Cosmo: a concurrent separation logic for the weak memory model of Multicore OCaml

---

Glen Mével

PhD defense
December 14, 2022

LMF & Inria Paris

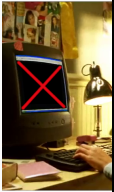
An introduction to weak-memory concurrency

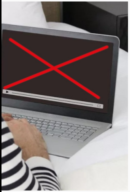
Movix

Family Cosmo **shares** a Movix account

Movix

Family Cos





Movix' policy: simultaneous accesses $\Longrightarrow$ account canceled

Movix

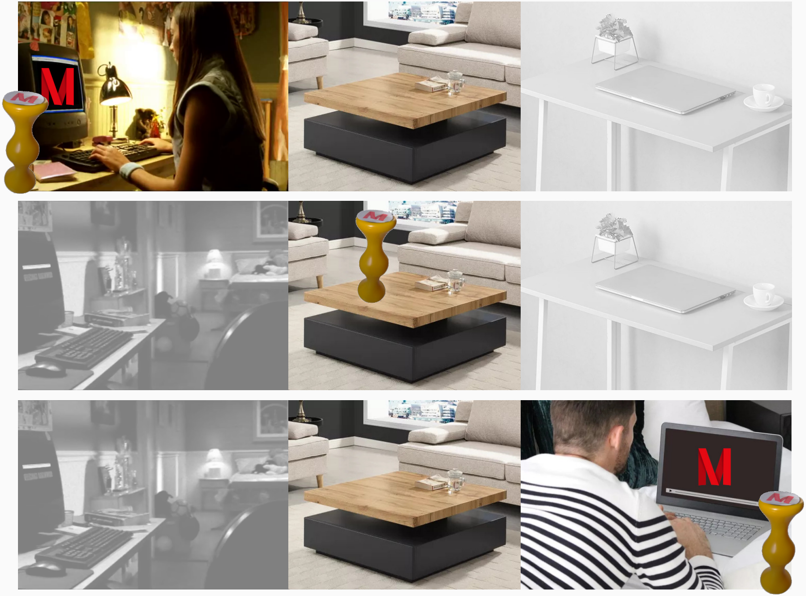
Family Cosmo **shares** a Movix account

Movix' policy: simultaneous accesses $\Longrightarrow$ account canceled

Solution: Family Cosmo has established a protocol:

- a **totem** in the living room, that anyone can borrow
- to watch Movix, one must have borrowed the totem

Movix: mutual exclusion



Movix password

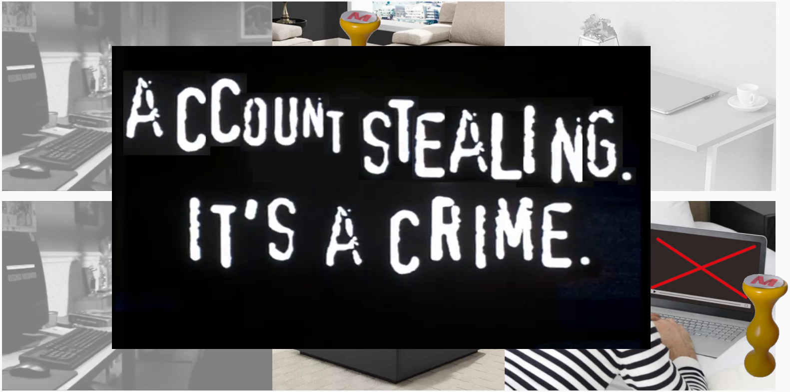
Family Cosmo's members can **change** the password

One day...

Movix password

Family Cosmo's members can **change** the password

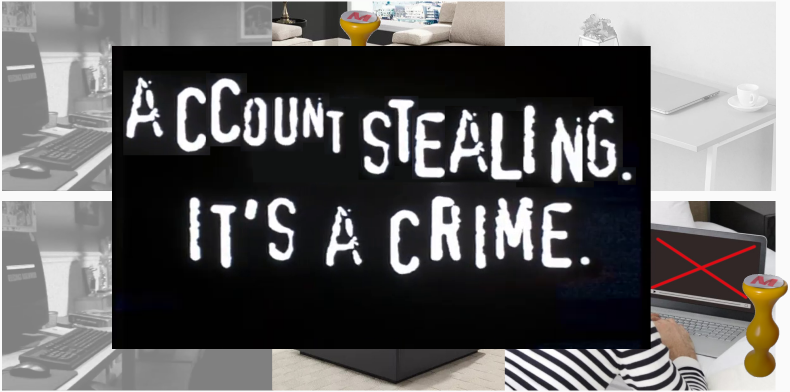
One day...



Movix password

Family Cosmo's members can **change** the password

One day...

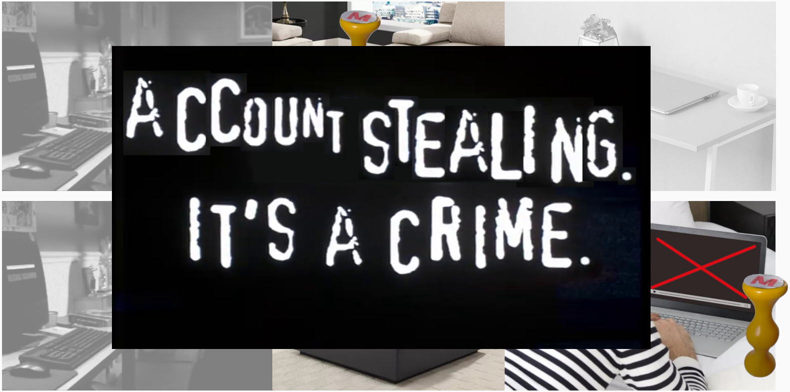


Alice had changed the password the day before, Bob didn't know

Movix password

Family Cosmo's members can **change** the password

One day...



Alice had changed the password the day before, Bob didn't know

Movix' security policy: wrong password $\Longrightarrow$ IP blocked

Movix: takeaway

Mutual exclusion is not enough

Alice and Bob have diverging **views** of their common password

Alice **must transmit** her (more up-to-date) knowledge to Bob

Movix: takeaway

Mutual exclusion is not enough

Alice and Bob have diverging **views** of their common password

Alice **must transmit** her (more up-to-date) knowledge to Bob

Solution: write the password on the totem

Weak memory models

Weak memory models:

multicore architecture, shared memory

each thread has its own **view** of the state of the shared memory

- example: C11
- example: Java
- example: Multicore OCaml (“OCaml 5”)

  [Dolan et al, PLDI 2018, *Bounding data races in space and time*]

Weak memory models

Weak memory models:

multicore architecture, shared memory

each thread has its own **view** of the state of the shared memory

- example: C11
- example: Java
- example: Multicore OCaml ("OCaml 5")

  [Dolan et al, PLDI 2018, *Bounding data races in space and time*]

Program verification

Programming is **error-prone**... Concurrent programming even more!

Some bugs might have catastrophic consequences

- losing a paid Movix account
- killing patients: Therac-25 radiotherapy machine
- ...

Program verification

Programming is **error-prone**... Concurrent programming even more!

Some bugs might have catastrophic consequences

- losing a paid Movix account
- killing patients: Therac-25 radiotherapy machine
- ...

How to improve confidence in software?

Program verification

Programming is **error-prone**... Concurrent programming even more!

Some bugs might have catastrophic consequences

- losing a paid Movix account
- killing patients: Therac-25 radiotherapy machine
- ...

How to improve confidence in software?

Specify it:
state the expected behavior of a program in mathematical terms

Program verification

Programming is **error-prone**... Concurrent programming even more!

Some bugs might have catastrophic consequences

- losing a paid Movix account
- killing patients: Therac-25 radiotherapy machine
- ...

How to improve confidence in software?

Specify it:
state the expected behavior of a program in mathematical terms

Verify it:
prove that the actual behavior matches the expected one

This work

My aim:

- verifying
- fine-grained concurrent programs
- in the setting of Multicore OCaml

My contributions:

- Cosmo, a concurrent separation logic with views [ICFP 2020]
- case studies: locks [ICFP 2020], concurrent queue [ICFP 2021]

Verifying SC concurrent programs with Concurrent Separation Logic

Specifying a program [Hoare, 1969]

Hoare triple: $\{Pre\}\, e\, \{Post\}$

- e: program code
- Pre: precondition (logical assertion about the computer state)
- $Post$: postcondition (ditto)

"If we run e from a state that satisfies Pre, and if it terminates, then it ends in a state that satisfies $Post$."

Specifying with Separation Logic [Reynolds 1999; O'Hearn 2007; Iris 2015; …]

Hoare triple: $\{Pre\}\ e\ \{Post\}$

- e: program code
- Pre: precondition (logical assertion about the computer state)
- $Post$: postcondition (ditto)

"If we run e from a state that satisfies Pre, and if it terminates, then it ends in a state that satisfies $Post$."

Pre and $Post$ are stated in **Separation Logic**:

- an assertion represents the ownership of a resource
- the separating conjunction $P * Q$ asserts ownership of two **distinct** resources P and Q
- in general, $P \not\Rightarrow P * P$

Resources and locks

Portions of memory are ownable **resources**

- example: $a \rightsquigarrow$ `"azerty"`
  "Movix account a, whose current password is `"azerty"`"

We can guard such resources by using a **lock** (like the totem)

Two operations for a lock lk guarding a resource R:

- `acquire` lk
  grants R: we become its unique owner
- `release` lk
  reclaims R: we give it back and stop owning it

Resources and locks

Portions of memory are ownable **resources**

- example: $a \rightsquigarrow$ `"azerty"`
  "Movix account a, whose current password is `"azerty"`"

We can guard such resources by using a **lock** (like the totem)

Two operations for a lock lk guarding a resource R:

- `acquire` lk
  grants R: we become its unique owner
- `release` lk
  reclaims R: we give it back and stop owning it

Formal specification?

Specification of a lock

if *lk* is a lock that guards *R*, then

acquiring *lk* assumes nothing and grants *R*

releasing *lk* reclaims *R* and grants nothing

Specification of a lock

$\boxed{\text{isLock } lk\ R} \vdash$ if *lk* is a lock that guards *R*, then

$$\begin{cases} \{\text{True}\}\,\texttt{acquire lk}\,\{R\} \\ \{R\}\,\texttt{release lk}\,\{\text{True}\} \end{cases}$$

acquiring *lk* assumes nothing and grants *R*

releasing *lk* reclaims *R* and grants nothing

Specification of a lock

$$\boxed{\text{isLock}\ \mathit{lk}\ R} \vdash \begin{cases} \{\text{True}\}\,\texttt{acquire lk}\,\{R\} \\ \{R\}\,\texttt{release lk}\,\{\text{True}\} \end{cases}$$

if *lk* is a lock that guards R, then

acquiring *lk* assumes nothing and grants R

releasing *lk* reclaims R and grants nothing

isLock lk R is an assertion describing the internal layout of *lk*
$\Longrightarrow$ asserts unique ownership of *lk*

$\boxed{\text{isLock}\ \mathit{lk}\ R}$ is an Iris invariant containing isLock lk R
$\Longrightarrow$ shares *lk* among all threads

The spin lock

A spin lock implements a lock using a Boolean reference:

```
let release lk =
  lk := false
```

```
let try_acquire lk =
  CAS lk false true
```

The spin lock

A spin lock implements a lock using a Boolean reference:

```
let release lk =
  lk := false
```

```
let try_acquire lk =
  CAS lk false true
```

Described by this assertion:

$$\text{isLock}\ lk\ R \quad \triangleq \quad \exists b.\ lk \rightsquigarrow b \ *\ (b = \texttt{false} \Rightarrow R)$$

The spin lock

A spin lock implements a lock using a Boolean reference:

```
let release lk =
  lk := false
```

```
let try_acquire lk =
  CAS lk false true
```

Described by this assertion:

$$\text{isLock}\ lk\ R \quad \triangleq \quad \exists b.\ lk \rightsquigarrow b \ast (b = \texttt{false} \Rightarrow R)$$

Specification of operations used by the spin lock:

$$\begin{Bmatrix} x \rightsquigarrow v \end{Bmatrix} \quad x := v' \quad \begin{Bmatrix} x \rightsquigarrow v' \end{Bmatrix}$$

$$\begin{Bmatrix} x \rightsquigarrow v \end{Bmatrix} \quad \texttt{CAS}\ x\ v_1\ v_2 \quad \begin{Bmatrix} \text{if } ret \\ \quad \text{then } x \rightsquigarrow v_2 \ast v = v_1 \\ \quad \text{else } x \rightsquigarrow v \end{Bmatrix}$$

Verifying the spin lock in Concurrent Separation Logic [Iris, 2015]

$$\text{isLock}\ lk\ R \quad \triangleq \quad \exists b.\ lk \rightsquigarrow b\ *\ (b = \texttt{false} \Rightarrow R)$$

$\boxed{\text{isLock}\ lk\ R} \vdash$

`// release:`

$\{R\}$

$lk \coloneqq \texttt{false}$

$\{\text{True}\}$

$\boxed{\text{isLock}\ lk\ R} \vdash$

`// try_acquire:`

$\{\text{True}\}$

$\texttt{CAS}\ lk\ \texttt{false}\ \texttt{true}$

$\{(ret = \texttt{true} \Rightarrow R)\}$

Verifying the spin lock in Concurrent Separation Logic [Iris, 2015]

$$\text{isLock}\ lk\ R \quad \triangleq \quad \exists b.\ \ lk \rightsquigarrow b \ * \ (b = \texttt{false} \Rightarrow R)$$

```
// release:
```

$\{\text{isLock}\ lk\ R \ * \ R\}$

$lk := \texttt{false}$

$\{\text{isLock}\ lk\ R\}$

```
// try_acquire:
```

$\{\text{isLock}\ lk\ R\}$

$\texttt{CAS}\ lk\ \texttt{false}\ \texttt{true}$

$\{\text{isLock}\ lk\ R \ * \ (ret = \texttt{true} \Rightarrow R)\}$

Verifying the spin lock in Concurrent Separation Logic [Iris, 2015]

$$\text{isLock}\ lk\ R \quad \triangleq \quad \exists b.\ lk \rightsquigarrow b \ * \ (b = \texttt{false} \Rightarrow R)$$

// release:

$\{\text{isLock}\ lk\ R \ * \ R\}$

$\{lk \rightsquigarrow \_ \quad\quad * \ R\}$

$lk := \texttt{false}$

$\{lk \rightsquigarrow \texttt{false} \ * \ R\}$

$\{\text{isLock}\ lk\ R\}$

// try_acquire:

$\{\text{isLock}\ lk\ R\}$

$\{lk \rightsquigarrow b \ * \ (b = \texttt{false} \Rightarrow R)\}$

$\texttt{CAS}\ lk\ \texttt{false}\ \texttt{true}$

$$\left\{\begin{array}{l} \text{if } ret \\ \quad \text{then } lk \rightsquigarrow \texttt{true} \ * \ R \\ \quad \text{else } \ lk \rightsquigarrow b \ * \ (b = \texttt{false} \Rightarrow R) \end{array}\right\}$$

$$\left\{\begin{array}{l} \text{if } ret \\ \quad \text{then isLock}\ lk\ R \ * \ R \\ \quad \text{else } \ \text{isLock}\ lk\ R \end{array}\right\}$$

$\{\text{isLock}\ lk\ R \ * \ (ret = \texttt{true} \Rightarrow R)\}$

Verifying Multicore OCaml programs with Cosmo

Using a lock

Example of using a lock *lk* to guard accesses to *pw*:

initially, `pw = 0`

initially, `lk = false`

```
acquire lk   ||
pw := 1      ||  acquire lk
release lk   ||  B := !pw
             ||  release lk
```

Using a lock

Example of using a lock *lk* to guard accesses to *pw*:

initially, `pw = 0`

initially, `lk = false`

```
acquire lk  ||
pw := 1     ||  acquire lk
release lk  ||  B := !pw
            ||  release lk
```

Passing a message

Passing a write to *pw* from the left thread to the right thread:

```
       initially, pw = 0
     initially, lk = true

pw := 1      ||  acquire lk
release lk   ||  B := !pw
```

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`
initially, `lk = true`

```
pw := 1      || A := !lk
lk := false  || B := !pw
```

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`
initially, `lk = true`

```
pw := 1     ||  A := !lk
lk := false ||  B := !pw
```

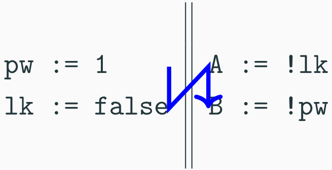
Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`



```
pw := 1       ||  A := !lk
lk := false   ||  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

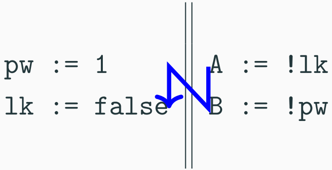
Traditional model of concurrency: interleavings

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`



```
pw := 1       || A := !lk
lk := false   || B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

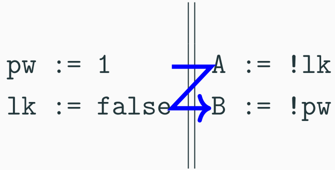
Traditional model of concurrency: interleavings

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`



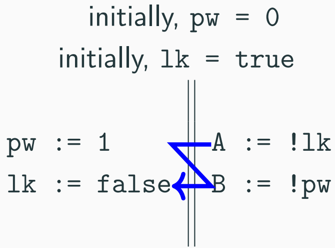
```
pw := 1        ||  A := !lk
lk := false    ||  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

Traditional model of concurrency: interleavings

Passing a message

Passing a write to *pw* from the left thread to the right thread:



initially, `pw = 0`

initially, `lk = true`

```
pw := 1      ||  A := !lk
lk := false  ||  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

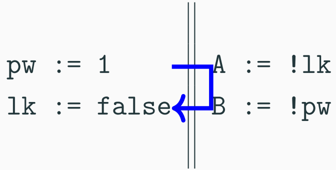
Traditional model of concurrency: interleavings

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`



```
pw := 1         ||  A := !lk
lk := false     ||  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

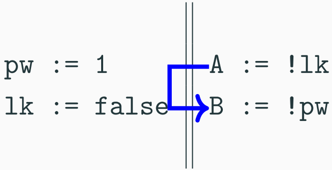
Traditional model of concurrency: interleavings

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`



```
pw := 1      ||  A := !lk
lk := false  ||  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

Traditional model of concurrency: interleavings

Passing a message

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`

```
pw := 1       ║  A := !lk
lk := false   ║  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1)

Traditional model of concurrency: interleavings

Passing a message in Multicore OCaml

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`
initially, `lk = true`

```
pw := 1      ||  A := !lk
lk := false  ||  B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1), (`false`, 0)

Traditional model of concurrency violated

Passing a message in Multicore OCaml

Passing a write to *pw* from the left thread to the right thread:

initially, `pw = 0`

initially, `lk = true`

```
pw := 1      ║ A := !lk
lk := false  ║ B := !pw
```

Possible (A, B): (`true`, 1), (`true`, 0), (`false`, 1), (`false`, 0)

Traditional model of concurrency violated

- hardware optimizations (e.g. buffering writes)
- compiler optimizations (e.g. reordering independent writes)

The essence of weak memory: subjectivity

Weak memory: each thread has its own **view** of memory [Dolan et al]

In Cosmo: [ICFP 2020]

Some assertions are **subjective**: they depend on the thread's view

- example: $pw \rightsquigarrow$ `"azerty"`

The essence of weak memory: subjectivity

Weak memory: each thread has its own **view** of memory [Dolan et al]

In Cosmo: [ICFP 2020]

Some assertions are **subjective**: they depend on the thread's view

- example: $pw \rightsquigarrow$ `"azerty"`

Invariants are **objective**:

because they are available to all threads

they cannot share subjective assertions

The essence of weak memory: subjectivity

Weak memory: each thread has its own **view** of memory [Dolan et al]

In Cosmo: [ICFP 2020]

Some assertions are **subjective**: they depend on the thread's view

- example: $pw \rightsquigarrow$ `"azerty"`

Invariants are **objective**:
because they are available to all threads
they cannot share subjective assertions

The lock's resource R might be subjective,
so cannot be put in an invariant as we did

Synchronization through atomic references

The left thread must transmit its view to the right thread
$\Longrightarrow$ need for a **synchronization** mechanism

Synchronization through atomic references

The left thread must transmit its view to the right thread
$\Longrightarrow$ need for a **synchronization** mechanism

In Multicore OCaml: **atomic references** [Dolan et al]

```
            initially, pw := 0
        initially, lk :={at} true
pw := 1               ‖ A := !{at} lk
lk :={at} false       ‖ B := !pw
```

Synchronization through atomic references

The left thread must transmit its view to the right thread
$\Longrightarrow$ need for a **synchronization** mechanism

In Multicore OCaml: **atomic references** [Dolan et al]

```
              initially, pw := 0
          initially, lk :={at} true
pw := 1              ||  A := !{at} lk
lk :={at} false      ||  B := !pw
```

Specification of atomic references?

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

In Cosmo: [ICFP 2020]

- $x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U})$

  "x stores the value v **and a view** (at least) $\mathcal{U}$"
- this assertion is objective
- reasoning rules (sample):

$$\underline{\left\{ x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}' \right\}} \qquad \underline{\left\{ x \rightsquigarrow_{\mathrm{at}} (v_1, \mathcal{U}) \right\}}$$

$$x :=_{\mathrm{at}} v' \qquad \mathtt{CAS}\ x\ v_1\ v_2$$

$$\overline{\left\{ x \rightsquigarrow_{\mathrm{at}} (v', \mathcal{U}') \right\}} \qquad \overline{\left\{ \mathit{ret} = \mathtt{true} * x \rightsquigarrow_{\mathrm{at}} (v_2, \mathcal{U}) * \uparrow \mathcal{U} \right\}}$$

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

views

views are mathematical objects;

they enjoy a lattice structure

In Cosmo: [ICFP 2020]

- $x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U})$

  "x stores the value v **and a view** (at least) $\mathcal{U}$"
- this assertion is objective
- reasoning rules (sample):

$$\left\{ x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U}) * \uparrow \mathcal{U}' \right\} \quad x :=_{\mathrm{at}} v' \quad \left\{ x \rightsquigarrow_{\mathrm{at}} (v', \mathcal{U}') \right\}$$

$$\left\{ x \rightsquigarrow_{\mathrm{at}} (v_1, \mathcal{U}) \right\} \quad \mathtt{CAS}\ x\ v_1\ v_2 \quad \left\{ \mathit{ret} = \mathtt{true} * x \rightsquigarrow_{\mathrm{at}} (v_2, \mathcal{U}) * \uparrow \mathcal{U} \right\}$$

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

In Cosmo: [ICFP 2020]

- $x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U})$
  "x stores the value v **and a view** (at least) $\mathcal{U}$"
- this assertion is objective
- reasoning rules (sample):

$$\{x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U}) * \uparrow\mathcal{U}'\} \quad x :=_{\mathrm{at}} v' \quad \{x \rightsquigarrow_{\mathrm{at}} (v', \mathcal{U}')\}$$

$$\{x \rightsquigarrow_{\mathrm{at}} (v_1, \mathcal{U})\} \quad \mathtt{CAS}\ x\ v_1\ v_2 \quad \{\mathit{ret} = \mathtt{true} * x \rightsquigarrow_{\mathrm{at}} (v_2, \mathcal{U}) * \uparrow\mathcal{U}\}$$

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

In Cosmo: [ICFP 2020]

- $x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U})$
  "x stores the value v **and a view** (at least) $\mathcal{U}$"
- this assertion is objective
- reasoning rules (sample):

current view

$\uparrow\mathcal{U}$: "the current view contains $\mathcal{U}$"

$$\left\{ x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U}) * \uparrow\mathcal{U}' \right\} \quad x :=_{\mathrm{at}} v' \quad \left\{ x \rightsquigarrow_{\mathrm{at}} (v', \mathcal{U}') \right\}$$

$$\left\{ x \rightsquigarrow_{\mathrm{at}} (v_1, \mathcal{U}) \right\} \quad \texttt{CAS}\ x\ v_1\ v_2 \quad \left\{ ret = \texttt{true} * x \rightsquigarrow_{\mathrm{at}} (v_2, \mathcal{U}) * \uparrow\mathcal{U} \right\}$$

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

In Cosmo: [ICFP 2020]

- $x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U})$

  "x stores the value v **and a view** (at least) $\mathcal{U}$"
- this assertion is objective
- reasoning rules (sample):

current view

$\uparrow\mathcal{U}$: "the current view contains $\mathcal{U}$"

$$\left\{x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U}) * \uparrow\mathcal{U}'\right\} \quad x :=_{\mathrm{at}} v' \quad \left\{x \rightsquigarrow_{\mathrm{at}} (v', \mathcal{U}')\right\}$$

release

$$\left\{x \rightsquigarrow_{\mathrm{at}} (v_1, \mathcal{U})\right\} \quad \texttt{CAS}\ x\ v_1\ v_2 \quad \left\{ret = \texttt{true} * x \rightsquigarrow_{\mathrm{at}} (v_2, \mathcal{U}) * \uparrow\mathcal{U}\right\}$$

Atomic references

Semantics of an atomic reference: [Dolan et al]

1. stores a value on which all threads agree at any point in time
2. achieves **release/acquire** synchronization

In Cosmo: [ICFP 2020]

- $x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U})$

  "x stores the value v **and a view** (at least) $\mathcal{U}$"
- this assertion is objective
- reasoning rules (sample):

current view

$\uparrow\mathcal{U}$: "the current view contains $\mathcal{U}$"

$$\begin{array}{l} \{x \rightsquigarrow_{\mathrm{at}} (v, \mathcal{U}) * \uparrow\mathcal{U}'\} \\ \quad x :=_{\mathrm{at}} v' \\ \{x \rightsquigarrow_{\mathrm{at}} (v', \mathcal{U}')\} \end{array} \qquad \begin{array}{l} \{x \rightsquigarrow_{\mathrm{at}} (v_1, \mathcal{U})\} \\ \quad \mathtt{CAS}\; x\; v_1\; v_2 \\ \{ret = \mathtt{true} * x \rightsquigarrow_{\mathrm{at}} (v_2, \mathcal{U}) * \uparrow\mathcal{U}\} \end{array}$$

acquire

The spin lock in Multicore OCaml

A spin lock implements a lock using an **atomic** Boolean reference:

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

```
let try_acquire lk =
  CAS lk false true
```

Described by this assertion:

$$\text{isLock}\ lk\ R \quad \triangleq \quad \exists b \quad .\ lk \rightsquigarrow_{\mathrm{at}}\ b \qquad * \ (b = \texttt{false} \Rightarrow R \quad)$$

The spin lock in Multicore OCaml

A spin lock implements a lock using an **atomic** Boolean reference:

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

```
let try_acquire lk =
  CAS lk false true
```

Described by this assertion:

$$\mathsf{isLock}\ lk\ R \quad \triangleq \quad \exists b \quad .\ lk \rightsquigarrow_{\mathrm{at}}\ b \qquad * \ (b = \mathtt{false} \Rightarrow R \quad)$$

R may be subjective!

The spin lock in Multicore OCaml

A spin lock implements a lock using an **atomic** Boolean reference:

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

```
let try_acquire lk =
  CAS lk false true
```

Described by this assertion:

$$\mathsf{isLock}\ lk\ R \quad \triangleq \quad \exists b, \mathcal{U}.\ lk \rightsquigarrow_{\mathrm{at}} (b, \mathcal{U}) \ast (b = \mathtt{false} \Rightarrow R @ \mathcal{U})$$

R may be subjective!

objective part of an assertion

$R\ @\ \mathcal{U}$: “R where the current view is fixed to $\mathcal{U}$”

Verifying the spin lock in Multicore OCaml

$$\mathsf{isLock}\ lk\ R \quad \triangleq \quad \exists b, \mathcal{U}.\ lk \rightsquigarrow_{\mathrm{at}} (b, \mathcal{U}) \ * \ (b = \mathtt{false} \Rightarrow R @ \mathcal{U})$$

`// release:`

$$\{ \quad \mathsf{isLock}\ lk\ R * R \}$$

$$\{ \quad lk \rightsquigarrow_{\mathrm{at}} \_ \quad * R \}$$

$$\left\{ \exists \mathcal{U}.\ lk \rightsquigarrow_{\mathrm{at}} \_ \quad * \overbrace{\uparrow \mathcal{U} * R @ \mathcal{U}} \right\}$$

$$lk :=_{\mathrm{at}} \mathtt{false}$$

$$\{ \quad lk \rightsquigarrow_{\mathrm{at}} (\mathtt{false}, \mathcal{U}) * R @ \mathcal{U} \}$$

$$\{ \quad \mathsf{isLock}\ lk\ R \}$$

splitting rule

$$P \iff \exists \mathcal{U}.\ \underbrace{P @ \mathcal{U}}_{\text{objective}} \ * \ \underbrace{\uparrow \mathcal{U}}_{\text{subjective}}$$

`// try_acquire:`

$$\{ \mathsf{isLock}\ lk\ R \}$$

$$\{ lk \rightsquigarrow_{\mathrm{at}} (b, \mathcal{U}) * (b = \mathtt{false} \Rightarrow R @ \mathcal{U}) \}$$

$$\mathtt{CAS}\ lk\ \mathtt{false}\ \mathtt{true}$$

$$\left\{ \begin{array}{l} \text{if } ret \\ \quad \text{then } lk \rightsquigarrow_{\mathrm{at}} (\mathtt{true}, \mathcal{U}) * \underbrace{\uparrow \mathcal{U} * R @ \mathcal{U}} \\ \quad \text{else } \ \ldots \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{if } ret \\ \quad \text{then } \mathsf{isLock}\ lk\ R \ * \ R \\ \quad \text{else } \ \ldots \end{array} \right\}$$

$$\{ \mathsf{isLock}\ lk\ R \ * \ (ret = \mathtt{true} \Rightarrow R) \}$$

Methodology

A 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.

Case study: a multiple-producer multiple-consumer queue

A MPMC queue

case study: [ICFP 2021]

specifying and verifying a fine-grained concurrent queue
in the weak memory model of Multicore OCaml

challenges:

1. shared ownership
   tool: logical atomicity (not in this talk) [Iris, 2015; ICFP 2021]

A MPMC queue

case study: [ICFP 2021]
specifying and verifying a fine-grained concurrent queue
in the weak memory model of Multicore OCaml

challenges:

1. shared ownership
   tool: logical atomicity (not in this talk) [Iris, 2015; ICFP 2021]
2. need to specify thread synchronization
   tool: views [ICFP 2020]

A MPMC queue

case study: [ICFP 2021]
specifying and verifying a fine-grained concurrent queue
in the weak memory model of Multicore OCaml

challenges:

1. shared ownership
   tool: logical atomicity (not in this talk) [Iris, 2015; ICFP 2021]
2. need to specify thread synchronization
   tool: views [ICFP 2020]
   - **fine-grained** specification, more permissive than lock-based

non-trivial implementation taking profit from the relaxed spec

A specification for concurrent queues in SC

$$\left\langle \underline{n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]} \right\rangle$$

$$\quad \texttt{enqueue}\ q\ v$$

$$\left\langle \overline{\lambda().\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}, v]} \right\rangle$$

$$\left\langle \underline{n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]} \right\rangle$$

$$\quad \texttt{dequeue}\ q$$

$$\left\langle \overline{\lambda v.\ \mathsf{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\ \ast\ 1 \leq n\ \ast\ v = v_0} \right\rangle$$

A specification for concurrent queues in SC

$$\left\langle \frac{n, v_0, \dots, v_{n-1}.\, \mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}]}{\begin{array}{l}\quad \texttt{enqueue}\ q\ v\end{array}} \right.$$

$$\left\langle \lambda().\, \mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}, v] \right\rangle$$

$$\left\langle \frac{n, v_0, \dots, v_{n-1}.\, \mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}]}{\begin{array}{l}\quad \texttt{dequeue}\ q\end{array}} \right.$$

$$\left\langle \lambda v.\, \mathsf{IsQueue}\ q\ [v_1, \dots, v_{n-1}] \ * \ 1 \leq n \ * \ v = v_0 \right\rangle$$

- IsQueue $q\ [v_0, \dots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant

A specification for concurrent queues in SC

$$\left\langle \underline{n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]} \right\rangle$$

$$\texttt{enqueue}\ q\ v$$

$$\left\langle \overline{\lambda().\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}, v]} \right\rangle$$

$$\left\langle \underline{n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]} \right\rangle$$

$$\texttt{dequeue}\ q$$

$$\left\langle \overline{\lambda v.\ \mathsf{IsQueue}\ q\ [v_1, \ldots, v_{n-1}] \ * \ 1 \leq n \ * \ v = v_0} \right\rangle$$

- IsQueue $q\ [v_0, \ldots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant

A specification for concurrent queues in SC

$$\begin{array}{l}
\left\langle \underline{n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]} \right\rangle \\
\quad \texttt{enqueue}\ q\ v \\
\left\langle \overline{\lambda ().\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}, v] \qquad\qquad} \right\rangle
\end{array}$$

$$\begin{array}{l}
\left\langle \underline{n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ q\ [v_0, \ldots, v_{n-1}] \qquad\qquad\qquad} \right\rangle \\
\quad \texttt{dequeue}\ q \\
\left\langle \overline{\lambda v.\ \mathsf{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\ *\ 1 \leq n\ *\ v = v_0} \right\rangle
\end{array}$$

- IsQueue $q\ [v_0, \ldots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant

A specification for concurrent queues in weak memory

$$\left\langle \begin{array}{l} n,\ v_0, \dots, v_{n-1}. \\ \qquad \mathsf{IsQueue}\ q\ [\ v_0, \dots, v_{n-1}\ ] \end{array} \right\rangle$$

$$\texttt{enqueue}\ q\ v$$

$$\left\langle \lambda ().\ \mathsf{IsQueue}\ q\ [\ v_0, \dots, v_{n-1}, v\ ] \right\rangle$$

$$\left\langle \begin{array}{l} n,\ v_0, \dots, v_{n-1}. \\ \qquad \mathsf{IsQueue}\ q\ [\ v_0, v_1, \dots, v_{n-1}\ ] \end{array} \right\rangle$$

$$\texttt{dequeue}\ q$$

$$\left\langle \lambda v.\ \mathsf{IsQueue}\ q\ [\ v_1, \dots, v_{n-1}\ ] \ast 1 \leq n \ast v = v_0 \right\rangle$$

- IsQueue $q\ [v_0, \dots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant

A specification for concurrent queues in weak memory

$$\left\langle \begin{array}{l} n,\ v_0,\ \dots,\ v_{n-1}. \\ \quad \mathsf{IsQueue}\ q\ [\,v_0,\ \dots,\ v_{n-1}\,] \end{array} \right\rangle$$

$$\texttt{enqueue}\ q\ v$$

$$\left\langle \lambda().\ \mathsf{IsQueue}\ q\ [\,v_0,\ \dots,\ v_{n-1},\ v\,] \right\rangle$$

$$\left\langle \begin{array}{l} n,\ v_0,\ \dots,\ v_{n-1}. \\ \quad \mathsf{IsQueue}\ q\ [\,v_0,\ v_1,\ \dots,\ v_{n-1}\,] \end{array} \right\rangle$$

$$\texttt{dequeue}\ q$$

$$\left\langle \lambda v.\ \mathsf{IsQueue}\ q\ [\,v_1,\ \dots,\ v_{n-1}\,] \ast 1 \le n \ast v = v_0 \right\rangle$$

- $\mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant
  $\Longrightarrow$ must be objective

A specification for concurrent queues in weak memory

$$\left\langle \begin{array}{l} n,\ v_0,\ldots,v_{n-1}.\\ \qquad \mathsf{IsQueue}\ q\ [\,v_0,\ldots,v_{n-1}\,] \end{array}\right\rangle$$

$$\quad \texttt{enqueue}\ q\ v$$

$$\left\langle \lambda().\, \mathsf{IsQueue}\ q\ [\,v_0,\ldots,v_{n-1},\ v\,] \right\rangle$$

$$\left\langle \begin{array}{l} n,\ v_0,\ldots,v_{n-1}.\\ \qquad \mathsf{IsQueue}\ q\ [\,v_0,\ v_1,\ldots,v_{n-1}\,] \end{array}\right\rangle$$

$$\quad \texttt{dequeue}\ q$$

$$\left\langle \lambda v.\, \mathsf{IsQueue}\ q\ [\,v_1,\ldots,v_{n-1}\,] \ast 1 \le n \ast v = v_0 \right\rangle$$

- IsQueue $q\ [v_0, \ldots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant
  $\Longrightarrow$ must be objective
- The queue transfers resources $\Longrightarrow$ must transfer views

A specification for concurrent queues in weak memory

$$\left\langle \underline{\begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathsf{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \qquad\qquad * \ \uparrow \mathcal{V} \end{array}} \right\rangle$$

$$\quad \texttt{enqueue}\ q\ v$$

$$\left\langle \overline{\lambda ().\ \mathsf{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}), (v, \mathcal{V})] \qquad\qquad} \right\rangle$$

$$\left\langle \underline{\begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathsf{IsQueue}\ q\ [(v_0, \mathcal{V}_0), (v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \qquad\qquad\qquad \end{array}} \right\rangle$$

$$\quad \texttt{dequeue}\ q$$

$$\left\langle \overline{\lambda v.\ \mathsf{IsQueue}\ q\ [(v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \ * \ \uparrow \mathcal{V}_0 \ * \ 1 \le n \ * \ v = v_0} \right\rangle$$

- IsQueue $q\ [v_0, \ldots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant
  $\Longrightarrow$ must be objective
- The queue transfers resources $\Longrightarrow$ must transfer views

A specification for concurrent queues in weak memory

$$\left\langle \begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathsf{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \qquad * \ {\uparrow}\mathcal{V} \end{array} \right\rangle$$

`enqueue` q v

$$\left\langle \lambda().\ \mathsf{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}), (v, \mathcal{V})] \right\rangle$$

$$\left\langle \begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathsf{IsQueue}\ q\ [(v_0, \mathcal{V}_0), (v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \end{array} \right\rangle$$

`dequeue` q

$$\left\langle \lambda v.\ \mathsf{IsQueue}\ q\ [(v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \ * \ {\uparrow}\mathcal{V}_0 \ * \ 1 \leq n \ * \ v = v_0 \right\rangle$$

- IsQueue q $[v_0, \ldots, v_{n-1}]$ is exclusive
  $\Longrightarrow$ must be shared through an invariant
  $\Longrightarrow$ must be objective
- The queue transfers resources $\Longrightarrow$ must transfer views

A specification for concurrent queues in weak memory

$$\left\langle \begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathrm{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \qquad\qquad * \ \uparrow\mathcal{V} \end{array} \right\rangle$$

$$\quad \texttt{enqueue}\ q\ v$$

$$\left\langle \lambda().\ \mathrm{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}), (v, \mathcal{V})] \right\rangle$$

$$\left\langle \begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathrm{IsQueue}\ q\ [(v_0, \mathcal{V}_0), (v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \end{array} \right\rangle$$

$$\quad \texttt{dequeue}\ q$$

$$\left\langle \lambda v.\ \mathrm{IsQueue}\ q\ [(v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \ * \ \uparrow\mathcal{V}_0 \ * \ 1 \leq n \ * \ v = v_0 \right\rangle$$

- IsQueue q $[v_0, \ldots, v_{n-1}]$ is exclusive
  $\implies$ must be shared through an invariant
  $\implies$ must be objective
- The queue transfers resources $\implies$ must transfer views

Comparison with refinement in weak memory

Refinement is another approach to specifying the queue:
“this queue can replace a sequential queue guarded by a lock”

Shortcoming of the refinement spec:
the lock induces synchronization between **all** operations

Comparison with refinement in weak memory

Refinement is another approach to specifying the queue:
"this queue can replace a sequential queue guarded by a lock"

Shortcoming of the refinement spec:
the lock induces synchronization between **all** operations

Our spec is **more permissive**:
no guaranteed synchronization from dequeuer to enqueuer

Allows for more relaxed implementations... like the one we verified

Conclusion

Contributions

- BaseCosmo: A low-level program logic for the weak memory model of Multicore OCaml [ICFP 2020]
  - closely reflects the operational semantics
- Cosmo: A higher-level logic, based on a notion of views [ICFP 2020]
  - easier to use, cannot reason about racy programs
- Verification of locks and mutual exclusion algorithms [ICFP 2020]
- Specification and verification of a non-trivial lock-free queue [ICFP 2021]
  - demonstrates the expressivity of Cosmo
  - methodology: add views wherever synchronization is relevant
- Mechanized in Coq (Iris)

Takeaways

The logic of views enables concise and natural reasoning about how threads synchronize

Enables fine-grained specifications

Don't hide views: make them apparent in specifications!

Prove specifications with the splitting rule

Fits naturally into a Hoare/Concurrent Separation Logic framework

Model of the logic in Iris

Assertions are predicates on views:

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

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

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

Iris builds a WP calculus for exprWithView in iProp.

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

$$\begin{aligned}
&\mathsf{WP}\ e\ \varphi \triangleq \lambda\mathcal{U}\ . \\
&\qquad\qquad \mathsf{valid}\,\mathcal{U} \mathrel{-\!\!*} \mathsf{WP}\ \langle e,\mathcal{U}\rangle\ \ \left(\lambda \left\langle v,\mathcal{U}'\right\rangle.\ \mathsf{valid}\,\mathcal{U}' * \varphi\ v\ \mathcal{U}'\right)
\end{aligned}$$

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

Model of the logic in Iris

Assertions are monotonic predicates on views:

$$\begin{aligned}
\mathsf{vProp} &\triangleq \mathsf{view} \xrightarrow{\mathsf{mon}} \mathsf{iProp} \\
\uparrow\mathcal{U}_0 &\triangleq \lambda\mathcal{U}.\, \mathcal{U}_0 \sqsubseteq \mathcal{U} \\
P * Q &\triangleq \lambda\mathcal{U}.\, P\,\mathcal{U} * Q\,\mathcal{U} \\
P \mathrel{-\!\!*} Q &\triangleq \lambda\mathcal{U}_1.\, \forall\mathcal{U} \sqsupseteq \mathcal{U}_1.\, P\,\mathcal{U} \mathrel{-\!\!*} Q\,\mathcal{U}
\end{aligned}$$

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

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

Iris builds a WP calculus for exprWithView in iProp.

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

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

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

Model of the logic in Iris

Assertions are monotonic predicates on views:

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

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

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

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

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

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

Iris builds a WP calculus for exprWithView in iProp.

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

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

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

Assertions are monotonic

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

One reason is the frame rule:

$$\begin{array}{l}\left\{\underline{x \rightsquigarrow_{\mathrm{na}} v * P \qquad\quad}\right\} \\ \quad x :=_{\mathrm{na}} v' \\ \left\{\overline{\lambda().\ x \rightsquigarrow_{\mathrm{na}} v' * P}\right\}\end{array}$$

Assertions are monotonic

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

One reason is the frame rule:

$$\underline{\left\{x \rightsquigarrow_{\mathrm{na}} v * P \text{ — holds at the thread's current view}\right\}}$$

$$x \mathrel{:=}_{\mathrm{na}} v'$$

$$\overline{\left\{\lambda().\ x \rightsquigarrow_{\mathrm{na}} v' * P \text{ — holds at the thread's now extended view}\right\}}$$

Key idea: decomposing subjective assertions

Decompose subjective assertions:

$$P \iff \exists \mathcal{U}.\ \underbrace{P \mathbin{@} \mathcal{U}}_{\text{objective}} * \underbrace{\uparrow \mathcal{U}}_{\text{subjective}}$$

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

Share parts via distinct mechanisms:

- $P \mathbin{@} \mathcal{U}$: via an objective **invariant**
- $\uparrow \mathcal{U}$: via **synchronization** offered by the memory model

Key idea: decomposing subjective assertions

Decompose subjective assertions:

$$P \iff \exists \mathcal{U}.\ \underbrace{P \mathbin{@} \mathcal{U}}_{\text{objective}} * \underbrace{\uparrow\mathcal{U}}_{\text{subjective}}$$

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

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

Share parts via distinct mechanisms:

- $P \mathbin{@} \mathcal{U}$: via an objective **invariant**
- $\uparrow\mathcal{U}$: via **synchronization** offered by the memory model