

Towards A Synthetic Formulation of Multiparty Session Types

David Castro-Perez, Francisco Ferreira

d.castro-perez@kent.ac.uk

10-12-2024



Background and Motivation

A Crash Course on Classic Multiparty Session Types

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
        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 {
        about
        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 *local type* given a *global types* and a *role*.
5. Well-formedness: guarantees **deadlock-freedom**, usually defined in terms of *projectability*.

processes { W_1 W_2 W_3 }

global type {

G

$G =$

$W_1 \rightarrow W_2 : \text{Int.}$

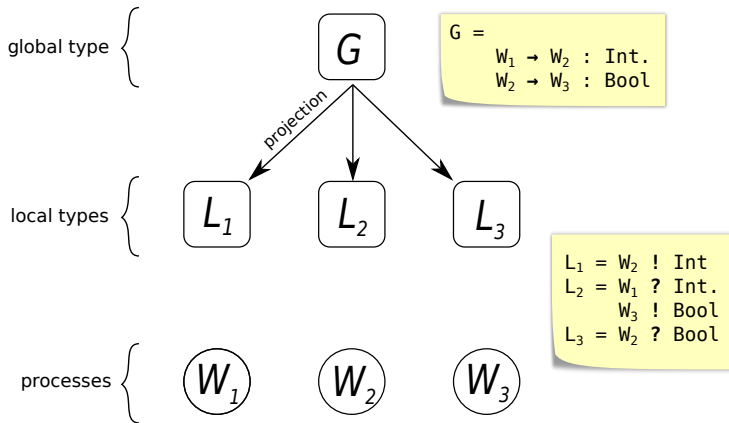
$W_2 \rightarrow W_3 : \text{Bool}$

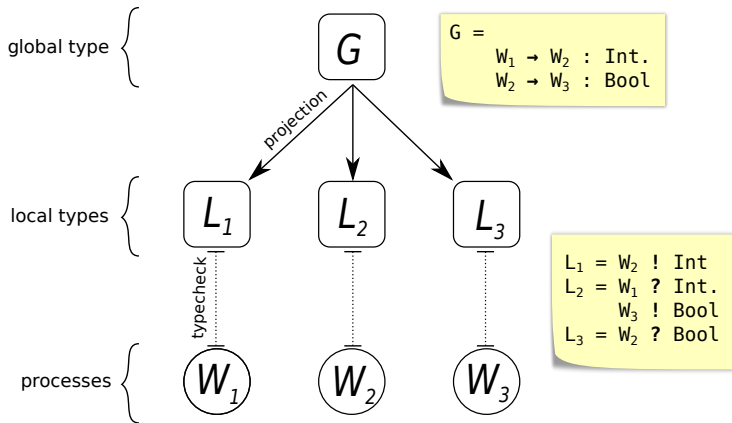
processes {

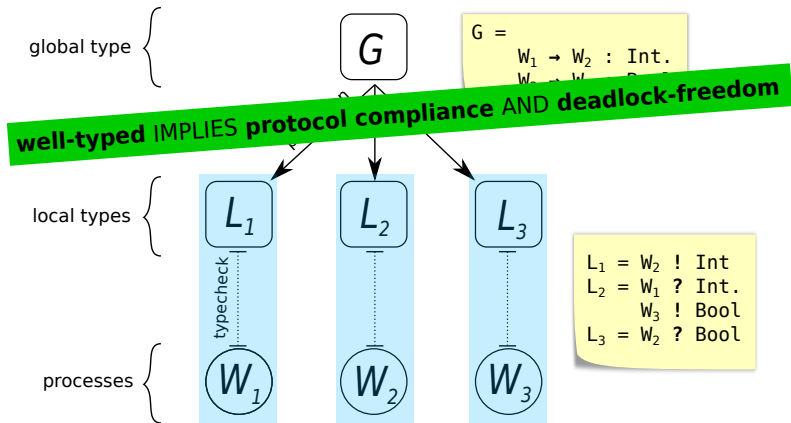
W_1

W_2

W_3







Global and Local Types

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

Projection

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

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

Projection

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

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

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

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

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

Projection

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

It gets complicated very quickly!

$$\mu \Delta.G \upharpoonright r = \begin{cases} \emptyset & (r \notin G) \end{cases} \quad \Delta \upharpoonright r = \Delta \quad \emptyset \upharpoonright r = \emptyset$$

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

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

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

What is the point of \sqcap ?

Consider the following protocol

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

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

What is the point of \sqcap ?

Consider the following protocol

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

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

Projecting r

$$\mu X. (q? \text{REQ}(\text{bool}).X) \sqcap (q? \text{END}().\emptyset)$$

=

What is the point of \sqcap ?

Consider the following protocol

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

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

Projecting r

$$\begin{aligned} & \mu X. (q? \text{REQ}(\text{bool}).X) \sqcap (q? \text{END}().\emptyset) \\ &= \mu X. q? \left\{ \begin{array}{l} \text{REQ}(\text{bool}).X \\ \text{END}() \quad .\text{done} \end{array} \right\} \end{aligned}$$

Processes and Typing

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

Process Typing (simplified)

Once we have local types, process typing is simple:

T-SEND

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

T-RECV

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



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: Tireore et al. (2023)
- Hard to extend

3. Imprecise about the uses of coinduction

Example of Imprecision in Classic MPST

“We identify $\mu X.G$ with $[\mu X.G/X]G$ ”

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, μ should not be in the syntax of our language!

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

$$p \rightarrow q : p' \rightarrow q' : G \neq p' \rightarrow q' : p \rightarrow q : G$$

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

Less Is More: Multiparty Session Types Revisited

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

A Few Attempts at Simplifying the Theory

Le

ALC
NO

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.

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 Typing: A New Paradigm in Type Theory

Mu

Sun

Depar

Centr

Frar

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

New judgement : $\Gamma \vdash P : G \upharpoonright p$

T-SEND

$$\frac{\Gamma \vdash P : G' \upharpoonright p \quad G \setminus \overset{\ell(S)}{p \rightarrow q} = G' \quad \Gamma \vdash e : S}{\Gamma \vdash q ! \ell(e).P : G \upharpoonright p}$$

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}{\Gamma \vdash \sum_{i \in I} q ? \ell_i(x_i).P_i : G \upharpoonright p}$$

T-SKIP

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

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

What is wrong with these rules?

T-SEND

$$\frac{\Gamma \vdash P : G' \upharpoonright p \quad G \setminus \overset{\ell(S)}{p \rightarrow q} = G' \quad \Gamma \vdash e : S}{\Gamma \vdash q ! \ell(e).P : G \upharpoonright p}$$

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}{\Gamma \vdash \sum_{i \in I} q ? \ell_i(x_i).P_i : G \upharpoonright p}$$

T-SKIP

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

New

Hint: the problem is in these rules

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}{\Gamma \vdash \sum_{i \in I} q? \ell_i(x_i). P_i : G \upharpoonright p}$$

T-SKIP

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

New

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

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}{\Gamma \vdash \sum_{i \in I} q ? \ell_i(x_i). P_i : G \upharpoonright p}$$

New (Synthetic) Core Typing Rules

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

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \boxed{\forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}}{\Gamma \vdash \sum_{i \in I} q ? \ell_i(x_i). P_i : G \upharpoonright p}$$

New (Synthetic) Core Typing Rules

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

T-RECV

$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \boxed{\forall G \setminus \overset{\ell_i(S_i)}{q} \rightarrow p = G'}}{\Gamma \vdash \sum_{i \in I} q? \ell_i(x_i). P_i : G \upharpoonright p}$$

↑ (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 $\mathcal{W}(r, G) = \exists \alpha, G \setminus \alpha = G' \wedge r \notin \text{ports}(\alpha)$

– this means that an interaction r can “wait” for another (possibly unrelated) interaction in G .

T-SEND

$$\frac{\Gamma \vdash P : G' \upharpoonright p \quad G \setminus \overset{\ell(S)}{p \rightarrow q} = G' \quad \Gamma \vdash e : S}{\Gamma \vdash q ! \ell\langle e \rangle . P : G \upharpoonright p}$$

↑ (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 $\mathcal{W}(r, G) = \exists \alpha, G \setminus \alpha = G' \wedge r \notin \text{ports}(\alpha)$

– this means that an interaction r can “wait” for another (possibly unrelated) interaction in G .

T-RECV

$$\frac{\exists(j \in I), \mathcal{R}(\overset{\ell_j(S_j)}{q \rightarrow p}, G) \quad \Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}{\Gamma \vdash \sum_{i \in I} q ? \ell_i(x_i). P_i : G \upharpoonright p}$$

↑ (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 $\mathcal{W}(r, G) = \exists \alpha, G \setminus \alpha = G' \wedge r \notin \text{ports}(\alpha)$

– this means that an interaction r can “wait” for another (possibly unrelated) interaction in G .

T-SKIP

$$\frac{\mathcal{W}(r, G) \quad \Gamma \vdash P : G' \upharpoonright r \quad \forall G \setminus \alpha = G' \text{ s.t. } r \notin \text{ports}(\alpha)}{\Gamma \vdash P : G \upharpoonright r}$$

(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 $\mathcal{W}(r, G) = \exists \alpha, G \setminus \alpha = G' \wedge r \notin \text{ports}(\alpha)$

– this means that an interaction r can “wait” for another (possibly unrelated) interaction in G .

T-SEND

$$\frac{\Gamma \vdash P : G' \upharpoonright p \quad G \setminus \overset{\ell(S)}{p \rightarrow q} = G' \quad \Gamma \vdash e : S}{\Gamma \vdash q ! \ell\langle e \rangle . P : G \upharpoonright p}$$

T-RECV

$$\frac{\boxed{\exists(j \in I), \mathcal{R}(\overset{\ell_j(S_j)}{q \rightarrow p}, G)} \quad \Gamma, x_i : S_i \vdash P_i : G' \upharpoonright p \quad \forall G \setminus \overset{\ell_i(S_i)}{q \rightarrow p} = G'}{\Gamma \vdash \sum_{i \in I} q ? \ell_i(x_i) . P_i : G \upharpoonright p}$$

T-SKIP

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

(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 $\mathcal{W}(r, G) = \exists \alpha, G \setminus \alpha = G' \wedge r \notin \text{ports}(\alpha)$

– this means

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

$$\Gamma \vdash \sum_{i \in I} q? \ell_i(x_i). P_i : G \upharpoonright p$$

T-SKIP

$\mathcal{W}(r, G)$

$\Gamma \vdash P : G' \upharpoonright r \quad \forall G \setminus \alpha = G' \text{ s.t. } r \notin \text{parts}(\alpha)$

$\Gamma \vdash P : G \upharpoonright r$

Semantics

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

Although the 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.

$$\frac{\mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \setminus \mathbf{p} \rightarrow \mathbf{q} = G_i \quad \frac{[\mu \mathbf{X}.G/X]G \setminus \alpha = G'}{\mu \mathbf{X}.G \setminus \alpha = G'}}{\frac{\forall(i \in I), G_i \setminus \alpha = G'_i \quad \text{parts}(\alpha) \cap \{\mathbf{p}, \mathbf{q}\} = \emptyset}{\mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \setminus \alpha = \mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G'_i\}_{i \in I}}}$$

Global Type Bisimilarity

We define our own custom equality for global types. In particular, our global type equality is 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 **never** use syntactic equality,
in our type system, only $G \sim G'$

We define
equality is

$G_1 \sim G_2$ 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 $[\mu X.G/X]G \sim \mu X.G$

Example

Consider again:

$$G = \mu X. p \rightarrow q : \left\{ \begin{array}{l} \text{REQ}(\text{nat}).q \rightarrow r : \text{REQ}(\text{bool}).X \\ \text{END}() \quad .q \rightarrow r : \text{END}().\text{done} \end{array} \right\}$$

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

Example

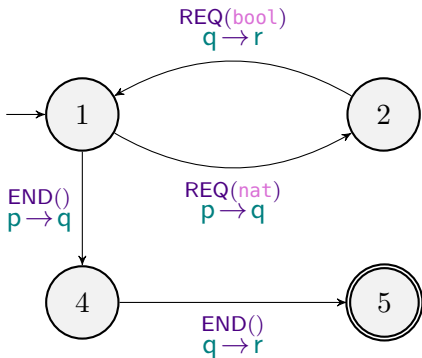
Consider again:

$$G = \mu X. p \rightarrow q : \left\{ \begin{array}{l} \text{REQ}(\text{nat}).q \rightarrow r : \text{REQ}(\text{bool}).X \\ \text{END}() \quad .q \rightarrow r : \text{END}().\text{done} \end{array} \right\}$$

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

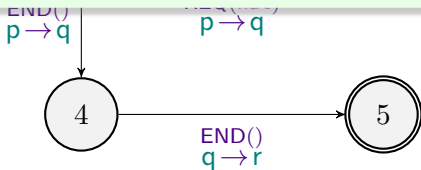
$$\mu X. p \rightarrow q : \left\{ \begin{array}{l} \text{REQ}(\text{nat}).q \rightarrow r : \text{REQ}(\text{bool}).X \\ \text{END}() \quad .q \rightarrow r : \text{END}().\text{done} \end{array} \right\}$$



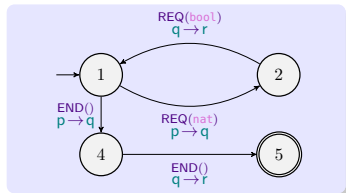
Example: Semantic View of Global Types

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

(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.)

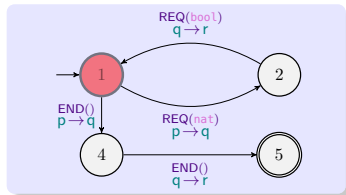


Example: Process & Typing



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

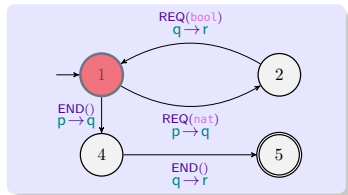
Example: Process & Typing



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

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

Example: Process & Typing

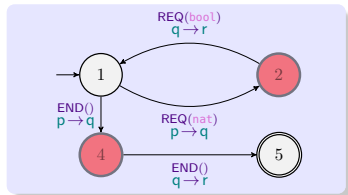


$$P = \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{print}(x). \text{rec } X . \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x). X \\ \text{q?END}(_).\text{done} \end{array} \right\} \\ \text{q?END}(_).\text{done} \end{array} \right\}$$

Goal: $\boxed{\cdot \vdash P : 1 \upharpoonright r}$ – 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.

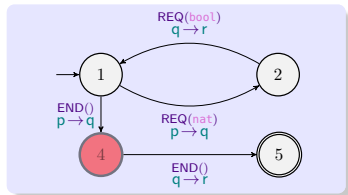
Example: Process & Typing



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

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

Example: Process & Typing

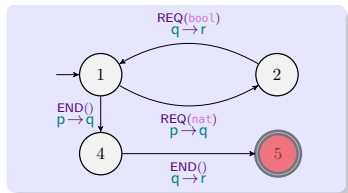


$$P = \sum \left\{ \begin{array}{l} q?REQ(x).print(x). \text{rec } X . \sum \left\{ \begin{array}{l} q?REQ(x).process(x). X \\ q?END(_).done \end{array} \right\} \\ q?END(_).done \end{array} \right\}$$

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

- At 4 , we have that $END() \quad q \rightarrow r$
- We transition to 5 , where the process is ended, and r can no longer take any action in G .

Example: Process & Typing

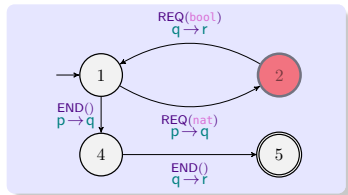


$$P = \sum \left\{ \begin{array}{l} q?REQ(x).print(x). \text{rec } X . \sum \left\{ \begin{array}{l} q?REQ(x).process(x). X \\ q?END(_).done \end{array} \right\} \\ q?END(_) \text{done} \end{array} \right\}$$

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

- At 4 , we have that $END() \quad q \rightarrow r$
- We transition to 5 , where the process is ended, and r can no longer take any action in G .

Example: Process & Typing

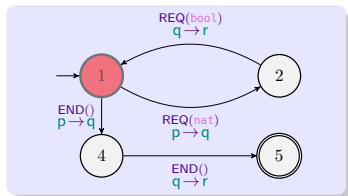


$$P = \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{print}(x). \text{rec } X . \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x). X \\ \text{q?END}(_).\text{done} \end{array} \right\} \\ \text{q?END}(_).\text{done} \end{array} \right\}$$

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

- At 2 , we have that $\text{REQ}(\text{bool})$
 $\text{q} \rightarrow \text{r}$
- We transition back to 1 .

Example: Process & Typing

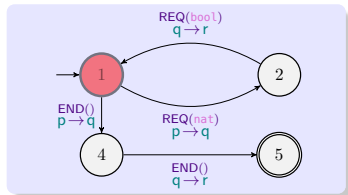


$$P = \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{print}(x). \\ \text{q?END}(_).\text{done} \end{array} \right. \text{rec } X . \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x). X \\ \text{q?END}(_).\text{done} \end{array} \right\} \left. \right\}$$

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

- At 2 , we have that $\text{REQ}(\text{bool})$
 $\text{q} \rightarrow \text{r}$
- We transition back to 1 .

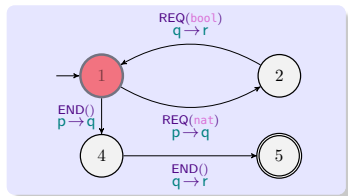
Example: Process & Typing



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

With a **rec** X ., we need to remember the state of the protocol, 1. Whenever we jump back to X , we will check that we are again in a bisimilar state.

Example: Process & Typing

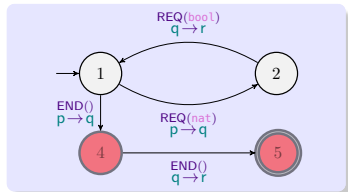


$$P = \sum \left\{ \begin{array}{l} q?REQ(x).print(x). \text{rec } X. \\ q?END(_).done \end{array} \right. \sum \left\{ \begin{array}{l} q?REQ(x).process(x). X \\ q?END(_).done \end{array} \right\}$$

With a **rec** X ., we need to remember the state of the protocol, 1.

– We use again T-SKIP and T-RECV.

Example: Process & Typing

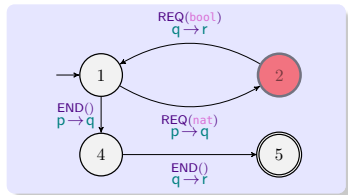


$$P = \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{print}(x). \text{rec } X . \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x). X \\ \text{q?END}(_).\text{done} \end{array} \right\} \\ \text{q?END}(_).\text{done} \end{array} \right\}$$

With a **rec** X ., we need to remember the state of the protocol, 1.

– We use again T-SKIP and T-RECV.

Example: Process & Typing

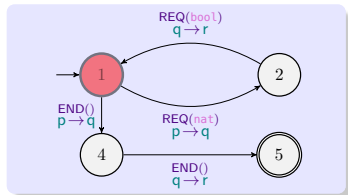


$$P = \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{print}(x). \text{rec } X . \sum \left\{ \begin{array}{l} \text{q?REQ}(x).\text{process}(x). X \\ \text{q?END}(_).\text{done} \end{array} \right\} \\ \text{q?END}(_).\text{done} \end{array} \right\}$$

With a **rec** X ., we need to remember the state of the protocol, 1.

– We use again T-SKIP and T-RECV.

Example: Process & Typing

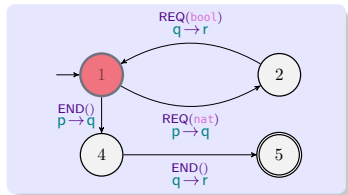


$$P = \sum \left\{ \begin{array}{l} q?REQ(x).print(x). \text{rec } X . \sum \left\{ \begin{array}{l} q?REQ(x).process(x). X \\ q?END(_).done \end{array} \right\} \\ q?END(_).done \end{array} \right\}$$

With a **rec X** ., we need to remember the state of the protocol, 1.

– We use again T-SKIP and T-RECV.

Example: Process & Typing

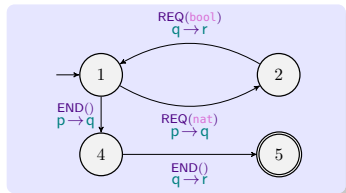


$$P = \sum \left\{ \begin{array}{l} q?REQ(x).print(x). \text{rec } X . \sum \left\{ \begin{array}{l} q?REQ(x).process(x). X \\ q?END(_).done \end{array} \right\} \\ q?END(_).done \end{array} \right\}$$

With a **rec** X ., we need to remember the state of the protocol, 1.


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

Example: Process & Typing



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

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



Properties of Synthetic MPST

Wrap Up



Benefits of Synthetic Typing

↑ TODO