Iron: Managing Obligations in Higher-Order Concurrent Separation Logic

Aleš Bizjak¹ **Daniel Gratzer**¹ Robbert Krebbers² Lars Birkedal¹

¹Aarhus University ²Delft 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.

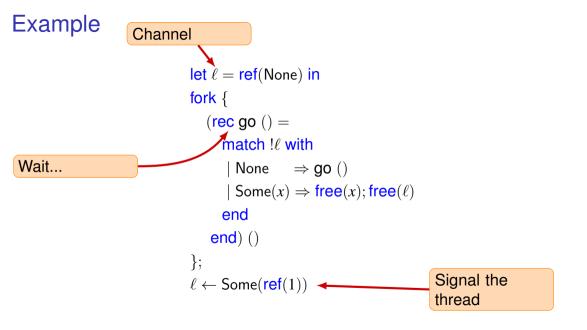
Example

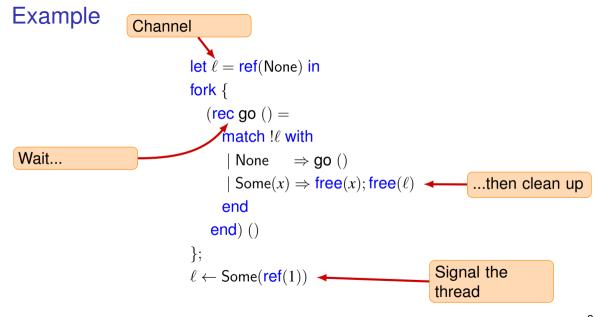
```
let \ell = \text{ref}(None) in
fork {
   (rec go () =
      match ! with
       | None \Rightarrow go ()
       | Some(x) \Rightarrow free(x); free(\ell)
      end
    end) ()
\ell \leftarrow \mathsf{Some}(\mathsf{ref}(1))
```

Example

```
Channel
            let \ell = \text{ref(None)} in
            fork {
               (rec go () =
                  match ! with
                   | None \Rightarrow go ()
                   | Some(x) \Rightarrow free(x); free(\ell)
                  end
                end)()
           \ell \leftarrow \mathsf{Some}(\mathsf{ref}(1))
```

```
Example
                       Channel
                                   let \ell = \text{ref(None)} in
                                   fork {
                                      (rec go () =
                                        match !ℓ with
 Wait...
                                          | None \Rightarrow go ()
                                          | Some(x) \Rightarrow free(x); free(\ell)
                                         end
                                       end)()
                                  \ell \leftarrow \mathsf{Some}(\mathsf{ref}(1))
```





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 $\{ True \} e \{ True \}$ 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*:

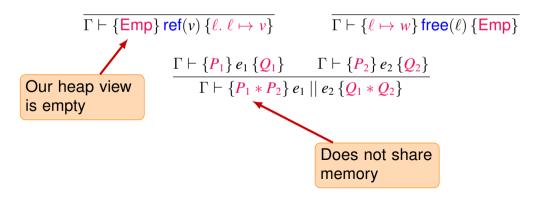
Other Approaches: CSL

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

$$\overline{\Gamma \vdash \{\mathsf{Emp}\}\,\mathsf{ref}(v)\,\{\ell.\,\,\ell\mapsto v\}} \qquad \overline{\Gamma \vdash \{\ell\mapsto w\}\,\mathsf{free}(\ell)\,\{\mathsf{Emp}\}}$$
 Our heap view is empty
$$\frac{\Gamma \vdash \{P_1\}\,e_1\,\{Q_1\} \qquad \Gamma \vdash \{P_2\}\,e_2\,\{Q_2\}}{\Gamma \vdash \{P_1*P_2\}\,e_1\mid\mid e_2\,\{Q_1*Q_2\}}$$

Other Approaches: CSL

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



Scoped Invariants

With only parallel composition scoped invariants are sufficient:

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

Scoped invariants are insufficient for the "unscoped concurrency":

```
\begin{split} & \text{let } \ell = \text{ref}(1) \text{ in} \\ & \text{resource } r \text{ in} \\ & \text{fork } \{ \text{with } r \text{ do } ! \ell \} \, ; \\ & \text{free}(\ell) \end{split}
```

Unscoped Invariants

With fork we need unscoped invariants:

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

- 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:

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

- 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}_{π} with $\pi \in (0,1]$

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

If we own...

- $\ell \mapsto_1 \nu$ then the only thing the heap contains is $\ell \mapsto \nu$.
- ε₁ the heap contains nothing at all.
- $\ell \mapsto_{\pi} \nu$ 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 ε_1 .

If we own...

- $\ell \mapsto_1 \nu$ then the only thing the heap contains is $\ell \mapsto \nu$.
- e₁ the heap contains nothing at all.
- $\ell \mapsto_{\pi} \nu$ 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 ε_1 .

If we own...

- $\ell \mapsto_1 \nu$ then the only thing the heap contains is $\ell \mapsto \nu$.
- ε₁ 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 e_1 .

If we own...

- $\ell \mapsto_1 \nu$ then the only thing the heap contains is $\ell \mapsto \nu$.
- ε₁ the heap contains nothing at all.
- $\ell \mapsto_{\pi} \nu$ 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 e_1 .

Fractional Permissions?

What does $\ell \mapsto_{\pi} \nu$ 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 $\ell \mapsto_{1/2} \nu$ in Iron but not in Boyland [2003].

Working with Fractions in Programs: The Heap

The program logic adapts to handle these fractions as follows:

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:

```
\label{eq:fork} \begin{split} & \text{fork } \{ \\ & \text{let } \ell = \text{ref}(1) \text{ in } \\ & \text{free}(\ell) \\ \}; \\ & 1+1 \end{split}
```

Working with Fractions in Programs: Concurrency

We must also allow the forked-off thread to terminate with e_{π} :

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

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 $(e,h)\mapsto^* ([v,\underbrace{v_0,...,v_n}_{\textit{thread results}}],h')$ 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 $(e,h)\mapsto^* ([v,\underbrace{v_0,...,v_n}_{\textit{thread results}}],h')$ then h=h'.

If we forget part of e_{π} 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:

```
 \begin{aligned} & \left\{ (\pi_1 + \pi_2 = 1) * (\ell_1 \mapsto_{\pi_1} \nu_1) * (\ell_2 \mapsto_{\pi_2/2} \nu_2) * (\ell_3 \mapsto_{\pi_2/2} \nu_3) \right\} \\ & \text{free}(\ell_1); \ \text{free}(\ell_2); \ \text{free}(\ell_3) \\ & \left\{ \mathfrak{e}_1 \right\} \end{aligned}
```

The Lifted Logic

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

$$(P * Q)(\pi) \triangleq \exists \pi_1, \pi_2. \pi_1 + \pi_2 = \pi \land P(\pi_1) * Q(\pi_2)$$

 $(\ell \widehat{\mapsto} v)(\pi) \triangleq \pi > 0 \land \ell \mapsto_{\pi} v$
 $\mathsf{Emp}(\pi) \triangleq \pi = 0$

Other operations are lifted point-wise.

The Lifted Logic

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

$$(P*Q)(\pi) \triangleq \exists \pi_1, \pi_2. \pi_1 + \pi_2 = \pi \land P(\pi_1) * Q(\pi_2)$$

 $(\ell \mapsto v)(\pi) \triangleq \pi > 0 \land \ell \mapsto_{\pi} v$
 $\mathsf{Emp}(\pi) \triangleq \pi = 0$

Other operations are lifted point-wise.

The lifted logic is really linear!

$$\ell_1 \widehat{\mapsto} \nu_1 * \ell_2 \widehat{\mapsto} \nu_2 \not\vdash \ell_1 \widehat{\mapsto} \nu_1$$

The Lifted Logic: New Rules

The lifted program logic mirrors standard linear separation logic:

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.

```
e_1 \mid\mid e_2 \triangleq 

let h = \operatorname{spawn}(\lambda_-.e_1) in

let v_2 = e_2 in

let v_1 = \operatorname{join}(h) in

(v_1, v_2)
```

Using Iron

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

- 1. An implementation of $e_1 \parallel e_2$
- 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/