Relational reasoning using concurrent separation logic

Robbert Krebbers\textsuperscript{1}

Delft University of Technology, The Netherlands

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

\textsuperscript{1}This is joint work with Dan Frumin (Radboud University) and Lars Birkedal (Aarhus University)
Why prove relational properties of programs?

- Specifying programs

\[
\text{implementation} \preceq_{\text{ctx}} \text{specification}
\]
Why prove relational properties of programs?

- Specifying programs

\[
\text{implementation} \preceq_{\text{ctx}} \text{specification}
\]

- Optimized versions of data structures

\[
\text{hash_table} \preceq_{\text{ctx}} \text{assoc_list}
\]
Why prove relational properties of programs?

- Specifying programs

\[ \text{implementation} \preceq_{\text{ctx}} \text{specification} \]

- Optimized versions of data structures

\[ \text{hash\_table} \preceq_{\text{ctx}} \text{assoc\_list} \]

- Proving program transformations

\[ \forall e_{\text{source}}. \; \text{compile}(e_{\text{source}}) \preceq_{\text{ctx}} e_{\text{source}} \]
Language features that complicate refinements

- **Mutable state**

\[
\left( \text{let } x = f() \text{ in } (x, x) \right) \not\sim_{\text{ctx}} \left( f(), f() \right)
\]
Language features that complicate refinements

- **Mutable state**
  \[
  \left( \text{let } x = f() \text{ in } (x, x) \right) \not\leq_{ctx} \left( f(), f() \right)
  \]

- **Higher-order functions**
  \[
  \left( \lambda(). 1 \right) \not\leq_{ctx} \left( \text{let } x = \text{ref}(0) \text{ in } (\lambda(). x \leftarrow (1 + !x); !x) \right)
  \]
Language features that complicate refinements

- **Mutable state**
  \[
  \left( \text{let } x = f() \text{ in } (x, x) \right) \not\preceq_{\text{ctx}} \left( f(), f() \right)
  \]

- **Higher-order functions**
  \[
  \left( \lambda(). 1 \right) \not\preceq_{\text{ctx}} \left( \text{let } x = \text{ref}(0) \text{ in } (\lambda(). x \leftarrow (1 + !x); !x) \right)
  \]

- **Concurrency**
  \[
  \left( x \leftarrow 10; x \leftarrow 11 \right) \not\preceq_{\text{ctx}} \left( x \leftarrow 11 \right)
  \]
What do such relational properties mean mathematically?
Contextual refinement

Contextual refinement: the “gold standard” of program refinement:

\[ e_1 \prec_{ctx} e_2 : \tau \triangleq \forall (C : \tau \to N). \forall v. C[e_1] \downarrow v \implies C[e_2] \downarrow v \]

“Any behavior of a (well-typed) client \( C \) using \( e_1 \) can be matched by a behavior of the same client using \( e_2 \)”
Contextual refinement: the “gold standard” of program refinement:

\[ e_1 \succeq_{\text{ctx}} e_2 : \tau \triangleq \forall (C : \tau \rightarrow \mathbb{N}). \forall v. C[e_1] \downarrow v \implies C[e_2] \downarrow v \]

“Any behavior of a (well-typed) client \( C \) 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**: 

\[ e_1 \preceq e_2 : \tau \]

- \( e_1 \preceq e_2 : \tau \) is defined structurally on the type \( \tau \)
- Does not involve quantification over all clients \( C \)
- Soundness \( e_1 \preceq e_2 : \tau \implies e_1 \preceq_{\text{ctx}} e_2 : \tau \) proved once and for all
A bit of history

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

Logical relations $e_1 \sim e_2 : \tau$ 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 $e_1 \sim e_2 : \tau$ 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 $e_1 \preceq e_2 : \tau$ 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 $e_1 \sim e_2 : \tau$ 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 $e_1 \preceq e_2 : \tau$

- 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: 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: 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

\[ P, Q \in \text{Prop} ::= \forall x. P \mid \exists x. P \mid P \lor Q \mid \ldots \]
ReLoC: (simplified) grammar

\[ P, Q \in \text{Prop} ::= \forall x. P \mid \exists x. P \mid P \lor Q \mid \ldots \]

\mid P \ast Q \mid P \rightarrow^* Q \mid \ell \mapsto_i v \mid \ell \mapsto_s v

Separation logic for handling mutable state

- \( \ell \mapsto_i v \) for the left-hand side (implementation)
- \( \ell \mapsto_s v \) for the right-hand side (specification)
ReLoC: (simplified) grammar

\[ P, Q \in \text{Prop} ::= \forall x.\ P \mid \exists x.\ P \mid P \lor Q \mid \ldots \]
\[ \mid P \ast Q \mid P \rightarrow Q \mid \ell \mapsto_i v \mid \ell \mapsto_s v \]
\[ \mid (\Delta \models e_1 \preceq e_2 : \tau) \mid \llbracket \tau \rrbracket_{\Delta}(v_1, v_2) \mid \ldots \]

Separation logic for handling mutable state

- \( \ell \mapsto_i v \) for the left-hand side (implementation)
- \( \ell \mapsto_s v \) for the right-hand side (specification)

Logic with first-class refinement propositions to allow conditional refinements

- \((\ell_1 \mapsto_i v) \ast (e_1 \preceq e_2 : \tau)\)
- \((e_1 \preceq e_2 : \mathbf{1} \rightarrow \tau) \ast (f(e_1) \preceq e_2(); e_2() : \tau)\)
Proving refinements of pure programs
Some rules for pure programs

Symbolic execution rules

\[
\Delta \models K[e_1'] \preceq e_2 : \tau \quad e_1 \rightarrow_{\text{pure}} e_1'
\]

\[
\Delta \models K[e_1] \preceq e_2 : \tau
\]
Some rules for pure programs

Symbolic execution rules

\[
\Delta \vdash K[e'_1] \preceq e_2 : \tau \quad \Rightarrow \quad e_1 \rightarrow_{\text{pure}} e'_1
\]

\[
\Delta \vdash K[e_1] \preceq e_2 : \tau
\]

\[
\Delta \vdash e_1 \preceq K[e'_2] : \tau \quad \Rightarrow \quad e_2 \rightarrow_{\text{pure}} e'_2
\]

\[
\Delta \vdash K[e_2] : \tau
\]
Some rules for pure programs

Structural rules

\[
\begin{align*}
\Delta \models e_1 \preceq e_2 : \tau \\
\Delta \models e'_1 \preceq e'_2 : \tau' \\
\Delta \models (e_1, e'_1) \preceq (e_2, e'_2) : \tau \times \tau'
\end{align*}
\]
Some rules for pure programs

Structural rules

\[
\Delta \models e_1 \not\sim e_2 : \tau \quad \ast \quad \Delta \models e'_1 \not\sim e'_2 : \tau'
\]

\[
\Delta \models (e_1, e'_1) \not\sim (e_2, e'_2) : \tau \times \tau'
\]

\[
\exists (R : Val \times Val \rightarrow \text{Prop}). \quad [\alpha := R], \Delta \models e_1 \not\sim e_2 : \tau
\]

\[
\Delta \models \text{pack}(e_1) \not\sim \text{pack}(e_2) : \exists \alpha. \tau
\]
Some rules for pure programs

Structural rules

\[
\begin{align*}
\Delta \vdash e_1 \preceq e_2 &: \tau & \Delta \vdash e'_1 \preceq e'_2 &: \tau' \\
\Delta \vdash (e_1, e'_1) \preceq (e_2, e'_2) &: \tau \times \tau'
\end{align*}
\]

\[\exists (R : \text{Val} \times \text{Val} \rightarrow \text{Prop}). \quad [\alpha := R], \Delta \vdash e_1 \preceq e_2 &: \tau \]

\[\Delta \vdash \text{pack}(e_1) \preceq \text{pack}(e_2) &: \exists \alpha. \tau\]

\[\begin{align*}
\llbracket \tau \rrbracket_\Delta (v_1, v_2) & \quad \Box \left( \forall v_1 \ v_2. \ \llbracket \tau \rrbracket_\Delta (v_1, v_2) \rightarrow^* \Delta \vdash e_1[v_1/x_1] \preceq e_2[v_2/x_2] &: \sigma \right) \\
\Delta \vdash v_1 \preceq v_2 &: \tau & \Delta \vdash \lambda x_1. e_1 \preceq \lambda x_2. e_2 &: \tau \rightarrow \sigma
\end{align*}\]
Example

A bit interface:

$$\text{bitT} \triangleq \exists \alpha. \quad \alpha \times (\alpha \to \alpha) \times (\alpha \to 2)$$

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

A bit interface:

\[
\text{bitT} \triangleq \exists \alpha. \, \alpha \times (\alpha \to \alpha) \times (\alpha \to 2)
\]

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

Two implementations:

\[
\text{bit_bool} \triangleq \text{pack}\left(\text{true}, (\lambda b. \neg b), (\lambda b. b)\right)
\]

\[
\text{bit_nat} \triangleq \text{pack}\left(1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)\right)
\]
Example

A bit interface:

$$\text{bitT} \triangleq \exists \alpha. \frac{}{} \alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow 2)$$

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

Two implementations:

bit\_bool \triangleq \text{pack}\left(true, (\lambda b. \neg b), (\lambda b. b)\right)

bit\_nat \triangleq \text{pack}\left(1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)\right)

Refinement (and vice versa):

$$\text{bit\_bool} \preceq \text{bit\_nat} : \text{bitT}$$
Proof of the refinement

pack\left(\text{true}, (\lambda b. \neg b), (\lambda b. b)\right)

\sim

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

:\exists \alpha. \alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow 2)
Proof of the refinement

\[
\begin{align*}
\text{pack} \left( \text{true}, (\lambda b. \neg b), (\lambda b. b) \right) \\
\sim \\
\text{pack} \left( 1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1) \right) \\
: \\
\exists \alpha. \alpha \times (\alpha \to \alpha) \times (\alpha \to 2)
\end{align*}
\]

Need to come with with an \( R : \text{Val} \times \text{Val} \to \text{Prop} \)
Proof of the refinement

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

\[
\text{pack}(\text{true}, (\lambda b. \neg b), (\lambda b. b)) \sim \\
\text{pack}(1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)) \\
: \\
\exists \alpha. \alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow 2)
\]

Need to come with an \( R : \text{Val} \times \text{Val} \rightarrow \text{Prop} \)
Proof of the refinement

\[ [\alpha := R] |\rightarrow \ \text{where } R \triangleq \{(\text{true, 1}), (\text{false, 0})\} \]

\[
\left(\text{true, } (\lambda b. \neg b), (\lambda b. b)\right)
\]

\[\sim\]

\[
\left(1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)\right)
\]

\[\vdash\]

\[
\alpha \times (\alpha \to \alpha) \times (\alpha \to 2)
\]
Proof of the refinement

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

\[
\left(\text{true}, (\lambda b. \neg b), (\lambda b. b)\right)
\]

\[
\leadsto
\left(1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)\right)
\]

\[
\alpha \times (\alpha \to \alpha) \times (\alpha \to 2)
\]

Use structural rule for products
Proof of the refinement

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

\[
\begin{align*}
\text{true} & \preceq 1 : \alpha \\
(\lambda b. \neg b) & \preceq (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) : \alpha \rightarrow \alpha \\
(\lambda b. b) & \preceq (\lambda n. n = 1) : \alpha \rightarrow 2
\end{align*}
\]
Proof of the refinement

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

\[
\begin{align*}
\text{true} & \preceq 1 : \alpha \\
(\lambda b. \neg b) & \preceq (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) : \alpha \rightarrow \alpha \\
(\lambda b. b) & \preceq (\lambda n. n = 1) : \alpha \rightarrow 2
\end{align*}
\]
Proof of the refinement

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

true \preceq 1 \quad : \alpha \quad \text{(by def of } R) \]

\((\lambda b. \neg b) \preceq (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) \quad : \alpha \to \alpha \]

\((\lambda b. b) \preceq (\lambda n. n = 1) \quad : \alpha \to 2 \]
Proof of the refinement

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

\[
\begin{align*}
\text{true} & \preceq 1 : \alpha \\
(\lambda b. \neg b) & \preceq (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) : \alpha \rightarrow \alpha \\
(\lambda b. b) & \preceq (\lambda n. n = 1) : \alpha \rightarrow 2
\end{align*}
\]

(by def of \( R \))

(\( \lambda \)-rule + symb. exec.)

After using the \( \lambda \) rule and case analysis on \( R \):

\[
\begin{align*}
\neg \text{true} & \preceq \text{if } 1 = 0 \text{ then } 1 \text{ else } 0 : \alpha \\
\neg \text{false} & \preceq \text{if } 0 = 0 \text{ then } 1 \text{ else } 0 : \alpha
\end{align*}
\]
Proof of the refinement

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

\[
\begin{align*}
\text{true} & \preceq 1 : \alpha \\
(\lambda b. \neg b) & \preceq (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) : \alpha \rightarrow \alpha \quad (\lambda\text{-rule + symb. exec.}) \\
(\lambda b. b) & \preceq (\lambda n. n = 1) : \alpha \rightarrow 2 \quad (\lambda\text{-rule + symb. exec.})
\end{align*}
\]
Proof of the refinement

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

\[ \text{true} \preceq 1 \quad : \alpha \quad \text{(by def of } R) \]

\[ (\lambda b. \neg b) \preceq (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) \quad : \alpha \rightarrow \alpha \quad \text{(\lambda-rule + symb. exec.)} \]

\[ (\lambda b. b) \preceq (\lambda n. n = 1) \quad : \alpha \rightarrow 2 \quad \text{(\lambda-rule + symb. exec.)} \]
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 $\ell$ with value $v$

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

Basic example:

$$\{\ell_1 \mapsto v_1 \ast \ell_2 \mapsto v_2\} \text{swap}(\ell_1, \ell_2)\{\ell_1 \mapsto v_2 \ast \ell_2 \mapsto v_1\}$$

the $\ast$ ensures that $\ell_1$ and $\ell_2$ are different memory locations
Mutable state and separation logic for refinements

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

- $\ell \mapsto_{i} v$ for the left-hand side/implementation
- $\ell \mapsto_{s} v$ for the right-hand side/specification
Mutable state and separation logic for refinements

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

- $\ell \mapsto_i v$ for the left-hand side/implementation
- $\ell \mapsto_s v$ for the right-hand side/specification

**Example:**

$$\ell_1 \mapsto_i 4 \quad \ast \quad \ell_2 \mapsto_s 0 \quad \ast \quad (! \ell_1) \preceq (\ell_2 \leftarrow 4; ! \ell_2) : \text{N}$$
Some rules for mutable state

Symbolic execution

\[
\ell_1 \rightsquigarrow_i - \quad \ast \quad (\ell_1 \rightsquigarrow_i v_1 \quad \ast \quad \Delta \models \; K[()] \not\simeq e_2 : \tau) \quad \ast
\]

\[
\Delta \models \; K[\ell_1 \leftarrow v_1] \not\simeq e_2 : \tau
\]
Some rules for mutable state

Symbolic execution

\[
\begin{align*}
\ell_1 \mapsto_i - & \quad \ast \quad (\ell_1 \mapsto_i v_1 \ast \Delta \models K[()] \preceq e_2 : \tau) \ast \\
\Delta \models K[\ell_1 \leftarrow v_1] \preceq e_2 : \tau &
\end{align*}
\]

\[
\begin{align*}
\ell_2 \mapsto_s - & \quad \ast \quad (\ell_2 \mapsto_s v_2 \ast \Delta \models e_1 \preceq K[()] : \tau) \ast \\
\Delta \models e_1 \preceq K[\ell_2 \leftarrow v_2] : \tau &
\end{align*}
\]
Some rules for mutable state

Symbolic execution

$$\ell_1 \mapsto_i - * \quad (\ell_1 \mapsto_i v_1 \rightarrow \Delta \models K[()] \preceq e_2 : \tau)$$

$$\Delta \models K[\ell_1 \leftarrow v_1] \preceq e_2 : \tau$$

$$\ell_2 \mapsto_s - * \quad (\ell_2 \mapsto_s v_2 \rightarrow \Delta \models e_1 \preceq K[()] : \tau)$$

$$\Delta \models e_1 \preceq K[\ell_2 \leftarrow v_2] : \tau$$

$$\forall \ell_1. \ell_1 \mapsto_i v_1 \rightarrow \Delta \models K[\ell_1] \preceq e_2 : \tau$$

$$\Delta \models K[\text{ref}(v_1)] \preceq e_2 : \tau$$

$$\forall \ell_2. \ell_2 \mapsto_s v_2 \rightarrow \Delta \models e \preceq K[\ell_2] : \tau$$

$$\Delta \models e \preceq K[\text{ref}(v_2)] : \tau$$
Reasoning about higher-order functions and concurrency
State encapsulation

Modules with **encapsulated state**:

\[
\texttt{let } x = \texttt{ref}(e) \texttt{in } (\lambda y. \ldots )
\]

The reference can only be used in the closure

---

Simple example:

\[
\texttt{counter } \triangleq (\lambda () \texttt{. let } x = \texttt{ref}(1) \texttt{in } (\lambda () \texttt{. } \texttt{FAA}(x, 1)))
\]

\[1 \rightarrow (1 \rightarrow \mathbb{N})\]

\[\texttt{counter}()\] constructs an instance \(c\): \(1 \rightarrow \mathbb{N}\) of the counter module

Calling \(c()\) in subsequently gives 0, 1, 2, ...

\(x\) is private to the module
State encapsulation

Modules with **encapsulated state**:

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

The reference can only be used in the closure

Simple example:

```latex
\texttt{counter} \triangleq \left( \lambda(). \texttt{let } x = \texttt{ref}(1) \texttt{in} \left( \lambda(). \texttt{FAA}(x, 1) \right) \right) : 1 \rightarrow (1 \rightarrow \mathbb{N})
```

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

Modules with **encapsulated state**:

\[
\text{let } x = \text{ref}(e) \text{ in } (\lambda y. \ldots )
\]

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 in parallel!
So, \( f \) cannot get exclusive access to \( x \)

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

Modules with **encapsulated state**:

\[
\text{let } x = \text{ref}(e) \text{ in } \left( \lambda y. \ldots \right)
\]

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**:

\[
\text{let } x = \text{ref}(e) \text{ in } \left( \lambda y. \ldots \right)
\]

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**:

\[
\text{let } x = \text{ref}(e) \text{ in } (\lambda y. \ldots)
\]

**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:

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

Examples:

- Equality is persistent: $(x = y) \vdash \Box (x = y)$
- Points-to connectives are not: $(\ell \rightarrow v) \nvdash \Box (\ell \rightarrow v)$
- More examples later...
ReLoC’s $\lambda$-rule again

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

\[
\begin{align*}
\square \left( \forall v_1, v_2. [[\tau]] \Delta(v_1, v_2) \rightarrow^* \\
\Delta \models e_1[v_1/x_1] \preceq e_2[v_2/x_2] : \sigma \right) \\
\Delta \models \lambda x_1. e_1 \preceq \lambda x_2. e_2 : \tau \rightarrow \sigma
\end{align*}
\]

Due to $\square$, the resource $x \mapsto \sigma$ cannot be used to prove the closure...
ReLoC’s $\lambda$-rule again

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

\[
\frac{
\Box \left( \forall v_1 \, v_2 \cdot [\tau] \Delta(v_1, v_2) \rightarrow^* \right)
\Delta \models e_1[v_1/x_1] \not\lesssim e_2[v_2/x_2] : \sigma
}{\Delta \models \lambda x_1. e_1 \not\lesssim \lambda x_2. e_2 : \tau \rightarrow \sigma}
\]

Prohibits “wrong” refinements, for example:

\[
\left( \lambda(). 1 \right) \not\lesssim_{ctx} \left( \text{let } x = \text{ref}(0) \text{ in } (\lambda(). x \leftarrow (1 + !x); !x) \right)
\]

Due to $\Box$, 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:

\[
\left( \lambda(). \text{let } x = \text{ref}(1) \text{ in } (\lambda(). \text{FAA}(x, 1)) \right)
\]

\[\approx\]

\[
\left( \lambda(). \text{let } x = \text{ref}(1), l = \text{newlock}() \text{ in }
\begin{align*}
\lambda(). & \text{acquire}(l); \\
\text{let } v = !x & \text{ in } \\
x & \leftarrow v + 1; \\
\text{release}(l); v
\end{align*}
\right)
\]
Iris-style Invariants

The invariant connective $\mathsf{R}$
expresses that $R$ is maintained as an invariant on the state
Iris-style Invariants

The invariant connective $\Box R$ expresses that $R$ is maintained as an invariant on the state.

Invariants allow to share resources:

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

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

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

**But that comes with a cost:**
- Invariants $\mathbf{R}$ can only be accessed during atomic steps on the left-hand side.
- ... while multiple steps on the right-hand side can be performed.
Example

```plaintext
let x = ref(1) in (λ(). FAA(x, 1))
```

```plaintext
let x = ref(1), l = newlock () in
  (λ(). acquire(l);
   let v = !x in
   x ← v + 1;
   release(l; v)
  )
```
Example

let \( x = \text{ref}(1) \) in \((\lambda(). \text{FAA}(x, 1))\)
Example

\[ x_1 \mapsto_i 1 \]

\[ (\lambda(). \text{FAA}(x_1, 1)) \]

\[ \sim \]

\[
\text{let } x = \text{ref}(1), l = \text{newlock}() \text{ in }
(\lambda(). \text{acquire}(l);
  \text{let } v = !x \text{ in }
  x \leftarrow v + 1;
  \text{release}(l); v)
\]
Example

$\exists \ n$.  $x_1 \mapsto i \ 1$

$(\lambda().\ FAA(x_1, 1))$

$\simeq$

let $x = \textbf{ref}(1), l = \textbf{newlock()}$ in

$(\lambda().\ \textbf{acquire}(l);$  
let $v = !x$ in 

$x \leftarrow v + 1;$  
release($l; v$)
Example

\[ \exists n. x_1 \mapsto_i 1 \quad x_2 \mapsto_s 1 \]

\[ (\lambda(). FAA(x_1, 1)) \]

\[ \sim \]

\[ \text{let } l = \text{newlock()} \text{ in} \]
\[ (\lambda(). \text{acquire}(l); \]
\[ \text{let } v = !x_2 \text{ in} \]
\[ x_2 \leftarrow v + 1; \]
\[ \text{release}(l); v) \]
Example

\[ \exists n. x_1 \mapsto i 1 \]
\[ x_2 \mapsto s 1 \]

\[
(\lambda(). FAA(x_1, 1))
\]

\[
\sim
\]

let \( l = \) newlock () in

(\( l = \) acquire(l);
  let \( v = !x_2 \) in
  \( x_2 \leftarrow v + 1; \)
  release(l); v)
Example

\[ \exists n. \]

\[ x_1 \mapsto i \ 1 \]

\[ x_2 \mapsto s \ 1 \]

isLock(l, unlocked)

\[ (\lambda(). \ FAA(x_1, 1)) \]

\[ \sim \]

\[ (\lambda(). \ acquire(l); \]

\[ \text{let } v = !x_2 \text{ in} \]

\[ x_2 \leftarrow v + 1; \]

\[ \text{release}(l); v ) \]
Example

\[ \exists n. \begin{align*}
    x_1 & \mapsto_i n \\
    x_2 & \mapsto_s n \\
    \text{isLock}(l, \text{unlocked})
\end{align*} \]
Example

\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ (\lambda(). \ FAA(x_1, 1)) \]

\[ \sim \]

\[ (\lambda(). \ acquire(l); \]
\[ \quad \text{let } v = !x_2 \text{ in} \]
\[ \quad x_2 \leftarrow v + 1; \]
\[ \quad \text{release}(l); v) \]
Example

\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ \text{FAA}(x_1, 1) \]
\[ \sim \]

acquire(l);

let \( v = !x_2 \) in

\( x_2 \leftarrow v + 1; \)

release(l); \( v \)
Example

\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ \text{FAA}(x_1, 1) \]

\[ \sim \]

acquire(l);

\[ \text{let } v = !x_2 \text{ in } \]

\[ x_2 \leftarrow v + 1; \]

release(l); \ v
Example

$\exists n. x_1 \mapsto_i n \ast$

$x_2 \mapsto_s n \ast$

isLock(l, unlocked)

$\mapsto$

$\mapsto$

FAA($x_1, 1$)

$\mapsto$

$x_1 \mapsto_i n$

$x_2 \mapsto_s n$

isLock(l, unlocked)

acquire(l);

let $v = !x_2$ in

$x_2 \leftarrow v + 1;$

release(l); $v$
Example

\[ \exists n. x_1 \mapsto_i n \ast \\
\quad x_2 \mapsto_s n \ast \\
\quad \text{isLock}(l, \text{unlocked}) \]

\[ x_1 \mapsto_i n + 1 \\
\quad x_2 \mapsto_s n \\
\quad \text{isLock}(l, \text{unlocked}) \]

\[
\text{acquire}(l); \\
\text{let } v = !x_2 \text{ in } \\
x_2 \leftarrow v + 1; \\
\text{release}(l); \\
v
\]
Example

\[ \exists n. x_1 \mapsto_i n \ast \\
\quad x_2 \mapsto_s n \ast \\
\quad \text{isLock}(l, \text{unlocked}) \]

\[ x_1 \mapsto_i n + 1 \]
\[ x_2 \mapsto_s n \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[
\sim
\]

\[
\text{acquire}(l); \\
\text{let } v = !x_2 \text{ in } \\
x_2 \leftarrow v + 1; \\
\text{release}(l); \ v
\]
Example

\[ \exists n. x_1 \mapsto_i n \ast \]
\[ x_2 \mapsto_s n \ast \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ x_1 \mapsto_i n + 1 \]
\[ x_2 \mapsto_s n \]
\[ \text{isLock}(l, \text{locked}) \]

let \( v = !x_2 \) in
\[ x_2 \leftarrow v + 1; \]
\[ \text{release}(l); \ v \]
Example

\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ x_1 \mapsto_i n + 1 \]
\[ x_2 \mapsto_s n \]
\[ \text{isLock}(l, \text{locked}) \]

\[ \text{let } v = !x_2 \text{ in} \]
\[ x_2 \leftarrow v + 1; \]
\[ \text{release}(l); \ v \]
Example

\[ \exists n. \begin{array}{l}
x_1 \mapsto i \ n \\
x_2 \mapsto s \ n \\
isLock(l, \text{unlocked})
\end{array} \]

\[ \begin{array}{l}
x_1 \mapsto i \ n + 1 \\
x_2 \mapsto s \ n \\
isLock(l, \text{locked})
\end{array} \]

\[ \text{acquire}(l); \]

let \( v = !x_2 \)

\[ x_2 \leftarrow n + 1; \]

release(l); \( n \)
Example

$$\exists n. x_1 \mapsto_i n \ast$$
$$x_2 \mapsto_s n \ast$$
$$\text{isLock}(l, \text{unlocked})$$

$$x_1 \mapsto_i n + 1$$
$$x_2 \mapsto_s n$$
$$\text{isLock}(l, \text{locked})$$

$$\neg$$

$$x_2 \leftarrow n + 1;$$
$$\text{release}(l); \ n$$
Example

$\exists n. x_1 \mapsto_i n \ast$

$\quad x_2 \mapsto_s n \ast$

isLock($l$, unlocked)

$x_1 \mapsto_i n + 1$

$x_2 \mapsto_s n + 1$

isLock($l$, locked)

$\sim$

release($l$); $n$
Example

\[ \exists n. \ x_1 \mapsto_i n \ *
\]
\[ x_2 \mapsto_s n \ *
\]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ x_1 \mapsto_i n + 1 \]
\[ x_2 \mapsto_s n + 1 \]
\[ \text{isLock}(l, \text{locked}) \]

\[ \text{acquire}(l); \]
\[ \text{let} \ v = !x_2 \]
\[ x_2 \leftarrow n + 1 \]

\[ \text{release}(l); \]
Example

$$\exists n. x_1 \mapsto_i n *$$
$$x_2 \mapsto_s n *$$
$$\text{isLock}(l, \text{unlocked})$$

$$x_1 \mapsto_i n + 1$$
$$x_2 \mapsto_s n + 1$$
$$\text{isLock}(l, \text{unlocked})$$
\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(l, \text{unlocked}) \]

\[ \therefore \]

\[ n \]
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 $e_1 \not\sim e_2 : \tau$ 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:** $e_1 \not\sim e_2 : \tau \implies e_1 \not\sim_{ctx} e_2 : \tau$
- **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) :

\[ P \ast (\exists a, \Psi a) \ast Q \rightarrow Q \ast \exists a, P \ast \Psi a. \]

**Proof.**
- iIntros "[H1 [H2 H3]]".
- iDestruct "H2" as (x) "H2".
- iSplitL "H3".
- iAssumption.
- iExists x.
  - iFrame.

Qed.
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.
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)
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.
   - iExists x.
   iFrame.
Qed.

The hypotheses for the left conjunct

* means: resources should be split
Lemma test \{A\} (P Q : iProp) (Ψ : A \rightarrow iProp) : P \ast (\exists a, \Psi a) \ast Q \rightarrow Q \ast \exists a, P \ast \Psi a.

Proof.
iIntros "[H1 [H2 H3]]".
iDestruct "H2" as (x) "H2".
iSplitL "H3".
- iAssumption.
- iExists x.
iFrame.
Qed.

2 subgoals
A : Type
P, Q : iProp
Ψ : A \rightarrow iProp
x : A
_________________________(1/2)
"H3" : Q
__________________________(2/2)
"H1" : P
"H2" : Ψ x
___________________________
∃ a : A, P \ast Ψ a

The hypotheses for the left conjunct
Lemma test \{A\} (P Q : iProp) (\Psi : A \to iProp) :
  P \ast (\exists a, \Psi a) \ast Q \rightarrow Q \ast \exists a, P \ast \Psi a.

Proof.
iIntros "[H1 [H2 H3]]".
iDestruct "H2" as (x) "H2".
iSplitL "H3".
- iAssumption.
- iExists x.
iFrame.

Qed.
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 \texttt{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
Lemma test \{PROP : bi\} \{A\}

\[(P \land Q : PROP) (\Psi : A \rightarrow PROP) : P \land \Box (\exists a, \Psi a) \rightarrow \exists a, \Psi a \land (P \land \Psi a)\].

Proof.

iIntros "[H1 #H2]."

iDestruct "H2" as (x) "H2".

iExists x.

iSplitL "H2".

- iAssumption.

- by iFrame.

Qed.
Lemma test {PROP : bi} {A}

(P Q : PROP) (Ψ : A → PROP) :
P ≡ □ (∃ a, Ψ a) −∗ ∃ a, Ψ a * (P * Ψ a).

Proof.
Proof. Persistent modality

iIntros H1#H2.
iDestruct H2 as (a) H2.
iExists x.
iSplitL "H2".
- iAssumption.
- by iFrame.
Qed.
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.

Moves hypothesis to persistent context
**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 \to PROP) :
P \ast □ (\exists a, Ψ a) \to \exists a, Ψ a \ast (P \ast Ψ a).

**Proof.**

iIntros "[H1 #H2]."

iDestruct "H2" as (x) "H2".

iExists x.

iSplitL "H2".

- iAssumption

- iAssumption

Qed.

1 subgoal

PROP : bi

A : Type

P, Q : PROP

Ψ : A \to PROP

x : A

________________________(1/1)

"H2" : Ψ x

________________________□

"H1" : P

________________________* 

Ψ x \ast (P \ast Ψ x)

Do not need to split persistent context
Lemma test \{PROP : bi\} \{A\}

\((P \ Q : PROP) (\Psi : A \to PROP) : 
P \ast \Box (\exists a, \Psi a) \to \exists a, \Psi a \ast (P \ast \Psi a)\).

Proof.

\text{iIntros "[H1 \#H2]".}
\text{idestruct "H2" as (x) "H2".}
\text{iExists x.}
\text{iSplitL "H2".}

\text{\text{-- iAssumption.}}
\text{\text{-- by iFrame.}}

Qed.
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
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 $e \preceq e': \tau$, which expresses that a program $e$ refines a program $e'$ at type $\tau$. 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, and show that these rules make it simpler to carry out refinement proofs.

Figure 1. Two concurrent counter implementations.
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!**