Towards a separation logic for Multicore OCaml

Glen Mével, Jacques-Henri Jourdan, François Pottier
May 25, 2020
PPS seminar, 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.
Consider this concurrent program:

\[
\begin{align*}
  &x := 0 \\
  &y := 0 \\
  &x := 1 \quad y := 1 \\
  &A := y \quad B := x
\end{align*}
\]

Possible outcomes \((A, B)\): (0, 1), (1, 0), (1, 1).

Observed (M.OCaml on 2-core x86-64): 91%, 9%, 0.001%.
Consider this concurrent program:

\[
\begin{align*}
x & := 0 \\
y & := 0 \\
x & := 1 \\
y & := 1 \\
A & := y \\
B & := x
\end{align*}
\]

Possible outcomes \((A, B)\): \((0, 1)\), \((1, 0)\), \((1, 1)\).

Observed \((\text{M.OCaml on 2-core x86-64})\): 91\%, 9\%, 0.001\%.
Consider this concurrent program:

\[
\begin{align*}
&x := 0 \\
y := 0 \\
x := 1 \\
y := 1 \\
A := y \\
B := x
\end{align*}
\]

Possible outcomes (A, B): (0, 1), (1, 0), (1, 1).

Observed (M.OCaml on 2-core x86-64): 91%, 9%, 0.001%.
Consider this concurrent program:

\[
\begin{align*}
x &:= 0 \\
y &:= 0 \\
x &:= 1 \quad y &:= 1 \\
A &:= y \quad B &:= x
\end{align*}
\]

Possible outcomes \((A, B)\): \((0, 1)\), \((1, 0)\), \((1, 1)\).

Observed (\texttt{M.OCaml on 2-core x86-64}): 91\%, 9\%, 0.001\%. 

A weaker memory model

Consider this concurrent program:

\[
\begin{align*}
  x & := 0 \\
  y & := 0 \\
  x & := 1 \\
  y & := 1 \\
  A & := y \quad B := x
\end{align*}
\]

Possible outcomes \((A, B)\): \((0, 1), (1, 0), (1, 1), (0, 0)\).

Observed (M.OCaml on 2-core x86-64): 91%, 9%, 0.001%, 0.1%. 
A weaker memory model

Consider this concurrent program:

\[
\begin{align*}
    x &:= 0 \\
    y &:= 0 \\
    x &:= 1 \\
    y &:= 1 \\
    A &:= y \\
    B &:= x
\end{align*}
\]

Possible outcomes \((A, B)\): \((0, 1), (1, 0), (1, 1), (0, 0)\).
Observed \((\text{M.OCaml on 2-core x86-64})\): 91%, 9%, 0.001%, 0.1%.

The compiler may reorder a write after a read.
(The processor too.)
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.
x := x_1 \parallel y := y_1
A := !y \parallel B := !x
An operational model for Multicore OCaml: non-atomics

\[ \begin{align*}
  x & := x_1 & y & := y_1 \\
  A & := !y & B & := !x
\end{align*} \]

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

\[ \begin{align*}
  x & : x_0 \\
  y & : y_0
\end{align*} \]
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.
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

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

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

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

thread A

thread B

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

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

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

x := \(x_1\)

\[\text{atomic } a\]

thread A

\[\text{thread B}\]

\(C := !_{\text{at }} a\)

\(B := !x\)

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

\(\text{atomic } a\)

x : \(x_0\)

a : \text{false}
An operational model for Multicore OCaml: atomics

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

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 := a_@ true
```

```
REPEAT
  C := !a_@ a
UNTIL C = 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

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

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:**

$$\{ x \mapsto v \}$$

$$! x$$

$$\{ \lambda v'. v' = v * x \mapsto v \}$$
Impact of the weak memory model on our CSL

**Invariants** are the mechanism by which threads can share propositions in a Concurrent Separation Logic such as Iris:

\[
\frac{\{P \times I\} e \{Q \times I\} \text{ atomic}}{I \vdash \{P\} e \{Q\}}
\]

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.
Rules of atomic locations (simplified)

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

It is \textit{objective}.

\textbf{Atomic write:}

\[
\begin{align*}
\{a \rightarrow_{\text{at}} v\} \\
\hspace{1em} a :=_{\text{at}} v' \\
\{\lambda(). \ a \rightarrow_{\text{at}} v'\}
\end{align*}
\]

\textbf{Atomic read:}

\[
\begin{align*}
\{a \rightarrow_{\text{at}} v\} \\
\hspace{1em} !_{\text{at}} a \\
\{\lambda v'. \ v' = v \ast a \rightarrow_{\text{at}} v\}
\end{align*}
\]
The predicate $a \mapsto_{at} v$ means that we own the atomic location $a$, which stores the value $v$.

It is objective.

**Atomic write:**

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

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

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

**Atomic read:**

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

$$!_{at} a$$

$$\{ \lambda v'. v' = v * a \mapsto_{at} v \}$$

**views**

Views are ordered by inclusion.

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

4 views
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:**

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

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

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

**Atomic read:**

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

$$!_{at} a$$

$$\{ \lambda v'.\ v' = v \ *\ a \mapsto_{at} (v, V) \ *\ V' \}$$

Views are ordered by inclusion.

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

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

It is objective.

**Atomic write:**

\[
\begin{align*}
\{ a \mapsto_{\text{at}} (v, \mathcal{V}) \} & \quad \ast \uparrow \mathcal{V}' \\
\lambda(). \ a \mapsto_{\text{at}} (v', \mathcal{V}' \sqcup \mathcal{V}) \} & \quad \ast \uparrow \mathcal{V}
\end{align*}
\]

**Atomic read:**

\[
\begin{align*}
\{ a \mapsto_{\text{at}} (v, \mathcal{V}) \} & \quad \!_{\text{at}} \ a \\
\lambda v'. \ v' = v \ast a \mapsto_{\text{at}} (v, \mathcal{V}) \ast \uparrow \mathcal{V}'
\end{align*}
\]

Views are ordered by inclusion.

The predicate \( \uparrow \mathcal{V} \) means “the current thread’s view includes \( \mathcal{V} \)”.

views
Subjective propositions are monotonic w.r.t. the thread’s view.

One reason: the frame rule:

\[
\{ a \mapsto_{at} v \ast P \}
\]

\[
a :=_{at} v'
\]

\[
\{ \lambda(). \ a \mapsto_{at} v' \ast P \}
\]
Subjective propositions are monotonic w.r.t. the thread’s view.

One reason: the frame rule:

\[
\{ a \mapsto_{\text{at}} v \uparrow \; \ast \; P \; \text{this holds at the thread’s current view} \}
\]

\[
a :=_{\text{at}} v'
\]

\[
\{ \lambda(). \; a \mapsto_{\text{at}} v' \uparrow \; \ast \; P \; \text{this holds at the thread’s now extended view} \}
\]
The message passing idiom

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

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

\[
\begin{array}{c}
\{ \}
\end{array}
\]

\[
\begin{array}{c}
P \}
\end{array}
\]
The message passing idiom

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

\[ \{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \} \]

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

\[ C := !_{\text{at}} a \]

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

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

\{a \mapsto_{at} (\text{false}, \emptyset) \ast P\}
\{a \mapsto_{at} (\text{false}, \emptyset) \ast P \text{ at } V \ast \uparrow V\}

\text{a :=}_{at} \text{true}

\text{C := } !_{at} \text{ a}

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

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

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

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

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

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

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

\[
C := \!_{at} a
\]

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

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

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

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

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

\[ \text{C := !}_{\text{at}} \ a \]

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

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

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

\[
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \}
\{ a \mapsto_{\text{at}} (\text{false}, \emptyset) \ast P \text{ at } V \ast \uparrow V \}
\]

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

\[
\{ a \mapsto_{\text{at}} (\text{true}, V) \ast P \text{ at } V \}
\]

\[
\text{inv}
\]

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

\[
\text{lockInv } lk \ P \triangleq
\]

\[
lk \xrightarrow{at} \text{true} \lor (\ lk \xrightarrow{at} \text{false} \ast P )
\]
A spin lock implements a lock using an atomic boolean variable.

```plaintext
def acquire lk = 
  if CAS lk false true
  then ()
  else acquire lk

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

The invariant in the weak model is:

\[\text{lockInv}\ lk \ P \triangleq\]

\[lk \mapsto_{\text{at}} \text{true} \lor (\exists \nu. \ lk \mapsto_{\text{at}} (\text{false}, \nu) \ast P \text{ at } \nu)\]
Example: ticket lock

A ticket lock implements a lock using two atomic integer variables.

The invariant in the sequentially consistent model is:

\[
\text{lockInv turn next } \gamma \ P \triangleq \\
\exists t, n. \\
\quad \text{turn} \rightarrow_{at} t \\
\quad \ast \ \text{next} \rightarrow_{at} n \\
\quad \ast \ (\text{ticket } \gamma \ t \lor (\text{locked } \gamma \ast P)) \\
\quad \ast \ \gamma \leftarrow (\bullet\ldots)
\]
A ticket lock implements a lock using two atomic integer variables.

The invariant in the weak model is:

\[
\text{lockInv \ turn \ next} \ \gamma \ \mathcal{P} \triangleq \\
\exists t, n, \mathcal{V}.
\]

\[
\begin{align*}
\text{turn} & \rightarrow_{at} (t, \mathcal{V}) \\
\ast \ \text{next} & \rightarrow_{at} n \\
\ast \ (\text{ticket} \ \gamma \ t \ \vee (\text{locked} \ \gamma \ \ast \ \mathcal{P} \ \text{at} \ \mathcal{V})) \\
\ast \ \gamma & \leftarrow (\bullet \ldots)
\end{align*}
\]
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:

DekkerInv turn flag_0 flag_1 \gamma \ P \triangleq

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

(\forall i \in \{0, 1\}. \ flag_i \rightarrow_{at} f_i \quad * \ \gamma_i \rightarrow \bullet c_i)

* (\neg c_0 \land \neg c_1 \rightarrow P)

* ...

isDekker \ i \ \gamma \triangleq

\gamma_i \rightarrow \circ 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:

\[ \text{DekkerInv turn flag}_0 \text{ flag}_1 \gamma P \triangleq \]
\[ \exists t, f_0, f_1, c_0, c_1, V_0, V_1. \]
\[ (\forall i \in \{0, 1\}. \text{flag}_i \mapsto \text{at} (f_i, V_i) \uparrow \gamma_i \mapsto \bullet c_i \uparrow \gamma'_i \mapsto \bullet V_i) \]
\[ \uparrow (\neg c_0 \land \neg c_1) \uparrow P \text{ at } (V_0 \sqcup V_1) \]
\[ \uparrow \ldots \]

\[ \text{isDekker } i \gamma \triangleq \]
\[ \exists \nu. \gamma_i \mapsto \circ \text{false} \uparrow \gamma'_i \mapsto \circ \nu \uparrow \nu \]
Propositions are predicates on views:

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

\[ \uparrow \nu_0 \triangleq \lambda \nu. \nu_0 \sqsubseteq \nu \]

\[ P \star Q \triangleq \lambda \nu. P \nu \star Q \nu \]

\[ P \rightarrow Q \triangleq \lambda \nu. P \nu \rightarrow Q \nu \]
Model of the logic in Iris

Propositions are **monotonic** predicates on views:

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

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

\[ P \star Q \triangleq \lambda V. P V \star Q V \]

\[ P \rightarrow Q \triangleq \lambda V_1. \forall V \sqsubseteq V_1. P V \rightarrow Q V \]
Model of the logic in Iris

Propositions are **monotonic** predicates on views:

\[
\begin{align*}
\nuProp & \triangleq \text{view} \xrightarrow{\text{mon}} \nuProp \\
\uparrow \nu_0 & \triangleq \lambda \nu. \ \nu_0 \sqsubseteq \nu \\
P \ast Q & \triangleq \lambda \nu. \ P \nu \ast Q \ \nu \\
P \rightarrowth Q & \triangleq \lambda \nu_1. \ \forall \nu \sqsubseteq \nu_1. \ P \nu \rightarrowth Q \ \nu
\end{align*}
\]

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:

\[
\text{WP} \ e \ \phi \triangleq \lambda \nu_1. \ \forall \nu \sqsubseteq \nu_1. \ \text{WP} \langle e, \nu \rangle \ (\lambda \langle \nu, \nu' \rangle. \ \phi \ \nu \ \nu')
\]

where \( \phi : \text{val} \rightarrow \nuProp \)
Future work

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?