Type Congruence, Duality and Iso-Recursive Binary Session Types

Marco Giunti Nobuko Yoshida

University of Oxford

PLACES
4 May 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:

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

- $\mu X.T$ is recursive type, $T_1 + T_2$ is choice between T_1 and T_2
- $\mathtt{p}!l(S).T$ and $\mathtt{p}?l(S).T$ 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:

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

- Type unfolding is the instantiation of X in the body with $T_{\mathtt{a}}$:

$$T^*_{\mathtt{a}} \stackrel{\text{def}}{=} \mathtt{c}?\text{pwd}(\text{str}).T_{\mathtt{a}} + \mathtt{c}?\text{cancel}(\text{unit}).\text{end}$$

- Equi-recursion establishes

$$T_{\mathtt{a}} = T^*_{\mathtt{a}} = T^{**}_{\mathtt{a}} = \cdots$$

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
  - Mechanisation relies on inductive types
  - Code can be extracted and ran (e.g. in OCaml, cf. Wednesday's talk @ESOP: Concurrency 1)

Iso-recursive ST

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

$$T_{\mathtt{a}} \stackrel{\text{def}}{=} \mu X.R_{\mathtt{a}}$$

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

- Type unfolding is the instantiation of X in $R_{\mathtt{a}}$ with $T_{\mathtt{a}}$

$$T^*_{\mathtt{a}} \stackrel{\text{def}}{=} R_{\mathtt{a}}\{T_{\mathtt{a}}/X\} = \mathtt{c}?\mathsf{pwd}(\mathsf{str}).T_{\mathtt{a}} + \mathtt{c}?\mathsf{cancel}(\mathsf{unit}).\mathsf{end}$$

- $T^*_{\mathtt{a}}$ isomorphic and not equal to $T_{\mathtt{a}}$: $T^*_{\mathtt{a}} \neq T_{\mathtt{a}}$

$$T^{**}_{\mathtt{a}} = R_{\mathtt{a}}\{T^*_{\mathtt{a}}/X\} = R_{\mathtt{a}}\{(R_{\mathtt{a}}\{T_{\mathtt{a}}/X\})/X\}\cdots$$

Typing recursive threads

- Folded session process of the service: $\mathsf{a} \lhd P_{\mathsf{a}}$

$$P_{\mathsf{a}} \stackrel{\text{def}}{=} \mu\chi.Q_{\mathsf{a}} \quad Q_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(y).\mathrm{Check}(\chi) + \mathsf{c}?\mathsf{cancel}(z)$$

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

- Unfolded session process of the service: $\mathsf{a} \lhd P_{\mathsf{a}}^*$

$$P_{\mathsf{a}}^* \stackrel{\text{def}}{=} Q_{\mathsf{a}}\{P_{\mathsf{a}}/\chi\} = \mathsf{c}?\mathsf{pwd}(y).\mathrm{Check}(P_{\mathsf{a}}) + \mathsf{c}!\mathsf{cancel}(z)$$

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

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

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

$$\emptyset \nvdash P_{\mathsf{a}} : T_{\mathsf{a}}^* \qquad \emptyset \nvdash P_{\mathsf{a}}^* : T_{\mathsf{a}}$$

Typing threads composition

- Consider a deployment of the OAuth2 protocol composed by
  1. Folded client $\mathtt{c} \triangleleft P_{\mathtt{c}}$ having folded type $T_{\mathtt{c}}$
  2. Unfolded authoriser $\mathtt{a} \triangleleft P^*_{\mathtt{a}}$ having unfolded type $T^*_{\mathtt{a}}$
- Composition typed iff types are dual w.r.t. congruence
- Top level rule for session composition

$$\frac{\Gamma \vdash P_{\mathtt{c}} : T_{\mathtt{c}} \qquad \Gamma \vdash P^*_{\mathtt{a}} : T^*_{\mathtt{a}} \qquad \overline{\mathtt{c}} = \mathtt{a} \qquad \overline{T_{\mathtt{c}}} \equiv T^*_{\mathtt{a}}}{\Gamma \Vdash \mathtt{c} \triangleleft P_{\mathtt{c}} \parallel \mathtt{a} \triangleleft P^*_{\mathtt{a}} : \bot}$$

- Intuition for congruence hypothesis:
  1. Programmer provides a proof that the composition is sound
  2. Similar to bisimilarity: supply $\mathscr{R}$ s.t. $\overline{T_{\mathtt{c}}}\ \mathscr{R}\ T^*_{\mathtt{a}}$ and $\mathscr{R} \subseteq \equiv$
  3. Subject reduction (SR) preserved

Type congruence

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

Type congruence in Coq

```
1  Notation X $ T := (substT T X (typ_mu X T)) (at level 40).
2  Definition equiv R := ∀ t1 t2, R t1 t2 →
3  match t1, t2 with
4  | typ_mu X1 U1, typ_mu X2 U2 ⇒ R U1 U2 ∧ R (X1 $ U1) t2 ∧ R t1 (X2 $ U2)
5  | typ_mu X U, _ ⇒ R (X $ U) t2 | _, typ_mu X U ⇒ R t1 (X $ U)
6  | typ_var X1, typ_var X2 ⇒ True | typ_end, typ_end ⇒ True
7  | typ_sum (typ_input p1 l1 s1 u1) r1, typ_sum (typ_input p2 l2 s2 u2) r2 ⇒
8  p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2 ∧ R r1 r2
9  | typ_sum (typ_output p1 l1 s1 u1) r1, typ_sum (typ_output p2 l2 s2 u2) r2⇒
10 p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2 ∧ R r1 r2
11 | typ_input p1 l1 s1 u1, typ_input p2 l2 s2 u2 ⇒
12 p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2
13 | typ_output p1 l1 s1 u1, typ_output p2 l2 s2 u2 ⇒
14 p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2
15 | _, _ ⇒ False
16 end.
17 Definition struct_equiv R := equiv R ∧ symmetric typ R.
18 Definition typ_scongr := union_st typ typ struct_equiv.
19 Notation "T1 == T2" := (typ_scongr T1 T2) (at level 40).
20 Check equiv_scongr. equiv_scongr : equiv 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 $T_1 \equiv T_2$

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

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

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

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

```
Check exist_struct_equiv.
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 $\mathscr{R} \subseteq \mathscr{T} \times \mathscr{T}$

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

2. Let $\mathscr{S}$ be symmetric closure of $\mathscr{R}$ and show:
   - 2.1 $\mathscr{S}$ is a symmetric equivalence
   - 2.2 $T_1 \mathscr{S} T_2$

Example: untyped session

- Composition of client and service cannot be typed

$$P_{\mathsf{c}} \stackrel{\text{def}}{=} \mu\chi.(\mathsf{a}!\mathsf{pwd}\langle \mathrm{fido} \rangle.\chi + \mathsf{a}!\mathsf{cancel}\langle\rangle)$$

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

$$\mathsf{c} \lhd P_{\mathsf{c}} \parallel \mathsf{a} \lhd P_{\mathsf{a}}^{\#} \Longrightarrow \mathsf{c} \lhd P_{\mathsf{c}} \parallel \mathsf{a} \lhd \mu\chi.\mathsf{c}?\mathsf{pwd}(z).\chi$$

$$\emptyset \vdash P_{\mathsf{c}} : T_{\mathsf{c}} \stackrel{\text{def}}{=} \mu X.(\mathsf{a}!\mathsf{pwd}(\mathsf{s}).X + \mathsf{a}!\mathsf{cancel}(\mathsf{u}).\mathsf{end})$$

$$\emptyset \vdash P_{\mathsf{a}}^{\#} : T_{\mathsf{a}}^{\#} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(\mathsf{s}).\mu\chi.\mathsf{c}?\mathsf{pwd}(\mathsf{s}).\chi + \mathsf{c}?\mathsf{cancel}(\mathsf{u})$$

- Claim: $\overline{T_{\mathsf{c}}} \equiv T_{\mathsf{a}}^{\#} \to \mathrm{False}$

Example: untyped session (continue)

$\overline{T_c} \equiv T_a^{\#} \to \mathrm{False}.$

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

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

```
Check scongr_symmetric.
scongr_symmetric : symmetric typ typ_scongr
Check scongr_input_sum_false.
scongr_input_sum_false : ∀ (p : participant) (l : label) (s : styp)
(t t1 t2 : typ),
(p ? l s . t) == typ_sum t1 t2 → 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 $\mathscr{R}$ be equivalence s.t. $T_1 \mathscr{R} T_2$ and $\mathscr{R} \subseteq \equiv$
- We apply the tactic `prove_scongr` to

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

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

```
Check scongr_dual.
scongr_dual : ∀ T1 T2 : typ, T1 == T2 → typ_dual T1 == typ_dual T2
```

Towards SR: properties of type congruence (2)

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

- Action duality:

$$\frac{}{\mathtt{p}?l(v) \mathscr{D} \overline{\mathtt{p}}!l\langle v\rangle} \qquad \frac{}{\mathtt{p}!l\langle v\rangle \mathscr{D} \overline{\mathtt{p}}?l(v)}$$

- Let $T_1 \equiv \overline{T_2}$ and $\alpha_1 \mathscr{D} \alpha_2$ 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 $T_2 \xrightarrow{\alpha_2} U_2$

```
Check lts_scongr_dual.
lts_scongr_dual : ∀ (T1 T2 : typ) (a : proc_action) (U1 U2 : typ)
(b : proc_action),
wf T1 → wf T2 → T1 == typ_dual T2 → typ_lts T1 a U1 → typ_lts T2 b U2 →
action_dual a b → U1 == typ_dual U2
```

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

```
Check types_substP.
types_substP : ∀ (P Q : process) (T U : typ) (G : typ_env) (K : proc_env)
(X : proc_var),
¬ In X (bvP P) → closedP Q → K X = Some T → types G K P U →
types G (removeE X K) Q T → types G (removeE X K) (substP P Q X) U
```

Subject Reduction

- Let $\mathscr{M}$ be a closed session and assume
  1. $\Gamma \Vdash \mathscr{M} : T$ and
  2. $\mathscr{M} \xrightarrow{\alpha} \mathscr{M}'$
- We have
  3. $\Gamma \Vdash \mathscr{M}' : T$ or
  4. $T \xrightarrow{\alpha} T'$ and $\Gamma \Vdash \mathscr{M}' : T'$.
- Proof: by induction on (2) using
  - type preservation under 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 theorem provers using `Inductive` constructs
  2. Generalised Multiparty Session Types (*GMPST*) and their automated verification [GY25a,GY25b] in deductive verification tools [PR21,FP13]
- (2) showcases how to deploy a certified type checker for GMPST in OCaml with GOSPEL [CFLP19] annotations

Thanks!

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 2025 (Wed. 7/5 2PM @MDCL 1305)