Relational reasoning using concurrent separation logic

Robbert Krebbers[1]

Delft University of Technology, The Netherlands

February 23, 2019 @ EU Types meeting, Krakow, Poland

[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

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

- Higher-order functions

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

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

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

- Concurrency

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

What do such relational properties mean mathematically?

Contextual refinement

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

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

"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:

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

"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"

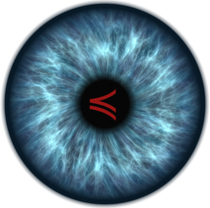
ReLoC [Frumin, Krebbers, Birkedal; LICS'18]

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]

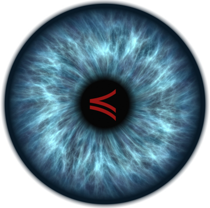
ReLoC: mechanized separation logic for interactive refinement proofs of fine-grained concurrent programs.

- **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

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

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

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

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)

Logic with first-class refinement propositions to allow conditional refinements

- $(\ell_1 \mapsto_{\mathsf{i}} v) \ \mathrel{-\!\!*}\ (e_1 \precsim e_2 : \tau)$
- $(e_1 \precsim e_2 : \mathbf{1} \to \tau) \ \mathrel{-\!\!*}\ (f(e_1) \precsim e_2(); e_2() : \tau)$

Proving refinements of pure programs

Some rules for pure programs

Symbolic execution rules

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

Some rules for pure programs

Symbolic execution rules

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

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

Some rules for pure programs

Structural rules

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

Some rules for pure programs

Structural rules

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

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

Some rules for pure programs

Structural rules

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

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

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

Example

A bit interface:

$$\mathtt{bitT} \triangleq \exists\alpha.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

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

Example

A bit interface:

$$\texttt{bitT} \triangleq \exists\alpha.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

- 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:

$$\texttt{bitT} \triangleq \exists\alpha.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

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

Two implementations:

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

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

Refinement (and vice versa):

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

Proof of the refinement

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

$$\precsim$$

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

$$:$$

$$\exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

Proof of the refinement

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

$$\precsim$$

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

$$:$$

$$\exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

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

Proof of the refinement

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

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

$$\precsim$$

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

$$:$$

$$\exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

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

Proof of the refinement

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

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

$$\precsim$$

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

$$:$$

$$\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

Proof of the refinement

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

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

$$\precsim$$

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

$$:$$

$$\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

Use structural rule for products

Proof of the refinement

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

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

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

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathbf{2}$$

Proof of the refinement

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

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

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

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathbf{2}$$

Proof of the refinement

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

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

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

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathbf{2}$$

Proof of the refinement

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

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

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

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathbf{2}$$

After using the λ rule and case analysis on R:

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

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

Proof of the refinement

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

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

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

$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathbf{2} \quad$ (λ-rule + symb. exec.)

Proof of the refinement

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

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

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

$$(\lambda b.\, b) \precsim (\lambda n.\, n = 1) \quad : \alpha \to \mathbf{2} \qquad (\lambda\text{-rule} + \text{symb. exec.})$$

$\square$

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:

$$\ell_1 \mapsto_{\mathsf{i}} 4 \quad -\!\!* \quad \ell_2 \mapsto_{\mathsf{s}} 0 \quad -\!\!* \quad (!\,\ell_1) \precsim (\ell_2 \leftarrow 4; !\,\ell_2) : \mathbf{N}$$

Some rules for mutable state

Symbolic execution

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

Some rules for mutable state

Symbolic execution

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

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

Some rules for mutable state

Symbolic execution

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

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

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

Reasoning about higher-order functions and concurrency

State encapsulation

Modules with **encapsulated state**:

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

The reference can only be used in the closure

State encapsulation

Modules with **encapsulated state**:

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

The reference can only be used in the closure

Simple example:

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

- $\texttt{counter}()$ constructs an instance $c : \mathbf{1} \to \mathbf{N}$ of the counter module
- Calling $c()$ in subsequently gives $0, 1, 2, \dots$
- The reference x is private to the module

The problem

Modules with **encapsulated state**:

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

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**:

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

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 $\Box$ in Iris/ReLoC:

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

Examples:

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

ReLoC's λ-rule again

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

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

ReLoC's λ-rule again

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

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

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 $\Box$, the resource $x \mapsto_{\mathsf{s}} 0$ cannot be used to prove the closure

But it should be possible to use resources in closures

For example:

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

$$\precsim$$

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

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: $\boxed{R} \vdash \Box \boxed{R}$
- ...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

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

$$\precsim$$

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

Example

$$\overline{}$$

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

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

$$\precsim$$

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

Example

$x_1 \mapsto_i 1$

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

$$\precsim$$

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

Example

$$\dfrac{}{\mathtt{x}_1 \mapsto_{\mathsf{i}} 1 \qquad \mathtt{x}_2 \mapsto_{\mathsf{s}} 1}$$

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

$$\precsim$$

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

Example

$$\overline{\qquad\qquad\qquad}$$

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

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

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

$$\precsim$$

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

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

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

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

$$\precsim$$

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

Example

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

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

$$\precsim$$

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

Example

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

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

$$\precsim$$

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

Example

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

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

$$\precsim$$

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

Example

$\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \ast$

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

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

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

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

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

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

$\precsim$

$\mathsf{acquire}(l);$

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

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

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

Example

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

$$\boxed{n}$$

$$\precsim$$

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

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

Example

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

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

$$n$$

$$\precsim$$

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

Example

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

$$\boxed{n}$$

$$\precsim$$

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

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

Example

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

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

$$n$$

$$\precsim$$

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

Example

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

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

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

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

$$n$$

$$\precsim$$

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

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

Example

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

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

$$n$$

$$\precsim$$

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

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

Example

$\exists n.\, \mathtt{x}_1 \mapsto_{\mathsf{i}} n \ast$

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

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

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

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

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

n

$\precsim$

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

Example

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

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

n

$\precsim$

release(l); n

Example

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

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

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

Logically atomic relational specifications

Problem

- The example that we have seen is a bit more subtle: the fetch-and-add (`FAA`) function is not a physically atomic instruction
- What kind of specification can we give to `FAA` as a compound program?

Logically atomic relational specifications

Problem

- The example that we have seen is a bit more subtle: the fetch-and-add (`FAA`) function is not a physically atomic instruction
- What kind of specification can we give to `FAA` as a compound program?

Our solution

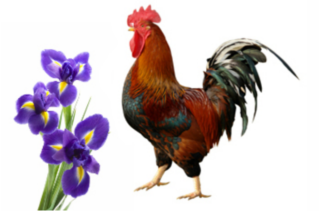
Relational version of TaDA-style logically atomic triples in ReLoC

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 `iApply`ed
- 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

Persistent propositions in Iris Proof Mode

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

Persistent propositions in Iris Proof Mode

```
Lemma test {PROP : bi} {A}
    (P Q : PROP) (Ψ : A → PROP) :
  P ∗ □ (∃ a, Ψ a) -∗ ∃ a, Ψ a ∗ (P ∗ Ψ a).
Proof.
  iI
  iDestruct "H2" as (x) "H2".
  iExists x.
  iSplitL "H2".
  - iAssumption.
  - by iFrame.
Qed.
```

Persistent modality

Persistent propositions in Iris Proof Mode

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

```
1 subgoal
PROP : bi
A : Type
P, Q : PROP
Ψ : A → PROP
_________________________(1/1)
"H2" : ∃ a : A, Ψ a
-------------------------□
"H1" : P
-------------------------∗
∃ a : A, Ψ a ∗ (P ∗ Ψ a)
```

Moves hypothesis to **persistent** context

Persistent propositions in Iris Proof Mode

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

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

Persistent propositions in Iris Proof Mode

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

Do not need to split persistent context

```
1 subgoal
PROP : bi
A : Type
P, Q : PROP
Ψ : A → PROP
x : A
______________________________________(1/1)
"H2" : Ψ x
--------------------------------------□
"H1" : P
--------------------------------------∗
Ψ x ∗ (P ∗ Ψ x)
```

Persistent propositions in Iris Proof Mode

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

```
2 subgoals
PROP : bi
A : Type
P, Q : PROP
Ψ : A → PROP
x : A
______________________________________(1/2)
"H2" : Ψ x
--------------------------------------□
Ψ x

______________________________________(2/2)
"H2" : Ψ x
--------------------------------------□
"H1" : P
--------------------------------------∗
P ∗ Ψ x
```

Conclusions

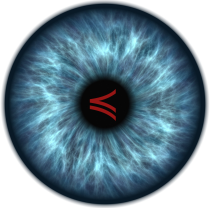
Conclusions and future work

Contributions

- ReLoC: a logic that allows to carry out refinement proofs interactively in Coq
- New approach to modular refinement specifications for logically atomic programs
- Case studies: concurrent data structures, and examples from the logical relations literature

Future work

- Program transformations
- Refinements between programs in different language
- Other relational properties of concurrent programs



Want to know more details

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,

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

Figure 1. Two concurrent counter implementations.

are often referred to as the gold standards of equivalence and refine-

Thank you!

Download ReLoC at `https://cs.ru.nl/~dfrumin/reloc/`
Download Iris at `https://iris-project.org/`

Advertisement. I currently have a vacancy for a fully funded PhD position (4 years) in the beautiful Netherlands

Topics: Separation logic for multilingual programs, asynchronous I/O, non-functional properties, verified compilation, proof automation, tactics, ...

Interested/Know someone? Get in touch!