

Type Congruence, Duality and Iso-Recursive Binary Session Types

Marco Giunti¹ Nobuko Yoshida²

¹University of Algarve, Portugal

²University of Oxford, UK

GulFest

7 October 2025

Session Types (ST)

- OAuth2: client c sending (1) password (**pwd**) or (2) **cancel** requests to authorisation service a
 1. service a waits for **pwd** from the client c and restarts
 2. service a waits for **cancel** from the client c and ends
- **Session type** of client c :

$$T_c \stackrel{\text{def}}{=} \mu X. (a! \text{pwd}(\text{str}).X + a! \text{cancel}(\text{unit}).\text{end})$$

- $\mu X.T$ is recursive type, $T_1 + T_2$ is choice between T_1 and T_2
- $p!l(S).T$ and $p?l(S).T$ indicate send to/receive from participant p on label l the payload S and continue as T

Equi-recursive ST

- ST follow an **equi-recursive** approach
- Session type of service **a**:

$$T_a \stackrel{\text{def}}{=} \mu X. \underbrace{(c?pwd(str).X + c?cancel(unit).end)}_{\text{body}}$$

- Type **unfolding** is the **instantiation** of X in the **body** with T_a :

$$T_a^* \stackrel{\text{def}}{=} c?pwd(str).T_a + c?cancel(unit).end$$

- Equi-recursion establishes

$$T_a = T_a^* = T_a^{**} = \dots$$

Mechanisation of ST

- **Requires:** mechanised co-induction for equi-recursion
- **Drawback:** utterly complex
- **Drawback:** code hardly extractable to non-lazy languages



This talk: Iso-Recursive ST

- Drawbacks of equi-recursive ST
 - Require mechanised co-induction for equi-recursion
 - Code hardly extractable to non-lazy languages
- Solution provided by iso-recursive ST
 - Type system uses **type congruence** to establish session duality
 - **Rocq mechanisation using Inductive types:**
type system satisfies subject reduction
 - Code can be extracted and ran (e.g. in OCaml)

[https://github.com/marcogiunti/
Iso-Recursive-Session-Types](https://github.com/marcogiunti/Iso-Recursive-Session-Types)

Iso-recursive ST

- In **our setting**, ST follow an **iso-recursive** approach

$$T_a \stackrel{\text{def}}{=} \mu X. R_a$$

$$R_a \stackrel{\text{def}}{=} c?pwd(str).X + c?cancel(unit).end$$

- Type **unfolding** is the instantiation of X in R_a with T_a

$$T_a^* \stackrel{\text{def}}{=} R_a\{T_a/X\} = c?pwd(str).T_a + \\ c?cancel(unit).end$$

- T_a^* **isomorphic** and not equal to T_a : $T_a^* \neq T_a$

$$T_a^{**} = R_a\{T_a^*/X\} = R_a\{(R_a\{T_a/X\})/X\} \dots$$

Typing recursive threads

- **Folded** session process of the **service**: $a \triangleleft P_a$

$$P_a \stackrel{\text{def}}{=} \mu\chi. Q_a \quad Q_a \stackrel{\text{def}}{=} c?\text{pwd}(y). \text{Check}(\chi) + c?\text{cancel}(z)$$

$$\text{Check}(R) \stackrel{\text{def}}{=} \text{if } y = \text{miau} \text{ then } \text{print}@@\text{success}.R \text{ else } \text{print}@@\text{fail}.R$$

- **Unfolded** session process of the **service**: $a \triangleleft P_a^*$

$$P_a^* \stackrel{\text{def}}{=} Q_a\{P_a/\chi\} = c?\text{pwd}(y). \text{Check}(P_a) + c!\text{cancel}(z)$$

- Folded (unfolded) **sessions** have folded (unfolded) **types**

$$\emptyset \vdash P_a : \mu X. (c?\text{pwd}(\text{str}).X + c?\text{cancel}(\text{unit}).\text{end}) \stackrel{\text{def}}{=} T_a$$

$$\emptyset \vdash P_a^* : c?\text{pwd}(\text{str}).T_a + c?\text{cancel}(\text{unit}).\text{end} \stackrel{\text{def}}{=} T_a^*$$

$$\emptyset \not\vdash P_a : T_a^* \quad \emptyset \not\vdash P_a^* : T_a$$

Typing threads composition

- Consider a deployment of the **OAuth2** protocol composed by
 - Folded** client $c \triangleleft P_c$ having folded type T_c
 - Unfolded** authoriser $a \triangleleft P_a^*$ having unfolded type T_a^*
- Composition **typed iff** types are **dual w.r.t. congruence**
- Top level rule for session composition

$$\frac{\Gamma \vdash P_c : T_c \quad \Gamma \vdash P_a^* : T_a^* \quad \bar{c} = a \quad \overline{T_c} \equiv T_a^*}{\Gamma \Vdash c \triangleleft P_c \parallel a \triangleleft P_a^* : \perp}$$

- Intuition for **congruence hypothesis**:
 - Programmer provides a **proof** that the composition is sound
 - Similar to bisimilarity: supply $\mathcal{R}$ s.t. $\overline{T_c} \mathcal{R} T_a^*$ and $\mathcal{R} \subseteq \equiv$
 - Subject reduction (SR) preserved

Type congruence

- A relation $\mathcal{R}$ is a **type equivalence** iff it relates types with their foldings/unfoldings
- E.g. selected cases for $T_1 \mathcal{R} T_2$:
 - $T_1 = \mu X.U_1$ and $T_2 = \mu X.U_2$ imply $U_1 \mathcal{R} U_2$ and $U_1\{\mu X.U_1/X\} \mathcal{R} T_2$ and $T_1 \mathcal{R} U_2\{\mu X.U_2/X\}$
 - $T_1 = p?l(S).U_1 + R_1$ and $T_2 = p?l(S).U_2 + R_2$ imply $U_1 \mathcal{R} U_2$ and $R_1 \mathcal{R} R_2$
 - $T_1 = p!l(S).U_1 + R_1$ and $T_2 = p!l(S).U_2 + R_2$ imply $U_1 \mathcal{R} U_2$ and $R_1 \mathcal{R} R_2$
 - $T_1 = p?l(S).U_1$ and $T_2 = p?l(S).U_2$ imply $U_1 \mathcal{R} U_2$
 - $T_1 = p!l(S).U_1$ and $T_2 = p!l(S).U_2$ imply $U_1 \mathcal{R} U_2$
- **Type congruence** is the union of all symmetric type equivalences



Type congruence in Rocq

Notation $X \$ T := (\text{substT } T \ X \ (\text{typ_mu } X \ T))$ (at level 40).

Definition $\text{equiv } R := \forall t1 \ t2, R \ t1 \ t2 \rightarrow$

match $t1, t2$ **with**

| $\text{typ_mu } X1 \ U1, \text{typ_mu } X2 \ U2 \Rightarrow R \ U1 \ U2 \wedge R \ (X1 \$ U1) \ t2 \wedge R \ t1 \ (X2 \$ U2)$

| $\text{typ_mu } X \ U, _ \Rightarrow R \ (X \$ U) \ t2$ | $_, \text{typ_mu } X \ U \Rightarrow R \ t1 \ (X \$ U)$

| $\text{typ_var } X1, \text{typ_var } X2 \Rightarrow \text{True}$ | $\text{typ_end}, \text{typ_end} \Rightarrow \text{True}$

| $\text{typ_sum } (\text{typ_input } p1 \ l1 \ s1 \ u1) \ r1, \text{typ_sum } (\text{typ_input } p2 \ l2 \ s2 \ u2) \ r2 \Rightarrow$

$p1 = p2 \wedge l1 = l2 \wedge s1 = s2 \wedge R \ u1 \ u2 \wedge R \ r1 \ r2$

| $\text{typ_sum } (\text{typ_output } p1 \ l1 \ s1 \ u1) \ r1, \text{typ_sum } (\text{typ_output } p2 \ l2 \ s2 \ u2) \ r2 \Rightarrow$

$p1 = p2 \wedge l1 = l2 \wedge s1 = s2 \wedge R \ u1 \ u2 \wedge R \ r1 \ r2$

| $\text{typ_input } p1 \ l1 \ s1 \ u1, \text{typ_input } p2 \ l2 \ s2 \ u2 \Rightarrow$

$p1 = p2 \wedge l1 = l2 \wedge s1 = s2 \wedge R \ u1 \ u2$

| $\text{typ_output } p1 \ l1 \ s1 \ u1, \text{typ_output } p2 \ l2 \ s2 \ u2 \Rightarrow$

$p1 = p2 \wedge l1 = l2 \wedge s1 = s2 \wedge R \ u1 \ u2$

| $_, _ \Rightarrow \text{False}$

end.

Definition $\text{struct_equiv } R := \text{equiv } R \wedge \text{symmetric } \text{typ } R.$

Definition $\text{typ_scongr} := \text{union_st } \text{typ } \text{typ } \text{struct_equiv}.$

Notation $"T1 == T2" := (\text{typ_scongr } T1 \ T2)$ (at level 40).

Lemma $\text{equiv_scongr}: \text{equiv } \text{typ_scongr}.$

Example: congruent types

- Take a participant q and labels l_1, l_2
- Let i and b be short for int and bool , respectively.
- Prove that $T_1 \equiv T_2$

$$T_1 \stackrel{\text{def}}{=} \mu X. (q?l_1(b).q!l_1(i).X + U_1)$$

$$U_1 \stackrel{\text{def}}{=} q?l_2(i).q!l_2(i).q!l_2(i).\mu Y.q!l_2(i).Y$$

$$T_2 \stackrel{\text{def}}{=} q?l_1(b).q!l_1(i).T_1 + U_2$$

$$U_2 \stackrel{\text{def}}{=} q?l_2(i).\mu Y.q!l_2(i).Y$$

```

Lemma exist_struct_equiv : ∀ (R : typ → typ → Prop) (t1 t2 : typ),
  struct_equiv R → R t1 t2 → t1 == t2.
Ltac prove_scongr R := match goal with | ⊢ ?W1 == ?W2 =>
  eapply (exist_struct_equiv R); eauto end.

```

Example: congruent types (continue)

- Prove that $T_1 \equiv T_2$ by using the tactic `prove_scongr`

1. Define relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$

$$V_1 \mathcal{R} V_2 = \begin{cases} X \$ U \mathcal{R} V_2 & V_1 = \mu X.U \\ U_1 \mathcal{R} U_2 & V_1 = \mu X.U_1, V_2 = \mu X.U_2 \\ U_1 \mathcal{R} U_2 \wedge R_1 \mathcal{R} R_2 & V_1 = \mathbf{p} ? I(S).U_1 + R_1, \\ & V_2 = \mathbf{p} ? I(S).U_2 + R_2 \\ \dots & (* \text{ selection not needed } *) \\ V_1 = V_2 \end{cases}$$

2. Let $\mathcal{S}$ be symmetric closure of $\mathcal{R}$ and show:

2.1 $\mathcal{S}$ is a symmetric equivalence

2.2 $T_1 \mathcal{S} T_2$

Example: untyped session

- Composition of client and service cannot be typed

$$P_c \stackrel{\text{def}}{=} \mu\chi.(\mathbf{a}!\text{pwd}\langle\text{fido}\rangle.\chi + \mathbf{a}!\text{cancel}\langle\rangle)$$

$$P_a^\# \stackrel{\text{def}}{=} \mathbf{c}?\text{pwd}(y).\mu\chi.\mathbf{c}?\text{pwd}(z).\chi + \mathbf{c}?\text{cancel}(w)$$

$$\mathbf{c} \triangleleft P_c \parallel \mathbf{a} \triangleleft P_a^\# \implies \mathbf{c} \triangleleft P_c \parallel \mathbf{a} \triangleleft \mu\chi.\mathbf{c}?\text{pwd}(z).\chi$$

$$\emptyset \vdash P_c : T_c \stackrel{\text{def}}{=} \mu X.(\mathbf{a}!\text{pwd}(s).X + \mathbf{a}!\text{cancel}(u).\text{end})$$

$$\emptyset \vdash P_a^\# : T_a^\# \stackrel{\text{def}}{=} \mathbf{c}?\text{pwd}(s).\mu\chi.\mathbf{c}?\text{pwd}(s).\chi + \mathbf{c}?\text{cancel}(u)$$

- Claim: $\overline{T_c} \equiv T_a^\# \rightarrow \text{False}$

Example: untyped session (continue)

$$\overline{T_c} \equiv T_a^\# \rightarrow \text{False}.$$

Outline of the Rocq proof ($T_a^* \stackrel{\text{def}}{=} c?\text{pwd}(s).T_a + c?\text{cancel}(u).\text{end}$)

- (unfold) $\overline{T_c} \equiv T_a^\# \rightarrow T_a^* \equiv T_a^\#$ (1)
- (sum-l) (1) $\rightarrow c?\text{pwd}(s).T_a \equiv c?\text{pwd}(s).\mu\chi.c?\text{pwd}(s).\chi$ (2)
- (input) (2) $\rightarrow T_a \equiv \mu\chi.c?\text{pwd}(s).\chi$ (3)
- (unfold) (3) $\rightarrow T_a \equiv c?\text{pwd}(s).\mu\chi.c?\text{pwd}(s).\chi$ (4)
- (unfold) (4) $\rightarrow T_a^* \equiv c?\text{pwd}(s).\mu\chi.c?\text{pwd}(s).\chi$ (5)
- We apply symmetry and Lemma below to (5) and obtain False

Lemma `scongr_symmetric` : symmetric typ typ_scongr.

Lemma `scongr_input_sum_false` : $\forall (p : \text{participant}) (l : \text{label}) (s : \text{styp})$
 $(t \ t1 \ t2 : \text{typ}),$
 $(p \ ? \ l \ s \ . \ t) == \text{typ_sum } t1 \ t2 \rightarrow \text{False}.$

Towards SR: properties of type congruence (1)

Closure under type duality

- If $T_1 \equiv T_2$ then $\overline{T_1} \equiv \overline{T_2}$
- Proof: let $\mathcal{R}$ be equivalence s.t. $T_1 \mathcal{R} T_2$ and $\mathcal{R} \subseteq \equiv$
- We apply the tactic `prove_scongr` to

$$\mathcal{S} \stackrel{\text{def}}{=} \lambda T_1 T_2. \overline{T_1} \mathcal{R} \overline{T_2}$$

- We prove that
 1. $\mathcal{S}$ is a symmetric equivalence
 2. $\overline{T_1} \mathcal{S} \overline{T_2}$

Lemma `scongr_dual` : $\forall T_1 T_2 : \text{typ},$
 $T_1 == T_2 \rightarrow \text{typ_dual } T_1 == \text{typ_dual } T_2.$

Towards SR: properties of type congruence (2)

Closure under type transitions: $T \xrightarrow{\alpha} T$

- Action duality:

$$\frac{}{p?!(v) \mathcal{D} \bar{p}?!(v)} \qquad \frac{}{p?!(v) \mathcal{D} \bar{p}?!(v)}$$

- Let $T_1 \equiv \overline{T_2}$ and $\alpha_1 \mathcal{D} \alpha_2$ and $T_1 \xrightarrow{\alpha_1} U_1$ and $T_2 \xrightarrow{\alpha_2} U_2$.
We have that $U_1 \equiv \overline{U_2}$
- Proof: by induction on $T_1 \xrightarrow{\alpha_1} U_1$ and $T_2 \xrightarrow{\alpha_2} U_2$

Lemma `lts_scongr_dual` : $\forall (T1\ T2 : \text{typ}) (a : \text{proc_action}) (U1\ U2 : \text{typ})$
 $(b : \text{proc_action}),$
 $\text{wf } T1 \rightarrow \text{wf } T2 \rightarrow T1 == \text{typ_dual } T2 \rightarrow \text{typ_lts } T1\ a\ U1 \rightarrow$
 $\text{typ_lts } T2\ b\ U2 \rightarrow \text{action_dual } a\ b \rightarrow U1 == \text{typ_dual } U2.$

Towards SR: value substitution

Closure of typing under value substitution

- Let

1. $\Gamma, x : S \vdash P : U$
2. $\Gamma \vdash v : S$

We have that $\Gamma \vdash P\{v/x\} : U$

- Proof: by induction on $\Gamma, x : S \vdash P : U$

Lemma substitution: $\forall (G : \text{typ_env}) (H : \text{proc_env}) (x : \text{var}) (s : \text{styp})$
 $(P : \text{process}) (t : \text{typ}) (v : \text{value}) (pf : \text{fresh } G \ x):$
 $\text{types } (G, x \multimap s @ pf) \ H \ P \ t \rightarrow \text{stypes } G \ (\text{expr_val } v) \ s \rightarrow$
 $\text{types } G \ H \ (\text{substV } P \ v \ x) \ t.$

Towards SR: process substitution

Closure of typing under process substitution

- Let
 1. $\Gamma, \chi : T \vdash P : U$
 2. $\Gamma \vdash Q : T$

We have that $\Gamma \vdash P\{Q/\chi\} : U$

- Proof: by induction on P

Lemma `types_substP` : $\forall (P\ Q : \text{process}) (T\ U : \text{typ}) (G : \text{typ_env})$
 $(K : \text{proc_env}) (X : \text{proc_var}) (\text{pf} : \text{fresh } K\ x),$
`closedP` $Q \rightarrow \text{disjoint } (\text{bv } P) (\text{bv } Q) \rightarrow \text{disjoint } (\text{bvP } P) (\text{bvP } Q) \rightarrow$
`types` $G\ (K, X =: T @ \text{pf})\ P\ U \rightarrow \text{types } G\ K\ Q\ T \rightarrow$
`types` $G\ K\ (\text{substP } P\ Q\ X)\ U.$

Subject Reduction

- Let $\mathcal{M}$ be a closed session and assume
 1. $\emptyset \Vdash \mathcal{M} : T$ and
 2. $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$
- There is T' s.t.
 3. $T \xrightarrow{\alpha} T'$ and
 4. $\emptyset \Vdash \mathcal{M}' : T'$.
- Proof: by induction on (2) using
 - type preservation of value and process substitution
 - closure of type congruence under duality and type transitions
 - ...

Discussion: iso-recursive sessions and PL

- This work started by studying **iso-recursive theories for (multiparty) session types**
- Two axes
 1. (Binary) Session Types and their mechanisation in Rocq  using **Inductive** constructs
 2. Generalised Multiparty Session Types (*GMPST*) and their automated verification [GY25a,GY25b,GY25c] in Why3 
- (2) showcases how to deploy a certified type checker for GMPST in OCaml with GOSPEL annotations [CFLP19,PR21]
- I believe that (1) can be turned into fully executable type checker by following ideas in (2):
 1. deploy notion of duality up-to type congruence (cf. *cstep* [GY25a])
 2. devise algorithmic variant of type system using type redexes

Thanks!

[https://github.com/marcogiunti/
Iso-Recursive-Session-Types](https://github.com/marcogiunti/Iso-Recursive-Session-Types)

References

- [FP13] Jean-Christophe Filliâtre, Andrei Paskevich: Why3 - Where Programs Meet Provers. ESOP 2013: 125-128
- [CFLP19] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, Mário Pereira: GOSPEL - Providing OCaml with a Formal Specification Language. FM 2019: 484-501
- [PR21] Mário Pereira, António Ravara: Cameleer: A Deductive Verification Tool for OCaml. CAV (2) 2021: 677-689
- [GY25a] Marco Giunti, Nobuko Yoshida: Iso-Recursive Multiparty Sessions and their Automated Verification - Technical Report. CoRR abs/2501.17778 (2025)
- [GY25b] Marco Giunti, Nobuko Yoshida: Iso-Recursive Multiparty Sessions and their Automated Verification. ESOP (1) 2025: 349-378
- [GY25c] Marco Giunti, Nobuko Yoshida: Iso-Recursive Multiparty Sessions and their Automated Verification. Zenodo, 2025