

Logical Relations in Iris

Amin Timany¹ Robert Krebbers² Lars Birkedal³

¹imec-Distrinet, KU Leuven, Belgium

²Delft University of Technology, The Netherlands

³Aarhus University, Denmark

January 21, 2017 @ CoqPL, Paris, France

Logical Relations

A powerful technique to prove properties of programs and programming languages

- ▶ Unary: **type safety**, (strong) normalization, ...
- ▶ Binary: **contextual refinement**, contextual equivalence, non-interference, ...

In this talk

- ▶ Formalization of a unary and binary logical relations
- ▶ In the Iris program logic which in turn is implemented in Coq
- ▶ For a programming language ($F_{\mu,ref,conc}$) with a very rich type system
- ▶ Use it to prove type safety and verify contextual refinement of concurrent algorithms

Unary logical relation

Proving type safety

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Prop$

Unary logical relation

Proving type safety

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Prop$

1. Prove *adequacy*: $\llbracket \tau \rrbracket^{\mathcal{E}}(e) \Rightarrow Safe_{\tau}(e)$

Unary logical relation

Proving type safety

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Prop$

1. Prove *adequacy*: $\llbracket \tau \rrbracket^{\mathcal{E}}(e) \Rightarrow Safe_{\tau}(e)$
2. Prove *compatibility lemmas* for typing rules, e.g.:

$$\frac{\llbracket \tau_1 \rrbracket^{\mathcal{E}}(e_1) \quad \llbracket \tau \rrbracket^{\mathcal{E}}(e_2)}{\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{E}}(e_1, e_2)}$$

Unary logical relation

Proving type safety

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Prop$

1. Prove *adequacy*: $\llbracket \tau \rrbracket^{\mathcal{E}}(e) \Rightarrow Safe_{\tau}(e)$
2. Prove *compatibility lemmas* for typing rules, e.g.:

$$\frac{\llbracket \tau_1 \rrbracket^{\mathcal{E}}(e_1) \quad \llbracket \tau \rrbracket^{\mathcal{E}}(e_2)}{\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{E}}(e_1, e_2)}$$

3. Corollary (*soundness*): $\cdot \vdash e : \tau \Rightarrow Safe_{\tau}(e)$

Unary logical relation

Proving type safety

Remember adequacy: $\llbracket \tau \rrbracket^{\mathcal{E}}(e) \Rightarrow \text{Safe}_{\tau}(e)$

- ▶ e need not syntactically be well-typed!

Unary logical relation

Proving type safety

Remember adequacy: $\llbracket \tau \rrbracket^{\mathcal{E}}(e) \Rightarrow \text{Safe}_{\tau}(e)$

- ▶ e need not syntactically be well-typed!
- ▶ *Compatibility lemmas* say nothing about well-typedness:

$$\frac{\llbracket \tau_1 \rrbracket^{\mathcal{E}}(e_1) \quad \llbracket \tau \rrbracket^{\mathcal{E}}(e_2)}{\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{E}}(e_1, e_2)}$$

Unary logical relation

Proving type safety

Remember adequacy: $\llbracket \tau \rrbracket^{\mathcal{E}}(e) \Rightarrow \text{Safe}_{\tau}(e)$

- ▶ e need not syntactically be well-typed!
- ▶ *Compatibility lemmas* say nothing about well-typedness:

$$\frac{\llbracket \tau_1 \rrbracket^{\mathcal{E}}(e_1) \quad \llbracket \tau \rrbracket^{\mathcal{E}}(e_2)}{\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{E}}(e_1, e_2)}$$

Logical relation allows one to modularly prove safety in the presence of untyped code, i.e., when linking with *untyped but verified code*, e.g., the *unsafe blocks* of Rust

Motivation for using a high-level logic (Iris)

Recursive types and higher order references

- ▶ **Recursive types:** the logical relation usually involves step-indexing
The *crux* of the matter: semantics of a recursive type $\mu X. \tau$ is the fixpoint of the semantics of τ
- ▶ **References:** the logical relation usually involves step-indexing and possible worlds
The *crux* of the matter: a memory location is of type $\text{ref}(\tau)$ if the value stored in it is in the semantics of type τ at *all times*
- ▶ **Iris** provides support for invariants and taking (guarded) fixpoints

$$\begin{aligned}
\text{HeapAtom}_n &\stackrel{\text{def}}{=} \{(W, h_1, h_2) \mid W \in \text{World}_n\} \\
\text{HeapRel}_n &\stackrel{\text{def}}{=} \{\psi \subseteq \text{HeapAtom}_n \mid \forall (W, h_1, h_2) \in \psi. \forall W' \sqsupseteq W. (W', h_1, h_2) \in \psi\} \\
\text{Island}_n &\stackrel{\text{def}}{=} \{t = (s, \delta, \varphi, \dot{z}, H) \mid s \in \text{State} \wedge \delta \subseteq \text{State}^2 \wedge \varphi \subseteq \delta \wedge \delta, \varphi \text{ reflexive} \wedge \\
&\quad \delta, \varphi \text{ transitive} \wedge \dot{z} \subseteq \text{State} \wedge H \in \text{State} \rightarrow \text{HeapRel}_n\} \\
\text{World}_n &\stackrel{\text{def}}{=} \{W = (k, \Sigma_1, \Sigma_2, \omega) \mid k < n \wedge \exists m. \omega \in \text{Island}_n^m\} \\
\text{ContAtom}_n[\tau_1, \tau_2] &\stackrel{\text{def}}{=} \{(W, K_1, K_2) \mid W \in \text{World}_n \wedge W.\Sigma_1; \cdot \vdash K_1 \div \tau_1 \wedge W.\Sigma_2; \cdot \vdash K_2 \div \tau_2\} \\
\text{TermAtom}_n[\tau_1, \tau_2] &\stackrel{\text{def}}{=} \{(W, e_1, e_2) \mid W \in \text{World}_n \wedge W.\Sigma_1; \cdot \vdash e_1 : \tau_1 \wedge W.\Sigma_2; \cdot \vdash e_2 : \tau_2\} \\
\text{HeapAtom}[\tau_1, \tau_2] &\stackrel{\text{def}}{=} \bigcup_n \text{HeapAtom}_n[\tau_1, \tau_2] \\
\text{World} &\stackrel{\text{def}}{=} \bigcup_n \text{World}_n \\
\text{ContAtom}[\tau_1, \tau_2] &\stackrel{\text{def}}{=} \bigcup_n \text{ContAtom}_n[\tau_1, \tau_2] \\
\text{TermAtom}[\tau_1, \tau_2] &\stackrel{\text{def}}{=} \bigcup_n \text{TermAtom}_n[\tau_1, \tau_2] \\
\text{ValRel}[\tau_1, \tau_2] &\stackrel{\text{def}}{=} \{r \subseteq \text{TermAtom}^{\text{val}}[\tau_1, \tau_2] \mid \forall (W, v_1, v_2) \in r. \forall W' \sqsupseteq W. (W', v_1, v_2) \in r\} \\
\text{SomeValRel} &\stackrel{\text{def}}{=} \{R = (\tau_1, \tau_2, r) \mid r \in \text{ValRel}[\tau_1, \tau_2]\} \\
\llbracket t_1, \dots, t_m \rrbracket_k &\stackrel{\text{def}}{=} \llbracket t_1 \rrbracket_k, \dots, \llbracket t_m \rrbracket_k & \llbracket H \rrbracket_k &\stackrel{\text{def}}{=}} \lambda s. \llbracket H(s) \rrbracket_k \\
\llbracket (s, \delta, \varphi, \dot{z}, H) \rrbracket_k &\stackrel{\text{def}}{=} \llbracket (s, \delta, \varphi, \dot{z}, \llbracket H \rrbracket_k) \rrbracket_k & \llbracket \psi \rrbracket_k &\stackrel{\text{def}}{=}} \{(W, h_1, h_2) \in r \mid W.k < k\}
\end{aligned}$$

$$\begin{aligned}
\triangleright(k+1, \Sigma_1, \Sigma_2, \omega) &\stackrel{\text{def}}{=} (k, \Sigma_1, \Sigma_2, \llbracket \omega \rrbracket_k) \\
\triangleright r &\stackrel{\text{def}}{=} \{(W, e_1, e_2) \mid W.k > 0 \Rightarrow (\triangleright W, e_1, e_2) \in r\} \\
(k', \Sigma'_1, \Sigma'_2, \omega') \sqsupseteq (k, \Sigma_1, \Sigma_2, \omega) &\stackrel{\text{def}}{=} k' \leq k \wedge \Sigma'_1 \supseteq \Sigma_1 \wedge \Sigma'_2 \supseteq \Sigma_2 \wedge \omega' \sqsupseteq \llbracket \omega \rrbracket_{k'} \\
(t'_1, \dots, t'_m) \sqsupseteq (t_1, \dots, t_m) &\stackrel{\text{def}}{=} m' \geq m \wedge \forall j \in \{1, \dots, m\}. t'_j \sqsupseteq t_j \\
(s', \delta', \varphi', \dot{z}', H') \sqsupseteq (s, \delta, \varphi, \dot{z}, H) &\stackrel{\text{def}}{=} (\delta', \varphi', \dot{z}', H') = (\delta, \varphi, \dot{z}, H) \wedge (s, s') \in \delta \\
(k', \Sigma'_1, \Sigma'_2, \omega') \sqsupseteq^{\text{pub}} (k, \Sigma_1, \Sigma_2, \omega) &\stackrel{\text{def}}{=} k' \leq k \wedge \Sigma'_1 \supseteq \Sigma_1 \wedge \Sigma'_2 \supseteq \Sigma_2 \wedge \omega' \sqsupseteq^{\text{pub}} \llbracket \omega \rrbracket_{k'} \\
(t'_1, \dots, t'_m) \sqsupseteq^{\text{pub}} (t_1, \dots, t_m) &\stackrel{\text{def}}{=} m' \geq m \wedge \forall j \in \{1, \dots, m\}. t'_j \sqsupseteq^{\text{pub}} t_j \wedge \\
&\quad \forall j \in \{m+1, \dots, m'\}. \text{safe}(t'_j) \\
(s', \delta', \varphi', \dot{z}', H') \sqsupseteq^{\text{pub}} (s, \delta, \varphi, \dot{z}, H) &\stackrel{\text{def}}{=} (\delta', \varphi', \dot{z}', H') = (\delta, \varphi, \dot{z}, H) \wedge (s, s') \in \varphi \\
\text{safe}(W) &\stackrel{\text{def}}{=} \forall t \in W.\omega. \text{safe}(t) & \text{safe}(t) &\stackrel{\text{def}}{=}} \forall s'. (t.s, s') \in t.\varphi \Rightarrow s' \notin t.\dot{z} \\
\text{consistent}(W) &\stackrel{\text{def}}{=}} \nexists t \in W.\omega. t.s \in t.\dot{z} \\
\psi \otimes \psi' &\stackrel{\text{def}}{=}} \{(W, h_1 \uplus h'_1, h_2 \uplus h'_2) \mid (W, h_1, h_2) \in \psi \wedge (W, h'_1, h'_2) \in \psi'\} \\
(h_1, h_2) : W &\stackrel{\text{def}}{=}} \vdash h_1 : W.\Sigma_1 \wedge \vdash h_2 : W.\Sigma_2 \wedge (W.k > 0 \Rightarrow (\triangleright W, h_1, h_2) \in \otimes \{t.H(t.s) \mid t \in W.\omega\})
\end{aligned}$$

Fig. 5. Worlds and auxiliary definitions.

Figure taken from: D. Dreyer, G. Neis, and L. Birkedal. **The impact of higher-order state and control effects on local relational reasoning.** *Journal of Functional Programming*, February 2012.

Unary logical relation (for type safety)

Definition in Iris

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e) \triangleq \{\text{True}\} e \{w. \llbracket \tau \rrbracket_{\Delta}(w)\}$$

Unary logical relation (for type safety)

Definition in Iris

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e) \triangleq \{\text{True}\} e \{w. \llbracket \tau \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mathbb{N} \rrbracket_{\Delta}(v) \triangleq v \in \mathbb{N}$$

Unary logical relation (for type safety)

Definition in Iris

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e) \triangleq \{\text{True}\} e \{w. \llbracket \tau \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mathbb{N} \rrbracket_{\Delta}(v) \triangleq v \in \mathbb{N}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket_{\Delta}(v) \triangleq \exists v_1, v_2. v = (v_1, v_2) \wedge \llbracket \tau_1 \rrbracket_{\Delta}(v_1) \wedge \llbracket \tau_2 \rrbracket_{\Delta}(v_2)$$

Unary logical relation (for type safety)

Definition in Iris

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e) \triangleq \{\text{True}\} e \{w. \llbracket \tau \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mathbb{N} \rrbracket_{\Delta}(v) \triangleq v \in \mathbb{N}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket_{\Delta}(v) \triangleq \exists v_1, v_2. v = (v_1, v_2) \wedge \llbracket \tau_1 \rrbracket_{\Delta}(v_1) \wedge \llbracket \tau_2 \rrbracket_{\Delta}(v_2)$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_{\Delta}(v) \triangleq \forall v'. \{\llbracket \tau_1 \rrbracket_{\Delta}(v')\} v v' \{w. \llbracket \tau_2 \rrbracket_{\Delta}(w)\}$$

Unary logical relation (for type safety)

Definition in Iris

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e) \triangleq \{\text{True}\} e \{w. \llbracket \tau \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mathbb{N} \rrbracket_{\Delta}(v) \triangleq v \in \mathbb{N}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket_{\Delta}(v) \triangleq \exists v_1, v_2. v = (v_1, v_2) \wedge \llbracket \tau_1 \rrbracket_{\Delta}(v_1) \wedge \llbracket \tau_2 \rrbracket_{\Delta}(v_2)$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_{\Delta}(v) \triangleq \forall v'. \{\llbracket \tau_1 \rrbracket_{\Delta}(v')\} v v' \{w. \llbracket \tau_2 \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mu X. \tau \rrbracket_{\Delta}(v) \triangleq \mu f. \exists w. v = \text{fold } w \wedge \triangleright \llbracket \tau \rrbracket_{\Delta[X \mapsto f]}(w)$$

$$\llbracket X \rrbracket_{\Delta}(v) \triangleq \Delta(X)(v)$$

Unary logical relation (for type safety)

Definition in Iris

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e) \triangleq \{\text{True}\} e \{w. \llbracket \tau \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mathbb{N} \rrbracket_{\Delta}(v) \triangleq v \in \mathbb{N}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket_{\Delta}(v) \triangleq \exists v_1, v_2. v = (v_1, v_2) \wedge \llbracket \tau_1 \rrbracket_{\Delta}(v_1) \wedge \llbracket \tau_2 \rrbracket_{\Delta}(v_2)$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_{\Delta}(v) \triangleq \forall v'. \{\llbracket \tau_1 \rrbracket_{\Delta}(v')\} v v' \{w. \llbracket \tau_2 \rrbracket_{\Delta}(w)\}$$

$$\llbracket \mu X. \tau \rrbracket_{\Delta}(v) \triangleq \mu f. \exists w. v = \text{fold } w \wedge \triangleright \llbracket \tau \rrbracket_{\Delta[X \mapsto f]}(w)$$

$$\llbracket X \rrbracket_{\Delta}(v) \triangleq \Delta(X)(v)$$

$$\llbracket \text{ref}(\tau) \rrbracket_{\Delta}(v) \triangleq \exists \ell. v = \ell \wedge \boxed{\exists w. \ell \mapsto w * \llbracket \tau \rrbracket_{\Delta}(w)}^{\mathcal{N}. \ell}$$

```

From iris.proofmode Require Import tactics.
From iris.program_logic Require Export weakestpre.
From iris_logrel.F_mu_ref_conc Require Export rules typing.
From iris.algebra Require Import list.
From iris.base_logic Require Import big_op namespaces invariants.
Import uPred.

Definition logN : namespace := nroot .@ "logN".

(** interp : is a unary logical relation. *)
Section logrel.
Context {heapIG Σ}.
Notation D := (valC D -> iProp Σ).
Implicit Types τi : D.
Implicit Types Δ : listC D.
Implicit Types interp : listC D → D.

Program Definition env_lookup (x : var) : listC D -> D := λne Δ,
  from_option id (cconst τ)%I (Δ !! x).
Solve Obligations with solve_proper_alt.

Definition interp_unit : listC D -> D := λne Δ w, (w = UnitV)%I.
Definition interp_nat : listC D -> D := λne Δ w, (∃ n, w = #nv n)%I.
Definition interp_bool : listC D -> D := λne Δ w, (∃ n, w = #bv n)%I.

Program Definition interp_prod
  (interp1 interp2 : listC D -> D) : listC D -> D := λne Δ w,
  (∃ w1 w2, w = PairV w1 w2 ∧ interp1 Δ w1 ∧ interp2 Δ w2)%I.
Solve Obligations with solve_proper.

Program Definition interp_sum
  (interp1 interp2 : listC D -> D) : listC D -> D := λne Δ w,
  ((∃ w1, w = InjLV w1 ∧ interp1 Δ w1) ∨ (∃ w2, w = InjRV w2 ∧ interp2 Δ w2))%I.
Solve Obligations with solve_proper.

Program Definition interp_arrow
  (interp1 interp2 : listC D -> D) : listC D -> D := λne Δ w,
  (□ ∀ v, interp1 Δ v → WP App (of_val w) (of_val v) {{ interp2 Δ }})%I.
Solve Obligations with solve_proper.

Program Definition interp_forall
  (interp : listC D -> D) : listC D -> D := λne Δ w,
  (□ ∀ τi : D,
    ■ (∀ v, PersistentP (τi v)) → WP TApp (of_val w) {{ interp (τi :: Δ) }})%I.
Solve Obligations with solve_proper.

```

```

Definition interp_rec1
  (interp : listC D -> D) (Δ : listC D) (τi : D) : D := λne w,
  (□ (∃ v, w = FoldV v v ∧ > interp (τi :: Δ) v))%I.

Global Instance interp_rec1_contractive
  (interp : listC D -> D) (Δ : listC D) : Contractive (interp_rec1 interp Δ).
Proof.
  intros n τi1 τi2 Hτi w; cbn.
  apply always_ne, exist_ne; intros v; apply and_ne; trivial.
  apply later_contractive -i Hi. by rewrite Hτi.
Qed.

Program Definition interp_rec (interp : listC D -> D) : listC D -> D := λne Δ,
  fixpoint (interp_rec1 interp Δ).

Next Obligation.
  intros interp n Δ1 Δ2 HΔ; apply fixpoint_ne → τi w. solve_proper.
Qed.

Program Definition interp_ref_inv (l : loc) : D -> iProp Σ := λne τi,
  (∃ v, l ↦ v * τi v)%I.
Solve Obligations with solve_proper.

Program Definition interp_ref
  (interp : listC D -> D) : listC D -> D := λne Δ w,
  (∃ l, w = LocV l ∧ inv (logN .@ l) (interp_ref_inv l (interp Δ)))%I.
Solve Obligations with solve_proper.

Fixpoint interp (τ : type) : listC D -> D :=
  match τ return _ with
  | TUnit → interp_unit
  | TNat → interp_nat
  | TBool → interp_bool
  | TProd τ1 τ2 → interp_prod (interp τ1) (interp τ2)
  | TSum τ1 τ2 → interp_sum (interp τ1) (interp τ2)
  | TArrow τ1 τ2 → interp_arrow (interp τ1) (interp τ2)
  | TVar x → env_lookup x
  | TForall τ' → interp_forall (interp τ')
  | TRec τ' → interp_rec (interp τ')
  | Tref τ' → interp_ref (interp τ')
  end.

Notation "[ τ ]" := (interp τ).

Definition interp_env (Γ : list type)
  (Δ : listC D) (vs : list val) : iProp Σ :=
  (length Γ = length vs * [ * ] zip_with (λ τ, [ τ ] Δ) Γ vs)%I.
Notation "[ Γ ]" := (interp_env Γ).

```

```

From iris_logic_ref_conc Require Export logrel_unary.
From iris_logic_ref_conc Require Import rules.
From iris_base_logic Require Export big_op_invariants.
From iris_proofsome Require Import tactics.

```

```

Definition log_typed `heap0 E (f : list type) (e : expr) (t : type) := v & vs,
  env_persistentP &
  heap0_ctx & ! f `w & vs > [ | t | & e.[env_subst vs].

```

```

Notation "t = e : τ" := (log_typed f e t) (at level 74, e, τ at next level).

```

```

Section typed_subst.

```

```

Context `heap0 E.

```

```

Notation D := (valc -> iProp E).

```

```

Local Tactic Notation "smart_wp_bind" uconstr(ctx) ident(v) constr(Hv) uconstr(Hp) :=
  |Apply wp_bind (ctx);

```

```

  |Apply wp_wand_l;

```

```

  |ISplit; [| |Apply Hp; trivial]; [|Intro (v) Hv|ISplit; trivial]; cbv.

```

```

Local Ltac value_case := |Apply wp_value; cbv; rewrite ?to_of_val; trivial|.

```

```

Theorem fundamental_Fx : F `Hv1 <- F `Hv2.

```

```

Proof.

```

```

  induction 1; iIntro (Δ & vs) "#[heap H]" />.

```

```

  |<= some n >
    iDestruct (interp_env_some_l with "H") as (v) "[? | ?]"; first done.
    rewrite /env_subst. simplify_option_eq. by value_case.

```

```

  |<= (unit <=) > value_case; trivial.

```

```

  |<= (nat <=) > value_case; simpl; eauto.

```

```

  |<= (bool <=) > value_case; simpl; eauto.

```

```

  |<= (nat_bin <=) >
    smart_wp_bind (BinOfCtx _ e2.[env_subst vs]) v "#H" Dtyped.

```

```

    smart_wp_bind (BinOfCtx _ v) v' "#H'" Dtyped.

```

```

    iDestruct "Hv" as (n) "#"; iDestruct "Hv'" as (n') "#"; simplify_env.

```

```

    |Apply wp_nat_binop. iNext. iIntro "H".

```

```

    destruct opt; simpl; try destruct eq_nat_dec;

```

```

    try destruct le_dec; try destruct lt_dec; eauto 10.

```

```

  |<= (pair <=) >
    smart_wp_bind (PairOfCtx e2.[env_subst vs]) v "#H" Dtyped.

```

```

    smart_wp_bind (PairOfCtx v) v' "#H'" Dtyped.

```

```

    value_case; eauto.

```

```

  |<= (fct <=) >
    smart_wp_bind (FctCtx v "#H" Dtyped) cbv.

```

```

    iDestruct "Hv" as (w) w'; iMod "#[H]"; subst.

```

```

    |Apply wp_fst; eauto using to_of_val.

```

```

  |<= (s2d <=) >
    smart_wp_bind (S2dCtx v "#H" Dtyped) cbv.

```

```

    iDestruct "Hv" as (w) w'; iMod "#[H]"; subst.

```

```

    |Apply wp_snd; eauto using to_of_val.

```

```

  |<= (l2r <=) >
    smart_wp_bind (L2rCtx v "#H" Dtyped) cbv.

```

```

    value_case; eauto.

```

```

  |<= (l2r <=) >
    smart_wp_bind (InjRCtx v "#H" Dtyped) cbv.

```

```

    value_case; eauto.

```

```

  |<= (l2r <=) >
    smart_wp_bind (InjRCtx v "#H" Dtyped) cbv.

```

```

    value_case; eauto.

```

```

  |<= (case <=) >
    smart_wp_bind (CaseCtx _ j) v "#H" Dtyped; cbv.

```

```

    iDestruct (interp_env_length with "H") as N.

```

```

    iDestruct "Hv" as "[H|H']"; iDestruct "Hv" as (w) "[H|H']"; simplify_env.

```

```

    |Apply wp_case_inj; auto 1 using to_of_val; asimpl. iNext.

```

```

    rewrite typed_subst_head_simpl w naive_solver.

```

```

    |Apply (Dtyped2 Δ (w :: vs)). iSplit; [| |Apply interp_env_cons]; auto.

```

```

    |Apply wp_case_inr; auto 1 using to_of_val; asimpl. iNext.

```

```

    rewrite typed_subst_head_simpl w naive_solver.

```

```

    |Apply (Dtyped2 Δ (w :: vs)). iSplit; [| |Apply interp_env_cons]; auto.

```

```

  |<= (if <=) >
    smart_wp_bind (IfCtx _ j) v "#H" Dtyped; cbv.

```

```

    iDestruct "Hv" as ([]) "#"; subst; simpl.

```

```

    |Apply wp_if_true |Apply wp_if_false; iNext.

```

```

    |Apply (Dtyped2) |Apply (Dtyped2); auto.

```

```

  |<= (rec <=) >

```

```

    value_case; iAlways. simpl. iDob as "D!"; iIntro (w) "#H".

```

```

    iDestruct (interp_env_length with "H") as N.

```

```

    |Apply wp_rec; auto 1 using to_of_val; iNext.

```

```

    asimpl. change (rec _ j with [of_val (Rec v.[env_subst vs])]) at 2.

```

```

    rewrite typed_subst_head_simpl w naive_solver.

```

```

    |Apply (Dtyped Δ ( : w :: vs)). iSplit; [done].

```

```

    |Apply (Dtyped2) & iSplit; [| |Apply interp_env_cons]; auto.

```

```

  |<= (app <=) >

```

```

    smart_wp_bind (AppCtx (e2.[env_subst vs]) v "#H" Dtyped).

```

```

    smart_wp_bind (AppCtx v) v' "#H'" Dtyped.

```

```

    |Apply wp_mono; [| |Apply "H"]; auto.

```

```

  |<= (lstm <=) >

```

```

    iAlways. iIntro (ti) "#". |Apply wp_lstm. iNext.

```

```

    |Apply (Dtyped). iFrame "heap". by |Apply interp_env_rec.

```

```

  |<= (app <=) >

```

```

    smart_wp_bind (AppCtx v "#H" Dtyped) cbv.

```

```

    |Apply wp_wand_r; iSplit.

```

```

    |Apply ("H" $! ([ t' | Δ]); iPureIntro; apply _); cbv.

```

```

    iIntro (w) "T". by |Apply interp_subst.

```

```

  |<= (fold <=) >

```

```

    |Apply (wp_bind [foldCtx]);

```

```

    |Apply wp_wand_l; iSplit; [| |Apply (Dtyped Δ vs)]; auto.

```

```

    iIntro (v) "#H". value_case.

```

```

    rewrite /interp_subst fixpoint_unfold /.

```

```

    iAlways; eauto.

```

```

  |<= (unfold <=) >

```

```

    |Apply (wp_bind [unfoldCtx]);

```

```

    |Apply wp_wand_l; iSplit; [| |Apply (Dtyped) auto].

```

```

    iIntro (v) "#H". rewrite /e_fixpoint_unfold.

```

```

    change (fixpoint _ with ([ Trec t | Δ]); simpl.

```

```

    iDestruct "Hv" as (w) "[H|H']"; subst.

```

```

    |Apply wp_fold; cbv; auto using to_of_val.

```

```

    iNext; iModIntro. by |Apply interp_subst.

```

```

  |<= (fork <=) >

```

```

    |Apply wp_fork. iNext; iSplit; trivial.

```

```

    |Apply wp_wand_l; iSplit; [| |Apply (Dtyped) auto]; auto.

```

```

  |<= (fupd <=) >

```

```

    smart_wp_bind (AllocCtx v "#H" Dtyped) cbv. iClear "H". |Apply wp_fupd.

```

```

    |Apply (wp_alloc with "heap []"); auto 1 using to_of_val.

```

```

    iNext; iIntro (l) "H".

```

```

    iMod (lcm_alloc with "[H1]" as "H");

```

```

    [| |iModIntro; iExists ; iSplit; trivial]; eauto.

```

```

  |<= (load <=) >

```

```

    smart_wp_bind (LoadCtx v "#H" Dtyped) cbv. iClear "H".

```

```

    iDestruct "Hv" as (l) "[H|H']"; subst.

```

```

    iInv (logN <= l) as (a) "[H1 #H2]" "Hclose".

```

```

    |Apply ((wp_load - 1) with "[H1] [Hclose]"); [| |iFrame "heap H1"];

```

```

    trivial. solve_ndisj. iNext.

```

```

    iIntro "H2". iMod ("Hclose" with "[ ]"); eauto.

```

```

  |<= (store <=) >

```

```

    smart_wp_bind (StoreCtx _ j) v "#H" Dtyped; cbv.

```

```

    smart_wp_bind (StoreCtx _ j) v "#H'" Dtyped; cbv. iClear "H".

```

```

    iDestruct "Hv" as (l) "[H|H']"; subst.

```

```

    iInv (logN <= l) as (a) "[H1 #H2]" "Hclose".

```

```

    |Apply (wp_store with "[H1] [Hclose]"); [| |iFrame "heap H1"];

```

```

    by rewrite to_of_val. solve_ndisj. iNext.

```

```

    iIntro "H2". iMod ("Hclose" with "[ ]"); eauto.

```

```

  |<= (cas <=) >

```

```

    smart_wp_bind (CasCtx _ j) v1 "#H1" Dtyped; cbv.

```

```

  |<= (cas <=) >

```

```

    smart_wp_bind (CasCtx _ j) v1 "#H1" Dtyped; cbv.

```

```

    smart_wp_bind (CasCtx _ j) v2 "#H2" Dtyped; cbv.

```

```

    smart_wp_bind (CasCtx _ j) v3 "#H3" Dtyped; cbv. iClear "H".

```

```

    iDestruct "Hv1" as (l) "[H|H']"; subst.

```

```

    iInv (logN <= l) as (a) "[H1 #H2]" "Hclose".

```

```

    destruct (decide (v2 = w)) as [Hreq|Hnot];

```

```

    |Apply (wp_cas_suc with "[H1] [Hclose]"); [| | |iFrame "heap H1"];

```

```

    eauto using to_of_val. solve_ndisj. iNext.

```

```

    |iIntro "H2". iMod ("Hclose" with "[ ]"); eauto.

```

```

    |Apply (wp_cas_fail with "[H1] [Hclose]"); [| | | |iFrame "heap H1"];

```

```

    eauto using to_of_val. solve_ndisj. iNext.

```

```

    iIntro "H2". iMod ("Hclose" with "[ ]"); eauto.

```

```

Qed.

```

```

End typed_interp.

```

Contextual refinement

e_i contextually refines e_s ($\Gamma \vdash e_i \preceq_{ctx} e_s : \tau$):

$$\begin{aligned} \Gamma \vdash e_i \preceq_{ctx} e_s : \tau &\triangleq \Gamma \vdash e_i : \tau \wedge \\ &\Gamma \vdash e_s : \tau \wedge \\ &\forall \mathcal{C} : (\Gamma \vdash \tau \rightsquigarrow \cdot \vdash \mathbf{1}). \mathcal{C}[e_i] \downarrow \Rightarrow \mathcal{C}[e_s] \downarrow \end{aligned}$$

Useful when: e_i is a more efficient version of e_s (e.g., optimized by the compiler) or when e_s is easier to verify than e_i

The idea: Use a binary logical relation such that being related implies contextual refinement

Binary logical relation

To establish contextual refinement

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Expr \rightarrow iProp$

Binary logical relation

To establish contextual refinement

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Expr \rightarrow iProp$

1. Prove *Adequacy*: $\llbracket \tau \rrbracket^{\mathcal{E}}(e, e') \Rightarrow e \Downarrow \Rightarrow e' \Downarrow$

Binary logical relation

To establish contextual refinement

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Expr \rightarrow iProp$

1. Prove *Adequacy*: $\llbracket \tau \rrbracket^{\mathcal{E}}(e, e') \Rightarrow e \Downarrow \Rightarrow e' \Downarrow$
2. Prove *compatibility* lemmas for typing rules, e.g.:

$$\frac{\llbracket \tau_1 \rrbracket^{\mathcal{E}}(e_1, e'_1) \quad \llbracket \tau \rrbracket^{\mathcal{E}}(e_2, e'_2)}{\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{E}}((e_1, e_2), (e'_1, e'_2))}$$

Binary logical relation

To establish contextual refinement

Define *semantics* of types (by recursion on τ): $\llbracket \tau \rrbracket^{\mathcal{E}} : Expr \rightarrow Expr \rightarrow iProp$

1. Prove *Adequacy*: $\llbracket \tau \rrbracket^{\mathcal{E}}(e, e') \Rightarrow e \Downarrow \Rightarrow e' \Downarrow$
2. Prove *compatibility* lemmas for typing rules, e.g.:

$$\frac{\llbracket \tau_1 \rrbracket^{\mathcal{E}}(e_1, e'_1) \quad \llbracket \tau \rrbracket^{\mathcal{E}}(e_2, e'_2)}{\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{E}}((e_1, e_2), (e'_1, e'_2))}$$

3. Corollary (*soundness*): $\Gamma \Vdash e \preceq_{log} e' : \tau \Rightarrow \Gamma \vdash e \preceq_{ctx} e' : \tau$

Binary logical relation (for contextual refinement)

Definition in Iris: value relations

$$\llbracket \mathbb{N} \rrbracket_{\Delta}(v, v') \triangleq v = v' \in \mathbb{N}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket_{\Delta}(v, v') \triangleq \exists v_1, v_2, v'_1, v'_2. v = (v_1, v_2) \wedge v' = (v'_1, v'_2) \wedge \llbracket \tau_1 \rrbracket_{\Delta}(v_1, v'_1) \wedge \llbracket \tau_2 \rrbracket_{\Delta}(v_2, v'_2)$$

$$\llbracket \mu X. \tau \rrbracket_{\Delta}(v, v') \triangleq \mu f. \exists w, w'. v = \mathbf{fold} w \wedge v' = \mathbf{fold} w' \wedge \triangleright \llbracket \tau \rrbracket_{\Delta[X \mapsto f]}(w, w')$$

$$\llbracket X \rrbracket_{\Delta}(v, v') \triangleq \Delta(X)(v, v')$$

$$\llbracket \mathbf{ref}(\tau) \rrbracket_{\Delta}(v, v') \triangleq \exists \ell, \ell'. v = \ell \wedge v' = \ell' \wedge \boxed{\exists w, w'. \ell \mapsto_i w * \ell' \mapsto_s w' * \llbracket \tau \rrbracket_{\Delta}(w, w')}^{\mathcal{N}. \ell. \ell'}$$

Binary logical relation (for contextual refinement)

Definition in Iris: expression relation

The idea¹: simulate the running of the right hand side as *ghost state*

- ▶ Ghost state for threads on the right hand side: $j \mapsto e$
- ▶ Ghost state for the heap of the right hand side: $\ell \mapsto_s v$

¹See: A. Turon, D. Dreyer, and L. Birkedal. **Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency**. In Proceedings of ICFP, 2013.
and M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. **A relational model of types-and-effects in higher-order concurrent separation logic**. In Proceedings of POPL 2017, 2017.

Binary logical relation (for contextual refinement)

Definition in Iris: expression relation

The idea¹: simulate the running of the right hand side as *ghost state*

- ▶ Ghost state for threads on the right hand side: $j \Vdash e$
- ▶ Ghost state for the heap of the right hand side: $\ell \mapsto_s v$

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e, e') \triangleq \forall j, K \{j \Vdash K[e']\} e \{w. \exists w'. j \Vdash K[w'] * \llbracket \tau \rrbracket_{\Delta}(w, w')\}$$

¹See: A. Turon, D. Dreyer, and L. Birkedal. **Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency**. In Proceedings of ICFP, 2013.
and M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. **A relational model of types-and-effects in higher-order concurrent separation logic**. In Proceedings of POPL 2017, 2017.

Binary logical relation (for contextual refinement)

Definition in Iris: expression relation

The idea¹: simulate the running of the right hand side as *ghost state*

- ▶ Ghost state for threads on the right hand side: $j \mapsto e$
- ▶ Ghost state for the heap of the right hand side: $\ell \mapsto_s v$

$$\llbracket \tau \rrbracket_{\Delta}^{\mathcal{E}}(e, e') \triangleq \forall j, K \{j \mapsto K[e']\} e \{w. \exists w'. j \mapsto K[w'] * \llbracket \tau \rrbracket_{\Delta}(w, w')\}$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_{\Delta}(v, v') \triangleq \forall w, w', j, K.$$

$$\{\llbracket \tau_1 \rrbracket_{\Delta}(w, w') * j \mapsto K[v' w']\} v w \{z. \exists z'. j \mapsto K[z'] * \llbracket \tau_2 \rrbracket_{\Delta}(z, z')\}$$

¹See: A. Turon, D. Dreyer, and L. Birkedal. **Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency**. In Proceedings of ICFP, 2013.

and M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. **A relational model of types-and-effects in higher-order concurrent separation logic**. In Proceedings of POPL 2017, 2017.

```

From iris.proofmode Require Import tactics.
From iris.program_logic Require Export weakestpre.
From iris.base_logic Require Export big_op invariants.
From iris_logrel.F_mu_ref_conc Require Export rules_binary typing.
From iris.algebra Require Import list.
From iris.incl Require Import tactics.
Import uPred.

(* NBLOCK: move somewhere else *)
Ltac auto_equiv :=
  (* Deal with "pointwise relation" *)
  repeat lazy match goal with
  | _ = pointwise_relation _ _ _ => intros ?
  end;
  (* Normalize many equalities. *)
  repeat match goal with
  | H : _ = (L _) _ = _ => apply (timeless_iff _ _) in H
  | _ = progress simplify_eq
  end;
  (* repeatedly apply congruence lemmas and use the equalities in the hypotheses. *)
  try (if _equiv) fast_done ! auto_equiv.

Definition logN : namespace := nroot 0 "logN".

{## interp : is a unary logical relation. #}
Section logrel.
Context {heapG I, cfgG I}.
Notation D := (prodC valC valC -> iProp I).
Implicit Types v1 : D.
Implicit Types A : listC D.
Implicit Types interp : listC D -> D.

Definition interp_expr (v1 : listC D -> D) (A : listC D)
  (ee : iExpr -> eExpr) : iProp I := (V J N,
  j => fill K (ee.2) =
  MP ee.1 ((V, J V', j => fill K (of_val v') = v1 A (v, v'))))%N1.
Global Instance interp_expr_ne n :
  Proper (dist n ==> dist n ==> D) := dist n.
Proof. solve_proper. Qed.

Program Definition ctx_lookup (x : var) : listC D -> D := lambda A,
  from_option id (const %N1 (A !! x)).
Solve Obligations with solve_proper_alt.

Program Definition interp_unit : listC D -> D := lambda A wv,
  (wv.1 = UnitV A wv.2 = UnitV)%N1.
Solve Obligations with solve_proper_alt.
Program Definition interp_nat : listC D -> D := lambda A wv,
  (S n : N, wv.1 = #wv n A wv.2 = #wv n)%N1.
Solve Obligations with solve_proper.
Program Definition interp_bool : listC D -> D := lambda A wv,
  (S b : B, wv.1 = #wv b A wv.2 = #wv b)%N1.
Solve Obligations with solve_proper.

Program Definition interp_prod
  (interp1 interp2 : listC D -> D) : listC D -> D := lambda A wv,
  (S wv1 wv2 wv = (PairV (wv1.1) (wv2.1), PairV (wv1.2) (wv2.2)) A
  interp1 & wv1 & interp2 & wv2)%N1.
Solve Obligations with solve_proper.

Program Definition interp_sum
  (interp1 interp2 : listC D -> D) : listC D -> D := lambda A wv,
  ((S wv wv = (InjLV (wv.1), InjRV (wv.2)) A interp1 & wv) v
  (S wv wv = (InjRV (wv.1), InjLV (wv.2)) A interp2 & wv))%N1.
Solve Obligations with solve_proper.

Program Definition interp_arrow
  (interp1 interp2 : listC D -> D) : listC D -> D :=
  lambda A wv,
  (S V wv, interp1 & wv -
  interp_expr
  interp2 & (App (of_val (wv.1)) (of_val (wv.1)),
  App (of_val (wv.2)) (of_val (wv.2))))%N1.
Solve Obligations with solve_proper.

Program Definition interp_forall
  (interp : listC D -> D) : listC D -> D := lambda A wv,
  (S V v1,
  (S V wv, PersistentP (v1 wv) =
  interp_expr
  interp (v1 :: A) (TApp (of_val (wv.1)), TApp (of_val (wv.2)))))%N1.
Solve Obligations with solve_proper.

Program Definition interp_recj
  (interp : listC D -> D) (A : listC D) (v1 : D) : D := lambda wv,
  (S wv wv = (FoldV (wv.1), FoldV (wv.2)) A > interp (v1 :: A) wv)%N1.
Solve Obligations with solve_proper.

Global Instance interp_refl_contractive
  (interp : listC D -> D) (A : listC D) : Contractive (interp_recj interp A).
Proof.
  intros n v1 v2 Hw1 wv1 wv2.
  apply always_ne, exist_ne; intros wv; apply and_ne; trivial.
  apply later_contractive -1 Hw1. by rewrite Hw1.
Qed.

Program Definition interp_rec (interp : listC D -> D) : listC D -> D := lambda A,
  fixpoint (interp_recj interp A).
Next Obligation.
  intros interp n A1 A2 Hw; apply fixpoint_ne - v1 wv. solve_proper.
Qed.

Program Definition interp_ref_inv (l1 : loc * loc) : D -> iProp I := lambda v1,
  (S wv l1.1 => wv.1 & l1.2 => wv.2 & v1 wv)%N1.
Solve Obligations with solve_proper.

Program Definition interp_ref
  (interp : listC D -> D) : listC D -> D := lambda A wv,
  (S l1 wv = (LocV (l1.1), LocV (l1.2)) A
  inv (logN 0 l1) (interp_ref_inv l1 (interp A)))%N1.
Solve Obligations with solve_proper.

fixpoint interp (v : type) : listC D -> D :=
  match v return _ with
  | Unit => interp_unit
  | Nat => interp_nat
  | Bool => interp_bool
  | TProd v1 v2 => interp_prod (interp v1) (interp v2)
  | TSum v1 v2 => interp_sum (interp v1) (interp v2)
  | TArrow v1 v2 => interp_arrow (interp v1) (interp v2)
  | TVar x => ctx_lookup x
  | Tforall v' => interp_forall (interp v')
  | Trec v' => interp_rec (interp v')
  | Tref v' => interp_ref (interp v')
  end.
Notation "[ v ]" := (interp v).

Definition interp_env (f : list type)
  (A : listC D) (wvs : list (val * val)) : iProp I :=
  (length f = length wvs & [a] zip_with (A v, [ v ] A) f wvs)%N1.
Notation "[ f ]" := (interp_env f).

```

```

FundamentalBinary : Type := inductive FundamentalBinary
  | Node : FundamentalBinary → FundamentalBinary → FundamentalBinary
  | Leaf : FundamentalBinary → FundamentalBinary
  | Root : FundamentalBinary → FundamentalBinary
  | ...

-- Constructors
def Node : FundamentalBinary → FundamentalBinary → FundamentalBinary := ...
def Leaf : FundamentalBinary → FundamentalBinary := ...
def Root : FundamentalBinary → FundamentalBinary := ...

-- Properties
def ... := ...
def ... := ...
def ... := ...

```

```

FundamentalBinary : Type := inductive FundamentalBinary
  | Node : FundamentalBinary → FundamentalBinary → FundamentalBinary
  | Leaf : FundamentalBinary → FundamentalBinary
  | Root : FundamentalBinary → FundamentalBinary
  | ...

-- Constructors
def Node : FundamentalBinary → FundamentalBinary → FundamentalBinary := ...
def Leaf : FundamentalBinary → FundamentalBinary := ...
def Root : FundamentalBinary → FundamentalBinary := ...

-- Properties
def ... := ...
def ... := ...
def ... := ...

```

```

FundamentalBinary : Type := inductive FundamentalBinary
  | Node : FundamentalBinary → FundamentalBinary → FundamentalBinary
  | Leaf : FundamentalBinary → FundamentalBinary
  | Root : FundamentalBinary → FundamentalBinary
  | ...

-- Constructors
def Node : FundamentalBinary → FundamentalBinary → FundamentalBinary := ...
def Leaf : FundamentalBinary → FundamentalBinary := ...
def Root : FundamentalBinary → FundamentalBinary := ...

-- Properties
def ... := ...
def ... := ...
def ... := ...

```

```

FundamentalBinary : Type := inductive FundamentalBinary
  | Node : FundamentalBinary → FundamentalBinary → FundamentalBinary
  | Leaf : FundamentalBinary → FundamentalBinary
  | Root : FundamentalBinary → FundamentalBinary
  | ...

-- Constructors
def Node : FundamentalBinary → FundamentalBinary → FundamentalBinary := ...
def Leaf : FundamentalBinary → FundamentalBinary := ...
def Root : FundamentalBinary → FundamentalBinary := ...

-- Properties
def ... := ...
def ... := ...
def ... := ...

```

Examples of refinement of concurrent programs

- ▶ Fine-grained/coarse-grained counter pair

```
let FG_counter =  
  let c = ref 0 in  
    let read () = !c in  
    let rec increment () =  
      let x = !c in if CAS(c, x, x+1) then () else increment ()  
    in (increment, read)
```

```
let CG_counter =  
  let c = ref 0 in let l = make_lock () in  
    let read () = !c in  
    let increment () = acquire l; c := !c + 1; release l  
  in (increment, read)
```

We show: $\llbracket (1 \rightarrow 1) \times (1 \rightarrow \mathbb{N}) \rrbracket_{\emptyset}^{\mathcal{E}}(\text{FG_counter}, \text{CG_counter})$

- ▶ Fine-grained/coarse-grained stack pair with push, pop and iter operations

Trusted computing base

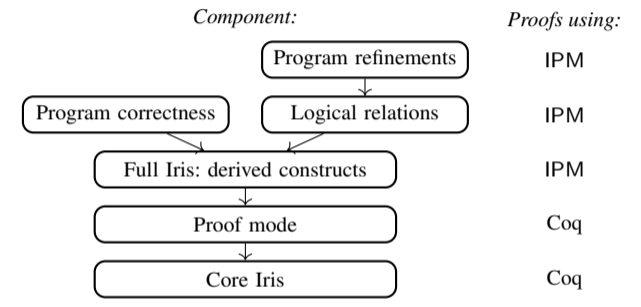


Figure 1. A formally verified stack of abstractions.

The only thing that needs to be trusted is **Coq**

We use the **adequacy** of Iris to prove theorems (contextual refinement, typesafety, ...) in Coq

The refinement proven in Coq

```
Theorem counter_ctx_refinement :  
  []  $\models$  FG_counter  $\leq_{\text{ctx}}$  CG_counter :  
    TProd (TArrow TUnit TUnit) (TArrow TUnit TNat).
```

```
Definition ctx_refines ( $\Gamma$  : list type)  
  (e e' : expr) ( $\tau$  : type) :=  $\forall$  K thp  $\sigma$  v,  
  typed_ctx K  $\Gamma$   $\tau$  [] TUnit  $\rightarrow$   
  rtc step ([fill_ctx K e],  $\emptyset$ ) (of_val v :: thp,  $\sigma$ )  $\rightarrow$   
   $\exists$  thp'  $\sigma'$  v', rtc step ([fill_ctx K e'],  $\emptyset$ ) (of_val v' :: thp',  $\sigma'$ )  
  ).
```

```
Notation " $\Gamma \models e \leq_{\text{ctx}} e' : \tau$ " :=  
  (ctx_refines  $\Gamma$  e e'  $\tau$ ) (at level 74, e, e',  $\tau$  at next level).
```

Future work

- ▶ On the technical side:
 - ▶ Use a more sensible binding representation (we currently use De Bruijn indexes)
 - ▶ Better facilitate symbolic execution for $F_{\mu,ref,conc}$
- ▶ Prove more interesting (and larger) cases of contextual refinements
- ▶ Logical relations for languages with richer type systems and features (e.g., continuations, type-and-effect systems, algebraic effects, ...)
- ▶ Apply it to other application (e.g., non-interference proofs, compiler correctness, secure compilation, ...)
- ▶ Your logical relation applications?! Please do not hesitate to talk to us!

Thanks

Thanks!