Towards A Synthetic Formulation of Multiparty Session Types

David Castro-Perez, Francisco Ferreira

d.castro-perez@kent.ac.uk

10-12-2024

University of Kent

Background and Motivation

A Crash Course on Classic Multiparty Session Types

What is wrong with this code?

```go
func Worker(n int, resp chan int, err chan error) { ... }
func Master(reqCh chan int, respCh chan []int, cErrCh chan error) {
  for {
    ubound := <-reqCh
    workerChs := make([]chan int, ubound)
    errCh := make(chan error)
    for i := 0; i < ubound; i++ {
      workerChs[i] = make(chan int)
      go Worker(i+1, workerChs[i], errCh)
    }
    var res []int
    for i := 0; i < ubound; i++ {
      select {
      case sql := <-workerChs[i]:
        res = append(res, sql)
      case err := <-errCh:
        cErrCh <- err
        return
      }}
    respCh <- res}}
```

What is wrong with this code?

```
func Worker(n int, resp chan int, err chan error) { ... }
func Master(reqCh chan int, respCh chan []int, cErrCh chan error) {
  for {
    ubour
    work
    errCh
    for
      wo
      go
    }
    var
    for
      sel
      case sql := <-workerChs[i]:
        res = append(res, sql)
      case err := <-errCh:
        cErrCh <- err
        return
      }}
    respCh <- res}}
```

DEADLOCK!

ORPHAN MESSAGES!

NO RESOURCE CLEANUP!

...

What is wrong with this code?

```
func Worker(n int, resp chan int, err chan error) { ... }
func Master(reqCh chan int, respCh chan []int, cErrCh chan error) {
  for {
    ubound := <-reqCh
    worke
    errCh
    for i
      wor
      go
    }
    var res []int
    for i := 0; i < ubound; i++ {
      select {
      case sql := <-workerChs[i]:
        res = append(res, sql)
      case err := <-errCh:
        cErrCh <- err
        return
      }}
    respCh <- res}}
```

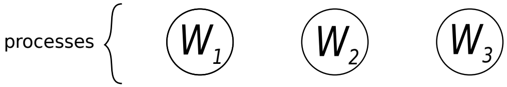
Master needs to guarantee that all Workers are notified when there is an error.

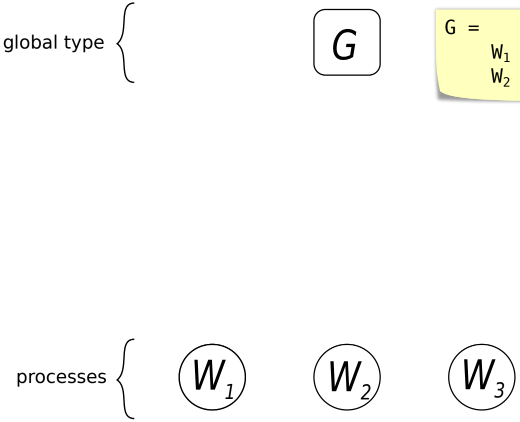
Key Idea

Multiparty Session Types prevent you from writing the code in the previous slide by enforcing syntactically that process implementations follow a given specification.

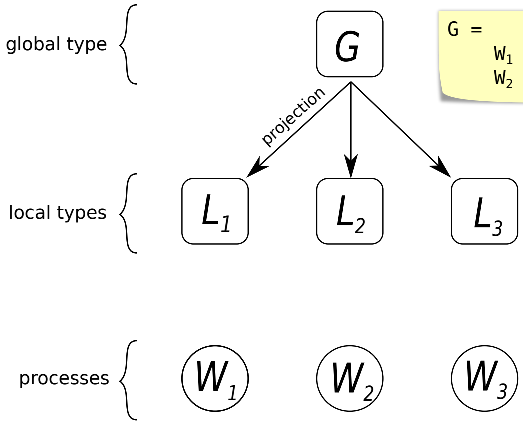
In a nutshell:

1. Global types: protocol specifications among a fixed number of different *roles*.
2. Role: sets of interactions that processes can do in a protocol.
3. Local types: protocol specifications *from the point of view of a single role*.
4. Projection: a *partial function* that extracts a *local type* given a *global type* and a *role*.
5. Well-formedness: guarantees **deadlock-freedom**, usually defined in terms of *projectability*.


processes
W_1
W_2
W_3

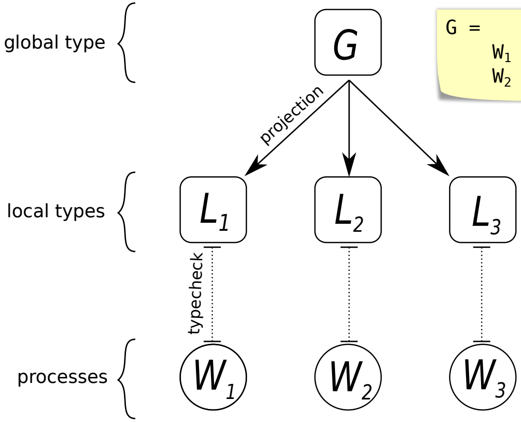


```
G =
    W₁ → W₂ : Int.
    W₂ → W₃ : Bool
```



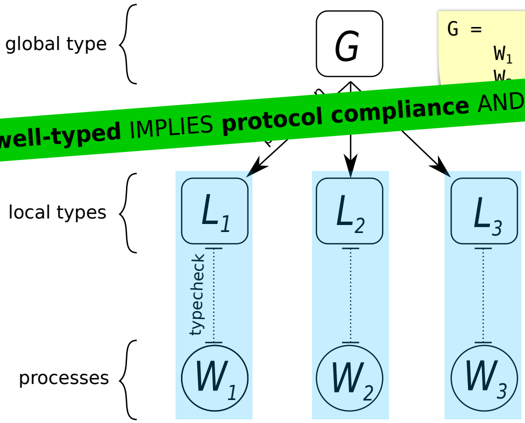
$$
\begin{aligned}
G = {} & \\
& W_1 \to W_2 : \mathtt{Int}. \\
& W_2 \to W_3 : \mathtt{Bool}
\end{aligned}
$$

$$
\begin{aligned}
L_1 &= W_2 \ ! \ \mathtt{Int} \\
L_2 &= W_1 \ ? \ \mathtt{Int}. \\
    & W_3 \ ! \ \mathtt{Bool} \\
L_3 &= W_2 \ ? \ \mathtt{Bool}
\end{aligned}
$$



```
G =
    W₁ → W₂ : Int.
    W₂ → W₃ : Bool
```

```
L₁ = W₂ ! Int
L₂ = W₁ ? Int.
     W₃ ! Bool
L₃ = W₂ ? Bool
```



```
G =
    W1 → W2 : Int.
    W2 → W
```

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

```
L1 = W2 ! Int
L2 = W1 ? Int.
     W3 ! Bool
L3 = W2 ? Bool
```

Global and Local Types

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

Projection

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

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

Projection

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

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

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

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

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

Projection

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

It gets complicated very quickly!

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

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

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

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

What is the point of $\sqcap$?

Consider the following protocol
– this is similar to the behaviour of the previous Go code snippet:

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

What is the point of $\sqcap$?

Consider the following protocol
– this is similar to the behaviour of the previous Go code snippet:

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

Projecting r

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

$$=$$

What is the point of $\sqcap$?

Consider the following protocol
– this is similar to the behaviour of the previous Go code snippet:

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

Projecting r

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

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

Processes and Typing

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

Process Typing (simplified)

Once we have local types, process typing is simple:

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

$$\frac{\Gamma, x_i : S_i \vdash P_i : L_i \qquad \forall i \in I}{\Gamma \vdash \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i : (\mathsf{p}?\{\ell_i(S_i).L_i\}_{i\in I})}\;\text{T-RECV}$$

Process Typing (simplified)

Once

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

T-SEND

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

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : L_i \qquad \forall i \in I}{\Gamma \vdash \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i : (\mathsf{p}?\{\ell_i(S_i).L_i\}_{i\in I})}$$

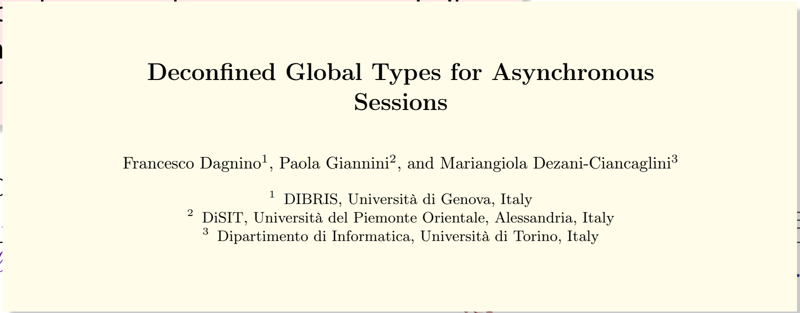
Process Typing (simplified)

Lo
th
pr

Once

T-SEND
$\Gamma \vdash P :$ … I

$\Gamma \vdash \mathsf{q} \, ! \, \ell$ … $.L_i\}_{i \in I})$


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 Classic 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
     - Complex graph-based representation of MPST: 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** about the uses of coinduction

Example of Imprecision in Classic MPST

$\boxed{\text{“We identify } \mu X.G \text{ with } [\mu X.G/X]G\text{”}}$

This is a common statement in proofs about MPST, which clearly specifies an equirecursive formulation, but...

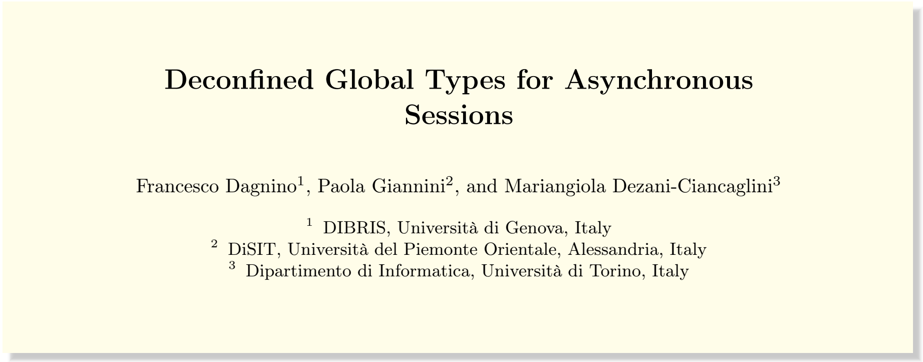
1. The rules still refer to open global types with variables X
2. The rules specify when and how to unfold $\mu X.G$ – if we are using equirecursion, $\mu.$ should not be in the syntax ouf our language!

Moreover, this “identification” of a global type and its unfolding is not powerful enough. E.g.

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

This forces the use of tedious syntactic proofs about how the swapping of unrelated actions does not affect the protocol.

A Few Attempts at Simplifying the Theory

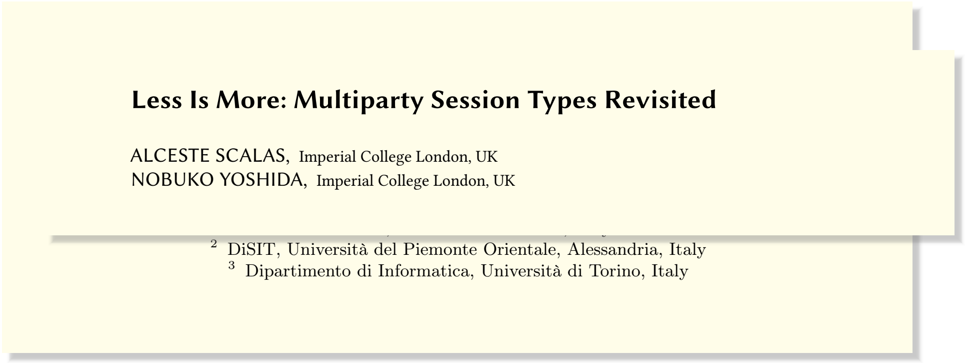


Deconfined Global Types for Asynchronous Sessions

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

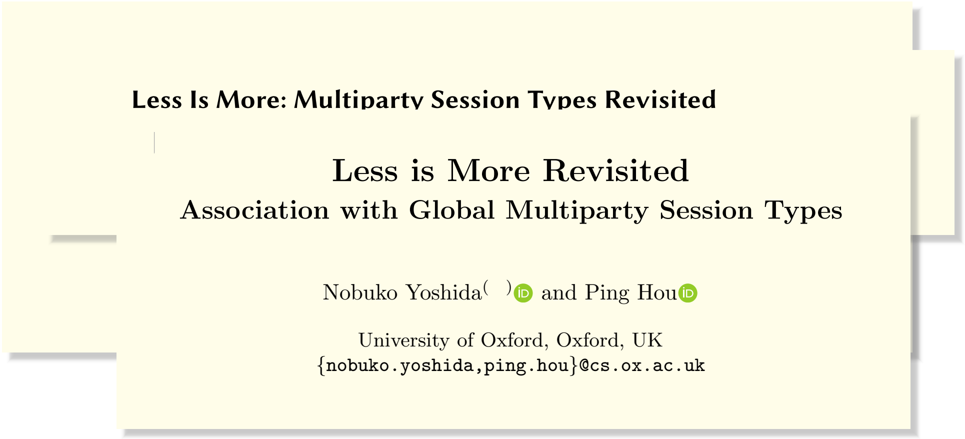


Less Is More: Multiparty Session Types Revisited

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

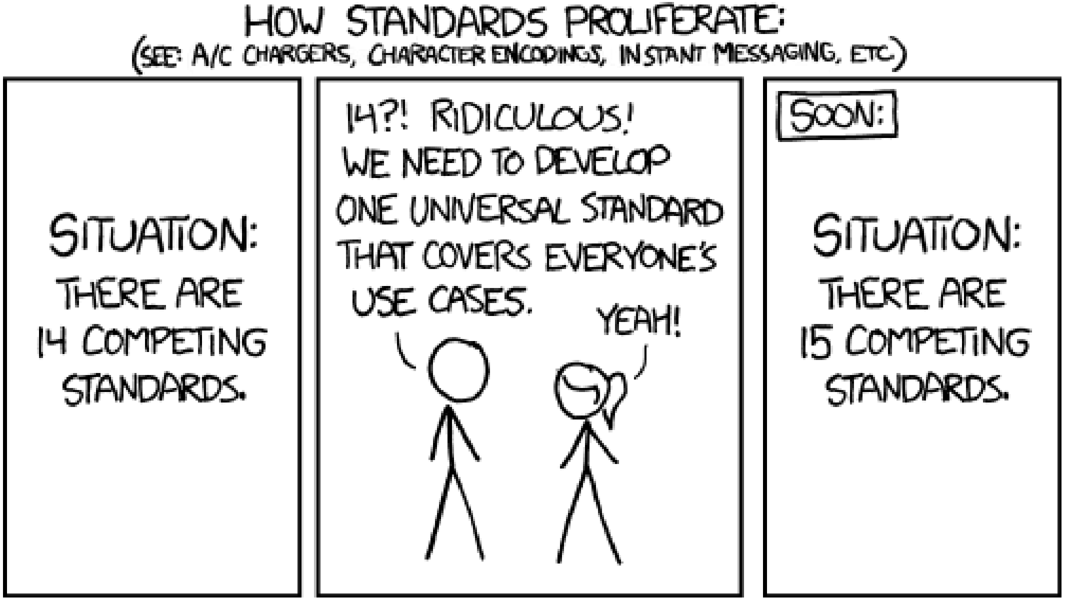
A Few Attempts at Simplifying the Theory


Less Is More: Multiparty Session Types Revisited

Less is More Revisited
Association with Global Multiparty Session Types

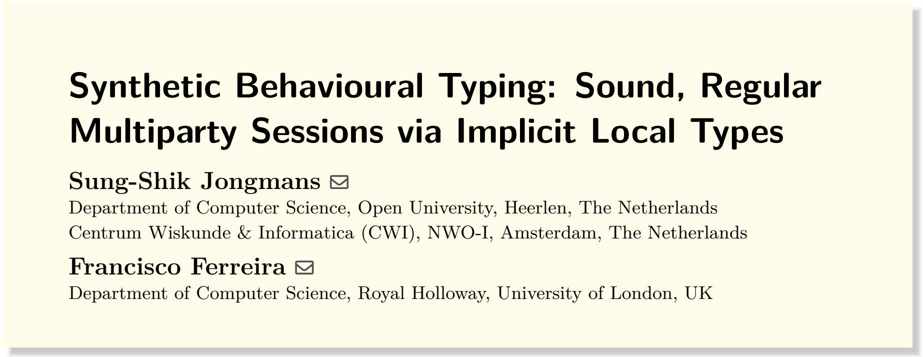
Nobuko Yoshida( ) and Ping Hou

University of Oxford, Oxford, UK
{nobuko.yoshida,ping.hou}@cs.ox.ac.uk


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

https://xkcd.com/927/

Our Approach: Synthetic Typing

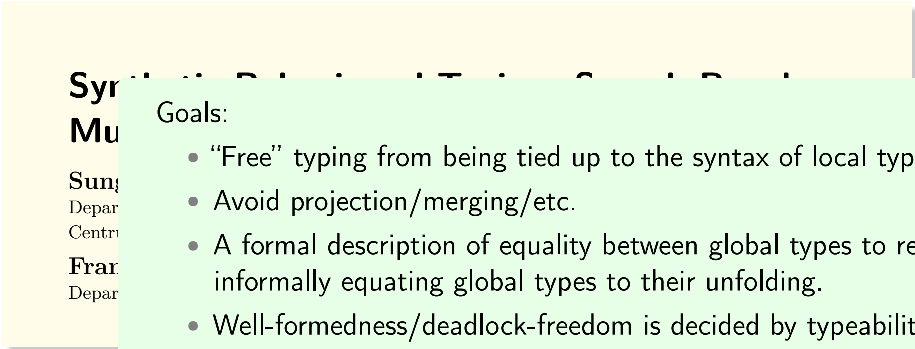


Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types

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



Syr
Mu

Sun
Depar
Centr

Fran
Depar

Goals:

- "Free" typing from being tied up to the syntax of local types.
- Avoid projection/merging/etc.
- A formal description of equality between global types to replace informally equating global types to their unfolding.
- Well-formedness/deadlock-freedom is decided by typeability.
- Mechanisation in Agda.

Towards Synthetic MPST (WIP)

New (Synthetic) Core Typing Rules

$$\boxed{\textbf{New judgement}: \Gamma \vdash P : G \upharpoonright \mathsf{p}}$$

$$\textrm{T-SKIP}\;\frac{\Gamma \vdash P : G' \upharpoonright \mathsf{r} \qquad \forall\, G \setminus \alpha = G' \text{ s.t. } \mathsf{r} \notin \mathsf{parts}(\alpha)}{\Gamma \vdash P : G \upharpoonright \mathsf{r}}$$

Synthetic, in that G' occurs only in the premise, not in the conclusion. G' needs to be *synthesised* by using the rules of the operational semantics of global types (Jongmans and Ferreira, 2023).

New (Synthetic) Core Typing Rules

What is wrong with these rules?

$$\frac{\Gamma \vdash P : G' \upharpoonright \mathsf{r} \qquad \forall\, G \setminus \alpha = G' \text{ s.t. } \mathsf{r} \notin \mathsf{parts}(\alpha)}{\Gamma \vdash P : G \upharpoonright \mathsf{r}}\ \text{T-SKIP}$$

New (Synthetic) Core Typing Rules

Hint: the problem is in these rules

T-SKIP

$$\frac{\Gamma \vdash P : G' \restriction \mathsf{r} \qquad \forall\, G \setminus \alpha = G' \text{ s.t. } \mathsf{r} \notin \mathsf{parts}(\alpha)}{\Gamma \vdash P : G \restriction \mathsf{r}}$$

New

Hint 2: the problem is the same in both rules, let's focus on this one

New (

What happens if G does not allow p to receive from q?

New (

This was a "rookie" mistake ... We cannot allow rules to be vacuously true!

(Hopefully) Fixed Typing Rules

Let $\mathcal{R}(\alpha, G) = \exists G',\ G \setminus \alpha = G'$
– this means that an interaction α is “ready” (i.e. can happen) in G.
Let $\mathcal{W}(\mathsf{r}, G) = \exists \alpha,\ \mathcal{R}(\alpha, G) \land \mathsf{r} \notin \mathsf{parts}(\alpha)$
– this means that r can “wait” for another (possibly unrelated) interaction in G.

(Hopefully) Fixed Typing Rules

Let $\mathcal{R}(\alpha, G) = \exists G',\ G \setminus \alpha = G'$
– this means that an interaction α is "ready" (i.e. can happen) in G.
Let $\mathcal{W}(\mathsf{r}, G) = \exists \alpha,\ \mathcal{R}(\alpha, G) \wedge \mathsf{r} \notin \mathsf{parts}(\alpha)$
– this means that r can "wait" for another (possibly unrelated) interaction in G.

(Hopefully) Fixed Typing Rules

Let $\mathcal{R}(\alpha, G) = \exists G',\ G \setminus \alpha = G'$
– this means that an interaction α is "ready" (i.e. can happen) in G.
Let $\mathcal{W}(\mathsf{r}, G) = \exists \alpha,\ \mathcal{R}(\alpha, G) \wedge \mathsf{r} \notin \mathsf{parts}(\alpha)$
– this means that r can "wait" for another (possibly unrelated) interaction in G.

$$\frac{\mathcal{W}(\mathsf{r}, G) \qquad \Gamma \vdash P : G' \restriction \mathsf{r} \qquad \forall\ G \setminus \alpha = G' \text{ s.t.} \mathsf{r} \notin \mathsf{parts}(\alpha)}{\Gamma \vdash P : G \restriction \mathsf{r}}\ \text{T-SKIP}$$

(Hopefully) Fixed Typing Rules

Let $\mathcal{R}(\alpha, G) = \exists G',\ G \setminus \alpha = G'$
– this means that an interaction α is "ready" (i.e. can happen) in G.
Let $\mathcal{W}(\mathsf{r}, G) = \exists \alpha,\ \mathcal{R}(\alpha, G) \wedge \mathsf{r} \notin \mathsf{parts}(\alpha)$
– this means that r can "wait" for another (possibly unrelated) interaction in G.

$$\frac{\boxed{\mathcal{W}(\mathsf{r}, G)} \qquad \Gamma \vdash P : G' \upharpoonright \mathsf{r} \qquad \forall\ G \setminus \alpha = G' \text{ s.t.} \mathsf{r} \notin \mathsf{parts}(\alpha)}{\Gamma \vdash P : G \upharpoonright \mathsf{r}}\ \text{T-SKIP}$$

(Hopefully) Fixed Typing Rules

- The rules look more complex than with a syntactic approach, but computing $G \setminus \mathsf{q} \xrightarrow{\ell_i(S_i)} \mathsf{p} = G'$ is entirely mechanical by using the semantics of global types.
- The proof of subject reduction is greatly simplified with this formulation.
- There is no need of projection/merging.

Let $\mathcal{R}(\alpha, G$
– this mea
Let $\mathcal{W}(\mathsf{r}, G$
– this mea

$$\text{T-SKIP}\ \dfrac{\boxed{\mathcal{W}(\mathsf{r}, G)} \qquad \Gamma \vdash P : G' \upharpoonright \mathsf{r} \qquad \forall\, G \setminus \alpha = G' \text{ s.t.} \mathsf{r} \notin \mathsf{parts}(\alpha)}{\Gamma \vdash P : G \upharpoonright \mathsf{r}}$$

Semantics

The semantics of global types is defined in a standard way.

Although the current semantics is synchronous, this does not prevent us from defining an asynchronous semantics for processes.

It deals with recursion: in our typing rules we do not need to deal with recursion variables or global type unfolding – a true equirecursive formulation in our type system.

$$\frac{j \in I}{\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \setminus \mathsf{p} \xrightarrow{\ell_j(S_j)} \mathsf{q} = G_j} \qquad \frac{[\mu X.G/X]G \setminus \alpha = G'}{\mu X.G \setminus \alpha = G'}$$

$$\frac{\forall (i \in I), G_i \setminus \alpha = G_i' \qquad \mathsf{parts}(\alpha) \cap \{\mathsf{p}, \mathsf{q}\} = \varnothing}{\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \setminus \alpha = \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i'\}_{i \in I}}$$

Global Type Bisimilarity

We use a coinductive definition of **strong bisimilarity**:

$G_1 \sim G_2$ iff:

- $\forall \alpha,\ G_1 \setminus \alpha = G_1' \Rightarrow \exists G_2',\ G_2 \setminus \alpha = G_2' \wedge G_1' \sim G_2'$
- $\forall \alpha,\ G_2 \setminus \alpha = G_2' \Rightarrow \exists G_1',\ G_1 \setminus \alpha = G_1' \wedge G_1' \sim G_2'$

It is straightforward that $[\mu X.G/X]G \sim \mu X.G$

Global Type Bisimilarity

We u

$G_1 \sim$

- \

- \

We **never** use syntactic equality,
in our type system, only $G \sim G'$

It is straightforward that $[\mu X.G/X]G \sim \mu X.G$

Example

Consider again:

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

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

Example

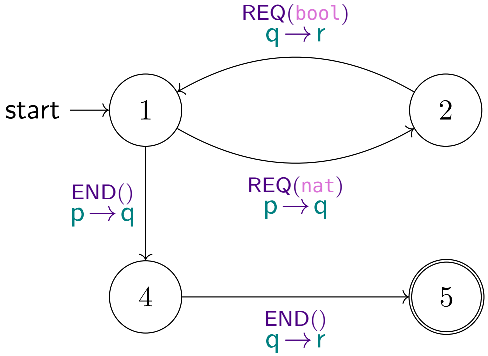
Consider again:

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

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

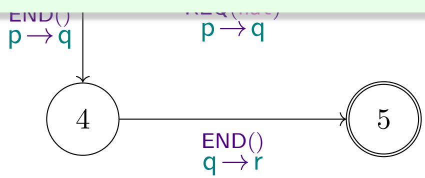
$$\mu X.\mathsf{p} \to \mathsf{q} : \left\{ \begin{array}{ll} \mathsf{REQ}(\mathtt{nat}).\mathsf{q} \to \mathsf{r} : \mathsf{REQ}(\mathtt{bool}).X \\ \mathsf{END}() \quad .\mathsf{q} \to \mathsf{r} : \mathsf{END}().\mathsf{done} \end{array} \right\}$$



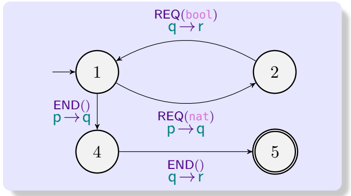
Example: Semantic View of Global Types

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

(Small parenthesis, and shameless advertising: I am working with Jonah – and hopefully joining efforts with Francisco, Marco Carbone, Alceste Scalas, any of you that is interested ... – on automating the mechanisation of this semantic view of LTS in Coq/OCaml.)

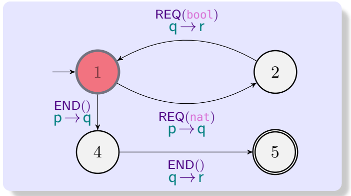


Example: Process & Typing



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

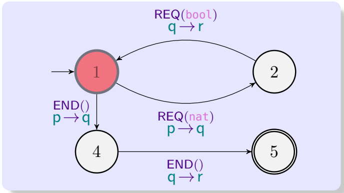
Example: Process & Typing



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

Goal: $\boxed{\cdot \vdash P : 1 \restriction \mathsf{r}}$ – for simplicity, this example uses LTS state numbers as global types.

Example: Process & Typing

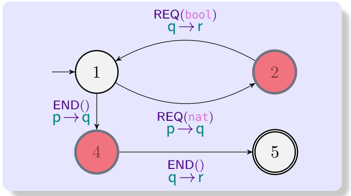

$$\begin{tikzcd}
\arrow[r] & 1 \arrow[rr, bend right, "\substack{\mathsf{REQ}(\mathtt{nat})\\ \mathsf{p}\to\mathsf{q}}"'] \arrow[d, "\substack{\mathsf{END}()\\ \mathsf{p}\to\mathsf{q}}"'] & & 2 \arrow[ll, bend right, "\substack{\mathsf{REQ}(\mathtt{bool})\\ \mathsf{q}\to\mathsf{r}}"'] \\
& 4 \arrow[rr, "\substack{\mathsf{END}()\\ \mathsf{q}\to\mathsf{r}}"'] & & 5
\end{tikzcd}$$

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

Goal: $\boxed{\cdot \vdash P : 1 \upharpoonright \mathsf{r}}$ – for simplicity, this example uses LTS state numbers as global types.

–We have a $\sum$, so we can only apply either T-RECV or T-SKIP. At 1, r cannot receive from q, so we must use T-SKIP.

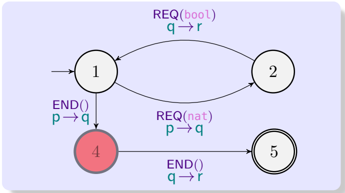
Example: Process & Typing



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

Two cases: $1 \to 2$, and $1 \to 4$

Example: Process & Typing



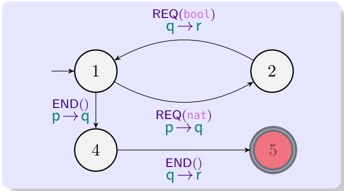
$$P = \sum \left\{ \begin{array}{l} \mathsf{q?REQ}(x).\mathsf{print}(x).\ \mathsf{rec}\, X\, .\sum \left\{ \begin{array}{l} \mathsf{q?REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q?END}(\_).\mathsf{done} \end{array} \right\} \\ \mathsf{q?END}(\_).\mathsf{done} \end{array} \right\}$$

Two cases: $1 \to 2$, and $1 \to 4$

– We have that $4 \setminus \overset{\mathsf{END}()}{\mathsf{q} \to \mathsf{r}} = 5$

– At 5, r can no longer take any action in G, so done is well typed.

Example: Process & Typing



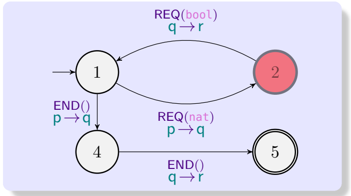
$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\ \mathsf{rec}\,X\,.\sum\left\{\begin{array}{l}\mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X\\ \mathsf{q}?\mathsf{END}(\_).\mathsf{done}\end{array}\right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathsf{done} \end{array} \right\}$$

Two cases: $1 \to 2$, and $1 \to 4$

– We have that $4 \setminus \overset{\mathsf{END}()}{\mathsf{q} \to \mathsf{r}} = 5$

– At 5, r can no longer take any action in G, so done is well typed.

Example: Process & Typing



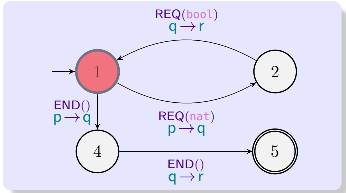
$$P = \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{print}(x).\ \text{rec}\ X\ .\sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ X \\ \text{q?END}(\_).\text{done} \end{array} \right\} \\ \text{q?END}(\_).\text{done} \end{array} \right\}$$

Two cases: $1 \to 2$, and $1 \to 4$

– We have that $2 \setminus \overset{\text{REQ(bool)}}{\text{q} \to \text{r}} = 1$

– We transition back to 1.

Example: Process & Typing



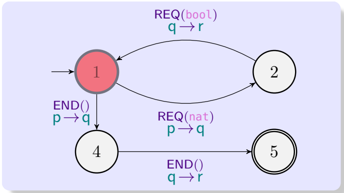
$$P = \sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{print}(x).\,\mathsf{rec}\ X\,.\sum \left\{ \begin{array}{l} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathsf{done} \end{array} \right\} \\ \mathsf{q}?\mathsf{END}(\_).\mathsf{done} \end{array}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

Two cases: $1 \to 2$, and $1 \to 4$

– We have that $2 \setminus \overset{\mathsf{REQ}(\mathtt{bool})}{\mathsf{q} \rightarrow \mathsf{r}} = 1$

– We transition back to 1.

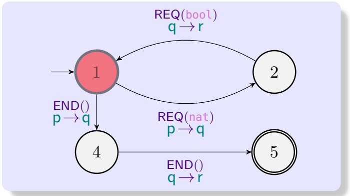
Example: Process & Typing



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

With a $\mathtt{rec}\,X\,.$, we need to remember the state of the protocol, $\boxed{1}$. Whenever we jump back to X, we will check that we are again in a bisimilar state.

Example: Process & Typing

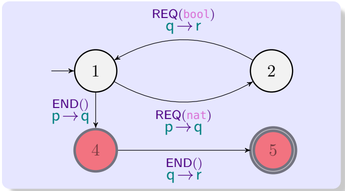


$$P = \sum \begin{cases} \mathsf{q?REQ}(x).\mathsf{print}(x).\ \mathsf{rec}\ X\ .\ \sum \begin{Bmatrix} \mathsf{q?REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q?END}(\_).\mathsf{done} \end{Bmatrix} \\ \mathsf{q?END}(\_).\mathsf{done} \end{cases}$$

With a $\mathsf{rec}\ X\ .$, we need to remember the state of the protocol, $\boxed{1}$.

– We use again T-SKIP and T-RECV.

Example: Process & Typing

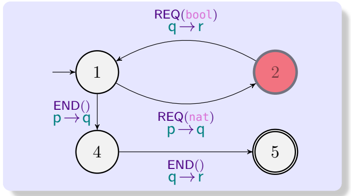


$$P = \sum \begin{cases} \mathsf{q?REQ}(x).\mathsf{print}(x).\ \mathsf{rec}\, X\,.\sum \begin{cases} \mathsf{q?REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q?END}(\_).\mathsf{done} \end{cases} \\ \mathsf{q?END}(\_).\mathsf{done} \end{cases}$$

With a $\mathsf{rec}\, X\,.$, we need to remember the state of the protocol, $\boxed{1}$.

– We use again T-SKIP and T-RECV.

Example: Process & Typing

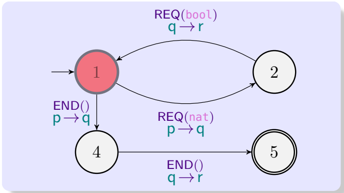


$$P = \sum \begin{cases} \mathsf{q?REQ}(x).\mathsf{print}(x).\ \mathsf{rec}\, X\,.\sum \begin{Bmatrix} \mathsf{q?REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q?END}(\_).\mathsf{done} \end{Bmatrix} \\ \mathsf{q?END}(\_).\mathsf{done} \end{cases}$$

With a $\mathsf{rec}\, X\,.$, we need to remember the state of the protocol, $\boxed{1}$.

– We use again T-SKIP and T-RECV.

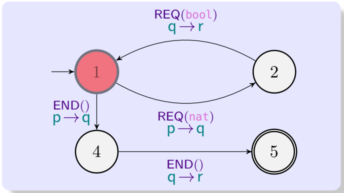
Example: Process & Typing



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

With a $\mathsf{rec}\, X\,.$, we need to remember the state of the protocol, $\boxed{1}$.
– We use again T-SKIP and T-RECV.

Example: Process & Typing

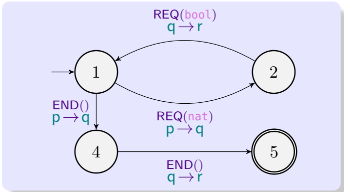


$$P = \sum \begin{cases} \text{q?REQ}(x).\text{print}(x).\ \text{rec}\, X\, .\sum \begin{cases} \text{q?REQ}(x).\text{process}(x).X \\ \text{q?END}(\_).\text{done} \end{cases} \\ \text{q?END}(\_).\text{done} \end{cases}$$

With a $\text{rec}\, X\, .$, we need to remember the state of the protocol, $\boxed{1}$.

– We landed in the same state where we used recursion.

Example: Process & Typing



$$\begin{tikzcd} \arrow[r] & 1 \arrow[rr, bend right, "\substack{\mathsf{REQ}(\mathtt{nat})\\ \mathsf{p}\to\mathsf{q}}"'] \arrow[d, "\substack{\mathsf{END}()\\ \mathsf{p}\to\mathsf{q}}"'] & & 2 \arrow[ll, bend right, "\substack{\mathsf{REQ}(\mathtt{bool})\\ \mathsf{q}\to\mathsf{r}}"'] \\ & 4 \arrow[rr, "\substack{\mathsf{END}()\\ \mathsf{q}\to\mathsf{r}}"'] & & 5 \end{tikzcd}$$

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

We finished building our type derivation: the process is well typed

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 $G \setminus \alpha = G'$, 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 progress and preservation.
If $\mathcal{M}$ is a collection of processes that implement all of the roles in G:

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

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 $G \setminus \alpha = G'$, with $\mathsf{r} \notin \alpha$, and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$, then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$

These are
If $\mathcal{M}$ is a

- If $\vdash$
  $G \setminus$

I am going to be annoying again ...

One of the above lemmas is wrong! It should be obvious which one ... But why?

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

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 $G \setminus \alpha = G'$, with $\mathsf{r} \notin \alpha$, and $\Gamma \vdash P : G \upharpoonright \mathsf{r}$, then $\Gamma \vdash P : G' \upharpoonright \mathsf{r}$

These are
If $\mathcal{M}$ is a

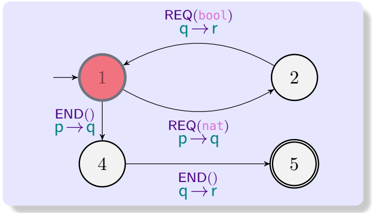
- If $\vdash$
  $G \setminus$

I am going to be annoying again ...

One of the above lemmas is wrong! It should be obvious which one ... But why?

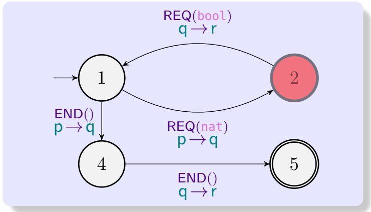
- If $\vdash \mathcal{M} : G$ and G is not ended, then there exists $\mathcal{M}'$ such that $\mathcal{M} \longrightarrow \mathcal{M}'$.

Why is the Previous Lemma False?



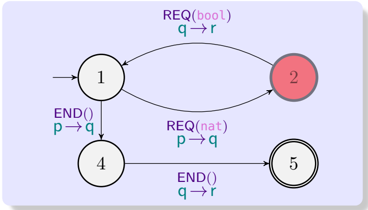
$$\mathsf{rec}\ X\,.\sum\begin{Bmatrix}\mathsf{q?REQ}(x).\mathsf{process}(x).\ X\\ \mathsf{q?END}(\_).\mathsf{done}\end{Bmatrix}$$

Why is the Previous Lemma False?



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

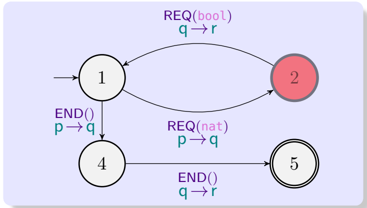
Why is the Previous Lemma False?



$$\text{rec}\ X\ .\ \sum \begin{Bmatrix} \text{q?REQ}(x).\text{process}(x).\ X \\ \text{q?END}(\_).\text{done} \end{Bmatrix}$$

Recursion state X: 2

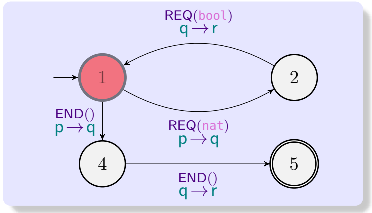
Why is the Previous Lemma False?



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

Recursion state X: 2

Why is the Previous Lemma False?

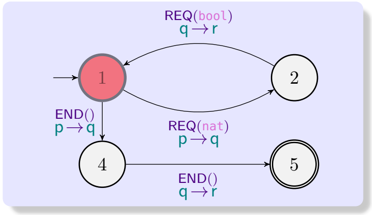


$$\mathsf{rec}\ X\,.\sum\begin{cases}\mathsf{q?REQ}(x).\mathsf{process}(x).\boxed{X}\\ \mathsf{q?END}(\_).\mathsf{done}\end{cases}$$

Recursion state X: 2

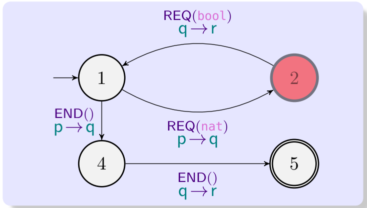
We should be at 2, but we are at 1!

A Potential Solution: Unfolding the Process



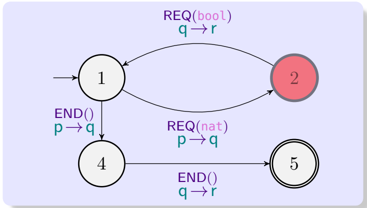
$$\sum \left\{ \begin{array}{l} \mathsf{q?REQ}(x).\mathsf{process}(x).\ \mathsf{rec}\ X\,.\sum \left\{ \begin{array}{l} \mathsf{q?REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q?END}(\_).\mathsf{done} \end{array} \right\} \\ \mathsf{q?END}(\_).\mathsf{done} \end{array} \right\}$$

A Potential Solution: Unfolding the Process



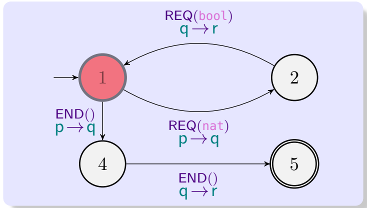
$$\sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ \text{rec}\ X\ .\ \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ X \\ \text{q?END}(\_).\text{done} \end{array} \right\} \\ \text{q?END}(\_).\text{done} \end{array} \right\}$$

A Potential Solution: Unfolding the Process



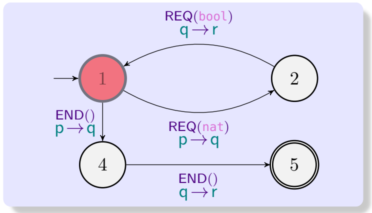
$$\sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ \text{rec}\ X\ .\ \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ X \\ \text{q?END}(\_).\text{done} \end{array} \right\} \\ \text{q?END}(\_).\text{done} \end{array} \right\}$$

A Potential Solution: Unfolding the Process



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

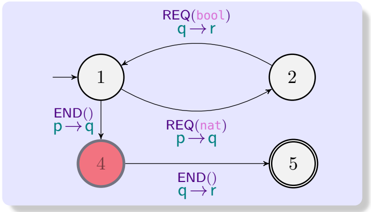
A Potential Solution: Unfolding the Process



$$\sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ \text{rec}\ X\ .\boxed{\ }\sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x).\ X \\ \text{q?END}(\_).\text{done} \end{array} \right\} \\ \text{q?END}(\_).\text{done} \end{array} \right\}$$

Recursion state X: 1

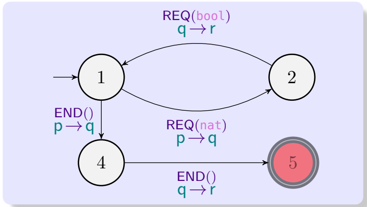
A Potential Solution: Unfolding the Process



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

Recursion state X: 1

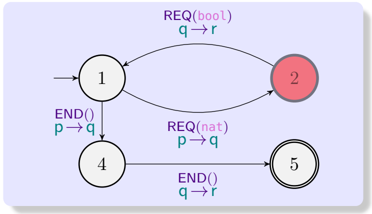
A Potential Solution: Unfolding the Process



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

Recursion state X: 1

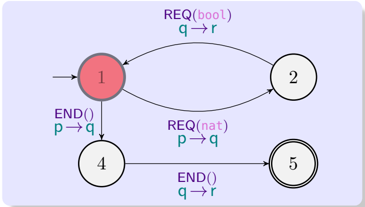
A Potential Solution: Unfolding the Process



$$\sum \begin{cases} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ \mathsf{rec}\ X\ .\sum \begin{cases} \mathsf{q}?\mathsf{REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q}?\mathsf{END}(\_).\mathsf{done} \end{cases} \\ \mathsf{q}?\mathsf{END}(\_).\mathsf{done} \end{cases}$$

Recursion state X: 1

A Potential Solution: Unfolding the Process



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

Recursion state X: 1

Wrap Up

Benefits of Synthetic Typing

1. Decoupling behavioural typing from the syntactic objects that describe the protocols.
2. No need for complex projections, merging, . . .
3. As long as the protocol specifications satisfy certain required properties, they can be extended without affecting the typing, or the progress and preservation of the type system.
4. (Hopefully) easier integration in a mainstream programming language: we would need to walk throught the AST, and step through the semantics of the protocol as needed.

TODO

We reached (somewhat) stable definitions in our Agda mechanisation, but we need to fix (or reformulate) the following (incorrect) property:
If $G \setminus \alpha = G'$, with $\mathsf{r} \notin \alpha$, and $\Gamma \vdash P : G \restriction \mathsf{r}$, then $\Gamma \vdash P : G' \restriction \mathsf{r}$

We still need to show that type inhabitation subsumes common well-formedness criteria for global types.