ReLoC: A mechanised relational logic for fine-grained concurrency

Dan Frumin Robbert Krebbers Lars Birkedal

LogSeminar, 9th of April

Aarhus University, Denmark

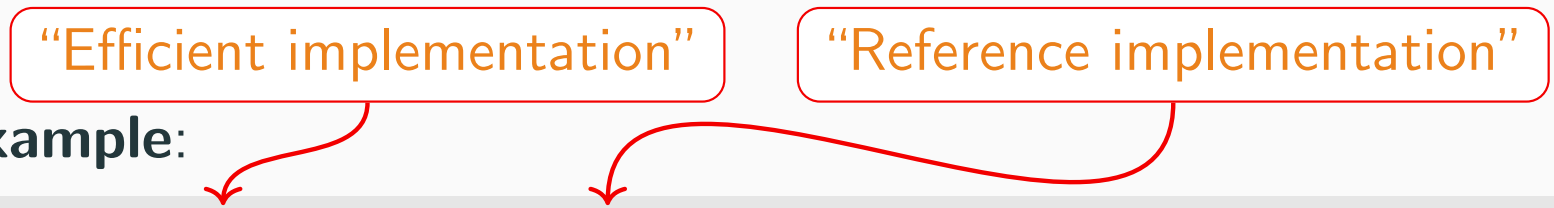
Overview and goals

Introduction

This talk: logic for interactive refinement proofs of stateful, concurrent, higher-order programs.

Features:

- Modular refinement proofs
- Using high-level tactics
- With a mechanised soundness proof



Example:

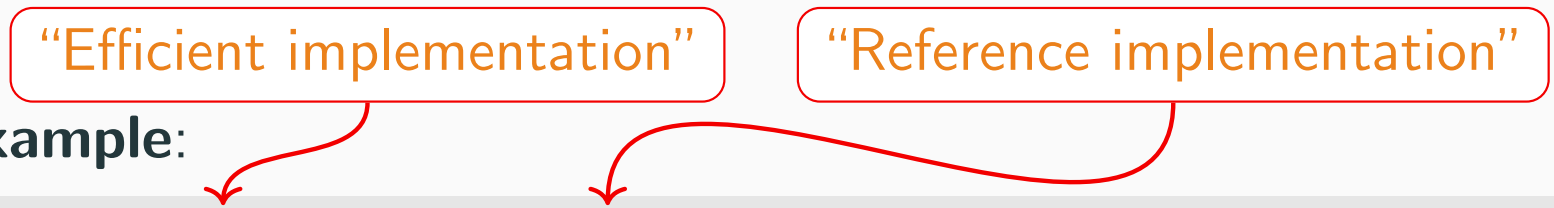
```
Γ ⊨ ticket_lock ≿ spin_lock
: ∃: (Unit → TVar 0) × (TVar 0 → Unit) × (TVar 0 → Unit).
```

Introduction

This talk: logic for interactive refinement proofs of stateful, concurrent, higher-order programs.

Features:

- Modular refinement proofs
- Using high-level tactics
- With a mechanised soundness proof



Example:

```
Γ ⊨ ticket_lock ≿ spin_lock
: ∃: (Unit → TVar 0) × (TVar 0 → Unit) × (TVar 0 → Unit).
```

Setting

- Object logic is higher-order separation logic with judgements:

$$\Delta; \Gamma \models e_1 \precsim e_2 : \tau.$$

- Programs in an ML-style language with concurrency:

$$\text{System } F + \exists + \mu + \textbf{ref} + \text{fork}.$$

- Soundness with regard to contextual equivalence:

$$\cdot\,; \Gamma \models e_1 \precsim e_2 : \tau \implies \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau.$$

History overview

How to define and reason about $\Delta; \Gamma \models e_1 \precsim e_2 : \tau$?

History overview

How to define and reason about $\Delta; \Gamma \models e_1 \precsim e_2 : \tau$?

- **Explicit step-indexing** (Appel-McAllester, Ahmed, . . .)
  - Definitions: low-level using explicit step-indexing and resources.
  - Proofs: unfolding the definitions, lots of low-level details.

History overview

How to define and reason about $\Delta; \Gamma \models e_1 \precsim e_2 : \tau$?

- **Explicit step-indexing** (Appel-McAllester, Ahmed, ...)
  - Definitions: low-level using explicit step-indexing and resources.
  - Proofs: unfolding the definitions, lots of low-level details.
- **Logical approach** (LSLR, LADR, CaReSL, Iris, ...)
  - Definitions: high-level using a logic to hide explicit step-indexing and/or resources.
  - Proofs: unfolding the definitions, simpler proofs in {LSLR, LADR, CaReSL, Iris, ...}

History overview

How to define and reason about $\Delta; \Gamma \models e_1 \precsim e_2 : \tau$?

- **Explicit step-indexing** (Appel-McAllester, Ahmed, ...)
  - Definitions: low-level using explicit step-indexing and resources.
  - Proofs: unfolding the definitions, lots of low-level details.
- **Logical approach** (LSLR, LADR, CaReSL, Iris, ...)
  - Definitions: high-level using a logic to hide explicit step-indexing and/or resources.
  - Proofs: unfolding the definitions, simpler proofs in {LSLR, LADR, CaReSL, Iris, ...}

Problem: breaks abstraction!

History overview

How to define and reason about $\Delta; \Gamma \models e_1 \precsim e_2 : \tau$?

- **Explicit step-indexing** (Appel-McAllester, Ahmed, ...)
  - Definitions: low-level using explicit step-indexing and resources.
  - Proofs: unfolding the definitions, lots of low-level details.
- **Logical approach** (LSLR, LADR, CaReSL, Iris, ...)
  - Definitions: high-level using a logic to hide explicit step-indexing and/or resources.
  - Proofs: unfolding the definitions, simpler proofs in {LSLR, LADR, CaReSL, Iris, ...}

Problem: breaks abstraction!

- **This work**
  - Definitions: high-level using a logic to hide explicit steps and/or resources.
  - Proofs: using high-level proof rules + reasoning principles of Iris (ghost state, invariants, ...)

Plan

Overview and goals

Symbolic execution

Handling mutable state

Handling concurrency/invariants

Relational specifications of compound programs

Perspectives

Symbolic execution

Basic approach

We prove statements like

$$\Delta; \Gamma \models e_1 \precsim e_2 : \tau$$

by structural arguments and *simulation arguments*: performing symbolic execution, matching every step of the LHS with zero or more steps on the RHS.

- For every program execution on the LHS we can pick an execution on the RHS...
- Every single step on the LHS is atomic, whereas such concept does not apply to the RHS (see Invariants);
- This approach leads to many simple proofs, but has its limitations (see Further Work);

Example: bit module

A bit interface:

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

- initial state
- flip the bit
- view the bit as a Boolean

Two implementations:

```
Definition bit_bool : val :=
  pack (#true,
        (λ: "b", ¬ "b"),
        (λ: "b", "b")).
```

```
Definition bit_nat : val :=
  pack (#1,
    (λ: "n", if: ("n" = #0)
             then #1 else #0),
    (λ: "b", "b" = #1)).
```

Example: bit refinement

In order to prove the refinement:

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

we select a relation linking together the underlying types of `bit_bool` and `bit_nat`:

$$R \subseteq \texttt{Bool} \times \texttt{Nat} = \{(\texttt{true}, 1), (\texttt{false}, 0)\}.$$

```
Definition f (b : bool) : nat := if b then 1 else 0.
Definition R : D := valrel (λ v1 v2,
  (∃ b : bool, ⌜ v1 = #b ⌝ ∗ ⌜ v2 = #(f b) ⌝ ))%I.
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.



Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{Δ; Γ} ⊨ bit_bool ≾ bit_nat : bitT
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.



Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{Δ; Γ} ⊨ bit_bool ≾ bit_nat : bitT
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
```

```
Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{Δ; Γ} ⊨
  pack (#true,
        λ: "b", ¬ "b",
        λ: "b", "b")
≿
  pack (#1,
        λ: "n", if: "n" = #0
                then #1 else #0,
        λ: "b", "b" = #1) : bitT
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
```

```
Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{Δ; Γ} ⊨
  pack (#true,
        λ: "b", ¬ "b",
        λ: "b", "b")
≾
  pack (#1,
        λ: "n", if: "n" = #0
                then #1 else #0,
        λ: "b", "b" = #1) : bitT
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).



Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{(R :: Δ); ↑Γ} ⊨
  (#true, λ: "b", ¬ "b", λ: "b", "b")
≾
  (#1, λ: "n", if: "n" = #0
                 then #1 else #0,
       λ: "b", "b" = #1)
:
  TVar 0 × (TVar 0 → TVar 0)
         × (TVar 0 → Bool)
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.



Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{(R :: Δ); ↑Γ} ⊨
  (#true, λ: "b", ¬ "b", λ: "b", "b")
≾
  (#1, λ: "n", if: "n" = #0
                 then #1 else #0,
       λ: "b", "b" = #1)
:
  TVar 0 × (TVar 0 → TVar 0)
         × (TVar 0 → Bool)
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.

Qed.
```

```
3 subgoals

Δ : list D
Γ : stringmap type
============================(1/3)
{(R :: Δ); ↑Γ} ⊨ #true ≾ #1 : TVar 0

Δ : list D
Γ : stringmap type
============================(2/3)
{(R :: Δ); ↑Γ} ⊨
  (λ: "b", ¬ "b")
≾
  (λ: "n", if: "n" = #0
           then #1 else #0)
: (TVar 0 → TVar 0)
...
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  -



Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{(R :: Δ); ↑Γ} ⊨ #true ≾ #1 : TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.

Qed.
```

```
Δ : list D
Γ : stringmap type
──────────────────────────────────── (1/1)
{(R :: Δ); ↑Γ} ⊨ #true ≾ #1 : TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.




Qed.
```

```
This subproof is complete, but there
are some unfocused goals.
Focus next goal with bullet —.
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  -



Qed.
```

```
Δ : list D
Γ : stringmap type
──────────────────────────────────── (1/1)
{(R :: Δ); ↑Γ} ⊨
  (λ: "b", ¬ "b")
≾
  (λ: "n", if: "n" = #0
           then #1 else #0)
: (TVar 0 → TVar 0)
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.




Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{(R :: Δ); ↑Γ} ⊨
  (λ: "b", ¬ "b")
≾
  (λ: "n", if: "n" = #0
           then #1 else #0)
: (TVar 0 → TVar 0)
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.




Qed.
```

```
Δ : list D
Γ : stringmap type
──────────────────────────────────────
□ (∀ v1 v2 : valC,
   □ (∃ b : bool, ⌜ v1 = #b ⌝ ∗ ⌜ v2 =
        #(f b) ⌝ ) -∗
   {(R :: Δ); ↑Γ} ⊨
     (let: "b" := v1 in ¬ "b")
   ≾
     (let: "n" := v2 in
      if: "n" = #0 then #1 else #0)
   : TVar 0)
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).




Qed.
```

```
Δ : list D
Γ : stringmap type
──────────────────────────────────────
□ (∀ v1 v2 : valC,
   □ (∃ b : bool, ⌜ v1 = #b ⌝ ∗ ⌜ v2 =
        #(f b) ⌝ ) -∗
   {(R :: Δ); ↑Γ} ⊨
     (let: "b" := v1 in ¬ "b")
   ≾
     (let: "n" := v2 in
      if: "n" = #0 then #1 else #0)
   : TVar 0)
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).




Qed.
```

```
Δ : list D
Γ : stringmap type
============================
□ (∃ b : bool, ⌜ v1 = #b ⌝ ∗ ⌜ v2 = #(
     f b) ⌝ ) -∗
{(R :: Δ); ↑Γ} ⊨
  (let: "b" := v1 in ¬ "b")
≿
  (let: "n" := v2 in
   if: "n" = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.



Qed.
```

```
Δ : list D
Γ : stringmap type
═══════════════════════════════
□ (∃ b : bool, ⌜ v1 = #b ⌝ ∗ ⌜ v2 = #(
     f b) ⌝ ) -∗
{(R :: Δ); ↑Γ} ⊨
  (let: "b" := v1 in ¬ "b")
≿
  (let: "n" := v2 in
   if: "n" = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.



Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  (let: "b" := #b in ¬ "b")
≿
  (let: "n" := #(f b) in
   if: "n" = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l.




Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  (let: "b" := #b in ¬ "b")
≾
  (let: "n" := #(f b) in
   if: "n" = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l.



Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  (¬ #b)
≿
  (let: "n" := #(f b) in
   if: "n" = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.




Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  (¬ #b)
≿
  (let: "n" := #(f b) in
   if: "n" = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.




Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  (¬ #b)
≿
  (if: #(f b) = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l.



Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  (¬ #b)
≿
  (if: #(f b) = #0 then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l.



Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  #(xorb b true)
≿
  (if: #(f b) = #0
   then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l. rel_op_r.



Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  #(xorb b true)
≿
  (if: #(f b) = #0
   then #1 else #0)
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l. rel_op_r.



Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  #(xorb b true)
≾
  (if: #(Nat.eqb (f b) 0)
   then #1 else #0) :
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≾ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l. rel_op_r.
    destruct b; simpl; rel_if_r;
      rel_finish.

Qed.
```

```
Δ : list D
Γ : stringmap type
b : bool
──────────────────────────────────
{(R :: Δ); ↑Γ} ⊨
  #(xorb b true)
≾
  (if: #(Nat.eqb (f b) 0)
   then #1 else #0) :
: TVar 0
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l. rel_op_r.
    destruct b; simpl; rel_if_r;
      rel_finish.

Qed.
```

```
This subproof is complete, but there
are some unfocused goals.
Focus next goal with bullet —.
```

Demo in Coq: bit module

```
Lemma bit_refinement Δ Γ :
  {Δ;Γ} ⊨ bit_bool ≿ bit_nat : bitT.
Proof.
  unlock bit_bool bit_nat; simpl.
  iApply (bin_log_related_pack _ R).
  repeat iApply bin_log_related_pair.
  - rel_finish.
  - rel_arrow_val; simpl.
    iIntros "!#" (v1 v2).
    iIntros ([b [??]]); simplify_eq/=.
    rel_let_l. rel_let_r.
    rel_op_l. rel_op_r.
    destruct b; simpl; rel_if_r;
      rel_finish.
  - ...
Qed.
```

```
This subproof is complete, but there
are some unfocused goals.
Focus next goal with bullet —.
```

Rules

Some of the rules we have used in the proof:

$$\frac{[\alpha := R], \Delta; \Gamma \models e \precsim e' : \tau}{\Delta; \Gamma \models \mathsf{pack}\ e \precsim \mathsf{pack}\ e' : \exists \alpha.\, \tau}$$

Rules

Some of the rules we have used in the proof:

$$\frac{[\alpha := R], \Delta; \Gamma \models e \precsim e' : \tau}{\Delta; \Gamma \models \mathsf{pack}\ e \precsim \mathsf{pack}\ e' : \exists \alpha.\, \tau}$$

$$\frac{\begin{array}{l}\square \forall v\, v',\ [\![\tau]\!]_\Delta(v, v') \twoheadrightarrow \\ \qquad \Delta; \Gamma \models (\lambda x.e)\ v \precsim (\lambda x'.e')\ v' : \tau'\end{array}}{\Delta; \Gamma \models \lambda x.e \precsim \lambda x'.e' : \tau \to \tau'}$$

Rules

Some of the rules we have used in the proof:

$$\frac{[\alpha := R], \Delta; \Gamma \models e \precsim e' : \tau}{\Delta; \Gamma \models \mathsf{pack}\ e \precsim \mathsf{pack}\ e' : \exists \alpha.\, \tau}$$

$$\frac{\begin{array}{l}\square \forall v\, v',\ [\![\tau]\!]_\Delta(v, v') \twoheadrightarrow \\ \qquad \Delta; \Gamma \models (\lambda x.e)\ v \precsim (\lambda x'.e')\ v' : \tau'\end{array}}{\Delta; \Gamma \models \lambda x.e \precsim \lambda x'.e' : \tau \to \tau'}$$

$$\frac{e \xrightarrow{\text{pure}} e' \qquad \Delta; \Gamma \models K[e'] \precsim t : \tau}{\Delta; \Gamma \models K[e] \precsim t : \tau}$$

(same for the RHS)

Handling mutable state

Mutable state

We have seen an example refinement in the pure fragment of the language, but what about:

- Mutable state
- Concurrency

Mutable state

We have seen an example refinement in the pure fragment of the language, but what about:

- *Mutable state*
- Concurrency

Separation logic to the rescue (CaReSL/Iris)

- $l \mapsto_i v$ for LHS/implementation
- $l \mapsto_s v$ for RHS/specification

Rules and lemmas

Inference rules correspond to lemmas inside Iris:

Rule

$$\frac{l \mapsto_s v \quad * \quad (l \mapsto_s v' \mathrel{-\!\!*} \Delta; \Gamma \models e \precsim K[()] : \tau)}{\Delta; \Gamma \models e \precsim K[l \leftarrow v'] : \tau}$$

Corresponding Coq lemma

```
Lemma bin_log_related_store_r Δ Γ K l e e' v v' τ :
  to_val e' = Some v' →
  l ↦ₛ v -∗
  (l ↦ₛ v' -∗ {Δ;Γ} ⊨ e ≾ fill K (#()) : τ) -∗
  {Δ;Γ} ⊨ e ≾ fill K (#l ← e') : τ.
```

Rules for load

```
Lemma bin_log_related_load_l' Δ Γ K l v t τ :
  l ↦ᵢ v -∗
  (l ↦ᵢ v -∗ ({Δ;Γ} ⊨ fill K (of_val v) ≾ t : τ)) -∗
  {Δ;Γ} ⊨ fill K !#l ≾ t : τ.
```

```
Lemma bin_log_related_load_r Δ Γ K l v t τ :
  l ↦ₛ v -∗
  (l ↦ₛ v -∗ {Δ;Γ} ⊨ t ≾ fill K (of_val v) : τ) -∗
  {Δ;Γ} ⊨ t ≾ fill K (! #l)) : τ.
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.



Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────
l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
{Δ;Γ} ⊨ ! #l ≾ (#k ← #1;; ! #k) : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≿ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".



Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────
l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
{Δ;Γ} ⊨ ! #l ≿ (#k ← #1;; ! #k) : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≿ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".



Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #0
──────────────────────────────────-∗
{Δ;Γ} ⊨ ! #l ≿ (#k ← #1;; ! #k) : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≿ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r.


Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #0
──────────────────────────────────∗
{Δ;Γ} ⊨ ! #l ≿ (#k ← #1;; ! #k) : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≿ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r.


Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────∗
{Δ;Γ} ⊨ ! #l ≿ (#();; ! #k) : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.



Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────∗
{Δ;Γ} ⊨ ! #l ≾ (#();; ! #k) : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.



Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────∗
{Δ;Γ} ⊨ ! #l ≾ ! #k : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.
 rel_load_l.

Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────∗
{Δ;Γ} ⊨ ! #l ≾ ! #k : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.
 rel_load_l.

Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────────
{Δ;Γ} ⊨ #1 ≾ ! #k : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.
 rel_load_l. rel_load_r.

Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────────────
{Δ;Γ} ⊨ #1 ≾ ! #k : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≾ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.
 rel_load_l. rel_load_r.

Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────────────
{Δ;Γ} ⊨ #1 ≾ #1 : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≿ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.
 rel_load_l. rel_load_r.
 iApply bin_log_related_nat.
Qed.
```

```
Δ : list D
Γ : stringmap type
l, k : loc
──────────────────────────────────
"Hl" : l ↦ᵢ #1
"Hk" : k ↦ₛ #1
──────────────────────────────────
{Δ;Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: simple mutable state

```
Lemma test_goal Δ Γ (l k : loc) :
 l ↦ᵢ #1 -∗ k ↦ₛ #0 -∗
 {Δ;Γ} ⊨ !#l ≲ (#k ← #1;; !#k) : TNat.
Proof.
 iIntros "Hl Hk".
 rel_store_r. rel_seq_r.
 rel_load_l. rel_load_r.
 iApply bin_log_related_nat.
Qed.
```

```
No more subgoals.
```

Tactic lemmas

In order to automatically discharge goals and introduce assumptions related to resource management, for each lemma there is a corresponding tactic lemma, which is a lemma outside Iris.

Lemma

```
Lemma bin_log_related_store_r Δ Γ K l (e e' : expr) (v v' : val) (τ : type) :
  to_val e' = Some v' →
  l ↦ₛ v -∗
  (l ↦ₛ v' -∗ {Δ;Γ} ⊨ e ≲ fill K (#()) : τ) -∗
  {Δ;Γ} ⊨ e ≲ fill K (#l ← e') : τ.
```

Tactic lemma

```
Lemma tac_rel_store_r `{logrelG Σ} Ξ1 Ξ2 Δ Γ K i1 (l : loc) e e' t τ v v' :
  t = fill K (Store (# l) t') →
  to_val e' = Some v' →
  envs_lookup i1 Ξ1 = Some (false, l ↦ₛ v)%I →
  envs_simple_replace i1 false (Esnoc Enil i1 (l ↦ₛ v')) Ξ1 = Some Ξ2 →
  envs_entails Ξ2 ({Δ;Γ} ⊨ e ≲ (fill K Unit) : τ) →
  envs_entails Ξ1 ({Δ;Γ} ⊨ e ≲ t : τ).
```

Tactic lemmas

In order to automatically discharge goals and introduce assumptions related to resource management, for each lemma there is a corresponding tactic lemma, which is a lemma outside Iris.

Lemma

```
Lemma bin_log_related_store_r Δ Γ K l (e e' : expr) (v v' : val) (τ : type) :
  to_val e' = Some v' →
  l ↦ₛ v -∗
  (l ↦ₛ v' -∗ {Δ;Γ} ⊨ e ≲ fill K (#()) : τ) -∗
  {Δ;Γ} ⊨ e ≲ fill K (#l ← e') : τ.
```

Tactic lemma

```
Lemma tac_rel_store_r `{logrelG Σ} Ξ1 Ξ2 Δ Γ K i1 (l : loc) e e' t τ v v' :
  t = fill K (Store (# l) t') →
  to_val e' = Some v' →
  envs_lookup i1 Ξ1 = Some (false, l ↦ₛ v)%I →
  envs_simple_replace i1 false (Esnoc Enil i1 (l ↦ₛ v')) Ξ1 = Some Ξ2 →
  envs_entails Ξ2 ({Δ;Γ} ⊨ e ≲ (fill K Unit) : τ) →
  envs_entails Ξ1 ({Δ;Γ} ⊨ e ≲ t : τ).
```

Old and new goals

Tactic lemmas

In order to automatically discharge goals and introduce assumptions related to resource management, for each lemma there is a corresponding tactic lemma, which is a lemma outside Iris.

Lemma

```
Lemma bin_log_related_store_r Δ Γ K l (e e' : expr) (v v' : val) (τ : type) :
  to_val e' = Some v' →
  l ↦ₛ v -∗
  (l ↦ₛ v' -∗ {Δ;Γ} ⊨ e ≲ fill K (#()) : τ) -∗
  {Δ;Γ} ⊨ e ≲ fill K (#l ← e') : τ.
```

Tactic lemma

Old and new resources

```
Lemma tac_rel_store_r `{logrelG Σ} Ξ1 Ξ2 Δ Γ K i1 (l : loc) e e' t t' v' :
  t = fill K (Store (# l) t') →
  to_val e' = Some v' →
  envs_lookup i1 Ξ1 = Some (false, l ↦ₛ v)%I →
  envs_simple_replace i1 false (Esnoc Enil i1 (l ↦ₛ v')) Ξ1 = Some Ξ2 →
  envs_entails Ξ2 ({Δ;Γ} ⊨ e ≲ (fill K Unit) : τ) →
  envs_entails Ξ1 ({Δ;Γ} ⊨ e ≲ t : τ).
```

Tactic lemmas

In order to automatically discharge goals and introduce assumptions related to resource management, for each lemma there is a corresponding tactic lemma, which is a lemma outside Iris.

Lemma

```
Lemma bin_log_related_store_r Δ Γ K l (e e' : expr) (v v' : val) (τ : type) :
  to_val e' = Some v' →
  l ↦ₛ v -∗
  (l ↦ₛ v' -∗ {Δ;Γ} ⊨ e ≲ fill K (#()) : τ) -∗
  {Δ;Γ} ⊨ e ≲ fill K (#l ← e') : τ.
```

Tactic lemma

```
Lemma tac_rel_store_r `{logrelG Σ} Ξ1 Ξ2 Δ Γ K i1 (l : loc)
  t = fill K (Store (# l) t') →
  to_val e' = Some v' →
  envs_lookup i1 Ξ1 = Some (false, l ↦ₛ v)%I →
  envs_simple_replace i1 false (Esnoc Enil i1 (l ↦ₛ v')) Ξ1 = Some Ξ2 →
  envs_entails Ξ2 ({Δ;Γ} ⊨ e ≲ (fill K Unit) : τ) →
  envs_entails Ξ1 ({Δ;Γ} ⊨ e ≲ t : τ).
```

Find `l ←v'` in the term `t`

Handling concurrency/invariants

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.



Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{Δ; Γ} ⊨
  (let: "x" := ref #1 in λ: "f", "f" #();; ! "x")
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".

Qed.
```

```
Δ : list D
Γ : stringmap type
============================ (1/1)
{Δ; Γ} ⊨
  (let: "x" := ref #1 in λ: "f", "f" #();; ! "x")
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".

Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────── (1/1)
"Hl" : l ↦ᵢ #1
──────────────────────────────────*
{Δ; Γ} ⊨
  (let: "x" := #l in λ: "f", "f" #();; ! "x")
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.

Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────── (1/1)
"Hl" : l ↦ᵢ #1
──────────────────────────────────*
{Δ; Γ} ⊨
  (let: "x" := #l in λ: "f", "f" #();; ! "x")
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.

Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────── (1/1)
"Hl" : l ↦ᵢ #1
──────────────────────────────────*
{Δ; Γ} ⊨
  (λ: "f", "f" #();; ! #l)
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────────── (1/1)
"Hl" : l ↦ᵢ #1
──────────────────────────────────────∗
{Δ; Γ} ⊨
  (λ: "f", "f" #();; ! #l)
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

```
Qed.
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
────────────────────────────────□
{Δ; Γ} ⊨
  (λ: "f", "f" #();; ! #l)
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
────────────────────────────────□
{Δ; Γ} ⊨
  (λ: "f", "f" #();; ! #l)
≿
  (λ: "f", "f" #();; #1) : ((Unit → Unit) → TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────────(1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
──────────────────────────────────□
□ (∀ v1 v2 : val,
   □ ({Δ; Γ} ⊨ v1 ≿ v2 : (Unit → Unit)) -∗
   {Δ; Γ} ⊨
     (let: "f" := v1 in "f" #();; ! #l)
   ≿
     (let: "f" := v2 in "f" #();; #1) : TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
──────────────────────────────────□
□ (∀ v1 v2 : val,
   □ ({Δ; Γ} ⊨ v1 ≿ v2 : (Unit → Unit)) -∗
   {Δ; Γ} ⊨
     (let: "f" := v1 in "f" #();; ! #l)
   ≿
     (let: "f" := v2 in "f" #();; #1) : TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".



Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨
  (let: "f" := f1 in "f" #();; ! #l)
≿
  (let: "f" := f2 in "f" #();; #1) : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨
  (let: "f" := f1 in "f" #();; ! #l)
≿
  (let: "f" := f2 in "f" #();; #1) : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
────────────────────────────────────□
{Δ; Γ} ⊨ (f1 #();; ! #l) ≿ (f2 #();; #1) : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ (f1 #();; ! #l) ≿ (f2 #();; #1) : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.




Qed.
```

```
2 subgoals

Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
============================ (1/2)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
----------------------------□
{Δ; Γ} ⊨ f1 #() ≿ f2 #() : ?H7

Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
============================ (2/2)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
----------------------------□
{Δ; Γ} ⊨ ! #l ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  -




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────□
{Δ; Γ} ⊨ f1 #() ≿ f2 #() : ?H7
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ f1 #() ≿ f2 #() : ?H7
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ #() ≿ #() : Unit
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ #() ≿ #() : Unit
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.




Qed.
```

```
This subproof is complete, but there are some
      unfocused goals.
Focus next goal with bullet —.
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  -




Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ ! #l ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.



Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ ! #l ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.



Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
"Hl" : ▷ l ↦ᵢ #1
"Hcl" : ▷ l ↦ᵢ #1 ={⊤ ∖ ↑N,⊤}=∗ True
──────────────────────────────────────∗
∃ v' : val,
▷ l ↦ᵢ v'
∗ ▷ (l ↦ᵢ v' -∗ {⊤ ∖ ↑N; Δ; Γ} ⊨ v' ≿ #1 : TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.



Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
"Hl" : ▷ l ↦ᵢ #1
"Hcl" : ▷ l ↦ᵢ #1 ={⊤ ∖ ↑N,⊤}=∗ True
──────────────────────────────────────∗
∃ v' : val,
▷ l ↦ᵢ v'
∗ ▷ (l ↦ᵢ v' -∗ {⊤ ∖ ↑N; Δ; Γ} ⊨ v' ≿ #1 : TNat)
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.



Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
"Hcl" : ▷ l ↦ᵢ #1 ={⊤ ∖ ↑N,⊤}=∗ True
──────────────────────────────────────∗
l ↦ᵢ #1 -∗ {⊤ ∖ ↑N; Δ; Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.
    iIntros "Hl".


Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
"Hcl" : ▷ l ↦ᵢ #1 ={⊤ ∖ ↑N,⊤}=∗ True
──────────────────────────────────────∗
l ↦ᵢ #1 -∗ {⊤ ∖ ↑N; Δ; Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.
    iIntros "Hl".


Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
"Hcl" : ▷ l ↦ᵢ #1 ={⊤ ∖ ↑N,⊤}=∗ True
"Hl" : l ↦ᵢ #1
──────────────────────────────────────∗
{⊤ ∖ ↑N; Δ; Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.
    iIntros "Hl".
    iMod ("Hcl" with "Hl") as "_".

Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
"Hcl" : ▷ l ↦ᵢ #1 ={⊤ ∖ ↑N,⊤}=∗ True
"Hl" : l ↦ᵢ #1
──────────────────────────────────────∗
{⊤ ∖ ↑N; Δ; Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.
    iIntros "Hl".
    iMod ("Hcl" with "Hl") as "_".

Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.
    iIntros "Hl".
    iMod ("Hcl" with "Hl") as "_".
    iApply bin_log_related_nat.
Qed.
```

```
Δ : list D
Γ : stringmap type
l : loc
f1, f2 : val
──────────────────────────────── (1/1)
"Hinv" : inv N (l ↦ᵢ #1)%I
"Hf" : {Δ; Γ} ⊨ f1 ≿ f2 : (Unit → Unit)
──────────────────────────────────────□
{Δ; Γ} ⊨ #1 ≿ #1 : TNat
```

Demo in Coq: higher-order reasoning

```
Lemma higher_order_stateful Δ Γ :
  {Δ;Γ} ⊨
   let: "x" := ref #1 in
   (λ: "f", "f" #();; !"x")
  ≿
   (λ: "f", "f" #();; #1)
  : ((Unit → Unit) → TNat).
Proof.
  rel_alloc_l as l "Hl".
  rel_let_l.
  iMod (inv_alloc N _ (l ↦ᵢ #1)%I with "Hl")
    as "#Hinv".
  rel_arrow.
  iIntros "!#" (f1 f2) "#Hf".
  rel_let_l; rel_let_r.
  iApply bin_log_related_seq; auto.
  - iApply (bin_log_related_app with "Hf").
    iApply bin_log_related_unit.
  - rel_load_l_atomic;
    iInv N as "Hl" "Hcl"; iModIntro.
    iExists #1; iNext; iFrame "Hl"; simpl.
    iIntros "Hl".
    iMod ("Hcl" with "Hl") as "_".
    iApply bin_log_related_nat.
Qed.
```

```
No more subgoals.
```

Compatibility lemmas

Compatibility rules that we used in the proof above:

Rule

$$\frac{\Delta; \Gamma \models e_1 \precsim e_1' : \tau_1 \qquad \Delta; \Gamma \models e_2 \precsim e_2' : \tau_2}{\Delta; \Gamma \models e_1; e_2 \precsim e_1'; e_2' : \tau_2}$$

Corresponding Coq lemma

```
Lemma bin_log_related_seq R Δ Γ e1 e2 e1' e2' τ1 τ2 :
  {(R::Δ);↑Γ} ⊨ e1 ≼ e1' : τ1 →*
  {Δ;Γ} ⊨ e2 ≼ e2' : τ2 →*
  {Δ;Γ} ⊨ (e1;; e2) ≼ (e1';; e2') : τ2.
```

Compatibility lemmas (2)

$$\frac{\Delta; \Gamma \models e_1 \precsim e_1' : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \models e_2 \precsim e_2' : \tau_1}{\Delta; \Gamma \models e_1\ e_2 \precsim e_1'\ e_2' : \tau_2}$$

$$\Delta; \Gamma \models () \precsim () : \mathbf{1}$$

$$\Delta; \Gamma \models n \precsim n : \mathbf{N}$$

Invariant allocation

We can allocate invariants (and other resources) using the following rule:

$$\frac{\Rrightarrow \Delta; \Gamma \models e \precsim e' : \tau}{\Delta; \Gamma \models e \precsim e' : \tau}$$

Opening invariants around atomic operations

In case we wish to open the invariant around the load operation (*e.g.*, $\ell \mapsto_{\mathsf{i}} -$ is in an invariant):

$$\frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists v.\, \ell \mapsto_{\mathsf{i}} v * \triangleright(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta; \Gamma \models_{\mathcal{E}} K[\,v\,] \precsim e' : \tau)}{\Delta; \Gamma \models K[\,!\ell\,] \precsim e' : \tau}$$

The role of the mask annotation $\mathcal{E}$ in the proposition $\Delta; \Gamma \models_{\mathcal{E}} K[\,v\,] \precsim e' : \tau$ is two-fold:

- Keeping track of which invariants have been opened (*i.e.*, at that point we can only open invariants from $\mathcal{E}$ to prevent reentrancy);
- Keeping track of *whether* an invariant has been opened (*i.e.*, whether $\mathcal{E} = \top$) to prevent performing several execution steps with an opened invariant.

$\Delta; \Gamma \models e \precsim e' : \tau$ is a shortcut for $\Delta; \Gamma \models_{\top} e \precsim e' : \tau$.

Closing the invariant

Once we arrive at the new goal $\Delta; \Gamma \models_{\mathcal{E}} K[\, v \,] \precsim e' : \tau$ we cannot perform any symbolic execution steps on the LHS (we can perform symbolic execution on the RHS tho).

We can restore that mask back to $\top$ by closing the invariant:

$$\frac{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \Delta; \Gamma \models_{\mathcal{E}_2} e \precsim e' : \tau}{\Delta; \Gamma \models_{\mathcal{E}_1} e \precsim e' : \tau}$$

Relational specifications of compound programs

Relational specifications

To facilitate modular reasoning about refinements of compound programs we need to come up with *relational specifications*, analogous to Hoare triples in unary setting.

RHS relational specification for locks:

$$\frac{\forall v.\, \mathsf{isLock}(v, \mathsf{false}) \mathrel{-\!\!*} \Delta; \Gamma \models_{\mathcal{E}} e \precsim K[\, v \,] : \tau}{\Delta; \Gamma \models_{\mathcal{E}} e \precsim K[\, \mathsf{newlock}\ () \,] : \tau}$$

$$\frac{\mathsf{isLock}(v, \mathsf{false}) \qquad \mathsf{isLock}(v, \mathsf{true}) \mathrel{-\!\!*} \Delta; \Gamma \models_{\mathcal{E}} e \precsim K[\, () \,] : \tau}{\Delta; \Gamma \models_{\mathcal{E}} e \precsim K[\, \mathsf{acquire}\ v \,] : \tau}$$

$$\frac{\mathsf{isLock}(v, b) \qquad \mathsf{isLock}(v, \mathsf{false}) \mathrel{-\!\!*} \Delta; \Gamma \models_{\mathcal{E}} e \precsim K[\, () \,] : \tau}{\Delta; \Gamma \models_{\mathcal{E}} e \precsim K[\, \mathsf{release}\ v \,] : \tau}$$

General form of relational specifications

A Hoare triple $\{P\}\, e\, \{v.\, Q(v)\}$ corresponds (informally) to the following rules for the RHS and LHS:

$$\frac{P \qquad \forall v.\, Q(v) \twoheadrightarrow \Delta; \Gamma \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta; \Gamma \models_{\mathcal{E}} e_1 \precsim K[\,e\,] : \tau}$$

$$\frac{P \qquad \forall v.\, Q(v) \twoheadrightarrow \Delta; \Gamma \models K[\,v\,] \precsim e_2 : \tau}{\Delta; \Gamma \models K[\,e\,] \precsim e_2 : \tau}$$

In the second case the correspondence is *formal* in the sense that we can lift any HT to a specification like that.

Atomic relational specifications

Note that no invariants can be opened for the relational specifications on the LHS.

HT-like specifications for the LHS suffer from the same issues as Hoare triples in unary case: if the program e is truly compound and the resources for the preconditions are shared via an invariant, then the rule is pretty much useless!

We need something like logically atomic triples, but for the relational case.

$$\langle x.\, \alpha(x) \rangle\, e\, \langle v.\, \beta(x, v) \rangle \mathcal{E}_i, \mathcal{E}_o$$

Logically atomic FAI

```
Definition FAI : val := rec: "inc" "x" :=
  let: "c" := !"x" in
  if: CAS "x" "c" (#1 + "c")
  then "c"
  else "inc" "x".
```

`FAI(x)` "atomically" updates $x \mapsto_i n$ to $x \mapsto_i n + 1$.

Logically atomic FAI

```
Definition FAI : val := rec: "inc" "x" :=
  let: "c" := !"x" in
  if: CAS "x" "c" (#1 + "c")
  then "c"
  else "inc" "x".
```

`FAI(x)` "atomically" updates $x \mapsto_i n$ to $x \mapsto_i n+1$.

$$\square\, {}^{\top}\!\Rrightarrow^{\mathcal{E}} \begin{pmatrix} \exists n.\, x \mapsto_{\mathsf{i}} n * R(n) * \\ (x \mapsto_{\mathsf{i}} n * R(n) \;{}^{\mathcal{E}}\!\!\Longrightarrow\!\!\!\!*^{\top}\; \mathsf{True}) \wedge \\ (x \mapsto_{\mathsf{i}} (n+1) * R(n) \mathrel{-\!\!*} \Delta; \Gamma \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau) \end{pmatrix}$$
$$\overline{\Delta; \Gamma \models K[\,\mathtt{FAI}(x)\,] \precsim e : \tau}$$

Using logically atomic rules

Coq demo!!!

- `counter.v`
- `ticket_lock.v`

General form of logically atomic relational specifications

$$R_2 \qquad \square\, {}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{l} \exists x : A.\, P(x) * R_1(x) * \\ \qquad \left(P(x) * R_1(x) \;{}^{\mathcal{E}}\!\!\Longrightarrow\!\!\!\!\!*\;{}^{\top}\; \mathsf{True}\right) \wedge \\ \qquad \left(\forall v.\, Q(x,v) * R_1(x) * R_2 \mathrel{-\!\!*} \right. \\ \qquad\qquad \left. \Delta; \Gamma \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\right) \end{array} \right)$$
$$\overline{\Delta; \Gamma \models K[\,e_1\,] \precsim e_2 : \tau}$$

The type A is chosen by the client; $P : A \to \mathrm{Prop}$ describes consumed resources; $Q : A \times \mathit{Val} \to \mathrm{Prop}$ describes consumed resources.

$R_1 : A \to \mathrm{Prop}$ is an *invariant frame* and R_2 is an *ephemeral frame*.

Lifting atomic specifications

Every atomic Hoare triple

$$\langle x.\, P(x)\rangle\ e_1\ \langle v.\, Q(x,v)\rangle_{\mathcal{E},\,\top}$$

can be lifted to logically atomic relational specification

$$\frac{R_2 \qquad \square\, {}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{l} \exists x : A.\, P(x) * R_1(x) * \\ \qquad \left(P(x) * R_1(x) \;{}^{\mathcal{E}}\!\Longrightarrow\!\!\!\!\ast^{\top}\ \mathsf{True}\right) \wedge \\ \qquad \left(\forall v.\, Q(x,v) * R_1(x) * R_2 \mathrel{-\!\!*}\right. \\ \qquad\qquad \left.\Delta; \Gamma \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\right) \end{array}\right)}{\Delta; \Gamma \models K[\,e_1\,] \precsim e_2 : \tau}$$

(for a closed expression e_1)

More examples?

More examples.

- `or.v`
- `symbol.v`
- `https://gitlab.mpi-sws.org/dfrumin/logrel-conc/tree/master/theories/examples`

Perspectives

Future work

- (Fully) automatic symbolic execution of simple programs.
- More expressive logic and more expressive languages/type systems.
- Carefuly considering notions of atomicity and realtion between logical atomicity and linearizability.
- Equivalences between programs in different langages.
- On the previous point: we are not exploiting the fact that we have the same PL on both sides of the refinement.

Thank you for listening!

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

Shout-outs to Robbert, Amin, Lars (IPM paper, POPL'17) and all the Iris contributors.

Additional slide: overview of the development

- Small-step CBV semantics for the object programming language (untyped);
- Reified syntax for solving some properties by reflection (e.g. whether an expression is closed, is a value);
- Typing system for the object language and typed contextual refinement;
- Encoding of logical relations in Iris with a soundness proof;
- Proofs of the primitive rules (in the model) and of the derived rules (in the logic);
- Tactics and tactic lemmas for actually using the calculus.

Plus some intermediate developments such as the machinery for the weakest precondition calculus and ghost state