ReLoC: A mechanised relational logic for fine-grained concurrency

Dan Frumin\textsuperscript{1} Robbert Krebbers\textsuperscript{2} Lars Birkedal\textsuperscript{3}
LICS 2018, July 9, 2018

\textsuperscript{1}Radboud University, Nijmegen, The Netherlands
\textsuperscript{2}Delft University of Technology, The Netherlands
\textsuperscript{3}Aarhus University, Denmark
**ReLoC**: mechanised separation logic for interactive refinement proofs of fine-grained concurrent programs.
**ReLoC**: mechanised separation logic for interactive **refinement** proofs of fine-grained concurrent programs.

- **Contextual refinement**: notion of program refinement
  
  E.g.: $\text{or}(e_1, e_2) \preceq_{\text{ctx}} \text{or}(e_2, e_1)$. 

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

- **Contextual refinement**: notion of program refinement
  E.g.: $\text{or}(e_1, e_2) \prec_{\text{ctx}} \text{or}(e_2, e_1)$.

- **Fine-grained concurrency**: programs use low-level synchronisation primitives for more granular parallelism.
**ReLoC**: mechanised separation logic for interactive refinement proofs of fine-grained concurrent programs.

- **Contextual refinement**: notion of program refinement
  E.g.: \( \text{or}(e_1, e_2) \preceq_{\text{ctx}} \text{or}(e_2, e_1) \).

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

- **Mechanised**: proven sound in Coq.
**ReLoC**: mechanised separation logic for interactive refinement proofs of fine-grained concurrent programs.

- **Contextual refinement**: notion of program refinement
  E.g.: \( \text{or}(e_1, e_2) \prec_{\text{ctx}} \text{or}(e_2, e_1) \).

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

- **Mechanised**: proven sound in Coq.

- Coq machinery for high level interactive proofs in the logic.
Refinements of concurrent programs

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

\[ e_1 \sim_{ctx} e_2 \triangleq \forall C, v. \ C[e_1] \downarrow v \implies C[e_2] \downarrow v \]

“Any behaviour of a (well-typed) client using \( e_1 \) can be matched by a behaviour of the same client using \( e_2 \)”
Refinements of concurrent programs

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

\[ e_1 \preceq_{\text{ctx}} e_2 \triangleq \forall C, \nu. \ C[e_1] \downarrow \nu \implies C[e_2] \downarrow \nu \]

“Any behaviour of a (well-typed) client using \( e_1 \) can be matched by a behaviour of the same client using \( e_2 \)”

- Applications: optimised versions of data structures; proving linearisability; proving program transformations.
- Example: `lock_free_data_structure \preceq_{\text{ctx}} atomic_data_structure`. 
Refinements of concurrent programs

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

\[ e_1 \sqsubseteq_{ctx} e_2 \triangleq \forall C, v. \ C[e_1] \downarrow v \implies C[e_2] \downarrow v \]

“Any behaviour of a (well-typed) client using \( e_1 \) can be matched by a behaviour of the same client using \( e_2 \)”

- Applications: optimised versions of data structures; proving linearisability; proving program transformations.
- Example: lock_free_data_structure \( \sqsubseteq_{ctx} \) atomic_data_structure.
Our proposed solution

Prove the refinements in the style of concurrent separation logic!

Instead of Hoare triples $\{P\} e \{Q\}$ we have refinement judgements $e_1 \preceq e_2 : \tau$.

- Soundness: $\vdash e_1 \preceq e_2 : \tau \implies e_1 \preceq_{ctx} e_2 : \tau$
- Proofs by symbolic execution.
- Modular and conditional specifications.
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

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

\[ \mid P \ast Q \mid P \rightarrow Q \mid \ell \mapsto_i v \mid \ell \mapsto_s v \]

\[ \mid (e_1 \preceq e_2 : \tau) \mid \ldots \]

- 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: allows conditional refinements

  - \( \ell_1 \mapsto_i v \ast e_1 \preceq e_2 : \tau \);
  - \( e_1 \preceq e_2 : 1 \rightarrow \tau \ast t_1(e_1) \preceq e_2(); e_2() : \tau \);
Example ReLoC rules

Structural rules

\[
\frac{e_1 \preceq e_2 : \tau \quad * \quad t_1 \preceq t_2 : \tau'}{(e_1, t_1) \preceq (e_2, t_2) : \tau \times \tau'}
\]
Example ReLoC rules

Structural rules

$$
\frac{e_1 \preceq e_2 : \tau \quad \ast \quad t_1 \preceq t_2 : \tau'}{(e_1, t_1) \preceq (e_2, t_2) : \tau \times \tau'} \ast
$$

Symbolic execution

$$
\frac{\ell \mapsto_s v \quad \ast \quad (\ell \mapsto_s v_2 \ast e_1 \preceq K[()] : \tau)}{e_1 \preceq K[\ell \leftarrow v_2] : \tau} \ast
$$

$$
\frac{\ell \mapsto_i v \quad \ast \quad (\ell \mapsto_i v_2 \ast K[()] \preceq e_2 : \tau)}{K[\ell \leftarrow v_2] \preceq e_2 : \tau} \ast
$$
What about concurrency?

Problem
Structural & symbolic execution rules are only sufficient when you do not have shared resources ("standard" separation logic).

Solution
For shared resources we require mechanisms for reflecting this in the logic: invariants and ghost state (concurrent separation logic).

ReLoC is built on top of an expressive CSL – Iris – borrowing the infrastructure for resource sharing.
let \( x = \text{ref}(1) \) in (\( \lambda () \). FAI\( x \))

\[ \sim \]

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

\[ x_1 \mapsto_i 1 \]

(\lambda(). \text{FAI}(x_1))

\sim \text{let } x = \text{ref}(1), \ell = \text{newlock} () \text{ in }

(\lambda(). \text{acquire}(\ell);

\text{let } v = ! x \text{ in }

x \leftarrow v + 1;

\text{release}(\ell); v)
\[ \exists n. \ x_1 \mapsto i 1 \quad x_2 \mapsto s 1 \]

\[ (\lambda(). \text{FAI}(x_1)) \]

\[ \text{let } \ell = \text{newlock } () \text{ in } \]
\[ (\lambda(). \text{acquire}(\ell); \]
\[ \text{let } \nu = !x_2 \text{ in } \]
\[ x_2 \leftarrow \nu + 1; \]
\[ \text{release}(\ell); \nu) \]
test

\[\begin{align*}
x_1 & \mapsto i 1 \\
x_2 & \mapsto s 1 \\
\text{isLock}(\ell, \text{unlocked})
\end{align*}\]

\[(\lambda(). \text{FAI}(x_1)) \Downarrow (\lambda(). \text{acquire}(\ell); \text{let } v = !x_2 \text{ in } x_2 \leftarrow v + 1; \text{release}(\ell); v)\]
∃n.
    x₁ \mapsto_i n
    x₂ \mapsto_s n
    isLock(ℓ, unlocked)

(\lambda(). FAI(x₁))

\sim

(\lambda(). acquire(ℓ);
    \text{let } v = !x₂ \text{ in }
    x₂ \leftarrow v + 1;
    \text{release}(ℓ); v)
\[ \exists n. x_1 \mapsto_i n * \]
\[ x_2 \mapsto_s n * \]
\[ \text{isLock}(\ell, \text{unlocked}) \]

---

test

\[ (\lambda(). \text{FAI}(x_1)) \]

\[ \sim \]

\[ (\lambda(). \text{acquire}(\ell); \]
\[ \text{let } v = !x_2 \text{ in} \]
\[ x_2 \leftarrow v + 1; \]
\[ \text{release}(\ell); v ) \]
∃n. x₁ ↦i n *
    x₂ ↦s n *
    isLock(ℓ, unlocked)

FAI(x₁)

∀

acquire(ℓ);
let ν = !x₂ in
x₂ ← ν + 1;
release(ℓ); ν
\[ \exists n. x_1 \mapsto_i n \ast \]
\[ x_2 \mapsto_s n \ast \]
\[ \text{isLock}(\ell, \text{unlocked}) \]

test

\[ \text{FAI}(x_1) \Rightarrow \]

\[ \text{acquire}(\ell); \]
\[ \text{let } v = !x_2 \text{ in} \]
\[ x_2 \leftarrow v + 1; \]
\[ \text{release}(\ell); v \]
\[\exists n. x_1 \mapsto i n *\]
\[x_2 \mapsto s n *\]
\[\text{isLock}(\ell, \text{unlocked})\]

\text{test}

\[x_1 \mapsto i n\]
\[x_2 \mapsto s n\]
\[\text{isLock}(\ell, \text{unlocked})\]

\text{FAI}(x_1)

\[\sim\]

\text{acquire}(\ell);\]
\text{let } v = !x_2 \text{ in}
\text{x}_2 \leftarrow v + 1;
\text{release}(\ell); v
\[ \exists n \cdot x_1 \mapsto_i n \ast \]
\[ x_2 \mapsto_s n \ast \]
\[ \text{isLock}(\ell, \text{unlocked}) \]

\[ \text{test} \]

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

\[ n \]

\[ \sim \]

\[ \text{acquire}(\ell); \]
\[ \text{let } v = ! x_2 \text{ in} \]
\[ x_2 \leftarrow v + 1; \]
\[ \text{release}(\ell); v \]
\[\exists n. x_1 \mapsto_i n \ast \]
\[x_2 \mapsto_s n \ast \]
\[\text{isLock}(\ell, \text{unlocked})\]

\[\text{test}\]

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

\[
\begin{align*}
\text{let } v &= !x_2 \\
&\text{x}_2 \leftarrow v + 1; \\
&\text{release(}\ell\text{); } v
\end{align*}
\]
∃n. \( x_1 \mapsto_i n \neq \)
\[ x_2 \mapsto_s n \neq \]
\[ \text{isLock}(\ell, \text{unlocked}) \]

test

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

\[ n \neq \]

\[ x_2 \leftarrow n + 1; \]
\[ \text{release}(\ell); n \neq \]
\[ \exists n. x_1 \mapsto_i n \ast \]
\[ x_2 \mapsto_s n \ast \]
\[ \text{isLock}(\ell, \text{unlocked}) \]

\[ \text{test} \]

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

\[ \text{release}(\ell); n \]
\exists n. x_1 \mapsto_i n * \\
    \quad x_2 \mapsto_s n * \\
    \quad \text{isLock}(\ell, \text{unlocked})

\text{test}

\quad x_1 \mapsto_i n + 1 \\
\quad x_2 \mapsto_s n + 1 \\
\quad \text{isLock}(\ell, \text{unlocked})
\[
\exists n. x_1 \rightarrow_i n * \\
\quad x_2 \rightarrow_s n * \\
\quad \text{isLock}(\ell, \text{unlocked})
\]

test
- ReLoC provides rules allowing this kind of simulation reasoning, formally.
- The example can be done in ReLoC 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-increment (FAI) function is not a physically atomic instruction.
- What kind of specification can we give to FAI as a compound program?
Logically atomic relational specifications

Problem

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

Our solution

Relational version of TaDA-style logically atomic triples in ReLoC.
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.

https://cs.ru.nl/~dfrumin/reloc/