Relational reasoning using concurrent separation logic

Robbert Krebbers\textsuperscript{1}

Delft University of Technology, The Netherlands

January 20, 2020 @ ADSL, New Orleans, USA

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

▶ Specifying programs

\[ \text{implementation} \prec_{\text{ctx}} \text{specification} \]
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
  \[ \text{implementation} \sim_{\text{ctx}} \text{specification} \]

- Optimized versions of data structures
  \[ \text{hash\_table} \sim_{\text{ctx}} \text{assoc\_list} \]

- Proving program transformations
  \[ \forall e_{\text{source}}. \, \text{compile}(e_{\text{source}}) \sim_{\text{ctx}} e_{\text{source}} \]
Language features that complicate refinements

-Mutable state

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

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

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

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

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

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

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

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

"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:

\[ e_1 \sim_{ctx} e_2 : \tau \overset{\triangleq}{=} \forall (C : \tau \rightarrow \text{int}). \forall v. \ C[e_1] \downarrow v \implies C[e_2] \downarrow v \]

"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**:

\[ e_1 \lesssim e_2 : \tau \]

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

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

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

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

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

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

Symbolic execution rules

\[
\Delta \models K[e'_1] \preceq e_2 : \tau \quad e_1 \rightarrow_{\text{pure}} e'_1 \quad \Rightarrow \\
\Delta \models K[e_1] \preceq e_2 : \tau
\]
Some rules for pure programs

Symbolic execution rules

\[
\begin{align*}
\Delta \models K[e'_1] \preceq e_2 : \tau & \quad e_1 \rightarrow_{\text{pure}} e'_1 \\
\Delta \models K[e_1] \preceq e_2 : \tau
\end{align*}
\]

\[
\begin{align*}
\Delta \models e_1 \preceq K[e'_2] : \tau & \quad e_2 \rightarrow_{\text{pure}} e'_2 \\
\Delta \models e_1 \preceq K[e_2] : \tau
\end{align*}
\]
Some rules for pure programs

Structural rules

$$\Delta \models e_1 \not\sim e_2 : \tau \quad \Delta \models e'_1 \not\sim e'_2 : \tau'$$

$$\Delta \models (e_1, e'_1) \not\sim (e_2, e'_2) : \tau \times \tau'$$
Some rules for pure programs

Structural rules

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

\[
\exists (R : Val \times Val \rightarrow Prop). \quad [\alpha := R], \Delta \models e_1 \preceq e_2 : \tau
\]

\[
\Delta \models \text{pack}(e_1) \preceq \text{pack}(e_2) : \exists \alpha. \tau
\]
Some rules for pure programs

Structural rules

\[
\begin{align*}
\Delta \vdash e_1 \not\sim e_2 : \tau \quad &\quad \Delta \vdash e'_1 \not\sim e'_2 : \tau' \\
\Delta \vdash (e_1, e'_1) \not\sim (e_2, e'_2) : \tau \times \tau' \\
\exists (R : Val \times Val \to Prop). \quad &\quad [\alpha := R], \Delta \vdash e_1 \not\sim e_2 : \tau \\
\Delta \vdash \text{pack}(e_1) \not\sim \text{pack}(e_2) : \exists \alpha. \tau \\
\llbracket \tau \rrbracket \Delta (v_1, v_2) \quad &\quad \Box \left( \forall v_1 v_2. \llbracket \tau \rrbracket \Delta (v_1, v_2) \vdash^* \right. \\
\Delta \vdash v_1 \not\sim v_2 : \tau \quad &\quad \left. \Delta \vdash e_1[v_1/x_1] \not\sim e_2[v_2/x_2] : \sigma \right) \\
\Delta \vdash \lambda x_1. e_1 \not\sim \lambda x_2. e_2 : \tau \to \sigma
\end{align*}
\]
Example

A bit interface:

\[ \text{bitT} \triangleq \exists \alpha. \; \alpha \times (\alpha \to \alpha) \times (\alpha \to \text{bool}) \]

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

A bit interface:

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

- 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:

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

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

Two implementations:

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

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

Refinement (and vice versa):

\[
\text{bit_bool} \preceq \text{bit_nat} : \text{bitT}
\]
Proof of the refinement

\[
\text{pack(} \text{true, } (\lambda b. \neg b), (\lambda b. b)\text{)} \vDash \\
\text{pack(} 1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)\text{)}
\]

\[
\exists \alpha. \alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{bool})
\]
Proof of the refinement

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

\[\sim\]

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

\[\vdash\]

\[\exists \alpha. \alpha \times (\alpha \to \alpha) \times (\alpha \to \text{bool})\]

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

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

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

\[
\overset{\sim}{\sim}
\]

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

:\

\[
\exists \alpha. \alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{bool})
\]

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

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

\[
\left(\text{true}, (\lambda b. \neg b), (\lambda b. b)\right) \rightsquigarrow \\
\left(1, (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0), (\lambda n. n = 1)\right)
:\alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{bool})
\]
Proof of the refinement

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

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

\(\approx\)

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

\[
\alpha \times (\alpha \to \alpha) \times (\alpha \to \text{bool})
\]

Use structural rule for products
Proof of the refinement

\[ \alpha := R \models -\text{true} : \alpha \]

\[ \alpha \models -\text{false} : \alpha \]

\[ (\lambda b. \neg b) \vartriangleleft (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 0) : \alpha \rightarrow \alpha \]

\[ (\lambda b. b) \vartriangleleft (\lambda n. n = 1) : \alpha \rightarrow \text{bool} \]
Proof of the refinement

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

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

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

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

(by def of \( R \))
Proof of the refinement

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

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

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

\[ (\lambda b. b) \prec (\lambda n. n = 1) : \alpha \rightarrow \text{bool} \]

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

\[ \neg \text{true} \prec \text{if } 1 = 0 \text{ then } 1 \text{ else } 0 : \alpha \]

\[ \neg \text{false} \prec \text{if } 0 = 0 \text{ then } 1 \text{ else } 0 : \alpha \]
Proof of the refinement

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

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

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

\[ (\lambda b. b) \preceq (\lambda n. n = 1) : \alpha \rightarrow \text{bool} \quad \text{(\(\lambda\)-rule + symb. exec.)} \]
Proof of the refinement

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

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

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

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

Example:
- $\ell_1 \mapsto_i 4 \rightarrow^* \ell_2 \mapsto_s 0 \rightarrow^* (!\ell_1) \preceq (\ell_2 \leftarrow 4; !\ell_2) : \mathbb{int}$
Mutable state and separation logic for refinements

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

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

Example:

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

Symbolic execution

\[
\ell_1 \mapsto_i ν_1 \quad \text{Δ } \models \quad K[()] \preceq e_2 : τ \\
\Delta \models K[\ell_1 \leftarrow ν_1] \preceq e_2 : τ
\]
Some rules for mutable state

Symbolic execution

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

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

Symbolic execution

\[
\begin{align*}
\ell_1 & \mapsto i - \quad * \\
\Delta & \models K[() \bowtie e_2 : \tau] \\
\Delta & \models K[\ell_1 \leftarrow v_1] \bowtie e_2 : \tau \\
\end{align*}
\]

\[
\begin{align*}
\ell_2 & \mapsto s - \quad * \\
\Delta & \models e_1 \bowtie K[() : \tau] \\
\Delta & \models e_1 \bowtie K[\ell_2 \leftarrow v_2] : \tau \\
\end{align*}
\]

\[
\begin{align*}
\forall \ell_1. \ell_1 & \mapsto i v_1 \quad \Delta \models K[l_1] \bowtie e_2 : \tau \\
\Delta & \models K[\text{ref}(v_1)] \bowtie e_2 : \tau \\
\end{align*}
\]

\[
\begin{align*}
\forall \ell_2. \ell_2 & \mapsto s v_2 \quad \Delta \models e \bowtie K[l_2] : \tau \\
\Delta & \models e_1 \bowtie K[\text{ref}(v_2)] : \tau \\
\end{align*}
\]
Reasoning about higher-order functions and concurrency
State encapsulation

Modules with **encapsulated state**:

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

The reference can only be used in the closure

Simple example:

\[
\text{counter} \equiv (\lambda ()\text{. let } x = \text{ref}(1)\text{ in } (\lambda ()\text{. FAA}(x, 1))):
\]

\[\text{unit} \to (\text{unit} \to \text{int})\]

\[\text{counter}()\] constructs an instance \(c\): \(\text{unit} \to \text{int}\) of the counter module

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

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

Modules with *encapsulated state:*

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

The reference can only be used in the closure

Simple example:

\[
counter \triangleq (\lambda(). \text{let } x = \text{ref}(1) \text{ in } (\lambda(). \text{FAA}(x, 1))) : \text{unit} \rightarrow (\text{unit} \rightarrow \text{int})
\]

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

Modules with **encapsulated state**:

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

Modules with **encapsulated state**:

```
let x = ref(e) 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
The problem

Modules with **encapsulated state**:

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

**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**:

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

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 \( x \mapsto v \)

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 "P \text{ holds without assuming exclusive resources}"

**Examples:**

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

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

$$\begin{align*}
\Box \left( \forall v_1, v_2. [\tau] \Delta(\nu_1, \nu_2) \rightarrow^* \\
\Delta \models e_1[v_1/x_1] \not\leq e_2[v_2/x_2] : \sigma \right)
\Rightarrow \\
\Delta \models \lambda x_1. e_1 \not\leq \lambda x_2. e_2 : \tau \rightarrow \sigma
\end{align*}$$

Due to $\Box$, the resource $x : s_0$ cannot be used to prove the closure.
The □ modality makes sure no exclusive resources can escape into closures:

$$
\begin{align*}
\Box \left( \forall v_1, v_2. \; \Delta \vdash (v_1, v_2) \to^* \\
\Delta \models e_1[v_1/x_1] \preceq e_2[v_2/x_2] : \sigma \\
\Delta \models \lambda x_1. e_1 \preceq \lambda x_2. e_2 : \tau \to \sigma
\end{align*}
$$

Prohibits “wrong” refinements, for example:

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

Due to □, 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:

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

\sim

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

\[
\left( \lambda(). \text{acquire}(l); \right.
\]

\[
\left( \text{let } v = !x \text{ in } \right.
\]

\[
 x \leftarrow v + 1; \right.
\]

\[
\text{release}(l); v
\]
Iris-style Invariants

The invariant connective $R$
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: $\boxed{R} \vdash \Box \overline{R}$.
- ... thus can be used to prove closures.
Iris-style Invariants

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

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

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

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

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

\[
\begin{align*}
\text{let } x &= \text{ref}(1) \text{ in } (\lambda(). \text{FAA}(x, 1)) \\
\end{align*}
\]

\[
\begin{align*}
\Rightarrow \\
\text{let } x &= \text{ref}(1), l = \text{newlock}() \text{ in } \\
&\quad (\lambda(). \text{acquire}(l); \\
&\quad \text{let } v = !x \text{ in } \\
&\quad x \leftarrow v + 1; \\
&\quad \text{release}(l); v)
\end{align*}
\]
Example

∃ \text{ } n . \ x_1 \mapsto i \ 1

(\lambda(). \ FAA(x_1,1)) \sim \bowtie

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

\[ x_1 \mapsto_i 1 \]

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

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

\[
\begin{aligned}
\exists n. \quad x_1 \mapsto_i 1 \\
\quad x_2 \mapsto_s 1
\end{aligned}
\]

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

\[
\approx
\]

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

\[
\begin{align*}
&\exists n.
x_1 \mapsto_i 1 \\
&x_2 \mapsto_s 1
\end{align*}
\]

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

\[
\leadsto
\]

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

$\exists n. x_1 \mapsto i 1$

$\exists n. x_2 \mapsto s 1$

isLock($l$, unlocked)

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

\[ \leadsto \]

$(\lambda(). \text{acquire}(l);$

\hspace{1cm} let $v = !x_2$ in

\hspace{1cm} $x_2 \leftarrow v + 1;$

\hspace{1cm} release($l$); $v)$
Example

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

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

\[ \sim \]

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

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

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

\[ \sim \]

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

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

\[
\begin{align*}
&\text{FAA}(x_1, 1) \\
&\sim \\
&\text{acquire}(l); \\
&\text{let } v = !x_2 \text{ in} \\
&x_2 \leftarrow v + 1; \\
&\text{release}(l); \ v
\end{align*}
\]
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

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

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

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

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

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

\[ n \]

\[ \sim \]

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

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

\[ x_2 \leftarrow v + 1; \]

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

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

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

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

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

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

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

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

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

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

\[n\]
\[\forall \]

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

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

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

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

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

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

\[ \sim \]

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

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

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

\[\land\]

\[n\]

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

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

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

\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(l, \text{unlocked}) \]
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 $e_1 \prec e_2 : \tau$ 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: $e_1 \prec e_2 : \tau \implies e_1 \prec_{ctx} e_2 : \tau$
- Actual program refinements: concurrent data structures, and examples from the logical relations literature
Need to reason in separation logic!
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.
\begin{itemize}
\item iIntros "[H1 [H2 H3]]".
\item iDestruct "H2" as (x) "H2".
\item iSplitL "H3".
\item - iAssumption.
\item - iExists x.
\item - iFrame.
\end{itemize}
Qed.
Lemma test \{A\} (P \land Q : \text{iProp}) (\Psi : A \to \text{iProp}) :

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

Proof.

\begin{itemize}
  \item iIntros \"[H1 [H2 H3]\".
  \item iDestruct \"H2\" as (x) \"H2\".
  \item iSplitL \"H3\".
  \item iAssumption.
  \item iExists x.
  \item iFrame.
\end{itemize}

 Qed.
Lemma test \{A\} (P Q : iProp) (Ψ : A → iProp) :

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

Proof.

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

1 subgoal
A : Type
P, Q : iProp
Ψ : A → iProp
x : A
_________________________ (1/1)
"H1" : P
"H2" : Ψ x
"H3" : Q
_________________________∗
Q \ast (\exists a : A, P \ast Ψ a)
Iris Proof Mode (IPM) [Krebbers et al.; POPL'17]

Lemma test \{A\} (P Q : iProp) (\Psi : 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.

1 subgoal
A : Type
P, Q : iProp
\Psi : A \rightarrow iProp
x : A

"H1" : P
"H2" : \Psi x
"H3" : Q

Q \ast (\exists a : A, P \ast \Psi a)

* means: resources should be split
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 → iProp) :
  P * (\exists a, Ψ a) * Q →* Q * \exists 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

| 2 subgoals |
|---|---|
| A : Type |
| P, Q : iProp |
| Ψ : A → iProp |
| x : A |

| (1/2) |
| "H3" : Q |

| (2/2) |
| "H1" : P |

| "H2" : Ψ x |

| \exists a : A, P * Ψ a |
Lemma test \{A\} (P \ Q : iProp) (\Psi : A \rightarrow iProp) :
   P * (\exists \ a, \Psi \ a) * Q \rightarrow Q * \exists \ a, P * \Psi \ a.

Proof.
iIntros "[H1 \ [H2 H3]]".
iDestruct "H2" as (x) "H2".
iSplitL "H3".
- iAssumption.
- iExists x.
  iFrame.
Qed.
Lemma test $\{A\} (P \land Q : iProp) (\psi : A \rightarrow iProp)$:
$P \leftrightarrow (\exists a, \psi a) \times Q \leftrightarrow Q \times \exists a, P \leftrightarrow \psi a$.

Proof.
- iIntros "[H1 [H2 H3]]".
- by iFrame.

Qed.

We can also solve this lemma automatically.
The ReLoC rules are just lemmas that can be \texttt{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.
Ongoing work:

Proving security properties using relational reasoning in separation logic
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 $L \sqsubseteq H$

\[ \vdash l_i : \text{ref int}^L \quad \vdash h_i : \text{ref int}^H \]

\[ \vdash e : \text{ref int}^\chi \quad \vdash e' : \text{int}^\xi \quad \xi \sqsubseteq \chi \]
\[ \vdash e \leftarrow e' : \text{unit} \]

\[ \vdash e + e' : \text{int}^{\chi \sqcup \xi} \]
Type systems for non-interference

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

\[
\begin{align*}
\vdash l_i : ref\ int^L & \quad \vdash h_i : ref\ int^H \\
\vdash e : ref\ int^\chi & \quad \vdash e' : int^\xi & \xi \sqsubseteq \chi
\vdash e \leftarrow e' : unit & \quad \vdash e + e' : int^{\chi \sqcup \xi}
\end{align*}
\]

Example:

\[
\begin{align*}
\vdash l_1 : ref\ int^L & \quad \vdash !h_1 : int^H & \vdash 1 : int^L
\vdash l_1 : ref\ int^L & \quad \vdash !h_1 + 1 : int^H & \quad H \sqsubseteq L
\vdash l_1 \leftarrow !h_1 + 1 : unit
\end{align*}
\]
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

```
let r = \{ data = ref(secret); is_classified = ref(true) \} in

while true do
  if \neg \! r.is_classified then
    out ← \! r.data
  else ()
  r.data ← 0;
  r.is_classified ← false

The classification of $r.data$ depends on the run-time value $r.is_classified$
```
Example: value-dependent classification

```
let r = \{ data = ref(secret); 
          is_classified = ref(true) \} in

while true do
  if \neg ! r.is_classified then out ← ! r.data 
  else ();
  r.data ← 0;
  r.is_classified ← false
```

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

```
let r = \{ data = ref(secret); 
   is_classified = ref(true) \} in

while true do
  if \neg !r.is_classified then
    out ← !r.data
  else
    ();

  r.data ← 0;
  r.is_classified ← false
```

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: 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: 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:

\[ \text{dwp } e_1 \& e_2 \{ \Phi \} \]
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 \( \Phi \)
Double weakest precondition

The basic component of SeLoC:

\[
dwp e_1 \& e_2 \{ \Phi \}
\]

- Any two runs of the \( e_1 \) and \( e_2 \) are in a bisimulation and their results satisfy \( \Phi \)
- \( e_1 \) and \( e_2 \) have different secret data, but must produce the same public output
Double weakest precondition

The basic component of SeLoC:

dwp e_1 & e_2 \{ \Phi \}

- Any two runs of the $e_1$ and $e_2$ are in a bisimulation and their results satisfy $\Phi$
- $e_1$ and $e_2$ have different secret data, but must produce the same public output
- Left-hand side and right-hand side resources: $\ell_1 \mapsto_L v_1$ and $\ell_2 \mapsto_R v_2$
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 \( \Phi \)
- \( e_1 \) and \( e_2 \) have different secret data, but must produce the same public output
- Left-hand side and right-hand side resources: \( \ell_1 \leftrightarrow_L v_1 \) and \( \ell_2 \leftrightarrow_R v_2 \)
- Rules are similar to ReLoC, but require execution in lock-step
Double weakest precondition

The basic component of SeLoC:

\[
\text{dwp } e_1 \land e_2 \{\Phi\}
\]

- Any two runs of the \(e_1\) and \(e_2\) are in a bisimulation and their results satisfy \(\Phi\)
- \(e_1\) and \(e_2\) have different secret data, but must produce the same public output
- Left-hand side and right-hand side resources: \(\ell_1 \rightarrow_L v_1\) and \(\ell_2 \rightarrow_R v_2\)
- Rules are similar to ReLoC, but require execution in lock-step
- Soundness statement:

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

\[
l_{\text{out}} \triangleq \exists v \in \mathbb{Z}. \text{out} \leftrightarrow_L v \ast \text{out} \leftrightarrow_R v
\]
Proof of the example: value-dependent classification

$$\text{let } r = \begin{cases} \text{data} = \text{ref}(\text{secret}); \\ \text{is}\_\text{classified} = \text{ref}(\text{true}) \end{cases} \text{ in}$$

while true do
  if $$\neg \! r.\text{is}\_\text{classified}$$ then
    $$\text{out} \leftarrow \! r.\text{data}$$
  else
    $$r.\text{data} \leftarrow 0;$$
    $$r.\text{is}\_\text{classified} \leftarrow \text{false}$$
  end if

Classified $\rightarrow$ Intermediate $\rightarrow$ Declassified
Proof of the example: value-dependent classification

\[
\text{let } r = \begin{cases} 
\text{data} = \text{ref} (\text{secret}); \\
\text{is\_classified} = \text{ref} (\text{true}) 
\end{cases} \quad \text{in}
\]

\[
\text{while true do}
\]

\[
\begin{align*}
\text{if } &\neg! r.\text{is\_classified} \\
\text{then } &\text{out} \leftarrow ! r.\text{data} \\
\text{else } &();
\end{align*}
\]

\[
\begin{align*}
&\text{if } \neg! r.\text{is\_classified} \\
&\text{then } \text{out} \leftarrow ! r.\text{data} \\
&\text{else } (); \\
&\quad r.\text{data} \leftarrow 0; \\
&\quad r.\text{is\_classified} \leftarrow \text{false}
\end{align*}
\]
Proof of the example: value-dependent classification

```
let r = \{ data = ref(secret); is_classified = ref(true) \} in

while true do
  if \neg ! r.is_classified
  then out ← ! r.data
  else ();
  r.data ← 0;
  r.is_classified ← false
```

Classified → Intermediate → Declassified
Proof of the example: value-dependent classification

\[
\text{let } r = \begin{cases} 
\text{data} = \text{ref}(\text{secret}) ; \\
\text{is} \_ \text{classified} = \text{ref}(\text{true}) 
\end{cases} \text{ in }
\]

while true do
\[
\text{if } \neg ! \ r. \text{is} \_ \text{classified} \text{ then } \text{out} \leftarrow ! r.\text{data} \else \text{();} \text{ end if}
\]
\[
r.\text{data} \leftarrow 0;
\]
\[
r.\text{is} \_ \text{classified} \leftarrow \text{false}
\]
Proof of the example: value-dependent classification

\[(\text{in\_state(Classified)} \land \exists i_1, i_2. r_1.\text{is\_classified} \mapsto_L \text{true} \land r_2.\text{is\_classified} \mapsto_R \text{true} \land r_1.\text{data} \mapsto_L i_1 \land r_2.\text{data} \mapsto_R i_2)\]

\[\lor (\text{in\_state(Intermediate)} \land \exists i. r_1.\text{is\_classified} \mapsto_L \text{true} \land r_2.\text{is\_classified} \mapsto_R \text{true} \land r_1.\text{data} \mapsto_L i \land r_2.\text{data} \mapsto_R i)\]

\[\lor (\text{in\_state(Declassified)} \land \exists i. r_1.\text{is\_classified} \mapsto_L \text{false} \land r_2.\text{is\_classified} \mapsto_R \text{false} \land r_1.\text{data} \mapsto_L i \land r_2.\text{data} \mapsto_R i)\]
Conclusions
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 ReLoC, a relational logic for proving non-interference of programs in a language with higher-order state, fine-grained concurrency, polymorphism and recursive types. The core of our logic in a judgement $C \nRightarrow e \xrightarrow{r} e'$, which expresses that a program $e$ refines a program $e'$ of type $r$. In contrast to other 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 relational specifications in ReLoC, allowing for more modular proofs.

ReLoC is built on top of the expressive concurrent separation logic lits, allowing us to leverage features of lits 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 how to formalise and prove relational specifications in ReLoC, allowing for more modular proofs.

The paper is available at: https://doi.org/10.1145/3209108.3209174

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

ACM Reference Format:

1 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—for example, to support dynamically allocated references and higher-order functions. 131, 131, 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], [9], [10]. Such systems have the benefit of providing strong automaticity (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...

ReLoC: non-interference
Thank you!

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

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