A Relational Separation Logic for Effect Handlers

joint work with Simcha van Collem, Ines Wright, and Robbert Krebbers

presented by **Paulo Emílio de Vilhena**

on the 15th of January, 2026

Introduction

Goal. Design of a *relational separation logic* for *effect handlers*.

Goal. Design of a *relational separation logic* for *effect handlers*.

In short, a *relational separation logic* consists of
an *assertion language*, to specify programs;
and a set of *proof rules*, to verify programs compositionally.

The *key* feature is the *refinement relation*, to assert that e_s is a correct abstraction of e_i:

$$e_i \lesssim e_s \; \{R\} \quad \triangleq \quad \text{“\textit{if} } e_i \text{ \textit{terminates with value} } v_i\text{, \textit{then} } e_s \text{ \textit{terminates with a value} } v_s \text{ \textit{s.t.} } R(v_i, v_s)\text{”}$$

Goal. Design of a *relational separation logic* for *effect handlers*.

In short, a *relational separation logic* consists of
an *assertion language*, to specify programs;
and a set of *proof rules*, to verify programs compositionally.

The *key* feature is the *refinement relation*, to assert that e_s is a correct abstraction of e_i:

$$e_i \precsim e_s \; \{R\} \quad \triangleq \quad \text{“if } e_i \text{ terminates with value } v_i\text{, then } e_s \text{ terminates with a value } v_s \text{ s.t. } R(v_i, v_s)\text{”}$$

Applications.

- ***Program Verification & Program Reasoning.***
  To *specify* and *understand* a program in terms of a *simpler implementation*.
- ***Compiler Optimisations.***
  An optimisation is *correct* if the *optimised program* does *not* introduce *behaviours*.
- ***Type Systems.***
  To show *soundness* and *abstraction properties* of type systems.

Goal. Design of a *relational separation logic* for *effect handlers*.

In short, a *relational separation logic* consists of
an *assertion language*, to specify programs;
and a set of *proof rules*, to verify programs compositionally.

The *key* feature is the *refinement relation*, to assert that e_s is a correct abstraction of e_i:

$$e_i \lesssim e_s \; \{R\} \quad \triangleq \quad \text{“\textit{if } } e_i \text{ \textit{terminates with value} } v_i\text{, \textit{then} } e_s \text{ \textit{terminates with a value} } v_s \text{ \textit{s.t.} } R(v_i, v_s)\text{”}$$

Applications.

- ***Program Verification & Program Reasoning.***
  To *specify* and *understand* a program in terms of a *simpler implementation*.
- ***Compiler Optimisations.***
  An optimisation is *correct* if the *optimised program* does *not* introduce *behaviours*.
- ***Type Systems.***
  To show *soundness* and *abstraction properties* of type systems.

Example

A *relational separation logic* allows an *effect-handler-based* implementation of *concurrency* to be explained in terms of a *direct* implementation:

```
effect Fork : (unit -> unit) -> unit
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

Example

A *relational separation logic* allows an *effect-handler-based* implementation of *concurrency* to be explained in terms of a *direct* implementation:

```
effect Fork : (unit -> unit) -> unit
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

$\lesssim$

```
main (fun f -> fork (f ()))
```

Example

A *relational separation logic* allows an *effect-handler-based* implementation of *concurrency* to be explained in terms of a *direct* implementation:

```
effect Fork : (unit -> unit) -> unit
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

$\lesssim$

```
main (fun f -> fork (f ()))
```

It formalises the intuition, that, under this handler, an effect `Fork` can be seen as **`fork`** itself:

`perform (Fork f)` $\lesssim$ `fork (f ())`

Challenges

The *meaning* of an *effect* depends on a *handler*.

1. *Definition of the Refinement Relation.*
The standard refinement relation does not *specify* the case of *effects*:

$e_i \lesssim e_s \; \{R\} \;\triangleq\;$ "*if* e_i *terminates with value* v_i, *then* e_s *terminates with a value* v_s *s.t.* $R(v_i, v_s)$"

2. *Compositional Reasoning (Handler vs. Handlee).*
How to *reason* about a program that *performs* effects *independently* of its *handler*?

3. *Context-Local Reasoning.*
How to *reason* about a program *independently* of its *evaluation context*?

Challenges

The *meaning* of an *effect* depends on a *handler*.

1. *Definition of the Refinement Relation.*
The standard refinement relation does not *specify* the case of *effects*:

$$e_i \precsim e_s \; \{R\} \quad \triangleq \quad \text{“}\textit{if } e_i \textit{ terminates with value } v_i\textit{, then } e_s \textit{ terminates with a value } v_s \textit{ s.t. } R(v_i, v_s)\text{”}$$

2. *Compositional Reasoning (Handler vs. Handlee).*
How to *reason* about a program that *performs* effects *independently* of its *handler*?

3. *Context-Local Reasoning.*
How to *reason* about a program *independently* of its *evaluation context*?

Challenges

The *meaning* of an *effect* depends on a *handler*.

1. ***Definition of the Refinement Relation.***
   The standard refinement relation does not *specify* the case of *effects*:

   $e_i \lesssim e_s \; \{R\} \quad \triangleq \quad$ "*if* e_i *terminates with value* v_i, *then* e_s *terminates with a value* v_s *s.t.* $R(v_i, v_s)$"

2. ***Compositional Reasoning (Handler vs. Handlee).***
   How to *reason* about a program that *performs* effects *independently* of its *handler*?

3. ***Context-Local Reasoning.***
   How to *reason* about a program *independently* of its *evaluation context*?

```
match main (fun f -> perform (Fork f)) with     ≲     main (fun f -> fork (f ()))
| effect (Fork f), k -> h
| _ -> r
```

Handlee Part

```
main (fun f -> perform (Fork f)) ≲
main (fun f -> fork (f ()))
```

Handler Part

The *meaning* of an *effect* depends on a *handler*.

1. ***Definition of the Refinement Relation.***
The standard refinement relation does not *specify* the case of *effects*:

$$e_i \lesssim e_s \; \{R\} \quad \triangleq \quad \text{“}\textit{if } e_i \textit{ terminates with value } v_i\textit{, then } e_s \textit{ terminates with a value } v_s \textit{ s.t. } R(v_i, v_s)\text{”}$$

2. ***Compositional Reasoning (Handler vs. Handlee).***
How to *reason* about a program that *performs* effects *independently* of its *handler*?

3. ***Context-Local Reasoning.***
How to *reason* about a program *independently* of its *evaluation context*?

$$\frac{e_i \lesssim e_s \; \{y_i, y_s.\; K_i[y_i] \lesssim K_s[y_s] \; \{R\}\}}{K_i[e_i] \lesssim K_s[e_s] \; \{R\}} \; \textit{(Standard) Bind}$$

Key Idea

The *key idea* is to extend the refinement relation with a *parameterised relational theory*, an *axiomatisation* of *relations* that should hold:

$$e_i \lesssim e_s \langle \mathcal{T} \rangle \{R\}$$

The resulting logic is called *baze*; it is built on top of *Iris*.

A *relational theory* is formalised in *Iris* as a *set* of *admitted relations* (on arbitrary expressions):

$$\mathcal{T} : (\underbrace{\texttt{expr}}_{\textit{impl.}} \times \underbrace{\texttt{expr}}_{\textit{spec.}} \times \underbrace{((\texttt{expr} \times \texttt{expr}) \to \texttt{iProp})}_{\substack{\textit{return condition} \\ \textit{(postcondition)}}}) \to \underbrace{\texttt{iProp}}_{\textit{precondition}}$$

Key Idea

The *key idea* is to extend the refinement relation with a *parameterised relational theory*, an *axiomatisation* of *relations* that should hold:

$$e_i \lesssim e_s \, \langle \mathcal{T} \rangle \, \{R\}$$

The resulting logic is called *baze*; it is built on top of *Iris*.

A *relational theory* is formalised in *Iris* as a *set* of *admitted relations* (on arbitrary expressions):

$$\mathcal{T} : (\underbrace{\texttt{expr}}_{\textit{impl.}} \times \underbrace{\texttt{expr}}_{\textit{spec.}} \times \underbrace{((\texttt{expr} \times \texttt{expr}) \to \texttt{iProp})}_{\substack{\textit{return condition} \\ \textit{(postcondition)}}}) \to \underbrace{\texttt{iProp}}_{\textit{precondition}}$$

Examples. Empty theory.

$$\bot(e_i,\ e_s,\ R) = \mathit{False}$$

$$e_i \lesssim e_s \, \{R\} \quad \Leftrightarrow \quad e_i \lesssim e_s \, \langle \bot \rangle \, \{R\}$$

Key Idea

The *key idea* is to extend the refinement relation with a *parameterised relational theory*, an *axiomatisation* of *relations* that should hold:

$$e_i \precsim e_s \langle \mathcal{T} \rangle \{R\}$$

The resulting logic is called *baze*; it is built on top of *Iris*.

A *relational theory* is formalised in *Iris* as a *set* of *admitted relations* (on arbitrary expressions):

$$\mathcal{T} : (\underbrace{\texttt{expr}}_{\textit{impl.}} \times \underbrace{\texttt{expr}}_{\textit{spec.}} \times \underbrace{((\texttt{expr} \times \texttt{expr}) \to \texttt{iProp})}_{\substack{\textit{return condition}\\ \textit{(postcondition)}}}) \to \underbrace{\texttt{iProp}}_{\textit{precondition}}$$

Examples. Concurrency effects.

$$\texttt{FORK}(\texttt{perform}\ (\texttt{Fork}\ f_i),\ \textbf{fork}\ (f_s\ ()),\ R) =$$
$$\quad \triangleright\ f_i\ () \precsim f_s\ () \langle \texttt{FORK} \rangle \{\mathit{True}\} * R((), ())$$

$$\triangleright\ f_i\ () \precsim f_s\ () \langle \texttt{FORK} \rangle \{\mathit{True}\} \twoheadrightarrow$$
$$\texttt{perform}\ (\texttt{Fork}\ f_i) \precsim \textbf{fork}\ (f_s\ ()) \langle \texttt{FORK} \rangle \{y_i, y_s.\ y_i = y_s = ()\}$$

Challenge 1 – Definition of the Refinement Relation in baze

Problem. The *meaning* of an *effect* depends on a *handler*.

Solution. (**Biorthogonality**) To *universally quantify* over *contexts* that *validate* a *theory*.

Under the hood, the *parameterised refinement relation* unfolds to a *standard refinement* with e_i and e_s under *universally quantified contexts*:

$$e_i \lesssim e_s \langle \mathcal{T} \rangle \{R\} \triangleq \forall K_i\ K_s\ S.\ \langle \mathcal{T} \rangle \{R\}\ K_i \lesssim K_s\ \{S\} \twoheadrightarrow K_i[e_i] \lesssim K_s[e_s]\ \{S\}$$

Definition of the *validation of a relational theory* $\mathcal{T}$ by a *pair of contexts*:

$$\langle \mathcal{T} \rangle \{R\}\ K_i \lesssim K_s\ \{S\} \triangleq$$

$$(\forall v_i\ v_s.\ R(v_i, v_s) \twoheadrightarrow K_i[v_i] \lesssim K_s[v_s]\ \{S\})$$

$$\wedge$$

$$(\forall e_i'\ e_s'.\ \underbrace{\mathcal{T} \langle e_i', e_s', R \rangle}_{\approx\ \mathcal{T}(e_i', e_s', R)} \twoheadrightarrow K_i[e_i'] \lesssim K_s[e_s']\ \{S\})$$

The *exhaustion rule* allows *compositional reasoning* about programs with *effect handlers*.

$$\frac{\begin{array}{l} e_i \lesssim e_s \langle \mathcal{T} \rangle \{R\} \\ (\forall\, v_i\ v_s.\ R(v_i, v_s) \twoheadrightarrow K_i[v_i] \lesssim K_s[v_s] \langle \mathcal{F} \rangle \{S\}) \\ \wedge \\ (\forall\, e_i'\ e_s'.\ \mathcal{T}\langle e_i', e_s', R\rangle \twoheadrightarrow K_i[e_i'] \lesssim K_s[e_s'] \langle \mathcal{F} \rangle \{S\}) \end{array}}{K_i[e_i] \lesssim K_s[e_s] \langle \mathcal{F} \rangle \{S\}}\ \textit{Exhaustion}$$

The rule allows one to see the *theory* $\mathcal{T}$ as a *boundary* between *handlee* and *handler*.

Challenge 3 – Context-Local Reasoning

The *bind rule* allows *context-local reasoning*:

$$\frac{\mathit{traversable}(\mathtt{K_i},\ \mathtt{K_s},\ \mathcal{T}) \qquad \mathtt{e_i} \lesssim \mathtt{e_s}\ \langle \mathcal{T} \rangle\ \{y_i,\ y_s.\ \mathtt{K_i}[y_i] \lesssim \mathtt{K_s}[y_s]\ \langle \mathcal{T} \rangle\ \{R\}\}}{\mathtt{K_i}[\mathtt{e_i}] \lesssim \mathtt{K_s}[\mathtt{e_s}]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Bind}$$

The *contexts* should be able to *"traverse"* the *relational theory* $\mathcal{T}$:

$\mathit{traversable}(\mathtt{K_i},\ \mathtt{K_s},\ \mathcal{T})$ = *"The theory $\mathcal{T}$ holds regardless of the contexts $\mathtt{K_i}$ and $\mathtt{K_s}$."*

The *context-closure* of a theory is *traversable by construction*:

$$(E_i,\ E_s) \Downarrow\!\upharpoonright \mathcal{T}$$

a pair of sets of effects

Properties.

1. The *context-closure* of $\mathcal{T}$ extends $\mathcal{T}$:

$$\mathcal{T}(e_i,\ e_s,\ R) \twoheadrightarrow ((E_i,\ E_s) \Downarrow\!\upharpoonright \mathcal{T})(e_i,\ e_s,\ R)$$

2. The *context-closure* of $\mathcal{T}$ is *traversable* by *neutral contexts*:

$$traversable(K_i,\ K_s,\ ((E_i,\ E_s) \Downarrow\!\upharpoonright \mathcal{T})) \Leftarrow neutral(E_i,\ K_i) \land \underbrace{neutral(E_s,\ K_s)}$$

K_s *has no handler for an effect in* E_s

Under a *context-closed theory*, the *bind rule* can be *simplified* as follows:

$$\frac{neutral(E_i,\ K_i) \qquad neutral(E_s,\ K_s) \qquad e_i \lesssim e_s \ \langle (E_i,\ E_s) \Downarrow\!\upharpoonright \mathcal{T} \rangle \ \{y_i,\ y_s.\ K_i[y_i] \lesssim K_s[y_s] \ \langle (E_i,\ E_s) \Downarrow\!\upharpoonright \mathcal{T} \rangle \ \{R\}\}}{K_i[e_i] \lesssim K_s[e_s] \ \langle (E_i,\ E_s) \Downarrow\!\upharpoonright \mathcal{T} \rangle \ \{R\}} \text{ Derived Bind}$$

We can now revisit the refinement between the two implementations of concurrency:

```
effect Fork : (unit -> unit) -> unit           ≲      main (fun f -> fork (f ()))
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

We can now revisit the refinement between the two implementations of concurrency:

```
effect Fork : (unit -> unit) -> unit
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

$\lesssim$

```
main (fun f -> fork (f ()))
```

Key Steps.

1. *Identify* the *theory* to reason about the Fork *effects*:

([Fork], []) $\Downarrow\!\!\Uparrow$ FORK

FORK(perform (Fork f_i), **fork** (f_s ()), R) =
  ▷ f_i () $\lesssim$ f_s () ⟨FORK⟩ {*True*} ∗ R((), ())

Concurrency

We can now revisit the refinement between the two implementations of concurrency:

```
effect Fork : (unit -> unit) -> unit
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

$\lesssim$

```
main (fun f -> fork (f ()))
```

Key Steps.

1. *Identify* the *theory* to reason about the `Fork` *effects*:

```
([Fork], []) ⇃↾ FORK

FORK(perform (Fork f_i), fork (f_s ()), R) =
  ▷ f_i () ≾ f_s () ⟨FORK⟩ {True} ∗ R((), ())
```

2. *Apply* the *exhaustion rule* to *decompose* the *proof* into a *handler* part and a *handlee* part:

```
main (fun f -> perform (Fork f)) ≾
main (fun f -> fork (f ()))
 ⟨([Fork], []) ⇃↾ FORK⟩ {True}
```

We can now revisit the refinement between the two implementations of concurrency:

```
effect Fork : (unit -> unit) -> unit
let q = Queue.create () in
let rec run f =
  match f () with
  | effect (Fork f), k ->
      Queue.push k q;
      run f
  | _ ->
      if not (Queue.empty q) then
        let k = Queue.pop q in continue k ()
in
run (fun () -> main (fun f -> perform (Fork f)))
```

$\lesssim$

```
main (fun f -> fork (f ()))
```

Key Steps.

1. *Identify* the *theory* to reason about the `Fork` *effects*:

   `([Fork], []) ⥯ FORK`

   `FORK(perform (Fork` f_i`), fork (`f_s` ()),` R`) =`
   `▷` f_i `() ≲` f_s `() ⟨FORK⟩ {`*True*`} *` R`((), ())`

2. *Apply* the *exhaustion rule* to *decompose* the *proof* into a *handler* part and a *handlee* part:

   `main (fun f -> perform (Fork f)) ≲`
   `main (fun f -> fork (f ()))`
   `⟨([Fork], []) ⥯ FORK⟩ {`*True*`}`

3. *Apply* the *bind rule* to *step through* the *verification*.

Concurrency

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\mathbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$\frac{\begin{array}{l} i \Rrightarrow K[e_s] \\ \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\} \end{array}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{\begin{array}{l} i \Rrightarrow K_s[e_s] \\ e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\} \end{array}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

Concurrency

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow\!\!\ast\ e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\textbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$\frac{}{\texttt{effect (Fork } f_i\texttt{), } k_i \texttt{ -> } h \lesssim K_s[\textbf{fork}\ (f_s\ ())]}\ \ast$$

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow\!\!\ast\ e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] \ast R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow\!\!\ast\ K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\mathbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$i \Rrightarrow f_s\ () \quad * \quad h \lesssim K_s[()]$$

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\mathbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$i \Rrightarrow f_s\ ()$
$\text{Queue.push}\ k_i\ q;\ \text{run}\ f_i \lesssim K_s[()]$ *

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\mathbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$\frac{j \Rrightarrow K'[K_s[()]]}{\text{Queue.push } k_i\ q;\ \text{run } f_i \lesssim f_s\ ()}\ *$$

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

Concurrency

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow\!\!\ast\ e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\textbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$\frac{}{\text{run } f_i \lesssim f_s\ ()}\ \ast$$

$$\frac{\begin{array}{l} i \Rrightarrow K[e_s] \\ \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow\!\!\ast\ e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] \ast R(v_i, v_s')\} \end{array}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{\begin{array}{l} i \Rrightarrow K_s[e_s] \\ e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow\!\!\ast\ K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\} \end{array}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

Concurrency

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\mathbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

```
------------------------------------------*
let kᵢ = Queue.pop q in continue kᵢ () ≲ ()
```

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i,\ \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i,\ v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

Concurrency

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow\!\!\!\ast\ e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\texttt{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$\frac{j \Rrightarrow K'[e_s] \qquad \texttt{continue}\ k_i\ () \lesssim e_s}{\texttt{continue}\ k_i\ () \lesssim ()}\ \ast$$

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow\!\!\!\ast\ e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] \ast R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow\!\!\!\ast\ K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

To *verify* the *handler*, we introduce *novel reasoning rules* for *concurrency*:

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \lesssim K_s[()]\ \langle \mathcal{T} \rangle\ \{R\}}{e_i \lesssim K_s[\mathbf{fork}\ e_s]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Fork-R}$$

$$\frac{\texttt{continue}\ k_i\ ()\ \lesssim e_s}{\texttt{continue}\ k_i\ ()\ \lesssim e_s}\ *$$

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\}}{e_i \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \lesssim e_s\ \langle \bot \rangle\ \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}\}}{K_i[e_i] \lesssim e_s'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textit{Logical-Fork}$$

Conclusion

In This Talk.

(Motivation) *Importance* of *relational SL* for program *verification* and *reasoning* (`Fork`).

(Challenge) The *meaning* of an *effect* depends on a *handler*.

(Key Idea) In *baze* (a logic build on top of *Iris*), the *refinement relation* is *parameterised* with a *theory*.

(Compositionality) *baze* allows one to *reason* about effects *independently* of the *handler*.

(Context-Local Reasoning) *baze* enjoys a powerful *context-local* reasoning principle.

(Concurrency) *Refinement* between *handler-based* and *direct* implementations of *concurrency*.
Introduction of *novel rules* in *relational SL* to *reason* about *thread scheduling*.

Conclusion

In This Talk.

(Motivation) *Importance* of *relational SL* for program *verification* and *reasoning* (`Fork`).

(Challenge) The *meaning* of an *effect* depends on a *handler*.

(Key Idea) In *baze* (a logic build on top of *Iris*), the *refinement relation* is *parameterised* with a *theory*.

(Compositionality) *baze* allows one to *reason* about effects *independently* of the *handler*.

(Context-Local Reasoning) *baze* enjoys a powerful *context-local* reasoning principle.

(Concurrency) *Refinement* between *handler-based* and *direct* implementations of *concurrency*.
Introduction of *novel rules* in *relational SL* to *reason* about *thread scheduling*.

In the Paper (A Relational Separation Logic for Effect Handlers).

(Dynamic Effects) *blaze*, a logic for *dynamic effects* built on top of *baze* (a logic for *static effects*).

(Deep vs. Shallow) Support for both *deep* and *shallow handlers*.

(One-Shot vs. Multi-Shot) Support for both *one-shot* and *multi-shot continuations*.

(Case Studies) *Refinement* between *asynchronous-programming* libraries (`Async` & `Await`);
Handler-correctness criteria in *blaze* for *algebraic effects* (*non-determinism*).

Acknowledgements

Thanks to everyone who contributed with comments on early versions of this talk:
Dragana Milovancevic, Raquel Sofia Silva, Simcha van Collem, Shing Hin Ho, Opale Sjöstedt, Carine Morel, Timéo Arnouts, Ines Wright, and Robbert Krebbers.

Thanks also to Amin Timany, who spotted a mistake in slide 15:
the slide incorrectly stated an equivalence ($\Leftrightarrow$) instead of a right-to-left implication ($\Leftarrow$).

Concurrency – Backup

The *complete* set of the *novel reasoning rules* for *concurrency*:

$$\frac{i \Rrightarrow e_s \qquad e_i \precsim e_s \langle \bot \rangle \{True\} \qquad K_i[()] \precsim e_s' \langle \mathcal{T} \rangle \{R\}}{K_i[\mathbf{fork}\ e_i] \precsim e_s' \langle \mathcal{T} \rangle \{R\}}\ \textit{Fork-L}$$

$$\frac{\forall i.\ i \Rrightarrow e_s \twoheadrightarrow e_i \precsim K_s[()] \langle \mathcal{T} \rangle \{R\}}{e_i \precsim K_s[\mathbf{fork}\ e_s] \langle \mathcal{T} \rangle \{R\}}\ \textit{Fork-R}$$

$$\frac{i \Rrightarrow K[e_s] \qquad \forall j\ K'.\ j \Rrightarrow K'[e_s'] \twoheadrightarrow e_i \precsim e_s \langle \bot \rangle \{v_i, \_.\ \exists v_s'.\ j \Rrightarrow K'[e_s'] * R(v_i, v_s')\}}{e_i \precsim e_s' \langle \mathcal{T} \rangle \{R\}}\ \textit{Thread-Swap}$$

$$\frac{i \Rrightarrow K_s[e_s] \qquad e_i \precsim e_s \langle \bot \rangle \{v_i, v_s.\ i \Rrightarrow K_s[v_s] \twoheadrightarrow K_i[v_i] \precsim e_s' \langle \mathcal{T} \rangle \{R\}\}}{K_i[e_i] \precsim e_s' \langle \mathcal{T} \rangle \{R\}}\ \textit{Logical-Fork}$$

Concurrency – Backup

Valid *OCaml 5* implementation:

```
type _ Effect.t += Fork : (unit -> unit) -> unit t

let run main =
  let q = Queue.create () in
  let rec run f =
    match f () with
    | effect (Fork f), k ->
        Queue.push k q;
        run f
    | _ ->
        if not (Queue.empty q) then
          let k = Queue.pop q in continue k ()
  in
  run (fun () -> main (fun f -> perform (Fork f)))
```

Examples of Relational Theories - Backup

State.

$$\mathtt{GET}(\mathtt{perform}\ (\mathtt{Get}\ ()),\ !\mathtt{r},\ R) = \exists x.\ \mathtt{r} \mapsto_{\mathtt{s}}^{1/2} x * (\mathtt{r} \mapsto_{\mathtt{s}}^{1/2} x \twoheadrightarrow R(x, x))$$

$$\mathtt{SET}(\mathtt{perform}\ (\mathtt{Set}\ y),\ \mathtt{r} := y,\ R) = \mathtt{r} \mapsto_{\mathtt{s}}^{1/2} \_ * (\mathtt{r} \mapsto_{\mathtt{s}}^{1/2} y \twoheadrightarrow R(v, v))$$

$$\mathtt{STATE} = \mathtt{GET} \oplus \mathtt{SET}$$

$$\mathtt{r} \mapsto_{\mathtt{s}}^{1/2} x \twoheadrightarrow \mathtt{perform}\ (\mathtt{Get}\ ()) \lesssim\ !\mathtt{r}\ \langle \mathtt{STATE} \rangle\ \{y_i, y_s.\ y_i = y_s = x * \mathtt{r} \mapsto_{\mathtt{s}}^{1/2} x\}$$

$$\mathtt{r} \mapsto_{\mathtt{s}}^{1/2} \_ \twoheadrightarrow \mathtt{perform}\ (\mathtt{Set}\ y) \lesssim \mathtt{r} := y\ \langle \mathtt{STATE} \rangle\ \{\_, \_.\ \mathtt{r} \mapsto_{\mathtt{s}}^{1/2} y\}$$

Non-Determinism (Selected Relations).

$$\mathtt{ASSOC}_1(e_{11}\ \mathbf{or}\ (e_{12}\ \mathbf{or}\ e_{13}),\ (e_{21}\ \mathbf{or}\ e_{22})\ \mathbf{or}\ e_{23},\ R) = \square R(e_{11}, e_{21}) * \square R(e_{12}, e_{22}) * \square R(e_{13}, e_{23})$$

$$\mathtt{UNIT}_1(e_1\ \mathbf{or}\ \mathbf{fail},\ e_2,\ R) = \square R(e_1, e_2)$$

$$\mathtt{ND} = \mathtt{ASSOC}_1 \oplus \mathtt{UNIT}_1 \oplus \ldots$$