A Synthetic Reconstruction of Multiparty Session Types

David Castro-Pérez
d.castro-perez@kent.ac.uk

Francisco Ferreira
francisco.ferreiraruiz@rhul.ac.uk

Sung-Shik Jongmans
s.s.t.q.jongmans@rug.nl

PLAS Seminar, 19-01-2026

Concurrency is hard!

- Deadlocks
- Protocol violations
- Resource contention
- etc.

Our work:

- Safety and liveness of message-passing concurrent programs
- A novel Multiparty Session Type system
- Full Agda mechanisation
- An implementation in Rascal

The Problem

```
P := send Q; ...
Q := receive P; receive S; send R; send S; ...
R := receive Q; send S; ...
S := receive R; receive Q; send Q; ...
system := P | Q | R | S
```

The Problem

```
P := send Q; ...
Q := receive P; receive S; send R; send S; ...
R := receive Q; send S; ...
S := receive R; receive Q; send Q; ...
system := P | Q | R | S
```

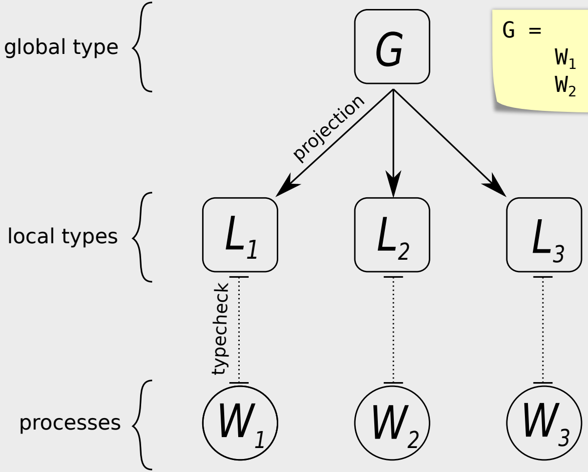
Question: Is `system` safe?

The Problem

```
P := send Q; ...
Q := receive P; receive S; send R; send S; ...
R := receive Q; send S; ...
S := receive R; receive Q; send Q; ...
system := P | Q | R | S
```

Question: Is `system` safe? **NO!**

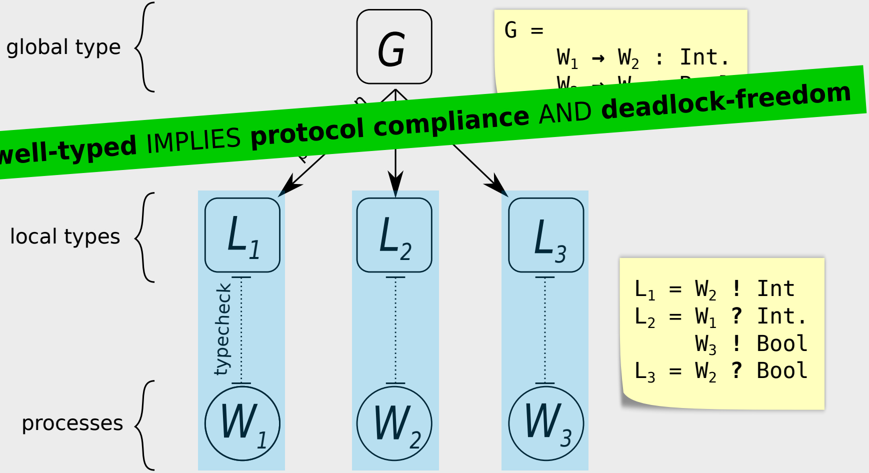
Multiparty Session Types (in a nutshell)



$$
\begin{aligned}
\mathtt{G} = \;& \\
& \mathtt{W}_1 \to \mathtt{W}_2 : \mathtt{Int}. \\
& \mathtt{W}_2 \to \mathtt{W}_3 : \mathtt{Bool}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{L}_1 &= \mathtt{W}_2 \; ! \; \mathtt{Int} \\
\mathtt{L}_2 &= \mathtt{W}_1 \; ? \; \mathtt{Int}. \\
& \;\mathtt{W}_3 \; ! \; \mathtt{Bool} \\
\mathtt{L}_3 &= \mathtt{W}_2 \; ? \; \mathtt{Bool}
\end{aligned}
$$

Multiparty Session Types (in a nutshell)



well-typed IMPLIES **protocol compliance** AND **deadlock-freedom**

MPST in more detail

Roles			$\mathsf{p}, \mathsf{q}, \ldots$	
Sorts	S	$\coloneqq$	$\texttt{bool} \mid \texttt{nat} \mid \ldots$	Basic data types.
Global Types	G	$\coloneqq$	$\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i \in I}$	Communication.
		$\mid$	$\mu X.G$	Recursion.
		$\mid$	X	Variable.
		$\mid$	$\varnothing$	End of protocol.
Local Types	L	$\coloneqq$	$\mathsf{p}!\{\ell_i(S_i).L_i\}_{i \in I}$	Send.
		$\mid$	$\mathsf{q}?\{\ell_i(S_i).L_i\}_{i \in I}$	Receive.
		$\mid$	$\mu X.L$	Recursion.
		$\mid$	X	Recursion variable.
		$\mid$	$\varnothing$	End of protocol.

Projection

$$\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I} \upharpoonright \mathsf{r} = \begin{cases} \mathsf{q}!\{\ell_i(S_i).G_i \upharpoonright \mathsf{r}\}_{i\in I} & (\mathsf{r} = \mathsf{p} \wedge \qquad\quad \wedge\, \mathsf{p} \neq \mathsf{q}) \\ \mathsf{p}?\{\ell_i(S_i).G_i \upharpoonright \mathsf{r}\}_{i\in I} & (\qquad\quad \wedge\, \mathsf{r} = \mathsf{q} \wedge \mathsf{p} \neq \mathsf{q}) \\ \sqcap_{i\in I}(G_i \upharpoonright \mathsf{r}) & (\mathsf{r} \neq \mathsf{p} \wedge \mathsf{r} \neq \mathsf{q} \wedge \mathsf{p} \neq \mathsf{q}) \end{cases}$$

$$\mu X.G \upharpoonright \mathsf{r} = \begin{cases} \mu X.G \upharpoonright \mathsf{r} & (\mathsf{r} \in G) \\ \varnothing & (\mathsf{r} \notin G) \end{cases} \qquad X \upharpoonright \mathsf{r} = X \qquad \varnothing \upharpoonright \mathsf{r} = \varnothing$$

Projection

$$p \to q : \{\ell_i(S_i).G_i\}_{i\in I} \upharpoonright r = \begin{cases} q!\{\ell_i(S_i).G_i \upharpoonright r\}_{i\in I} & (r = p \land \qquad \land p \neq q) \\ p?\{\ell_i(S_i).G_i \upharpoonright r\}_{i\in I} & (\qquad \land r = q \land p \neq q) \\ \sqcap_{i\in I}(G_i \upharpoonright r) & (r \neq p \land r \neq q \land p \neq q) \end{cases}$$

$$\mu X.G \upharpoonright r = \begin{cases} \mu X.G \upharpoonright r & (r \in G) \\ \varnothing & (r \notin G) \end{cases} \qquad X \upharpoonright r = X \qquad \varnothing \upharpoonright r = \varnothing$$

$$p?\{\ell_i(S_i).L_i\}_{i\in I} \sqcap p?\{\ell_j(S_j).L'_j\}_{j\in J}$$
$$= p?\{\ell_i(S_i).L_i\}_{i\in I\setminus J} \cup \{\ell_j(S_j).L'_j\}_{j\in J\setminus I} \cup \{\ell_i(S_i).L_i \sqcap L'_i\}_{i\in I\cap J}$$

$$p!\{\ell_i(S_i).L_i\}_{i\in I} \sqcap p!\{\ell_i(S_i).L'_i\}_{i\in I} = p!\{\ell_i(S_i).L_i \sqcap L'_i\}_{i\in I}$$

$$\mu X.L \sqcap \mu X.L' = \mu X.(L \sqcap L') \qquad L \sqcap L = L$$

Projection

$$p \to q : \{\ell_i(S_i).G_i\}_{i\in I} \restriction r = \begin{cases} q!\{\ell_i(S_i).G_i \restriction r\}_{i\in I} & (r = p \land \qquad \land p \neq q) \\ p?\{\ell_i(S_i).G_i \restriction r\}_{i\in I} & (\qquad \land r = q \land p \neq q) \\ \sqcap_{i\in I}(G_i \restriction r) & (r \neq p \land r \neq q \land p \neq q) \end{cases}$$

It gets complicated very quickly!

$$\varnothing \qquad (r \notin G)$$

$$p?\{\ell_i(S_i).L_i\}_{i\in I} \sqcap p?\{\ell_j(S_j).L'_j\}_{j\in J}$$
$$= p?\{\ell_i(S_i).L_i\}_{i\in I\setminus J} \cup \{\ell_j(S_j).L'_j\}_{j\in J\setminus I} \cup \{\ell_i(S_i).L_i \sqcap L'_i\}_{i\in I\cap J}$$

$$p!\{\ell_i(S_i).L_i\}_{i\in I} \sqcap p!\{\ell_i(S_i).L'_i\}_{i\in I} = p!\{\ell_i(S_i).L_i \sqcap L'_i\}_{i\in I}$$

$$\mu X.L \sqcap \mu X.L' = \mu X.(L \sqcap L') \qquad L \sqcap L = L$$

Projection (Example)

Consider the following protocol

$$\mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{ll} \mathsf{REQ}(\texttt{nat}).\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\texttt{bool}).X \\ \mathsf{END}() \quad .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\texttt{done} \end{array} \right\}$$

Projection (Example)

Consider the following protocol

$$\mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{ll} \mathsf{REQ}(\mathtt{nat}) & .\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\mathtt{bool}).X \\ \mathsf{END}() & .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\mathtt{done} \end{array} \right\}$$

Projecting r

$$\mu X.(\mathsf{q}?\mathsf{REQ}(\mathtt{bool}).X) \sqcap (\mathsf{q}?\mathsf{END}().\varnothing)$$

$$=$$

Projection (Example)

Consider the following protocol

$$\mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{ll} \mathsf{REQ}(\mathtt{nat}).\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\mathtt{bool}).X \\ \mathsf{END}() \quad .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\mathtt{done} \end{array} \right\}$$

Projecting r

$$\mu X.(\mathsf{q}?\mathsf{REQ}(\mathtt{bool}).X) \sqcap (\mathsf{q}?\mathsf{END}().\varnothing)$$

$$= \mu X.\mathsf{q}?\left\{ \begin{array}{ll} \mathsf{REQ}(\mathtt{bool}).X \\ \mathsf{END}() & .\mathtt{done} \end{array} \right\}$$

Processes and Typing

$$
\begin{array}{llll}
\text{Process} \quad P & := & \mathsf{p}\,!\,\ell\langle e\rangle.P & \text{Send a message.} \\
& \mid & \sum_{i \in I} \mathsf{p}?\ell_i(x_i).P_i & \text{Receive a message.} \\
& \mid & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P' & \text{Conditional process.} \\
& \mid & \texttt{rec}\ X\,.\,P & \text{Recursive process.} \\
& \mid & X & \text{Recursion variable.} \\
& \mid & \texttt{done} & \text{Inactive process.}
\end{array}
$$

Process Typing (simplified)

Once we have local types, process typing is simple:

$$\textsc{T-Send}\ \frac{\Gamma \vdash P : L_i \qquad \Gamma \vdash e : S_i \qquad i \in I}{\Gamma \vdash \mathsf{q} \mathbin{!} \ell_i\langle e\rangle.P : (\mathsf{p}!\{\ell_i(S_i).L_i\}_{i\in I})}$$

$$\textsc{T-Recv}\ \frac{\forall(i \in I), [\Gamma, x_i : S_i \vdash P_i : L_i]\, z}{\Gamma \vdash \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i : (\mathsf{p}?\{\ell_i(S_i).L_i\}_{i\in I})}$$

Process Typing (simplified)

Local types and processes are so similar that some developments omit them, and projection produces directly processes.

Once we have local types, process typing is simple:

$$\textsc{T-Send}\ \frac{\Gamma \vdash P : L_i \qquad \Gamma \vdash e : S_i \qquad i \in I}{\Gamma \vdash \mathsf{q} \,!\, \ell_i\langle e\rangle.P : (\mathsf{p}!\{\ell_i(S_i).L_i\}_{i\in I})}$$

$$\textsc{T-Recv}\ \frac{\forall(i \in I), [\Gamma, x_i : S_i \vdash P_i : L_i]\, z}{\Gamma \vdash \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i : (\mathsf{p}?\{\ell_i(S_i).L_i\}_{i\in I})}$$

Process Typing (simplified)

Lo

Once

T-

Γ

Γ $_{i\in I})$

Deconfined Global Types for Asynchronous Sessions

Francesco Dagnino[1], Paola Giannini[2], and Mariangiola Dezani-Ciancaglini[3]

[1] DIBRIS, Università di Genova, Italy
[2] DiSIT, Università del Piemonte Orientale, Alessandria, Italy
[3] Dipartimento di Informatica, Università di Torino, Italy

Problems with Classical Formulation

1. **Too syntactic:**
   - Processes and local types must align
   - Too restrictive, rules out correct processes
   - ...
2. **Unnecessarily complex:**
   - Hard to implement/mechanise, e.g.:
     - Use of runtime coinductive global types: Our PLDI 2021 paper, Jacobs et al. (2022).
     - Graph-based reasoning and decision procedure for the equality of recursive types: Tirore et al. (2023)
   - Hard to extend
3. **Imprecise** (coinduction, safety)

Example of Imprecision in Classical MPST

$\boxed{\text{Equirecursion: ``We identify } \mu X.G \text{ with } [\mu X.G/X]G\text{''}}$

Common statement in proofs about MPST, but...

1. The rules specify how to deal with variables X
2. The rules specify when and how to unfold $\mu X.G$

Moreover: Equirecursion alone distinguishes too many protocols that "are the same":

$$\mathsf{p} \to \mathsf{q} : \mathsf{p}' \to \mathsf{q}' : G \neq \mathsf{p}' \to \mathsf{q}' : \mathsf{p} \to \mathsf{q} : G$$

Mechanising the classical theory of MPST is notoriously hard, in part due to this.

Another Example of Imprecision in Classical MPST

This source of imprecision **did** cause flawed proofs in the literature.

Preservation theorem:

$$\begin{array}{l} \varphi(L_1, \ldots, L_n) \wedge (P_1 \mid \ldots \mid P_n \xrightarrow{\alpha} P'_1 \mid \ldots \mid P'_n) \\ \wedge(\vdash P_1 : L_1 \wedge \ldots \wedge \vdash P_n : L_n) \end{array} \implies \begin{array}{l} \exists L_1{}' \cdots L_n{}' \wedge (L_1 \mid \ldots \mid L_n \xrightarrow{\alpha} L'_1 \mid \ldots \mid L'_n) \\ \wedge(\vdash P'_1 : L_1{}' \wedge \ldots \wedge \vdash P'_n : L_n{}') \end{array}$$

Another Example of Imprecision in Classical MPST

This source of imprecision **did** cause flawed proofs in the literature.

Preservation theorem:

$$\begin{array}{l}\varphi(L_1, \ldots, L_n) \wedge (P_1 \mid \ldots \mid P_n \xrightarrow{\alpha} P_1' \mid \ldots \mid P_n') \\ \wedge(\vdash P_1 : L_1 \wedge \ldots \wedge \vdash P_n : L_n)\end{array} \quad \Longrightarrow \quad \begin{array}{l}\exists {L_1}' \cdots {L_n}' \wedge (L_1 \mid \ldots \mid L_n \xrightarrow{\alpha} L_1' \mid \ldots \mid L_n') \\ \wedge(\vdash P_1' : {L_1}' \wedge \ldots \wedge \vdash P_n' : {L_n}')\end{array}$$

Implicit assumption:

$$(\forall i, L_i = G \upharpoonright \mathsf{r_i}) \Longrightarrow \varphi(L_1, \ldots, L_n)$$

Another Example of Imprecision in Classical MPST

This source of imprecision **did** cause flawed proofs in the literature.

Preservation theorem:

$\varphi(L_1, \ldots$ … $_n \xrightarrow{\alpha} L'_1 \mid \ldots \mid L'_n)$

$\wedge(\vdash P_1 :$ … $_n' : L_n{}')$

This assumption is **wrong** (Scalas & Yoshida, POPL'19)!

- Trivially holds for the basic case.
- Breaks as soon as you extend the theory slightly (e.g. full merge).

Implicit assumption:

$$(\forall i, L_i = G \upharpoonright \mathsf{r_i}) \Longrightarrow \varphi(L_1, \ldots, L_n)$$

A Few Attempts at Simplifying the Theory

Deconfined Global Types for Asynchronous Sessions

Francesco Dagnino[1], Paola Giannini[2], and Mariangiola Dezani-Ciancaglini[3]

[1] DIBRIS, Università di Genova, Italy
[2] DiSIT, Università del Piemonte Orientale, Alessandria, Italy
[3] Dipartimento di Informatica, Università di Torino, Italy

A Few Attempts at Simplifying the Theory

Less Is More: Multiparty Session Types Revisited

ALCESTE SCALAS, Imperial College London, UK
NOBUKO YOSHIDA, Imperial College London, UK

[2] DiSIT, Università del Piemonte Orientale, Alessandria, Italy
[3] Dipartimento di Informatica, Università di Torino, Italy

A Few Attempts at Simplifying the Theory

Less is More Revisited

Association with Global Multiparty Session Types

Nobuko Yoshida and Ping Hou

University of Oxford, Oxford, UK

{nobuko.yoshida,ping.hou}@cs.ox.ac.uk

Our Approach: Synthetic Typing

Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types

Sung-Shik Jongmans ✉
Department of Computer Science, Open University, Heerlen, The Netherlands
Centrum Wiskunde & Informatica (CWI), NWO-I, Amsterdam, The Netherlands

Francisco Ferreira ✉
Department of Computer Science, Royal Holloway, University of London, UK

Our Approach: Synthetic Typing

Synthetic Behavioural Typing: Sound, Regular

<u>Our Contributions</u>:

- "Free" typing from being tied up to the syntax of local types.
- An MPST system that avoids projection/merging/etc.
- Type-checking against arbitrary (well-formed) LTSs.
- Well-formedness/deadlock-freedom is decided by typeability, not by projectability.
- Mechanisation in Agda.
- Implementation in Rascal.

(Slightly Simplified) Core SyntheticTyping Rules

$$\textsc{T-Send}\ \frac{\Gamma \vdash P : G' \upharpoonright \mathsf{p} \qquad G \xrightarrow{\mathsf{p}\to\mathsf{q}:\ell(S)} G' \qquad \Gamma \vdash e : S}{\Gamma \vdash \mathsf{q}\,!\,\ell\langle e\rangle.P : G \upharpoonright \mathsf{p}}$$

$$\textsc{T-Recv}\ \frac{(G \text{ allows } \mathsf{p} \to \mathsf{q} : \ell_j, \text{ for some j}) \qquad \forall\, i\ G'\ (G \xrightarrow{\mathsf{q}\to\mathsf{p}:\ell_i(S_i)} G');\ [\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright \mathsf{p}]}{\Gamma \vdash \sum_{i\in I} \mathsf{q}?\ell_i(x_i).P_i : G \upharpoonright \mathsf{p}}$$

$$\textsc{T-Skip}\ \frac{(\text{It is safe for } \mathsf{p} \text{ to wait in } G) \qquad \forall (G \xrightarrow{\overline{\{\mathsf{p}\}}} {*}G' \xrightarrow{\{\mathsf{p}\}});\ [\Gamma \vdash P : G' \upharpoonright \mathsf{p}]}{\Gamma \vdash P : G \upharpoonright \mathsf{p}}$$

Synthetic Behavioural Typing

Key idea: The syntax of G is irrelevant!

Synthetic Behavioural Typing

Key idea: The syntax of G is irrelevant!

G is just the state of a **labelled transition system (LTS)**!

Well-behavedness

Our type system is parameterised by an LTS, where labels must specify send/receive interactions.

Well-behavedness

Our type system is parameterised by an LTS, where labels must specify send/receive interactions.

But not all LTSs are valid types! We can only guarantee safety/liveness for **well-behaved** LTSs.

Well-behavedness

Our type system is parameterised by an LTS, where labels must specify send/receive interactions.

But not all LTSs are valid types! We can only guarantee safety/liveness for **well-behaved** LTSs.

Well-behavedness:

1. **Sender determinacy**: If a state allows multiple transitions, these cannot have the same receiver but different senders.
2. **Determinism**: A state can have at most one transition with the same action label.
3. **Conditional commutativity**: In any state, if a later independent action has an earlier enabled branch, it can be commuted earlier.
4. **Diamond property**.
5. **"Stepback" property?** (not in the paper – likely an artifact of our mechanisation only used to prove one minor case in our mechanisation). "Bisimilarity is preserved when moving to past states" (?)

Well-behavedness

Our type system is parameterised by an LTS, where labels must specify send/receive interactions.

But not all LTSs are valid types! We can only guarantee safety/liveness for **well-behaved** LTSs.

Well-behavedness:

1. **Sender determinacy**: If a state allows multiple transitions, these cannot have the same receiver but different senders.
2. **Dete**
3. **Conc** enab

Every syntactic global type in the classical theory of MPST is well-behaved!

4. **Diamond property**.
5. **"Stepback" property?** (not in the paper – likely an artifact of our mechanisation only used to prove one minor case in our mechanisation). "Bisimilarity is preserved when moving to past states" (?)

Example

$$G = \mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{l} \mathsf{REQ}(\texttt{nat}).\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\texttt{bool}).X \\ \mathsf{END}() \qquad .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\texttt{done} \end{array} \right\}$$

We are going to typecheck a process implementing role r...

Example

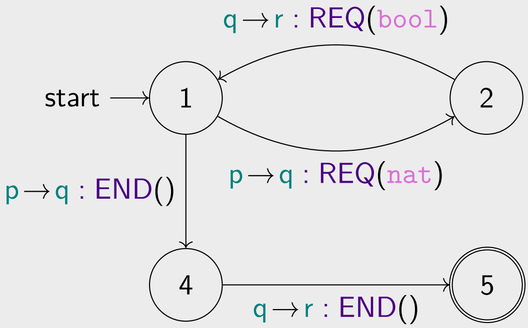
$$G = \mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{ll} \mathsf{REQ}(\texttt{nat}).\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\texttt{bool}).X \\ \mathsf{END}() \quad .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\texttt{done} \end{array} \right\}$$

We are going to typecheck a process implementing role r...

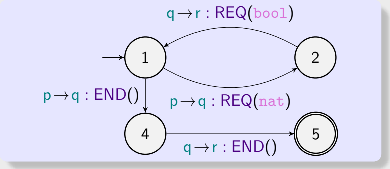
but first, let's get rid of the syntax for G!

Example: Semantic View of Global Types

$$\mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{ll} \mathsf{REQ}(\mathtt{nat}) & .\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\mathtt{bool}).X \\ \mathsf{END}() & .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\mathtt{done} \end{array} \right\}$$

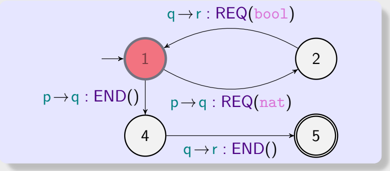


Example: Process & Typing



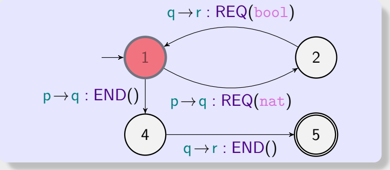
$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\, \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

Example: Process & Typing



$$P = \sum \left\{ \begin{array}{l} q?\mathrm{REQ}(x).\mathrm{print}(x).\ \mathtt{rec}\ X\ .\ \sum \left\{ \begin{array}{l} q?\mathrm{REQ}(x).\mathrm{process}(x).\ X \\ q?\mathrm{END}(\_).\mathtt{done} \end{array} \right\} \\ q?\mathrm{END}(\_).\mathtt{done} \end{array} \right\}$$

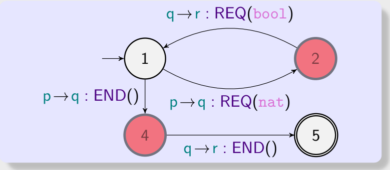
Example: Process & Typing



$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\ X\ .\ \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

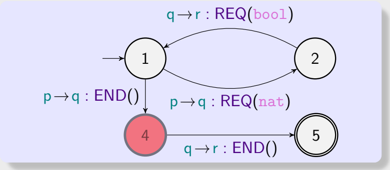
Our goal is to show that $\vdash P : 1 \upharpoonright \mathsf{r}$

Example: Process & Typing



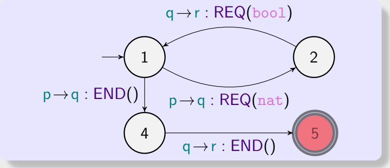
$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\ X\,.\sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

Example: Process & Typing



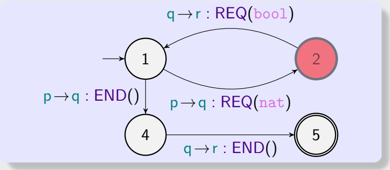
$$P = \sum \left\{ \begin{array}{l} q?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\ \sum \left\{ \begin{array}{l} q?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ q?\mathsf{END}(\_).\mathtt{done} \end{array} \right\} \\ \boxed{q?\mathsf{END}(\_).\mathtt{done}} \end{array} \right\}$$

Example: Process & Typing



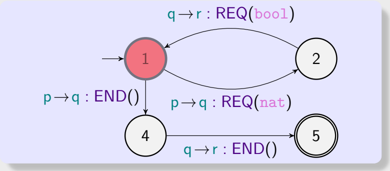
$$P = \sum \begin{Bmatrix} q?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\sum \begin{Bmatrix} q?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ q?\mathsf{END}(\_).\mathtt{done} \end{Bmatrix} \\ q?\mathsf{END}(\_).\ \boxed{\mathtt{done}} \end{Bmatrix}$$

Example: Process & Typing



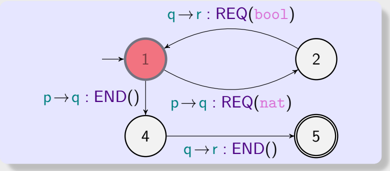
$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\ X\ .\ \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

Example: Process & Typing



$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \boxed{\mathtt{rec}\, X \,.\, \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}} \\ \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

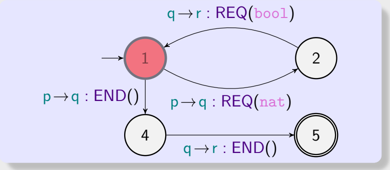
Example: Process & Typing



$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \boxed{\mathtt{rec}\, X\, .\, \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}} \\ \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

$$\cdot \vdash \mathtt{rec}\, X\, .\, \boxed{\ldots} : 1 \restriction \mathsf{r}$$

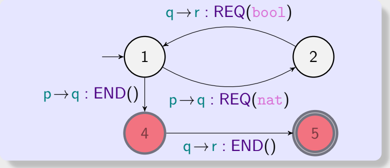
Example: Process & Typing



$$P = \sum \begin{cases} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\ \boxed{\sum \begin{cases} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{cases}} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{cases}$$

$$X : 1 \vdash \boxed{\ldots} : \boxed{\ldots} \upharpoonright \mathsf{r}$$

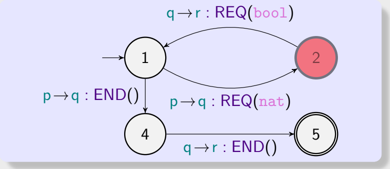
Example: Process & Typing



$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\, \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \boxed{\mathsf{q}?\mathsf{END}(\_).\mathtt{done}} \end{array} \right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{array} \right\}$$

$$X : 1 \vdash \boxed{\ldots} : \boxed{\ldots} \restriction \mathsf{r}$$

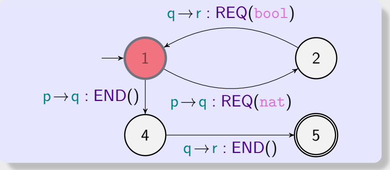
Example: Process & Typing



$$P = \sum \begin{cases} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\sum \begin{Bmatrix} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{Bmatrix} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{cases}$$

$$X : 1 \vdash \ldots : \ldots \upharpoonright \mathsf{r}$$

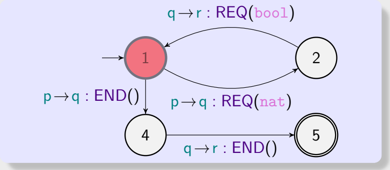
Example: Process & Typing



$$P = \sum \begin{cases} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\sum \begin{Bmatrix} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ \boxed{X} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{Bmatrix} \\ \mathsf{q}?\mathsf{END}(\_).\mathtt{done} \end{cases}$$

$$X : 1 \vdash \boxed{\ldots} : \boxed{\ldots} \upharpoonright \mathsf{r}$$

Example: Process & Typing



$$P = \sum \begin{cases} q?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathtt{rec}\, X\, .\, \sum \begin{Bmatrix} q?\mathsf{REQ}(x).\mathsf{process}(x).\ \boxed{\mathsf{X}} \\ q?\mathsf{END}(\_).\mathtt{done} \end{Bmatrix} \\ q?\mathsf{END}(\_).\mathtt{done} \end{cases}$$

$$X : 1 \vdash \boxed{\mathsf{X}} : \boxed{1} \upharpoonright r$$

Properties of Synthetic MPST

Some key lemmas:

- If $G \sim G'$ and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$ then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$
- If $G \xrightarrow{\alpha} G'$, with $\mathsf{r} \notin \alpha$, and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$, then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$

Properties of Synthetic MPST

Some key lemmas:

- If $G \sim G'$ and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$ then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$
- If $G \xrightarrow{\alpha} G'$, with $\mathsf{r} \notin \alpha$, and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$, then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$

These are needed for proving safety and liveness theorems (i.e. preservation andf progress). Suppose that $\mathcal{M}$ is a collection of processes, and G is well-behaved:

- If $\vdash \mathcal{M} : G$ and $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$, then there exists G' such that $G \xrightarrow{\alpha} G'$ and $\vdash \mathcal{M}' : G'$
- If $\vdash \mathcal{M} : G$ and G is not ended, then there exists $\mathcal{M}'$ and α such that $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$.

Properties of Synthetic MPST

Some key lemmas:

- If $G \sim G'$ and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$ then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$
- If $G \xrightarrow{\alpha} G'$, with $\mathsf{r} \notin \alpha$, and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$, then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$

These are needed for proving safety and liveness theorems (i.e. preservation andf progress).
Suppose that $\mathcal{M}$ is a collection of processes, and G is well-behaved:

- If $\vdash \mathcal{M} : G$ and $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$, then there exists G' such that $G \xrightarrow{\alpha} G'$ and $\vdash \mathcal{M}' : G'$
- If $\vdash \mathcal{M} : G$ and G is not ended, then there exists $\mathcal{M}'$ and α such that $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$.

Finally, we proved that for all global type G, the LTS of G is well-behaved.

Contributions

- Special case: MPST system with global types (without projection, merging, local types)
- Special case: Expressive enough to capture all benchmarks of (Scalas & Yoshida 2019)

- General case: MPST system with *well-behaved LTSs*
- Type soundness – much simpler than the literature (roughly 550 LOC of Agda!)

- Artifact: Full mechanisation in Agda.
- Artifact: Implementation of the special case in Rascal (Thanks, Sung!)

Contributions

THANKS!

- Special case: MPST system with global types (without projection, merging, local types)
- Special case: Expressive enough to capture all benchmarks of (Scalas & Yoshida 2019)

- General case: MPST system with *well-behaved LTSs*
- Type soundness – much simpler than the literature (roughly 550 LOC of Agda!)

- Artifact: Full mechanisation in Agda.
- Artifact: Implementation of the special case in Rascal (Thanks, Sung!)