Relational reasoning using concurrent separation logic

Robbert Krebbers[1]

Delft University of Technology, The Netherlands

January 20, 2020 @ ADSL, New Orleans, USA

[1] This is joint work with Dan Frumin (Radboud University) and Lars Birkedal (Aarhus University)

Why prove relational properties of programs?

- Specifying programs

$$\texttt{implementation} \precsim_{ctx} \texttt{specification}$$

Why prove relational properties of programs?

- Specifying programs

$$\texttt{implementation} \precsim_{ctx} \texttt{specification}$$

- Optimized versions of data structures

$$\texttt{hash\_table} \precsim_{ctx} \texttt{assoc\_list}$$

Why prove relational properties of programs?

- Specifying programs

$$\texttt{implementation} \precsim_{ctx} \texttt{specification}$$

- Optimized versions of data structures

$$\texttt{hash\_table} \precsim_{ctx} \texttt{assoc\_list}$$

- Proving program transformations

$$\forall e_{\mathsf{source}}.\ \texttt{compile}(e_{\mathsf{source}}) \precsim_{ctx} e_{\mathsf{source}}$$

Language features that complicate refinements

- Mutable state

$$\left(\texttt{let}\, x = f()\, \texttt{in}\, (x, x)\right) \not\precsim_{ctx} \left(f(), f()\right)$$

Language features that complicate refinements

- Mutable state

$$\left(\texttt{let}\, x = f()\,\texttt{in}\,(x, x)\right) \not\precsim_{ctx} \left(f(), f()\right)$$

- Higher-order functions

$$\left(\lambda().\,1\right) \not\precsim_{ctx} \left(\texttt{let}\, x = \texttt{ref}(0)\,\texttt{in}\,(\lambda().\,x \leftarrow (1 + !\,x); !\,x)\right)$$

Language features that complicate refinements

- Mutable state

$$\big(\texttt{let}\, x = f()\, \texttt{in}\, (x, x)\big) \not\precsim_{ctx} \big(f(), f()\big)$$

- Higher-order functions

$$\big(\lambda().\, 1\big) \not\precsim_{ctx} \big(\texttt{let}\, x = \texttt{ref}(0)\, \texttt{in}\, (\lambda().\, x \leftarrow (1 + !\, x); !\, x)\big)$$

- Concurrency

$$\big(x \leftarrow 10; x \leftarrow 11\big) \not\precsim_{ctx} \big(x \leftarrow 11\big)$$

What do such relational properties mean mathematically?

Contextual refinement

Contextual refinement: the "gold standard" of program refinement:

$$e_1 \precsim_{ctx} e_2 : \tau \triangleq \forall (\mathcal{C} : \tau \to \mathtt{int}).\, \forall v.\ \mathcal{C}[e_1] \downarrow v \implies \mathcal{C}[e_2] \downarrow v$$

"Any behavior of a (well-typed) client $\mathcal{C}$ using e_1 can be matched by a behavior of the same client using e_2"

Contextual refinement

Contextual refinement: the "gold standard" of program refinement:

$$e_1 \precsim_{ctx} e_2 : \tau \triangleq \forall(\mathcal{C} : \tau \to \texttt{int}).\, \forall v.\ \mathcal{C}[e_1] \downarrow v \implies \mathcal{C}[e_2] \downarrow v$$

"Any behavior of a (well-typed) client $\mathcal{C}$ using e_1 can be matched by a behavior of the same client using e_2"

Very hard to prove: Quantification over all clients

Logical relations to the rescue!

Do not prove contextual refinement directly, but use a **binary logical relation**:

$$e_1 \precsim e_2 : \tau$$

- $e_1 \precsim e_2 : \tau$ is defined structurally on the type τ
- Does not involve quantification over all clients $\mathcal{C}$
- Soundness $e_1 \precsim e_2 : \tau \implies e_1 \precsim_{ctx} e_2 : \tau$ proved once and for all

A bit of history

Logical relations $e_1 \precsim e_2 : \tau$ are notoriously hard to define when having recursive types, higher-order state (type-world circularity), . . .

A bit of history

Logical relations $e_1 \precsim e_2 : \tau$ are notoriously hard to define when having recursive types, higher-order state (type-world circularity), ...

Solutions to define such logical relations:

A bit of history

Logical relations $e_1 \precsim e_2 : \tau$ are notoriously hard to define when having recursive types, higher-order state (type-world circularity), ...

Solutions to define such logical relations:

- **Step-indexing** (Appel-McAllester, Ahmed, ...)

  Solve circularities by stratifying everything by a natural number corresponding to the number of computation steps

A bit of history

Logical relations $e_1 \precsim e_2 : \tau$ are notoriously hard to define when having recursive types, higher-order state (type-world circularity), ...

Solutions to define such logical relations:

- **Step-indexing** (Appel-McAllester, Ahmed, ...)

  Solve circularities by stratifying everything by a natural number corresponding to the number of computation steps
- **Logical approach** (LSLR, LADR, CaReSL, Iris, ...)

  Hide step-indexing using modalities to obtain clearer definitions and proofs

A bit of history

Logical relations $e_1 \precsim e_2 : \tau$ are notoriously hard to define when having recursive types, higher-order state (type-world circularity), ...

Solutions to define such logical relations:

- **Step-indexing** (Appel-McAllester, Ahmed, ...)

  Solve circularities by stratifying everything by a natural number corresponding to the number of computation steps

- **Logical approach** (LSLR, LADR, CaReSL, Iris, ...)

  Hide step-indexing using modalities to obtain clearer definitions and proofs

We tried to take this one step further

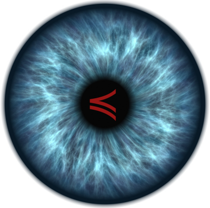
Prove program refinements using inference rules à la concurrent separation logic

Instead of Hoare triples $\{P\}\, e\, \{Q\}$ we have refinement judgments $e_1 \precsim e_2 : \tau$

- Refinement proofs by symbolic execution as we know from separation logic
- Modular and conditional specifications
- Modeled using the "logical approach"

ReLoC [Frumin, Krebbers, Birkedal; LICS'18]

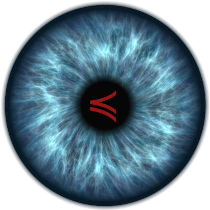
ReLoC: mechanized separation logic for interactive refinement proofs of fine-grained concurrent programs



ReLoC [Frumin, Krebbers, Birkedal; LICS'18]

ReLoC: mechanized separation logic for interactive refinement proofs of fine-grained concurrent programs

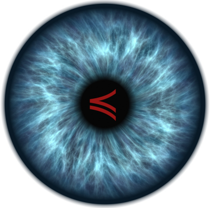
- **Fine-grained concurrency**: programs use low-level synchronization primitives for more granular parallelism



ReLoC [Frumin, Krebbers, Birkedal; LICS'18]

ReLoC: mechanized separation logic for interactive refinement proofs of fine-grained concurrent programs

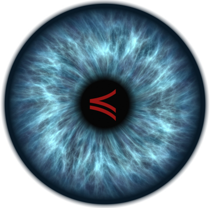
- **Fine-grained concurrency**: programs use low-level synchronization primitives for more granular parallelism
- **Mechanized**: soundness proven sound using the Iris framework in Coq



ReLoC [Frumin, Krebbers, Birkedal; LICS'18]

ReLoC: mechanized separation logic for interactive refinement proofs of fine-grained concurrent programs

- **Fine-grained concurrency**: programs use low-level synchronization primitives for more granular parallelism
- **Mechanized**: soundness proven sound using the Iris framework in Coq
- **Interactive refinement proofs**: using high-level tactics in Coq



ReLoC: (simplified) grammar

$$P, Q \in \mathsf{Prop} ::= \forall x.\, P \mid \exists x.\, P \mid P \vee Q \mid \ \ldots$$

ReLoC: (simplified) grammar

$$\begin{aligned} P, Q \in \mathrm{Prop} ::= {} & \forall x.\, P \mid \exists x.\, P \mid P \vee Q \mid \ldots \\ \mid {} & \boxed{P * Q} \mid \boxed{P \mathrel{-\!\!*} Q} \mid \boxed{\ell \mapsto_{\mathsf{i}} v} \mid \boxed{\ell \mapsto_{\mathsf{s}} v} \end{aligned}$$

Separation logic for handling mutable state

- $\ell \mapsto_{\mathsf{i}} v$ for the left-hand side (implementation)
- $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side (specification)

ReLoC: (simplified) grammar

$$P, Q \in \mathrm{Prop} ::= \forall x.\, P \mid \exists x.\, P \mid P \vee Q \mid \ldots$$
$$\mid \; P * Q \;\mid\; P \mathrel{-\!\!*} Q \;\mid\; \ell \mapsto_{\mathsf{i}} v \;\mid\; \ell \mapsto_{\mathsf{s}} v$$
$$\mid \; (\Delta \mid\!\models e_1 \precsim e_2 : \tau) \;\mid\; [\![\tau]\!]_\Delta(v_1, v_2) \;\mid\; \ldots$$

Separation logic for handling mutable state

- $\ell \mapsto_{\mathsf{i}} v$ for the left-hand side (implementation)
- $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side (specification)

Logic with first-class refinement propositions to allow conditional refinements

- $(\ell_1 \mapsto_{\mathsf{i}} v) \;\mathrel{-\!\!*}\; (e_1 \precsim e_2 : \tau)$
- $(e_1 \precsim e_2 : \mathtt{unit} \to \tau) \;\mathrel{-\!\!*}\; (f(e_1) \precsim e_2(); e_2() : \tau)$

Proving refinements of pure programs

Some rules for pure programs

Symbolic execution rules

$$\frac{\Delta \mathrel{|}\!\models K[\,e_1'\,] \precsim e_2 : \tau \qquad e_1 \rightarrow_{\text{pure}} e_1'}{\Delta \mathrel{|}\!\models K[\,e_1\,] \precsim e_2 : \tau}\ast$$

Some rules for pure programs

Symbolic execution rules

$$\frac{\Delta \mathrel{|}\!\models K[\,e_1'\,] \precsim e_2 : \tau \qquad e_1 \to_{\mathsf{pure}} e_1'}{\Delta \mathrel{|}\!\models K[\,e_1\,] \precsim e_2 : \tau}*$$

$$\frac{\Delta \mathrel{|}\!\models e_1 \precsim K[\,e_2'\,] : \tau \qquad e_2 \to_{\mathsf{pure}} e_2'}{\Delta \mathrel{|}\!\models e_1 \precsim K[\,e_2\,] : \tau}*$$

Some rules for pure programs

Structural rules

$$\frac{\Delta \mid\models e_1 \precsim e_2 : \tau \qquad * \qquad \Delta \mid\models e_1' \precsim e_2' : \tau'}{\Delta \mid\models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \tau'} *$$

Some rules for pure programs

Structural rules

$$\frac{\Delta \mid\models e_1 \precsim e_2 : \tau \qquad * \qquad \Delta \mid\models e_1' \precsim e_2' : \tau'}{\Delta \mid\models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \tau'}*$$

$$\frac{\exists (R : \mathit{Val} \times \mathit{Val} \to \mathrm{Prop}). \quad [\alpha := R], \Delta \mid\models e_1 \precsim e_2 : \tau}{\Delta \mid\models \texttt{pack}(e_1) \precsim \texttt{pack}(e_2) : \exists \alpha. \tau}*$$

Some rules for pure programs

Structural rules

$$\frac{\Delta \mid\models e_1 \precsim e_2 : \tau \quad * \quad \Delta \mid\models e_1' \precsim e_2' : \tau'}{\Delta \mid\models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \tau'}*$$

$$\frac{\exists (R : \mathit{Val} \times \mathit{Val} \to \mathsf{Prop}).\quad [\alpha := R], \Delta \mid\models e_1 \precsim e_2 : \tau}{\Delta \mid\models \mathtt{pack}(e_1) \precsim \mathtt{pack}(e_2) : \exists \alpha.\, \tau}*$$

$$\frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \mid\models v_1 \precsim v_2 : \tau}* \qquad \frac{\Box \begin{pmatrix} \forall v_1\, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \twoheadrightarrow \\ \qquad \Delta \mid\models e_1[v_1/x_1] \precsim e_2[v_2/x_2] : \sigma \end{pmatrix}}{\Delta \mid\models \lambda x_1.\, e_1 \precsim \lambda x_2.\, e_2 : \tau \to \sigma}*$$

Example

A bit interface:

$$\texttt{bitT} \triangleq \exists\alpha.\ \boxed{\alpha} \times \boxed{(\alpha \to \alpha)} \times \boxed{(\alpha \to \texttt{bool})}$$

- constructor
- flip the bit
- view the bit as a Boolean

Example

A bit interface:

$$\texttt{bitT} \triangleq \exists\alpha.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \texttt{bool})$$

- constructor
- flip the bit
- view the bit as a Boolean

Two implementations:

$$\texttt{bit\_bool} \triangleq \texttt{pack}\Big(\texttt{true},\ (\lambda b.\, \neg b),\ (\lambda b.\, b)\Big)$$

$$\texttt{bit\_nat} \triangleq \texttt{pack}\Big(1,\ (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0),\ (\lambda n.\, n = 1)\Big)$$

Example

A bit interface:

$$\mathtt{bitT} \triangleq \exists \alpha.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathtt{bool})$$

- constructor
- flip the bit
- view the bit as a Boolean

Two implementations:

$$\mathtt{bit\_bool} \triangleq \mathtt{pack}\Big(\mathtt{true},\ (\lambda b.\, \neg b),\ (\lambda b.\, b)\Big)$$

$$\mathtt{bit\_nat} \triangleq \mathtt{pack}\Big(1,\ (\lambda n.\, \mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0),\ (\lambda n.\, n = 1)\Big)$$

Refinement (and vice versa):

$$\mathtt{bit\_bool} \precsim \mathtt{bit\_nat} : \mathtt{bitT}$$

Proof of the refinement

$$\texttt{pack}\Big(\texttt{true},\ (\lambda b.\, \neg b),\ (\lambda b.\, b)\Big)$$

$$\precsim$$

$$\texttt{pack}\Big(1,\ (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0),\ (\lambda n.\, n = 1)\Big)$$

$$:$$

$$\exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to \texttt{bool})$$

Proof of the refinement

$$\texttt{pack}\Big(\texttt{true},\ (\lambda b.\,\neg b),\ (\lambda b.\,b)\Big)$$

$$\precsim$$

$$\texttt{pack}\Big(1,\ (\lambda n.\,\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0),\ (\lambda n.\,n = 1)\Big)$$

$$:$$

$$\exists \alpha.\,\alpha \times (\alpha \to \alpha) \times (\alpha \to \texttt{bool})$$

$\boxed{\text{Need to come with with an } R : \mathit{Val} \times \mathit{Val} \to \text{Prop}}$

Proof of the refinement

where $R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$

$$\texttt{pack}\Big(\texttt{true},\ (\lambda b.\, \neg b),\ (\lambda b.\, b)\Big)$$

$$\precsim$$

$$\texttt{pack}\Big(1,\ (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0),\ (\lambda n.\, n = 1)\Big)$$

$$:$$

$$\exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to \texttt{bool})$$

Need to come with with an $R : \mathit{Val} \times \mathit{Val} \to \mathrm{Prop}$

Proof of the refinement

$$[\alpha := R] \models \qquad \text{where } R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$$

$$\Big(\texttt{true},\ (\lambda b.\, \neg b),\ (\lambda b.\, b)\Big)$$

$$\precsim$$

$$\Big(1,\ (\lambda n.\, \texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } 0),\ (\lambda n.\, n = 1)\Big)$$

$$:$$

$$\alpha \times (\alpha \to \alpha) \times (\alpha \to \texttt{bool})$$

Proof of the refinement

$[\alpha := R] \models$ where $R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$

$$\Big(\texttt{true},\ (\lambda b.\, \neg b),\ (\lambda b.\, b)\Big)$$

$$\precsim$$

$$\Big(1,\ (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0),\ (\lambda n.\, n = 1)\Big)$$

$$:$$

$$\alpha \times (\alpha \to \alpha) \times (\alpha \to \texttt{bool})$$

Use structural rule for products

Proof of the refinement

$$[\alpha := R] \models \qquad \text{where } R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$$

$$\texttt{true} \precsim 1 \qquad : \alpha$$

$$(\lambda b.\, \neg b) \precsim (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0) \qquad : \alpha \to \alpha$$

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \qquad : \alpha \to \texttt{bool}$$

Proof of the refinement

$$[\alpha := R] \models \qquad \text{where } R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$$

$$\texttt{true} \precsim 1 \qquad : \alpha$$

$$(\lambda b.\, \neg b) \precsim (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0) \qquad : \alpha \to \alpha$$

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \qquad : \alpha \to \texttt{bool}$$

Proof of the refinement

$[\alpha := R] \models$ where $R \triangleq \{(\mathtt{true}, 1), (\mathtt{false}, 0)\}$

$$\mathtt{true} \precsim 1 \qquad : \alpha \qquad \text{(by def of } R\text{)}$$

$$(\lambda b.\, \neg b) \precsim (\lambda n.\, \mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \qquad : \alpha \to \alpha$$

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \qquad : \alpha \to \mathtt{bool}$$

Proof of the refinement

$[\alpha := R] \models \qquad$ where $R \triangleq \{(\mathtt{true}, 1), (\mathtt{false}, 0)\}$

$$\mathtt{true} \precsim 1 \quad : \alpha \qquad \text{(by def of } R\text{)}$$

$$(\lambda b.\, \neg b) \precsim (\lambda n.\, \mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \quad : \alpha \to \alpha \qquad (\lambda\text{-rule + symb. exec.})$$

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathtt{bool}$$

After using the λ rule and case analysis on R:

$$\neg\mathtt{true} \precsim \mathtt{if}\ 1 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \quad : \alpha$$

$$\neg\mathtt{false} \precsim \mathtt{if}\ 0 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \quad : \alpha$$

Proof of the refinement

$$[\alpha := R] \models \qquad \text{where } R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$$

$$\texttt{true} \precsim 1 \qquad : \alpha \qquad \text{(by def of } R\text{)}$$

$$(\lambda b.\, \neg b) \precsim (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0) \qquad : \alpha \to \alpha \qquad (\lambda\text{-rule + symb. exec.})$$

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \qquad : \alpha \to \texttt{bool} \qquad (\lambda\text{-rule + symb. exec.})$$

Proof of the refinement

$[\alpha := R] \models$ where $R \triangleq \{(\texttt{true}, 1), (\texttt{false}, 0)\}$

$$\texttt{true} \precsim 1 \quad : \alpha \qquad \text{(by def of } R\text{)}$$

$$(\lambda b.\, \neg b) \precsim (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ 0) \quad : \alpha \to \alpha \qquad (\lambda\text{-rule} + \text{symb. exec.})$$

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \texttt{bool} \qquad (\lambda\text{-rule} + \text{symb. exec.})$$

$\square$

Reasoning about mutable state

Separation logic to the rescue!

"Vanilla" separation logic [O'Hearn, Reynolds, Yang; CSL'01]

Propositions P, Q denote ownership of resources

Points-to connective $\ell \mapsto v$:
Exclusive ownership of location ℓ with value v

Separating conjunction $P * Q$:
The resources consists of separate parts satisfying P and Q

Basic example:

$$\{\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2\}\texttt{swap}(\ell_1, \ell_2)\{\ell_1 \mapsto v_2 * \ell_2 \mapsto v_1\}$$

the $*$ ensures that ℓ_1 and ℓ_2 are different memory locations

Mutable state and separation logic for refinements

There are two versions of the **points-to connective:**

- $\ell \mapsto_{\mathsf{i}} v$ for the left-hand side/implementation
- $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side/specification

Mutable state and separation logic for refinements

There are two versions of the **points-to connective:**

- $\ell \mapsto_{\mathsf{i}} v$ for the left-hand side/implementation
- $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side/specification

Example:

$$\ell_1 \mapsto_{\mathsf{i}} 4 \quad -\!\!\ast \quad \ell_2 \mapsto_{\mathsf{s}} 0 \quad -\!\!\ast \quad (!\,\ell_1) \precsim (\ell_2 \leftarrow 4; !\,\ell_2) : \mathtt{int}$$

Some rules for mutable state

Symbolic execution

$$\frac{\ell_1 \mapsto_{\mathsf{i}} - \qquad * \qquad (\ell_1 \mapsto_{\mathsf{i}} v_1 \ \mathrel{-\!\!*}\ \Delta \mathrel{\|\!\!\models} K[\,()\,] \precsim e_2 : \tau)}{\Delta \mathrel{\|\!\!\models} K[\,\ell_1 \leftarrow v_1\,] \precsim e_2 : \tau}\,*$$

Some rules for mutable state

Symbolic execution

$$\frac{\ell_1 \mapsto_{\mathsf{i}} - \qquad * \qquad (\ell_1 \mapsto_{\mathsf{i}} v_1 \;\mathrel{-\!\!*}\; \Delta \Vdash K[\,()\,] \precsim e_2 : \tau)}{\Delta \Vdash K[\,\ell_1 \leftarrow v_1\,] \precsim e_2 : \tau}*$$

$$\frac{\ell_2 \mapsto_{\mathsf{s}} - \qquad * \qquad (\ell_2 \mapsto_{\mathsf{s}} v_2 \;\mathrel{-\!\!*}\; \Delta \Vdash e_1 \precsim K[\,()\,] : \tau)}{\Delta \Vdash e_1 \precsim K[\,\ell_2 \leftarrow v_2\,] : \tau}*$$

Some rules for mutable state

Symbolic execution

$$\frac{\ell_1 \mapsto_{\mathsf{i}} - \qquad * \qquad (\ell_1 \mapsto_{\mathsf{i}} v_1 \;\twoheadrightarrow\; \Delta \mathrel{|}\!\models K[\,()\,] \precsim e_2 : \tau)}{\Delta \mathrel{|}\!\models K[\,\ell_1 \leftarrow v_1\,] \precsim e_2 : \tau}*$$

$$\frac{\ell_2 \mapsto_{\mathsf{s}} - \qquad * \qquad (\ell_2 \mapsto_{\mathsf{s}} v_2 \;\twoheadrightarrow\; \Delta \mathrel{|}\!\models e_1 \precsim K[\,()\,] : \tau)}{\Delta \mathrel{|}\!\models e_1 \precsim K[\,\ell_2 \leftarrow v_2\,] : \tau}*$$

$$\frac{\forall \ell_1.\, \ell_1 \mapsto_{\mathsf{i}} v_1 \;\twoheadrightarrow\; \Delta \mathrel{|}\!\models K[\,\ell_1\,] \precsim e_2 : \tau}{\Delta \mathrel{|}\!\models K[\,\texttt{ref}(v_1)\,] \precsim e_2 : \tau}* \qquad \frac{\forall \ell_2.\, \ell_2 \mapsto_{\mathsf{s}} v_2 \;\twoheadrightarrow\; \Delta \mathrel{|}\!\models e \precsim K[\,\ell_2\,] : \tau}{\Delta \mathrel{|}\!\models e_1 \precsim K[\,\texttt{ref}(v_2)\,] : \tau}*$$

Reasoning about higher-order functions and concurrency

State encapsulation

Modules with **encapsulated state**:

$$\texttt{let}\, x = \texttt{ref}(e)\, \texttt{in} \left(\lambda y.\ \ldots\right)$$

The reference can only be used in the closure

State encapsulation

Modules with **encapsulated state**:

$$\texttt{let}\, x = \texttt{ref}(e)\, \texttt{in}\, \Big(\lambda y.\ \ldots\Big)$$

The reference can only be used in the closure

Simple example:

$$\texttt{counter} \triangleq \Big(\lambda().\, \texttt{let}\, x = \texttt{ref}(1)\, \texttt{in}\, (\lambda().\, \texttt{FAA}(x, 1))\Big) : \texttt{unit} \to (\texttt{unit} \to \texttt{int})$$

- $\texttt{counter}()$ constructs an instance $c : \texttt{unit} \to \texttt{int}$ of the counter module
- Calling $c()$ in subsequently gives $1, 2, \ldots$
- The reference x is private to the module

The problem

Modules with **encapsulated state**:

$$\texttt{let}\; x = \texttt{ref}(e)\;\texttt{in}\; \underbrace{\Big(\lambda y.\; \ldots\Big)}_{f}$$

The problem

Modules with **encapsulated state**:

$$\texttt{let}\, x = \texttt{ref}(e)\, \texttt{in}\, \underbrace{\left(\lambda y.\ \ldots\right)}_{f}$$

Reasoning about such modules is challenging:

- f can be called multiple times by clients

  So, the value of x can change in each call

The problem

Modules with **encapsulated state**:

$$\texttt{let}\, x = \texttt{ref}(e)\, \texttt{in}\, \underbrace{\left(\lambda y.\, \ldots\right)}_{f}$$

Reasoning about such modules is challenging:

- f can be called multiple times by clients

  So, the value of x can change in each call

- f can even be called even in parallel!

  So, f cannot get exclusive access to $x \mapsto v$

The problem

Modules with **encapsulated state**:

$$\texttt{let}\, x = \texttt{ref}(e)\,\texttt{in}\, \underbrace{\left(\lambda y.\, \ldots\right)}_{f}$$

Reasoning about such modules is challenging:

- f can be called multiple times by clients

  So, the value of x can change in each call

- f can even be called even in parallel!

  So, f cannot get exclusive access to $x \mapsto v$

We need to guarantee that closures do not get access to exclusive resources

Persistent resources

The "persistent" modality $\square$ in Iris/ReLoC:

$$\square\, P \triangleq \text{“}P \text{ holds } \textbf{without} \text{ assuming exclusive resources”}$$

Examples:

- Equality is persistent: $(x = y) \vdash \square(x = y)$
- Points-to connectives are not: $((\ell \mapsto v) \nvdash \square(\ell \mapsto v)$
- More examples later...

ReLoC's λ-rule again

The $\square$ modality makes sure no exclusive resources can escape into closures:

$$\frac{\square \begin{pmatrix} \forall v_1\, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \twoheadrightarrow\ast \\ \qquad \Delta \mid\models e_1[v_1/x_1] \precsim e_2[v_2/x_2] : \sigma \end{pmatrix}}{\Delta \mid\models \lambda x_1.\, e_1 \precsim \lambda x_2.\, e_2 : \tau \to \sigma} \ast$$

ReLoC's λ-rule again

The $\square$ modality makes sure no exclusive resources can escape into closures:

$$\frac{\square \begin{pmatrix} \forall v_1\, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \twoheadrightarrow \\ \qquad \Delta \mid\models e_1[v_1/x_1] \precsim e_2[v_2/x_2] : \sigma \end{pmatrix}}{\Delta \mid\models \lambda x_1.\, e_1 \precsim \lambda x_2.\, e_2 : \tau \to \sigma} *$$

Prohibits "wrong" refinements, for example:

$$\Big(\lambda().\, 1\Big) \not\precsim_{ctx} \Big(\texttt{let}\, x = \texttt{ref}(0)\, \texttt{in}\, (\lambda().\, x \leftarrow (1 + !\, x);\, !\, x)\Big)$$

Due to $\square$, the resource $x \mapsto_s 0$ cannot be used to prove the closure

But it should be possible to use resources in closures

For example:

$$\Big(\lambda().\,\texttt{let}\,x = \texttt{ref}(1)\,\texttt{in}\,(\lambda().\,\texttt{FAA}(x,1))\Big)$$

$$\precsim$$

$$\left(\begin{array}{l} \lambda().\,\texttt{let}\,x = \texttt{ref}(1), l = \text{newlock}\ ()\ \texttt{in} \\ \qquad \lambda().\,\text{acquire}(l); \\ \qquad\qquad \texttt{let}\,v = !\,x\,\texttt{in} \\ \qquad\qquad x \leftarrow v + 1; \\ \qquad\qquad \text{release}(l);\, v \end{array}\right)$$

Iris-style Invariants

The invariant connective $\boxed{R}$
expresses that R is maintained as an invariant on the state

Iris-style Invariants

The invariant connective $\boxed{R}$
expresses that R is maintained as an invariant on the state

Invariants allow to share resources:

- A resource R can be turned into $\boxed{R}$ at any time
- Invariants are persistent: $\boxed{R} \vdash \Box \boxed{R}$
- ...thus can be used to prove closures

Iris-style Invariants

The invariant connective $\boxed{R}$
expresses that R is maintained as an invariant on the state

Invariants allow to share resources:

- A resource R can be turned into $\boxed{R}$ at any time
- Invariants are persistent: $\boxed{R} \vdash \square \boxed{R}$
- ...thus can be used to prove closures

But that comes with a cost:

- Invariants $\boxed{R}$ can only be accessed during atomic steps on the left-hand side
- ...while multiple steps on the right-hand side can be performed

Example

$$\texttt{let}\ x = \texttt{ref}(1)\ \texttt{in}\ (\lambda().\ \texttt{FAA}(x, 1))$$

$$\precsim$$

$$\begin{aligned}&\texttt{let}\ x = \texttt{ref}(1)\ , l = \text{newlock}\ ()\ \texttt{in}\\ &\quad (\lambda().\ \text{acquire}(l);\\ &\qquad \texttt{let}\ v = !\,x\ \texttt{in}\\ &\qquad x \leftarrow v + 1;\\ &\qquad \text{release}(l);\ v)\end{aligned}$$

Example

$$\texttt{let}\, x = \boxed{\texttt{ref}(1)}\ \texttt{in}\, (\lambda().\, \texttt{FAA}(x, 1))$$

$$\precsim$$

$$\begin{aligned}\texttt{let}\, x = \texttt{ref}(1)\,, l = \text{newlock}\ ()\ \texttt{in}\\ (\lambda().\, \text{acquire}(l);\\ \texttt{let}\, v = !\, x\ \texttt{in}\\ x \leftarrow v + 1;\\ \text{release}(l); v)\end{aligned}$$

Example

$$\overline{\mathtt{x}_1 \mapsto_{\mathsf{i}} 1}$$

$$(\lambda ().\, \mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{array}{l}
\mathtt{let}\, x = \mathtt{ref}(1)\, , l = \text{newlock}\ ()\ \mathtt{in} \\
\quad (\lambda ().\, \text{acquire}(l); \\
\qquad \mathtt{let}\, v = !\, x\ \mathtt{in} \\
\qquad x \leftarrow v + 1; \\
\qquad \text{release}(l); v)
\end{array}$$

Example

$\mathtt{x}_1 \mapsto_{\mathsf{i}} 1$

$$(\lambda().\,\mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{aligned}
&\mathtt{let}\; x = \boxed{\mathtt{ref}(1)},\; l = \mathsf{newlock}\;()\;\mathtt{in}\\
&\quad (\lambda().\,\mathsf{acquire}(l);\\
&\qquad \mathtt{let}\; v = !\,x\;\mathtt{in}\\
&\qquad x \leftarrow v + 1;\\
&\qquad \mathsf{release}(l);\, v)
\end{aligned}$$

Example

$$\mathtt{x}_1 \mapsto_{\mathsf{i}} 1$$

$$\mathtt{x}_2 \mapsto_{\mathsf{s}} 1$$

$$(\lambda().\,\mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{aligned}
&\mathtt{let}\ l = \mathsf{newlock}\ ()\ \mathtt{in}\\
&(\lambda().\,\mathsf{acquire}(l);\\
&\quad \mathtt{let}\ v = !\,\mathtt{x}_2\ \mathtt{in}\\
&\quad \mathtt{x}_2 \leftarrow v + 1;\\
&\quad \mathsf{release}(l);\, v)
\end{aligned}$$

Example

$\mathtt{x}_1 \mapsto_{\mathsf{i}} 1$

$\mathtt{x}_2 \mapsto_{\mathsf{s}} 1$

$$(\lambda().\,\mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{array}{l}
\mathtt{let}\ l = \mathsf{newlock}\ ()\ \mathtt{in} \\
(\lambda().\,\mathsf{acquire}(l); \\
\quad \mathtt{let}\ v = !\,\mathtt{x}_2\ \mathtt{in} \\
\quad \mathtt{x}_2 \leftarrow v + 1; \\
\quad \mathsf{release}(l);\ v)
\end{array}$$

Example

$\mathtt{x}_1 \mapsto_{\mathsf{i}} 1$

$\mathtt{x}_2 \mapsto_{\mathsf{s}} 1$

$\mathsf{isLock}(l, \mathtt{unlocked})$

$$(\lambda().\, \mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{aligned}(\lambda().\, &\mathsf{acquire}(l);\\ &\mathtt{let}\ v = !\,\mathtt{x}_2\ \mathtt{in}\\ &\mathtt{x}_2 \leftarrow v + 1;\\ &\mathsf{release}(l);\, v)\end{aligned}$$

Example

$$\frac{}{\exists n.\; \mathtt{x}_1 \mapsto_{\mathsf{i}} n \quad \mathtt{x}_2 \mapsto_{\mathsf{s}} n \quad \mathsf{isLock}(l, \mathtt{unlocked})}$$

$$(\lambda().\,\mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{aligned}(\lambda().\,& \mathsf{acquire}(l);\\ & \mathtt{let}\ v = !\,\mathtt{x}_2\ \mathtt{in}\\ & \mathtt{x}_2 \leftarrow v + 1;\\ & \mathsf{release}(l);\, v)\end{aligned}$$

Example

$$\boxed{\begin{array}{l} \exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \ast \\ \quad \mathtt{x}_2 \mapsto_{\mathsf{s}} n \ast \\ \quad \mathsf{isLock}(l, \mathtt{unlocked}) \end{array}}$$

$$(\lambda().\, \mathtt{FAA}(x_1, 1))$$

$$\precsim$$

$$\begin{array}{l} (\lambda().\, \mathsf{acquire}(l); \\ \quad \mathtt{let}\ v = \,!\,\mathtt{x}_2\ \mathtt{in} \\ \quad \mathtt{x}_2 \leftarrow v + 1; \\ \quad \mathsf{release}(l);\, v) \end{array}$$

Example

$$\boxed{\begin{aligned}\exists n.\ & \mathtt{x}_1 \mapsto_{\mathsf{i}} n\ * \\ & \mathtt{x}_2 \mapsto_{\mathsf{s}} n\ * \\ & \mathsf{isLock}(l, \mathtt{unlocked})\end{aligned}}$$

$$\mathtt{FAA}(x_1, 1)$$

$$\precsim$$

$$\begin{aligned}&\mathsf{acquire}(l);\\ &\mathtt{let}\ v = \,!\,\mathtt{x}_2\ \mathtt{in}\\ &\mathtt{x}_2 \leftarrow v + 1;\\ &\mathsf{release}(l);\ \ v\end{aligned}$$

Example

$$\boxed{\begin{aligned}\exists n.\ & \mathtt{x}_1 \mapsto_{\mathsf{i}} n \ * \\ & \mathtt{x}_2 \mapsto_{\mathsf{s}} n \ * \\ & \mathsf{isLock}(l, \mathtt{unlocked})\end{aligned}}$$

$\mathtt{FAA}(x_1, 1)$

$\precsim$

$\mathsf{acquire}(l);$

$\mathtt{let}\ v = \ !\,\mathtt{x}_2\ \mathtt{in}$

$\mathtt{x}_2 \leftarrow v + 1;$

$\mathsf{release}(l);\quad v$

Example

$$\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \ast$$
$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n \ast$$
$$\mathrm{isLock}(l, \mathtt{unlocked})$$

$$\mathtt{x}_1 \mapsto_{\mathsf{i}} n$$
$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n$$
$$\mathrm{isLock}(l, \mathtt{unlocked})$$

$$\mathtt{FAA}(x_1, 1)$$

$$\precsim$$

$$\mathrm{acquire}(l);$$
$$\mathtt{let}\; v = \; !\,\mathtt{x}_2 \;\mathtt{in}$$
$$\mathtt{x}_2 \leftarrow v + 1;$$
$$\mathrm{release}(l);\quad v$$

Example

$$\boxed{\begin{array}{l}\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \,*\\ \quad \mathtt{x}_2 \mapsto_{\mathsf{s}} n \,*\\ \quad \mathsf{isLock}(l, \mathtt{unlocked})\end{array}}$$

$$\boxed{n}$$

$$\precsim$$

$$\mathtt{x}_1 \mapsto_{\mathsf{i}} n + 1$$

$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n$$

$$\mathsf{isLock}(l, \mathtt{unlocked})$$

$\mathsf{acquire}(l);$

$\mathtt{let}\ v = \ !\,\mathtt{x}_2\ \mathtt{in}$

$\mathtt{x}_2 \leftarrow v + 1;$

$\mathsf{release}(l);\ \ v$

Example

$$\boxed{\begin{array}{l}\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \,* \\ \quad \mathtt{x}_2 \mapsto_{\mathsf{s}} n \,* \\ \quad \mathsf{isLock}(l, \mathtt{unlocked})\end{array}}$$

$\mathtt{x}_1 \mapsto_{\mathsf{i}} n + 1$

$\mathtt{x}_2 \mapsto_{\mathsf{s}} n$

$\mathsf{isLock}(l, \mathtt{unlocked})$

n

$\precsim$

$\mathsf{acquire}(l);$

$\mathtt{let}\ v = \ !\,\mathtt{x}_2\ \mathtt{in}$

$\mathtt{x}_2 \leftarrow v + 1;$

$\mathsf{release}(l);\ \ v$

Example

$$\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \ast$$
$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n \ast$$
$$\mathsf{isLock}(l, \mathtt{unlocked})$$

$$\boxed{n}$$

$$\precsim$$

$$\mathtt{x}_1 \mapsto_{\mathsf{i}} n + 1$$
$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n$$
$$\mathsf{isLock}(l, \mathtt{locked})$$

$$\mathtt{let}\ v = \ !\,\mathtt{x}_2\ \mathtt{in}$$
$$\mathtt{x}_2 \leftarrow v + 1;$$
$$\mathsf{release}(l);\ \ v$$

Example

$$\boxed{\begin{aligned}\exists n.\, \mathtt{x}_1 &\mapsto_{\mathsf{i}} n \,*\\ \mathtt{x}_2 &\mapsto_{\mathsf{s}} n \,*\\ &\mathsf{isLock}(l, \mathtt{unlocked})\end{aligned}}$$

$\mathtt{x}_1 \mapsto_{\mathsf{i}} n + 1$

$\mathtt{x}_2 \mapsto_{\mathsf{s}} n$

$\mathsf{isLock}(l, \mathtt{locked})$

$$\boxed{n}$$

$$\precsim$$

$\mathtt{let}\ v = \boxed{!\,\mathtt{x}_2}\ \mathtt{in}$

$\mathtt{x}_2 \leftarrow v + 1;$

$\mathsf{release}(l);\quad v$

Example

$\exists n.\, \texttt{x}_1 \mapsto_{\mathsf{i}} n \; *$

$\texttt{x}_2 \mapsto_{\mathsf{s}} n \; *$

$\mathsf{isLock}(l, \texttt{unlocked})$

$\texttt{x}_1 \mapsto_{\mathsf{i}} n + 1$

$\texttt{x}_2 \mapsto_{\mathsf{s}} n$

$\mathsf{isLock}(l, \texttt{locked})$

n

$\precsim$

$\texttt{x}_2 \leftarrow n + 1;$

$\mathsf{release}(l); \quad n$

Example

$\exists n. \, \texttt{x}_1 \mapsto_{\mathsf{i}} n \, *$

$\texttt{x}_2 \mapsto_{\mathsf{s}} n \, *$

$\mathsf{isLock}(l, \texttt{unlocked})$

$\texttt{x}_1 \mapsto_{\mathsf{i}} n + 1$

$\texttt{x}_2 \mapsto_{\mathsf{s}} n$

$\mathsf{isLock}(l, \texttt{locked})$

n

$\precsim$

$\texttt{x}_2 \leftarrow n + 1;$

$\mathsf{release}(l); \quad n$

Example

$$\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n * \mathtt{x}_2 \mapsto_{\mathsf{s}} n * \mathsf{isLock}(l, \mathtt{unlocked})$$

$$\mathtt{x}_1 \mapsto_{\mathsf{i}} n + 1$$

$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n + 1$$

$$\mathsf{isLock}(l, \mathtt{locked})$$

n

$\precsim$

$\mathsf{release}(l);\quad n$

Example

$$\exists n.\, \texttt{x}_1 \mapsto_{\mathsf{i}} n \; *$$
$$\texttt{x}_2 \mapsto_{\mathsf{s}} n \; *$$
$$\mathsf{isLock}(l, \texttt{unlocked})$$

$$\texttt{x}_1 \mapsto_{\mathsf{i}} n+1$$
$$\texttt{x}_2 \mapsto_{\mathsf{s}} n+1$$
$$\mathsf{isLock}(l, \texttt{locked})$$

n

$\precsim$

$\mathsf{release}(l);\; n$

Example

$$\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \,*\, \mathtt{x}_2 \mapsto_{\mathsf{s}} n \,*\, \mathsf{isLock}(l, \mathtt{unlocked})$$

$$\mathtt{x}_1 \mapsto_{\mathsf{i}} n + 1$$

$$\mathtt{x}_2 \mapsto_{\mathsf{s}} n + 1$$

$$\mathsf{isLock}(l, \mathtt{unlocked})$$

n

$\precsim$

n

Example

$$\boxed{\begin{aligned}\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n &* \\ \mathtt{x}_2 \mapsto_{\mathsf{s}} n &* \\ \mathrm{isLock}(l, \mathtt{unlocked})&\end{aligned}}$$

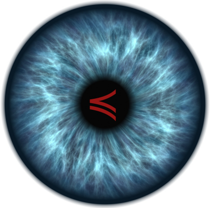
n

$\precsim$

n

Wrapping up...

- ReLoC provides rules allowing this kind of simulation reasoning, formally
- The example can be done in Coq in almost the same fashion
- The approach scales to: lock-free concurrent data structures, generative ADTs, examples from the logical relations literature

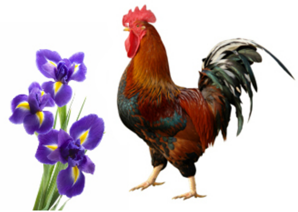


Implementation in Coq

ReLoC

ReLoC is build on top of the **Iris framework**, so we can inherit:

- Iris's invariants
- Iris's ghost state
- Iris's Coq infrastructure
- ...



The proofs we have done in Coq

ReLoC judgments $e_1 \precsim e_2 : \tau$ are modeled as a shallow embedding using the "logical approach" to logical relations

Proved in Coq:

- Proof rules: All the ReLoC rules hold in the shallow embedding
- Soundness: $e_1 \precsim e_2 : \tau \implies e_1 \precsim_{ctx} e_2 : \tau$
- Actual program refinements: concurrent data structures, and examples from the logical relations literature

Need to reason in separation logic!

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iInt
  iDes
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Lemma in the Iris logic

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
______________________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
--------------------------------------∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

```
1 subgoal
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
------------------------∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

∗ means: resources should be split

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
Qed.
```

The hypotheses for the left conjunct

```
1 subgoal
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
______________________________________(1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
--------------------------------------∗
Q ∗ (∃ a : A, P ∗ Ψ a)
```

∗ means: resources should be split

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
```

The hypotheses for the left conjunct

```
Qed.
```

```
2 subgoals
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
______________________(1/2)
"H3" : Q
_______________________∗
Q

______________________(2/2)
"H1" : P
"H2" : Ψ x
_______________________∗
∃ a : A, P ∗ Ψ a
```

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  iDestruct "H2" as (x) "H2".
  iSplitL "H3".
  - iAssumption.
  - iExists x.
    iFrame.
Qed.
```

Iris Proof Mode (IPM) [Krebbers *et al.*; POPL'17]

```
Lemma test {A} (P Q : iProp) (Ψ : A → iProp) :
  P ∗ (∃ a, Ψ a) ∗ Q −∗ Q ∗ ∃ a, P ∗ Ψ a.
Proof.
  iIntros "[H1 [H2 H3]]".
  by iFrame.
Qed.
```

We can also solve this lemma automatically

```
No more subgoals.
```

ReLoC in Iris Proof Mode

- The ReLoC rules are just lemmas that can be `iApplyed`
- We have more automated support for symbolic execution
- Iris Proof Mode features a special context for persistent hypotheses, which is crucial for dealing with invariants

Ongoing work:

Proving security properties using relational reasoning in separation logic

Language-based security

Program variables are divided into two groups:

- low-sensitivity variables $l_1, l_2, \ldots$
- high-sensitivity variables $h_1, h_2, \ldots$.

Confidentiality: the data stored in high-sensitivity variables should not leak to low-sensitivity variables, *e.g.*, $l_1 \leftarrow\, !\, h_1 + 1$ does not happen

Proved via **non-interference**: changing the values of $h_1, h_2, \ldots$ and running the program does not affect the resulting values of $l_1, l_2, \ldots$.

Type systems for non-interference

Type system where types are annotated with labels from a lattice $\mathbf{L} \sqsubseteq \mathbf{H}$

$$\vdash l_i : \texttt{ref int}^{\mathbf{L}} \qquad \vdash h_i : \texttt{ref int}^{\mathbf{H}}$$

$$\frac{\vdash e : \texttt{ref int}^{\chi} \qquad \vdash e' : \texttt{int}^{\xi} \qquad \xi \sqsubseteq \chi}{\vdash e \leftarrow e' : \texttt{unit}} \qquad \frac{\vdash e : \texttt{int}^{\chi} \qquad \vdash e' : \texttt{int}^{\xi}}{\vdash e + e' : \texttt{int}^{\chi \sqcup \xi}}$$

Type systems for non-interference

Type system where types are annotated with labels from a lattice $\mathbf{L} \sqsubseteq \mathbf{H}$

$$\vdash l_i : \texttt{ref int}^{\mathbf{L}} \qquad\qquad \vdash h_i : \texttt{ref int}^{\mathbf{H}}$$

$$\frac{\vdash e : \texttt{ref int}^{\chi} \qquad \vdash e' : \texttt{int}^{\xi} \qquad \xi \sqsubseteq \chi}{\vdash e \leftarrow e' : \texttt{unit}} \qquad\qquad \frac{\vdash e : \texttt{int}^{\chi} \qquad \vdash e' : \texttt{int}^{\xi}}{\vdash e + e' : \texttt{int}^{\chi \sqcup \xi}}$$

Example:

$$\frac{\vdash l_1 : \texttt{ref int}^{\mathbf{L}} \qquad \dfrac{\vdash !h_1 : \texttt{int}^{\mathbf{H}} \qquad \vdash 1 : \texttt{int}^{\mathbf{L}}}{\vdash !h_1 + 1 : \texttt{int}^{\mathbf{H}}} \qquad {\color{red}\mathbf{H} \sqsubseteq \mathbf{L}}}{\vdash l_1 \leftarrow !h_1 + 1 : \texttt{unit}}$$

Shortcomings of type systems

- Type systems can be extended to cover more PL features (dynamic references, higher-order functions, exceptions), although it is not straightforward
- Type systems are too weak: in many situations non-interference depends on functional correctness

Example: value-dependent classification

$$\texttt{let}\ r = \left\{ \begin{array}{l} data = \texttt{ref}(secret); \\ is\_classified = \texttt{ref}(\texttt{true}) \end{array} \right\}\ \texttt{in}$$

$$\begin{array}{l} \texttt{while true do} \\ \qquad \texttt{if}\ \neg\, !\, r.is\_classified \\ \qquad \texttt{then}\ out \leftarrow\, !\, r.data \\ \qquad \texttt{else}\ (); \end{array} \;\Bigg\|\; \begin{array}{l} r.data \leftarrow 0; \\ r.is\_classified \leftarrow \texttt{false} \end{array}$$

The classification of $r.data$ depends on the run-time value $r.is\_classified$

Example: value-dependent classification

$$\texttt{let}\ r = \left\{\begin{array}{l} data = \texttt{ref}(secret); \\ is\_classified = \texttt{ref}(\texttt{true}) \end{array}\right\}\ \texttt{in}$$

$$\begin{array}{l} \texttt{while true do} \\ \qquad \texttt{if}\ \neg\,!\,r.is\_classified \\ \qquad \texttt{then}\ out \leftarrow\ !\,r.data \\ \qquad \texttt{else}\ (); \end{array} \;\Bigg\|\; \begin{array}{l} r.data \leftarrow 0; \\ r.is\_classified \leftarrow \texttt{false} \end{array}$$

The classification of $r.data$ depends on the run-time value $r.is\_classified$

- Can we type this program with conventional type systems?
- Is this program secure?

Example: value-dependent classification

$$\texttt{let}\ r = \left\{ \begin{array}{l} \mathit{data} = \texttt{ref}(\mathit{secret}); \\ \mathit{is\_classified} = \texttt{ref}(\texttt{true}) \end{array} \right\}\ \texttt{in}$$

$$\begin{array}{l|l} \texttt{while true do} & \\ \quad \texttt{if}\ \neg\,!\,r.\mathit{is\_classified} & r.\mathit{data} \leftarrow 0; \\ \quad \texttt{then}\ \mathit{out} \leftarrow\, !\,r.\mathit{data} & r.\mathit{is\_classified} \leftarrow \texttt{false} \\ \quad \texttt{else}\ (); & \end{array}$$

The classification of $r.data$ depends on the run-time value $r.is\_classified$

- Can we type this program with conventional type systems? **No**
- Is this program secure? **Yes**

SeLoC [Frumin, Krebbers, Birkedal; Under submission]

SeLoC: a relational extension of Iris for non-interference

- A relational variant of weakest preconditions
- A type system built on top of SeLoC using logical relations
- Soundness w.r.t. scheduler-independent notion of non-interference

SeLoC [Frumin, Krebbers, Birkedal; Under submission]

SeLoC: a relational extension of Iris for non-interference

- A relational variant of weakest preconditions
  - to combine reasoning about non-interference with functional correctness
  - in the presence of fine-grained concurrency
- A type system built on top of SeLoC using logical relations

- Soundness w.r.t. scheduler-independent notion of non-interference

SeLoC [Frumin, Krebbers, Birkedal; Under submission]

SeLoC: a relational extension of Iris for non-interference

- A relational variant of weakest preconditions
  - to combine reasoning about non-interference with functional correctness
  - in the presence of fine-grained concurrency
- A type system built on top of SeLoC using logical relations
  - Compatibility rules for composing typed programs
  - Can "drop down" to separation logic to prove more complicated programs
- Soundness w.r.t. scheduler-independent notion of non-interference

Double weakest precondition

The basic component of SeLoC:

$$\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}$$

Double weakest precondition

The basic component of SeLoC:

$$\mathsf{dwp}\; e_1 \;\&\; e_2 \;\{\Phi\}$$

- Any two runs of the e_1 and e_2 are in a bisimulation and their results satisfy Φ

Double weakest precondition

The basic component of SeLoC:

$$\mathsf{dwp}\ e_1 \,\&\, e_2\ \{\Phi\}$$

- Any two runs of the e_1 and e_2 are in a bisimulation and their results satisfy Φ
- e_1 and e_2 have different secret data, but must produce the same public output

Double weakest precondition

The basic component of SeLoC:

$$\mathsf{dwp}\ e_1 \ \&\ e_2\ \{\Phi\}$$

- Any two runs of the e_1 and e_2 are in a bisimulation and their results satisfy Φ
- e_1 and e_2 have different secret data, but must produce the same public output
- Left-hand side and right-hand side resources: $\ell_1 \mapsto_{\mathsf{L}} v_1$ and $\ell_2 \mapsto_{\mathsf{R}} v_2$

Double weakest precondition

The basic component of SeLoC:

$$\text{dwp}\ e_1 \,\&\, e_2\ \{\Phi\}$$

- Any two runs of the e_1 and e_2 are in a bisimulation and their results satisfy Φ
- e_1 and e_2 have different secret data, but must produce the same public output
- Left-hand side and right-hand side resources: $\ell_1 \mapsto_{\mathsf{L}} v_1$ and $\ell_2 \mapsto_{\mathsf{R}} v_2$
- Rules are similar to ReLoC, but require execution in lock-step

Double weakest precondition

The basic component of SeLoC:

$$\mathrm{dwp}\ e_1 \,\&\, e_2\ \{\Phi\}$$

- Any two runs of the e_1 and e_2 are in a bisimulation and their results satisfy Φ
- e_1 and e_2 have different secret data, but must produce the same public output
- Left-hand side and right-hand side resources: $\ell_1 \mapsto_{\mathrm{L}} v_1$ and $\ell_2 \mapsto_{\mathrm{R}} v_2$
- Rules are similar to ReLoC, but require execution in lock-step
- Soundness statement:

$$(\forall h_1,\ h_2 \in \mathbb{Z}.\ I_{out} \vdash \mathrm{dwp}\ e[h_1/x] \,\&\, e[h_2/x]\ \{v_1 v_2.\ v_1 = v_2\}) \implies e \text{ is secure}$$

$$I_{out} \triangleq \boxed{\exists v \in \mathbb{Z}.\ out \mapsto_{\mathrm{L}} v * out \mapsto_{\mathrm{R}} v}$$

Proof of the example: value-dependent classification

$$\mathtt{let}\ r = \left\{\begin{array}{l} data = \mathtt{ref}(secret); \\ is\_classified = \mathtt{ref}(\mathtt{true}) \end{array}\right\}\ \mathtt{in}$$

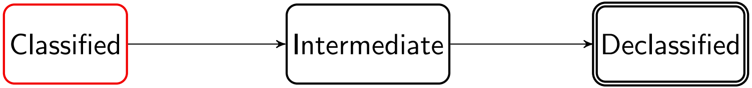
$$\begin{array}{l|l}
\mathtt{while\ true\ do} & \\
\quad \mathtt{if}\ \neg\,!\,r.is_classified & r.data \leftarrow 0; \\
\quad \mathtt{then}\ out \leftarrow\ !\,r.data & r.is_classified \leftarrow \mathtt{false} \\
\quad \mathtt{else}\ (); &
\end{array}$$

$$\boxed{\text{Classified}} \longrightarrow \boxed{\text{Intermediate}} \longrightarrow \boxed{\text{Declassified}}$$

Proof of the example: value-dependent classification

$$\texttt{let}\ r = \boxed{\left\{\begin{array}{l} data = \texttt{ref}(secret); \\ is\_classified = \texttt{ref}(\texttt{true}) \end{array}\right\}}\ \texttt{in}$$

$$\begin{array}{l|l} \texttt{while true do} & \\ \quad \texttt{if}\ \neg\,!\,r.is\_classified & r.data \leftarrow 0; \\ \quad \texttt{then}\ out \leftarrow\ !\,r.data & r.is\_classified \leftarrow \texttt{false} \\ \quad \texttt{else}\ (); & \end{array}$$



Proof of the example: value-dependent classification

$$\texttt{let}\; r = \left\{ \begin{array}{l} \mathit{data} = \texttt{ref}(\mathit{secret}); \\ \mathit{is\_classified} = \texttt{ref}(\texttt{true}) \end{array} \right\} \;\texttt{in}$$

$$\begin{array}{l|l} \texttt{while true do} & \\ \qquad \texttt{if}\; \neg\, !\, r.\mathit{is\_classified} & r.\mathit{data} \leftarrow 0; \\ \qquad \texttt{then}\; \mathit{out} \leftarrow\, !\, r.\mathit{data} & r.\mathit{is\_classified} \leftarrow \texttt{false} \\ \qquad \texttt{else}\; (); & \end{array}$$

Proof of the example: value-dependent classification

$$\texttt{let}\; r = \left\{ \begin{array}{l} \mathit{data} = \texttt{ref}(\mathit{secret}); \\ \mathit{is\_classified} = \texttt{ref}(\texttt{true}) \end{array} \right\} \;\texttt{in}$$

$$\begin{array}{l} \texttt{while true do} \\ \qquad \texttt{if}\; \neg\, !\, r.\mathit{is\_classified} \\ \qquad \texttt{then}\; \mathit{out} \leftarrow\, !\, r.\mathit{data} \\ \qquad \texttt{else}\; (); \end{array} \quad \Bigg\| \quad \begin{array}{l} r.\mathit{data} \leftarrow 0; \\ \colorbox{pink}{$r.\mathit{is\_classified} \leftarrow \texttt{false}$} \end{array}$$

$$\boxed{\text{Classified}} \longrightarrow \boxed{\text{Intermediate}} \longrightarrow \boxed{\text{Declassified}}$$

Proof of the example: value-dependent classification

$$
\boxed{
\begin{aligned}
&\big(\mathsf{in\_state}(\mathsf{Classified}) * \exists i_1, i_2.\; r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \texttt{true} * \\
&\quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \texttt{true} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i_1 * r_2.\mathit{data} \mapsto_{\mathsf{R}} i_2\big) \\
\vee &\big(\mathsf{in\_state}(\mathsf{Intermediate}) * \exists i.\; r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \texttt{true} * \\
&\quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \texttt{true} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i * r_2.\mathit{data} \mapsto_{\mathsf{R}} i\big) \\
\vee &\big(\mathsf{in\_state}(\mathsf{Declassified}) * \exists i.\; r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \texttt{false} * \\
&\quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \texttt{false} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i * r_2.\mathit{data} \mapsto_{\mathsf{R}} i\big)
\end{aligned}
}
$$

$$
\boxed{\text{Classified}} \longrightarrow \boxed{\text{Intermediate}} \longrightarrow \boxed{\boxed{\text{Declassified}}}
$$

Conclusions

Want to know more details

ReLoC: contextual refinements

ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency

Dan Frumin
Radboud University
dfrumin@cs.ru.nl

Robbert Krebbers
Delft University of Technology
mail@robbertkrebbers.nl

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Abstract

We present ReLoC: a logic for proving refinements of programs in a language with higher-order state, fine-grained concurrency, polymorphism and recursive types. The core of our logic is a judgement $e \precsim e' : \tau$, which expresses that a program e refines a program e' at type τ. In contrast to earlier work on refinements for languages with higher-order state and concurrency, ReLoC provides type- and structure-directed rules for manipulating this judgement, whereas previously, such proofs were carried out by unfolding the judgement into its definition in the model. These more abstract proof rules make it simpler to carry out refinement proofs.

Moreover, we introduce *logically atomic relational specifications*: a novel approach for relational specifications for compound expressions that take effect at a single instant in time. We demonstrate how to formalise and prove such relational specifications in ReLoC, allowing for more modular proofs.

ReLoC is built on top of the expressive concurrent separation logic Iris, allowing us to leverage features of Iris such as invariants and ghost state. We provide a mechanisation of our logic in Coq, which does not just contain a proof of soundness, but also tactics for interactively carrying out refinements proofs. We have used these tactics to mechanise several examples, which demonstrates the practicality and modularity of our logic.

CCS Concepts • **Theory of computation → Logic and verification**; **Separation logic**; *Concurrency*; *Program verification*;

Keywords Separation logic, logical relations, fine-grained concurrency, Iris, atomicity

ACM Reference Format:
Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July*

$$
\begin{aligned}
\mathsf{read} &\triangleq \lambda x\,().\ !\,x \\
\mathsf{inc}_s &\triangleq \lambda x\, l.\ \mathsf{acquire}\ l; \mathtt{let}\ n = !\,x\ \mathtt{in}\ x \leftarrow 1 + n; \mathsf{release}\ l;\ n \\
\mathsf{counter}_s &\triangleq \mathtt{let}\ l = \mathsf{newlock}\ ()\ \mathtt{in}\ \mathtt{let}\ x = \mathtt{ref}(0)\ \mathtt{in} \\
&\qquad (\mathsf{read}\ x, \lambda().\ \mathsf{inc}_s\ x\ l) \\
\mathsf{inc}_i &\triangleq \mathtt{rec}\ \mathit{inc}\ x = \mathtt{let}\ c = !\,x\ \mathtt{in} \\
&\qquad\qquad \mathtt{if}\ \mathtt{CAS}(x, c, 1 + c)\ \mathtt{then}\ c\ \mathtt{else}\ \mathit{inc}\ x \\
\mathsf{counter}_i &\triangleq \mathtt{let}\ x = \mathtt{ref}(0)\ \mathtt{in}\ (\mathsf{read}\ x, \lambda().\ \mathsf{inc}_i\ x)
\end{aligned}
$$

Figure 1. Two concurrent counter implementations.

are often referred to as the gold standards of equivalence and refinement of program expressions: contextual equivalence of e and e' means that it is safe for a compiler to replace any occurrence of e by e', and contextual refinement is often used to specify the behaviour of programs, *e.g.*, one can show the correctness of a fine-grained concurrent implementation of an abstract data type by proving that it contextually refines a coarse-grained implementation, which is understood as the specification.

A simple example is the specification of a fine-grained concurrent counter by a coarse-grained version, $\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N})$, see Figure 1 for the code. The increment operation of the coarse-grained version, $\mathsf{counter}_s$ is performed inside a critical section guarded by a lock, whereas the fine-grained version, $\mathsf{counter}_i$, takes an "optimistic" lock-free approach to incrementing the value using a compare-and-set inside a loop. We will use the counter as a simple running example throughout the paper. Proving program refinements and equivalence directly is difficult because of the quantification over all contexts. As such, it is often the case that

At LICS'18

SeLoC: non-interference

Compositional Non-Interference for Fine-Grained Concurrent Programs

Dan Frumin
Radboud University

Robbert Krebbers
Delft University of Technology

Lars Birkedal
Aarhus University

***Abstract*—We present SeLoC: a relational separation logic for verifying non-interference of fine-grained concurrent programs in a compositional way. SeLoC is more expressive than previous approaches, both in terms of the features of the target programming language, and in terms of the logic. The target programming language supports dynamically allocated references (pointers), higher-order functions, and fine-grained fork-based concurrency with low-level atomic operators like compare-and-set. The logic provides an invariant mechanism to establish protocols on data that is not protected by locks. This allows us to verify programs that were beyond the reach of previous approaches.**

A key technical innovation in SeLoC is a relational version of weakest-preconditions to track information flow using separation logic resources. On top of these weakest-preconditions we build a type system-like abstraction, using invariants and logical relations. SeLoC has been mechanized on top of the Iris framework in the Coq proof assistant.

***Index Terms*—non-interference, fine-grained concurrency, invariants, logical relations, separation logic, Coq, Iris**

I. Introduction

Non-interference is a form of *information flow control* (IFC) used to express security properties like confidentiality and secrecy, which guarantee that confidential information does not leak to attackers. In order to establish non-inference of programs used in practice, it is necessary to develop techniques that scale up to programming paradigms and programming constructs found in modern programming languages. Much effort has been put into that direction—*e.g.*, to support dynamically allocated references and higher-order functions [1]–[3], and

about each single run of a program, non-interference is stated in terms of multiple runs of the same program. One has to show that for different values of confidential inputs, the attacker cannot observe a different behavior.

Another reason for the discrepancy between the lack of expressiveness for techniques for non-interference compared to those for functional correctness is that a lot of prior work on non-interference has focused on type systems and type system-like logics, *e.g.*, [1], [4], [6], [9], [10]. Such systems have the benefit of providing strong automation (by means of type checking), but lack capabilities to reason about functional correctness, and therefore to establish non-interference of more challenging programs.

In order to overcome aforementioned shortcomings, we take a different and more expressive approach that combines the power of type systems and concurrent separation logic. In our approach, one assigns flexible interfaces to individual program modules using types. The program modules can then be composed using typing rules, ensuring non-interference of the whole system. Individual programs can be verified against those interfaces using a relational concurrent separation logic, which allows one to carry out non-interference proofs intertwined with functional correctness proofs.

Although ideas from concurrent separation logic have been employed for establishing non-interference (for first-order programs) before, see [9], [10], we believe that the combination

Under submission

Thank you!

Download **ReLoC** at `https://gitlab.mpi-sws.org/iris/reloc`

Download **Iris** at `https://iris-project.org/`