
A ticket lock implements a lock using two atomic integer variables.

The invariant in the weak model is:

\[
\text{lockInv \ turn next } \gamma \ P \triangleq \\
\exists t, n, V. \\
\text{turn} \rightarrow_{\text{at}} (t, V) \\
* \text{next} \rightarrow_{\text{at}} n \\
* (\text{ticket } \gamma \ t \lor (\text{locked } \gamma \ * \ P \ \text{at } V)) \\
* \gamma \leftarrow (\bullet \ldots)
\]
Example: Dekker’s mutual exclusion

Dekker’s algorithm solves the mutual exclusion problem using three atomic variables.

The invariant and representation predicate in the SC model are:

\[ \text{DekkerInv} \quad \text{turn} \quad \text{flag}_0 \quad \text{flag}_1 \quad \gamma \quad P \triangleq \]

\[ \exists t, f_0, f_1, c_0, c_1. \]

\[ (\forall i \in \{0, 1\}. \ \text{flag}_i \rightarrow_{\text{at}} f_i \quad \star \quad \gamma_i \rightarrow \bullet c_i) \]

\[ \star \quad ((\neg c_0 \land \neg c_1) \rightarrow P) \]

\[ \star \quad \ldots \]

\[ \text{isDekker} \quad i \quad \gamma \triangleq \]

\[ \gamma_i \rightarrow \circ \text{false} \]
Example: Dekker’s mutual exclusion

Dekker’s algorithm solves the mutual exclusion problem using three atomic variables.

The invariant and representation predicate in the weak model are:

$$\text{DekkerInv turn flag}_0 \text{ flag}_1 \gamma \ P \triangleq$$

$$\exists t, f_0, f_1, c_0, c_1, V_0, V_1.$$

$$(\forall i \in \{0, 1\}. \text{flag}_i \mapsto \text{at} (f_i, V_i) * \gamma_i \hookrightarrow \bullet c_i * \gamma'_i \hookrightarrow \bullet V_i)$$

$$* ((\neg c_0 \land \neg c_1) \hookrightarrow P \ \text{at} (V_0 \sqcup V_1))$$

$$* \ldots$$

$$\text{isDekker } i \gamma \triangleq$$

$$\exists V. \gamma_i \hookrightarrow \circ \text{false} * \gamma'_i \hookrightarrow \circ V * \uparrow V$$
Model of the logic in Iris

Propositions are predicates on views:

\[
\begin{align*}
\text{vProp} & \triangleq \text{view} \rightarrow \text{iProp} \\
\uparrow V_0 & \triangleq \lambda V. \ V_0 \subseteq V \\
P \rightarrow^* Q & \triangleq \lambda V. \ \quad P \ V \rightarrow^* Q \ V
\end{align*}
\]
Propositions are \textbf{monotonic} predicates on views:

\[
\begin{align*}
    \nu \text{Prop} & \triangleq \text{view} \xrightarrow{\text{mon}} \nu \text{Prop} \\
    \uparrow \nu_0 & \triangleq \lambda \nu. \ nu_0 \sqsubseteq \nu \\
    P \rightarrow^* Q & \triangleq \lambda \nu_1. \ \forall \nu \sqsupseteq \nu_1. \ P \ \nu \rightarrow^* Q \ \nu
\end{align*}
\]
Propositions are **monotonic** predicates on views:

\[
vProp \triangleq \text{view} \xrightarrow{\text{mon}} iProp
\]

\[
\uparrow V_0 \triangleq \lambda V. \ V_0 \subseteq V
\]

\[
P \leftrightarrow Q \triangleq \lambda V_1. \ \forall V \supseteq V_1. \ P V \leftrightarrow Q V
\]

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:

\[
\text{WP} \ e \ \varphi \triangleq \lambda V_1. \ \forall V \supseteq V_1. \ \text{WP} \langle e, V \rangle (\lambda \langle v, V' \rangle. \ \varphi \ v \ V')
\]

where \( \varphi : \text{val} \rightarrow vProp \)
Plans for the future:

- Prove more elaborate shared data structures
  - e.g. bounded queues with a circular buffer
- Data races on non-atomics:
  - How to allow them?
  - What are they useful for?