A Separation Logic Framework for Asynchronous Programming

João Mota - *jd.mota@campus.fct.unl.pt*

Advisers: António Ravara and Marco Giunti

NOVA LINCS and NOVA School of Science & Technology

Iris Workshop 2026

Producer + Consumer in JavaScript

```
const queue = new Queue();
```

```
async function producer() {
  const data = await otherTask1();
  queue.add(data);
  queue.unuse();
}
```

```
async function consumer() {
  while (queue.inUse()) {
    const data = await queue.take();
    await otherTask2(data);
  }
}
```

```
queue.use();
await Promise.all([ producer(), consumer() ]);
```

Producer + Consumer in Java

```
Queue queue = new Queue(); // Locks inside
```

```
Thread producer = new Thread(() -> {
  Data data = otherTask1();
  queue.add(data);
  queue.unuse();
});
```

```
Thread consumer = new Thread(() -> {
  while (queue.inUse()) {
    Data data = queue.take();
    otherTask2(data);
  }
});
```

```
queue.use();
producer.start(); consumer.start();
```

Problem

- Languages have *different concurrency models*
  - Single-threaded asynchrony (e.g., JavaScript)
  - Multi-threading (e.g., Java)
- How to guarantee
  memory safety, **thread-safety**, and **avoid memory leaks**?
  - No distinction between *thread-local* and *thread-shared* data forces access to shared data to be *synchronized*
  - Forced to choose between a fully *affine* or fully *linear* logic

We need a unified approach

Separation Logic for Async. Prog.

Framework heavily inspired on **Iris** [1]. Features:

- Affine and linear resources/invariants
- Thread-local and thread-shared invariants
- Invariants are precisely tracked
- Weakest-precondition tailored for a concurrent language with threads and asynchronous tasks in each thread

1. https://iris-project.org/ ↩

Affine and linear resources

- **affine** resources can be "dropped" (and be garbage collected)
- **linear** resources must be necessarily used

  even without `free()`, these are useful to avoid memory leaks

E.g., queue as a linear resource until it is empty
E.g., a socket as a linear resource until calling `close()`

Thread-local/shared invariants

- **thread-shared** invariants are opened around atomic expressions
- **thread-local** may remain open for several steps until reaching a yield-point (such as the *await* keyword)

E.g., queue as a thread-shared resource in Java
E.g., queue as a thread-local resource in JS

Base Logic

- Non-step-indexed
- We extend algebras with an *affinity predicate* $\mathcal{A}$
- Connectives are obtained from Iris by replacing $\preccurlyeq$ with $\sqsubseteq$ [1]
  - $\sqsubseteq \triangleq \exists c.\ b \equiv a \cdot b \wedge \mathcal{A}(c)$
  - Precise tracking of linear resources requires *cancelability*

1. Krebbers et al. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. ICFP 2018 ↩

Basic Update Modality

- Issue: $P \vdash \Rightarrow\!\!\!\mid \epsilon$
- Extend the modality with a *controlled update relation*
  - $\Rightarrow\!\!\!\mid\{u\} P \triangleq$
    $\lambda a.\ \forall c. \mathcal{V}(a \cdot c) \Rightarrow \exists b. \mathcal{V}(b \cdot c) \land P(b) \land (a, b) \in u$
- For a logic parameterized with the *authoritative algebra*, we can define the following relation to preserve linear resources:

$$\frac{\forall f.\ b_1 \cdot f \equiv a_1 \Rightarrow b_2 \cdot f \sqsubseteq a_2}{\bullet a_1, \circ b_1 \dashrightarrow_{linear} \bullet a_2, \circ b_2} \quad \bullet\bullet\bullet$$

Linear Update Example

$$\frac{\forall f.\ b_1 \cdot f \equiv a_1 \Rightarrow b_2 \cdot f \sqsubseteq a_2}{\bullet a_1, \circ b_1 \dashrightarrow_{linear} \bullet a_2, \circ b_2}$$

$$\frac{[\iota_2 \leftarrow v_2] \cdot [\iota_1 \leftarrow v_1] \equiv [\iota_1 \leftarrow v_1, \iota_2 \leftarrow v_2] \qquad \varepsilon \cdot [\iota_1 \leftarrow v_1] \sqsubseteq [\iota_1 \leftarrow v_1, \iota_2 \leftarrow v_2]}{\bullet [\iota_1 \leftarrow v_1, \iota_2 \leftarrow v_2], \circ [\iota_2 \leftarrow v_2] \dashrightarrow_{linear} \bullet [\iota_1 \leftarrow v_1, \iota_2 \leftarrow v_2], \circ \varepsilon}$$

Forgetting about reference ι_2

Linear Update Example

$$\frac{\forall f.\ b_1 \cdot f \equiv a_1 \Rightarrow b_2 \cdot f \sqsubseteq a_2}{\bullet a_1, \circ b_1 \dashrightarrow_{linear} \bullet a_2, \circ b_2}$$

$$\frac{[\iota_2 \leftarrow v_2] \cdot [\iota_1 \leftarrow v_1] \equiv [\iota_1 \leftarrow v_1, \iota_2 \leftarrow v_2] \qquad \varepsilon \cdot [\iota_1 \leftarrow v_1] \sqsubseteq [\iota_1 \leftarrow v_1]}{\bullet [\iota_1 \leftarrow v_1, \iota_2 \leftarrow v_2], \circ [\iota_2 \leftarrow v_2] \dashrightarrow_{linear} \bullet [\iota_1 \leftarrow v_1], \circ \varepsilon}$$

Deallocation of reference ι_2

Invariant Masks

To support both thread-shared and thread-local invariants,
we extend *masks* to be sets of pairs,
with *thread-sharedness* and *invariant identifier*

$$ts \in \mathcal{T} \triangleq \mathsf{ThShared} \mid \mathsf{ThLocal}(t : \mathbb{N})$$
$$\text{where } t \text{ is a thread identifier}$$
$$\mathcal{E} \in \mathsf{Mask} \triangleq \mathcal{P}(\mathcal{T} \times \mathbb{N})$$

Fancy Update Modalities

The "normal" fancy update modality is the same, but with a globally parameterized controlled update relation

$$\mathcal{E}_1 \Rrightarrow_{\mathcal{E}_2} P \triangleq \mathsf{wsat} * \boxed{\mathcal{E}_1}^{En} \mathrel{-\!\!*} \dot{\Rrightarrow}\{u\}(\mathsf{wsat} * \boxed{\mathcal{E}_2}^{En} * P)$$

The novel "predicated" fancy update modality allows to pick the mask that must be reestablished after the update (if it fulfills $\Phi_{\mathcal{E}}$), enabling thread-local invariants to remain opened for longer

$$\mathcal{E}_1 \Rrightarrow_{\Phi_{\mathcal{E}}} \Phi \triangleq \mathcal{E}_1 \Rrightarrow_{\emptyset} (\exists \mathcal{E}_2. \ulcorner \Phi_{\mathcal{E}}(\mathcal{E}_2) \urcorner \wedge \boxed{\mathcal{E}_2}^{En} * \Phi(\mathcal{E}_2))$$

World Satisfaction

- Two maps:
  1. Stores the propositions (persistently)
  2. Tracks the allocated invariants
- Allocated invariants in (2) are composed by *external id*, *internal id* to (1), affinity, thread-sharedness
- Propositions in (1) belong to a logic parameterized with user-given resource algebras, excluding the algebra that encodes (1)
- The indirection avoids circularity in the model and allows storing invariant tokens inside of invariants without step-indexing

World Satisfaction

- Invariants are precisely tracked with some given model of permissions (e.g., fractional or counting)
- As in Iron[1], we have *opening* and *deallocation* tokens
  - Deallocation tokens for linear invariants cannot be put inside of invariants
  - Both kinds can be used to open invariants

1. Ales Bizjak et al. Iron: managing obligations in higher-order concurrent separation logic. POPL 2019 ↩

World Satisfaction

$$\mathsf{holds}(P, \mathit{aff}) \triangleq \begin{cases} \mathsf{Emp} \land \Uparrow P & \text{if } \mathit{aff} = \mathsf{True} \\ \mathsf{NoLinearDealloc} \land \Uparrow P & \text{if } \mathit{aff} = \mathsf{False} \end{cases}$$

$$\mathsf{holds\_id}(id, \mathit{aff}) \triangleq \exists P \in \mathsf{fProp}_0.\ \boxed{\circ [id \leftarrow \mathsf{ag}(P)]}^{Store} * \mathsf{holds}(P, \mathit{aff})$$

$$\mathsf{assert}(i, inv^?) \triangleq \begin{cases} \mathsf{Emp} & \text{if } inv^? = \bot \\ (\mathsf{holds\_id}(id, \mathit{aff}) * \boxed{\{i\}}^{Dis}) \lor \boxed{\{(ts, i)\}}^{En} & \text{if } inv^? = (id, \mathit{aff}, ts) \end{cases}$$

$$\mathsf{wsat} \triangleq \left(\exists S.\ \boxed{\bullet S}^{Store}\right) * \left(\exists R.\ \boxed{\bullet R}^{Reg} * \mathop{\Large *}_{i \in \mathsf{dom}(R)} \mathsf{assert}(i, R(i))\right)$$

"holds" enforces that affine invariants only hold affine resources, and that linear invariants do not hold deallocation tokens for linear invariants

Concurrent Language

- User provides small-step operational semantics for expressions
- User provides definitions of *allowed* and *forbidden* stuckness
  - More customization of when code may be stuck

$$(\mathsf{step}\ -\ -\ -\ -\ -\ -) \subseteq ExecEnv \times Code \times State \times Code \times State \times Effect$$
$$(\mathsf{allowed\_stuck}\ -\ -\ -) \subseteq ExecEnv \times Code \times State$$
$$(\mathsf{forbidden\_stuck}\ -\ -\ -) \subseteq ExecEnv \times Code \times State$$

Concurrent Language

- We define a concurrent language with threads, and asynchronous tasks in each thread
  - Asynchronous tasks follow "run-to-completion" scheduling (similar to JS)
  - In each thread, context switch occurs only at yield-points

$$\frac{tp(t) = \langle a_{active}, ap \rangle \qquad ap(a) = tk \qquad (a = a_{active} \vee \mathsf{can\_yield}(ap(a_{active}))) \qquad \mathsf{task\_step}\ t\ p\ tp\ h\ tk\ h'\ tk'\ eff}{\mathsf{thread\_step}\ t\ p\ tp\ h\ \mathsf{update\_pool}(\mathsf{handle\_eff}(t, tp, eff), t, a, tk')\ h'}$$

Weakest Precondition

$$\mathsf{wp}_{p;t;\mathcal{E}_1;\mathcal{E}_2}\ c\ \{v, \mathcal{E}.\ P\}\ \{c', \mathcal{E}'.\ Q\}$$

Program, Thread Identifier

Mask to be restored at yield-points, Current enabled mask

Code expression

Post-condition, Post-condition for coinduction[1]

1. Lennard Gäher et al. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. POPL 2022 ↩

Weakest Precondition

$$
\begin{aligned}
&\mathsf{wp}_p \triangleq \mu\ wp\_rec.\ \lambda t, \mathcal{E}_1, \mathcal{E}_2, c, \Phi, \Phi_{\mathsf{coind}}.\ \forall tp, h.\\
&\quad \mathsf{state\_inter}(p, tp, h) \mathrel{-\!\!*} {}_{\mathcal{E}_2}\!\Rightarrow_{\emptyset} (\\
&\quad\quad {}_{\emptyset}\!\Rightarrow_{\lambda\_.\mathsf{True}} (\lambda \mathcal{E}_3.\ \mathsf{state\_inter}(p, tp, t) * \Phi_{\mathsf{coind}}(c, \mathcal{E}_3)) \lor \qquad (\textsc{Coinductive Hyp. App.})\\
&\quad\quad {}_{\emptyset}\!\Rightarrow_{\circlearrowleft(\mathcal{E}_1, c)} (\lambda \mathcal{E}_3.\ \mathsf{state\_inter}(p, tp, t) * \exists v.\ \ulcorner \mathsf{to\_val}(c) = v \urcorner \land \Phi(v, \mathcal{E}_3)) \lor \qquad (\textsc{Value Case})\\
&\quad\quad (\ulcorner \mathsf{red}((p, tp, t), h, c) \lor \mathsf{allowed\_stuck}\ (p, tp, t)\ h\ c \urcorner \land \qquad (\textsc{Safety Condition})\\
&\quad\quad\quad \forall h', c', \mathit{eff}.\ \ulcorner \mathsf{step}\ (p, tp, t)\ h\ c\ h'\ c'\ \mathit{eff} \urcorner \Rightarrow {}_{\emptyset}\!\Rightarrow_{\circlearrowleft(\mathcal{E}_1, c')} (\lambda \mathcal{E}_3. \qquad (\textsc{Step Case})\\
&\quad\quad\quad\quad \mathsf{state\_inter}(p, \mathsf{handle\_eff}(t, tp, \mathit{eff}), h') * wp\_rec(t, \mathcal{E}_1, \mathcal{E}_3, c', \Phi, \Phi_{\mathsf{coind}}) *\\
&\quad\quad\quad\quad \mathsf{handle\_new\_task}(\ldots)))
\end{aligned}
$$

$$
\circlearrowleft(\mathcal{E}_1, c) \triangleq \lambda \mathcal{E}_2. \begin{cases} \mathcal{E}_1 = \mathcal{E}_2 & \text{if } \mathsf{is\_yield\_point}(c) \\ \{(\mathsf{ThShared}, i) \mid (\mathsf{ThShared}, i) \in \mathcal{E}_1\} \subseteq \mathcal{E}_2 & \text{otherwise} \end{cases}
$$

If c is a yield-point, we must enable all invariants (in $\mathcal{E}_1$).
Otherwise, we just need to enable all thread-shared invariants.

Object Calculus

$v \in$ **Values**	::=	**null**	| **true**	| **false**	| bv	| ι	| a_t	async task id
$e \in$ **Exprs**	::=							
	|	v	value	|	x	variable		
	|	**let** $x = e; e'$	let binding	|	$e.f$	field access		
	|	$e.f = e'$	field assign	|	**new** C	new object		
	|	**free** e	free object	|	$e.m(e')$	method call		
	|	$uop\ e$	unary operation	|	$e\ bop\ e'$	binary operation		
	|	**if** $e_1\{e_2\}$ **else** $\{e_3\}$	if then else	|	**async** e	create async task		
	|	**fork** e	create thread	|	**await** e	await task		
	|	**lock** e	lock object	|	**unlock** e	unlock object		

Both examples verified

```
let queue = new Queue.init(); queue.use();
```

```
let consumer =
  async { new Consumer.init(queue) };
let producer =
  async { new Producer.init(queue) };
```

or

```
let consumer =
  fork { new Consumer.init(queue) };
let producer =
  fork { new Producer.init(queue) };
```

```
await producer; await consumer; queue.isEmpty()
```

$$\{\mathsf{Emp}\}\ main\ \{v, \mathcal{E}.\ulcorner v = \mathbf{true} \land \mathcal{E} = \mathsf{imask}(0)\urcorner \land \mathsf{Emp}\}_{p;0;\mathsf{imask}(0);\mathsf{imask}(0)}$$

Proof strategy

1. Use counting permissions instead of fractional ones
2. Allocate invariant that stores `useCount` opening tokens for itself
3. Give the full deallocation token to the consumer
4. Give one opening token to the producer

$$\mathsf{QueueInv}(i, \iota) \triangleq \exists\ uc\ l.\ \mathsf{Queue}(1, \iota, uc, l) * \mathsf{OPermToken}(i, uc)$$

$$\{\mathsf{QueueDPerm}(i, t, 0, \iota)\}\ \mathbf{new}\ \texttt{Consumer.init}(\iota)\ \{\_, \mathcal{E}.\ulcorner \mathcal{E} = \mathsf{imask}(t)\urcorner \wedge \mathsf{Queue}(1, \iota, 0, [])\}_{p;t;\mathsf{imask}(t);\mathsf{imask}(t)}$$

$$\{\mathsf{QueueOPerm}(i, t, -1, \iota)\}\ \mathbf{new}\ \texttt{Producer.init}(\iota)\ \{\_, \mathcal{E}.\ulcorner \mathcal{E} = \mathsf{imask}(t)\urcorner \wedge \mathsf{Emp}\}_{p;t;\mathsf{imask}(t);\mathsf{imask}(t)}$$

Results

```
Theorem main_safe tp' h' :
  tpool_steps prog {[0 := (0, {[ 0 := Start main ]})]} ⌀ tp' h' ->
  ~ forbidden_stuck_tp prog tp' h'.
Proof. ... Qed.

Theorem main_sound tp' h' v :
  tpool_steps prog {[0 := (0, {[ 0 := Start main ]})]} ⌀ tp' h' ->
  get_finished_task tp' 0 0 = Some v ->
  v = v_True.
Proof. ... Qed.

Theorem main_leak_free tp' h' :
  tpool_steps prog {[0 := (0, {[ 0 := Start main ]})]} ⌀ tp' h' ->
  terminated_tp tp' ->
  map_Forall (λ _ obj, let (C, F, lk) := obj in get_aff prog C = true) h'.
Proof. ... Qed.
```

No forbidden configurations are reached

If the program terminates, it yields **true** and all objects in the heap are affine

Thank you!

jd.mota@campus.fct.unl.pt

jdmota.github.io

- Affine and linear resources/invariants
- Thread-local/shared invariants
- Weakest-precondition for concurrent language ensuring safety and leak freedom