Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 241 (2009) 3–33
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Synchronous Multiparty Session Types

Andi Bejleri[1] Nobuko Yoshida[2]

Imperial College London

Abstract

Synchronous communication is useful to model multiparty sessions where control for timing events and strong sequentially order of messages are essential to the problem specification. This paper continues the work on multiparty session types initiated by Honda et al. [10] for synchronous communications. It provides a more relaxed syntax of the calculus, multicasting, higher-order communication via multipolarity labels and a clear definition of delegation in global types. The linearity property defines when a channel can be used in two different communications without creating a race condition and the type system checks if all the processes of a session implement the communication behavior specified in the global type. The type system of the calculus is proved to be sound with respect to the operational semantics and coherent with respect to the global types.

Key words: Synchronous Communications, Multipolarity Labels, Multicasting, Delegation, Linearity, Subject Reduction Theorem

1 Introduction

Multiparty session types for a calculus of asynchronous communication have recently been introduced by Honda et al. [10] and Bonelli-Compagnoni [1]. The idea of multiparty session types in the first work is based on the choreography metaphor to describe interactions between processes: interactions are described as a global scenario. Whilst, the second work is based on the orchestration metaphor: interactions between processes are described as a centralised scenario between one master process and many slave processes. The system introduced in this paper follows the metaphor of the first work.

Controlling the timing of events becomes important in multiparty sessions: for example, in a fire alarm system of a building, we expect that all fire alarms run before that elevators become blocked. This scenario would be modeled by a control process that sends in multicast an *ON* message to fire alarms and after that a *BLOCK* message to elevators. The timing of events in the example can be obtained

[1] Email: ab406@doc.ic.ac.uk
[2] Email: yoshida@doc.ic.ac.uk

doi:10.1016/j.entcs.2009.06.002

by modeling the session using *synchronous communications*; the second multicast send will happen only after the first message is received by the multicast group of fire alarms.

Binary session types [16,9] on their own are not rich enough to express dependencies between different interactions in a multiparty session. A notion of global type is therefore introduced in [10] to formalise the global behaviour of a multiparty session. The example below illustrates the key ideas of multiparty sessions, dependencies between interactions and global description. In a `Client-Addition-Successor-Predecessor` session, the communication protocol (conversation) is defined as: `Client` sends two natural numbers to `Addition` and waits to receive from him the sum of them. If the second operand is equal to 0 then `Addition` sends to `Client` the first operand as result, otherwise it sends the first operand to `Successor` and receives from him its successor; after that it sends the second operand to `Predecessor` and receives from him its predecessor; this behaviour is repeated until the second operand is equal to 0. The global description of the communication protocol in a name-arrow based representation is:

$$
\begin{array}{l}
\texttt{Client} \to \texttt{Addition}\colon \langle int \rangle.\\
\texttt{Client} \to \texttt{Addition}\colon \langle int \rangle.\\
\mu \mathbf{t}.\texttt{Addition} \to \{\texttt{Successor}, \texttt{Predecessor}\}\colon \{\\
\qquad\qquad true : \texttt{Addition} \to \texttt{Client}\colon \langle int \rangle.\mathsf{end},\\
\qquad\qquad false : \texttt{Addition} \to \texttt{Successor}\colon \langle int \rangle.\\
\qquad\qquad\qquad \texttt{Successor} \to \texttt{Addition}\colon \langle int \rangle.\\
\qquad\qquad\qquad \texttt{Addition} \to \texttt{Predecessor}\colon \langle int \rangle.\\
\qquad\qquad\qquad \texttt{Predecessor} \to \texttt{Addition}\colon \langle int \rangle.\mathbf{t}\}
\end{array}
$$

where $\texttt{A} \to \{\texttt{B}, \texttt{C}\}\colon \langle U \rangle$ means that participant `A` sends simultaneously a message of type U to participant `B` and `C`, and $\texttt{A} \to \{\texttt{B}, \texttt{C}\}\colon \{l_1\colon \cdots, ..., l_j\colon \cdots\}$ means that participant `A` sends simultaneously a label l_i where $i \in \{1, ..., j\}$ to participant `B` and `C`. We have omitted channels from the example for simplicity.

In binary sessions, the `Client-Addition-Successor-Predecessor` conversation is represented by three sessions: $\texttt{Client-Addition}_1$, $\texttt{Successor-Addition}_2$ and $\texttt{Predecessor-Addition}_3$. The interactive structure of each participant is:

$$
\begin{aligned}
\texttt{Client} &= !\langle int \rangle; !\langle int \rangle; ?\langle int \rangle; \mathsf{end}\\
\texttt{Addition}_1 &= ?\langle int \rangle; ?\langle int \rangle; !\langle int \rangle; \mathsf{end}\\
\texttt{Successor} &= \mu \mathbf{t}.\&\{true : \mathsf{end}, false : ?\langle int \rangle; !\langle int \rangle; \mathbf{t}\}\\
\texttt{Addition}_2 &= \mu \mathbf{t}. \oplus \{true : \mathsf{end}, false : !\langle int \rangle; ?\langle int \rangle; \mathbf{t}\}\\
\texttt{Predecessor} &= \mu \mathbf{t}.\&\{true : \mathsf{end}, false : ?\langle int \rangle; !\langle int \rangle; \mathbf{t}\}\\
\texttt{Addition}_3 &= \mu \mathbf{t}. \oplus \{true : \mathsf{end}, false : !\langle int \rangle; ?\langle int \rangle; \mathbf{t}\}
\end{aligned}
$$

where $!\langle U\rangle$ denotes an output of type U, $?\langle U\rangle$ denotes an input of type U, $\oplus\{l_1\colon \cdots, ..., l_j\colon \cdots\}$ denotes a choice of a label and $\&\{l_1\colon \cdots, ..., l_j\colon \cdots\}$ denotes branching on a set of labels. Processes that implement these interaction structures are well-typed by a binary session type system as the interactive structures between `Client-Addition`$_1$, `Successor-Addition`$_2$ and `Predecessor-Addition`$_3$ are reciprocal to each other. However, the binary session representation of the conversation breaks the order of messages because in the case when the second operand is not 0, `Addition` should add the second operand to the first one before returning it to `Client` and this dependency between the sessions `Client-Addition`$_1$ and `Addition`$_{2,3}$`-Successor-Predecessor` can not be captured by binary session types.

In multiparty sessions types, the conversation is represented by the global description given above and as a consequence the interactive structure of `Addition` is:

$$?\langle int\rangle; ?\langle int\rangle; \mu\mathbf{t}. \oplus \{true :!\langle int\rangle; \mathsf{end}, false :!\langle int\rangle; ?\langle int\rangle; !\langle int\rangle; ?\langle int\rangle; \mathbf{t}\}.$$

and of the other participant `Client`, `Successor` and `Predecessor` is the same to binary session types. `Addition` is represented now by only one interactive structure. Hence, the use of global types allows a more complete and intelligible definition of communication protocols in multiparty sessions.

In the global type definition, a programmer does not specify only the communications of a protocol but also the channels where the communications take place. This is an important feature in multiparty session types since global types are not a simple human interpretable descriptive language; global types together with the projection algorithm and type-system represent a type-checking tool for communication-based processes.

In synchronous communication calculi, the runtime sequence of interactions follows more strictly the sequence of global types than in the asynchronous communication calculus with queue [10], resulting in a simpler typing system of the global behaviour. Consider a global type:

$$\mathtt{A} \to \mathtt{B}\colon m_1\langle U\rangle.\mathtt{A} \to \mathtt{C}\colon m_2\langle U'\rangle.\mathsf{end} \tag{1}$$

where m_1 and m_2 are abstraction of channels. This ordering means "only after the first sending and receiving take place, the second sending and receiving take place" and it is modeled for calculi of synchronous communication but not for asynchronous ones; e.g. in the asynchronous calculus [10] `C` may receive its message before `B`.

A race condition problem is introduced if different interactions use the same channel. That is, two senders are sending two different messages on the same channel and two receivers are trying to receive at the same time a message on that same channel. Even though, there is no communication error, as for each send corresponds one receive and vice versa, the ambiguity introduced on which of the two receivers will the message be delivered may break the causalities. Unfortunately, global types alone do not guarantee from a possible race condition problem.

A *linearity property* defines when the use of a same channel in two different communications of a global type does not break the causalities of it. A precise analysis of ordering two communications without breaking their causalities is used to define a partial order between two non consecutive communications. The existence of this partial order defines if the causalities between the two communications break or not.

Finally, each process is type-checked with the type obtained by projecting the global type on each participant. The type system analyses a set of initialization processes, i.e. processes that are willing/waiting to initiate a session, via a bottom-up strategy.

Contributions. This paragraph summarizes the main technical contributions of this paper.

- **Synchronous Communications**. Synchronous communications are useful to model multiparty sessions where control for timing events and strong sequentially order of messages are essential to the problem specification. The runtime sequence of interactions follows more strictly the one of the global behaviour description than the asynchronous communication calculus with queue [10], resulting in a simpler linear property.
- **A Simpler Calculus**. The syntax of the calculus is more relaxed than the one introduced by Honda et al. [10]. We do not distinguish syntactically between a primitive value send and a session channel send following the idea firstly proposed in [8]. The syntax of the calculus does not introduce queues, neither at the programmer code level nor at the runtime code level, in contrast to [10].
- **Multicasting**. The calculus supports the delivery of messages to a group of peers simultaneously. Multicast increases the expressivity of communication behaviors in this calculus, mainly due to the restriction introduced by projection on parallel composition and branching. That is, the global type $\mathtt{A} \to \mathtt{B} : m_1\langle U\rangle,\ \mathtt{A} \to \mathtt{C} : m_2\langle U\rangle$, where the two interactions can take place in parallel, is not well-formed due to the definition of projection but we can model this behaviour if the values sent are the same, using multicast as $\mathtt{A} \to \{\mathtt{B},\mathtt{C}\} : \{m_1, m_2\}\langle U\rangle$. Also, the global type $\mathtt{A} \to \mathtt{B} : m_1\{l_1 : \mathtt{B} \to \mathtt{A} : m_2\langle U\rangle,\ l_2 : \mathtt{C} \to \mathtt{A} : m_2\langle U'\rangle\}$ is not well-formed due to projection but we can model the branching behaviour by using multicast on labels as $\mathtt{A} \to \{\mathtt{B},\mathtt{C}\} : \{m_1, m_3\}\{l_1 : \mathtt{B} \to \mathtt{A} : m_2\langle U\rangle,\ l_2 : \mathtt{C} \to \mathtt{A} : m_2\langle U'\rangle\}$.
- **Higher Order Communication**. High-order communication is defined as $k!\langle k'\rangle \mid k?\langle k'\rangle$ in the first systems [9,10], where the receiver posseses the transmitted channel (k') before the communication takes place. The calculus of this paper models the transmission of channels with the receiver not possessing the channel until the communication happens. This feature is modeled safely by adding multipolarity labels to session channels as in [8,19].
- **Delegation**. Higher-order communication models the capability of a process to delegate its session participation to another one. Global types define the interactions only between the participants of a session. In the asynchronous multiparty calculus, the global types of the Alice-Bob-Carol example (Section 4.4)[10] $G_a = \mathtt{A} \to \mathtt{B}\!:\ t_1\langle s_1!\langle int\rangle; \mathtt{end@B}\rangle.\mathtt{end}$ and $G_b = \mathtt{B} \to \mathtt{C}\!:\ s_1!\langle int\rangle.\mathtt{end}$ do not satisfy the said definition. The session at b is started between $\mathtt{A}$ and $\mathtt{C}$, and the global

type G_b should define the interaction between these two participant, $\mathtt{A} \rightarrow \mathtt{C}$, and not between the participants that may be involved to send the message due to delegation at runtime, $\mathtt{B} \rightarrow \mathtt{C}$. This inconsistency of information between G_b and the implementation of session initiated on b makes process Alice not type-checked even though it is correct. Following the above definition, the sequent global types $G_a = \mathtt{A} \rightarrow \mathtt{B}\colon\ t_1\langle s_1!\langle int\rangle; \mathtt{end@A}\rangle$ and $G_b = \mathtt{A} \rightarrow \mathtt{C}\colon\ s_1!\langle int\rangle.\mathtt{end}$ type-check all three processes.

Organization. In the remainder of this paper, Section 2 defines the syntax and operational semantics of the calculus. Section 3 defines the syntax of global types and introduces the linearity property. Section 4 gives the programming methodology, syntax of local types, projection algorithm and type system. Section 5 concludes by comparing the system with related works and gives possible future work. Appendixes A gives the full proofs of the theorems presented in the paper.

2 A Synchronous Multiparty-Session Calculus

The syntax and the operational semantics of the multiparty-session synchronous calculus are basically the ones of binary session calculus [9] extended with construct and operational semantic rule for session initiation and multicasting. Throughout the paper we will refer to the calculus as the MS-calculus.

2.1 Syntax

The syntax of the MS-calculus is a more relaxed version of the one introduced by Honda et al. [9]. The calculus does not distinguish between a primitive value send and a session channel send following the idea firstly proposed in [8]. The syntax of the calculus does not introduce queues, neither at the programmer code level nor at the runtime code level. The multiparty-session request is represented in the same way as in the asynchronous calculus. The MS-calculus supports higher-order communications by introducing multipolarity labels [19,8] to session channels and multicasting by sending messages to a group of peers simultaneously.

The next part of this section will introduce some of the definitions and notations used in the formal definition of the calculus.

Definition 2.1 *a, b, c, ...* represent shared names*; e, e', ...* represent expressions*;* l, l_1, l_2*, ...* refer to labels*;* $\mathtt{p}$*,* $\mathtt{q}$*,* $\mathtt{n}$*, r, ...* range over naturals*;* $i, j, ...$ denote *indexes over a set of naturals;* $\kappa_{\mathtt{p}}$*, ...* refer to session channels*;* $x, y, z, ...$ refer to variables of the calculus*; X, Y, ...* refer to process variables and $P, P_1, P_2, ...Q$*, ...* refer to processes.

Notation 2.2 The notation $\tilde{k}$ denotes a list of channels $k_1, ..., k_n$.

Figure 1 introduces the abstract grammar of the calculus syntax. The terms of the calculus represent a range of processes from simple inactive one to processes that implement complex communication behaviours. The paragraphs below will

P	$::= \overline{a}_{[2..\mathtt{n}]}\,(\tilde{x}).P$	multicast session request
	$\mid a_{[\mathtt{p}]}\,(\tilde{x}).P$	session acceptance
	$\mid \tilde{k}!\langle\tilde{e}\rangle; P$	value sending
	$\mid k?(\tilde{x}); P$	value reception
	$\mid \tilde{k} \lhd l; P$	label selection
	$\mid k \rhd \{l_i\colon P_i\}_{i\in I}$	label branching
	$\mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	conditional branch
	$\mid P \mid Q$	parallel composition
	$\mid \mathbf{0}$	inaction
	$\mid (\nu n)P$	hiding
	$\mid \mathtt{def}\ D\ \mathtt{in}\ P$	recursion
	$\mid X\langle\tilde{e}\rangle$	process call
e	$::= x \mid v \mid e\ \mathtt{and}\ e' \mid \mathtt{not}\ e\ \ \ldots$	expressions
v	$::= a \mid \mathtt{true} \mid \mathtt{false} \mid k$	values
k	$::= x \mid \kappa_{\mathtt{p}}$	session channels variables and values
$\mathtt{p}$	$::= \mathtt{1} \mid \ldots \mid \mathtt{n}$	channels multipolarity
D	$::= X_1(\tilde{y}_1) = P_1\ \mathtt{and}\cdots\mathtt{and}\ X_n(\tilde{y}_n) = P_n$	declaration for recursion

Fig. 1. Syntax

give an informal description of the constructs introduced in the figure.

A session is established among peers via shared names, which represent public points of communication. In the calculus a session initiation would be

$$\overline{a}[2,3](y_1, y_2, y_3).P_1 \mid a[2](y_1, y_2, y_3).P_2 \mid a[3](y_1, y_2, y_3).P_3$$

where a represents the shared name. The process with over-lined a represents the process willing to initiate a session with participant numbered two and three and the others represent processes waiting to initiate a session. The set of bound variables $\{y_1, y_2, y_3\}$ represent placeholders for session channels which will be generated at runtime.

Sending of values is defined by the channel (channels in case of multicasting) and the values to send; receiving of values is defined by the channel and the placeholders of the values to receive; selecting a label is defined by the channel (channels in case of multicasting) and the label to send; branching labels is defined by the channel and the set of labels which contains the label to receive.

The conditional branch and parallel composition have the same definition as in other process calculi. **0** represents the process that cannot do any action (inaction). The hiding operation on n has the standard definition of restricting or generating new session channels ($\kappa_{\mathtt{p}}$) or shared names (a). The recursion and process call constructs define recursion in the calculus; the recursion construct defines terms with a recursive behaviour and the process call construct invokes that behaviour.

The values sent among peers can be session channels ($\kappa_{\mathtt{p}}$) and other primitive values as booleans, strings, natural, etc.

The association of "|" is the weakest over all operators ($\boldsymbol{\nu}$, $\mathtt{def}\ D\ \mathtt{in}\ P$). Below, we define free names (fn) and free process variables (fpv) on $\mathtt{MS}$-terms:

$$\begin{aligned}
\mathrm{fn}(\overline{a}_{[2..\mathtt{n}]}\,(\tilde{x}).P) &\triangleq \{a\} \bigcup \mathrm{fn}(P)\\
\mathrm{fn}(a_{[\mathtt{p}]}\,(\tilde{x}).P) &\triangleq \{a\} \bigcup \mathrm{fn}(P)\\
\mathrm{fn}((\nu n)P) &\triangleq \mathrm{fn}(P) \setminus \{n\}\\
\mathrm{fpv}(\mathtt{def}\ D\ \mathtt{in}\ P) &\triangleq \mathrm{fpv}(P)\setminus\mathrm{dpv}(D)\\
\mathrm{fpv}(X\langle\tilde{e}\rangle) &\triangleq \{X\}
\end{aligned}$$

where dpv(D) represents the set of process variables $\{X_i\}_{i\in I}$ introduced in $X_1(\tilde{y}_1) = P_1\ \mathtt{and} \cdots \mathtt{and}\ X_n(\tilde{y}_n) = P_n$.

2.2 Operational Semantics

The two communication-based operations on session channels are value sending-receiving and label selection-branching. Multicast session request-acceptance represents a communication idiom that is used only at session initiation.

Structural congruence $\equiv$ is the smallest congruence on processes that satisfies the axioms showed in Figure 2.

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\nu n)P \mid Q \equiv (\nu n)(P \mid Q) \quad \text{ if } n \notin \mathrm{fn}(Q)$$

$$(\nu n n')P \equiv (\nu n' n)P \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \qquad \mathtt{def}\ D\ \mathtt{in}\ \mathbf{0} \equiv \mathbf{0}$$

$$\mathtt{def}\ D\ \mathtt{in}\ (\nu n)P \equiv (\nu n)\mathtt{def}\ D\ \mathtt{in}\ P \quad \text{ if } n \notin \mathrm{fn}(D)$$

$$(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q \equiv \mathtt{def}\ D\ \mathtt{in}\ (P \mid Q) \quad \text{ if } \mathrm{dpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$$

$$\mathtt{def}\ D\ \mathtt{in}\ (\mathtt{def}\ D'\ \mathtt{in}\ P) \equiv\ \mathtt{def}\ D\ \mathtt{and}\ D'\ \mathtt{in}\ P \quad \text{ if } \mathrm{dpv}(D) \cap \mathrm{dpv}(D') = \emptyset$$

Fig. 2. Structural congruence.

The operational semantics of the calculus is given via the reduction relation $\rightarrow$ where the state of the machine is defined by only $\mathtt{MS}$-terms. Figure 3 defines the rules of a small step operational semantics of the calculus. The paragraphs below give a description of the rules.

Notation 2.3 $\kappa[m_1, ..., m_r]_{\mathtt{p}}$ *denotes* $\kappa_{m_1\mathtt{p}}, ..., \kappa_{m_r\mathtt{p}}$.

[Link] initiates a session between n peers. The result of the reduction is generation of session channels and substitution of them in processes. Note that session channels for each process are labeled by a multipolarity index ranging [1, ..., n], where n is the number of participants involved in a session. The calculus uses multipolarity session channels to support higher-order communications safely (see Section 2.4). An example of session initiation reduction is:

$$\overline{a}[2,3](y_1,y_2,y_3).y_1!\langle 5\rangle; P_1 \mid a[2](y_1,y_2,y_3).y_1?(y_4); P_2 \mid a[3](y_1,y_2,y_3).y_2!\langle \text{"}blue\text{"}\rangle; P_3 \\ \rightarrow\ (\nu\kappa_1,\kappa_2,\kappa_3)(\kappa_{11}!\langle 5\rangle; P_1[\kappa_{11},\kappa_{21},\kappa_{31}/y_1,y_2,y_3] \mid \\ \kappa_{12}?(y_4); P_2[\kappa_{12},\kappa_{22},\kappa_{32}/y_1,y_2,y_3] \mid \kappa_{23}!\langle \text{"}blue\text{"}\rangle; P_3[\kappa_{13},\kappa_{23},\kappa_{33}/y_1,y_2,y_3]).$$

[Multicasting] actions the value sending-receiving communication between two and more peers. The result of the communication is the substitution of the place holders with the received values by the receivers. Note that the reduction holds if the channels are the same in both peers despite the polarity is different. The relation $\downarrow$ evaluates the expression e to the value v and the value v to itself. An example of value communication is:

$$\kappa_{11}!\langle 5\rangle; P_1' \mid \kappa_{12}?(y_4); P_2' \ \rightarrow\ P_1' \mid P_2[5/y_4].$$

[MultiLabel] actions the selection-branching communication between two and more participants. The selection process sends the label l_i to the branching processes and the result of the communication is the resting part of the label selection process (P_1) in parallel with the process labeled by l_i $(P_{[2..k]i})$.

[If1] and [If2] action the evaluation of e; if e evaluates to *true* then rule [If1] is applied otherwise rule [If2].

[Def] invokes the behaviour (P) identified by X with values for arguments $\tilde{v}$ in the context.

[Scop] actions the reduction of the process inside the scope of the ν operator.

$$(\nu\kappa_1,\kappa_2,\kappa_3)(\kappa_{11}!\langle 5\rangle; P_1' \mid \kappa_{12}?(y_4); P_2' \mid \kappa_{23}!\langle \text{"}blue\text{"}\rangle; P_3') \\ \rightarrow\ (\nu\kappa_1,\kappa_2,\kappa_3)(P_1' \mid P_2'[5/y_4] \mid \kappa_{23}!\langle \text{"}blue\text{"}\rangle; P_3').$$

[Par] states that if one process (P) evolves to another process (P') then the parallel composition process $(P \mid Q)$ can evolve to another parallel composition process with the evolved component process $(P' \mid Q)$; i.e.

$$\kappa_{11}!\langle 5\rangle; P_1' \mid \kappa_{12}?(y_4); P_2' \mid P_3' \ \rightarrow\ P_1' \mid P_2'[5/y_4] \mid P_3'.$$

[Defin] states that if process P can evolve to a process P' then the entire recursive term can evolve to a new recursive term.

[Str] states that the reduction relation is defined on structural congruent terms.

2.3 Examples

Addition Protocol. The program below implements the session between `Client-Addition-Successor-Predecessor` introduced in Section 1. The process *addition* implements the communication pattern of adding two natural numbers; *successor* and *predecessor* processes implement the communication pattern of receiving a number and sending its successor and predecessor, respectively.

- *client* $\triangleq a[2,3,4]\,(x_1,x_2,x_3).x_1!\langle 5\rangle; x_1!\langle 4\rangle; x_1?(y_1); 0$
- *addition* $\triangleq$ `def` $X_1\langle y_1,y_2,x_1,x_2,x_3\rangle =$

$$\overline{a}_{[2..\mathtt{n}]}\,(\tilde{y}).P_1 \mid a_{[2]}\,(\tilde{y}).P_2 \mid \cdots \mid a_{[\mathtt{n}]}\,(\tilde{y}).P_n \;\rightarrow\; (\nu\tilde{\kappa})(P_1[\tilde{\kappa_1}/\tilde{y}] \mid P_2[\tilde{\kappa_2}/\tilde{y}] \mid \ldots \mid P_{\mathtt{n}}[\tilde{\kappa_{\mathtt{n}}}/\tilde{y}]) \qquad [\textsc{Link}]$$

$$\kappa[m_1,...,m_r]_{\mathtt{p_1}}!\langle\tilde{e}\rangle;P_1 \mid \kappa[m_1]_{\mathtt{p_2}}?(\tilde{y});P_2 \mid \cdots \mid \kappa[m_r]_{\mathtt{p_{r+1}}}?(\tilde{y});P_{r+1}$$
$$\rightarrow\; P_1 \mid P_2[\tilde{v}/\tilde{y}] \mid \cdots \mid P_{r+1}[\tilde{v}/\tilde{y}] \quad (\mathtt{p_1} \neq \mathtt{p_2} \neq \cdots \neq \mathtt{p_{r+1}}, \tilde{e}\downarrow\tilde{v}) \qquad [\textsc{Multicasting}]$$

$$\kappa[m_1,...,m_r]_{\mathtt{p_1}} \lhd l_i;P_1 \mid \kappa[m_1]_{\mathtt{p_2}} \rhd \{l_j\colon P_{2j}\}_{j\in I} \mid \cdots \mid \kappa[m_r]_{\mathtt{p_{r+1}}} \rhd \{l_j\colon P_{r+1j}\}_{j\in I}$$
$$\rightarrow\; P_1 \mid P_{2i} \mid \cdots \mid P_{r+1i} \qquad (\mathtt{p_1} \neq \mathtt{p_2} \neq \cdots \neq \mathtt{p_{r+1}}, i \in I) \qquad [\textsc{MultiLabel}]$$

$$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \;\rightarrow\; P \qquad (e\downarrow\mathtt{true}) \qquad [\textsc{If1}]$$

$$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \;\rightarrow\; Q \qquad (e\downarrow\mathtt{false}) \qquad [\textsc{If2}]$$

$$\mathtt{def}\ D\ \mathtt{in}\ (X\langle\tilde{e}\rangle \mid Q) \;\rightarrow\; \mathtt{def}\ D\ \mathtt{in}\ (P[\tilde{v}/\tilde{y}] \mid Q) \qquad (\tilde{e}\downarrow\tilde{v}, X(\tilde{y}) = P \in D) \qquad [\textsc{Def}]$$

$$P \rightarrow P' \;\Rightarrow\; (\nu n)P \rightarrow (\nu n)P' \qquad [\textsc{Scop}]$$

$$P \rightarrow P' \;\Rightarrow\; P \mid Q \rightarrow P' \mid Q \qquad [\textsc{Par}]$$

$$P \rightarrow P' \;\Rightarrow\; \mathtt{def}\ D\ \mathtt{in}\ P \rightarrow \mathtt{def}\ D\ \mathtt{in}\ P' \qquad [\textsc{Defin}]$$

$$P \equiv P' \text{ and } P' \rightarrow Q' \text{ and } Q' \equiv Q \;\Rightarrow\; P \rightarrow Q \qquad [\textsc{Str}]$$

Fig. 3. Operational Semantics

$$\begin{aligned}&\mathtt{if}\ (y_2 = 0)\ \mathtt{then}\ x_2, x_3 \lhd\ \mathit{true}; x_1!\langle y_1\rangle; P\\ &\mathtt{else}\ x_2, x_3 \lhd\ \mathit{false}; x_2!\langle y_1\rangle; x_2?(y_1); x_3!\langle y_2\rangle; x_3?(y_2);\\ &\qquad X_1\langle y_1, y_2, x_1, x_2, x_3\rangle\\ \mathtt{in}\ &a_{[2]}\,(x_1, x_2, x_3).x_1?(y_1); x_1?(y_2); X_1\langle y_1, y_2, x_1, x_2, x_3\rangle\end{aligned}$$

- $\mathit{successor} \triangleq \mathtt{def}\ X_2\langle x_2\rangle = x_2 \rhd \{\mathit{true}\colon \mathbf{0}, \mathit{false}\colon x_2?(y_1); x_2!\langle y_1 + 1\rangle; X_2\langle x_2\rangle\}$ $\mathtt{in}\ a_{[3]}\,(x_1, x_2, x_3).X_2\langle x_2\rangle$
- $\mathit{predecessor} \triangleq \mathtt{def}\ X_3\langle x_3\rangle = x_3 \rhd \{\mathit{true}\colon \mathbf{0}, \mathit{false}\colon x_3?(y_1); x_2!\langle y_1 - 1\rangle; X_3\langle x_3\rangle\}$ $\mathtt{in}\ a_{[4]}\,(x_1, x_2, x_3).X_3\langle x_3\rangle$

Processes *successor* and *predecessor* use two different channels to communicate with *addition*. One receive for the first operand ($x_2?(y_1)$ in *successor*) and another for the second operand ($x_3?(y_1)$ in *predecessor*) are both from *addition*. Hence there is no guarantee that the receptions are in a fixed order, even though the deliveries are ordered. Thus if we were to use x_2 for both actions, the message of first operand can be received by *predecessor*. The problem becomes visible after the fifth step of the below reduction. Later we shall show our type discipline can detect such an error. Processes Q, R and S are equal by definition to:

$$Q \triangleq \texttt{if } (y_2 = 0) \texttt{ then } x_2, x_3 \lhd \ true; x_1!\langle y_1\rangle; P$$
$$\texttt{else } x_2, x_3 \lhd \ false; x_2!\langle y_1\rangle; x_2?(y_1); x_3!\langle y_2\rangle; x_3?(y_2); X_1\langle y_1, y_2, x_1, x_2, x_3\rangle$$
$$R \triangleq x_2 \rhd \{true\colon \mathbf{0}, false\colon x_2?(y_1); x_2!\langle y_1+1\rangle; X_2\langle x_2\rangle\}$$
$$S \triangleq x_3 \rhd \{true\colon \mathbf{0}, false\colon x_3?(y_1); x_2!\langle y_1-1\rangle; X_3\langle x_3\rangle\}$$

then the first reduction steps of the parallel composition of the above processes are:

$$client \mid addition \mid successor \mid predecessor \ \longrightarrow^{[\textsc{Str}],[\textsc{Link}]}$$

$$\texttt{def } X_1\langle y_1, y_2, x_1, x_2, x_3\rangle = Q, X_2\langle x_2\rangle = R, X_3\langle x_3\rangle = S \texttt{ in}$$
$$(\nu\kappa_1, \kappa_2, \kappa_3)\ (\kappa_{11}!\langle 5\rangle; \kappa_{11}!\langle 4\rangle; \kappa_{11}?(y_1); 0$$
$$\mid \kappa_{12}?(y_1); \kappa_{12}?(y_2); X_1\langle y_1, y_2, \kappa_{12}, \kappa_{22}, \kappa_{32}\rangle$$
$$\mid X_2\langle \kappa_{23}\rangle$$
$$\mid X_3\langle \kappa_{34}\rangle) \ \longrightarrow^{[\textsc{Multicasting}],[\textsc{Multicasting}]}$$

$$\texttt{def } X_1\langle y_1, y_2, x_1, x_2, x_3\rangle = Q, X_2\langle x_2\rangle = R, X_3\langle x_3\rangle = S \texttt{ in}$$
$$(\nu\kappa_1, \kappa_2, \kappa_3)\ (\kappa_{11}?(y_1); 0$$
$$\mid X_1\langle 5, 4, \kappa_{12}, \kappa_{22}, \kappa_{32}\rangle$$
$$\mid X_2\langle \kappa_{23}\rangle$$
$$\mid X_3\langle \kappa_{34}\rangle) \ \longrightarrow^{[\textsc{Def}]}$$

$$\texttt{def } X_1\langle y_1, y_2, x_1, x_2, x_3\rangle = Q, X_2\langle x_2\rangle = R, X_3\langle x_3\rangle = S \texttt{ in}$$
$$(\nu\kappa_1, \kappa_2, \kappa_3)\ (\kappa_{11}?(y_1); 0$$
$$\mid \texttt{if } (4 = 0) \texttt{ then } \kappa_{22}, \kappa_{32} \lhd \ true; \kappa_{12}!\langle y_1\rangle; P$$
$$\texttt{else } \kappa_{22}, \kappa_{32} \lhd \ false; \kappa_{22}!\langle 5\rangle; \kappa_{22}?(y_1); \kappa_{32}!\langle 4\rangle; \kappa_{32}?(y_2);$$
$$X_1\langle y_1, y_2, \kappa_{12}, \kappa_{22}, \kappa_{32}\rangle$$
$$\mid \kappa_{23} \rhd \{true\colon \mathbf{0}, false\colon \kappa_{23}?(y_1); \kappa_{23}!\langle y_1+1\rangle; X_2\langle \kappa_{23}\rangle\}$$
$$\mid \kappa_{34} \rhd \{true\colon \mathbf{0},$$
$$false\colon \kappa_{34}?(y_1); \kappa_{34}!\langle y_1-1\rangle; X_2\langle \kappa_{34}\rangle\}) \ \longrightarrow^{[\textsc{If2}],[\textsc{MultiLabel}]}$$

$$\texttt{def } X_1\langle y_1, y_2, x_1, x_2, x_3\rangle = Q, X_2\langle x_2\rangle = R, X_3\langle x_3\rangle = S \texttt{ in}$$

$$\begin{aligned}(\nu\kappa_1,\kappa_2,\kappa_3)\ &(\kappa_{11}?(y_1);0\\ &\mid \kappa_{22}!\langle 5\rangle;\kappa_{22}?(y_1);\kappa_{32}!\langle 4\rangle;\kappa_{32}?(y_2);X_1\langle y_1,y_2,\kappa_{12},\kappa_{22},\kappa_{32}\rangle\\ &\mid \kappa_{23}?(y_1);\kappa_{23}!\langle y_1+1\rangle;X_2\langle\kappa_{23}\rangle\\ &\mid \kappa_{34}?(y_1);\kappa_{34}!\langle y_1-1\rangle;X_2\langle\kappa_{34}\rangle)\ \rightarrow^{[\text{MULTICASTING}]}\ldots\end{aligned}$$

Fire Alarm Protocol. The fire alarm protocol given in Section 1 is a simple representation of the communication pattern of a fire alarm system. Only two of the main components are considered: notification appliances and building safety interfaces. The implementation of the fire alarm protocol in the MS-calculus is:

$$controller \triangleq \overline{a}[\mathtt{2,3,...,j+k}](x_1,x_2,...,x_{j-1},y_j,...,y_{j+k-1}).x_1,...,x_{j-1}!\langle\text{“}ON\text{”}\rangle;$$
$$y_j,...,y_{j+k-1}!\langle\text{“}BLOCK\text{”}\rangle;P$$

$$firealarm_1 \triangleq a[\mathtt{2}](x_1,x_2,...,x_{j-1},y_j,...,y_{j+k-1}).x_1?(x);P_1$$

$$\ldots$$

$$firealarm_{j-1} \triangleq a[\mathtt{j}](x_1,x_2,...,x_{j-1},y_j,...,y_{j+k-1}).x_{j-1}?(x);P_{j-1}$$

$$elevator_j \triangleq a[\mathtt{j+1}](x_1,x_2,...,x_{j-1},y_j,...,y_{j+k-1}).y_j?(x);Q_1$$

$$\ldots$$

$$elevator_{j+k-1} \triangleq a[\mathtt{j+k}](x_1,x_2,...,x_{j-1},y_j,...,y_{j+k-1}).y_{j+k-1}?(x);Q_{k-1}$$

First, the controller sends in multicast an *ON* message to the fire alarms to notify the persons in the building and then sends a *BLOCK* message to the elevators to safely lead the persons towards safety exits. Due to the synchronous nature of communications in the MS-calculus, the second send will take place only after the first message has been received by all the fire alarms; the implementation follows correctly the timing specification of the events in a fire alarm system.

2.4 Higher-order Communications

The system developed by Honda et al. [9,10] does not define the term

$$\mathtt{throw}\ k[k'];P_1 \mid \mathtt{catch}\ k(k'')\,^{3}\ \mathtt{in}\ P_2 \not\rightarrow$$

semantically correct. In order to reduce, the receiver should possess the channel k' before the communication take place. It would be nice if a system could allow the transmission of channels with the receiver not possessing the channel before the communication. Yoshida and Vasconcelos [19] describe different extensions to the operational semantics and analyze soundness of the type system with respect to the operational semantics. The first solution is to rename the bound channel k'' into k' but that might bind the free occurrences of k' in P_2. Another solution could be to change the operational semantic rule in

[3] The terms throw $k[k']$ and catch $k(k'')$ translate $k!\langle k'\rangle$ and $k?(k')$ in the syntax of the MS-calculus.

$$\texttt{throw } k[k'];P_1 \mid \texttt{catch } k(k'') \texttt{ in } P_2 \;\rightarrow\; P_1 \mid P_2[k'/k''].$$

This rule breaks soundness of the type system. Indeed the process

$$\texttt{accept } b(k') \texttt{ accept } a(k) \texttt{ in throw } k[k'] \mid \texttt{request } b(k') \texttt{ request } a(k) \texttt{ in catch}$$
$$k(k'') \texttt{ in } k''?(y) \texttt{ in } k'![1]$$

is well typed by [9] type system but the derived term

$$k'?(y) \texttt{ in } k'![1]$$

is not well-typed under the same type system because in the derived term the type of k' involves one read and one write rather than only one write as it was in the starting definition of the second process. Even though, this example might be controversial if it is useful or not in practice, it is a well formed term of the calculus that breaks soundness of the type system. The solution proposed by Yoshida and Vasconcelos defines channels as runtime entities; i.e. they are not part of the syntax used by programmers and are generated at initiation time, as in the calculus introduced in this paper. The above example written in their system is

$$\texttt{accept } b(y_1) \texttt{ accept } a(y_2) \texttt{ in throw } y_2[y_1] \mid \texttt{request } b(y_1) \texttt{ request } a(y_2) \texttt{ in}$$
$$\texttt{catch } y_2(y_3) \texttt{ in } y_3?(y_4) \texttt{ in } y_1![1].$$

Session channels are labeled by a polarity sign (+, -) when substituted in each process at session initiation time. By convention, the polarity label - is assigned to a channel that is substituted in the process that is requesting to establish a session and + to the process that is waiting to establish a session. The polarity label is syntax added to channels in order to extend their definition as a communication abstraction entity. In other words, a channel is not only an entity that belongs to a communication but also that belongs to one of the two endpoint processes. The reduced term of the above process

$$\kappa'^{+}?(y) \texttt{ in } \kappa'^{-}![1]$$

is well-typed as k' of one endpoint differs by a polarity sign from the other. SJ [12], an implementation of binary session types in Java, rejects at runtime the above example. Indeed, this term stucks in a non final value state and fails progress of the system.

The system given in this paper uses the same logic of [19,8] to represent channels. The only change is that of having multipolarity. That is, their system used a binary polarity (+, -) because in binary sessions only two processes are involved, but for the multiparty calculus the number processes participating in a session is generally more than two, thus we introduce an index label ranging [`1`, ..., `n`], where `n` is the number of processes involved in a session. As it can be noticed in the operational semantics rule, a polarity label is assigned to every channel of the session when substituted in a process.

Higher-order communication models the capability of a process to delegate its session participation to another one. The example below gives the implementation of a session on a where participants `1` and `2` send author and title of a book to participant `3`. Participant `1` delegates its part of the conversation to another participant, implemented by process D, by sending all the channels of the session. Participant $\mathtt{1}, \mathtt{2}$ and $\mathtt{3}$ are implemented by processes A, B and C, respectively.

- $A \triangleq \overline{a}[\mathtt{2},\mathtt{3}]\,(x_1, x_2).\bar{b}[\mathtt{2}]\,(y_1).y_1!\langle x_1, x_2\rangle;$
- $B \triangleq a[\mathtt{2}]\,(x_1, x_2).x_2!\langle$"*The computer and the brain*"$\rangle;$
- $C \triangleq a[\mathtt{3}]\,(x_1, x_2).x_1?(y'); x_2?(y'');$
- $D \triangleq b[\mathtt{2}]\,(y_1).y_1?(y_2, y_3); y_2!\langle$"*John von Neumann*"$\rangle;$

3 Global Session Types and Causality Analysis

Programming multiparty sessions without errors requires a lot of programming effort to define all the communications dependencies between all the participants and avoid race conditions on channels. As illustrated by the addition protocol in Section 1, binary session types can not capture all the interaction dependencies, thus a notion of global type is introduced in [10]. However, global types do not guarantee programs from having broken causalities introduced by a race condition on channels; a linearity property checks global types for the presence of this condition. Global types and causalities will be discussed in this section.

3.1 Syntax

The syntax of global session types or global types as we will refer to them through the paper, is presented in Figure 4. The constructors to build global session types for the MS-calculus are those of [10] extended with multicast send of values and labels. Type $\mathtt{p} \rightarrow \{\mathtt{p}_1, ..., \mathtt{p}_r\} \colon \{m_1, ..., m_r\}\ \langle\tilde{S}\rangle.G'$ represents all sessions, where participant $\mathtt{p}$ sends a message of type $\tilde{S}$ to all participants $\{\mathtt{p}_1, ..., \mathtt{p}_r\}$ through channels indexed $m_1, ..., m_r$ where $\forall i, j \in \{1, ..., r\}$ s.t. $i \neq j$ then $m_i \neq m_j$, and that the rest of session is represented by G'. The calculus does not support global types that have multicasting delegation (see Section 5). Type $\mathtt{p} \rightarrow \{\mathtt{p}_1, ..., \mathtt{p}_r\} \colon \{m_1, ..., m_r\}\ \{l_h \colon G_h\}_{h \in J}$ represents all sessions, where participant $\mathtt{p}$ selects and sends to all participants $\{\mathtt{p}_1, ..., \mathtt{p}_r\}$ one of the J labels through channels indexed $m_1, ..., m_r$ where $\forall i, j \in \{1, ..., r\}$ s.t. $i \neq j$ then $m_i \neq m_j$ and that the rest of the session is represented by G'. We abbreviate to $\mathtt{p} \rightarrow \mathtt{p}' : m$ when there is a single receiver.

Type G, G' represents all sessions where parts of them, in this case represented by global types G and G', run in parallel. Type $\mu\mathbf{t}.G$ represents all sessions that define a recursive behaviour on G. Type $\mathtt{end}$ represents the empty session and is used as a base type to build more complex global session types.

Type U represents the types of values sent among participants, such as booleans, naturals, strings, channels or names. The type $\langle G\rangle$ is a set $\langle T@\mathtt{p}_1, ..., T@\mathtt{p}_r\rangle$ where $T@\mathtt{p}$ (see Section 4.2) is an end-point type for participant $\mathtt{p}$, and is used to type shared names.

$$
\begin{array}{rlr}
G & ::= \mathsf{p} \to \{\mathtt{p_1},...,\mathtt{p_r}\} \colon \{m_1,...,m_r\}\ \langle \tilde{S} \rangle.G' & \text{values} \\
 & \mid \mathsf{p} \to \mathsf{p}'' \colon m\ \langle T@_{\mathtt{q}} \rangle.G' & \text{values} \\
 & \mid \mathsf{p} \to \{\mathtt{p_1},...,\mathtt{p_r}\} \colon \{m_1,...,m_r\}\ \{l_h \colon G_h\}_{h\in J} & \text{branching} \\
 & \mid G, G' & \text{parallel} \\
 & \mid \mu\mathbf{t}.G & \text{recursion} \\
 & \mid \mathbf{t} & \text{variable} \\
 & \mid \mathsf{end} & \text{end} \\
S & ::= \mathsf{bool} \mid \mathsf{nat} \mid ... \mid \langle G \rangle & \text{Sort} \\
m & ::= 1 \mid 2 \mid \cdots &
\end{array}
$$

Fig. 4. Syntax of Global Session Types

3.2 Prefix Ordering

The definitions below formally define the ordering of communications on a global type. The ordering relation will be later used to define the linearity property.

Definition 3.1 (prefix) We say the initials "$\mathsf{p} \to \mathtt{p_i} : m_i$" for all $i \in \{1..r\}$ in $\mathsf{p} \to \{\mathtt{p_1},...,\mathtt{p_r}\} \colon \{m_1,...,m_r\}\langle U \rangle.G'$ and $\mathsf{p} \to \{\mathtt{p_1},...,\mathtt{p_r}\} \colon \{m_1,...,m_r\}\{l_h \colon G_h\}_{h\in J}$ are called *prefixes from* p *to* $\mathtt{p_i}$ *at* m_i *over* G' and $\{G_h\}_{h\in J}$, where in the former U is called a *carried type*. If U is a carried type in a prefix in G then U is also a carried type in G.

Conventions 1 We assume that in each prefix from p to p' we have $\mathsf{p} \neq \mathsf{p}'$, i.e. we prohibit reflexive interaction.

Definition 3.2 (prefix ordering) Write $\mathsf{n}, \mathsf{n}', ..$ for prefixes occurring in a global type, say G (but not in its carried types), seen as nodes of G as a graph. We write $\mathsf{n} \in G$ when n occurs in G. Then we write $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$ when n_1 directly or indirectly prefixes n_2 in G. Formally $\prec$ is the least partial order generated by:

$$
\mathsf{n}_i \prec \mathsf{n}_{r+1} \in \mathsf{p} \to \mathtt{p_1},...,\mathtt{p_r} \colon m_1,...,m_r\ \langle U \rangle.G' \text{ if } \mathsf{n}_i = \mathsf{p} \to \mathtt{p_i} : m_i, \mathsf{n}_{r+1} \in G' \quad i \in \{1,...,r\}
$$

$$
\mathsf{n}_i \prec \mathsf{n}_{r+1} \in \mathsf{p} \to \mathtt{p_1},...,\mathtt{p_r} \colon m_1,...,m_r\ \{l_h \colon G_h\}_{h\in J} \text{ if } \mathsf{n}_i = \mathsf{p} \to \mathtt{p_i} : m_i, \quad \exists h \in J.\, \mathsf{n}_{r+1} \in G_h, i \in \{1,...,r\}
$$

Further we set $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$ if $\mathsf{n}_1 \prec \mathsf{n}_2 \in G'$ and G' occurs in G but not in its carried types.

Consider a global type:

$$
\mathtt{A} \to \mathtt{B} \colon m_1\ \langle U \rangle.\mathtt{A} \to \mathtt{C} \colon m_2\ \langle U' \rangle.\mathsf{end} \tag{2}
$$

The two prefixes are ordered by $\prec$, $\mathtt{A} \to \mathtt{B} : m_1 \prec \mathtt{A} \to \mathtt{C} : m_2$. This ordering means "only after the first sending and receiving take place, the second sending and receiving take place". It is modeled for calculi of synchronous communications but not for asynchronous ones; e.g. in the asynchronous calculus [10] `C` may receive its message before `B`.

3.3 Causality Analysis

Section 2.3 discussed why the causalities between `Successor-Addition` and `Predecessor-Addition` can be broken if it is used the same channel in this two communications. It would be nice if we could statically check programs from race conditions on channels. Global types provide a global representation of a session's causalities and channels used. The global type of the addition protocol is:

$$
\begin{array}{ll}
1 & \texttt{Client} \to \texttt{Addition}\colon\ 1\langle int\rangle.\\
2 & \texttt{Client} \to \texttt{Addition}\colon\ 1\langle int\rangle.\\
3 & \mu\mathbf{t}.\texttt{Addition} \to \{\texttt{Successor},\texttt{Predecessor}\}\colon 2,3\{\\
4 & \qquad\qquad true : \texttt{Addition} \to \texttt{Client}\colon\ 1\langle int\rangle.\mathsf{end},\\
5 & \qquad\qquad false : \texttt{Addition} \to \texttt{Successor}\colon\ 2\langle int\rangle.\\
6 & \qquad\qquad\qquad\quad \texttt{Successor} \to \texttt{Addition}\colon\ 2\langle int\rangle.\\
7 & \qquad\qquad\qquad\quad \texttt{Addition} \to \texttt{Predecessor}\colon\ 3\langle int\rangle.\\
8 & \qquad\qquad\qquad\quad \texttt{Predecessor} \to \texttt{Addition}\colon\ 3\langle int\rangle.\mathbf{t}\}
\end{array}
$$

Even though $\texttt{Addition} \to \texttt{Successor} \prec \texttt{Addition} \to \texttt{Predecessor}$ the receptions are not ordered so if a same channel is used in both communications then the causalities can be broken.

Figure 5 presents all the possible scenarios of ordering two consecutive communications without breaking the causalities. The letters A and S represent respectively the asynchronous and synchronous calculus where the cases are considered. All the six cases are considered for ordering in the synchronous `MS`-calculus unlike the asynchronous one [10] where the output-input (OI) and output-output (OO) are not considered. The output-input case is not consider in [10] because the reception of the message from $\mathtt{P}_2$ can occur before that the message sent is received by $\mathtt{P}_1$. The situation is the same for the output-output case, the second message sent can be received before the first one. If channels are the same, in the (II) case the order of messages can break, in the (IO) case the message sent by participant $\mathtt{P}_1$ can be received by participant $\mathtt{P}_2$ breaking therefore causalities, in the case (OI) the message sent by participant $\mathtt{P}_2$ can be received by participant $\mathtt{P}_1$ as in (IO) and in the (OO) case the order of messages can break as in (II). The break of messages order turns to be as harmful as a broken causality when the values of messages sent are different.

The above observations lead to causalities order on global types.

Definition 3.3 (dependency relations) Fix G. The relation $\prec_\phi$, with $\phi \in \{\mathsf{II}, \mathsf{IO}, \mathsf{OI}, \mathsf{OO}\}$, over its prefixes is generated from:

(II) A, S	(II) A, S	(IO) A, S	(IO) A, S
(Good)	(Bad)	(Good)	(Bad)
$\mathtt{P}_1 \to \mathtt{P} : k_1$	$\mathtt{P}_1 \to \mathtt{P} : k$	$\mathtt{P}_1 \to \mathtt{P} : k_1$	$\mathtt{P}_1 \to \mathtt{P} : k$
$\mathtt{P}_2 \to \mathtt{P} : k_2$	$\mathtt{P}_2 \to \mathtt{P} : k$	$\mathtt{P} \to \mathtt{P}_2 : k_2$	$\mathtt{P} \to \mathtt{P}_2 : k$
(OI) S	(OI) S	(OO) S	(OO) S
(Good)	(Bad)	(Good)	(Bad)
$\mathtt{P} \to \mathtt{P}_1 : k_1$	$\mathtt{P} \to \mathtt{P}_1 : k$	$\mathtt{P} \to \mathtt{P}_1 : k_1$	$\mathtt{P} \to \mathtt{P}_1 : k$
$\mathtt{P}_2 \to \mathtt{P} : k_2$	$\mathtt{P}_2 \to \mathtt{P} : k$	$\mathtt{P} \to \mathtt{P}_2 : k_2$	$\mathtt{P} \to \mathtt{P}_2 : k$
	$(\mathsf{OO}, \mathsf{II})$ A, S	$(\mathsf{IO}, \mathsf{OI})$ A, S	
	(Good)	(Good)	
	$\mathtt{P} \to \mathtt{P}_1 : k$	$\mathtt{P}_1 \to \mathtt{P} : k$	
	$\mathtt{P} \to \mathtt{P}_1 : k$	$\mathtt{P} \to \mathtt{P}_1 : k$	

Fig. 5. Causality Analysis

$$\mathsf{n}_1 \prec_{\mathtt{II}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2 \text{ and } \mathsf{n}_i = \mathtt{p_i} \to \mathtt{p} : m_i \ (i = 1, 2)$$

$$\mathsf{n}_1 \prec_{\mathtt{IO}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2,\ \mathsf{n}_1 = \mathtt{p_1} \to \mathtt{p} : m_1 \text{ and } \mathsf{n}_2 = \mathtt{p} \to \mathtt{p_2} : m_2.$$

$$\mathsf{n}_1 \prec_{\mathtt{OI}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2,\ \mathsf{n}_1 = \mathtt{p} \to \mathtt{p_1} : m_1 \text{ and } \mathsf{n}_2 = \mathtt{p_2} \to \mathtt{p} : m_2.$$

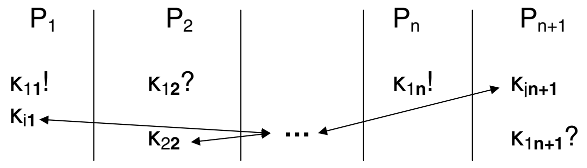
$$\mathsf{n}_1 \prec_{\mathtt{OO}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2,\ \mathsf{n}_i = \mathtt{p} \to \mathtt{p_i} : m_i \ (i = 1, 2)$$

An *input dependency* from n_1 to n_2 is a chain of the form $\mathsf{n}_1 \prec_{\phi_1} \cdots \prec_{\phi_n} \mathsf{n}_2$ $(n \geq 0)$ such that if

$$\phi_i \in \{\mathsf{OI}, \mathsf{II}\} \text{ then } \phi_{i+1} \in \{\mathsf{OO}, \mathsf{OI}\} \textit{ or}$$

$$\phi_i \in \{\mathsf{IO}, \mathsf{OO}\} \text{ then } \phi_{i+1} \in \{\mathsf{II}, \mathsf{IO}\}$$

for $1 \leq i \leq n-1$ and $\phi_n \in \{\mathsf{II}, \mathsf{OI}\}$.

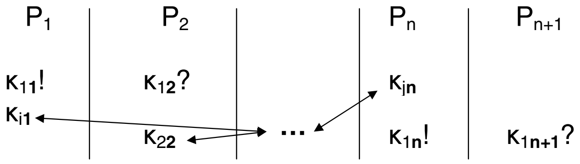


An *output dependency* from n_1 to n_2 is a chain of the form $\mathsf{n}_1 \prec_{\phi_1} \cdots \prec_{\phi_n} \mathsf{n}_2$ $(n \geq 0)$ such that if

$$\phi_i \in \{\mathsf{OI}, \mathsf{II}\} \text{ then } \phi_{i+1} \in \{\mathsf{OO}, \mathsf{OI}\} \textit{ or}$$

$$\phi_i \in \{\mathsf{IO}, \mathsf{OO}\} \text{ then } \phi_{i+1} \in \{\mathsf{II}, \mathsf{IO}\}$$

for $1 \leq i \leq n-1$ and $\phi_n \in \{\mathsf{OO}, \mathsf{IO}\}$.



Definition 3.4 (linearity) G is *linear* if, whenever $\mathsf{n}_i = \mathtt{p_i} \to \mathtt{p'_i} : m$ $(i = 1, 2)$ are in G for some m and do not occur in different branches of a branching, then both input and output dependencies exist from n_1 to n_2. In case of multicasting (values or labels), all the chains achieved by distributing each prefix of multicasting on the rest of G have to be checked if they satisfy the above conditions. If G carries other global types, we inductively demand the same.

We illustrate the condition on branching by an example:

$$\begin{array}{ll} 1. & \mathtt{A} \to \mathtt{B} : m\{ok : \mathtt{C} \to \mathtt{D} : m_1.\mathsf{end} \\ 2. & \qquad\qquad quit : \mathtt{C} \to \mathtt{D} : m_1.\mathsf{end}\ \} \end{array}$$

branching

The type represents branching: since only one of two branches is selected, there is no conflict between the two prefixes $\mathtt{C} \to \mathtt{D} : m_1$ in Lines 1 and 2.

Linearity and its violation can be detected algorithmically, without infinite unfoldings. First we observe we do need to unfold once.

$$\mu X.(\mathtt{A} \to \mathtt{B} : m.\mathsf{end}, \mathtt{B} \to \mathtt{A} : m_1.X)$$

This is linear in its 0-th unfolding (i.e. we replace X with end): but when unfolded once, it becomes non-linear, as witnessed by:

$$\mathtt{A} \to \mathtt{B} : m.\mathsf{end}, \mathtt{B} \to \mathtt{A} : m_1.\mu X.(\mathtt{A} \to \mathtt{B} : m.\mathsf{end}, \mathtt{B} \to \mathtt{A} : m_1.X)$$

since the two prefixes $\mathtt{A} \to \mathtt{B} : m$ appear in parallel. But in fact unfolding once turns out to be enough. Taking G as a syntax, let us call the *one-time unfolding of* G the result of unfolding once for each recursion in G (but never in carried types), and replacing the remaining variable with end.

Proposition 3.5 *(1) The one-time unfolding of a global type is linear iff its n-th unfolding is linear. (2) The linearity of a global type is decidable at worst case in cubic time-complexity.*

Proof. See [13]. □

4 Typing Discipline

4.1 Programming Methodology

The programming methodology of this calculus follows a top-down approach as in the asynchronous calculus [10], CDL [18] and End Point Projection [4].

The **first step** when programming a multiparty session is the definition of the global type. The global type defines the communication protocol (conversation) between only the participants of a session.

In the **second step**, the programmer programs each participant of the session. Participants can be implemented by different programmers and they may be different from the one who has written the global type.

4.2 End-point Session Types

End-point session types (see Figure 6) or end-point types capture the behaviour of a process; the constructs used to build them are those of binary session types [9] extended with multicasting send of values and labels.

$$
\begin{array}{rllr}
U & ::= & \tilde{S} \mid T@\mathtt{p} & \text{Value} \\
S & ::= & \mathsf{bool} \mid \ldots \mid \langle G \rangle & \text{Sort} \\
T & ::= & m?\langle U\rangle;T & \text{receive} \\
 & \mid & \tilde{m}!\langle U\rangle;T & \text{send} \\
 & \mid & \tilde{m} \oplus \{l_i : T_i\}_{i\in I} & \text{selection} \\
 & \mid & m\&\{l_i : T_i\}_{i\in I} & \text{branching} \\
 & \mid & \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end} &
\end{array}
$$

Fig. 6. Syntax of Local Types

Session type $\tilde{m}!\langle U\rangle;T$ represents all processes that send a value of type U on channels indexed $\tilde{m}$ and that the rest of behaviour is abstracted by T. Session type $m?\langle U\rangle;T$ represents all processes that receive a value of type U on channel indexed m and that the rest of behaviour is abstracted by T. Session type $\tilde{m} \oplus \{l_i : T_i\}_{i\in I}$ represents all processes that send one of the i labels and that the rest of behaviour is abstracted by T_i. Session type $m\&\{l_i : T_i\}_{i\in I}$ represents all processes that receive one of the i labels and that the rest of behaviour is abstracted by T_i. Session type $\mu t.\alpha$ represents all processes that have a recursive behaviour captured by T. Type session end represents the $\mathbf{0}$ process. The type U represents the same set of values as in global types. When U defines a session type then the local type represents a session channel send or receive.

4.3 Projection and Coherence

This section defines formally the projection of a global type over its participants. The results of the projection are the end-point types that will be used by the type system to type-check the process that implements the session.

Definition 4.1 (Projection) Let G be linear. The *projection of* G *onto* $\mathtt{p}'$, written $G \upharpoonright \mathtt{p}'$, is inductively given as:

$$(\mathtt{p} \to \{\mathtt{p_1}, ..., \mathtt{p_r}\} \colon \{m_1, ..., m_r\}\ \langle \tilde{S} \rangle.G') \upharpoonright \mathtt{p}' \stackrel{\text{def}}{=}$$
$$\begin{cases} m_1, ..., m_r!\langle \tilde{S} \rangle; (G' \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' = \mathtt{p} \textit{ and } \mathtt{p}' \notin \{\mathtt{p_1}, ..., \mathtt{p_r}\} \\ m_i?\langle \tilde{S} \rangle; (G' \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' \in \{\mathtt{p_1}, ..., \mathtt{p_r}\} \textit{ and } i \in \{1, ..., r\} \\ & \quad \textit{and } \mathtt{p}' \neq \mathtt{p} \\ (G' \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' \notin \{\mathtt{p_1}, ..., \mathtt{p_r}\} \textit{ and } \mathtt{p}' \neq \mathtt{p} \end{cases}$$

$$(\mathtt{p} \to \mathtt{p}'' \colon m\ \langle T@_{\mathtt{q}} \rangle.G') \upharpoonright \mathtt{p}' \stackrel{\text{def}}{=} \begin{cases} m!\langle T@_{\mathtt{q}} \rangle; (G' \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' = \mathtt{p} \textit{ and } \mathtt{p}' \neq \mathtt{p}'' \\ m?\langle T@_{\mathtt{q}} \rangle; (G' \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' = \mathtt{p}'' \textit{ and } \mathtt{p}' \neq \mathtt{p} \\ (G' \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' \neq \mathtt{p}'' \textit{ and } \mathtt{p}' \neq \mathtt{p} \end{cases}$$

$$(\mathtt{p} \to \{\mathtt{p_1}, ..., \mathtt{p_r}\} \colon \{m_1, ..., m_r\}\ \{l_k \colon G_k\}_{k \in J}) \upharpoonright \mathtt{p}' \stackrel{\text{def}}{=}$$
$$\begin{cases} m_1, ..., m_r \oplus \{l_k \colon (G_k \upharpoonright \mathtt{p}')\}_{k \in J} & \textit{if } \mathtt{p}' = \mathtt{p} \textit{ and } \mathtt{p}' \notin \{\mathtt{p_1}, ..., \mathtt{p_r}\} \\ m_i \& \{l_k \colon (G_k \upharpoonright \mathtt{p}')\}_{k \in J} & \textit{if } \mathtt{p}' \in \{\mathtt{p_1}, ..., \mathtt{p_r}\} \textit{ and } i \in \{1, ..., r\} \\ & \quad \textit{and } \mathtt{p}' \neq \mathtt{p} \\ (G_1 \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' \notin \{\mathtt{p_1}, ..., \mathtt{p_r}\} \textit{ and } \mathtt{p}' \neq \mathtt{p}, \\ & \forall k, j \in J. G_k \upharpoonright \mathtt{p}' = G_j \upharpoonright \mathtt{p}' \end{cases}$$

$$(G_1, G_2) \upharpoonright \mathtt{p}' \stackrel{\text{def}}{=} \begin{cases} G_i \upharpoonright \mathtt{p}' & \textit{if } \mathtt{p}' \in G_i \textit{ and } \mathtt{p}' \notin G_j, i \neq j \in \{1, 2\} \\ \mathsf{end} & \textit{if } \mathtt{p}' \notin G_1 \textit{ and } \mathtt{p}' \notin G_2 \end{cases}$$

$$(\mu \mathbf{t}.G) \upharpoonright \mathtt{p}' \stackrel{\text{def}}{=} \begin{cases} \mu \mathbf{t}.(G \upharpoonright \mathtt{p}') & \textit{if } \mathtt{p}' \in G \\ \mathsf{end} & \textit{if } \mathtt{p}' \notin G \end{cases}$$

$$\mathbf{t} \upharpoonright \mathtt{p}' = \mathbf{t}, \quad \textit{and } \mathsf{end} \upharpoonright \mathtt{p}' = \mathsf{end}$$

When a side condition does not hold the map is undefined.

The mapping is intuitive. In the branching, all projections should generate an identical end-point type (otherwise undefined). In the parallel composition, $\mathtt{p}'$ should be contained in at most a single type, ensuring each type is single-threaded. The single-threaded definition of the calculus does not allow programmers to define global types such as $\mathtt{A} \to \mathtt{B} : m_1\langle U \rangle$, $\mathtt{A} \to \mathtt{C} : m_2\langle U \rangle$. However, by using multicast we can model them as $\mathtt{A} \to \mathtt{B}, \mathtt{C} : \{m_1, m_2\}\langle U\rangle$, only when the values send are the same. Also, the global type $\mathtt{A} \to \mathtt{B} : m_1\{l_1 : \mathtt{B} \to \mathtt{A} : m_2\langle U\rangle,\ l_2 : \mathtt{C} \to \mathtt{A} : m_2\langle U'\rangle\}$ is not well-formed due to branching condition on projection but we can model the branching behaviour by using multicast on labels as $\mathtt{A} \to \{\mathtt{B}, \mathtt{C}\} : \{m_1, m_3\}\{l_1 : \mathtt{B} \to \mathtt{A} : m_2\langle U\rangle,\ l_2 : \mathtt{C} \to \mathtt{A} : m_2\langle U'\rangle\}$. Below $\mathsf{pid}(G)$ denotes the set of participant numbers occurring in G (but not in carried types).

Definition 4.2 (Coherence) (1) We say G is *coherent* if it is linear and $G \upharpoonright \mathtt{p}$ is well-defined for each $\mathtt{p} \in \mathsf{pid}(G)$, similarly for each carried global type inductively. (2) $\{T_{\mathtt{p}} @ \mathtt{p}\}_{\mathtt{p} \in I}$ is *coherent* if for some coherent G s.t. $I = \mathsf{pid}(G)$, we have $G \upharpoonright \mathtt{p} = T_{\mathtt{p}}$ for each $\mathtt{p} \in I$.

Theorem 4.3 *Coherence of G is decidable at the worst case in $O(n^6)$ time complexity.*

Proof. See [13]. □

4.4 Static Semantics

The static type system rules are basically the ones for binary session calculi [9] extended with rules that type multicasting session initiation. The typing system uses a map from shared names to their sorts $(S, S', ..)$. As given in Figure 6, other than atomic types, a sort has the shape $\langle G\rangle$ assuming G is coherent. Using these sorts we define the grammar of sortings and typings as follows. Below in "$\Gamma, a : S$", we assume a does not occur in Γ and in "$\Delta, \tilde{k} : \{T @ \mathtt{p}\}_{\mathtt{p} \in I}$", we assume *no* channel in $\tilde{k}$ occurs in the domain of Δ.

$$\Gamma \ ::= \ \emptyset \ \mid \ \Gamma, a : S \ \mid \ \Gamma, X : \tilde{S}\tilde{T}$$

$$\Delta \ ::= \ \emptyset \ \mid \ \Delta, \tilde{k} : \{T @ \mathtt{p}\}_{\mathtt{p} \in I}$$

A *sorting* $(\Gamma, \Gamma', ..)$ is a finite map from names to sorts and from process variables to sequences of sorts and types. *Typing* $(\Delta, \Delta', ..)$ records linear usage of session channels. In the binary sessions, it mapped each channel in its domain to a type: now it maps each vector of session channels in its domain to a family of located types. We also write $\mathsf{sid}(G)$ for the set of session channel numbers in G.

Definition 4.4 A partial operator $\cdot$ is defined as:

$$\{T_{\mathtt{p}} @ \mathtt{p}\}_{\mathtt{p} \in I} \cdot \{T'_{\mathtt{p}'} @ \mathtt{p}'\}_{\mathtt{p}' \in J} = \{T_{\mathtt{p}} @ \mathtt{p}\}_{\mathtt{p} \in I} \cup \{T'_{\mathtt{p}'} @ \mathtt{p}'\}_{\mathtt{p}' \in J}$$

if $I \cap J = \emptyset$. Then we say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if for all $\tilde{\kappa}_{\mathtt{p}} \in \mathrm{dom}(\Delta_1)$ and $\tilde{\kappa}_{\mathtt{q}} \in \mathrm{dom}(\Delta_2)$ such that $\tilde{\kappa} = \tilde{\kappa}_{\mathtt{p}} = \tilde{\kappa}_{\mathtt{q}}$ and $\Delta_1(\tilde{\kappa}_{\mathtt{p}}) \cdot \Delta_2(\tilde{\kappa}_{\mathtt{q}})$ is defined. When $\Delta_1 \asymp \Delta_2$, the *composition of* Δ_1 *and* Δ_2, written $\Delta_1 \circ \Delta_2$, is given as:

$$\Delta_1 \circ \Delta_2 = \{\tilde{\kappa}_{\mathtt{p}}, \tilde{\kappa}_{\mathtt{q}} : \Delta_1(\tilde{\kappa}_{\mathtt{p}}) \cdot \Delta_2(\tilde{\kappa}_{\mathtt{q}}) \mid \tilde{\kappa} \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)\}$$
$$\cup \Delta_1 \setminus \mathrm{dom}(\Delta_2) \cup \Delta_2 \setminus \mathrm{dom}(\Delta_1)$$

$$\Gamma, a\colon S \vdash a\colon S \qquad \Gamma \vdash \mathtt{true}, \mathtt{false}\colon \mathsf{bool} \qquad \begin{array}{c}\Gamma \vdash e_i \triangleright \mathsf{bool}\\ \Gamma \vdash e_1 \mathtt{or}\ e_2 \colon \mathsf{bool}\end{array} \qquad \text{[NameI], [Bool], [Or]}$$

$$\frac{\Gamma \vdash a\colon \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{x}\colon (G\restriction 1)@1 \quad |\tilde{x}| = |\mathsf{sid}(G)|}{\Gamma \vdash \overline{a}[2..\mathtt{n}]\,(\tilde{x}).P \triangleright \Delta} \qquad \text{[Mcast]}$$

$$\frac{\Gamma \vdash a\colon \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{x}\colon (G\restriction \mathtt{p})@\mathtt{p} \quad |\tilde{x}| = |\mathsf{sid}(G)|}{\Gamma \vdash a[\mathtt{p}]\,(\tilde{x}).P \triangleright \Delta} \qquad \text{[Macc]}$$

$$\frac{\Gamma \vdash \tilde{e}\colon \tilde{S} \qquad \Gamma \vdash P \triangleright \Delta, \tilde{k}\colon T@\mathtt{p}}{\Gamma \vdash k[m_1, ..., m_n]!\langle\tilde{e}\rangle; P \triangleright \Delta, \tilde{k}\colon m_1, ..., m_n!\langle\tilde{S}\rangle; T@\mathtt{p}} \qquad \text{[Send]}$$

$$\begin{array}{c}\Gamma, \tilde{y}\colon \tilde{S} \vdash P \triangleright \Delta, \tilde{k}\colon T@\mathtt{p}\\ \Gamma \vdash k[m]?(\tilde{y}); P \triangleright \Delta, \tilde{k}\colon m?\langle\tilde{S}\rangle; T@\mathtt{p}\end{array} \qquad \text{[Rcv]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{k}\colon T@\mathtt{p}}{\Gamma \vdash k[m]!\langle\tilde{t}\rangle; P \triangleright \Delta, \tilde{k}\colon m!\langle T'@\mathtt{p}'\rangle; T@\mathtt{p},\ \tilde{t}\colon T'@\mathtt{p}'} \qquad \text{[Thr]}$$

$$\begin{array}{c}\Gamma \vdash P \triangleright \Delta, \tilde{k}\colon T@\mathtt{p},\ \tilde{y}\colon T'@\mathtt{p}'\\ \Gamma \vdash k[m]?(\tilde{y}); P \triangleright \Delta, \tilde{k}\colon m?\langle T'@\mathtt{p}'\rangle; T@\mathtt{p}\end{array} \qquad \text{[Cat]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{k}\colon T_j@\mathtt{p} \qquad j \in I}{\Gamma \vdash k[m_1, ..., m_n] \lhd l_j; P \triangleright \Delta, \tilde{k} : m_1, ..., m_n \oplus \{l_i : T_i\}_{i\in I}@\mathtt{p}} \qquad \text{[Sel]}$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta, \tilde{k}\colon T_i@\mathtt{p} \qquad \forall i \in I}{\Gamma \vdash\ k[m] \rhd \{l_i\colon P_i\}_{i\in I} \triangleright \Delta, \tilde{k}\colon m\,\&\{l_i : T_i\}_{i\in I}@\mathtt{p}} \qquad \text{[Br]}$$

$$\begin{array}{c}\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta' \qquad \Delta \asymp \Delta'\\ \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'\end{array} \qquad \text{[Conc]}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta} \qquad \text{[If]}$$

$$\begin{array}{c}\Gamma \vdash P \triangleright \Delta \qquad \Delta <: \Delta'\\ \Gamma \vdash P \triangleright \Delta'\end{array} \qquad \text{[<:]}$$

$$\begin{array}{c}\Delta\ \mathsf{end\ only}\\ \Gamma \vdash \mathbf{0} \triangleright \Delta\end{array} \qquad \begin{array}{c}\Gamma, a\colon \langle G\rangle \vdash P \triangleright \Delta\\ \Gamma \vdash (\nu a)P \triangleright \Delta\end{array} \qquad \text{[Inact],[NRes]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}_1} : T_1@\mathtt{p}_1 \circ ... \circ \tilde{\kappa}_{\mathtt{p}_n} : T_n@\mathtt{p}_n}{\Gamma \vdash (\nu\tilde{\kappa})P \triangleright \Delta} \qquad \text{[CRes]}$$

$$\begin{array}{c}\Gamma \vdash \tilde{e}\colon \tilde{S} \qquad \Delta\ \mathsf{end\ only}\\ \Gamma, X\colon \tilde{S}\tilde{T} \vdash X\langle\tilde{e}, \tilde{k}\rangle \triangleright \Delta, \tilde{k} : \tilde{T}@\tilde{\mathtt{p}}\end{array} \qquad \text{[Var]}$$

$$\begin{array}{c}\Gamma, X\colon \tilde{S}\tilde{T}, \tilde{y} : \tilde{S} \vdash P \triangleright \tilde{y'} : \tilde{T}@\tilde{\mathtt{p}}\\ \Gamma, X\colon \tilde{S}\tilde{T} \vdash Q \triangleright \Delta\\ \Gamma \vdash \mathtt{def}\ X(\tilde{y}, \tilde{y'}) = P\ \mathtt{in}\ Q \triangleright \Delta\end{array} \qquad \text{[Def]}$$

Fig. 7. Typing System for Expressions and Processes

The paragraph gives a description of the static type system rules. A multicast session initiation (accept or request) process is well-typed if the process under (the prefix) is well-typed with the end-point type obtained by projection. The end-point types for each process of a session are stored in the typing of the shared name where the session has initiated ($a : \langle G\rangle$). A value process that sends or receive a value is well-typed if the process under (the prefix) is well-typed. The notation of channels in rules [Send] and [Sel] is defined in Notation 2.2 and Notation 2.3. A process that selects a label is well-typed if the continuation process is well-typed. A process that branches over a set of labels is well-typed if the continuation processes over that set are well-typed. A parallel composition process is well-typed if each component process is well-typed and the two components belong to two different process entities. The system considers as a structural congruence rule the following one on restriction of channels: $(\nu k)(\nu k')P \equiv (\nu k, k')P$.

4.5 Type-checking Examples

This section gives the type-checking of the processes under the prefixes of multicasting session request or session accept introduced in Section 2.3 with the end-point types obtained by the projection algorithm.

Addition Protocol. Type-checking is defined for the process under the prefix of the multicasting session request or session accept with the sorting list $\Gamma = \{a : \langle G\rangle\}$, letting `Client` = 1, `Addition` = 2, `Successor` = 3, `Predecessor` = 4:

$$\Gamma \vdash \mathit{client} \triangleright x_1, x_2, x_3 : 1!\langle \mathit{int}\rangle; 1!\langle \mathit{int}\rangle; 1?\langle \mathit{int}\rangle; \mathsf{end}@\mathtt{Client}$$

$$\Gamma \vdash \mathit{addition} \triangleright x_1, x_2, x_3 : 1?\langle \mathit{int}\rangle; 1?\langle \mathit{int}\rangle; \mu t.2, 3\oplus$$
$$\{\mathit{true} : 1!\langle \mathit{int}\rangle; \mathsf{end},$$
$$\mathit{false} : 2!\langle \mathit{int}\rangle; 2?\langle \mathit{int}\rangle; 3!\langle \mathit{int}\rangle; 3?\langle \mathit{int}\rangle; t\}@\mathtt{Addition}$$

$$\Gamma \vdash \mathit{successor} \triangleright x_1, x_2, x_3 : \mu t.2\&\{\mathit{true} : \mathsf{end}, \mathit{false} : 2?\langle \mathit{int}\rangle; 2!\langle \mathit{int}\rangle; t\}@\mathtt{Successor}$$

$$\Gamma \vdash \mathit{predecessor} \triangleright x_1, x_2, x_3 : \mu t.3\&\{\mathit{true} : \mathsf{end}, \mathit{false} : 3?\langle \mathit{int}\rangle; 3!\langle \mathit{int}\rangle; t\}@\mathtt{Predecessor}$$

Fire Alarm Protocol. With the sorting list $\Gamma = \{a : \langle G'\rangle\}$ where G' is:

$$\mathtt{Controller} \to \mathtt{FireAlarm_1}, ..., \mathtt{FireAlarm_{j-1}} : 1, ..., j-1\langle \mathit{string}\rangle$$

$$\mathtt{Controller} \to \mathtt{Elevator_j}, ..., \mathtt{Elevator_{j+k-1}} : j, ..., j+k-1\langle \mathit{string}\rangle$$

letting $\mathtt{Controller}$ = 1, $\mathtt{FireAlarm_1}$ = 2, ..., $\mathtt{FireAlarm_{j-1}}$ = j, $\mathtt{Elevator_j}$ = j+1, ..., $\mathtt{Elevator_{j+k-1}}$ = j+k, the processes under the prefix of the multicasting session request or session accept are type-checked:

$$\Gamma \vdash \mathit{controller} \triangleright x_1, ..., x_{j+k-1} : 1, ..., j-1!\langle \mathit{string}\rangle; j, ..., j+k-1!\langle \mathit{string}\rangle; \mathsf{end}@\mathtt{Controller}$$

$$\Gamma \vdash \mathit{fire\ alarm}_1 \triangleright x_1, ..., x_{j+k-1} : 1?\langle \mathit{string}\rangle; \mathsf{end}@\mathtt{FireAlarm_1}$$

$$\ldots$$

$$\Gamma \vdash \mathit{fire\ alarm}_{j-1} \triangleright x_1, ..., x_{j+k-1} : j-1?\langle \mathit{string}\rangle; \mathsf{end}@\mathtt{FireAlarm_{j-1}}$$

$$\Gamma \vdash \mathit{elevator}_j \triangleright x_1, ..., x_{j+k-1} : j?\langle \mathit{string}\rangle; \mathsf{end}@\mathtt{Elevator_j}$$

$$\ldots$$

$$\Gamma \vdash \mathit{elevator}_{j+k-1} \triangleright x_1, ..., x_{j+k-1} : j+k-1?\langle \mathit{string}\rangle; \mathsf{end}@\mathtt{Elevator_{j+k-1}}$$

Delegation. With the assumption list $\Gamma = \{a : \langle G_a\rangle, b : \langle G_b\rangle\}$ where G_a and G_b are:

$$G_b = \mathtt{A} \to \mathtt{D}\colon 1\ \langle 1!\langle string\rangle; \mathsf{end}@\mathtt{A}\rangle.\mathsf{end}$$

$$G_a = \mathtt{A} \to \mathtt{C}\colon 1\ \langle string\rangle.\mathtt{B} \to \mathtt{C}\colon 2\ \langle string\rangle.\mathsf{end}.$$

letting $\mathtt{A} = 1$, $\mathtt{B} = 2$, $\mathtt{C} = 3$, $\mathtt{D} = 4$, the typechecking of the processes is defined as follows:

$$\Gamma \vdash A \triangleright y_1 : 1!\langle 1!\langle string\rangle; \mathsf{end}@\mathtt{A}\rangle@\mathtt{A}, x_1, x_2 : 1!\langle string\rangle@\mathtt{A}$$

$$\Gamma \vdash B \triangleright x_1, x_2 : 2!\langle string\rangle@\mathtt{B}$$

$$\Gamma \vdash C \triangleright x_1, x_2 : 1?\langle string\rangle; 2?\langle string\rangle@\mathtt{C}$$

$$\Gamma \vdash D \triangleright y_1 : 1?\langle 1!\langle string\rangle; \mathsf{end}@\mathtt{A}\rangle@\mathtt{D}$$

4.6 Soundness

We now prove that the type system we have introduced is sound: its type-checking rules prove only terms that are valid with respect to the operational semantics.

We need subject congruence when proving subject reduction for [Str].

Theorem 4.5 (subject congruence) *$\Gamma \vdash P \triangleright \Delta$ and $P \equiv P'$ imply $\Gamma \vdash P' \triangleright \Delta$.*

Proof. See Appendix A. □

Theorem 4.6 (subject reduction) *$\Gamma \vdash P \triangleright \Delta$ with Δ coherent and $P \to P'$ imply $\Gamma \vdash P' \triangleright \Delta'$ where $\Delta = \Delta'$ or $\Delta \to \Delta'$ with Δ' coherent.*

Note the definition of coherence for Δ is given in Definition 4.2(2).

Proof. See Appendix A. □

5 Related and Future Work

Synchronous Session Types

Multiparty session types have been firstly studied for asynchronous communication calculi [10,1]. In these calculi, for problems that specify a strict order of communications, programmers have to specify the order by adding extra communications, that send an empty message, and channels to preserve linearity of the late ones. Considering the fire alarm system introduced in Section 1 for a calculus of asynchronous communications e.g. [10], the global type is now defined:

$$\mathtt{Controller} \to \mathtt{FireAlarm_1}, ..., \mathtt{FireAlarm_{j-1}} : 1, ..., j-1\langle string\rangle.$$

$$\mathtt{FireAlarm_1} \to \mathtt{Controller} : j\langle\rangle.$$

$$...$$

$$\mathtt{FireAlarm_{j-1}} \to \mathtt{Controller} : 2*j-1\langle\rangle.$$

$$\mathtt{Controller} \to \mathtt{Elevator_j}, ..., \mathtt{Elevator_{j+k-1}} : 2*j, ..., 2*j+k-1\langle string\rangle$$

where the additional communications between the fire alarms and the controller, and the new channels are introduced to preserve the order of communications between $\mathtt{Controller\text{-}FireAlarm}_1$, ..., $\mathtt{FireAlarm}_{\mathtt{j}-1}$ and $\mathtt{Controller\text{-}Elevator}_{\mathtt{j}}$, ..., $\mathtt{Elevator}_{\mathtt{j}+\mathtt{k}-1}$ and linearity.

Recently, contracts for web-services [5] have been studied for a calculus of synchronous communications. A contract is a binary session type between a client and a service, that can capture a combination of an internal and external choice at participants. The work on contracts does not address ordering and causalities of communications in a multiparty session. Merging of sessions is modeled via interactions inside a session and locations in [3]; the result of the type safety property is left as future work.

A calculus of service-oriented computing is introduced in [17], where a conversation models the interactions between a client and various services. New primitives of communication are introduced such as conversation context (shared interaction point) communication and communication inside an end-point. An exception handling mechanism similar to those proposed for functional languages is introduced for the calculus. The calculus in essence is similar to the ones presented for session types [9,16] but does not address resolution of safe communications at static time.

Choreography* and *Orchestration

WS-CDL [18] is the first language that uses the metaphor of *choreography* to describe interactions between participants of a session. A distilled version of WS-CDL [4] is used to study a theory of end-point projection (EPP). The global calculus syntax given in EPP offers syntactic sugar useful to write invocation-based protocols such as assignment of processes to local variables and independent choice over global behaviors. In contrast to the global calculus, global types support high-order communication and multicasting. "Choreography" is used to describe cryptographic protocols [7], which protect session execution from both external attackers and malicious participants. The work in [7] defines a model to program cryptographic systems rather than a typing discipline for programming languages.

WS-BPEL [2] is the first language that uses the metaphor of *orchestration* to describe interactions between participants of a session.

Implementation of Session Types

Several academic projects address the implementation of binary session types in Java [11], Haskell [14] and C++ [6]. Scribble [15] is an implementation for Java of multiparty session types as an industry project.

Future work:

- **Inner Delegation.** A scenario of *inner delegation* is defined when a participant delegates its part of session to a participant that is already part of that session. Such scenario can reduce to a process that can stuck at runtime; e.g.

$$G_a = \mathtt{A} \to \mathtt{C}\colon\ 1\langle \mathit{string}\rangle.\mathtt{B} \to \mathtt{C}\colon\ 2\langle \mathit{string}\rangle.\mathsf{end}$$

$$G_b = \mathtt{A} \to \mathtt{B}\colon\ 1\langle 1!\langle \mathit{string}\rangle.\mathsf{end}\rangle.\mathsf{end}$$

where the session at a is defined between participants A, B and C, and A delegates the ability of sending a-string-to-C to B. The implementation of the global type:

- $A \triangleq \overline{a}[2,3]\,(x_1, x_2).\overline{b}[2]\,(y_1).y_1!\langle x_1, x_2\rangle;$
- $B \triangleq a[2]\,(x_1, x_2).b[2]\,(y_1).y_1?(y_2, y_3); x_2!\langle\text{“}\mathit{The\ computer\ and\ the\ brain}\text{”}\rangle;$
  $y_2!\langle\text{“}\mathit{John\ von\ Neumann}\text{”}\rangle;$
- $C \triangleq a[3]\,(x_1, x_2).x_1?(y'); x_2?(y'');$

stucks on the first interaction of process B with process C as the late one is waiting to receive on channel place hold by x_1 whilst, the former is sending on channel place hold by x_2.

- **Delegation in Multicast.** The actual syntax of global types does not allow programmers to write global types that contain delegation in multicast. It is intuitive that delegating to more than one participant the same behaviour breaks progress.

  Multicast in delegation as an operation where a behaviour is split and delegated to several participants, can be an interesting construct; e.g.

$$G_a = \mathtt{A} \to \mathtt{C}\colon 1\ \langle \mathit{string}\rangle.\mathtt{A} \to \mathtt{C}\colon 1\ \langle \mathit{string}\rangle.\mathtt{B} \to \mathtt{C}\colon 2\ \langle \mathit{int}\rangle.\mathsf{end}.$$

$$G_b = \mathtt{A} \to \mathtt{D}, \mathtt{E}\colon 1, 2\ \langle 1!\langle \mathit{string}\rangle @A, 1!\langle \mathit{string}\rangle @A\rangle.\mathsf{end}.$$

  where participant A splits the behaviour of sending the title and the author of a book into two independent behaviors and delegates each of them to D and E respectively. However, the splitting operation should be defined only when there is no order between the two behaviors. The implementation below illustrates the problem introduced by the new construct:

  - $A \triangleq \overline{a}[2,3]\,(x_1, x_2).\overline{b}[2,3]\,(y_1, y_2).y_1, y_2!\langle x_1, x_2\rangle;$
  - $B \triangleq a[2]\,(x_1, x_2).x_2!\langle 11\rangle;$
  - $C \triangleq a[3]\,(x_1, x_2).x_1?(y'); x_1?(y'); x_2?(y'');$
  - $D \triangleq b[2]\,(y_1, y_2).y_1?(y_3, y_4); y_3!\langle\text{“}\mathit{The\ computer\ and\ the\ brain}\text{”}\rangle;$
  - $E \triangleq b[3]\,(y_1, y_2).y_2?(y_3, y_4); y_3!\langle\text{“}\mathit{John\ von\ Neumann}\text{”}\rangle;$

  where processes D and E are both able to send their messages at the same time, breaking therefore their order.

As future work, we plan to develop a typing theory that checks global types from inner-delegation scenarios and rejects implementation of such scenarios. We plan to extend delegation with multicasting; i.e. allowing delegation of different behaviors that do not have dependencies between them to a group of participants simultaneously.

Acknowledgement

We thank the reviewers for their comments. The work is partially supported by EPSRC GR/T03208, EPSRC GR/T03215, and IST2005–015905 MOBIUS.

References

[1] Bonelli, E. and A. B. Compagnoni, *Multipoint session types for a distributed calculus*, in: *TGC*, Lecture Notes in Computer Science **4912** (2008), pp. 240–256.

[2] *Web services business process execution language version 2.0*, Available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[3] Bruni, R., I. Lanese, H. C. Melgratti and E. Tuosto, *Multiparty sessions in SOC*, in: D. Lea and G. Zavattaro, editors, *COORDINATION*, Lecture Notes in Computer Science **5052** (2008), pp. 67–82.

[4] Carbone, M., K. Honda and N. Yoshida, *Structured communication-centred programming for web services*, in: *ESOP*, Lecture Notes in Computer Science **4421** (2007), pp. 2–17.

[5] Castagna, G., N. Gesbert and L. Padovani, *A theory of contracts for web services*, in: G. C. Necula and P. Wadler, editors, *POPL* (2008), pp. 261–272.

[6] Collingbourne, P. and P. Kelly, *Inference of session types from control flow*, FESCA, ENTCS (2008).

[7] Corin, R., P.-M. Deniélou, C. Fournet, K. Bhargavan and J. J. Leifer, *Secure implementations for typed session abstractions*, in: *CSF* (2007), pp. 170–186.

[8] Gay, S. J. and M. Hole, *Subtyping for session types in the pi calculus*, Acta Inf. **42** (2005), pp. 191–225.

[9] Honda, K., V. T. Vasconcelos and M. Kubo, *Language primitives and type discipline for structured communication-based programming*, in: C. Hankin, editor, *ESOP*, Lecture Notes in Computer Science **1381** (1998), pp. 122–138.

[10] Honda, K., N. Yoshida and M. Carbone, *Multiparty asynchronous session types*, in: G. C. Necula and P. Wadler, editors, *POPL* (2008), pp. 273–284.

[11] Hu, R., *Session-based Distributed Programming in JAVA*, Available at http://www.doc.ic.ac.uk/~rh105/sessiondj.html.

[12] Hu, R., N. Yoshida and K. Honda, *Session-based distributed programming in JAVA*, in: J. Vitek, editor, *ECOOP*, Lecture Notes in Computer Science **5142** (2008), pp. 516–541.

[13] *Multiparty asynchronous session types*, Available at http://www.doc.ic.ac.uk/~ab406/journal.pdf.

[14] Sackman, M., *Session Types in Haskell*, Available at http://www.wellquite.org/sessions/tutorial_1.html.

[15] *Scribble*, Available at http://pi4scribble.wiki.sourceforge.net/.

[16] Takeuchi, K., K. Honda and M. Kubo, *An interaction-based language and its typing system*, in: C. Halatsis, D. G. Maritsas, G. Philokyprou and S. Theodoridis, editors, *PARLE*, Lecture Notes in Computer Science **817** (1994), pp. 398–413.

[17] Vieira, H. T., L. Caires and J. C. Seco, *The conversation calculus: A model of service-oriented computation*, in: S. Drossopoulou, editor, *ESOP*, Lecture Notes in Computer Science **4960** (2008), pp. 269–283.

[18] *Web services choreography description language version 1.0*, Available at http://www.w3.org/2002/ws/chor/edcopies/cdl/cdl.html.

[19] Yoshida, N. and V. T. Vasconcelos, *Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication*, Electr. Notes Theor. Comput. Sci. **171** (2007), pp. 73–93.

A Soundness

Notation A.1 *"By inversion" denotes inversion on a rule. That is, a conclusion judgment that is achieved by applying a certain rule is true if the premises on that rule are true.*

Notation A.2 *"By rule" denotes applying a rule. That is, given the premises and side conditions of a rule then we can conclude the judgment by applying that rule.*

A.1 Subject Reduction

Subject reduction ensures that the type of an expression is preserved during its evaluation. For the proof of subject reduction, we need three standard properties: *channel replacement*, *weakening* and *substitution* lemma. We need the channel replacement lemma for rules [Link], [Multicasting] and [Def], weakening for rule [Def] and subject congruence, and substitution for rule [Multicasting] and [Def].

Lemma A.3 (substitution and weakening) (1) $\Gamma, \tilde{x}:\tilde{S} \vdash P \triangleright \Delta$ *and* $\Gamma \vdash \tilde{v}:\tilde{S}$ *imply* $\Gamma \vdash P[\tilde{v}/\tilde{x}] \triangleright \Delta$. (2) *Whenever* $\Gamma \vdash P \triangleright \Delta$ *is derivable then its weakening,* $\Gamma \vdash P \triangleright \Delta, \Delta'$ *for disjoint* Δ' *where* Δ' *contains only empty type contexts and for types* end*, is also derivable.*

Proof. Standard, see [19]. □

Lemma A.4 (Channel Replacement) *If* $\Gamma \vdash P \triangleright \Delta, \tilde{x}: T@\mathtt{p}$ *and* $\tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta)$*, then* $\Gamma \vdash P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p}$.

Proof. A straightforward induction on the derivation tree for P. We give the proof of the most interesting cases.
Case: [Conc]

$$\frac{\Gamma \vdash P \triangleright \Delta_1 \qquad \Gamma \vdash Q \triangleright \Delta_2 \qquad \Delta_1 \asymp \Delta_2}{\Gamma \vdash P \mid Q \triangleright \Delta, \tilde{x}: T@\mathtt{p}}$$

$$\begin{array}{lr}
\Gamma \vdash P \mid Q \triangleright \Delta, \tilde{x}: T@\mathtt{p} \text{ and } \tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta) & \text{By assumption} \\
\Gamma \vdash P \triangleright \Delta_1 \quad \Gamma \vdash Q \triangleright \Delta_2 \quad \Delta_1 \asymp \Delta_2 & \\
\text{where } \Delta, \tilde{x}: T@\mathtt{p} = \Delta_1 \circ \Delta_2 \text{ and } \tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta) & \text{By inversion on [Conc]} \\
\textbf{First Subcase}: \tilde{x}: T@\mathtt{p} \in \Delta_1 \text{ and } \tilde{x}: T@\mathtt{p} \notin \Delta_2 & \text{By } \Delta_1 \asymp \Delta_2 \\
\Delta_1 = \Delta_1', \tilde{x}: T@\mathtt{p} \quad \Gamma \vdash P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta_1', \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \text{By induction} \\
\Gamma \vdash P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \mid Q \triangleright \Delta_1', \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} \circ \Delta_2 & \text{By rule [Conc]} \\
\Delta_1', \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} \circ \Delta_2 = \Delta_1' \circ \Delta_2, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta) \\
\Gamma \vdash P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \mid Q \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \\
\textbf{Second Subcase}: \tilde{x}: T@\mathtt{p} \in \Delta_2 \text{ and } \tilde{x}: T@\mathtt{p} \notin \Delta_1 & \text{By } \Delta_1 \asymp \Delta_2 \\
\Delta_2 = \Delta_2', \tilde{x}: T@\mathtt{p} \quad \Gamma \vdash Q[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta_2', \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \text{By induction} \\
\Gamma \vdash P \mid Q[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta_1 \circ (\Delta_2', \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p}) & \text{By rule [Conc]} \\
(\Delta_2', \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p}) = \Delta_1' \circ \Delta_2, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta) \\
\Gamma \vdash P \mid Q[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} &
\end{array}$$

Note that $(P \mid Q)[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] = P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \mid Q$ or $(P \mid Q)[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] = P \mid Q[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}]$.

Case: [If]

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta, \tilde{x}: T@\mathtt{p} \qquad \Gamma \vdash Q \triangleright \Delta, \tilde{x}: T@\mathtt{p}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta, \tilde{x}: T@\mathtt{p}}$$

$$\begin{array}{lr}
\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta, \tilde{x}: T@\mathtt{p} \text{ and } \tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta) & \text{By assumption} \\
\Gamma \vdash P \triangleright \Delta, \tilde{x}: T@\mathtt{p} \text{ and } \Gamma \vdash Q \triangleright \Delta, \tilde{x}: T@\mathtt{p} \text{ and } \tilde{\kappa}_{\mathtt{p}} \notin \mathrm{dom}(\Delta) & \text{By inversion on [If]} \\
\Gamma \vdash P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} \text{ and } \Gamma \vdash Q[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \text{By induction} \\
\Gamma \vdash (\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q)[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] \triangleright \Delta, \tilde{\kappa}_{\mathtt{p}}: T@\mathtt{p} & \text{By rule [If]}
\end{array}$$

Note that $(\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q)[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}] = \mathtt{if}\ e\ \mathtt{then}\ P[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}]\ \mathtt{else}\ Q[\tilde{\kappa}_{\mathtt{p}}/\tilde{x}]$.
□

Next we introduce reduction over session typings, which abstractly represent interaction (message delivery) in processes. We also assume well-formedness of types.

$$m_1, ..., m_n!\langle\tilde{S}\rangle; T@\mathtt{p}, m_1?\langle\tilde{S}\rangle; T_1@\mathtt{p}_1, ..., m_n?\langle\tilde{S}\rangle; T_n@\mathtt{p}_n \rightarrow T@\mathtt{p}, T_1@\mathtt{p}_1, ..., T_n@\mathtt{p}_n \qquad \text{[TR-Mult]}$$

$$m!\langle T_2@\mathtt{p}_2\rangle; T@\mathtt{p}, m?\langle T_2@\mathtt{p}_2\rangle; T_1@\mathtt{p}_1 \rightarrow T@\mathtt{p}, T_1@\mathtt{p}_1 \qquad \text{[TR-MultD]}$$

$$\begin{array}{r}
m_1, ...m_n \oplus \{...,\ l: T,\ ...\}@\mathtt{p}, m_1\&\{..., l: T_1,\ ...\}@\mathtt{p}_1, ..., m_n\&\{..., l: T_n,\ ...\}@\mathtt{p}_n \\
\rightarrow T@\mathtt{p}, T_1@\mathtt{p}_1, ..., T_n@\mathtt{p}_n \qquad \text{[TR-MultL]}
\end{array}$$

$$\frac{T_1@\mathtt{p}_1, ..., T_n@\mathtt{p}_n \rightarrow T_1'@\mathtt{p}_1, ..., T_n'@\mathtt{p}_n}{\Delta, \tilde{\kappa}_{\mathtt{p}_1}, ..., \tilde{\kappa}_{\mathtt{p}_n}: T_1@\mathtt{p}_1, ..., T_n@\mathtt{p}_n, \_ \rightarrow \Delta, \tilde{\kappa}_{\mathtt{p}_1}, ..., \tilde{\kappa}_{\mathtt{p}_n}: T_1'@\mathtt{p}_1, ..., T_n'@\mathtt{p}_n, \_} \qquad \text{[TR-Context]}$$

Definition A.5 **(1)** *(coherence of typings) We say Δ is coherent if $\Delta(\tilde{k})$ is coherent for each $\tilde{k} \in \mathrm{dom}(\Delta)$.*

(2) *(full projection) Assume G is coherent and let $G \upharpoonright \mathsf{p}_i = T_i$ for each $\mathsf{p}_i \in \mathsf{pid}(G)$. Then $[\![G]\!]$, called* full projection of P, *denotes the family $\{T_i@\mathsf{p}_i\}$.*

(3) *(causal edges on $[\![G]\!]$) For $[\![G]\!]$ given above, regarding each type in $[\![G]\!]$ as the corresponding regular tree, we define the causal edges $\prec_{\mathtt{II}}$, $\prec_{\mathtt{IO}}$, $\prec_{\mathtt{OI}}$ and $\prec_{\mathtt{OO}}$ among its prefixes precisely we have done in G.*

Proposition A.6 *Each causal edge in G is preserved and reflected through the projection onto $[\![G]\!]$.*

Proof. This is because every causal edges record prefixing on the same participant is preserved by projection. The statement becomes more clear when considering the four possible cases of edges on prefixes:

Case II. $\mathtt{B} \to \mathtt{A}\colon m_1.G.\mathtt{C} \to \mathtt{A}\colon m_2$, where $\mathtt{B} \to \mathtt{A}\colon m_1 \prec_{\mathtt{II}} \mathtt{C} \to \mathtt{A}\colon m_2$, is mapped into $\{(m_1!\langle\rangle; G \upharpoonright \mathtt{B})@\mathtt{B}, (m_1?\langle\rangle; G \upharpoonright \mathtt{A}; m_2?\langle\rangle)@\mathtt{A}, (G \upharpoonright \mathtt{C}; m_2!\langle\rangle)@\mathtt{C}\}$

Case IO. $\mathtt{B} \to \mathtt{A}\colon m_1.G.\mathtt{A} \to \mathtt{C}\colon m_2$, where $\mathtt{B} \to \mathtt{A}\colon m_1 \prec_{\mathtt{IO}} \mathtt{A} \to \mathtt{C}\colon m_2$, is mapped into $\{(m_1!\langle\rangle; G \upharpoonright \mathtt{B})@\mathtt{B}, (m_1?\langle\rangle; G \upharpoonright \mathtt{A}; m_2!\langle\rangle)@\mathtt{A}, (G \upharpoonright \mathtt{C}; m_2?\langle\rangle)@\mathtt{C}\}$

Case OI. $\mathtt{A} \to \mathtt{B}\colon m_1.G.\mathtt{C} \to \mathtt{A}\colon m_2$, where $\mathtt{A} \to \mathtt{B}\colon m_1 \prec_{\mathtt{OI}} \mathtt{C} \to \mathtt{A}\colon m_2$, is mapped into $\{(m_1?\langle\rangle; G \upharpoonright \mathtt{B})@\mathtt{B}, (m_1!\langle\rangle; G \upharpoonright \mathtt{A}; m_2?\langle\rangle)@\mathtt{A}, (G \upharpoonright \mathtt{C}; m_2!\langle\rangle)@\mathtt{C}\}$

Case OO. $\mathtt{A} \to \mathtt{B}\colon m_1.G.\mathtt{A} \to \mathtt{C}\colon m_2$, where $\mathtt{A} \to \mathtt{B}\colon m_1 \prec_{\mathtt{OO}} \mathtt{A} \to \mathtt{C}\colon m_2$, is mapped into $\{(m_1?\langle\rangle; G \upharpoonright \mathtt{B})@\mathtt{B}, (m_1!\langle\rangle; G \upharpoonright \mathtt{A}; m_2!\langle\rangle)@\mathtt{A}, (G \upharpoonright \mathtt{C}; m_2?\langle\rangle)@\mathtt{C}\}$

□

Definition A.7 (merge set) *Assume G is coherent. Then we say two prefixes in G in different branches of a branching prefix are* mergeable *with each other when they are collapsed in its projection. A prefix is always mergeable with itself. Given a prefix n, its* merge set *is the set of prefixes mergeable with n.*

Proposition A.8 *Two prefixes in G are mergeable iff they are related to one common input prefix and one common output prefix in $[\![G]\!]$ through projection.*

Proof. This is because, in the defining clauses of projection, there are no other cases than the one for branching which collapse two prefixes. □

Proposition A.9 (1) *If a pair of prefixes in $[\![G]\!]$ form a redex with respect to $\to$ then they are not prefixed by any pair of prefixes that form a $\prec_{\mathtt{II}}$, $\prec_{\mathtt{IO}}$, $\prec_{\mathtt{OI}}$ or $\prec_{\mathtt{OO}}$ dependency.*

(2) *Given coherent G, let G' be the result of taking off the merge set of a prefix from G which is not prefixed by any of $\prec_{\mathtt{II}}$, $\prec_{\mathtt{IO}}$, $\prec_{\mathtt{OI}}$ or $\prec_{\mathtt{OO}}$. Then G' is again coherent.*

(3) *Let G be coherent. Then the causal edges are preserved and reflected between the two merge sets in G and their images in $[\![G]\!]$. Further each redex pair in $[\![G]\!]$ is the image of some prefix in G.*

Proof. For (1), observe that redexes in the base rules over session typing, [TR-Mult], [TR-MultD] and [TR-MultL], are in the minimal positions and since there is no permuation of prefixes, as it is for the asynchronoous calculus for OO, we conclude. (Two output-output actions are strictly orderd due to synchrony.)

For (2) , first, for linearity, suppose $\mathsf{n}_{1,2}$ are in G' sharing a channel. Then they are also in G and causal edges between them do not differ so they have the same dependencies as in G. Second, the coherence in projection is immediate since we lose one prefix from the projection of each branch.

For (3), the first part is immediate from the construction. For the second point assume there is a redex pair in $[\![G]\!]$ whose two parts have different pre-images. Then we have co-occurring prefixes in G which are not related by the two dependencies, by (1) and the first part of (3), a contradiction. □

Lemma A.10 **(1)** *$\Delta_1 \to \Delta_1'$ for $\tilde{\kappa}$ and $\Delta_1 \asymp \Delta_2$ imply $\Delta_1' \asymp \Delta_2$ and $\Delta_1 \circ \Delta_2 \to \Delta_1' \circ \Delta_2$.*

(2) *Let Δ be coherent. Then $\Delta \to \Delta'$ implies Δ' is coherent.*

Proof.

(1) For (1) suppose $\Delta_1 \to \Delta_1'$ and $\Delta_1 \asymp \Delta_2$. Note $\Delta_1 \asymp \Delta_2$ means that each pair of vectors of channels from $\Delta_{1,2}$ either coincide or are disjoint, and that, if they coincide, their image are participant-wise composable by $\circ$. Since no typed reduction rule invalidate either condition we conclude $\Delta_1' \asymp \Delta_2$. $\Delta_1 \circ \Delta_2 \to \Delta_1' \circ \Delta_2$ follows directly from [TR-Context].

(2)For (2), suppose Δ is coherent and $\Delta \to \Delta'$. Suppose the associated redex is in $\Delta(\tilde{s})$. By coherence we can write $\Delta(\tilde{s})$ as $[\![G]\!]$ for some coherent G. Now consider the preimage of the associated redex in $[\![G]\!]$, whose existence is guaranteed by Proposition A.9 (3). This preimage is not suppressed (related) by causal edges by Proposition A.9 (1,3). reducing $[\![G]\!]$ corresponds to eliminating its preimage from G, say G', whose projection $[\![G']\!]$ precisely gives the result of reducing $[\![G]\!]$. Since G' is coherent by Proposition A.9 (2) we are done.

□

We need subject congruence when proving subject reduction for [Str].

Theorem A.11 (subject congruence) *$\Gamma \vdash P \triangleright \Delta$ and $P \equiv P'$ imply $\Gamma \vdash P' \triangleright \Delta$.*

Proof. By rule induction on the derivation of $\Gamma \vdash P \triangleright \Delta$ when assuming that $P \equiv P'$ and $\Gamma \vdash P' \triangleright \Delta$ when assuming that $P' \equiv P$. For each structural congruence axiom, we consider each session type system rule that can generate $\Gamma \vdash P \triangleright \Delta$.

Case: $P \mid \mathbf{0} \equiv P$

$\Gamma \vdash P \mid \mathbf{0} \triangleright \Delta$ By assumption
$\Gamma \vdash P \triangleright \Delta_1$ and $\Gamma \vdash \mathbf{0} \triangleright \Delta_2$ where $\Delta = \Delta_1 \circ \Delta_2, \Delta_1 \asymp \Delta_2$ By inversion
Δ_2 is only end and for Δ_2 such that $\mathrm{dom}\,(\Delta_1) \cap \mathrm{dom}\,(\Delta_2) = \emptyset$ By inversion
then $\Gamma \vdash P \triangleright \Delta_1, \Delta_2$ By weakening
and $\Delta = \Delta_1 \circ \Delta_2 = \Delta_1, \Delta_2$

$\Gamma \vdash P \triangleright \Delta$ By assumption
$\Gamma \vdash \mathbf{0} \triangleright \Delta'$ where Δ' is only end and $\mathrm{dom}\,(\Delta) \cap \mathrm{dom}\,(\Delta') = \emptyset$ By rule [Inact]
$\Gamma \vdash P \mid \mathbf{0} \triangleright \Delta, \Delta'$ By rule [Conc]
for $\Delta' = \emptyset$ we have that
$\Gamma \vdash P \mid \mathbf{0} \triangleright \Delta$

Case: $P \mid Q \equiv Q \mid P$

$\Gamma \vdash P \mid Q \triangleright \Delta$ By assumption
$\Gamma \vdash P \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta_2$
where $\Delta = \Delta_1 \circ \Delta_2$ and $\Delta_1 \asymp \Delta_2 = \Delta_2 \asymp \Delta_1$ By inversion
$\Gamma \vdash Q \mid P \triangleright \Delta$ By rule [Conc]

The other case is symmetric to the above one.

Case: $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

$\Gamma \vdash (P \mid Q) \mid R \triangleright \Delta$ By assumption
$\Gamma \vdash P \triangleright \Delta_1$, $\Gamma \vdash Q \triangleright \Delta_2$ and $\Gamma \vdash R \triangleright \Delta_3$ where $\Delta = \Delta_1 \circ \Delta_2 \circ \Delta 3$
and $\Delta_1 \asymp \Delta_2 \asymp \Delta_3$ By inversion
$\Gamma \vdash P \mid (Q \mid R) \triangleright \Delta$ By rule [Conc]

The other case is symmetric to the above one.

The other axioms are proved in the similar way as in Vasconcelos and Yoshida [19]. □

Theorem A.12 (subject reduction) *$\Gamma \vdash P \triangleright \Delta$ with Δ coherent and $P \to P'$ imply $\Gamma \vdash P' \triangleright \Delta'$ where $\Delta = \Delta'$ or $\Delta \to \Delta'$ with Δ' coherent.*

Proof. By rule induction on the derivation of $P \to P'$. There is a case for each operational semantics rule. For each operational semantics rule, we consider each type system rule that can generate $\Gamma \vdash P \triangleright \Delta$. By Lemma A.10(2) we have that Δ' is coherent as well.

Case: [Link]

$$\overline{a}[2..\mathtt{n}]\,(\tilde{x}).P_1 \mid a[2]\,(\tilde{x}).P_2 \mid \cdots \mid a[\mathtt{n}]\,(\tilde{x}).P_n \;\;\to\;\; (\nu\tilde{\kappa})(P_1[\tilde{\kappa}_1/\tilde{x}] \mid P_2[\tilde{\kappa}_2/\tilde{x}] \mid \ldots \mid P_{\mathtt{n}}[\tilde{\kappa}_n/\tilde{x}])$$

$\Gamma \vdash a[2..\mathtt{n}]\,(\tilde{x}).P_1 \mid a[2]\,(\tilde{x}).P_2 \mid \cdots \mid a[\mathtt{n}]\,(\tilde{x}).P_n \triangleright \Delta$ By assumption
$\Gamma \vdash a[2..\mathtt{n}]\,(\tilde{x}).P_1 \triangleright \Delta_1 \;\ldots\; \Gamma \vdash a[\mathtt{n}]\,(\tilde{x}).P_n \triangleright \Delta_n$ where $\Delta = \Delta_1 \circ \ldots \circ \Delta_n$
and $\Delta_1 \asymp \ldots \asymp \Delta_n$ By inversion on [Conc]

$a\colon\langle G\rangle, \Gamma \vdash P_1 \triangleright \Delta_1, \tilde{x}\colon(G\upharpoonright 1)@1 \quad |\tilde{x}| = \max(\mathsf{sid}(G))$ By inversion on [Mcast]
$a\colon\langle G\rangle, \Gamma \vdash P_2 \triangleright \Delta_2, \tilde{x}\colon(G\upharpoonright 2)@2, |\tilde{x}| = \max(\mathsf{sid}(G))$ By inversion on [Macc]
...
$a\colon\langle G\rangle, \Gamma \vdash P_n \triangleright \Delta_n, \tilde{x}\colon(G\upharpoonright \mathtt{n})@\mathtt{n}, |\tilde{x}| = \max(\mathsf{sid}(G))$ By inversion on [Macc]

$\tilde{\kappa}_1 \notin \mathrm{dom}\,(\Delta_1), \ldots, \tilde{\kappa}_n \notin \mathrm{dom}\,(\Delta_n)$ for every $i \in \{1, \ldots, n\}.\tilde{\kappa}_i$ are newly generated

$\Gamma \vdash P_1[\tilde{\kappa}_1/\tilde{x}] \triangleright \Delta_1, \tilde{\kappa}\colon (G\upharpoonright 1)@1$ By Lemma A.4
...
$\Gamma \vdash P_n[\tilde{\kappa}_n/\tilde{x}] \triangleright \Delta_n, \tilde{\kappa}\colon (G\upharpoonright \mathtt{n})@\mathtt{n}$ By Lemma A.4

$\tilde{\kappa}_1\colon (G\upharpoonright 1)@1 \asymp \ldots \asymp \tilde{\kappa}_n\colon (G\upharpoonright \mathtt{n})@\mathtt{n}$ By Definition 4.4
$\Gamma \vdash P_1[\tilde{\kappa}_1/\tilde{x}] \mid P_2[\tilde{\kappa}_2/\tilde{x}] \mid \ldots \mid P_{\mathtt{n}}[\tilde{\kappa}_n/\tilde{x}] \triangleright$
$\Delta_1, \tilde{\kappa}_1\colon (G\upharpoonright 1)@1 \circ \ldots. \circ \Delta_n, \tilde{\kappa}_n\colon (G\upharpoonright \mathtt{n})@\mathtt{n}$ By rule [Conc]

$\Delta_1, \tilde{\kappa}_1\colon (G\upharpoonright 1)@1 \circ \ldots. \circ \Delta_n, \tilde{\kappa}_n\colon (G\upharpoonright \mathtt{n})@\mathtt{n} =$
$\Delta_1 \circ \ldots \circ \Delta_n, \tilde{\kappa}_1\colon (G\upharpoonright 1)@1 \circ \ldots. \circ \tilde{\kappa}_n\colon (G\upharpoonright \mathtt{n})@\mathtt{n}$
$\Gamma \vdash (\nu\tilde{\kappa})(P_1[\tilde{\kappa}_1/\tilde{x}] \mid P_2[\tilde{\kappa}_2/\tilde{x}] \mid \ldots \mid P_{\mathtt{n}}[\tilde{\kappa}_n/\tilde{x}]) \triangleright \Delta$ By rule [CRes]

Case: [Multicasting]

$$\kappa[m_1,...,m_n]_{\mathtt{p}}!\langle\tilde{e}\rangle; P_p \mid \kappa_{m_1\mathtt{p}_1}?(\tilde{y}); P_{p_1} \mid \cdots \mid \kappa_{m_n\mathtt{p}_n}?(\tilde{y}); P_{p_n} \;\;\rightarrow\;\; P \mid P_{p_1}[\tilde{v}/\tilde{y}] \mid \cdots \mid P_{p_n}[\tilde{v}/\tilde{y}]$$

$$(\mathtt{p} \neq \mathtt{p}_1 \neq \cdots \neq \mathtt{p}_n, \tilde{e} \downarrow \tilde{v})$$

$\Gamma \vdash \kappa[m_1,...,m_n]_{\mathtt{p}}!\langle\tilde{e}\rangle; P_p \mid \kappa_{m_1\mathtt{p}_1}?(\tilde{y}); P_{p_1} \mid \cdots \mid \kappa_{m_n\mathtt{p}_n}?(\tilde{y}); P_{p_n} \triangleright \Delta$ By assumption
$\Gamma \vdash \kappa[m_1,...,m_n]_{\mathtt{p}}!\langle\tilde{e}\rangle; P_p \triangleright \Delta_1$
$\Gamma \vdash \kappa_{m_1\mathtt{p}_1}?(\tilde{y}); P_{p_1} \triangleright \Delta_2$
...
$\Gamma \vdash \kappa_{m_n\mathtt{p}_n}?(\tilde{y}); P_{p_n} \triangleright \Delta_{n+1}$
where $\Delta = \Delta_1 \circ ... \circ \Delta_{n+1}$ and $\Delta_1 \asymp ... \asymp \Delta_{n+1}$ By inversion on [Conc]

$\Delta_1 = \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : m_1,...,m_n!\langle\tilde{S}\rangle; T@\mathtt{p}$ By rule [Send]
$\Delta_2 = \Delta'_2, \tilde{\kappa}_{\mathtt{p}_1} : m_1?\langle\tilde{S}\rangle; T_1@\mathtt{p}_1$ By rule [Rcv]
...
$\Delta_{n+1} = \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}_n} : m_n?\langle\tilde{S}\rangle; T_n@\mathtt{p}_n$ By rule [Rcv]

$\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \Gamma \vdash P_p \triangleright \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p}$ By inversion on [Send]
$\Gamma, \tilde{y} : \tilde{S} \vdash P_{p_1} \triangleright \Delta'_2, \tilde{\kappa}_{\mathtt{p}_1} : T_1@\mathtt{p}_1$ By inversion on [Rcv]
...
$\Gamma, \tilde{y} : \tilde{S} \vdash P_{p_n} \triangleright \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}_n} : T_n@\mathtt{p}_n$ By inversion on [Rcv]

$\Gamma \vdash P_{p_1}[\tilde{v}/\tilde{y}] \triangleright \Delta'_2, \tilde{\kappa}_{\mathtt{p}_1} : T_1@\mathtt{p}_1$ By Lemma A.3.1
...
$\Gamma \vdash P_{p_n}[\tilde{v}/\tilde{y}] \triangleright \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}_n} : T_n@\mathtt{p}_n$ By Lemma A.3.1

$\Gamma \vdash P_p \mid P_{p_1}[\tilde{v}/\tilde{y}] \mid \cdots \mid P_{p_n}[\tilde{v}/\tilde{y}]$
$\triangleright \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ \Delta'_2, \tilde{\kappa}_{\mathtt{p}_1} : T_1@\mathtt{p}_1 \circ ... \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}_n} : T_n@\mathtt{p}_n$ By rule [Conc]

$\Delta'_1, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ \Delta'_2, \tilde{\kappa}_{\mathtt{p}_1} : T_1@\mathtt{p}_1 \circ ... \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}_n} : T_n@\mathtt{p}_n =$
$\Delta'_1 \circ ... \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ ... \circ \tilde{\kappa}_t : T_n@\mathtt{p}_n$
$\Delta'_1 \circ ... \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}} : m_1,...,m_n!\langle\tilde{S}\rangle.T@\mathtt{p} \circ \tilde{\kappa}_{\mathtt{p}_1} : m_1?\langle\tilde{S}\rangle.T_1@\mathtt{p}_1 \circ ... \circ \tilde{\kappa}_t : m_n?\langle\tilde{S}\rangle.T_n@\mathtt{p}_n$
$\rightarrow \Delta'_1 \circ ... \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ ... \circ \tilde{\kappa}_{\mathtt{p}_n} : T_n@\mathtt{p}_n$ By [TR-Context, TR-Mult]

Case: [Multicasting]

$$\kappa_{m\mathtt{p}}!\langle\tilde{t}\rangle; P_p \mid \kappa_{m\mathtt{p}'}?(\tilde{y}); P_{p'} \;\;\rightarrow\;\; P_p \mid P_{p'}[\tilde{t}/\tilde{y}]$$

$\Gamma \vdash \kappa_{m\mathtt{p}}!\langle\tilde{t}\rangle; P_p \mid \kappa_{m\mathtt{p}'}?(\tilde{y}); P_{p'} \triangleright \Delta$ By assumption
$\Gamma \vdash \kappa_{m\mathtt{p}}!\langle\tilde{t}\rangle; P_p \triangleright \Delta_1 \;\; \Gamma \vdash \kappa_{m\mathtt{p}'}?(\tilde{y}); P_{p'} \triangleright \Delta_2$
where $\Delta = \Delta_1 \circ \Delta_2$ and $\Delta_1 \asymp .\Delta_2$ By inversion on [Conc]

$\Delta_1 = \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : m!\langle T''@\mathtt{p}''\rangle; T@\mathtt{p}, \tilde{t} : T''@\mathtt{p}''$ By rule [Thr]
$\Delta_2 = \Delta'_2, \tilde{\kappa}_{\mathtt{p}'} : m?\langle T''@\mathtt{p}''\rangle.T'@\mathtt{p}'$ By rule [Cat]

$\Gamma \vdash P_p \triangleright \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p}$ By inversion on [Send]
$\Gamma \vdash P_{p'} \triangleright \Delta'_2, \tilde{\kappa}_{\mathtt{p}'} : T'@\mathtt{p}', \tilde{y} : T''@\mathtt{p}''$ By inversion on [Cat]
$\Gamma \vdash P_{p'}[\tilde{t}/\tilde{y}] \triangleright \Delta'_2, \tilde{\kappa}_{\mathtt{p}'} : T_1@\mathtt{p}', \tilde{t} : T''@\mathtt{p}''$ By Lemma A.4

$\Gamma \vdash P_p \mid P_{p'}[\tilde{t}/\tilde{y}] \mid \triangleright \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ \Delta'_2, \tilde{\kappa}_{\mathtt{p}'} : T'@\mathtt{p}', \tilde{t} : T''@\mathtt{p}''$ By rule [Conc]

$\Delta'_1, \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ \Delta'_2, \tilde{\kappa} : T'@\mathtt{p}', \tilde{t} : T''@\mathtt{p}'' = \Delta'_1 \circ \Delta'_2, \tilde{t} : T''@\mathtt{p}'', \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ \tilde{\kappa}_{\mathtt{p}'} : T'@\mathtt{p}'$

$\Delta'_1 \circ \Delta'_2, \tilde{t} : \tilde{T''}@\tilde{\mathtt{p}''}, \tilde{\kappa}_{\mathtt{p}} : m!\langle T''@\mathtt{p}''\rangle; T@\mathtt{p} \circ \tilde{\kappa}_{\mathtt{p}'} : m?\langle T''@\mathtt{p}''\rangle.T'@\mathtt{p}' \rightarrow$
$\Delta'_1 \circ \Delta'_2, \tilde{t} : T''@\mathtt{p}'', \tilde{\kappa}_{\mathtt{p}} : T@\mathtt{p} \circ \tilde{\kappa}_{\mathtt{p}'} : T'@\mathtt{p}'$ By [TR-Context, TR-MultD]

Case: [MultiLabel]

$$\kappa[m_1,...,m_n]_{\mathtt{p}} \lhd l_i; P \mid \kappa_{m_1\mathtt{p}_1} \rhd \{l_j : P_{1j}\}_{j\in I} \mid \cdots \mid \kappa_{m_n\mathtt{p}_n} \rhd \{l_j : P_{nj}\}_{j\in I} \;\;\rightarrow\;\; P \mid P_{1i} \mid \cdots \mid P_{ni}$$

$$(\mathtt{p} \neq \mathtt{p}_1 \neq \cdots \neq \mathtt{p}_n, i \in I)$$

$\Gamma \vdash \kappa[m_1,...,m_n]_{\mathtt{p}} \lhd l_i; P \mid \kappa_{m_1\mathtt{p}_1} \rhd \{l_j : P_{1j}\}_{j\in I} \mid \cdots \mid \kappa_{m_n\mathtt{p}_n} \rhd \{l_j : P_{nj}\}_{j\in I}$
$\triangleright \Delta$ By assumption
$\Gamma \vdash \kappa[m_1,...,m_n]_{\mathtt{p}} \lhd l_i; P \triangleright \Delta_1$
$\Gamma \vdash \kappa_{m_1\mathtt{p}_1} \rhd \{l_j : P_{1j}\}_{j\in I} \triangleright \Delta_2$
...

$\Gamma \vdash \kappa_{m_n \mathtt{p_n}} \rhd \{l_j : P_{nj}\}_{j\in I} \triangleright \Delta_{n+1}$ where $\Delta = \Delta_1 \circ \ldots \circ \Delta_{n+1}$
and $\Delta_1 \asymp \ldots \asymp \Delta_{n+1}$ — By inversion on [Conc]

$\Delta_1 = \Delta'_1, \tilde{\kappa}_{\mathtt{p}} : m_1, \ldots, m_n \oplus \{l_j : T_j\}_{j\in J}@\mathtt{p}$ — By rule [Sel]
$\Delta_2 = \Delta'_2, \tilde{\kappa}_{\mathtt{p_1}} : m_1\&\{l_j : T_{1j}\}_{j\in J}@\mathtt{p_1}$ — By rule [Br]
...
$\Delta_{n+1} = \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p_n}} : m_n\&\{l_j : T_{nj}\}_{j\in J}@\mathtt{p_n}$ — By rule [Br]

$\Gamma \vdash P \triangleright \Delta'_1, \tilde{\kappa}_{\mathtt{p_1}} : T_i@\mathtt{p}$ and $i \in J$ — By inversion on [Sel]
$\forall j \in J, \Gamma \vdash P_{1_j} \triangleright \Delta'_2, \tilde{\kappa}_{\mathtt{p}} : T_{1_j}@\mathtt{p_1}$ — By inversion on [Br]
...
$\forall j \in J, \Gamma \vdash P_{n_j} \triangleright \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p_n}} : T_{n_j}@\mathtt{p_n}$ — By inversion on [Br]

$\Gamma \vdash P \mid P_{1i} \mid \cdots \mid P_{ni} \triangleright \Delta'_1 \circ \ldots \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p_1}} : T_i@\mathtt{p} \circ \ldots \circ \tilde{\kappa}_{\mathtt{p_n}} : T_{n_i}@\mathtt{p_n}$ — By rule [Conc]

$\Delta'_1, \tilde{\kappa}_{\mathtt{p_1}} : m_1, \ldots, m_n \oplus \{l_j : T_j\}_{j\in J}@\mathtt{p}, \Delta'_2, \tilde{\kappa}_{\mathtt{p}} : m_1\&\{l_j : T_{1j}\}_{j\in J}@\mathtt{p_1}, \ldots,$
$\Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p_n}} : m_n\&\{l_j : T_{nj}\}_{j\in J}@\mathtt{p_n} \rightarrow$
$\Delta'_1 \circ \ldots \circ \Delta'_{n+1}, \tilde{\kappa}_{\mathtt{p_1}} : T_i@\mathtt{p} \circ \ldots \circ \tilde{\kappa}_{\mathtt{p_n}} : T_{n_i}@\mathtt{p_n}$ — By [TR-Context, TR-MultL]

Case: [If1] and [If2] are trivial by induction.

Case: [Def]

$$\mathtt{def}\ D\ \mathtt{in}\ (X\langle\tilde{e}\rangle \mid Q) \quad\rightarrow\quad \mathtt{def}\ D\ \mathtt{in}\ (P[\tilde{v}/\tilde{y}] \mid Q) \qquad (\tilde{e} \downarrow \tilde{v}, X(\tilde{y}) = P \in D)$$

$\Gamma \vdash \mathtt{def}\ D\ \mathtt{in}\ (X\langle\tilde{e}\rangle \mid Q) \triangleright \Delta$ — By assumption
$\Gamma, X : \tilde{S}\tilde{T}, \tilde{y_1} : \tilde{S} \vdash P \triangleright \tilde{y_2} : \tilde{T}@\tilde{\mathtt{p}},\ \Gamma, X : \tilde{S}\tilde{T} \vdash X\langle\tilde{e}\rangle \mid Q \triangleright \Delta$
where $\tilde{y_1} \in \tilde{y}$ and $\tilde{y_2} \in \tilde{y}$ — By inversion on rule [Def]
$\Gamma, X : \tilde{S}\tilde{T} \vdash X\langle\tilde{e}\rangle \triangleright \Delta_1$ and $\Gamma, X : \tilde{S}\tilde{T} \vdash Q \triangleright \Delta_2$
where $\Delta = \Delta_1 \circ \Delta_2$ and $\Delta_1 \asymp \Delta_2$ — By inversion on rule [Conc]
$\Gamma \vdash \tilde{e_1} \triangleright \tilde{S}$ and $\Delta_1 = \Delta'_1, \tilde{\kappa} : \tilde{T}@\tilde{\mathtt{p}}$ where $\tilde{\kappa}, \tilde{e_1} \in \tilde{e}$ — By inversion on rule [Var]
$\Gamma, X : \tilde{S}\tilde{T} \vdash P[\tilde{v}/\tilde{y}] \triangleright \tilde{\kappa} : \tilde{T}@\tilde{\mathtt{p}}$ — By Lemma A.3.1 and Lemma A.4
$\Gamma, X : \tilde{S}\tilde{T} \vdash P[\tilde{v}/\tilde{y}] \triangleright \Delta'_1, \tilde{\kappa} : \tilde{T}@\tilde{\mathtt{p}}$ — By Lemma A.3.2
$\Gamma, X : \tilde{S}\tilde{T} \vdash P[\tilde{v}/\tilde{y}] \mid Q \triangleright \Delta'_1, \tilde{\kappa} : \tilde{T}@\tilde{\mathtt{p}} \circ \Delta_2 = \Delta$ — By rule [Conc]
$\Gamma \vdash \mathtt{def}\ D\ \mathtt{in}\ (P[\tilde{v}/\tilde{y}] \mid Q) \triangleright \Delta$ — By rule [Def]

Case: [Par]

$$P \ \rightarrow\ P' \ \Rightarrow\ P \mid Q \ \rightarrow\ P' \mid Q$$

$\Gamma \vdash P \mid Q \triangleright \Delta$ — By assumption
$\Gamma \vdash P \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta_2$ where $\Delta = \Delta_1 \circ \Delta_2$ and $\Delta_1 \asymp \Delta_2$ — By rule [Conc]
$\Gamma \vdash P' \triangleright \Delta'_1$ where $\Delta_1 = \Delta'_1$ or $\Delta_1 \rightarrow \Delta'_1$ — By induction
when $\Delta_1 = \Delta'_1$ then the proof is trivial so we investigate the second case
when $\Delta_1 \rightarrow \Delta'_1$
$\Delta_2 \asymp \Delta'_1$ and $\Delta_1 \circ \Delta_2 \rightarrow \Delta'_1 \circ \Delta_2$ — By Lemma A.10 (1)
$\Gamma \vdash P' \mid Q \triangleright \Delta'_1 \circ \Delta_2$ — By rule [Conc]

Case: [Defin] is trivial by induction.
Case: [Str]

$$P \equiv P' \text{ and } P' \ \rightarrow\ Q' \text{ and } Q' \equiv Q \ \Rightarrow\ P \ \rightarrow\ Q$$

$\Gamma \vdash P \triangleright \Delta,\ P \equiv P',\ P' \ \rightarrow\ Q'$ and $Q', Q' \equiv Q$ — By assumption
$\Gamma \vdash P' \triangleright \Delta,\ P' \ \rightarrow\ Q'$ and $Q', Q' \equiv Q$ — By Theorem A.11
$\Gamma \vdash Q' \triangleright \Delta,\ Q' \equiv Q$ — By induction
$\Gamma \vdash Q \triangleright \Delta$ — By Theorem A.11

□