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.

Sequential consistency

Consider this concurrent program:

```
      x := 0
      y := 0
x := 1 || y := 1
A := y || B := x
```

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

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

Sequential consistency

Consider this concurrent program:

```
      x := 0
      y := 0
x := 1 ║ y := 1
A := y ║ B := x
```

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

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

Sequential consistency

Consider this concurrent program:

```
      x := 0
      y := 0
x := 1  ||  y := 1
A := y  ||  B := x
```

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

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

Sequential consistency

Consider this concurrent program:

```
      x := 0
      y := 0
x := 1 || y := 1
A := y || B := x
```

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

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

A weaker memory model

Consider this concurrent program:

```
      x := 0
      y := 0
x := 1 || y := 1
A := y || B := x
```

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:

```
        x := 0
        y := 0
x := 1  ||  y := 1
A := y  ||  B := x
```

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%.

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.

An operational model for Multicore OCaml: non-atomics

$$
\begin{array}{l}
\texttt{x := } x_1 \\
\texttt{A := !y}
\end{array}
\;\Bigg\|\;
\begin{array}{l}
\texttt{y := } y_1 \\
\texttt{B := !x}
\end{array}
$$

An operational model for Multicore OCaml: non-atomics

$$\begin{array}{l} \texttt{x := } x_1 \\ \texttt{A := !y} \end{array} \;\Bigg\|\; \begin{array}{l} \texttt{y := } y_1 \\ \texttt{B := !x} \end{array}$$

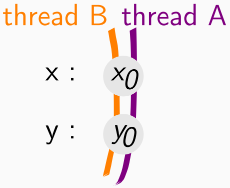
Each **non-atomic** location has a **history**, *i.e.* a map from timestamps to values (timestamps are per location).

x : x_0

y : y_0

An operational model for Multicore OCaml: non-atomics

`x := ` x_1	‖	`y := ` y_1
`A := !y`	‖	`B := !x`

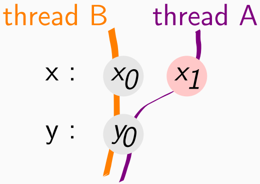


Each **non-atomic** location has a **history**, *i.e.* a map from timestamps to values (timestamps are per location).

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

An operational model for Multicore OCaml: non-atomics

`x := `x_1	`y := `y_1
`A := !y`	`B := !x`

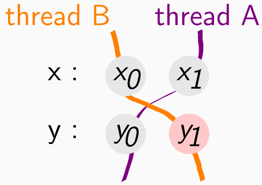


non-atomic write

- 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

```
x := x₁  ||  y := y₁
A := !y  ||  B := !x
```

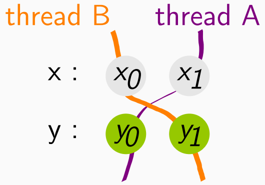


non-atomic write

- 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

```
x := x1  ||  y := y1
A := !y  ||  B := !x
```

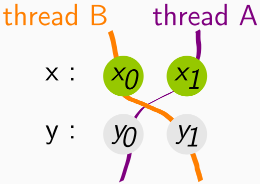


non-atomic read

- 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 := x1`	‖	`y := y1`
`A := !y`	‖	`B := !x`



non-atomic read

- Returns **any** value at least as recent as current thread's view.
- Current thread's view is unchanged.

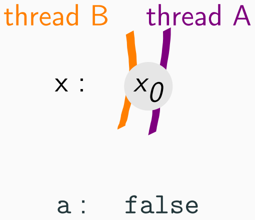
An operational model for Multicore OCaml: atomics

Non-atomic locations are useful for updating the state locally, but they don't provide synchronization.

Atomic locations allow the message-passing idiom.

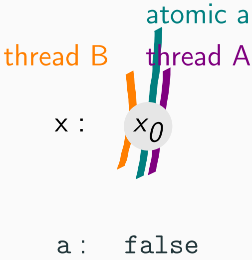
An operational model for Multicore OCaml: atomics

$$\begin{array}{l||l}
\texttt{x} \;\texttt{:=}\; x_1 & \texttt{REPEAT} \\
\texttt{a} \;\texttt{:=}_{\texttt{at}}\; \texttt{true} & \quad \texttt{C} \;\texttt{:=}\; \texttt{!}_{\texttt{at}}\; \texttt{a} \\
 & \texttt{UNTIL C = true} \\
 & \texttt{B := !x}
\end{array}$$



An operational model for Multicore OCaml: atomics

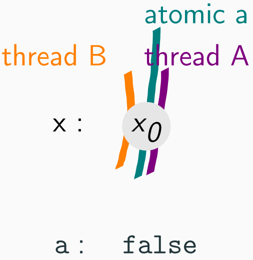
x := x_1	REPEAT
a :=$_{at}$ true	C := !$_{at}$ a
	UNTIL C = true
	B := !x



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

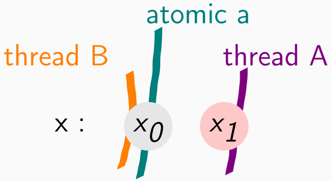
An operational model for Multicore OCaml: atomics

`x := ` x_1	`REPEAT`
`a :=`$_{\texttt{at}}$` true`	`   C := !`$_{\texttt{at}}$` a`
	`UNTIL C = true`
	`B := !x`



An operational model for Multicore OCaml: atomics

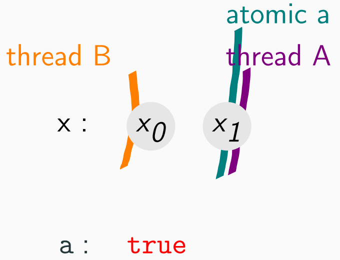
`x := ` x_1	`REPEAT`
`a` $:=_{at}$ `true`	`C :=` $!_{at}$ `a`
	`UNTIL C = true`
	`B := !x`



a : `false`

An operational model for Multicore OCaml: atomics

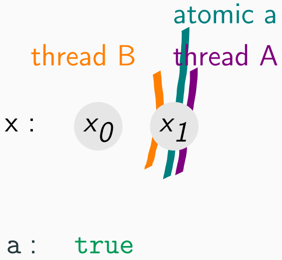
```
x := x₁          ||  REPEAT
a :=at true      ||    C := !at a
                 ||  UNTIL C = true
                 ||  B := !x
```



atomic write

Merges the writer's view into the atomic location's view.

x := x_1	REPEAT
a :=$_{at}$ true	C := !$_{at}$ a
	UNTIL C = true
	B := !x

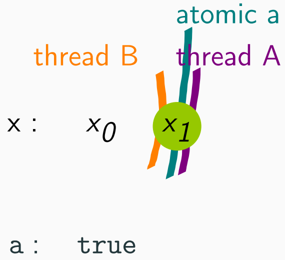


atomic read

Merges the atomic location's view into the reader's view.

An operational model for Multicore OCaml: atomics

x := x_1	‖	REPEAT
a $:=_{at}$ true	‖	C := $!_{at}$ a
	‖	UNTIL C = true
	‖	B := !x



Our program logic

Rules of non-atomic locations

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:

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

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 $\gamma \hookrightarrow \circ 5$
- atomic state, such as $a \mapsto_{\text{at}} (v, \mathcal{V})$

Only objective propositions can be put in an invariant.

Rules of atomic locations (simplified)

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

It is **objective**.

Atomic write:

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

Atomic read:

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

Rules of atomic locations (simplified)

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

It is **objective**.

Atomic write:

$$\{a \mapsto_{\mathrm{at}} v\}$$
$$a :=_{\mathrm{at}} v'$$
$$\{\lambda().\ a \mapsto_{\mathrm{at}} v'\}$$

Atomic read:

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

views

Views are ordered by inclusion.

The predicate $\uparrow \mathcal{V}$ means "the current thread's view includes $\mathcal{V}$".

Rules of atomic locations

The predicate $a \mapsto_{\mathsf{at}} (v, \mathcal{V})$ 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:

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

Atomic read:

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

views

Views are ordered by inclusion.

The predicate $\uparrow\mathcal{V}$ means "the current thread's view includes $\mathcal{V}$".

Rules of atomic locations

The predicate $a \mapsto_{\mathrm{at}} (v, \mathcal{V})$ 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:

$$\{a \mapsto_{\mathrm{at}} (v, \mathcal{V}) \ast \uparrow \mathcal{V}'\}$$

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

$$\{\lambda().\ a \mapsto_{\mathrm{at}} (v', \mathcal{V}' \sqcup \mathcal{V}) \ast \uparrow \mathcal{V}\}$$

Atomic read:

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

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

$$\{\lambda v'.\ v' = v \ast a \mapsto_{\mathrm{at}} (v, \mathcal{V}) \ast \uparrow \mathcal{V}\}$$

views

Views are ordered by inclusion.

The predicate $\uparrow \mathcal{V}$ means "the current thread's view includes $\mathcal{V}$".

Propositions are monotonic

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

One reason: the frame rule:

$$\{a \mapsto_{\mathsf{at}} v \ * \ P\}$$
$$a \coloneqq_{\mathsf{at}} v'$$
$$\{\lambda().\ a \mapsto_{\mathsf{at}} v' \ * \ P\}$$

Propositions are monotonic

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

One reason: the frame rule:

$$\{a \mapsto_{\mathsf{at}} v \ * \ P \quad \text{this holds at the thread's current view}\}$$

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

$$\{\lambda().\ a \mapsto_{\mathsf{at}} v' \ * \ P \quad \text{this holds at the thread's now extended view}\}$$

The message passing idiom

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

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

($\Rightarrow$) 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

$$P \iff \exists \mathcal{V}.\ (\uparrow \mathcal{V}) \ * \ (P \,\mathsf{at}\, \mathcal{V})$$

$\{ \qquad\qquad\qquad\qquad\qquad P\}$	
	$\{ \qquad\qquad\qquad\qquad\qquad P\}$

The message passing idiom

$$P \iff \exists \mathcal{V}.\ (\uparrow \mathcal{V}) * (P \,\mathsf{at}\, \mathcal{V})$$

$\{a \mapsto_{\mathsf{at}} (\mathtt{false}, \varnothing) * P\}$

```
a :=at true
```

```
C := !at a
```

$\{a \mapsto_{\mathsf{at}} (\mathtt{true}, \mathcal{V}) * P\}$

The message passing idiom

$$P \iff \exists \mathcal{V}.\ (\uparrow \mathcal{V}) * (P \operatorname{at} \mathcal{V})$$

$\{a \mapsto_{\mathsf{at}} (\mathtt{false}, \varnothing) * P\}$

$\{a \mapsto_{\mathsf{at}} (\mathtt{false}, \varnothing) * P \operatorname{at} \mathcal{V} * \uparrow \mathcal{V}\}$

```
a :=at true
```

||

```
C := !at a
```

$\{a \mapsto_{\mathsf{at}} (\mathtt{true}, \mathcal{V}) * P\}$

The message passing idiom

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

$\{a \mapsto_{\text{at}} (\texttt{false}, \varnothing) * P\}$	
$\{a \mapsto_{\text{at}} (\texttt{false}, \varnothing) * P \text{ at } \mathcal{V} * \uparrow \mathcal{V}\}$	
`a :=`$_{\texttt{at}}$ `true`	
$\{a \mapsto_{\text{at}} (\texttt{true}, \mathcal{V}) * P \text{ at } \mathcal{V}\}$	
	`C := !`$_{\texttt{at}}$ `a`
	$\{a \mapsto_{\text{at}} (\texttt{true}, \mathcal{V}) * P\}$

The message passing idiom

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

$\{a \mapsto_{\text{at}} (\mathtt{false}, \varnothing) \ast P\}$

$\{a \mapsto_{\text{at}} (\mathtt{false}, \varnothing) \ast P \text{ at } \mathcal{V} \ast \uparrow \mathcal{V}\}$

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

$\{a \mapsto_{\text{at}} (\mathtt{true}, \mathcal{V}) \ast P \text{ at } \mathcal{V}\}$

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

||

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

$\{a \mapsto_{\text{at}} (\mathtt{true}, \mathcal{V}) \ast P\}$

The message passing idiom

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

$\{a \mapsto_{\mathsf{at}} (\mathtt{false}, \varnothing) * P\}$

$\{a \mapsto_{\mathsf{at}} (\mathtt{false}, \varnothing) * P \text{ at } \mathcal{V} * \uparrow \mathcal{V}\}$

`a :=`$_{\mathtt{at}}$ `true`

$\{a \mapsto_{\mathsf{at}} (\mathtt{true}, \mathcal{V}) * P \text{ at } \mathcal{V}\}$

inv

$\{a \mapsto_{\mathsf{at}} (\mathtt{true}, \mathcal{V}) * P \text{ at } \mathcal{V}\}$

`C := !`$_{\mathtt{at}}$ `a`

$\{a \mapsto_{\mathsf{at}} (\mathtt{true}, \mathcal{V}) * P\}$

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

The message passing idiom

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

$\{a \mapsto_{\mathsf{at}} (\texttt{false}, \varnothing) * P\}$

$\{a \mapsto_{\mathsf{at}} (\texttt{false}, \varnothing) * P \text{ at } \mathcal{V} * \uparrow \mathcal{V}\}$

$\texttt{a} :=_{\texttt{at}} \texttt{true}$

$\{a \mapsto_{\mathsf{at}} (\texttt{true}, \mathcal{V}) * P \text{ at } \mathcal{V}\}$

inv

$\{a \mapsto_{\mathsf{at}} (\texttt{true}, \mathcal{V}) * P \text{ at } \mathcal{V}\}$

$\texttt{C} := \texttt{!}_{\texttt{at}}\ \texttt{a}$

$\{a \mapsto_{\mathsf{at}} (\texttt{true}, \mathcal{V}) * P \text{ at } \mathcal{V} * \uparrow \mathcal{V}\}$

$\{a \mapsto_{\mathsf{at}} (\texttt{true}, \mathcal{V}) * P\}$

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.

```
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:

$$\mathsf{lockInv}\ \mathtt{lk}\ P \triangleq$$
$$\mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true} \ \lor\ (\ \mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{false} \ * \ P)$$

Example: spin lock

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

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

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

The invariant in the weak model is:

$$\mathsf{lockInv}\ \mathtt{lk}\ P \triangleq$$

$$\mathtt{lk} \mapsto_{\mathsf{at}} \mathtt{true} \quad \vee \quad (\exists \mathcal{V}.\ \mathtt{lk} \mapsto_{\mathsf{at}} (\mathtt{false}, \mathcal{V}) * P\ \mathsf{at}\ \mathcal{V})$$

Example: ticket lock

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

The invariant in the sequentially consistent model is:

$$
\begin{aligned}
&\mathsf{lockInv}\ \mathtt{turn}\ \mathtt{next}\ \gamma\ P \triangleq \\
&\quad \exists t, n. \\
&\qquad\quad \mathtt{turn} \mapsto_{\mathsf{at}} t \\
&\qquad * \ \mathtt{next} \mapsto_{\mathsf{at}} n \\
&\qquad * \ (\mathsf{ticket}\ \gamma\ t \vee (\mathsf{locked}\ \gamma \ * \ P)) \\
&\qquad * \ \gamma \hookrightarrow (\bullet \ \ldots)
\end{aligned}
$$

Example: ticket lock

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

The invariant in the weak model is:

$$
\begin{aligned}
&\mathsf{lockInv}\ \mathtt{turn}\ \mathtt{next}\ \gamma\ P \triangleq \\
&\quad \exists t, n, \mathcal{V}. \\
&\qquad\quad \mathtt{turn} \mapsto_{\mathsf{at}} (t, \mathcal{V}) \\
&\qquad * \ \mathtt{next} \mapsto_{\mathsf{at}} n \\
&\qquad * \ (\mathsf{ticket}\ \gamma\ t \vee (\mathsf{locked}\ \gamma \ * \ P \ \mathsf{at}\ \mathcal{V})) \\
&\qquad * \ \gamma \hookrightarrow (\bullet \ \ldots)
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathsf{DekkerInv}\ \texttt{turn}\ \texttt{flag}_0\ \texttt{flag}_1\ \gamma\ P \triangleq \\
&\qquad \exists t, f_0, f_1, c_0, c_1. \\
&\qquad\qquad \big(\forall i \in \{0,1\}.\ \texttt{flag}_i \mapsto_{\mathsf{at}} f_i \qquad * \ \gamma_i \hookrightarrow \bullet\, c_i\big) \\
&\qquad\quad * \ ((\neg c_0 \land \neg c_1) \mathrel{-\!\!*} P) \\
&\qquad\quad * \ \ldots \\
&\mathsf{isDekker}\ i\ \gamma \triangleq \\
&\qquad\quad \gamma_i \hookrightarrow \circ\, \texttt{false}
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathrm{DekkerInv}\ \texttt{turn}\ \texttt{flag}_0\ \texttt{flag}_1\ \gamma\ P \triangleq \\
&\quad \exists t, f_0, f_1, c_0, c_1, \mathcal{V}_0, \mathcal{V}_1. \\
&\qquad \left(\forall i \in \{0,1\}.\ \texttt{flag}_i \mapsto_{\mathsf{at}} (f_i, \mathcal{V}_i) \ast \gamma_i \hookrightarrow \bullet\, c_i \ast \gamma'_i \hookrightarrow \bullet\, \mathcal{V}_i\right) \\
&\quad\ \ast\ ((\neg c_0 \wedge \neg c_1) \mathrel{-\!\!\ast} P\ \mathsf{at}\ (\mathcal{V}_0 \sqcup \mathcal{V}_1)) \\
&\quad\ \ast\ \ldots \\
&\mathrm{isDekker}\ i\ \gamma \triangleq \\
&\quad \exists \mathcal{V}. \gamma_i \hookrightarrow \circ\, \texttt{false} \ast \gamma'_i \hookrightarrow \circ\, \mathcal{V} \ast \uparrow \mathcal{V}
\end{aligned}
$$

Model of the logic in Iris

Propositions are predicates on views:

$$
\begin{aligned}
\mathrm{vProp} &\triangleq \mathrm{view} \longrightarrow \mathrm{iProp} \\
\uparrow \mathcal{V}_0 &\triangleq \lambda \mathcal{V}.\ \mathcal{V}_0 \sqsubseteq \mathcal{V} \\
P * Q &\triangleq \lambda \mathcal{V}.\ P\ \mathcal{V} * Q\ \mathcal{V} \\
P \mathrel{-\!\!*} Q &\triangleq \lambda \mathcal{V}\ . \qquad\qquad P\ \mathcal{V} \mathrel{-\!\!*} Q\ \mathcal{V}
\end{aligned}
$$

Model of the logic in Iris

Propositions are monotonic predicates on views:

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

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

$$P * Q \triangleq \lambda\mathcal{V}.\ P\ \mathcal{V} * Q\ \mathcal{V}$$

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

Model of the logic in Iris

Propositions are monotonic predicates on views:

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

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

$$P * Q \triangleq \lambda \mathcal{V}.\ P\ \mathcal{V} * Q\ \mathcal{V}$$

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

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:

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

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

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?