Iron: Managing Obligations in Higher-Order Concurrent Separation Logic

Aleš Bizjak[1] **Daniel Gratzer**[1]
Robbert Krebbers[2] Lars Birkedal[1]

[1] Aarhus University
[2] Delft University of Technology

January 17, 2019 POPL 2019

`https://iris-project.org/iron/`

The Problem

Resources we use in programs impose obligations:

- Memory must be properly freed.
- File handles must be closed after use.
- Locks must be acquired and released properly.

Example

```
let ℓ = ref(None) in
fork {
  (rec go () =
    match !ℓ with
     | None     ⇒ go ()
     | Some(x)  ⇒ free(x); free(ℓ)
    end
   end) ()
};
ℓ ← Some(ref(1))
```

Example

Channel

```
let ℓ = ref(None) in
fork {
  (rec go () =
    match !ℓ with
    | None    ⇒ go ()
    | Some(x) ⇒ free(x); free(ℓ)
    end
  end) ()
};
ℓ ← Some(ref(1))
```

Example

Channel

Wait...

```
let ℓ = ref(None) in
fork {
  (rec go () =
    match !ℓ with
     | None    ⇒ go ()
     | Some(x) ⇒ free(x); free(ℓ)
    end
   end) ()
};
ℓ ← Some(ref(1))
```

Example

Channel

```
let ℓ = ref(None) in
fork {
  (rec go () =
    match !ℓ with
     | None    ⇒ go ()
     | Some(x) ⇒ free(x); free(ℓ)
    end
  end) ()
};
ℓ ← Some(ref(1))
```

Wait...

Signal the thread

Example

Channel

```
let ℓ = ref(None) in
fork {
  (rec go () =
    match !ℓ with
     | None    ⇒ go ()
     | Some(x) ⇒ free(x); free(ℓ)
    end
  end) ()
};
ℓ ← Some(ref(1))
```

Wait...

...then clean up

Signal the thread

Wish list

We want a concurrent separation logic to prove these properties:

- Has thread-local reasoning
- Can express complex and modular specifications
- Handles complicated language features (especially fork)
- Is amenable to mechanization
- Can prove leak-freedom

Wish list

We want a concurrent separation logic to prove these properties:

- Has thread-local reasoning
- Can express complex and modular specifications
- Handles complicated language features (especially fork)
- Is amenable to mechanization
- Can prove leak-freedom

Iris (a state of the art concurrent separation logic) gives us the first 4.

Our Contribution

Trackable resources: a general mechanism for managing obligations

and

Iron: a separation logic implementing it:

- Includes all proof techniques of Iris
  (ghost state, impredicative invariants, updates, etc...)
- Supports all the language features of Iris
- Fully mechanized in Coq

Other Approaches: Iris

Iris and other affine logics gives us safety (and correctness):

Theorem

If $\{\textit{True}\}\ e\ \{\textit{True}\}$ *holds then* e *does not get stuck.*

We wish to strengthen this to ensure leak-freedom.

Other Approaches: CSL

O'Hearn [2007] and Brookes [2007] ensured leak-freedom through linearity for *statically scoped concurrency*:

$$\frac{}{\Gamma \vdash \{\mathsf{Emp}\}\ \mathsf{ref}(v)\ \{\ell.\ \ell \mapsto v\}} \qquad \frac{}{\Gamma \vdash \{\ell \mapsto w\}\ \mathsf{free}(\ell)\ \{\mathsf{Emp}\}}$$

$$\frac{\Gamma \vdash \{P_1\}\ e_1\ \{Q_1\} \qquad \Gamma \vdash \{P_2\}\ e_2\ \{Q_2\}}{\Gamma \vdash \{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{Q_1 * Q_2\}}$$

Other Approaches: CSL

O'Hearn [2007] and Brookes [2007] ensured leak-freedom through linearity for *statically scoped concurrency*:

$$\frac{}{\Gamma \vdash \{\mathsf{Emp}\}\ \mathsf{ref}(v)\ \{\ell.\ \ell \mapsto v\}} \qquad \frac{}{\Gamma \vdash \{\ell \mapsto w\}\ \mathsf{free}(\ell)\ \{\mathsf{Emp}\}}$$

Our heap view is empty

$$\frac{\Gamma \vdash \{P_1\}\ e_1\ \{Q_1\} \qquad \Gamma \vdash \{P_2\}\ e_2\ \{Q_2\}}{\Gamma \vdash \{P_1 * P_2\}\ e_1 \mathbin{||} e_2\ \{Q_1 * Q_2\}}$$

Other Approaches: CSL

O'Hearn [2007] and Brookes [2007] ensured leak-freedom through linearity for *statically scoped concurrency*:

$$\frac{}{\Gamma \vdash \{\mathsf{Emp}\}\ \mathsf{ref}(v)\ \{\ell.\ \ell \mapsto v\}} \qquad \frac{}{\Gamma \vdash \{\ell \mapsto w\}\ \mathsf{free}(\ell)\ \{\mathsf{Emp}\}}$$

$$\frac{\Gamma \vdash \{P_1\}\ e_1\ \{Q_1\} \qquad \Gamma \vdash \{P_2\}\ e_2\ \{Q_2\}}{\Gamma \vdash \{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{Q_1 * Q_2\}}$$

Our heap view is empty

Does not share memory

Scoped Invariants

With only parallel composition scoped invariants are sufficient:

$$\frac{\Gamma, r : R \vdash \{P\}\, e\, \{Q\}}{\Gamma \vdash \{P * R\}\, \textsf{resource}\ r\ \textsf{in}\ e\, \{Q * R\}}$$

Scoped invariants are insufficient for the "unscoped concurrency":

$$\begin{array}{l}
\textsf{let}\ \ell = \textsf{ref}(1)\ \textsf{in} \\
\textsf{resource}\ r\ \textsf{in} \\
\quad \textsf{fork}\ \{\textsf{with}\ r\ \textsf{do}\ !\ell\}\ ; \\
\textsf{free}(\ell)
\end{array}$$

Unscoped Invariants

With fork we need unscoped invariants:

$$\frac{\left\{P * \boxed{R}^{\mathcal{N}}\right\} e \{v.\, Q\}}{\{P * R\} e \{v.\, Q\}}$$

- Invariants persist forever and can be duplicated freely.
- There is no deallocation rule; it must be encoded.

Unscoped Invariants

With fork we need unscoped invariants:

$$\frac{\left\{P * \boxed{R}^{\mathcal{N}}\right\} e \{v.\ Q\}}{\{P * R\} e \{v.\ Q\}}$$

- Invariants persist forever and can be duplicated freely.
- There is no deallocation rule; it must be encoded.

We can always put resources in an invariant and forget them – no linearity!

Resolving This Tension

- Scoped **tracks** obligations but does **not handle** fork.
- Unscoped **handles** fork but does **not track** obligations.
- Invariants are complex to prove sound; we prefer not to modify them.

We will modify $\mapsto$ instead so that unscoped invariants are suitable.

Crucial Idea: Trackable Resources

We keep the affine logic so Iris's implementation of invariants can be reused.

- Index $\ell \mapsto_\pi v$ with $\pi \in (0, 1]$
- Add a new proposition $\mathfrak{e}_\pi$ with $\pi \in (0, 1]$

π indicates *how much of the heap* we know about through the proposition.

Intuition for Fractions

If we own...

- $\ell \mapsto_1 v$ then the only thing the heap contains is $\ell \mapsto v$.
- $\mathfrak{e}_1$ the heap contains nothing at all.
- $\ell \mapsto_\pi v$ and $\pi < 1$ the heap may contain other locations.
- $\ell_1 \mapsto_{1/2} v$ and $\ell_2 \mapsto_{1/2} w$ the heap contains just ℓ_1 and ℓ_2.

We can prove leak-freedom by owning $\mathfrak{e}_1$.

Intuition for Fractions

If we own...

- $\ell \mapsto_1 v$ then the only thing the heap contains is $\ell \mapsto v$.
- $\mathfrak{e}_1$ the heap contains nothing at all.
- $\ell \mapsto_\pi v$ and $\pi < 1$ the heap may contain other locations.
- $\ell_1 \mapsto_{1/2} v$ and $\ell_2 \mapsto_{1/2} w$ the heap contains just ℓ_1 and ℓ_2.

We can prove leak-freedom by owning $\mathfrak{e}_1$.

Intuition for Fractions

If we own...

- $\ell \mapsto_1 v$ then the only thing the heap contains is $\ell \mapsto v$.
- $\mathfrak{e}_1$ the heap contains nothing at all.
- $\ell \mapsto_\pi v$ and $\pi < 1$ the heap may contain other locations.
- $\ell_1 \mapsto_{1/2} v$ and $\ell_2 \mapsto_{1/2} w$ the heap contains just ℓ_1 and ℓ_2.

We can prove leak-freedom by owning $\mathfrak{e}_1$.

Intuition for Fractions

If we own...

- $\ell \mapsto_1 v$ then the only thing the heap contains is $\ell \mapsto v$.
- $\mathfrak{e}_1$ the heap contains nothing at all.
- $\ell \mapsto_\pi v$ and $\pi < 1$ the heap may contain other locations.
- $\ell_1 \mapsto_{1/2} v$ and $\ell_2 \mapsto_{1/2} w$ the heap contains just ℓ_1 and ℓ_2.

We can prove leak-freedom by owning $\mathfrak{e}_1$.

Fractional Permissions?

What does $\ell \mapsto_\pi v$ mean?

- With fractional permissions we own π of the *location*.
- With Iron we own π of the *entire heap*.

Crucial difference: we can write to $\ell \mapsto_{1/2} v$ in Iron but not in Boyland [2003].

Working with Fractions in Programs: The Heap

The program logic adapts to handle these fractions as follows:

$$\frac{}{\{\mathfrak{e}_\pi\}\ \mathsf{ref}(v)\ \{\ell.\ \ell \mapsto_\pi v\}} \qquad \frac{}{\{\ell \mapsto_\pi w\}\ \mathsf{free}(\ell)\ \{\mathfrak{e}_\pi\}}$$

$$\frac{}{\{\ell \mapsto_\pi v\}\ !\ell\ \{w.\ w = v \wedge \ell \mapsto_\pi v\}} \qquad \frac{}{\{\ell \mapsto_\pi w\}\ \ell \leftarrow v\ \{\ell \mapsto_\pi v\}}$$

$$\frac{}{\mathfrak{e}_{\pi_1} * \mathfrak{e}_{\pi_2} \dashv\vdash \mathfrak{e}_{\pi_1+\pi_2}} \qquad \frac{}{(\ell \mapsto_{\pi_1} v) * \mathfrak{e}_{\pi_2} \dashv\vdash \ell \mapsto_{\pi_1+\pi_2} v}$$

Working with Fractions in Programs: Concurrency

The standard rule for fork holds:

$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\ \mathsf{fork}\ \{e\}\ \{v.\, v = ()\}}$$

This rule is insufficient if the forked-off thread outlives its parent:

$$\begin{array}{l} \mathsf{fork}\ \{ \\ \quad \mathsf{let}\ \ell = \mathsf{ref}(1)\ \mathsf{in} \\ \quad \mathsf{free}(\ell) \\ \}; \\ 1 + 1 \end{array}$$

Working with Fractions in Programs: Concurrency

We must also allow the forked-off thread to terminate with $\mathfrak{e}_\pi$:

$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\ \mathsf{fork}\ \{e\}\ \{v.\, v = ()\}} \qquad \frac{\{P\}\, e\, \{\mathfrak{e}_\pi\}}{\{P\}\ \mathsf{fork}\ \{e\}\ \{v.\, v = () * \mathfrak{e}_\pi\}}$$

Adequacy

Iron provides us with strong guarantees about programs:

Theorem

If $\{\mathfrak{e}_\pi\}\, e\, \{\mathfrak{e}_\pi\}$*:*

1. e *does not get stuck*
2. *If* $(e, h) \mapsto^* ([v, \underbrace{v_0, \ldots, v_n}_{\textit{thread results}}], h')$ *then* $h = h'$*.*

Adequacy

Iron provides us with strong guarantees about programs:

Theorem

If $\{\mathfrak{e}_\pi\}\, e\, \{\mathfrak{e}_\pi\}$*:*

1. e *does not get stuck*
2. *If* $(e, h) \mapsto^* ([v, \underbrace{v_0, \ldots, v_n}_{\text{thread results}}], h')$ *then* $h = h'$*.*

If we forget part of $\mathfrak{e}_\pi$ then we cannot apply our adequacy theorem; the triple won't hold!

Taking Stock

At this point, Iron is already useful!

Taking Stock

At this point, Iron is already useful!

But it isn't easy; there's boilerplate with fractions everywhere:

$$\{(\pi_1 + \pi_2 = 1) * (\ell_1 \mapsto_{\pi_1} v_1) * (\ell_2 \mapsto_{\pi_2/2} v_2) * (\ell_3 \mapsto_{\pi_2/2} v_3)\}$$

$$\quad \textsf{free}(\ell_1);\ \textsf{free}(\ell_2);\ \textsf{free}(\ell_3)$$

$$\{\mathfrak{e}_1\}$$

The Lifted Logic

We can lift the operators of BI to functions, $[0, 1] \to$ *iProp*.

$$
\begin{aligned}
(P * Q)(\pi) &\triangleq \exists \pi_1, \pi_2.\, \pi_1 + \pi_2 = \pi \land P(\pi_1) * Q(\pi_2) \\
(\ell \mathrel{\widehat{\mapsto}} v)(\pi) &\triangleq \pi > 0 \land \ell \mapsto_\pi v \\
\mathsf{Emp}(\pi) &\triangleq \pi = 0
\end{aligned}
$$

Other operations are lifted point-wise.

The Lifted Logic

We can lift the operators of BI to functions, $[0,1] \to$ *iProp*.

$$(P * Q)(\pi) \triangleq \exists \pi_1, \pi_2.\, \pi_1 + \pi_2 = \pi \wedge P(\pi_1) * Q(\pi_2)$$
$$(\ell \mathrel{\widehat{\mapsto}} v)(\pi) \triangleq \pi > 0 \wedge \ell \mapsto_\pi v$$
$$\mathsf{Emp}(\pi) \triangleq \pi = 0$$

Other operations are lifted point-wise.

The lifted logic is really linear!

$$\ell_1 \mathrel{\widehat{\mapsto}} v_1 * \ell_2 \mathrel{\widehat{\mapsto}} v_2 \not\vdash \ell_1 \mathrel{\widehat{\mapsto}} v_1$$

The Lifted Logic: New Rules

The lifted program logic mirrors standard linear separation logic:

$$\frac{}{\{\mathsf{Emp}\}\ \mathsf{ref}(v)\ \{\ell.\ \ell \mathrel{\widehat{\mapsto}} v\}}$$

$$\frac{}{\{\ell \mathrel{\widehat{\mapsto}} -\}\ \mathsf{free}(\ell)\ \{\mathsf{Emp}\}}$$

$$\frac{}{\{\ell \mathrel{\widehat{\mapsto}} v\}\ !\ell\ \{w.\ w = v \wedge \ell \mathrel{\widehat{\mapsto}} v\}}$$

$$\frac{}{\{\ell \mathrel{\widehat{\mapsto}} -\}\ \ell \leftarrow v\ \{\ell \mathrel{\widehat{\mapsto}} v\}}$$

$$\frac{\{P\}\ e\ \{\mathsf{Emp}\}}{\{P\}\ \mathsf{fork}\ \{e\}\ \{v.\ v = () \wedge \mathsf{Emp}\}}$$

The Lifted Logic: Invariants

- We developed a specialized form of invariants for lifted propositions.
- They require permission to open in order to support deallocation.
- They integrate well with the lifted logic but are limited.

The Lifted Logic: Invariants

- We developed a specialized form of invariants for lifted propositions.
- They require permission to open in order to support deallocation.
- They integrate well with the lifted logic but are limited.

Sometimes we still need the unlifted logic for the more general invariants.

$$
\begin{aligned}
& e_1 \mid\mid e_2 \triangleq \\
& \quad \textsf{let } h = \textsf{spawn}(\lambda\_.\, e_1) \textsf{ in} \\
& \quad \textsf{let } v_2 = e_2 \textsf{ in} \\
& \quad \textsf{let } v_1 = \textsf{join}(h) \textsf{ in} \\
& \quad (v_1, v_2)
\end{aligned}
$$

Using Iron

We've used Iron to formalize a number of examples:

1. An implementation of $e_1 \mid\mid e_2$
2. Various examples of resource transfer
3. A lock-free queue
4. An asynchronous message system with cleanup

Aside from the first, all of these are proven in the lifted logic.

Conclusions

Trackable resources: a general mechanism for managing obligations

and

Iron: a separation logic implementing it:

- Includes all proof techniques of Iris
  (ghost state, impredicative invariants, updates, etc...)
- Supports all the language features of Iris
- Fully mechanized in Coq

`https://iris-project.org/iron/`