Iso-Recursive Multiparty Sessions and their Automated Verification

Marco Giunti Nobuko Yoshida

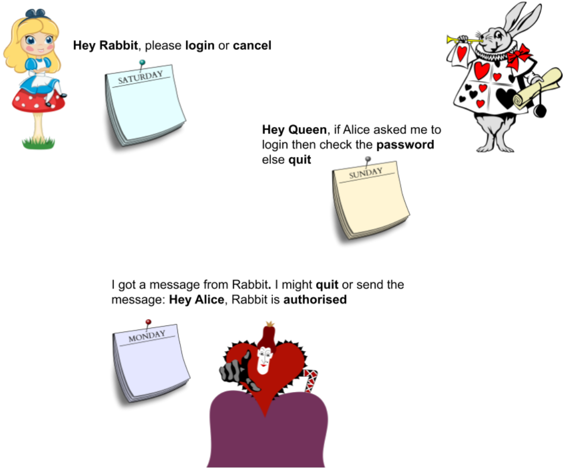
University of Oxford

ESOP

7 May 2025

Multiparty sessions

- OAuth2 example: Alice asks Rabbit to login and waits for Queen's authorisation, or cancels



MultiParty Session Types (*MPST*)

- OAuth2: (Alice) service s sending (1) **login** or (2) **cancel** requests to (Rabbit) client c
  1. service waits for **auth**orisation from the (Queen) authoriser a
  2. service ends

- Session type of service s:

$$T_{\mathtt{s}} \stackrel{\text{def}}{=} \mu X.(\mathtt{c}!\text{login}(\text{unit}).\mathtt{a}?\text{auth}(\text{bool}).X + \mathtt{c}!\text{cancel}(\text{unit}).\text{end})$$

- $\mu X.T$ 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 $\mathtt{p}$ on label l the payload S and continue as T

Equi-recursive MPST

- MPST follow an equi-recursive approach
- Session type of service s:

$$T_s \stackrel{\text{def}}{=} \mu X.(\underbrace{c!\text{login}(\text{unit}).a?\text{auth}(\text{bool}).X + c!\text{cancel}(\text{unit}).\text{end}}_{\text{body}})$$

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

$$T_s^* \stackrel{\text{def}}{=} c!\text{login}(\text{unit}).a?\text{auth}(\text{bool}).T_s + c!\text{cancel}(\text{unit}).\text{end}$$

- Equi-recursion establishes

$$T_s = T_s^* = T_s^{**} = \cdots$$

MPST: Top Down or Bottom Up?

- Original formulation of MPST is Top Down and relies on Global Types that
  - describe entire interaction scenario
  - are projected in local types used for type checking sessions
- Recent works consider Generalised MPST (*GMPST*) or Bottom Up approach:
  - property ϕ holds for a set of participants if ϕ holds for environment built from the participants' types
  - global types are not required

MPST: Top Down or Bottom Up?

	Express.	Complexity	Self-cont.	Mechanis.
Top-Down	✗	✓	✓	✗
Bottom-Up	✓	✗	✗	✗

- Main drawbacks of global types
  - Limited by projectability or implementability
  - Require mechanised co-induction for equi-recursion
- Main drawbacks of generalised MPST
  - Perform worse than global types (PSPACE-hard)
  - Rely on model checkers to establish the environment's properties
  - Require mechanised co-induction for equi-recursion

This talk: Bottom-Up with Iso-Recursive GMPST

	Express.	Complexity	Self-cont.	Mechanis.
Top-Down	✗	✓	✓	✗
Bottom-Up*	✓	✗	✗	✗
Bottom-Up£	✓	?	✓	✓

∗ equi-recursive £ iso-recursive

- Attacked drawbacks of equi-recursive GMPST
  - Rely on model checkers to establish the environment's properties
  - Require mechanised co-induction for equi-recursion
- Solution provided by iso-recursive GMPST
  - The environment's property is checked by the type system (cf. duality in binary session types)
  - Mechanisation relies on inductive types and automated verification

Iso-recursive GMPST

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

$$T_{\mathsf{s}} \stackrel{\text{def}}{=} \mu X.R_{\mathsf{s}}$$
$$R_{\mathsf{s}} \stackrel{\text{def}}{=} \mathsf{c}!\text{login}(\text{unit}).\mathsf{a}?\text{auth}(\text{bool}).X + \mathsf{c}!\text{cancel}(\text{unit}).\text{end}$$

- Type unfolding is the instantiation of X in R_{s} with T_{s}

$$T^*_{\mathsf{s}} \stackrel{\text{def}}{=} R_{\mathsf{s}}\{T_{\mathsf{s}}/X\} = \mathsf{c}!\text{login}(\text{unit}).\mathsf{a}?\text{auth}(\text{bool}).T_{\mathsf{s}} + \mathsf{c}!\text{cancel}(\text{unit}).\text{end}$$

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

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

Typing recursive threads

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

$$P_{\mathsf{s}} \stackrel{\text{def}}{=} \mu\chi.Q_{\mathsf{s}} \qquad Q_{\mathsf{s}} \stackrel{\text{def}}{=} \mathsf{c}!\text{login}\langle\rangle.\mathsf{a}?\text{auth}(x).\chi + \mathsf{c}!\text{cancel}\langle\rangle$$

- Unfolded session process of the service: $\mathsf{s} \lhd P^*_{\mathsf{s}}$

$$P^*_{\mathsf{s}} \stackrel{\text{def}}{=} Q_{\mathsf{s}}\{P_{\mathsf{s}}/\chi\} = \mathsf{c}!\text{login}\langle\rangle.\mathsf{a}?\text{auth}(x).P_{\mathsf{s}} + \mathsf{c}!\text{cancel}\langle\rangle$$

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

$$\emptyset \vdash P_{\mathsf{s}} : \mu X.(\mathsf{c}!\text{login}(\text{unit}).\mathsf{a}?\text{auth}(\text{bool}).X + \mathsf{c}!\text{cancel}(\text{unit}).\text{end}) = T_{\mathsf{s}}$$

$$\emptyset \vdash P^*_{\mathsf{s}} : \mathsf{c}!\text{login}(\text{unit}).\mathsf{a}?\text{auth}(\text{bool}).T_{\mathsf{s}} + \mathsf{c}!\text{cancel}(\text{unit}).\text{end} = T^*_{\mathsf{s}}$$

$$\emptyset \nvdash P_{\mathsf{s}} : T^*_{\mathsf{s}} \qquad \emptyset \nvdash P^*_{\mathsf{s}} : T_{\mathsf{s}}$$

Typing threads composition

- Consider a deployment of the OAuth2 protocol composed by
  1. Unfolded service $\mathsf{s} \lhd P^*_{\mathsf{s}}$ having unfolded type T^*_{s}
  2. Folded client $\mathsf{c} \lhd P_{\mathsf{c}}$ having folded type T_{c}
  3. Folded authoriser $\mathsf{a} \lhd P_{\mathsf{a}}$ having folded type T_{a}
- Composition should be typed iff typings are compliant
- Top level rule for session composition

$$\frac{\Gamma \vdash P^*_{\mathsf{s}} : T^*_{\mathsf{s}} \qquad \Gamma \vdash P_{\mathsf{c}} : T_{\mathsf{c}} \qquad \Gamma \vdash P_{\mathsf{a}} : T_{\mathsf{a}} \\ \Delta = \mathsf{s} : T^*_{\mathsf{s}}, \mathsf{c} : T_{\mathsf{c}}, \mathsf{a} : T_{\mathsf{a}} \qquad \mathrm{comp}(\Delta)}{\Gamma \Vdash \mathsf{s} \lhd P^*_{\mathsf{s}} \parallel \mathsf{c} \lhd P_{\mathsf{c}} \parallel \mathsf{a} \lhd P_{\mathsf{a}} : \Delta}$$

- Three desiderata for comp abstraction:
  1. Is a terminating function
  2. Enforces mismatch-freedom and deadlock-freedom
  3. Can be deployed/verified in mainstream languages and tools

Compliance and termination

- We follow the approach "*types as processes*" and start by defining non-deterministic transitions of the form

$$T \xrightarrow{\alpha} T' \qquad\qquad D \diamond \Delta \xrightarrow{\tau} D\backslash_{\Delta} \diamond \Delta'$$

- D is a decreasing set which is a subset of a fixed point $W \ni \Delta$
- Intuition: W contains unfoldings of iso-recursive types in Δ up-to length n

$$W \supseteq \{\mathtt{s}\colon T_{\mathtt{s}}, \underbrace{\mathtt{s}\colon T_{\mathtt{s}}^{*}, \mathtt{s}\colon T_{\mathtt{s}}^{**}, \ldots, \mathtt{s}\colon T_{\mathtt{s}}^{**\cdots*}}_{n}\}$$

- Termination: if $\Delta \notin D$ then $D \diamond \Delta$ is stuck (cf. ended computation)

Compliance as a function

1. Introduce the notion of deterministic LTS:[1]

   $D \diamond \Delta \xrightarrow{\alpha_1}_d D\backslash_\Delta \diamond \Delta_1$ and $D \diamond \Delta \xrightarrow{\alpha_2}_d D\backslash_\Delta \diamond \Delta_2$ imply $\alpha_1 = \alpha_2$ and $\Delta_1 = \Delta_2$

2. Define relation $\Longrightarrow$ on top of $\longrightarrow_d$. Let $C \overset{\text{def}}{=} D \diamond \Delta$:

   - $C \Longrightarrow C$ whenever C is stuck
   - $C \Longrightarrow \widetilde{C_1}, \widetilde{C_2}$ whenever C does a step and reaches C' with *continuation* C'' and $C' \Longrightarrow \widetilde{C_1}$ and $C'' \Longrightarrow \widetilde{C_2}$

3. Define closure function by stripping decreasing sets:

$$\text{closure}_D(\Delta) = \Delta_1, \ldots, \Delta_n$$

   whenever $D \diamond \Delta \Longrightarrow D_1 \diamond \Delta_1, \ldots, D_n \diamond \Delta_n$

App.

[1] Some parameters are omitted

Compliance and error-freedom

- Compliance is designed to avoid errors
- Let $I = (1, \ldots, n)$, $n \geq 1$, and define

$$\oplus_{i \in I} \mathtt{r}!l_i(S_i).T_i \stackrel{\text{def}}{=} \mathtt{r}!l_1(S_1).T_1 + \cdots + \mathtt{r}!l_n(S_n).T_n$$

$$\&_{i \in I} \mathtt{r}?l_i(S_i).T_i \stackrel{\text{def}}{=} \mathtt{r}?l_1(S_1).T_1 + \cdots + \mathtt{r}?l_n(S_n).T_n$$

- Δ is a communication error if $\exists \mathtt{p}, \mathtt{q}$ s.t.
  - $\Delta(\mathtt{p}) = \oplus_{i \in I} \mathtt{q}!l_i(S_i).T_i$ and $\Delta(\mathtt{q}) = \oplus_{j \in J} \mathtt{p}!l_j(S_j).T_j$ (cf. &)
  - $\Delta(\mathtt{p}) = \oplus_{i \in I} \mathtt{q}!l_i(S_i).T_i$ and $\Delta(\mathtt{q}) = \&_{j \in J} \mathtt{p}?l_j(S_j).T_j$ and $\nexists i, j$ s.t. $l_i @ S_i = l_j @ S_j$ (cf. symmmetric case)
- Δ is a deadlock if $\Delta \xrightarrow{\tau}\!\!\!\!\!\twoheadrightarrow \Delta'$ and there is $\mathtt{p}$ s.t. $\Delta(\mathtt{p}) \neq$ end

Compliance definition

- Remember $\text{closure}_D(\Delta) = \Delta_1, \ldots, \Delta_n$. Define $\text{comp}(\Delta)$:

  if $\Delta_i \in \text{closure}_D(\Delta)$ then Δ_i is not

  1. a communication error
  2. a deadlock

- OAuth example: environment Δ is

$$\begin{aligned}
&\mathtt{s}: \mathtt{c}!\text{login}(\text{u}).\mathtt{a}?\text{auth}(\text{b}).T_{\mathtt{s}} + \mathtt{c}!\text{cancel}(\text{u}).\text{end},\\
&\mathtt{c}: \mu X.(\mathtt{s}?\text{login}(\text{u}).\mathtt{a}!\text{pwd}(\text{s}).X + \mathtt{s}?\text{cancel}(\text{u}).\mathtt{a}!\text{quit}(\text{u}).\text{end}),\\
&\mathtt{a}: \mu X.(\mathtt{c}?\text{pwd}(\text{s}).\mathtt{s}!\text{auth}(\text{b}).X + \mathtt{c}?\text{quit}(\text{u}).\text{end})
\end{aligned}$$

- Closure of Δ w.r.t. fixed point D is

$$\{\Delta, (\mathtt{s}: \text{end}, \mathtt{c}: \text{end}, \mathtt{a}: \text{end})\}$$

- We have $\text{comp}(\Delta)$, e.g. Δ is not a deadlock since

$$\Delta \xrightarrow{\tau} \Delta\backslash_{\mathtt{a}}, \mathtt{a}: \mathtt{c}?\text{pwd}(\text{s}).\mathtt{s}!\text{auth}(\text{b}).T_{\mathtt{a}} + \mathtt{c}?\text{quit}(\text{u}).\text{end}$$

Properties of the type system

- Let $\mathscr{M} = \mathtt{p_1} \triangleleft P_1 \parallel \cdots \parallel \mathtt{p_n} \triangleleft P_n$ be a session
- Let $D \ni \Delta$ be a fixed point of the form $\Delta_1, \cdots, \Delta_m$

1. Subject reduction

   If $\Gamma \Vdash \mathscr{M} : \Delta$ and $\mathscr{M} \xrightarrow{\alpha} \mathscr{M}'$ then

   - $\Gamma \Vdash \mathscr{M}' : \Delta$ or
   - $D \diamond \Delta \xrightarrow{\alpha} D' \diamond \Delta'$ and $\Gamma \Vdash \mathscr{M}' : \Delta'$

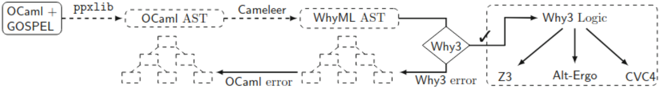
2. Progress

   If $\mathscr{M}$ is closed and $\Gamma \Vdash \mathscr{M} : \Delta$ and $\nexists \mathscr{M}' .\ \mathscr{M} \xrightarrow{\tau} \mathscr{M}'$ then Ended$(\mathscr{M})$

Deployment and verification of compliance

- Challenge: simultaneous implementation and verification

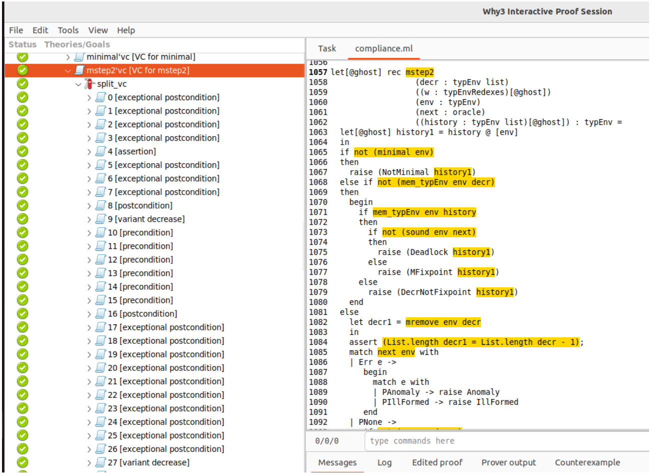
1. Deploy closure and compliance in OCaml by relying on
   - Exception handling to deal with non-deterministic choices
   - Fixed points and history of visited environments
2. Use Cameleer [PR21] pipeline to
   - Annotate OCaml with GOSPEL [CFLP19] specifications
   - Compile functions and specifications into Why3 [FP13]
3. Mechanise proof of:
   - The compliance function terminates
   - If Δ is compliant then the final environment is not an error or a deadlock



Example: closure in Cameleer/Why3

- Specification: 6 pre-conditions, 1 variant, 7 post-conditions
- Verification conditions (*VC*): 505

App.



Decidable Type Checking

- We achieve decidability in two steps

1. Termination of $\Gamma \vdash P\colon T \in \{\top, \bot\}$ by passing a fixed point of type redexes
2. Termination of $\Gamma \Vdash \mathscr{M} : \Delta \in \{\top, \bot\}$ by (1) and by termination of compliance

- Proof mechanised in Cameleer by relying on fixed point parameters
- Production type checker derived by generating fixed points w.r.t. a *max depth* parameter

Future work

	Express.	Complexity	Self-cont.	Mechanis.
Top-Down	✗	✓	✓	✗
Bottom-Up*	✓	✗	✗	✗
Bottom-Up£	✓	?	✓	✓

- Compare complexity w.r.t. other approaches
- Mechanise subject reduction
- Add features
  1. session delegation
  2. (bounded) asynchronous MPST

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)

Deterministic transitions of environments

- Ingredients for deterministic computation:[2]
  1. collect information about discarded branches and selections

- The resulting LTS has the form below, where Δ is minimal

$$D \diamond \Delta \xrightarrow{\alpha} D\backslash_{\Delta} \diamond \Delta_1 \blacktriangleright \Delta_2$$

- Symbol $\blacktriangleright$ is separator of the sum continuation
- Example: deterministic LTS of types

$$\mathsf{c}!\mathsf{login}(\mathsf{u}).\mathsf{a}?\mathsf{auth}(\mathsf{b}).T_{\mathsf{s}} + \mathsf{c}!\mathsf{canc}(\mathsf{u}).\mathsf{end} \xrightarrow{\mathtt{c!canc}\langle\rangle}_d \mathsf{end} \blacktriangleright \mathsf{c}!\mathsf{login}(\mathsf{u}).\mathsf{a}?\mathsf{auth}(\mathsf{b}).T_{\mathsf{s}}$$

Back

[2] Some items are omitted.

Environment closure

- Closure $\mathscr{C}$ relies on semi-reflexive transitive relation $\Longrightarrow$
- Consider the configuration $C \overset{\text{def}}{=} D \diamond \Delta$

1. C is stuck: $C \Longrightarrow C$
2. If
   - C moves to $D\backslash_\Delta \diamond \Delta_1 \blacktriangleright \Delta_2$ and
   - $D\backslash_\Delta \diamond \Delta_1 \Longrightarrow \widetilde{E_1}$ and
   - $D\backslash_\Delta \diamond \Delta_2 \Longrightarrow \widetilde{E_2}$

   then $C \Longrightarrow \widetilde{E_1}, \widetilde{E_2}$

Back

Example: behavioural specification of fixed points

```
type typRedexes = typ list

(* sterling X T = T{μX.T/X} *)
let[@logic] sterling x t = substT t x t

let[@logic] rec produceRedexes (t : typ) (n : int) : typRedexes  =
if n ≤ 0 then []
else let m = n − 1 in t ::
match t with
| Typ_input (_, _, _, r) | Typ_output (_, _, _, r) → produceRedexes r m
| Typ_mu (x, r) → produceRedexes (sterling x r) m
| Typ_sum (r1, r2) → produceRedexes r1 m @ produceRedexes r2 m
| Typ_end | Typ_var _ → []
(*@ m = produceRedexes t n
ensures n > 0 → t ∈ m
ensures n > 1 →
(∀ l p s r. t = Typ_input l p s r → r ∈ m) ∧
(∀ l p s r. t = Typ_output l p s r → r ∈ m) ∧
(∀ x r. t = Typ_mu x r → sterling x r ∈ m) ∧
(∀ r1 r2. t = Typ_sum r1 r2 → r1 ∈ m ∧ r2 ∈ m)
variant n *)
```

Mechanised closure

```
let[@logic] rec pre = function | [_] → [] | s :: tl → s :: pre tl
(*@ m = pre param \n requires param ≠ ∅ \n variant param *)
let[@logic] rec last = function | [x] → x | _ :: tl → last tl
(*@ m = last param \n requires param ≠ ∅ \n variant param *)

(* Some parameters and exceptions are omitted *)
let[@ghost] rec mstep (decr : typEnv list) ((w : typEnvRedexes)[@ghost])
(env : typEnv) ((history : typEnv list)[@ghost]) : typEnv = ···
(*@ m = mstep decr w env history
requires nodup decr
requires mfixpoint w env (|decr| * 2)
requires env ∈ combinations w
requires decr ∩ history = ∅
requires decr ∪ history = combinations w
variant |decr|
raises MFixpoint h → h ≠ ∅ ∧ last h ∈ pre h ∧ error_free (last h)
raises Deadlock h → h ≠ ∅ ∧ (last h ∈ pre h ∧ ¬ error_free (last h)
  ∨ stuck (last h) ∧ ¬ consumed (last h))
raises WrongBranch h → h ≠ ∅ ∧
  ∃ p q t1 t2. typOf p (last h) = Some t1 ∧ typOf q (last h) = Some t2
  ∧ error t1 t2 ···
ensures consumed m *)
```

Mechanised Compliance

Post-conditions of mstep: Back

```
raises MFixpoint h → h ≠ ∅ ∧ last h ∈ pre h ∧ error_free (last h)
raises Deadlock h → h ≠ ∅ ∧ last h ∈  pre h ∧ ¬ error_free (last h)
  ∨ stuck (last h) ∧ ¬ consumed (last h)
raises WrongBranch h → h ≠ ∅ ∧ ∃p q t1 t2. typOf p (last h) = Some t1
∧   typOf q (last h) = Some t2 ∧ error t1 t2
ensures consumed m

(* Some parameters and exceptions are omitted *)
let[@ghost] compliance (all_combs : typEnv list)
((w : typEnvRedexes)[@ghost]) (env : typEnv) : bool =
try let m = mstep all_combs w env ([] : typEnv list) in consumed m
with
| MFixpoint hist → let e = last hist in let h0 = pre hist in
e ∈ h0 ∧ error_free e
| Deadlock _ → raise NotCompliant | WrongBranch _ → raise NotCompliant
(*@ m = compliance all_combs w env next
requires all_combs = combinations w
requires nodup all_combs
requires mfixpoint w env (|all_combs| * 2)
requires env ∈ all_combs
raises NotCompliant → true
ensures m = true *)
```