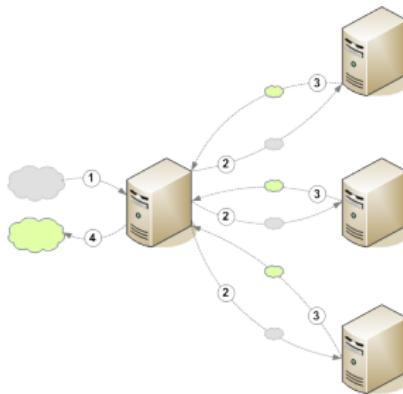


# Dynamic Multirole Session Types

Pierre-Malo Deniélou    Nobuko Yoshida

Imperial College London



malo@doc.ic.ac.uk    yoshida@doc.ic.ac.uk

[POPL'11]

## Binary session types

They describe the use of channel endpoints in the  $\pi$ -calculus.

# Safety of communicating systems by typing

## Binary session types

They describe the use of channel endpoints in the  $\pi$ -calculus.

$$\begin{array}{lll} \text{alice} & = & c! \langle 4 \rangle . c?(x) . c?(z) . \mathbf{0} \quad !\langle \text{nat} \rangle . ?\langle \text{nat} \rangle . ?\langle \text{string} \rangle \\ \text{bob} & = & c?(y) . c! \langle y + 1 \rangle . c! \langle \text{"apple"} \rangle \quad ?\langle \text{nat} \rangle . !\langle \text{nat} \rangle . !\langle \text{string} \rangle \end{array} \quad \checkmark$$

# Safety of communicating systems by typing

## Binary session types

They describe the use of channel endpoints in the  $\pi$ -calculus.

$$\begin{array}{ll} \text{alice} = c! \langle 4 \rangle . c?(x) . c?(z) . \mathbf{0} & !\langle \text{nat} \rangle . ?\langle \text{nat} \rangle . ?\langle \text{string} \rangle \\ \text{bob} = c?(y) . c! \langle y + 1 \rangle . c! \langle \text{"apple"} \rangle & ?\langle \text{nat} \rangle . !\langle \text{nat} \rangle . !\langle \text{string} \rangle \end{array}$$

✓

- Type safety: payload types correspond.

$$\begin{array}{ll} \text{alice} = c! \langle \text{"apple"} \rangle . c?(x) . c?(z) . \mathbf{0} & !\langle \text{string} \rangle . ?\langle \text{nat} \rangle . !\langle \text{string} \rangle \\ \text{bob} = c?(y) . c! \langle y + 1 \rangle . c! \langle \text{"apple"} \rangle & ?\langle \text{nat} \rangle . !\langle \text{nat} \rangle . ?\langle \text{string} \rangle \end{array}$$

✗

# Safety of communicating systems by typing

## Binary session types

They describe the use of channel endpoints in the  $\pi$ -calculus.

$$\begin{array}{ll} \text{alice} = c! \langle 4 \rangle . c?(x) . c?(z) . \mathbf{0} & !\langle \text{nat} \rangle . ?\langle \text{nat} \rangle . ?\langle \text{string} \rangle \\ \text{bob} = c?(y) . c! \langle y+1 \rangle . c! \langle \text{"apple"} \rangle & ?\langle \text{nat} \rangle . !\langle \text{nat} \rangle . !\langle \text{string} \rangle \end{array} \quad \checkmark$$

- Type safety: payload types correspond.

$$\begin{array}{ll} \text{alice} = c! \langle \text{"apple"} \rangle . c?(x) . c?(z) . \mathbf{0} & !\langle \text{string} \rangle . ?\langle \text{nat} \rangle . !\langle \text{string} \rangle \\ \text{bob} = c?(y) . c! \langle y+1 \rangle . c! \langle \text{"apple"} \rangle & ?\langle \text{nat} \rangle . !\langle \text{nat} \rangle . ?\langle \text{string} \rangle \end{array} \quad \times$$

- Communication safety: sends are matched by receives and vice-versa.

$$\begin{array}{ll} \text{alice} = c?(x) . c! \langle \text{"apple"} \rangle & c : ?\langle \text{nat} \rangle . !\langle \text{string} \rangle \\ \text{bob} = c?(y) . c! \langle y+1 \rangle & c : ?\langle \text{nat} \rangle . !\langle \text{nat} \rangle \end{array} \quad \times$$

Ref.: [Takeushi,Honda,Kubo,PARLE'94][Honda,Vasconcelos,Kubo,ESOP'98]

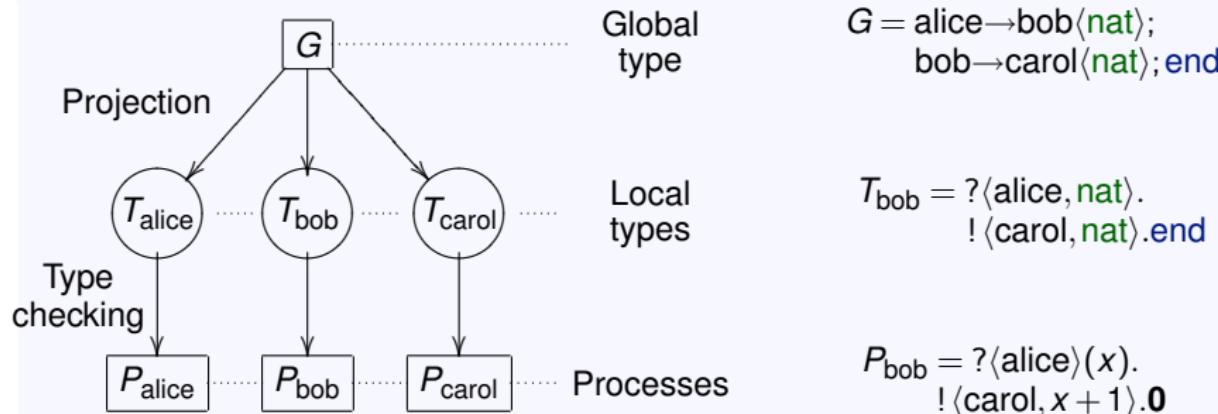
# Multiparty session types (MPST)

- Today's distributed applications involve more and more agents that interact through complex communication patterns.
- Multiparty sessions types can describe these interactions and statically ensure type and communication safety, progress and fidelity to a stipulated protocol involving the orchestration of several channels.

# Multiparty session types (MPST)

- Today's distributed applications involve more and more agents that interact through complex communication patterns.
- Multiparty sessions types can describe these interactions and statically ensure type and communication safety, progress and fidelity to a stipulated protocol involving the orchestration of several channels.

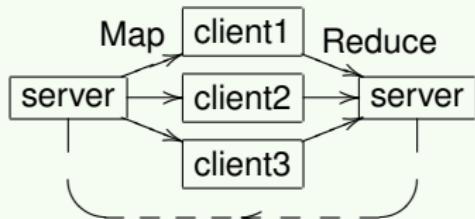
## Multiparty session types in a nutshell



Ref.: [Carbone, Honda, Yoshida, POPL'08] [Bettini et al., CONCUR'08]

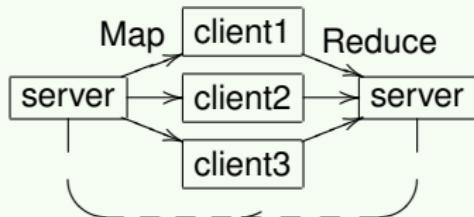
# Multiparty session example

## Map-Reduce in MPST


$$G_{\text{org}} = \mu x. (\text{server} \rightarrow \text{client1} \langle \text{Map} \rangle ; \text{server} \rightarrow \text{client2} \langle \text{Map} \rangle ; \text{server} \rightarrow \text{client3} \langle \text{Map} \rangle ; \text{client1} \rightarrow \text{server} \langle \text{Reduce} \rangle ; \text{client2} \rightarrow \text{server} \langle \text{Reduce} \rangle ; \text{client3} \rightarrow \text{server} \langle \text{Reduce} \rangle); x$$

# Multiparty session example

## Map-Reduce in MPST

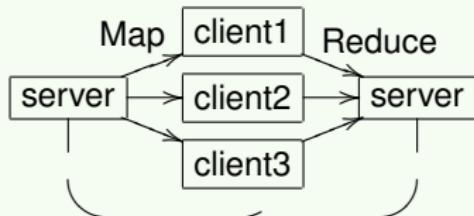

$$G_{\text{org}} = \mu x. (\text{server} \rightarrow \text{client1} \langle \text{Map} \rangle; \\ \text{server} \rightarrow \text{client2} \langle \text{Map} \rangle; \\ \text{server} \rightarrow \text{client3} \langle \text{Map} \rangle; \\ \text{client1} \rightarrow \text{server} \langle \text{Reduce} \rangle; \\ \text{client2} \rightarrow \text{server} \langle \text{Reduce} \rangle; \\ \text{client3} \rightarrow \text{server} \langle \text{Reduce} \rangle); x$$

## Multiparty session type system properties

- Type safety (no payload type error)
- Communication safety (no communication mismatch)
- Progress (deadlock freedom)

# Multiparty session example

## Map-Reduce in MPST


$$G_{\text{org}} = \mu x. (\text{server} \rightarrow \text{client1} \langle \text{Map} \rangle; \\ \text{server} \rightarrow \text{client2} \langle \text{Map} \rangle; \\ \text{server} \rightarrow \text{client3} \langle \text{Map} \rangle; \\ \text{client1} \rightarrow \text{server} \langle \text{Reduce} \rangle; \\ \text{client2} \rightarrow \text{server} \langle \text{Reduce} \rangle; \\ \text{client3} \rightarrow \text{server} \langle \text{Reduce} \rangle); x$$

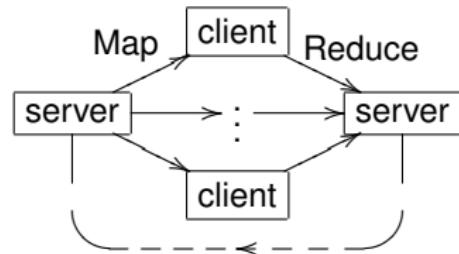
## Multiparty session type system properties

- Type safety (no payload type error)
- Communication safety (no communication mismatch)
- Progress (deadlock freedom)

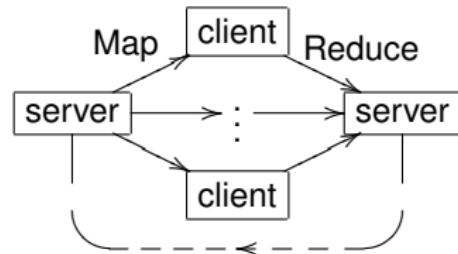
## Open problem: dynamic sets of participants

Multiparty protocols are often more **dynamic**: participants **join** and **leave** long-running instances. Ex: Pub-sub, clusters, web services, ... etc.

# Map-Reduce with dynamic multirole session types



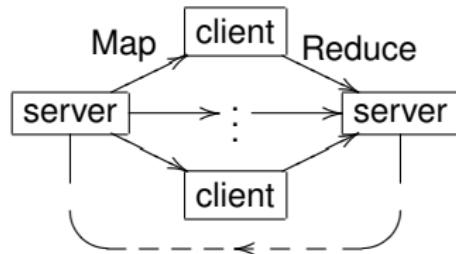
# Map-Reduce with dynamic multirole session types



## Roles

Roles correspond to a communication pattern within a session. Multiple participants can instantiate a given role in the same session. A participant can play several roles.

# Map-Reduce with dynamic multirole session types



## Roles

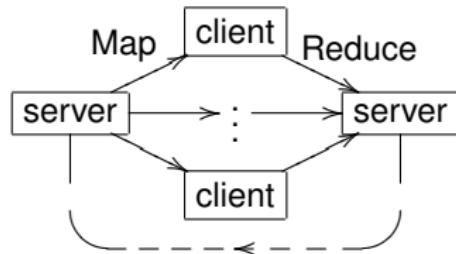
Roles correspond to a communication pattern within a session. Multiple participants can instantiate a given role in the same session. A participant can play several roles.

## Universal quantification

$\forall x : r. G'$  quantifies over the current participants  $p_1, \dots, p_n$  of role  $r$  and, in parallel processes, binds  $x$  to each in the subsequent interaction, as in

$$\forall x : r. G' \equiv G'\{p_1/x\} \parallel \dots \parallel G'\{p_n/x\}$$

# Map-Reduce with dynamic multirole session types



## Global type

$$G = \mu x. \forall x: \text{client}. \{ \text{server} \rightarrow x \langle \text{Map} \rangle; x \rightarrow \text{server} \langle \text{Reduce} \rangle \}; x$$

## Roles

Roles correspond to a communication pattern within a session. Multiple participants can instantiate a given role in the same session. A participant can play several roles.

## Universal quantification

$\forall x: r. G'$  quantifies over the current participants  $p_1, \dots, p_n$  of role  $r$  and, in parallel processes, binds  $x$  to each in the subsequent interaction, as in

$$\forall x: r. G' \equiv G' \{ p_1 / x \} \parallel \dots \parallel G' \{ p_n / x \}$$

# Dynamic multirole session types

## Roles

Roles correspond to a communication pattern within a session. Multiple participants can instantiate a given role in the same session. A participant can play several roles.

## Universal quantification

$\forall x:r.G'$  quantifies over the current participants  $p_1, \dots, p_n$  of role  $r$  and, in parallel processes, binds  $x$  to each in the subsequent interaction, as in

$$\forall x:r.G' \equiv G'\{p_1/x\} \parallel \dots \parallel G'\{p_n/x\}$$

## Target properties for dynamic multirole session types

- Type safety (no payload type error)
- Communication safety (no communication mismatch)
- Progress (deadlock freedom)
- **Join progress (inclusion of joining participants)**

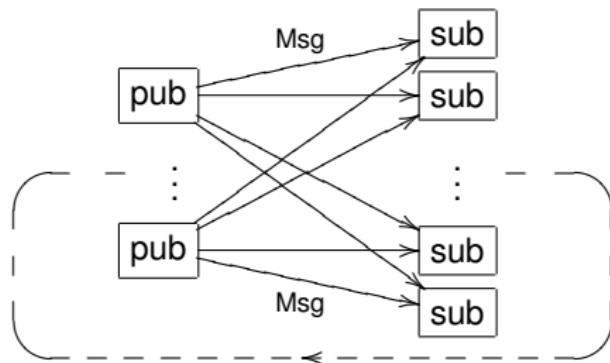
# Global Types

Global types follow standard Multiparty Session Type syntax, with the addition of universal quantification and explicit parallel composition.

$G ::=$	Global types
$p \rightarrow p' \{ I_i \langle \vec{p}_i \rangle \langle U_i \rangle . G_i \}_{i \in I}$	Labelled messages
$\forall x : r \setminus \vec{p}. G$	Universal quantification
$G \parallel G'$	Parallel composition
$G ; G'$	Sequential composition
$\mu x. G$	Recursion
$x$	Recursion variable
$\varepsilon$	Inaction
$\text{end}$	End

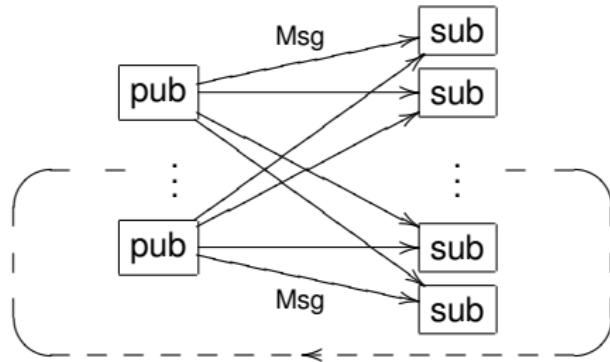
# Publisher-Subscriber example

A set of publishers repeatedly broadcast their messages to a set of subscribers.



# Publisher-Subscriber example

A set of publishers repeatedly broadcast their messages to a set of subscribers.



## Global type for Pub-Sub

We write the global type using the universal quantifier for both the pub and the sub roles. The global type is the following:

$$\mu \mathbf{x}. (\forall x : \text{pub}. \forall y : \text{sub}. x \rightarrow y \langle \text{Msg} \rangle); \mathbf{x}$$

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ a[\text{alice}:\text{pub}](s).\mu X.(\end{aligned}$$

$s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$

$$P_0 = a\langle G \rangle$$

$$\begin{aligned} P_{\text{bob}} = \\ a[\text{bob}:\text{sub}](s).\mu X.(\end{aligned}$$

$s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$

$$\begin{aligned} P_{\text{carol}} = \\ a[\text{carol}:\text{sub}](s).\mu X.(\end{aligned}$$

$s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ a[\text{alice}:\text{pub}](s).\mu X.(\end{aligned}$$

$$(v s)(a(s)[\emptyset] \quad | \quad s:\varepsilon)$$

$$\begin{aligned} P_{\text{bob}} = \\ a[\text{bob}:\text{sub}](s).\mu X.(\end{aligned}$$

$$\begin{aligned} P_{\text{carol}} = \\ a[\text{carol}:\text{sub}](s).\mu X.(\end{aligned}$$

[INIT]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\boxed{P_{\text{alice}} = s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}; \mu X \dots}$$

$$(v s)(a(s)[\text{pub}:\{\text{alice}\}] | s : \epsilon)$$

$$\boxed{P_{\text{bob}} = a[\text{bob}:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X}$$

$$\boxed{P_{\text{carol}} = a[\text{carol}:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X}$$

[INIT] [JOIN]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ s \forall (y:\text{sub}).\{ \\ s! \langle y, \text{Msg}(m) \rangle\}; \mu X \dots \end{aligned}$$

$$(v s)(a(s)[\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob}\}] | \\ s : \varepsilon)$$

$$\begin{aligned} P_{\text{bob}} = \\ s \forall (x:\text{pub}).\{ \\ s? \langle x, \text{Msg}(w) \rangle\}; \mu X \dots \end{aligned}$$

$$\begin{aligned} P_{\text{carol}} = \\ a[\text{carol}:\text{sub}](s).\mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

[INIT] [JOIN] [JOIN]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a(G)$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ s \forall (y:\text{sub}).\{ \\ s! \langle y, \text{Msg}(m) \rangle\}; \mu X \dots \end{aligned}$$

$$(\nu s)(a(s)[\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob}\}] | \\ s : \varepsilon)$$

$$\begin{aligned} P_{\text{bob}} = \\ s? \langle \text{alice}, \text{Msg}(w) \rangle; \mu X \dots \end{aligned}$$

$$\begin{aligned} P_{\text{carol}} = \\ a[\text{carol}:\text{sub}](s).\mu X. \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

[INIT] [JOIN] [JOIN] [POLL]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu \mathbf{x}. (\forall x : \text{pub}. \forall y : \text{sub}. x \rightarrow y(\text{Msg}); \mathbf{x})$$

$$P_0 = a(G)$$

$$P(z : \text{pub}, m) = a[z : \text{pub}](s). \mu X. (s \forall (y : \text{sub}). \{ s! \langle y, \text{Msg}(m) \rangle \}); X$$

$$P(z : \text{sub}) = a[z : \text{sub}](s). \mu X. (s \forall (x : \text{pub}). \{ s? \langle x, \text{Msg}(w) \rangle \}); X$$

$$P_{\text{alice}} = \\ s! \langle \text{bob}, \text{Msg}(m) \rangle; \mu X \dots$$

$$(v s)(a(s)[\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob}\}] | \\ s : \epsilon)$$

$$P_{\text{bob}} = \\ s? \langle \text{alice}, \text{Msg}(w) \rangle; \mu X. (\\ s \forall (x : \text{pub}). \{ s? \langle x, \text{Msg}(w) \rangle \}); X$$

$$P_{\text{carol}} = \\ a[\text{carol} : \text{sub}](s). \mu X. (\\ s \forall (x : \text{pub}). \{ s? \langle x, \text{Msg}(w) \rangle \}); X$$

[INIT] [JOIN] [JOIN] [POLL] [POLL]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ \mu X. ( \\ s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X \end{aligned}$$

$$(v s)(a(s)[\text{pub}:\{\text{alice}\}, \text{sub}:\{\text{bob}\}] | \\ s: \langle \text{alice}, \text{bob}, \text{Msg}(m) \rangle)$$

$$\begin{aligned} P_{\text{bob}} = \\ s? \langle \text{alice}, \text{Msg}(w) \rangle; \mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

$$\begin{aligned} P_{\text{carol}} = \\ a[\text{carol}:\text{sub}](s).\mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

[INIT] [JOIN] [JOIN] [POLL] [POLL] [SEND]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}\langle m \rangle \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ \mu X. ( \\ s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}\langle m \rangle \rangle\}); X \end{aligned}$$

$$(\nu s)(a(s)[\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob}\}] | \\ s : \varepsilon)$$

$$\begin{aligned} P_{\text{bob}} = \\ \mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

$$\begin{aligned} P_{\text{carol}} = \\ a[\text{carol}:\text{sub}](s).\mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

[INIT] [JOIN] [JOIN] [POLL] [POLL] [SEND] [RECV]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}\langle m \rangle \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}\langle w \rangle \rangle\}); X$$

$$P_{\text{alice}} =$$
$$\mu X. ($$

$$s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}\langle m \rangle \rangle\}); X$$

$$(v s)(a(s)[\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob}, \text{carol}\}] |$$
$$s : \varepsilon)$$

$$P_{\text{bob}} =$$
$$\mu X. ($$

$$s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}\langle w \rangle \rangle\}); X$$

$$P_{\text{carol}} =$$
$$\mu X. ($$

$$s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}\langle w \rangle \rangle\}); X$$

[INIT] [JOIN] [JOIN] [POLL] [POLL] [SEND] [RECV] [JOIN]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ (s! \langle \text{bob}, \text{Msg}(m) \rangle \mid \\ s! \langle \text{carol}, \text{Msg}(m) \rangle); \mu X \dots \end{aligned}$$

$$(\nu s)(a\langle s \rangle [\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob, carol}\}] \mid \\ s : \varepsilon)$$

$$\begin{aligned} P_{\text{bob}} = \\ \mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

$$\begin{aligned} P_{\text{carol}} = \\ \mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

[INIT] [JOIN] [JOIN] [POLL] [POLL] [SEND] [RECV] [JOIN] [POLL]

# Processes and operational semantics by example

## Processes for Pub-Sub

$$G = \mu x.(\forall x:\text{pub}.\forall y:\text{sub}.x \rightarrow y(\text{Msg})); x$$

$$P_0 = a\langle G \rangle$$

$$P(z:\text{pub}, m) = a[z:\text{pub}](s).\mu X.(s \forall (y:\text{sub}).\{s! \langle y, \text{Msg}(m) \rangle\}); X$$

$$P(z:\text{sub}) = a[z:\text{sub}](s).\mu X.(s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X$$

$$\begin{aligned} P_{\text{alice}} = \\ (s! \langle \text{bob}, \text{Msg}(m) \rangle \mid \\ s! \langle \text{carol}, \text{Msg}(m) \rangle); \mu X \dots \end{aligned}$$

$$(\nu s)(a\langle s \rangle [\text{pub} : \{\text{alice}\}, \text{sub} : \{\text{bob, carol}\}] \mid \\ s : \varepsilon)$$

$$\begin{aligned} P_{\text{bob}} = \\ \mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

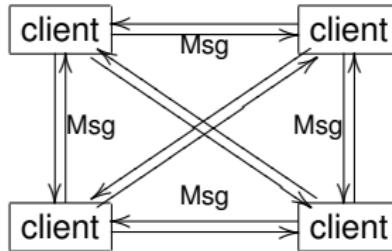
$$\begin{aligned} P_{\text{carol}} = \\ \mu X. ( \\ s \forall (x:\text{pub}).\{s? \langle x, \text{Msg}(w) \rangle\}); X \end{aligned}$$

[INIT] [JOIN] [JOIN] [POLL] [POLL] [SEND] [RECV] [JOIN] [POLL] ... [SEND] [SEND] [POLL] [RECV] ...

# Another example: peer-to-peer chat

At every step, each client sends a message to every other client.

$$G = \mu \mathbf{x}. (\forall x : \text{client}. \forall y : \text{client} \setminus x. \{x \rightarrow y \text{Msg(string)}\}); \mathbf{x}$$



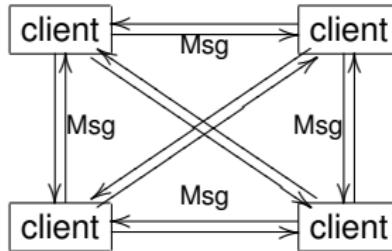
## Local Type

$$T_{\text{client}}(z) = \mu \mathbf{x}. (\forall y : \text{client} \setminus z. \{! \langle y, \text{Msg(string)} \rangle\} \mid \forall x : \text{client} \setminus z. \{? \langle x, \text{Msg(string)} \rangle\}); \mathbf{x}$$

# Another example: peer-to-peer chat

At every step, each client sends a message to every other client.

$$G = \mu \mathbf{x}. (\forall x : \text{client}. \forall y : \text{client} \setminus x. \{x \rightarrow y \text{Msg(string)}\}); \mathbf{x}$$



## Local Type

$$T_{\text{client}}(z) = \mu \mathbf{x}. (\forall y : \text{client} \setminus z. \{! \langle y, \text{Msg(string)} \rangle\} \mid \forall x : \text{client} \setminus z. \{? \langle x, \text{Msg(string)} \rangle\}); \mathbf{x}$$

How do we go from the global type to the local type?

We need a **projection algorithm** that handles universal quantifiers.

# Projection: $G \uparrow z:r$

The projection from a global type  $G$  to a participant  $z$  playing role  $r$  is written  $G \uparrow z:r$ .

## Intuition

$$\begin{aligned} & (\forall x:r.G) \uparrow p_i:r \\ & (G\{p_1/x\} \mid \dots \mid G\{p_i/x\} \mid \dots \mid G\{p_k/x\}) \uparrow p_i:r \\ & (G\{p_1/x\} \uparrow p_i:r) \mid \dots \mid (G\{p_i/x\} \uparrow p_i:r) \mid \dots \mid (G\{p_k/x\} \uparrow p_i:r) \\ & (G\{p_i/x\} \uparrow p_i:r) \mid \forall x:r \setminus p_i.(G \uparrow p_i:r) \end{aligned}$$

# Projection: $G \uparrow z:r$

The projection from a global type  $G$  to a participant  $z$  playing role  $r$  is written  $G \uparrow z:r$ .

## Intuition

$$\begin{aligned} & (\forall x:r.G) \uparrow p_i:r \\ & (G\{p_1/x\} \mid \dots \mid G\{p_i/x\} \mid \dots \mid G\{p_k/x\}) \uparrow p_i:r \\ & (G\{p_1/x\} \uparrow p_i:r) \mid \dots \mid (G\{p_i/x\} \uparrow p_i:r) \mid \dots \mid (G\{p_k/x\} \uparrow p_i:r) \\ & (G\{p_i/x\} \uparrow p_i:r) \mid \forall x:r \setminus p_i.(G \uparrow p_i:r) \end{aligned}$$

## Main rules

$$\begin{aligned} (\forall x:r \setminus \vec{p}.G) \uparrow z:r &= G\{z/x\} \uparrow z:r \mid \forall x:r \setminus z \setminus \vec{p}.(G \uparrow z:r) \quad (z \notin \vec{p}) \\ (\forall x:r \setminus \vec{p}.G) \uparrow z:r &= \forall x:r \setminus \vec{p}.(G \uparrow z:r) \quad (z \in \vec{p}) \\ (\forall x:r' \setminus \vec{p}.G) \uparrow z:r &= \forall x:r \setminus \vec{p}.(G \uparrow z:r) \quad (\text{otherwise}) \end{aligned}$$

The other projection rules are standard.

Ref.: [Yoshida, Deniélo, Bejleri, Hu, FOSSACS'10]

# Typing system: $\Gamma \vdash P \triangleright \Delta$

We show only a selection of rules.

( $\Gamma$  is the environment,  $P$  a process,  $\Delta$  a session type)

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \uparrow p}{\Gamma \vdash u[p](y).P \triangleright \Delta} \text{[JOIN]} \quad \frac{\Gamma \vdash P \triangleright \Delta, c : \text{end}}{\Gamma \vdash \text{quit}\langle c \rangle ; P \triangleright \Delta, c : \text{end}} \text{[LEAVE]}$$
$$\frac{\Gamma, x : r \vdash P \triangleright c : T \quad \Gamma \vdash \vec{p}}{\Gamma \vdash c \forall (x : r \setminus \vec{p}). \{P\} \triangleright c : \forall x : r \setminus \vec{p}. T} \text{[POLLING]}$$

## Theorem (Type safety)

Suppose  $\Gamma \vdash P \triangleright \Delta$ . For any  $P'$  such that  $P \rightarrow^* P'$ ,  $P'$  has no type error.

## Theorem (Type Safety)

$$\begin{array}{lll} G & = & \text{alice} \rightarrow \text{bob} \text{ Msg}(\text{nat}) \\ \text{alice} & = & s! \langle \text{bob}, \text{Msg "apple"} \rangle \quad s! \langle \text{bob}, \text{Msg(string)} \rangle \\ \text{bob} & = & s? \langle \text{alice}, \text{Msg}(y) \rangle \quad s? \langle \text{alice}, \text{Msg(nat)} \rangle \end{array}$$

×

# Limitations of the system seen so far ...

The semantics and type system presented so far are not constrained enough ...

## Leaving a session

The typing rule [LEAVE] only allows a participant to leave when its local type is `end`. It means that if  $G$  is of the form  $\mu x.G_0; x; \text{end}$ , no one can leave ...

$$\mu x. \forall x : \text{client}. \forall y : \text{client} \setminus x. \{x \rightarrow y \text{Msg}(\text{string})\}; x; \text{end}$$

## Polling consistency for communication safety

$$a[z : \text{client}](s). \mu X. (s \forall (y : \text{client} \setminus z). \{s! \langle y, \text{Msg}(m) \rangle\} \\ | s \forall (x : \text{client} \setminus z). \{s? \langle x, \text{Msg}(w) \rangle\}); X)$$

All local polling operations should give the same list, otherwise messages are unexpected or absent.

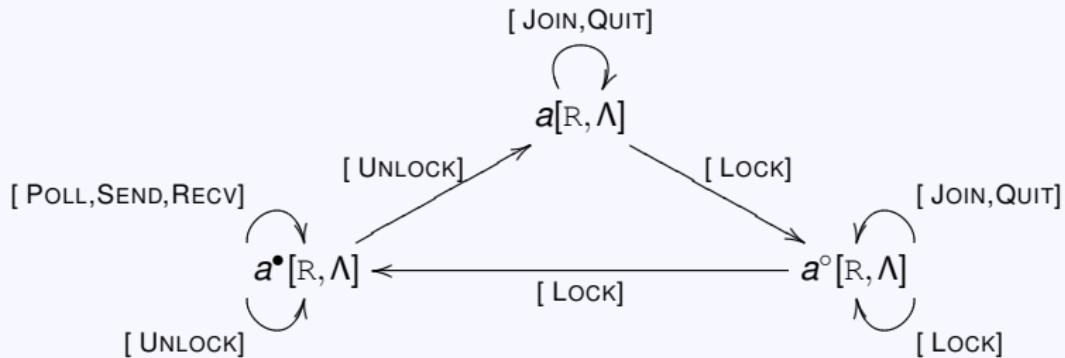
# Multiparty locking

We need to temporarily *block* late participants from joining in the middle of a session execution in order to prevent any interference with polling: we introduce a locking mechanism written  $\text{lock}\{G\}$ .

$$G = \mu x. \text{lock}\{\forall x: \text{client}. \forall y: \text{client} \setminus x. \{x \rightarrow y \text{Msg}(\text{string})\}\}; x; \text{end}$$

It translates locally into two operations of `lock` and `unlock`.

## Semantics



# Locking

## Typing locks

$$G ::= \dots \mid \text{lock}\{G\} \quad T ::= \dots \mid \text{lock} \mid \text{unlock}$$

Well-locked global types are of the form  $\text{lock}\{G_0\}; \text{end}$ .

Persistently well-locked global types are of the form  $\mu x.\text{lock}\{G_0\}; x; \text{end}$

$$\text{lock}\{G\} \uparrow z:r = \text{lock};(G \uparrow z:r); \text{unlock}$$

Typing ensures that locks are in the right positions to avoid deadlocks.  
Participants can safely leave a session instead of locking.

## Single iteration chat client

$$G = \mu x.\text{lock}\{\forall x:\text{client}.\forall y:\text{client} \setminus x.\{x \rightarrow y \text{Msg}(\text{string})\}\}; x; \text{end}$$

$$P_{\text{client}}(p) = a[p:\text{client}](s).s \text{lock};(s \forall(y:\text{client} \setminus z).\{s! \langle y, \text{Msg}(m) \rangle\} \mid s \forall(x:\text{client} \setminus z).\{s? \langle x, \text{Msg}(w) \rangle\}); s \text{unlock}; \text{quit}(s)$$

# Theorems

## Theorem (Type safety)

Suppose  $\Gamma \vdash P \triangleright \Delta$ . For any  $P'$  such that  $P \rightarrow^* P'$ ,  $P'$  has no type error.

 $\forall x : r. G$ 

## Theorem (Communication Safety)

Every sent message is expected by a receiver. Every receiver will receive a message.

 $\text{lock}\{G\}$ 

## Theorem (Progress)

Well-locked and well-typed processes do not reach a deadlock state.

 $\mu x. \text{lock}\{G\}; x$ 

## Theorem (Join progress)

Persistantly well-locked and well-typed processes can progress and integrate new joiners.

 $\mu x. \text{lock}\{G\}; x$

# Implementation

- An extension to OCaml: addition of session declarations.
- The compiler generates from the global type a taylored runtime
- The runtime deals with transport (UDP, TCP, AMQP) and registry
- Programmers can use a continuation-based programming interface
- Type safety comes from the Ocaml type system.
- Communication safety, progress and join progress are ensured by the runtime.

## Demo

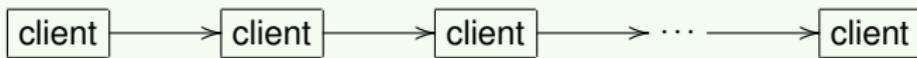
Don't hesitate to ask for one ;-).

Ref.: [Corin,Deniélová,TGC'07][Barghavan,Corin,Deniélová,Fournet,Leifer,CSF'09]

# Extension (work in progress) to dynamic topologies

Many situations are still outside of the scope of dynamic multirole session types.

## Line topology



We equip roles with *topologies*, ie relations between participants

We introduce new quantifiers for global and local types:

- $\text{foreach}(x < y : r)\{G\}$  follows sequentially the topology;
  - The protocol defined above is:  $\text{foreach}(x < y : \text{client})\{x \rightarrow y \langle \text{Msg} \rangle\}$ .
  - Projection to  $z$  gives:  $\forall(x : \text{pred}(z))\{\text{?}\langle x, \text{Msg} \rangle\}; \forall(y : \text{succ}(z))\{\text{!}\langle y, \text{Msg} \rangle\}$
- $\text{pareach}(x < y : r)\{G\}$  executes all edges concurrently.

## Topologies are external

- Absent from the global types, local types, processes.

Consequently, type checking is as easy as in DMST, yet all the strong safety properties still hold!

# Conclusion and future work

## Dynamic multirole session types

- A notion of **roles** to abstract dynamic sets of participants
- A new ***universal quantification***  $\forall x : r. G$  to ease programming and typing
- Statically enforced strong ***type***, ***communication safety*** and ***progress*** guarantees

## Ongoing work

- Integration to a programming language with more distributed implementations.
- Give a structured topology to roles: participants form more than just sets
- Security and runtime-enforcement
- Collaborations with industry partners (see next slide ...)

<http://www.doc.ic.ac.uk/~malo/dynamic/>

# Industrial collaboration



## Ocean Observatory Initiative

Building a self-governing cyberinfrastructure  
for oceanography observation.

- Savara, JBoss, RedHat
- RabbitMQ, VMware
- Scribble language



# Appendix

- ➊ Existential
- ➋ Definition of processes
- ➌ Operational semantics
- ➍ Auction example
- ➎ Well-formedness conditions
- ➏ Locked semantics

# Existential quantification?

Ok, let's add it:

$$G ::= \dots \mid \exists x:r.G \mid \dots$$

## Who chooses?

$$G' = \forall x:\text{client.} \{ \exists y:\text{server.} x \rightarrow y \langle \text{Msg} \rangle \}$$

Potentially, a server  $y$  can be chosen by every client  $x$  or by none: for communication safety, every server needs to know for every client if he was chosen.

If the registry chooses, it becomes a special case of role topology or of the notion of subroles. The question of how to specify how the participant is chosen is still open.

# Definition of processes

$P ::=$ Processes		
$a\langle G \rangle$	Session Init	
$a[p](s).P$	Join	$\text{if } e \text{ then } P \text{ else } P$ Conditional
$\text{quit}\langle s \rangle$	Quit	$\mu X.P \mid X \mid \mathbf{0}$ Recursion
$s!\langle p, I \langle \vec{p} \rangle(e) \rangle$	Send	$(\nu a:G)P$ Restriction
$s?\langle p, \{I_i \langle \vec{p}_i \rangle(x_i).P_i\}_{i \in I} \rangle$	Receive	$(\nu s)P$ Session restriction
$s\forall(x:r \setminus \vec{p}).\{P\}$	Poll	$s:h$ Message buffer
$P \parallel P$	Parallel	$a\langle s \rangle[\mathbb{R}]$ Session registry
$P; P$	Sequential	

## Processes for Pub-Sub

$$\begin{aligned} P_0 &= a\langle G \rangle \\ P(z:\text{pub}, m) &= a[z:\text{pub}](s).\mu X.(s\forall(y:\text{sub}).\{s! \langle y, \text{Msg}\langle m \rangle \rangle\}); X \\ P(z:\text{sub}) &= a[z:\text{sub}](s).\mu X.(s\forall(x:\text{pub}).\{s? \langle x, \text{Msg}\langle w \rangle \rangle\}); X \end{aligned}$$

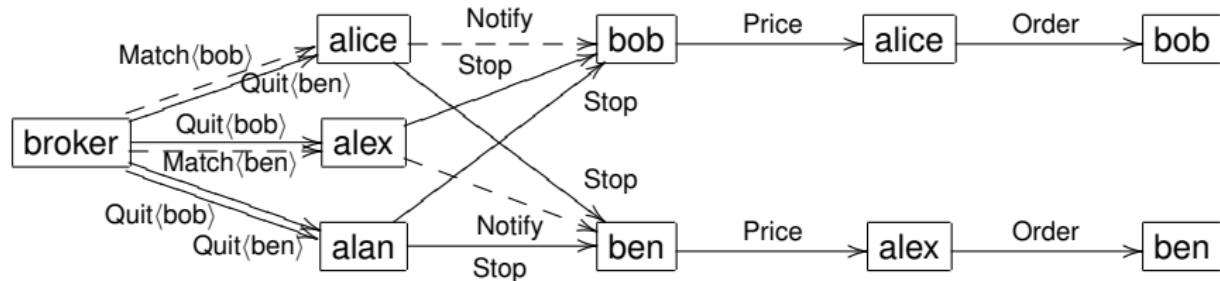
# Operational semantics

$a(s)[\mathbb{R}]$  keeps the current list of participants in  $\mathbb{R}$ .

$$\begin{array}{lcl} a(G) \rightarrow (v s)(a(s)[\mathbb{R}] \mid s : \varepsilon) & & (\forall r_i \in G, \mathbb{R}(r_i) = \emptyset) \mid \text{INIT} \\ a[p:r](y).P \mid a(s)[\mathbb{R} \cdot r:P] \rightarrow P\{s[p:r]/y\} \mid a(s)[\mathbb{R} \cdot r:P \uplus \{p\}] & & \mid \text{JOIN} \\ \text{quit}\langle s[p:r] \rangle \mid a(s)[\mathbb{R} \cdot r:P] \rightarrow a(s)[\mathbb{R} \cdot r:P \setminus p] & & \mid \text{QUIT} \\ \\ s[p:r]! \langle p':r', I(\vec{p})\langle v \rangle \rangle \mid a(s)[\mathbb{R}] \mid s:h \rightarrow a(s)[\mathbb{R}] \mid s:h \cdot (p:r, p':r', I(\vec{p})\langle v \rangle) \\ \quad (p \in \mathbb{R}(r) \wedge p' \in \mathbb{R}(r')) & & \mid \text{SEND} \\ \\ s[p:r]? \langle p':r', \{I_i(\vec{p}_i)\langle x_i \rangle.P_i\}_{i \in I} \rangle \mid a(s)[\mathbb{R}] \\ \quad | s:(p':r', p:r, I_k(\vec{p}_k)\langle v \rangle) \cdot h \rightarrow P_k\{v/x_k\} \mid a(s)[\mathbb{R}] \mid s:h \\ \quad \quad (p \in \mathbb{R}(r) \wedge k \in I) & & \mid \text{RECV} \\ \\ s[p:r'] \forall(x:r \setminus \vec{p}).\{P\} \mid a(s)[\mathbb{R}] \rightarrow P\{p_1/x\} \mid \dots \mid P\{p_k/x\} \mid a(s)[\mathbb{R}] \\ \quad (\mathbb{R}(r) \setminus \vec{p} = \{p_1, \dots, p_k\} \wedge p \in \mathbb{R}(r')) & & \mid \text{POLL} \end{array}$$

# Auction example, disambiguation of parallel branches

A single broker forms pairs of buyers and sellers.



## Global type for Auction

$$G = \forall x : \text{buyer}. \forall y : \text{seller}. \text{broker} \rightarrow x \{ \text{Match}(y). x \rightarrow y \langle \text{Notify} \rangle. y \rightarrow x \langle \text{Price} \rangle. x \rightarrow y \langle \text{Order} \rangle, \\ \text{Quit}(y). x \rightarrow y \langle \text{Stop} \rangle \}; \text{end}$$

# Well-formedness

- Syntax correctness
  - ✗  $G_1 = \mu x.(\text{server} \rightarrow \text{client}\langle\text{Msg}\rangle; x \mid \text{server} \rightarrow \text{broker}\langle\text{Notify}\rangle; x)$
  - ✓  $G_2 = \mu x.(\text{server} \rightarrow \text{client}\langle\text{Msg}\rangle \mid \text{server} \rightarrow \text{broker}\langle\text{Notify}\rangle); x$
  - ✓  $G_3 = \mu x.\text{server} \rightarrow \text{client}\langle\text{Msg}\rangle; x \mid \mu y.\text{server} \rightarrow \text{broker}\langle\text{Notify}\rangle; y$
- Projectability (projection always returns)
  - ✗  $G_4 = \text{broker} \rightarrow \text{buyer}\{\text{Notify}.\text{buyer} \rightarrow \text{seller}\langle\text{Msg}\rangle; \text{seller} \rightarrow \text{buyer}\langle\text{Pay}\rangle, \text{Quit}.\text{buyer} \rightarrow \text{seller}\langle\text{Msg}\rangle\}$
  - ✓  $G_5 = \text{broker} \rightarrow \text{buyer}\{\text{Notify}.\text{buyer} \rightarrow \text{seller}\langle\text{Price}\rangle; \text{seller} \rightarrow \text{buyer}\langle\text{Pay}\rangle, \text{Quit}.\text{buyer} \rightarrow \text{seller}\langle\text{Stop}\rangle\}$
- Linearity (no possible confusion between parallel branches)
  - ✗  $G_6 = \forall x:\text{buyer}.\forall y:\text{seller}. \{\text{broker} \rightarrow x\langle\text{Msg}\rangle.x \rightarrow y\langle\text{Notify}\rangle\}$
  - ✓  $G_7 = \forall x:\text{buyer}.\forall y:\text{seller}. \{\text{broker} \rightarrow x\langle\text{Msg}(y)\rangle.x \rightarrow y\langle\text{Notify}\rangle\}$

# Locked semantics

## Syntax and semantics

$$P ::= \dots \mid c\text{ lock} \mid c\text{ unlock} \mid a^\circ[R, \Lambda] \mid a^\bullet[R, \Lambda]$$
$$\Lambda ::= \emptyset \mid \Lambda \cup \{p:r\}$$
$$s[p:r]\text{lock} \mid a(s)[R] \rightarrow a^\circ(s)[R, \{p:r\}] \quad [\text{LOCK}]$$
$$s[p:r]\text{lock} \mid a^\circ(s)[R, \Lambda] \rightarrow \begin{cases} a^\circ(s)[R, \Lambda \cup \{p:r\}] & (R \not\approx \Lambda \cup \{p:r\}) \\ a^\bullet(s)[R, \Lambda \cup \{p:r\}] & (R \approx \Lambda \cup \{p:r\}) \end{cases} \quad [\text{UP}]$$
$$s[p:r]\text{unlock} \mid a^\bullet(s)[R, \Lambda \cup \{p:r\}] \rightarrow \begin{cases} a^\bullet(s)[R, \Lambda] & (\Lambda \neq \emptyset) \\ a(s)[R] & (\Lambda = \emptyset) \end{cases} \quad [\text{DOWN}]$$
$$s[p:r]! \langle p':r', I(\vec{p})\langle v \rangle \rangle \mid a^\bullet(s)[R, \Lambda] \mid s:h \rightarrow a^\bullet(s)[R, \Lambda] \mid s:h \cdot (p:r, p':r', I(\vec{p})\langle v \rangle)$$
$$\dots$$
$$[\text{SEND}]$$