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?

```go
func Worker(n int, resp chan int, err chan error) { ... }
func Master(reqCh chan int, respCh chan []int, cErrCh chan error) {
  for {
    uboun        DEADLOCK!
    work
    errCh        ORPHAN MESSAGES!
    for
      wo           NO RESOURCE CLEANUP!
      go
    }
    var           ...
    for
      sel
      case sql := <-workerChs[i]:
        res = append(res, sql)
      case err := <-errCh:
        cErrCh <- err
        return
      }}
    respCh <- res}}
```

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
    worke
    errCh      Master needs to guarantee that all Workers are notified
    for i      when there is an error.
      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}}
```

Key Idea

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

In a nutshell:

1. <u>Global types</u>: protocol specifications among a fixed number of different *roles*.
2. <u>Role</u>: sets of interactions that processes can do in a protocol.
3. <u>Local types</u>: protocol specifications *from the point of view of a single role*.
4. <u>Projection</u>: a *partial function* that extracts *local type* given a *global types* and a *role*.

5. <u>Well-formedness</u>: guarantees **deadlock-freedom**, usually defined in terms of *projectability*.

global type { G

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

projection

local types { L_1 L_2 L_3

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

processes { W_1 W_2 W_3

global type { G

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

projection

local types { L_1 L_2 L_3

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

typecheck

processes { W_1 W_2 W_3

global type $\{$ G

G =
    $W_1 \rightarrow W_2$: Int.

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

local types $\{$ L_1 L_2 L_3

typecheck

processes $\{$ W_1 W_2 W_3

$L_1 = W_2$! Int
$L_2 = W_1$? Int.
    W_3 ! Bool
$L_3 = W_2$? Bool
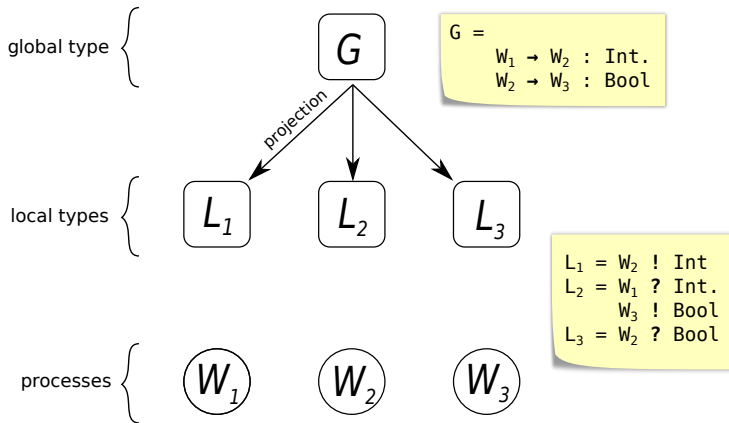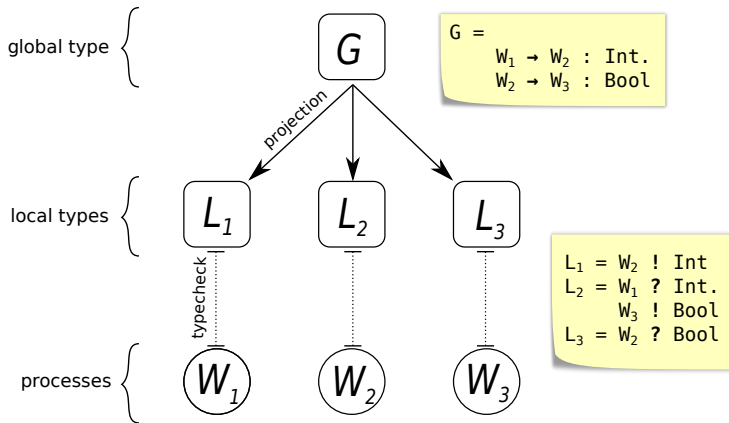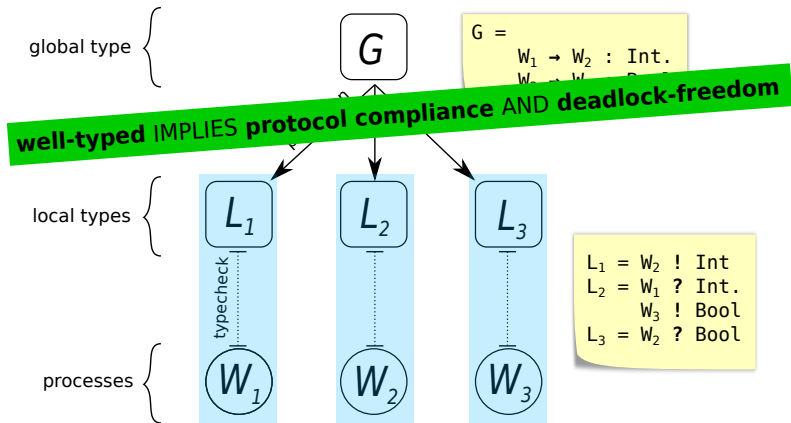
Global and Local Types

Roles			$\mathsf{p}, \mathsf{q}, \ldots$	
Sorts	S	$:=$	$\mathsf{bool} \mid \mathsf{nat} \mid \cdots$	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

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

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

Projection

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

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

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

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

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

Projection

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

It gets complicated very quickly!

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

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

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

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

What is the point of ⊓?

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

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

What is the point of ⊓?

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

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

Projecting r

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

$$=$$

What is the point of ⊓?

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

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

Projecting r

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

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

Processes and Typing

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

Process Typing (simplified)

Once we have local types, process typing is simple:

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

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

"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 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

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

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

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

Synthetic Behavioural Typing for Multiparty Sessions

Sung...
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

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

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

T-RECV
$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \upharpoonright \mathsf{p} \qquad \forall \, G \setminus \mathsf{q} \xrightarrow{\ell_i(S_i)} \mathsf{p} = G'}{\Gamma \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i : G \upharpoonright \mathsf{p}}$$

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).

What is wrong with these rules?

T-SEND

$$\dfrac{\Gamma \vdash P : G' \restriction \mathsf{p} \qquad G \setminus \mathsf{p} \xrightarrow{\ell(S)} \mathsf{q} = G' \qquad \Gamma \vdash e : S}{\Gamma \vdash \mathsf{q}\,!\,\ell\langle e\rangle.P : G \restriction \mathsf{p}}$$

T-RECV

$$\dfrac{\Gamma, x_i : S_i \vdash P_i : G' \restriction \mathsf{p} \qquad \forall\, G \setminus \mathsf{q} \xrightarrow{\ell_i(S_i)} \mathsf{p} = G'}{\Gamma \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i : G \restriction \mathsf{p}}$$

T-SKIP

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

Hint: the problem is in these rules

T-RECV
$$\frac{\Gamma, x_i : S_i \vdash P_i : G' \restriction \mathsf{p} \qquad \forall\, G \setminus \mathsf{q} \overset{\ell_i(S_i)}{\to} \mathsf{p} = G'}{\Gamma \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i : G \restriction \mathsf{p}}$$

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

T-RECV

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

New (Synthetic) Core Typing Rules

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

T-RECV

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

New (Synthetic) Core Typing Rules

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

T-RECV

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

(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 $\mathcal{W}(\mathsf{r}, G) = \exists \alpha, \ G \setminus \alpha = G' \wedge \mathsf{r} \notin \mathsf{ports}(\alpha)$
– this means that an interaction r can "wait" for another (possibly unrelated) interaction in G.

T-SEND

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

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

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

T-RECV

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

(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 $\mathcal{W}(r, G) = \exists \alpha, \ G \setminus \alpha = G' \wedge r \notin \mathsf{ports}(\alpha)$
– this means that an interaction r can "wait" for another (possibly unrelated) interaction in G.

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

(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 $\mathcal{W}(\mathsf{r}, G) = \exists \alpha, \ G \setminus \alpha = G' \wedge \mathsf{r} \notin \mathsf{ports}(\alpha)$
– this means that an interaction r can "wait" for another (possibly unrelated) interaction in G.

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

T-RECV
$$\frac{\boxed{\exists (j \in I), \mathcal{R}(\mathsf{q} \xrightarrow{\ell_j(S_j)} \mathsf{p}, G)} \qquad \Gamma, x_i : S_i \vdash P_i : G' \upharpoonright \mathsf{p} \qquad \forall \, G \setminus \mathsf{q} \xrightarrow{\ell_i(S_i)} \mathsf{p} = G'}{\Gamma \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i : G \upharpoonright \mathsf{p}}$$

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

(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 $\mathcal{W}(\mathsf{r}, G) = \exists \alpha, \; G \setminus \alpha = G' \wedge \mathsf{r} \notin \mathsf{ports}(\alpha)$
– this mea G.

- The rules look more complex than with a syntactic approach, but computing $G \setminus \overset{\ell_i(S_i)}{\mathsf{q} \to \mathsf{p}} = G'$ 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.

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

T-SKIP

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

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.

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

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

Global Type Bisimilarity

We 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 define
equality is

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

$G_1 \sim G_2$ iff:

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

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

Example

Consider again:

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

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

Example

Consider again:

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

We are going to typecheck a process implementing role r...
but first, let's get rid of the syntax for G!

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

Example: Semantic View of Global Types

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

(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.)
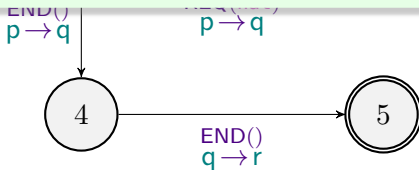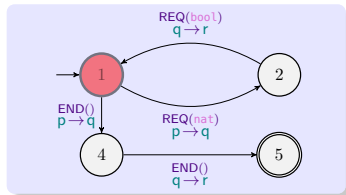
END()
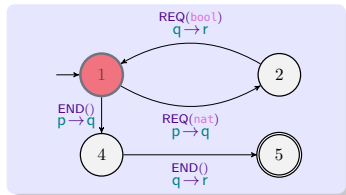p → q

REQ(bool)
p → q

(4) → (5)

END()
q → r

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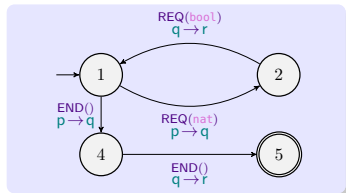
$$P = \sum \left\{ \begin{array}{l} \mathsf{q?REQ}(x).\mathsf{print}(x).\ \mathsf{rec}\,X\,.\sum \left\{ \begin{array}{l} \mathsf{q?REQ}(x).\mathsf{process}(x).\ X \\ \mathsf{q?END}(\_).\mathsf{done} \end{array} \right\} \\ \mathsf{q?END}(\_).\mathsf{done} \end{array} \right\}$$

Example: Process & Typing

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

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

Example: Process & Typing

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

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.

Properties of Synthetic MPST

Wrap Up

Benefits of Synthetic Typing

TODO