ReLoC: A mechanised relational logic for fine-grained concurrency

Dan Frumin[1] Robbert Krebbers[2] Lars Birkedal[3]

LICS 2018, July 9, 2018

[1]Radboud University, Nijmegen, The Netherlands

[2]Delft University of Technology, The Netherlands

[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.: $\mathtt{or}(\mathtt{e}_1, \mathtt{e}_2) \precsim_{ctx} \mathtt{or}(\mathtt{e}_2, \mathtt{e}_1)$.

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

- **Contextual refinement**: notion of program refinement
  E.g.: $\texttt{or}(\texttt{e}_1, \texttt{e}_2) \precsim_{ctx} \texttt{or}(\texttt{e}_2, \texttt{e}_1)$.
- **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.: $\mathtt{or}(\mathtt{e}_1, \mathtt{e}_2) \precsim_{ctx} \mathtt{or}(\mathtt{e}_2, \mathtt{e}_1)$.
- **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.: $\mathtt{or}(\mathtt{e}_1, \mathtt{e}_2) \precsim_{ctx} \mathtt{or}(\mathtt{e}_2, \mathtt{e}_1)$.
- **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:

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

"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:

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

"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` $\precsim_{ctx}$ `atomic_data_structure`.

Refinements of concurrent programs

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

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

Quantification over all clients

"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` $\precsim_{ctx}$ `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 $e_1 \precsim e_2 : \tau$.

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

ReLoC: (simplified) grammar

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

ReLoC: (simplified) grammar

$$P, Q \in \mathrm{Prop} ::= \forall x.\, P \mid \exists x.\, P \mid P \vee Q \mid \ \ldots$$
$$\mid \quad P * Q \quad \mid \quad P \twoheadrightarrow Q \quad \mid \quad \ell \mapsto_{\mathsf{i}} v \quad \mid \quad \ell \mapsto_{\mathsf{s}} v$$

- Separation logic for handling mutable state;
  - $\ell \mapsto_{\mathsf{i}} v$ for the left-hand side (implementation);
  - $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side (specification);

ReLoC: (simplified) grammar

$$P, Q \in \text{Prop} ::= \forall x.\, P \mid \exists x.\, P \mid P \vee Q \mid \ \ldots$$
$$\mid \quad P * Q \quad \mid \quad P \mathrel{-\!\!*} Q \quad \mid \quad \ell \mapsto_{\mathsf{i}} v \quad \mid \quad \ell \mapsto_{\mathsf{s}} v$$
$$\mid \quad (e_1 \precsim e_2 : \tau) \quad \mid \ \ldots$$

- Separation logic for handling mutable state;
  - $\ell \mapsto_{\mathsf{i}} v$ for the left-hand side (implementation);
  - $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side (specification);
- Logic with first-class refinement propositions: allows conditional refinements
  - $\ell_1 \mapsto_{\mathsf{i}} v \ \mathrel{-\!\!*}\ e_1 \precsim e_2 : \tau$;
  - $e_1 \precsim e_2 : \mathbf{1} \to \tau \ \mathrel{-\!\!*}\ t_1(e_1) \precsim e_2(); e_2() : \tau$;

Example ReLoC rules

Structural rules

$$\frac{e_1 \precsim e_2 : \tau \qquad * \qquad t_1 \precsim t_2 : \tau'}{(e_1, t_1) \precsim (e_2, t_2) : \tau \times \tau'} *$$

Example ReLoC rules

Structural rules

$$\frac{e_1 \precsim e_2 : \tau \qquad * \qquad t_1 \precsim t_2 : \tau'}{(e_1, t_1) \precsim (e_2, t_2) : \tau \times \tau'}*$$

Symbolic execution

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

$$\frac{\ell \mapsto_{\mathsf{i}} v \qquad * \qquad (\ell \mapsto_{\mathsf{i}} v_2 \;-\!\!*\; K[\,()\,] \precsim e_2 : \tau)}{K[\,\ell \leftarrow v_2\,] \precsim e_2 : \tau}*$$

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.

test

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

$$\precsim$$

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

test

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

$$(\lambda().\, \mathtt{FAI}(\mathtt{x}_1))$$

$$\precsim$$

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

test

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

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

$$(\lambda().\,\mathtt{FAI}(\mathtt{x}_1))$$

$$\precsim$$

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

---

test

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

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

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

$$(\lambda().\, \mathtt{FAI}(\mathtt{x}_1))$$

$$\precsim$$

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

$$\exists n.\; x_1 \mapsto_{\mathsf{i}} n \;*\; x_2 \mapsto_{\mathsf{s}} n \;*\; \mathsf{isLock}(\ell, \texttt{unlocked})$$

$$(\lambda ().\, \texttt{FAI}(\texttt{x}_1))$$

$$\precsim$$

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

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

---

test

$$(\lambda().\, \mathtt{FAI}(\mathtt{x}_1))$$

$$\precsim$$

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

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

---

test

$$\mathtt{FAI}(\mathtt{x}_1)$$

$$\precsim$$

$\mathsf{acquire}(\ell);$

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

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

$\mathsf{release}(\ell);\, v$

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

---

test

$$\mathtt{FAI}(\mathtt{x}_1)$$

$$\precsim$$

$\mathsf{acquire}(\ell);$

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

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

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

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

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

$$\mathtt{FAI}(\mathtt{x}_1)$$

$$\precsim$$

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

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

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

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

---

test

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

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

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

n

$\precsim$

$\mathsf{acquire}(\ell);$

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

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

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

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

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

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

test

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

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

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

$$n$$

$$\precsim$$

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

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

$$\mathsf{release}(\ell);\, v$$

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

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

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

$$n$$

$$\precsim$$

---

test

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

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

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

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

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

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

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

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

---

test

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

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

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

n

$\precsim$

$\mathsf{release}(\ell);\, n$

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

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

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

test

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

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

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

n

$\precsim$

n

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

test

n

$\precsim$

n

- 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/`